An OS page cache for heterogeneous systems

Tanya Brokhman

Technion - Computer Science Department - M.Sc. Thesis MSC-2019-15 - 2019

An OS page cache for heterogeneous systems

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Tanya Brokhman

Submitted to the Senate of
the Technion — Israel Institute of Technology
Adar 5779 Haifa March 2019

The research thesis was done under the supervision of Prof. Mark Silberstein in the Electrical Engineering Department.

Acknowledgments

I would like to thanks my advisor Mark Silberstein for all the support and guidance during my masters.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Efficient access to files from GPUs is of growing importance in data-intensive applications. Unfortunately, current OS design 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, much tighter integration of GPU memory into the OS page cache and file I/O mechanisms is required. Achieving such integration is one of the primary goals of this thesis.

We propose a principled approach to integrating GPU memory with an OS page cache. GAIA extends the CPU OS page cache to the physical memory of accelerators to enable seamless management of the distributed page cache (spanning CPU and GPU memories) by the CPU OS. We adopt a variation of CPU-managed lazy relaxed consistency shared memory model while maintaining compatibility with unmodified CPU programs. We highlight the main hardware and software interfaces to support this architecture, and show a number of optimizations, such as tight integration with the OS prefetcher, to achieve efficient peer-to-peer caching of file contents. 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 among physical memories 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.

3. Optimizes the readahead prefetcher to support accesses to caches in GPUs.

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

Chapter 1

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 [6], 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, much tighter *integration of GPU memory into the OS page cache and file I/O mechanisms* is required. Achieving such integration is one of the primary goals of this work.

Prior works demonstrate that mapping files into GPU memory provides a number of benefits [31, 33, 32]. Files can be accessed from the GPU using an intuitive pointer-based programming model. 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, data can be shared between legacy CPU and GPU-accelerated processes, and applications with data-dependent access patterns can be implemented.

Extending the OS page cache into GPU memory is advantageous even for CPU I/O performance. Today, with the advent of servers with up to 8 GPUs, the total GPU memory available (100-200GB) is large enough to be feasible for systems tasks, i.e., for caching files, used by legacy CPU processes. As we show empirically, doing so may boost the I/O performance by up to $3\times$ compared to accesses to a high-performance SSD (§6). Finally, the OS management of the page cache in the GPU memory may allow caching GPU file read accesses directly in the GPU

page cache, preventing CPU-side page cache pollution [12].

Unfortunately, today's commodity systems fall short of providing full integration of GPU memory with the OS page cache. ActivePointers [31] 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 [5]. NVIDIA's Unified Virtual Memory (UVM) [30] and the Heterogeneous Memory Management (HMM) [3] 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 high-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).

Recent high-end IBM Power-9 systems introduced hardware cache-coherent shared virtual memory between CPUs and discrete GPUs [27]. GPU memory is managed as another NUMA node, thus enabling the OS 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.

CPUs with integrated GPUs support coherent shared *virtual* memory in hardware. In contrast to discrete GPUs, however, integrated GPUs lack large separate physical memory, obviating the need for supporting such memory by the OS.

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 OS file I/O layer. With GAIA, CPU programs use regular file I/O to share files with GPUs, whereas `mmap` with a new `MMAP_ONGPU` flag makes the mapping accessible to the GPU kernels, thus providing support for GPU accesses to shared files. To prototype and evaluate this architecture, we leverage hardware page fault support in NVIDIA GPUs. This approach allows access to memory-mapped files from *unmodified* GPU kernels.

This work makes the following contributions:

- We characterize the overheads of CPU-GPU false sharing in existing systems

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

(§3.1). We propose a unified page cache which avoids false sharing entirely by using a lazy release consistency model [19, 7].

- We extend the OS page cache to control the *unified* (CPU and GPU) page cache and its consistency (§4.1). 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 (§6.2.1).
- We present a fully functional generic implementation in Linux, *not tailored* to any particular GPU.
- We prototype GAIA on NVIDIA Pascal GPU, leveraging its support for page faults by modifying public parts of the GPU driver and reliably emulating the closed-source components.
- We evaluate GAIA using real workloads, including (1) an *unmodified* Graph processing framework - Gunrock [35], (2) a Mosaic application that creates an image collage from a large image database [31], and (3) a multi-GPU image stitching application [13, 15] that determines the optimal way to combine multiple image tiles, demonstrating the advantages and the ease-of-use of GAIA for real-life scenarios.

Chapter 2

Background

We briefly and informally reiterate the main principles of Version Vectors and several existing consistency models.

2.1 Strict consistency

The most stringent consistency model is called strict consistency. It is defined by the following condition:

Definition 2.1.1 *[34] Any read to a memory location x returns the value stored by the most recent write operation to x.*

Informally, in strict consistency models each write performed by some node on a shared object becomes immediately visible to all of the nodes in the system. The writes on the same object are seen by all nodes in the same order. The strict consistency model can have a disproportional effect on performance in DSM systems sharing mutable-data due to frequent invalidations of the mutable object.

