GAIA: An OS Page Cache for Heterogeneous Systems

Tanya Brokhman, Pavel Lifshits, and Mark Silberstein, *Technion—Israel Institute of Technology*

https://www.usenix.org/conference/atc19/presentation/brokhman

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

GAIA: An OS Page Cache for Heterogeneous Systems

Tanya Brokhman, Pavel Lifshits and Mark Silberstein
Technion – Israel Institute of Technology

Abstract

We propose a principled approach to integrating GPU memory with an OS page cache. We design *GAIA*, a weakly-consistent page cache that spans CPU and GPU memories. GAIA enables the standard `mmap` system call to map files into the GPU address space, thereby enabling data-dependent GPU accesses to large files and efficient write-sharing between the CPU and GPUs. Under the hood, GAIA (1) integrates lazy release consistency protocol into the OS page cache while maintaining backward compatibility with CPU processes and *unmodified* GPU kernels; (2) improves CPU I/O performance by using data cached in GPU memory, and (3) optimizes the readahead prefetcher to support accesses to files cached in GPUs.

We prototype GAIA in Linux and evaluate it on NVIDIA Pascal GPUs. We show up to 3× speedup in *CPU* file I/O and up to 8× in *unmodified* realistic workloads such as Gunrock GPU-accelerated graph processing, image collage, and microscopy image stitching.

Introduction

GPUs have come a long way from fixed-function accelerators to fully-programmable, high-performance processors. Yet their integration with the host Operating System (OS) is still quite limited. In particular, GPU physical memory, which today may be as large as 32GB [9], has been traditionally managed entirely by the GPU driver, without the host OS control. One crucial implication of this design is that the OS cannot provide core system services to GPU kernels, such as efficient access to memory mapped files, nor can it optimize I/O performance for CPU applications sharing files with GPUs. To mitigate these limitations, tighter *integration of GPU memory into the OS page cache and file I/O mechanisms* is required. Achieving such integration is one of the goals of this paper.

Prior works demonstrate that mapping files into GPU memory provides a number of benefits [34, 36, 35]. Files can be accessed from the GPU using an intuitive pointer-based programming model, enabling GPU applications with data-dependent access patterns. Transparent system-level performance optimizations such as prefetching and double buffering can be implemented to achieve high performance for I/O intensive GPU kernels. Finally, file contents can be easily shared between legacy CPU and GPU-accelerated processes.

Extending the OS page cache into GPU memory is advantageous even for CPU I/O performance. With modern servers commonly hosting 8 and more GPUs, the total GPU memory available (100-200GB) is large enough to be used for caching file contents. As we show empirically, doing so may boost the I/O performance by up to 3× compared to accesses to a high-performance SSD (§6). Finally, the OS management of the page cache in GPU memory may allow caching GPU file accesses directly in the GPU page cache, bypassing CPU-side page cache and avoiding its pollution [15].

Unfortunately, today's commodity systems fall short of providing full integration of GPU memory with the OS page cache. ActivePointers [34] enable a memory-mapped files abstraction for GPUs, but their use of special pointers requires intrusive modifications to GPU kernels, making them incompatible with closed-source libraries such as cuBLAS [7]. NVIDIA's Unified Virtual Memory (UVM) [8] and the Heterogeneous Memory Management (HMM) [4] module in Linux allow GPUs and CPUs to access shared virtual memory space. However, neither UVM nor HMM allow mapping files into GPU memory, which makes them inefficient when processing large files (§6.3.2). More fundamentally, both UVM and HMM force the physical page to be present in the memory of only one processor. This results in a performance penalty in case of false sharing in data-parallel write-sharing workloads. Moreover, false sharing has a significant impact on the system as a whole, as we show in (§3).

Several hardware architectures introduce cache coherence between CPUs and GPUs. In particular, CPUs with integrated GPUs support coherent shared *virtual* memory in hardware. In contrast to discrete GPUs, however, integrated GPUs lack large separate physical memory. Therefore, today's OSes do not provide any memory management services for them.

Recent high-end IBM Power-9 systems feature hardware cache-coherent shared virtual memory between CPUs and discrete GPUs [31]. GPU memory is managed as another NUMA node. Thus, the OS is able to provide memory management services to GPUs, including memory-mapped files. Unfortunately, cache coherence between the CPU and discrete GPUs is not available in x86-based commodity systems, and it is unclear when it will be introduced (see §4.3). Clearly, using the NUMA mechanisms for non-coherent GPU memory management would not work. For example, migrating a CPU-accessible page into GPU memory will break the expected memory behavior for CPU processes, e.g., due to the lack of atomic operations across the PCIe bus, among other issues.

To resolve these limitations, we propose **GAIA**[1], a distributed, weakly-consistent page cache architecture for heterogeneous multi-GPU systems that extends the OS page cache into GPU memories and integrates with the OS file I/O layer. With GAIA, CPU programs use regular file I/O to share files with GPUs. Calling `mmap` with a new `MMAP_ONGPU` flag makes the mapping accessible to the GPU kernels, thus providing support for GPU accesses to shared files. This approach allows access to memory-mapped files from *unmodified* GPU kernels.

This paper makes the following contributions:

- We characterize the overheads of CPU-GPU false sharing in existing systems (§3.1). We propose a unified page cache which eliminates false sharing by using a lazy release consistency model [22, 10].
- We extend the OS page cache to control the *unified* (CPU and GPU) page cache and its consistency (§4.1), without requiring CPU-GPU hardware cache coherence. We introduce a *peer-caching* mechanism and integrate it with the OS readahead prefetcher, enabling any processor accessing files to retrieve them from the best location, and in particular, from GPU memory (§6.2).
- We present a fully functional generic implementation in Linux, *not tailored* to any particular GPU.
- We prototype GAIA on NVIDIA Pascal GPU, leveraging its page fault support. We modify public parts of the GPU driver and reliably emulate the functionality which cannot be implemented due to the closed-source driver.
- We evaluate GAIA using real workloads, including (1) an *unmodified* graph processing framework - Gunrock [38], (2) a Mosaic application that creates an image collage from a large image database [34], and (3) a multi-GPU image stitching application [16, 18] that determines the optimal way to combine multiple image tiles, demonstrating the advantages and the ease-of-use of GAIA for real-life scenarios.

Background

We briefly explain the main principles of several existing memory consistency models relevant to our work.

Release consistency. Release consistency (RC) [22] is a form of relaxed memory consistency that permits delaying the effects of writes to distributed shared memory. The programmer controls the visibility of the writes from each processor by means of the *acquire* and *release* synchronization operations. Informally, the writes are guaranteed to be visible to the readers of a shared memory region after the writer *release*-s the region and the reader *acquire*-s it.

RC permits concurrent updates to different versions of the page in multiple processors, which get *merged* upon later accesses. A common way to resolve merge conflicts is by using the version vectors mechanism, explained below.

Lazy release consistency. In Lazy Release Consistency (LRC) [22, 10] the propagation of updates to a page is delayed until *acquire*. At synchronization time, the acquiring processor receives the updates from the other processors. Usually, the underlying implementation leverages page faults to trigger the updates [22]. Specifically, a stale local copy of the page is marked inaccessible, causing the processor to fault on the first access. The faulting processor then retrieves the up-to-date copy of the page from one or more processors. In the home-based version of the protocol, a home node is assigned to a page to maintain the most up-to-date version of the page. The requesting processor contacts the home node to retrieve the latest version of the page.

Version vectors. Version vectors (VVs) [32] are used in distributed systems to keep track of replica versions of an object. The description below is transcribed from Parker et al. [32].