2.2 Weak consistency

In the weak consistency model, synchronization accesses are sequentially consistent. Before a synchronization access can be performed, all previous regular data accesses must be completed. Before an ordinary data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the

problem of consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation.

2.3 Release consistency

Release consistency (RC) [19] is a form of relaxed memory consistency that allows the effects of shared memory accesses to be delayed until certain specially labeled accesses occur.

In the Release consistency model the programmer controls the visibility of writes by means of the synchronization operations *acquire* and *release*. 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.

Definition 2.3.1 *[19] A system is release consistent if:*

1. *Before an ordinary access can be performed with respect to any other processor, all previous acquires must be performed.*
2. *Before a release can be performed with respect to any other processor, all previous ordinary reads and writes must be performed.*
3. *Special accesses are sequentially consistent with respect to one another.*

RC implementations can delay the effects of shared memory accesses as long as they meet the constraints of Definition 2.3.1.

2.3.1 Eager release consistency

Eager release consistency is based on Munin's write-shared protocol, introduced in [11, 19]. A processor delays propagating its modifications to shared data until it comes to a *release*. At that time, it propagates the modifications to all other processors that cache the modified pages as a *diff* of each modified page. A *diff* describes the modifications made to the page, which are then merged in the other cached copies. The *release* blocks until acknowledgments have been received from all other cachers of the page.

No consistency-related operations occur on an *acquire*.

On an access miss, a message is sent to the directory manager for the page. The directory manager forwards the request to the current owner, and the current owner sends the page to the processor that incurred the access miss.

2.3.2 Lazy release consistency (LRC)

In Lazy Release Consistency (LRC) [19, 7] the propagation of updates to a page (the *diffs*) is delayed until *acquire*. At synchronization time, the acquiring node gets updated by the other nodes. In order for the acquiring node to discover whether or not the memory accessed is the most up-to-date, page-based DSM systems make use of the page fault mechanism from the virtual memory system. If a node accesses a stale copy of the page, a page fault occurs. The faulting node must get the most up-to-date copy of the page from another node or nodes which have that copy.

There are two implementations for the LRC protocol: homeless and home-based. The difference between the two is what happens at the time of the page fault on an acquiring node:

Homeless LRC. In the homeless protocol, when there is a page fault on the acquiring node, the faulting node requests timely ordered *diff*-s of the page from the last writer or writers. Once all the *diff*-s are received by the faulting node, it then merges them to construct the up-to-date version of the page. Since the *diff*-s may be distributed over many nodes, this process is complicated and time-consuming.

Home-based LRC. In the home-based protocol, a home node is assigned to a page. To get the most up-to-date version, the faulting node requests it from the home node. When a node writes on a page it does not own, it updates the home of the page at synchronization time (*release*) by sending *diff*-s according to LRC. Obtaining the updated page in this manner is much simpler and faster since the home-based protocol asks for the most-up-date copy of the fault page, not the timely ordered *diff*-s of the page, and the amount of data traffic is not more than the page size. The location of the most up-to-date page is also easy to determine.

2.4 Version vectors

Version vectors (VVs) [28] are used in distributed systems to keep track of replica versions of an object. The description below is adopted from Parker [28].

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 i_{th} vector entry counts the number v_i 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 (see below), the new vector is created by taking the largest version among the two for each entry in the vector.

Version vectors provide a convenient way to identify conflicting replicas. A version vector V *dominates* another vector W if every component of V is no less than W; V strictly dominates W if it dominates W and one component of V is greater, which implies that the replica with W is strictly more recent. Two object replicas that are incomparable by the domination relation are conflicting.

Chapter 3

Consistency model considerations

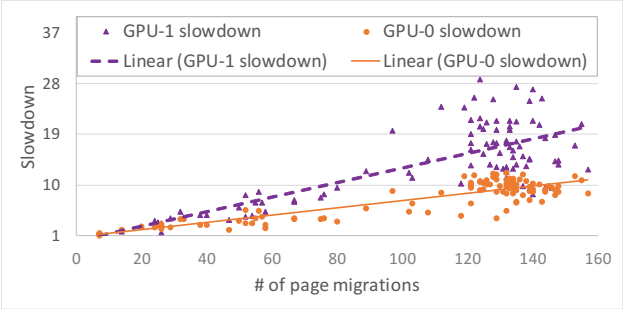
The choice of a model for the unified page cache is an important design question. We briefly describe the options we considered and explain our choice of LRC.

POSIX: strong consistency. POSIX defines that writes are immediately visible to all the processes using the same file [4]. In x86 systems without coherent shared memory with GPUs connected via a high latency (relative to local memory) PCIe bus, supporting strong consistency for a unified page cache would be inefficient [33].

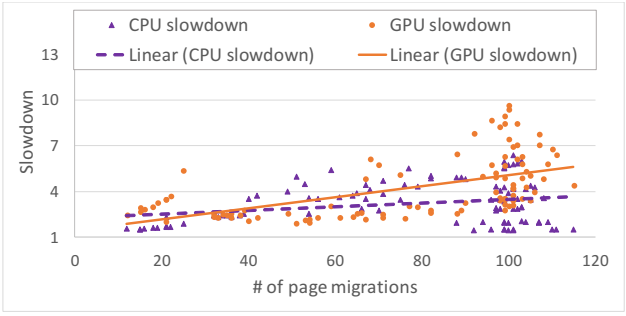
GPUfs: session semantics. GPUfs [33] introduces a GPU-side library for file I/O from GPU kernels. GPUfs implements a distributed page cache with session semantics. Session semantics, however, couple between the file open/close operations and data synchronization. As a result, they cannot be used with `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.

UVM: page-level strict coherence. NVIDIA UVM [30] and Linux (HMM) [3] implement strict coherence [21] at the level of a GPU memory page (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.

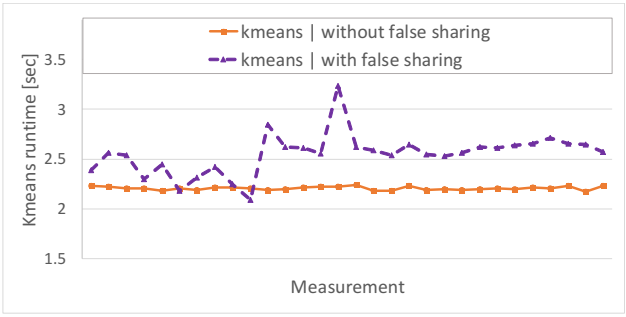
Although this model might seem appealing for page cache management, it suffers from sporadic performance degradation due to *false sharing*.



(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 [29]

Figure 3.1: Impact of false sharing on GPU and system performance

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 [14]. 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 [14]. False sharing has been also reported in multi-GPU applications that use NVIDIA's UVM [30].

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 [13]. 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. Specifically, 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).

3.1 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 3.1a shows the scatter graph of the measurements. False sharing causes a slowdown that grows with the number of page migrations, reaching $28\times$, and results in large runtime variance [1].

Figure 3.1b 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.

[1] The difference in the slowdown between two GPUs stems from imperfect synchronization between them. Thus the one invoked first (GPU0) is able to briefly run without contention.

System impact of false sharing. We find that false sharing among GPUs affects the performance of the system as a whole. We run the kmeans multi-threaded benchmark from the Phoenix benchmark suite [29] *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). In this configuration, the GPU activity *should not* interfere with kmeans if invoked in parallel.

However, *we observe significant interference when GPUs experience false sharing*. Figure 3.1c 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 performance isolation properties.

Implications for Unified Page Cache design. *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, which only worsens with scaling the number of GPUs. At the conceptual level, building a system-level service with such inherent limitations is undesirable.

Thus, we chose to build a unified cache that follows the lazy-release consistency model and sidesteps the false sharing issues entirely.

3.2 GAIA LCR

The CPU-multi-GPU architecture requires some changes to the earlier described LRC model, mainly because the GPU is subordinate to the CPU: it cannot perform operations on its own and is linked to a CPU running process. Thus it can neither maintain any meta data related to the shared object nor initiate *release* or *acquire* events. Due to the above, GAIA follows the home-based LRC model with the CPU taking on the role of a ”directory manager”, keeping note of all of the *acquire* and *release* events in the system.

Chapter 4

Design

Overview. Figure 4.1 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 location(s) 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, a *peer-caching* mechanism allows the page to be retrieved via the most efficient path, e.g., from GPU memory for the CPU I/O request, or directly from storage for the GPU as in SPIN [12]. This mechanism is integrated with the OS readahead prefetcher to achieve high performance (§6.2.1).

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 4.2 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

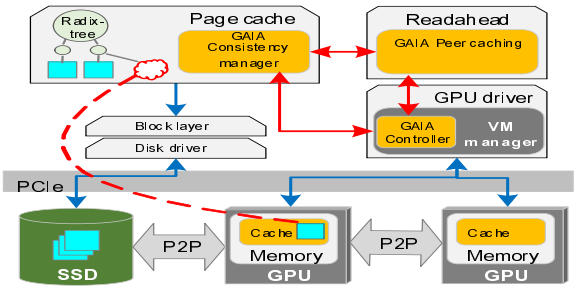


Figure 4.1: GAIA high-level design in the OS kernel. The modified parts are highlighted.

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 4.2: 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.

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 the backward compatibility, this design simplifies integration with the CPU file I/O stack.