A version vector of an object O is a sequence of n pairs, where n is the number of sites at which O is stored. The pair $\{S_i : v_i\}$ is the latest version of O made at site S_i. That is, the vector entry v_i counts the number of updates to O made at site S_i. Each time O is copied from site S_i to S_j at which it was not present, the version vector of site S_i is adopted by site S_j. If the sites have a conflicting version of the replica, the new vector is created by taking the largest version among the two for each entry in the vector.

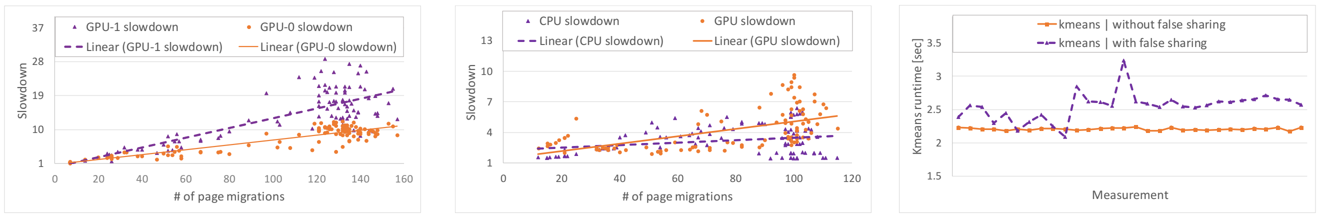
Consistency model considerations

The choice of the consistency model for the unified page cache is an important design question. We describe the options we considered and justify our choice of LRC.

POSIX: strong consistency. In POSIX writes are immediately visible to all the processes using the same file [5]. In x86 systems without coherent shared memory and with GPUs connected via a high latency (relative to local memory) PCIe bus, such a strong consistency model in a unified page cache would be inefficient [36].

GPUfs: session semantics. GPUfs [36] introduces a GPU-side library for file I/O from GPU kernels. GPUfs implements a distributed page cache with session semantics. Session semantics, however, couple between the file open/close operations and data synchronization. As a result, they cannot be used with `mmap`, as sometimes the file contents need to be synchronized across processors without having to unmap and close the file and then reopen and map it again. Therefore, we find session semantics unsuitable for GAIA.

[1]**G**lob**A**l un**I**fied p**A**ge cache, or simply the daughter's name of one of the authors, born during this project.



(a) False sharing between two GPUs

(b) False sharing between CPU and GPU

(c) The effect of false sharing in GPUs on an isolated CPU-only kmeans benchmark [33]

Figure 1: Impact of false sharing on the performance of GPU kernels and the system as a whole.

UVM: page-level strict coherence. NVIDIA UVM [8] and Linux (HMM) [4] implement strict coherence [24] at the GPU page granularity (e.g., 64KB in NVIDIA GPUs). In this model, a page can be mapped only by one processor at a time. Thus, two processors cannot observe different replicas of the page (multiple read-only replicas are allowed). If a processor accesses a non-resident page, the page fault causes the page to migrate, i.e., the data is transferred, and the page is remapped at the requestor and unmapped at the source.

Although this model might seem appealing for page cache management, it suffers from sporadic performance degradation due to *false sharing*. False sharing of a page occurs when two processors inadvertently share the same page, at least one of them for write, while performing non-overlapping data accesses [17]. False sharing is known to dramatically degrade the performance of distributed shared memory systems with strict coherence because it causes repetitive and costly page migration among different physical locations [17]. False sharing has been also reported in multi-GPU applications that use NVIDIA's UVM [8]. The official recommended solution is to allocate private replicas of the shared buffer in each processor and manually merge them after use.

False sharing in a page cache. If strict coherence is used for managing a unified page cache, false sharing of the page cache pages might occur quite often. Consider an image processing task that stitches multiple image tiles into a large output image stored in a file, e.g., when processing samples from a microscope [16]. False sharing will likely occur when multiple GPUs process the images in a data-parallel way, each writing its results to a shared output file. Consider two GPUs, one processing the left and another the right half of the image. In this case, false sharing might occur at every *row* of the output. This is because for large images (thousands of pixels in each dimension) stored in row-major format, each row will occupy at least one memory page in the page cache. Since each half of the row is processed on a different GPU, the same page will be updated by both GPUs. We observe this effect in real applications (§ 6.3.3).

False sharing with UVM

Impact of false sharing on application performance. To experimentally quantify the cost of false sharing in multi-GPU systems, we allocate a 64KB-buffer (one GPU page) and divide it between two NVIDIA GTX1080 GPUs. Each GPU executes read-modify-write operations (so they are not optimized out) on its half in a loop. We run a total of 64 threadblocks per GPU, each accessing its own part of the array, all active during the run. To control the degree of contention, we vary the number of loop iterations per GPU.

We compare the execution time when both GPUs use a shared UVM buffer (with false sharing) with the case when both use private buffers and merge them at the end of the run (no false sharing). Figure 1a shows the scatter graph of the measurements. False sharing causes slowdown that grows with the number of page migrations, reaching 28×, and results in large runtime variance [2]. Figure 1b shows similar results when one of the GPUs is replaced with a single CPU thread. This also indicates that adding more GPUs is likely to cause even larger degradation due to higher contention and increased data transfer costs.

System impact of false sharing. False sharing among GPUs affects the performance of the system as a whole. We run the CPU-only kmeans benchmark from Phoenix [33] *in parallel* with the multi-GPU false sharing benchmark above. We allocate two CPU cores for GPU management, and the remaining four cores to running kmeans (modified to spawn four threads). The GPU activity *should not* interfere with kmeans because kmeans does not use the GPU.

However, *we observe significant interference when GPUs experience false sharing*. Figure 1c depicts the runtime of kmeans when run together with the multi-GPU run, with and without false sharing. Not only does kmeans become up to 47% slower, but the execution times vary substantially. Thus, false sharing affects an *unrelated* CPU application, breaking the fundamental OS performance isolation properties.

Preventing page bouncing via pinning. In theory, the false sharing overheads could be reduced by pinning the page in

[2]The difference in the slowdown between the two GPUs stems from the imperfect synchronization between them. Thus, the one invoked first (GPU0) can run briefly without contention.

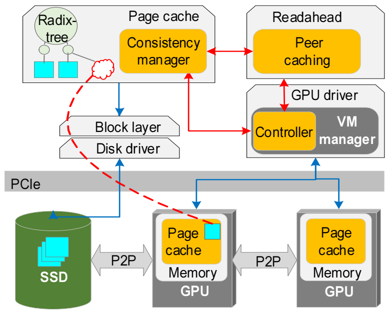


Figure 2: GAIA high-level design in the OS kernel. The modified parts are highlighted.

CPU legacy: producer

```
fd=open("file"); write(fd,...); close(fd);
```

GPU controller: consumer

```
fd=open("file");
ptr=mmap(..,size,
      MAP_ONGPU|MAP_SHARED,fd);
macquire(ptr,size,GPU0);
<<<>>>gpu_kernel(ptr);
wait_for_gpu_completion();
mrelease(ptr,size,GPU0);
```

GPU 0: consumer

```
gpu_kernel(
    char* ptr)
{
    ptr[thrd]++;
}
```

Figure 3: Code sketch of `mmap` for GPU. The CPU writes data into the file and then invokes the GPU controller, which maps the file and runs the GPU kernel.

memory of one of the processors, and mapping it into the address space of the other processors for remote access over PCIe. Unfortunately, pinning page cache pages is quite problematic. It would require substantial modifications to the existing page cache management mechanisms. For example, to be evicted, the pinned page would need to be unmapped from the virtual address space of all the mapping processors.