4.1 Consistency manager

Version vectors. GAIA maintains a version number of each file-backed 4K page for every entity that might hold the copy of the page. We call such an entity a *page*

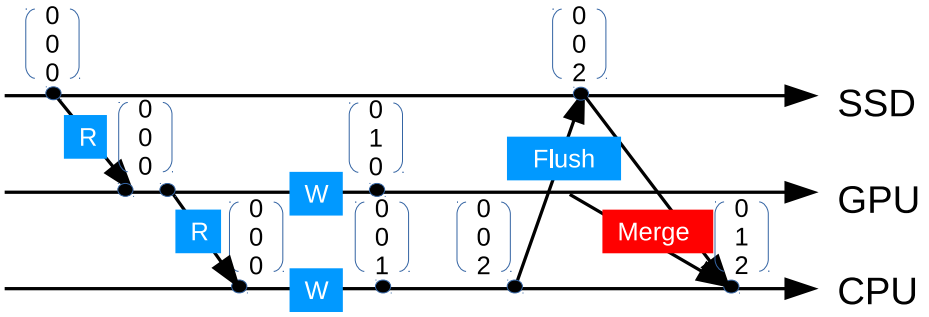


Figure 4.3: Version vectors in GAIA. The CPU flushes its replica to disk, the GPU keeps its version. The following CPU read must merge two replicas.

owner. We use the well-known version vector mechanism to allow scalable version tracking for each page [28] (refer to §2).

In GAIA, 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 of a page to/from GPU memory. For example, consider a page that is concurrently modified by the CPU and the GPU and flushed to storage by the CPU. Flushing it from the CPU removes both the data and its version information from CPU memory (refer to Figure 4.3). The next reader must be able to retrieve the most recent version of the page, which in our example is a *merge* of the page versions on disk and the one 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. As a result, the next access to the page from the device (or the CPU) 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.

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 one another.

Consistency and GPU kernel execution. GAIA does not preclude invocation of `macquire` and `mrelease` during the kernel execution on the target GPU. However, in the current prototype, it is not possible to invoke them, 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 4.2), 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 [1]. We leave this for future work.

4.1.1 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 other memories or from the storage.

This process involves handling the merge of multiple replicas of the same page. The overlapping writes are resolved via an ”any page wins” policy, in a deterministic order (i.e., based on the device hardware ID). However, non-overlapping writes must be explicitly merged via 3-way merge, as in other LRC implementations [19].

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. Nor is it 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 because the base copy has already been created for the modifying GPU.

4.2 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 posibly multiple 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 (§6.2.1).

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 GPU hardware scheduler, 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.

Chapter 5

Implementation

We implement GAIA in the Linux kernel and integrate it with the NVIDIA UVM driver, by modifying/adding about 3300 and 1200 LOC respectively.

5.1 OS changes

5.1.1 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 256K file segment. Each leaf holds an array of addresses (or NULLs) of 64 physical pages holding the file data.

GAIA extends the leaf data structure to store the addresses of GPU pages. We add 4 pointers per leaf node per GPU, to cover a contiguous 256KB file segment (GPU page is 64KB). Note that the CPU only keeps track of file-backed GPU pages rather than all the GPU physical memory.

5.1.2 Page version management

In order to support version control, each leaf node stores all the versions (TSVT) for each of the 64 4KB pages.Linux Radix trees introduce several features driven by kernel-specific needs, including the ability to associate tags with specific entries. For example, the flush process retrieves all pages in the radix tree tagged with the PAGE_CACHE_TAG_DIRTY tag and flushes them to disk, clearing the tag afterwards. GAIA introduces a new page cache tag, PAGE_CACHE_TAG_CPU_DIRTY. The purpose of this flag is to mark all the pages that were updated by the CPU

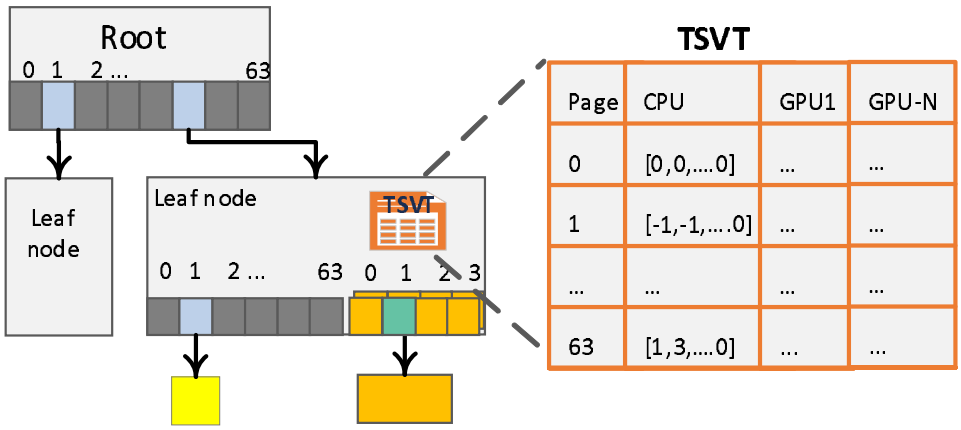


Figure 5.1: GAIA modifications to the Linux radix-tree.

but whose version has yet to reflect the update. In order to implement this design, GAIA piggybacks the PAGE_CACHE_TAG_DIRTY tag, setting PAGE_CACHE_TAG_CPU_DIRTY together with it. For memory mapped files, a physical page is allocated for a process upon the first access attempt, during the page fault handling sequence. If the page fault occurred due to a write attempt, the page software dirty bit is set and the page is tagged with both the PAGE_CACHE_TAG_DIRTY and PAGE_CACHE_TAG_CPU_DIRTY page cache tags. If the page is allocated due to a read access attempt, the OS will unset its ”writable” flag in order to catch a write access and set the software dirty bit accordingly. PAGE_CACHE_TAG_CPU_DIRTY will not be cleared by the flush process when propagating the changes to disk, but rather by the `macquire` system call, after increasing the CPU page version in TSVT.

The modifications introduced by GAIA are presented in Figure 5.1.

5.1.3 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.

5.1.4 Data persistence

GAIA inherits the persistence semantics of the Linux file I/O. Namely, it updates

both `msync` and `fsync` to retrieve the most updated version of the file content and write it to persistent storage.

5.1.5 GPU cache size limit

GAIA enforces 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 policy [33], due to the lack of access statistics about GPU pages.

5.2 Integration with GPU driver

The NVIDIA GPU driver provides no public interface for low-level virtual memory management.

One might argue that handing the OS the full control of GPU memory is undesirable. For example, various device/vendor-specific architectural policies, such as different page sizes, texture memory management, alignment requirements, and physical memory constraints, are better be handled by the vendors. However, we believe that a minimal subset of APIs is enough to allow 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 5.1. 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.

5.2.1 Using the proposed 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.

	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 5.1: Main GPU Virtual Memory management functions and their implementation in UVM

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 most recent content of the data (possibly merging multiple replicas), updates the page's TSVT structure in the page cache to reflect the new page replica, and maps the page to the GPU virtual memory. If necessary, GAIA creates a pristine copy of the page to support merge later on.

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

`macquire` is called prior to GPU kernel invocation. In order to transparently support *release* for CPU processes, it is performed as part of the `macquire` system call implementation. The first step identifies all the CPU-dirty pages (tagged with PAGE_CACHE_TAG_CPU_DIRTY tag) and increases their version in the TSVT. Once all the CPU updates are reflected in the TSVT, `macquire` scans through the GPU-resident pages of the page cache to identify outdated GPU page replicas and unmaps the respective pages via `unmapV`.

`mrelease` is called after GPU kernel completion in order to reflect the updates made by the GPU in the TSVT data structures. In order to transparently support *acquire* for CPU processes, it is performed as part of the `mrelease` system call implementation. GPU `mrelease` retrieves the modified status via `fetchPage-ModifiedBit` for all the GPU-resident pages in the page cache. For each page modified by the GPU (1) the page version is increased, (2) the page is unmapped from all CPU processes mapping it, and (3) the page is invalidated in all other GPUs holding a version of the page.

This guarantees that a page fault will occur upon the next access to a modified page by any processor, resulting in bringing the latest page version to its memory.

5.2.2 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 available user-level GPU APIs to achieve some of the required functionality. We use NVIDIA's UVM memory management to implement a limited version of the API for allocation and mapping physical and virtual pages in GPU (refer to Table 5.1). Specifically, `cudaMallocManaged` is used to allocate the GPU virtual address range, and `cudaFree` to tear down the mapping and de-allocate both physical and virtual memory.

Second, we modify the open-source part of the NVIDIA UVM driver. In particular, 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 5.1 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 work. We believe that in the future these APIs will be implemented properly by GPU vendors.

5.2.3 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 reliance on 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.

Chapter 6

Evaluation

We evaluate the following aspects of our system:

- 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, 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 emulated functionalities introduce a performance penalty that does not exist in the CPU analogues of the simulated 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 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 takes less than 10 μseconds on the CPU. If implemented for the GPU, they might be 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 explicitly flush the system page cache before each run (unless stated otherwise). We do not report standard deviations below 5%.

6.1 Overhead analysis

6.1.1 Impact on CPU I/O

GAIA introduces additional version checks into the CPU I/O path. To measure this overhead for legacy CPU applications we, run the TIO benchmark suite [20] for evaluating CPU file 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.*

6.1.2 Memory overheads

The main dynamic cost stems from pristine page copies maintained for proper 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 GPU memory size, since GPU versions are stored only for GPU-resident pages, and quadratically with the number of GPUs.

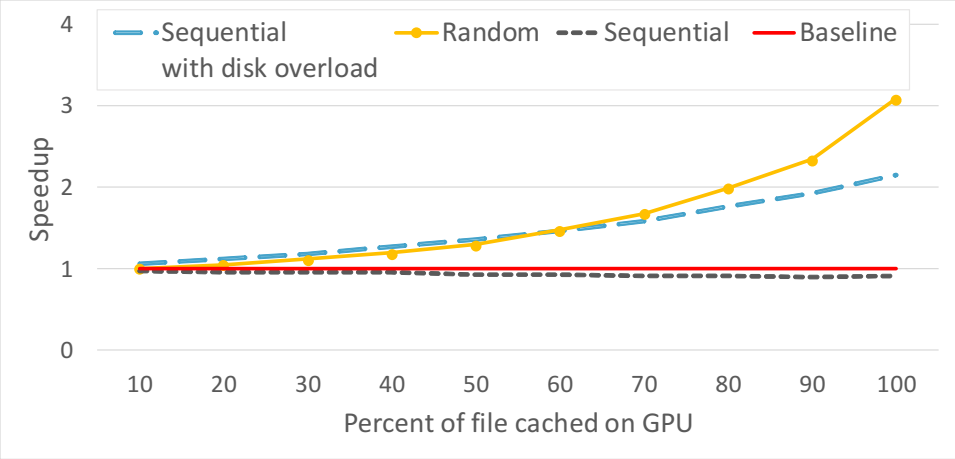


Figure 6.1: CPU I/O speedup of GAIA over unmodified Linux. Higher is better.