Moreover, even though pinning is likely to yield better system performance for pages experiencing false sharing, in the common case remote accesses from other processors would be slower than accessing the pages locally. Thus, robust false sharing detection heuristics should be designed, such that only the actual page bouncing triggers the pinning mechanism. On the other hand, enabling the programmer to pin pages manually at the `mmap` time is not efficient either, because then the pages must be initialized with the contents of the file. Mapping large files would thus require reading them in full from the disk, which not only nullifies the on-demand file loading benefits of `mmap`, but might not even be possible for the large files exceeding physical memory.

Implications for Unified Page Cache design. We conclude that *the UVM strict coherence model is unsuitable for implementing a unified page cache*. It may suffer from spurious and hard-to-debug performance degradation that affects the whole system, and only worsens as the number of GPUs increases. A system-level service with such inherent limitations would be a poor design choice. Thus, we chose to build a unified cache that follows the lazy-release consistency model and sidesteps the false sharing issues entirely.

Design

Overview. Figure 2 shows the main GAIA components in the OS kernel. A distributed page cache spans across the CPU and GPU memories. The OS page cache is extended to include a *consistency manager* that implements home-based lazy release consistency (LRC). It keeps track of the versions of all the file-backed memory pages and their locations. When a page is requested by the GPU or the CPU (due to a page fault), the consistency manager determines the locations of the most recent versions, and retrieves and merges them if necessary. We introduce new `macquire` and `mrelease` system calls which follow standard Release Consistency semantics and have to be used when accessing shared files. We explain the page cache design in (§4.1).

If an up-to-date page replica is available in multiple locations, the *peer-caching* mechanism retrieves the page via the most efficient path, e.g., from GPU memory for the CPU I/O request, or directly from storage for the GPU access as in SPIN [15]. This mechanism is integrated with the OS readahead prefetcher to achieve high performance (§6.2). To enable proper handling of memory-mapped files on GPUs, the GAIA controller in the GPU driver keeps track of all the GPU virtual ranges in the system that are backed by files.

File-sharing example. Figure 3 shows a code sketch of sharing a file between a legacy CPU application (producer) and a GPU-accelerated one (consumer). This example illustrates two important aspects of the design. First, *no GPU kernel changes* are necessary to access files, and no new system code runs on the GPU. The consistency management is performed by the CPU consumer process that uses the GPU, which we call the *GPU controller*. Second, *no modifications to legacy CPU programs* are required to share files with GPUs or among themselves, despite the weak page cache consistency model. The consistency control logic is confined to the GPU controller process. Besides being backward compatible, this design simplifies integration with the CPU file I/O stack.

Consistency manager

Version vectors. GAIA maintains the version of each file-backed 4K page for every entity that might hold the copy of the page. We call such an entity a *page owner*. We use the well-known version vector mechanism (§2) to allow scalable version tracking for each page [32].

A page owner might be a CPU, each one of the GPUs, or the storage device. Keeping track of the storage copy is important because GAIA supports direct transfer from the disk to GPU memory. Consider the example in Figure 4, where a page is first concurrently modified by the CPU and the GPU, and then flushed to storage by the CPU. Flushing it from the CPU

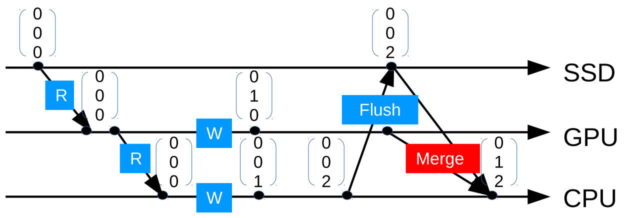


Figure 4: Version vectors in GAIA. The CPU flushes its replica to disk, the GPU keeps its version. The following CPU read must merge two replicas.

removes both the data and its version information from CPU memory. The next reader must be able to retrieve the most recent version of the page, which in our example requires *merging* the page versions on the disk and on the GPU. The storage entry in the version vector is always zero.

A new *Time Stamp Version Table (TSVT)* stores all the version vectors for a page. This table is located in the respective node of the page cache radix tree. The GPU entries are updated by the CPU-side GPU controller on behalf of the GPU. We choose the CPU-centric design to avoid intrusive modifications to GPU software and hardware.

Synchronizing system calls for consistency control. We introduce two new system calls to implement LRC.

`macquire(void *addr, size len, void* device)` must be called to ensure that the `device` accesses the latest version of the data in the specified address range. `macquire` scans through the address range on the `device` and invalidates (unmaps and drops) all the outdated local pages. When called for the CPU, it unmaps such pages from all the CPU processes. Thus, the following access to the page will cause a page fault trap to retrieve the most recent version of the page, as we describe later in (§4.1.1).

`mrelease(void *addr, size_t len, void* device)` must be called by the `device` that writes to the respective range to propagate the updates to the rest of the system. Similarly to `macquire`, this operation does not involve data movements. It only increases the versions of all the modified (since the last `macquire`) pages in the owner's entry of its version vector.

Tracking the status of CPU pages requires a separate `LRC_modified` flag in the page cache node, in addition to the original modified flag used by the OS page cache. This is because the latter can be reset by other OS mechanisms, e.g, flush, resulting in a missed update. The new flag is set together with the original one, but is reset by `mrelease` call as part of the version vector update.

Transparent consistency support for the CPU. GAIA does not change the original POSIX semantics when sharing files among CPU processes, because all the CPUs share the same replica of the cached page. However, `macquire` and `mrelease` calls must be invoked by *all* the CPU processes that might inadvertently share files with GPUs. In GAIA we seek to eliminate this requirement.

Our solution is to perform the CPU synchronization calls eagerly, combining them with `macquire` and `mrelease` calls issued on behalf of GPUs. The `macquire` call for the GPU is invoked after internally calling `mrelease` for the CPU, and `mrelease` of the GPU is always followed by `macquire` for the CPU. This change does not affect the correctness of the original LRC protocol, because it maintains the relative ordering of the acquire and release calls on different processors, simply moving them closer to each other.

Consistency and GPU kernel execution. GAIA's design does not preclude invocation of `macquire` and `mrelease` during the kernel execution on the target GPU. However, the current prototype does not support such functionality, because we cannot retrieve the list of dirty pages from the GPU while it is running, which is necessary for implementing `mrelease`. Therefore, we support the most natural scenario (also in Figure 3), which is to invoke `macquire` and `mrelease` at the kernel execution boundaries. Integrating these calls with the CUDA streaming API might be possible by using CUDA CPU callbacks [2]. We leave this for future work.

Page faults and merge

Page faults from any processor are handled by the CPU (hence, home-based LRC). CPU and GPU-originated page faults are handled similarly. For the latter, the data is moved to the GPU. The handler locates the latest versions of the page according to its TSVT in the page cache. If the faulting processor holds the latest version in its own memory (minor page fault), the page is remapped. If, however, the present page is outdated or not available, the page is retrieved from the memory of other processors or from storage.

This process involves handling the merge of multiple replicas of the same page. The overlapping writes to the same memory locations (i.e., the actual data races) are resolved via an "any write wins" policy, in a deterministic order (i.e., based on the device hardware ID). However, non-overlapping writes to the same page must be explicitly merged via 3-way merge, as in other LRC implementations [22].

3-way merge. The CPU creates a *pristine* base copy of the page when a GPU maps the page as writable. Conflicting pages are compared with their base copies first to detect the changes.

The storage overheads due to pristine copies might compromise scalability, as we discuss in (§6.1). The naive solution is to store a per-GPU copy of each page. A more space-efficient design might use a single page base copy for all the processors, employing copy-on-write and eagerly propagating the updates after the processor `mrelease`-s the page, instead of waiting for the next page fault (lazy update). Our current implementation uses the simple variant.

The overheads of maintaining the base copy are not large in practice. First, the base copy can be discarded after the page is evicted from GPU memory. Further, it is not required for read-only accesses, or when there is only one-page owner (excluding the storage) in the system. Most importantly, creating the base copy is not necessary for writes from CPU processes. This is because the CPU is either the sole owner, or the base copy has already been created for the modifying GPU.

Interaction with file I/O

Peer-caching. GAIA architecture allows a page replica to be cached in multiple locations, so that the best possible I/O path (or possibly multiple I/O paths in parallel) can be chosen, to serve page access requests. In particular, the CPU I/O request can be served from GPU memory. Note that access to the GPU-cached page does not invalidate it for the GPU.

A naive approach to peer-caching is to determine the best location individually for each page. However, this approach *degrades* the performance for sequential accesses by an order of magnitude, due to the overheads of small data transfers over the PCIe bus. Instead, GAIA leverages the OS prefetcher to optimize PCIe transfers. We modify the prefetcher to determine the data location in conjunction with deciding how much data to read at once. This modification results in a substantial performance boost, as we show in (§6.2).

Readahead for GPU streaming access. GPUs may concurrently run hundreds of thousands of threads that access large amounts of memory at once. GPU hardware coalesces multiple page faults together (up to 256, one for 64KB page). If the page faults are triggered by accesses to the memory-mapped file on the GPU, GAIA reads the file according to the GPU-requested locations and copies the data to GPU pages. We call such accesses *GPU I/O*.

We observe that the existing OS readahead prefetcher does not work well for GPU I/O. It is often unable to optimize streaming access patterns where a GPU kernel reads the whole file in data-parallel strides, one stride per group of GPU threads. The file accesses from the GPU in such a case appear random when delivered to the CPU due to the non-deterministic hardware schedule of GPU threads, thereby confusing the CPU read-ahead heuristics.

We modify the existing OS prefetcher by adjusting the upper bound on the read-ahead window to 16MB (64$\times$ of the CPU), but only for GPU I/O accesses. We also add `madvise` hints that increase the minimum read size from a disk to 512KB for sequential accesses. These changes allow the prefetcher to retrieve more data faster when the sequential pattern is recognized, but it does not fully recover the performance. Investigating a better prefetcher heuristic that can cope with massive multi-threading is left for future work.

Discussion

GAIA and cache-coherent accelerator architectures. Cache-coherent systems with global virtual address space may allow a simpler solution to the page cache management. However, we believe that cache coherence between the CPU and discrete accelerators is unlikely to fully replace existing systems soon. Despite the cache coherent technologies (CAPI [37]) having been available, the high cost and the need for industry-wide coordination on open interfaces have hindered their adoption thus far. Nor is it apparent how and to what extent these technologies will improve commodity applications (i.e., graphics, deep learning). Many additional open issues (for example, scalability) also must be addressed. Therefore, in GAIA we choose not to rely on cache-coherence among CPUs and accelerators.

No snapshot isolation. GAIA does not provide snapshot isolation. This is consistent with prior work on GPU file system support [36]. While adding such guarantee is possible, we did not find applications that require it.

Prefetching hints. Our current prototype could be extended to support more advanced prefetching hints similar to UVM [1]. For example, it could employ eager data copy into the page cache of a specific GPU that is known to exclusively access the data. We leave this for future work.

Using huge CPU pages. GAIA design and implementation is tailored for 4KB pages managed by the OS. However, GAIA can be adapted to support different page sizes as well, i.e. 2MB huge pages. Huge pages require only minor modifications to the TSVT management logic and tables, and might improve performance for applications with sequential file access. This is because transferring 2MB pages over PCIe is about 5$\times$ more efficient than 4KB pages. On the other hand, increasing the page sizes would affect the workloads with poor spatial locality, such as Mosaic (§6.3.1).

GAIA compatibility with other accelerators. GAIA's design can be extended to other GPUs and accelerators with paging capabilities. In fact, support for paging was introduced recently in AMD GPUs [4, 6]. However, implementation in GAIA would require an accelerator to expose minimal page management APIs, as we explain in the next section.

Implementation

GAIA implementation requires changing 3300 LOC and 1200 LOC in Linux kernel and the NVIDIA UVM driver respectively.

OS changes

Page cache with GPU pointers. In Linux, the page cache is represented as a per-file radix tree with each leaf node corresponding to a continuous 256KB file segment. Each leaf holds an array of addresses (or NULLs) of 64 physical pages caching the file content.

GAIA extends the tree leaf node data structure to store the addresses of GPU pages. We add 4 pointers per leaf node per GPU, to cover a continuous 256KB file segment (GPU page is

	Function	Purpose	UVM implementation\ GAIA emulation	Used by
Available in UVM	allocVirtual/allocPhysical mapP2V	allocate virtual/physical range map physical-to-virtual	cudaMallocManaged()	mmap
	freeVirtual/freePhysical	free virtual/physical range unmap virtual	cudaFree()	munmap
Emulated by GAIA	unmapV	Invalidate mapping in GPU	Migrate page to CPU	maquire
	fetchPageModifiedBit	Retrieve dirty bit in GPU	Copy page to CPU and compute diff	mrelease

Table 1: Main GPU Virtual Memory management functions and their implementation with UVM

64KB). The CPU only keeps track of file-backed GPU pages rather than the entire GPU physical memory. The leaf node stores all the versions (TSVT) for the 64 4KB pages.

Linking the page cache with the GPU page ranges. GAIA keeps track of all the GPU virtual ranges in the system that are backed by files to properly handle the GPU faults for file-backed pages. When `mmap` allocates a virtual address range in the GPU via the driver, it registers the range with GAIA and associates it with the file radix tree.

Data persistence. GAIA inherits the persistence semantics of the Linux file I/O. It updates both `msync` and `fsync` to fetch the fresh versions of the cache pages (similarly to the logic in the page fault handler) and write their contents to storage.

GPU cache size limit. GAIA enforces an upper bound on the GPU cache size by evicting pages. The evicted pages can be discarded from the system memory entirely (after syncing with the disk if necessary) or cached by moving them to available memory of other processors. In our current implementation, we cache the evicted GPU pages in CPU memory. We implement the Least Recently Allocated eviction policy [36], due to the lack of the access statistics for GPU pages.

Integration with GPU driver

The NVIDIA GPU driver provides no public interface for low-level virtual memory management. Indeed, giving the OS the full control over GPU memory management might seem undesirable. For example, only the vendors might have the intimate knowledge of the device/vendor-specific properties that require special handling, such as different page sizes, texture memory, alignment requirements, and physical memory constraints. However, we believe that a minimal subset of APIs is enough to allow generic advanced OS services for GPUs, such as unified page cache management, without forcing the vendors to give up on the GPU memory control.

We define such APIs in Table 1. The driver is in full control of the GPU memory, i.e., it performs allocations and implements the page eviction policy, only notifying the OS about the changes (callbacks are not shown in the table). We demonstrate the utility of such APIs for GAIA, encouraging GPU vendors to add them in the future.