With 512 bytes per version vector entry, the worst-case memory overhead (all GPUs fill their memories with files) for 100 accelerators each with 8GB memory reaches about 5GB, which is less than 1% of the total amount of the utilized GPU memory (800GB).

6.2 Microbenchmarks

6.2.1 Benefits of peer caching

We measure the performance of *CPU POSIX* `read` *accesses* to a 1GB file while it is partially cached in GPU memory We vary the cached portion of the file, reading the reminder from the SSD. We run three experiments: (1) random reads (2) sequential reads, and (3) sequential reads while the SSD is busy serving other processes. We compare to unmodified Linux.

Figure 6.1 shows that peer-caching can boost the CPU performance by up to 3× for random reads, and up to 2× faster when the SSD is loaded. 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 transfers one GPU page at a time.

6.2.2 False sharing

We run the same CPU-GPU 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, followed by the merge phase after the kernel terminates. We compare GAIA with running each of the processors on 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: 7 μsecond.

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

6.2.3 Streaming read performance

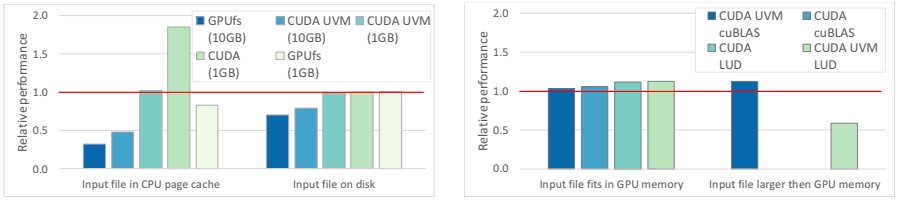
We evaluate the case where the GPU reads and processes the whole file. We use three kernels: (1) a zero-compute kernel that copies the data from the input pointer into an internal buffer, one stride at a time per threadblock; (2) an unmodified LUD kernel, representative of compute-intensive kernels in the Rodinia benchmark suite [16]; (3) a *closed-source* cuBLAS SGEMM library kernel [5]. We modify the CPU code of LUD and cuBLAS such that the input is read from a file as in prior work [37].

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. Large files cannot be evaluated with CUDA.

Figure 6.2a 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



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

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

Figure 6.2: Streaming read I/O performance analysis

page cache, due to the inefficiency of the GPU data transfers. 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 extra data copy, and faster than GPUfs with 10GB because of GPUfs trashing.

Figure 6.2b 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). GAIA is faster than LUD on UVM, yet slightly slower than other methods. With cuBLAS, the specific access pattern results in a large working set, causing trashing with GAIA.

The insufficient interplay with the OS I/O prefetcher in the current implementation makes it less preferable 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.*

6.3 Applications

6.3.1 Performance of on-demand data I/O

We use an open-source image collage benchmark [31]. 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. Only about 25% of the dataset is required to completely process one input, yet the subset of read data is input-dependent. The

	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 6.1: Image collage: GAIA vs. UVM vs. ActivePointers

dataset of 19GB does not fit in GPU memory.

We compare three alternative implementations: (1) original ActivePointers, (2) UVM and (3) GAIA. For the last two we modify the original code by replacing ActivePointers [31] with regular pointers. In UVM the entire dataset is read into a shared UVM buffer. With 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 6.1 shows the end-to-end performance of GAIA over the alternatives. GAIA is $9\times$ and $6\times$ 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 $2\times$ faster when reading data from disk, but still $2\times$ 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.

6.3.2 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 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 up-

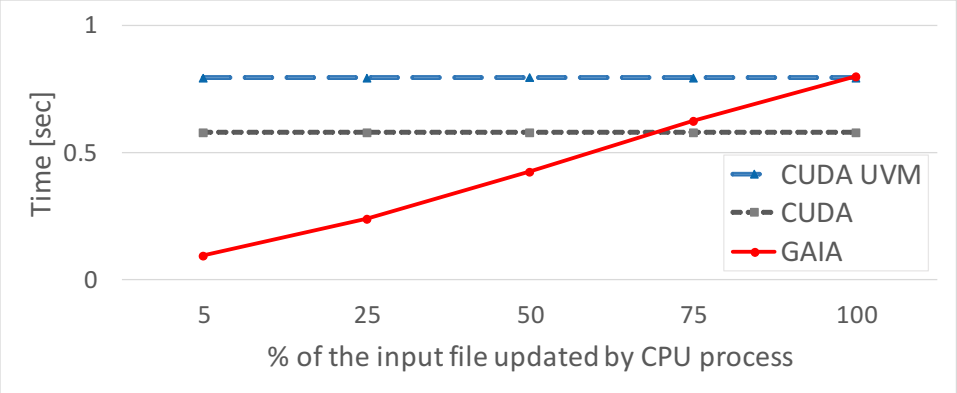
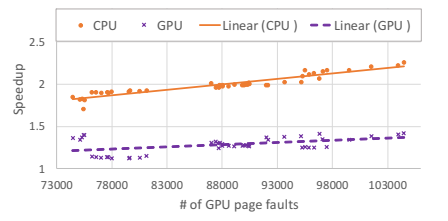
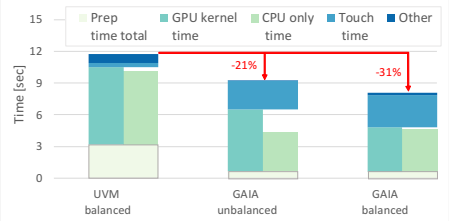


Figure 6.3: Graph processing with dynamic graph updates, while varying the number of updates. Lower is better.



(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 6.4: Performance impact of false sharing in image stitching.

dates (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 [12].

We use an *unmodified* Gunrock library [35] for fast graph processing. We run the Single Source Shortest Path application 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 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.

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 6.3 shows the performance while varying the portion of the updated graph. 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 experiments shows *the utility of GAIA's fine-grain consistency control, which enables efficient computations in a read-write sharing case.*

6.3.3 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 [15, 13]. 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 [2] stitching dataset with 5.3GB of input tiles and 1.3GB of output, and use the correct patch locations included in the dataset. 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 (UVM). Note that the page write coordinates determine the amount of false sharing in this applications.

Figure 6.4a shows the speedup of GAIA over UVM while varying the number of computations. 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). By *eliminating the false sharing in its page cache, GAIA enables significant performance gain.*

To evaluate the complete system, we pick one of the runtime parameters in the middle of Figure 6.4a, 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 effect.

Figure 6.4b 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, achieving overall 31% performance improvement.

Chapter 7

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.

7.1 Memory coherence for distributed systems

Lazy Release Consistency [19, 7] serves as the basis for GAIA. GAIA implements the home-based version of LRC [38]. Munin [11] 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 [28] 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.

7.2 Heterogeneous, multi-core, and distributed OS design for systems without cache coherence

Helios. The Helios operating system [25] targets heterogeneous systems with multiple programmable devices. It is designed to simplify the task of writing, deploying and tuning applications for heterogeneous platforms. It consists of satellite kernels (which are basically micro kernels) that allow developers to write applications against familiar OS APIs and abstractions. Each satellite kernel consists of scheduler, memory manager, namespace manager and a code to coordinate and communicate with other kernels. All traditional OS drivers and services (such as

the file system) run as individual processes. Helios is not designed to address the cache-coherence usecase; rather, it provides transparent communication between system processes, regardless of where in the system the processes execute. Nor do the authors address the question of establishing a distributed page cache abstraction. One of the hardware primitives required by Helios is interrupt support. At the time the paper was published, there were no GPUs supporting interrupts. Thus Helios was not tested with a GPU.

Barrelfish. The authors of Barrelfish [10] argue that the challenge of heterogeneous hardware is best met by rethinking OS architecture using ideas from distributed systems. A shared memory operating system is indeed a poor fit for today's heterogeneous hardware due to the very high cost of hardware cache-coherence. Several peripheral devices, such as smart NICs and GPUs, don't support cache coherence with a CPU. The Barrelfish solution is to present a new, zero-share OS design, in which the entire state is replicated and consistency is maintained using agreement protocols. Barrelfish is a multikernel operating system with a single kernel per core. System state is replicated among cores and each core updates its local copy. Consistency is maintained by exchanging messages using distributed-systems protocols. All system services run in user level processes as in microkernel. The authors do not address the question of a distributed page cache, but it might be implemented as another system-service running in a user level process.

Popcorn. Popcorn [9] is a Linux-based multi-kernel OS in which several instances of the Linux kernel boot up, one per core or group of cores. The kernels communicate between themselves via a new low-level message-passing layer in order to maintain a common OS state that is partially replicated among the kernels. Popcorn presents a single system image to the user that includes: single file system namespace, single process identification (PID), inter-process communication (IPC), and CPU namespaces. Popcorn allows processes to migrate transparently between different kernels. Most of the process related data-structures are replicated between kernels and kept consistent throughout the lifetime of the process. The consistency is achieved via the inter-kernel communication but the protocols are not explained. Page-cache functionality is not mentioned. The authors state that no OS-level resources are shared between tasks running on different kernels. Instead these resources are replicated and kept consistent through protocols tailored to satisfy the requirements of each replicated component. The protocols are not elaborated on.

K2. K2 [22] is a recent shared-most OS for mobile devices running over several coherence domains. Its goal is to allow better power savings by combining

low-performance cores with high-performance cores on the same SoC. Because hardware coherence forbids the weaker core to be much weaker than the strong one, hardware coherence is not applicable to such SoC. As a result, programming becomes complicated since the programmer must take explicit responsibility for coherency between the two cores, also making this architecture not backward compatible with the millions of existing mobile applications.