Using the GPU VM management API

Implementing GAIA functionality is fairly straightforward, and closely follows the implementation of the similar functionality in the CPU OS. We provide a brief sketch below just to illustrate the use of the API.

mmap/munmap. When `mmap` with `MAP_ONGPU` is called, the system allocates a new virtual memory range in GPU memory via `allocVirtual` and registers it with the GAIA controller in the driver to associate it with the respective file radix tree, similar to the CPU `mmap` implementation. The `munmap` function performs the reverse operation using `unmapV` call.

Page fault handling. To serve the GPU page fault, GAIA determines the file offset and searches for the pages that cache the content in the associated page cache radix tree. If the most recent version of the page is found, and it is already located on the requesting GPU (minor page fault), GAIA maps the page at the appropriate virtual address using `mapP2V` call.

Otherwise (major page fault), GAIA allocates a new GPU physical page via `allocPhysical` call, populates it with the appropriate data (merging replicas if needed), updates the page's TSVT structure in the page cache to reflect the version, and maps the page to the GPU virtual memory. If necessary, GAIA creates a pristine copy of the page.

If a page has to be evicted to free space in GPU memory, GAIA chooses the victim page, unmaps it via `unmapV`, retrieves its dirty status via `fetchPageModifiedBit`, stores the page content on the disk or CPU memory if marked as dirty, removes the page reference from the page cache, and finally frees it via `freePhysical`.

Consistency system calls. GPU `macquire` scans through the GPU-resident pages of the page cache to identify outdated GPU page replicas and unmaps the respective pages via `unmapV`. GPU `mrelease` retrieves the modified status via `fetchPageModifiedBit` for all the GPU-resident pages in the page cache, and updates their versions in TSVT.

Functional emulation of the API

Implementing the proposed GPU memory management API without vendor support requires access to low-level internals of the closed-source GPU driver. Therefore, we choose to implement it in a limited form.

First, we use the user-level NVIDIA's UVM memory management APIs to implement a limited version of the API for allocation and mapping physical and virtual pages in GPU (refer to Table 1). Specifically, `cudaMallocManaged` is used to allocate the GPU virtual address range, and `cudaFree` to tear down the mapping and de-allocate memory.

Second, we modify the open-source part of the NVIDIA UVM driver. The GPU physical page allocation and mapping of the virtual to physical addresses are all part of the GPU page fault handling mechanism, yet they are implemented in the closed-source part of the UVM driver. To use them, GAIA modifies the open-source UVM page fault handler to perform the file I/O and page cache-related operations, effectively implementing the scheme described above (§5.2.1).

Finally, whenever the public APIs and open-source part of the driver are insufficient, we resort to emulation. To implement `unmapV`, we use a public driver function to *migrate the page* to the CPU, which also unmaps the GPU page. The `fetchPageModifiedBit` call is emulated by copying the respective page to the CPU *without unmapping it on the GPU* and computing *diff* with the base copy.

In Table 1 we highlight the emulated functions (in red) and specify where they are used.

Ultimately, this pragmatic approach allows us to build a functional prototype to evaluate the concepts presented in the paper. We hope that tese APIs will be implemented properly by GPU vendors in the future.

Limitations due to UVM

Our forced reliance on NVIDIA UVM leads to several limitations. The page cache lifetime and scope are limited to those of the process where the mapping is created, as UVM does not allow allocating physical pages that do not belong to a GPU context. Therefore, the page cache cannot be shared among GPU kernels belonging to different CPU processes. Further, the maximum mapped file size is limited by the size of the maximum UVM buffer allocation, which must fit in the CPU physical memory. Finally, UVM controls memories of all the system GPUs, *preventing us from implementing a distributed page cache between multiple NVIDIA GPUs*.

These limitations are rooted in our use of UVM, and *are not pertinent to GAIA design*. They limit the scope of our evaluation to a single CPU process and a single GPU, but allow us to implement a substantial prototype to perform thorough and reliable performance analysis of the heterogeneous page cache architecture.

Evaluation

We evaluate the following aspects of our system [3]:

- Benefits of peer-caching and prefetching optimizations;
- Memory and compute overheads;
- End-to-end performance in real-life applications with read and write-sharing.

Platform. We use an Intel Xeon CPU E5-2620 v2 at 2.10GHz with 78GB RAM, GeForce GTX 1080 (with 8GB GDDR) GPU and 800GB Intel NVMe SSD DC P3700 with 2.8GB/s sequential read throughput. We use Ubuntu 16.04.3 with kernel 4.4.15 that includes GAIA modifications, CUDA SDK 8.0.34, and NVIDIA-UVM driver 384.59.

Performance cost of functional emulation. The emulation introduces performance penalties that do not exist in the CPU analogues of the emulated functions. In particular, `unmapV` constitutes more than 99% of `macquire` latency, and `fetchPageModifiedBit` occupies nearly 100% of `mrelease`.

These functions are expensive only because of the lack of the appropriate GPU driver and hardware support. We expect them to be as fast as their CPU analogues if implemented by GPU vendors. For example, `mmap` or `mprotect` calls for a 1GB region take less than 10 μseconds on the CPU. If implemented for the GPU, they might last slightly longer due to over-PCIe access to update the page tables.

To be safe, we conservatively assume that `unmapV` and `fetchPageModifiedBit` are as slow as *10 msec* in all the reported results. These are the worst-case estimates, yet they allow us to provide a *reliable estimate* of GAIA performance in future systems.

Evaluation methodology. We run each experiment 11 times, omit the first result as a warmup, and report the average. We flush the system page cache before each run (unless stated otherwise). We do not report standard deviations below 5%.

Overhead analysis

Impact on CPU I/O. GAIA introduces additional version checks into the CPU I/O path. To measure the overheads, we run the standard TIO benchmark suite [23] for evaluating CPU I/O performance. We run random/sequential reads/writes using the default configuration with 256KB I/O requests, accessing a 1GB file. As a baseline, we run the benchmark on the unmodified Linux kernel 4.4.15.

We vary the number of supported GPUs in the system as it affects the number of version checks. We observe less than 1% and up to 5% overhead for 32GPUs and 160GPUs respectively for random reads, and no measurable overheads for sequential accesses. We conclude that *GAIA introduces negligible performance overheads for legacy CPU file I/O*.

Memory overheads. The main dynamic cost stems from pristine page copies for 3-way merge. However, common read-only workloads require no such copies, therefore incurring no extra memory cost. Otherwise, the memory overheads depend on the write intensity of the workload. GAIA creates one copy for every writable GPU page in the system.

The static cost is due to the addition of version vectors to the page cache. This cost scales linearly with the utilized

[3] GAIA source code is publicly available at https://github.com/acsl-technion/GAIA.

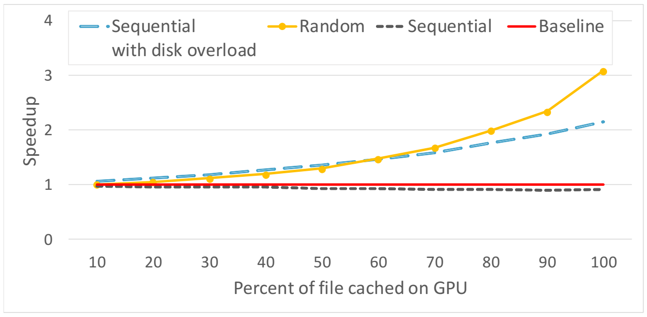


Figure 5: CPU I/O speedup of GAIA over unmodified Linux. Higher is better.

GPU memory size as GPU versions are stored only for GPU-resident pages, but grows quadratically with the number of GPUs. With 512 bytes per version vector entry, the worst-case memory overhead (all GPUs fill their memories with files) and 100 GPUs each with 8GB memory, the overhead reaches about 5GB, which is less than 1% of the total utilized GPU memory (800GB).

Microbenchmarks

Benefits of peer caching. We measure the performance of *CPU* POSIX `read` accesses to a 1GB file which is partially cached in GPU memory. Thus, these accesses result in reading the GPU-resident pages from the GPU instead of the SSD. We vary the cached portion of the file, reading the rest from the SSD. We run three experiments: (1) random reads (2) sequential reads, and (3) sequential reads while the SSD is busy serving other I/O requests. We compare to unmodified Linux, where all the data is fetched from the SSD.

Figure 5 shows that peer-caching can boost the CPU performance by up to 3× for random reads, and up to 2× when the SSD is under load. GAIA is no faster than SSD for sequential reads, however. This is due to the GPU DMA controller bottleneck, stemming from the lack of public interfaces to program its scatter-gather lists. Therefore, GAIA is forced to perform the I/O one GPU page at a time.

False sharing. We run the same CPU-GPU false-sharing microbenchmark as in §3.1. We map the shared buffer from a file, and let the CPU and the GPU update it concurrently with non-overlapping writes. GAIA merges the page when the GPU kernel terminates. We compare with the execution where each of the processors writes into a private buffer, without false sharing. We observe that GAIA with shared buffer is *the same* as the baseline. The cost of merging the page in the end is constant per 4KB page: 1.4 μsecond.

This experiment highlights the fact that *GAIA eliminates the overheads of false sharing entirely*.

Streaming read performance. We evaluate the case where the GPU reads and processes the whole file. We use three kernels: (1) a zero-compute kernel that copies the data from the input pointer into an internal buffer, one stride at a time per threadblock; (2) an unmodified LUD kernel, representative of compute-intensive kernels in Rodinia benchmark suite [19]; (3) a *closed-source* cuBLAS SGEMM library kernel [7]. We modify their CPU code such that the GPU input is read from a file rather than from memory, as in prior work [40].

We evaluate several techniques to read files into the GPU:

1. **CUDA-[must fit in GPU memory]**: read from the host, copy to GPU;
2. **GPUfs-[requires GPU code changes]**: read from the GPU kernel via GPUfs API. GPUfs uses 64KB pages as in GPU hardware;
3. **UVM**: read from the host into UVM memory (physically residing in CPU), read from the GPU kernel;
4. **GAIA**: map the file into GPU, read from the GPU kernel.

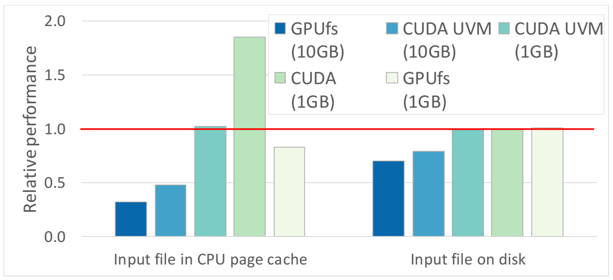
We implement all four variants for the zero-compute kernel. LUD and cuBLAS cannot run with GPUfs because that requires changing their code. We run the experiments with two input files, one smaller and one larger than GPU memory.

Figure 6a shows the results of reading a 1GB and 10GB file for the zero-compute kernel. GAIA is competitive with UVM and GPUfs for all of the evaluated use cases, but slower than CUDA when working with a small file in the CPU page cache, due to inefficiency of the GPU data transfers in GAIA. In CUDA, the data is copied in a single transfer over PCIe, whereas in GAIA the I/O is broken into multiple non-consecutive 64KB blocks, which is much slower.

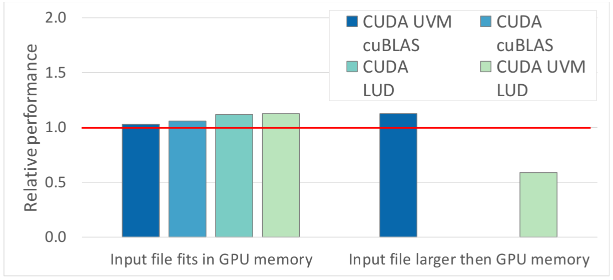
GAIA is faster than UVM when reading cached files due to the UVM's extra data copy, and faster than GPUfs with 1GB because of GPUfs trashing. The thrashing occurs because the GPUfs buffer cache in GPU memory is not large enough to store the whole file. Pages are constantly evicted as a result, introducing high runtime overheads.

Figure 6b shows the results of processing a 1GB and a 10GB file for the compute-intensive LUD and cuBLAS kernels for techniques (1), (3) and (4). Large files cannot be evaluated with CUDA as they do not fit in GPU memory. GAIA is faster than LUD on UVM, yet slightly slower than the other methods. With cuBLAS, the specific access pattern results in a large working set, causing trashing with GAIA.

The insufficient coordination with the OS I/O prefetcher in the current implementation makes it slower when the file fits GPU memory. The performance can be improved via a more sophisticated prefetcher design and transfer batching in future GPU drivers. With large files, however, *GAIA is on par with and faster than UVM, while offering the convenience of using a GPU-agnostic OS API and supporting unmodified GPU kernels.*



(a) Zero-compute kernel, normalized to GAIA. Higher is better.



(b) LUD and cuBLAS small and large files (from disk). Higher is better.

Figure 6: Streaming read I/O performance analysis

	Prefetched	On disk
GAIA (sec)	1.2	2.9
UVM (sec)	11.4 (↑ 9×)	17.8 (↑ 6×)
ActivePointers(4 CPU threads) (sec)	0.5 (↓ 2×)	1.7 (↓ 2×)
ActivePointers(1 CPU thread) (sec)	0.6 (↓ 2×)	5.5 (↑ 2×)

Table 2: Image collage: GAIA vs. UVM vs. ActivePointers

Applications

Performance of on-demand data I/O

We use an open-source image collage benchmark [34]. It processes an input image by replacing its blocks with "similar" tiny images from a large indexed dataset stored in a file. The access pattern depends on the input: each block of the input image is processed separately to generate the index into the dataset, fetching the respective tiny image afterward. The dataset is 19GB, thus it does not fit in memory of our GPU. While only about 25% of the dataset is required to completely process one input, the accesses are input-dependent.

We compare three implementations: (1) original ActivePointers, (2) UVM and (3) GAIA. For the last two we modify the original code by replacing ActivePointers [34] with regular pointers. In UVM the dataset is first read into a shared UVM buffer. In GAIA the file is `mmap`-ed into GPU memory.

Both ActivePointers and GAIA allow random file access from the GPU, but they differ in that they rely on software-emulated and hardware page faults respectively.

Table 2 shows the end-to-end performance comparison. GAIA is 9× and 6× faster than UVM, because it accesses the data in the file on-demand, whereas in UVM the file must be read in full prior to kernel invocation.

We investigate the performance difference between ActivePointers and GAIA. We observe that ActivePointers use four I/O threads on the CPU to serve GPU I/O requests. Reducing the number of I/O threads to only one as in GAIA provides a more fair comparison. In this case, GAIA is 2× faster when reading data from disk, but still 2× slower when the file is prefetched. The reasons are not yet clear, however.

We conclude that GAIA's on-demand data access is competitive with highly optimized ActivePointers, and *significantly faster* than UVM.

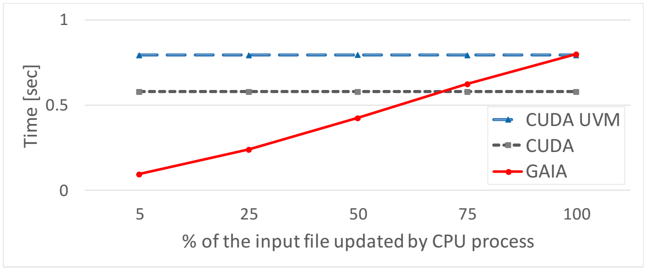


Figure 7: Graph processing with dynamic graph updates, while varying the number of updates. Lower is better.

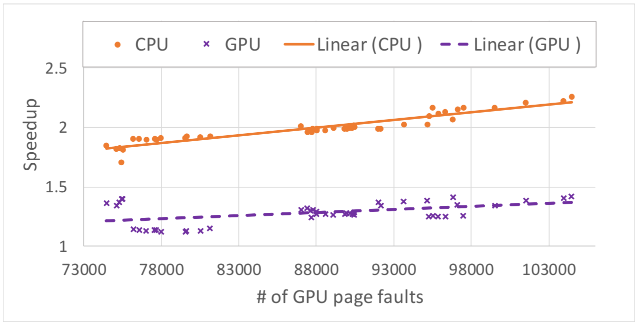
Dynamic graph processing with Gunrock

We focus on a scenario where graph computations are invoked multiple times, but the input graph periodically changes. This is the case, for example, for a road navigation service such as Google Maps, which needs to accommodate the traffic changes or road conditions while responding to user queries.

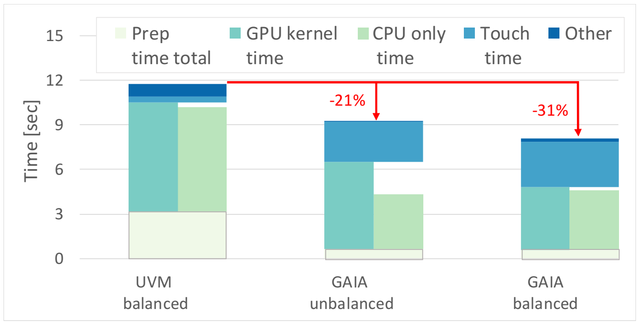
We assume the following service design. There are two processes: an updater daemon (producer) and a GPU-accelerated compute server (consumer), which share the graph database. The daemon running on the CPU (1) retrieves the graph updates (traffic status) from an external server; (2) updates the graph database file; and (3) signals to the compute service to recalculate the routes. The latter reads these updates from the file each time it recomputes in response to a user query. The producer updates only part of the graph (i.e., edge weights representing traffic conditions). This design is modular, easy to implement, and supports very large datasets. Similar producer-consumer scenarios have also been used in prior work [15].

We use an *unmodified* Gunrock library [38] for fast graph processing. We run the Single Source Shortest Path algorithm provided with Gunrock, modifying it to read input from a file, which is updated in a separate (legacy) CPU process that uses standard I/O (no consistency-synchronizing system calls). The file updates and graph computations are interleaved: the compute server invokes the updater, which in turn invokes the computations, and so on. We run a loop of 100 iterations and measure the total running time.

We implement the benchmark using UVM and GAIA, and also compare it with the original CUDA implementation. For



(a) CPU and GPU speedups over UVM while varying the number of computations. Higher is better.



(b) End-to-end runtime comparison. Other: memory (de)/allocation. Touch time = time of first access from CPU, includes merge for GAIA. Prep time = time of reading input. Lower is better.

Figure 8: Performance impact of false sharing in image stitching.

both UVM and CUDA, the *whole file* must be copied into a GPU-accessible buffer on every update because the locations of the modified data in the file are unknown. No such copy is required for GAIA, where each GPU kernel invocation is surrounded by `macquire` and `mrelease` calls.

We run the experiment on the uk_2002 input graph provided with Gunrock examples, extended to include edge weights. The file size is 5.2GB.

Figure 7 shows the performance as a function of the portion of the graph being updated. GAIA is faster than the alternatives with fewer changes to the file, automatically detecting the changes in the pages that were indeed modified. For the worst case of full file update (above 75%) GAIA becomes slower than the original CUDA implementation. This is due to the inefficiency of the GPU data transfers, as we also observed in Fig. 6a. This experiments shows *the utility of GAIA's fine-grain consistency control, which enables efficient computations in a read-write sharing case.*

Effects of false sharing in image stitching

We consider an image processing application used in optical microscopy to acquire large images. Microscopes acquire these images as grids of overlapping patches that are then stitched together. Recent works accelerate this process on multiple GPUs [18, 16]. The output image is split into non-overlapping sub-images, where each GPU produces its output independently, and the final result is merged into a single output image.

This application benefits from using GAIA to write into the shared output file directly from GPUs, eliminating the need to store large intermediate buffers in CPU memory. We seek to show the effects of false sharing in this workload if it were implemented with GAIA. Unfortunately, we cannot fully evaluate GAIA on multiple GPUs, as we explained earlier (§5.2.3). Instead, we implement only its I/O-intensive component in the CPU and the GPU.

Both the CPU and GPU load their patches, with already pre-computed output coordinates. Each patch is sharpened via a convolution filter, and then is written to the output file. The convolution filter is invoked several times per output to explore a range of different compute loads. In GAIA, we map both the input patches and the output file into the CPU and the GPU. For the UVM baseline, the inputs are read into UVM memory before kernel invocation.

We run the experiment on a public Day2 Plate [3] stitching dataset with 5.3GB of input tiles and 1.3GB of output. We use the patch locations included in the dataset to write the patches, which ensures realistic access to the output file.

We split the output image over the vertical dimension and load-balance the input such that both the CPU and the GPU run about the same time in the baseline implementation which writes into a UVM shared output buffer. The patch coordinates determine the amount of false sharing in this application.

Figure 8a shows the speedup of GAIA over UVM while varying the amount of computations per patch. We observe up to 45% speedup for the CPU, and over 2.3× speedup for the GPU. This experiment corroborates the conclusions of the microbenchmark in (§3.1), now in a realistic write-sharing workload. GAIA enables *significant performance gain by eliminating false sharing in the unified page cache*.

To evaluate the complete system rather than the performance of each processor separately, we pick one of the runtime parameters in the middle of Figure 8a, and measure the end-to-end runtime, including file read, memory allocation, and page merging. We prefetch the input into the page cache on the CPU to highlight the performance impact.

Figure 8b shows the results. For exactly the same runtime configuration GAIA *outperforms UVM by 21%*. Moreover, GAIA allows further improvements by rebalancing the load between the processors (GPU runs faster without false sharing), achieving overall 31% performance improvement.

Related work

To the best of our knowledge, GAIA is the first system to offer a distributed page cache abstraction for GPUs that is integrated into the OS. Our work builds on prior research in the following areas.

Memory coherence for distributed systems. Lazy Release Consistency [22, 10] serves as the basis for GAIA. GAIA implements the home-based version of LRC [41]. Munin [14] implements eager RC by batching the updates. GAIA adopts this idea by using LRC-dirty bit to batch multiple updates. Version Vectors is an idea presented in [32] for detecting mutual inconsistency in multiple-writers systems. We believe that GAIA is the first to apply these ideas to building a heterogeneous OS page cache.

Heterogeneous, multi-core, and distributed OS design for systems without cache coherence. Several proposed OSes support heterogeneous and non-cache coherent systems by applying distributed system principles to their design [29, 13, 12]. None of these systems implements shared page cache support, which is the main tenet of our work.

K2 [25] is a shared-most OS for mobile devices running over several coherence domains. It implements a sequentially consistent software DSM for the OS structures shared among the domains. K2 DSM implements sequential consistency, which is a strict coherence model. Similarly to GAIA, K2 relies on page faults as the trigger for consistency operations. K2 DSM implementation of the coherence model relies heavily on the underlying hardware. Thus, even read-only sharing is not possible as it requires a different MMU for handling reads and writes. GAIA does not have this limitation. Solros [27] proposes a data-centric operating system to enable efficient I/O access for XeonPhi accelerators. Solros includes a buffer cache for faster I/O, but unlike GAIA, it is limited to host memory, and does not explicitly discuss inter-accelerator sharing.

The file system in the FOS multikernel [39] shares data between cores but is limited to read-only workloads. Hare [21] is a file system for non-cache-coherent multicores in which each node may cache file data in the private memory and a shared global page cache. Hare uses close-to-open semantics, which GAIA refines. Distributed OSes such as Sprite [30], Amoeba [28], and MOSIX [11] aim to provide a single system image abstraction and in particular, coherent and transparent access to files from different nodes, but they achieve this via process migration/use home nodes to forward their I/O.

GPU file I/O. GPUfs [36] allows file access from GPU programs and implements a distributed weakly consistent page cache with session semantics. ActivePointers [34] extend GPUfs with a software-managed address translation and page faults, enabling GPU memory mapped files. However, unlike GAIA, ActivePointers require intrusive changes to GPU kernels, its session semantics is too coarse-grain for our needs, and its page cache is not integrated with the CPU page cache, thus lacks peer-caching support. SPIN [15] integrates direct GPU-SSD communications into the OS. As in GAIA peer-caching, SPIN adds a mechanism for choosing the best path for file I/O to the GPU. However, it does not extend the page cache into the GPU.

Memory management in GPUs. NVIDIA Unified Virtual Memory (UVM) and Heterogeneous Memory Management (HMM) [4] allow both the CPU and GPU to share virtual memory, migrating the pages to/from GPU/CPU upon page fault. Neither currently supports memory mapped files on x86 processors. Both introduce a strict coherence model that suffers from false sharing overheads. Asymmetric Distributed Shared Memory [20] is a precursor of UVM that emulates a unified address space between CPUs and GPUs in software.

IBM Power9 CPU with NVIDIA V100 GPUs provides hardware support for coherent memory between the CPU and the GPU. Since GPU memory is managed as another NUMA node, memory-mapped files are naturally accessible from the GPUs. This approach would not work for commodity x86 architectures which lack CPU-GPU hardware coherence.

Dragon [26] extends NVIDIA UVM to enable GPU access to large data sets residing in NVM storage, by mapping them into the GPU address space. Dragon focuses exclusively on accessing the NVM from the GPU, does not integrate GPU memory into a unified page cache, has no peer-caching support, and does not consider CPU-GPU file sharing, all of which are the main contributions of our work.

Conclusions

GAIA enables GPUs to map files into their address space via a weakly consistent page cache abstraction over GPU memories that is fully integrated with the OS page cache. This design optimizes both CPU and GPU I/O performance while being backward compatible with legacy CPU and unmodified GPU kernels. GAIA's implementation in Linux for NVIDIA GPUs shows promising results for a range of realistic application scenarios, including image and graph processing. It demonstrates the benefits of lazy release consistency for write-shared workloads.

GAIA demonstrates the importance of integrating GPU memory in the OS page cache, and proposes the minimum set of memory management extensions required for future OSes to provide the unified page cache services introduced by GAIA.

Acknowledgments

We thank Idit Keidar, Isaac Gelado and our shepherd David Nellans for their valuable feedback. We also gratefully acknowledge the support of the Israel Science Foundation (grant No. 1027/18).

References

[1] 'Beyond GPU Memory Limits with Unified Memory on Pascal'. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/.

[2] CUDA toolkit documentation - cudaStreamAddCallback(). https://docs.nvidia.com/cuda/

cuda-runtime-api/group__CUDART__STREAM.html.

[3] Data dissemination: Reference Image Stitching Data. https://isg.nist.gov/deepzoomweb/data/referenceimagestitchingdata.

[4] Heterogeneous Memory Management (HMM). https://www.kernel.org/doc/html/v4.18/vm/hmm.html.

[5] IEEE 1003.1-2001 - IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(R)). https://standards.ieee.org/standard/1003_1-2001.html.

[6] Radeon's next-generation Vega architecture. https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf.

[7] cuBLAS Library User Guide. https://docs.nvidia.com/pdf/CUBLAS_Library.pdf, October 2018.

[8] Everything you need to know about Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-/know-about-unified-memory.pdf, February 2018.

[9] NVIDIA Tesla V100 GPU accelerator data sheet. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf, March 2018.

[10] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, February 1996.

[11] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[12] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[13] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44. ACM, 2009.

[14] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. *SIGPLAN Notices*, 25(3):168–176, February 1990.

[15] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 167–179, Santa Clara, CA, 2017. USENIX Association.

[16] Timothy Blattner, Walid Keyrouz, Joe Chalfoun, Bertrand Stivalet, Mary Brady, and Shujia Zhou. A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images. In *2014 43rd International Conference on Parallel Processing*, pages 1–9, Sept 2014.

[17] William J. Bolosky and Michael L. Scott. False Sharing and Its Effect on Shared Memory Performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, volume 57, 1993.

[18] Joe Chalfoun, Michael Majurski, Tim Blattner, Kiran Bhadriraju, Walid Keyrouz, Peter Bajcsy, and Mary Brady. MIST: Accurate and Scalable Microscopy Image Stitching Tool with Stage Modeling and Error Minimization. *Scientific reports*, 7(1):4988, 2017.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[20] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 347–358. ACM, 2010.

[21] Charles Gruenwald III, Filippo Sironi, M Frans Kaashoek, and Nickolai Zeldovich. Hare: A File System for Non-cache-coherent Multicores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 30:1–30:16. ACM, 2015.

[22] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21. ACM, 1992.

[23] Mika Kuoppala. Tiobench-threaded I/O bench for Linux, 2002.

[24] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, November 1989.

[25] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. *ACM SIGARCH Computer Architecture News*, 42(1):285–300, 2014.

[26] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. DRAGON: breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 32. IEEE Press, 2018.

[27] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proceedings of the Thirteenth EuroSys Conference*, page 36. ACM, 2018.

[28] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[29] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 221–234. ACM, 2009.

[30] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23–36, February 1988.

[31] Tom Papatheodore. Summit System Overview. `https://www.olcf.ornl.gov/wp-content/uploads/2018/05/Intro_Summit_System_Overview.pdf`, June 2018.

[32] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[33] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Feb 2007.

[34] Sagi Shahar, Shai Bergman, and Mark Silberstein. ActivePointers: A Case for Software Address Translation on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 596–608, June 2016.

[35] Sagi Shahar and Mark Silberstein. Supporting Data-driven I/O on GPUs Using GPUfs. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 12:1–12:11. ACM, 2016.

[36] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498. ACM, 2013.

[37] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[38] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A High-performance Graph Processing Library on the GPU. *SIGPLAN Notices*, 51(8):11:1–11:12, February 2016.

[39] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, April 2009.

[40] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 13–24. IEEE Computer Society, 2015.

[41] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 75–88. ACM, 1996.