K2 addresses this issue by spanning the OS across multiple domains, ensuring the coherency between the domains in a matter transparent to the user. K2 introduces a *shared-most* OS model where most of the OS services are replicated in all domains with transparent state coherence, while a small, selective set of services operates independently in each domain. K2 presents a coherent, single Linux image to applications, therefore preserving the widely used programming model.

The transparent coherence for the replicated services is implemented by K2 via the Distributed Shared Memory (DSM). 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. In implementing sequential consistency, the K2 DSM performs coherence operations on each fault and keeps the order of coherence messages. The K2 DSM adopts a page-based granularity, using 4KB page as the smallest memory unit that is kept coherent. This is for leveraging MMU to trap accesses to shared memory. GAIA, on the other hand, uses 64KB page granularity in order to leverage the GPU page fault mechanism to trap accesses to shared memory.

K2 DSM implementation of the coherence model relies heavily on the underlying hardware (hardware mailboxes for inter-domain communication and MMU for shared memory access). Thus, even read-only sharing is not possible as it requires a different MMU for handling reads and writes. This design limitation does not exist in GAIA, since it is not restricted by a specific hardware architecture.

SOLROS. SOLROS [23] proposes a data-centric operating system architecture that aims to improve the sub-optimal I/O path between co-processors and the residence of the input data. SOLROS is a response to the need for a centralized and coordinated I/O operation manager, necessary because a global view of the system is required to provide best I/O path between the co-processor and the origin/destination of the data. Similarly to GPUfs, SOLROS implemented a light-weight file system stub that runs on the co-processor (data-plane OS), forwards all file system services to the host OS for processing, and waits for completion notification from the host OS. As in SPIN, SOLROS recognizes the best data path for the re-

quired data (host cache or disk) and initiates a P2P transfer directly from disk to co-processor memory if required. SOLROS caches ”files accessed by many coprocessors” in its buffer cache for faster I/O. Unlike GAIA, the caching is limited to host OS memory only. It is unclear from the SOLROS design what consistency model is used when more than one coprocessor updates the same file. GAIA handles the multiple updates use case by supporting the LRC model described earlier.

None of these systems considered a distributed page cache, which is the main tenet of our work.

FOS. FOS [36] is a new operating system targeting manycore systems with scalability as the primary design constraint, where space sharing replaces time sharing to increase scalability. The authors believe that a new OS design is required, since the number of cores in future systems is much larger then what the contemporary OS was designed to handle. The traditional evolutionary approach of redesigning OS subsystems will cease to work because the rate of increasing parallelism is much faster than the rate at which OS subsystems can be redesigned to accommodate the changes.

The authors investigate current operating systems scalability bottlenecks and base FOS design on their conclusions:

1. HW lock should be avoided.

2. Separate the operating system execution resources from the application execution resources.

3. Avoid global cache coherent shared memory.

FOS is designed as a collection of (distributed) services, each running on a dedicated set of cores, communicating between themselves via message passing. Each service in FOS is bound to a distinct processing cores and by doing so do not contend with user end applications for implicit resources such as TLB and caches. This results, for example, in a fast context switch when a user end application requires an OS service.

In FOS, the cores which provide the file system and the virtual memory management do not share memory with each other, as opposed to traditional OS design, which relies heavily on this sharing. FOS replaces wide sharing of data with techniques utilized by distributed services. For example, FOS utilizes replication and aggressive caching to replace a widely shared page cache. Whether the design

proposed by FOS will yield better performance for such services remains an open question.

The file system service is not yet implemented and is currently limited to read-only workloads.

Hare. Hare [18] is a new file system that provides a POSIX-like interface on multicore processors without cache coherence. Similarly to GAIA, Hare strives to provide an illusion of a single, coherent system to the user, taking care of system consistency transparently to the user. The Hare prototype does not support a persistent on-disk file system: all file data is lost after a reboot. Instead, Hare provides an in-memory file system. This limitation is absent in GAIA, as all file data is synchronized to disk. Since Hare implements only in-memory file system, its use of DRAM can be viewed as an external disk; all cores have access to the shared DRAM but each core can also cache file data in its own private cache. This architecture is similar to GAIA and requires a consistency protocol implementation. The authors of Hare choose to implement the close-to-open consistency protocol, adopted also in GPUfs. As explained in 3, the close-to-open consistency model is unsuitable to GAIA.

Hare assumes a single failure domain: either all cores and DRAM are working, or the entire machine crashes. If only a subset of the system components crash, Hare will malfunction as a whole.

Sprite. Sprite [26] is an experimental network operating system, designed for a cluster of multiprocessor workstations at the University of California at Berkeley back in 1988. Although Sprite is similar to Unix in the provided kernel calls, its main goal is to encourage resource sharing by easing the process for developers. Sprite achieves this by implementing three new facilities: (1) Transparent network file system, (2) a simple mechanism for sharing writable memory between processes, and (3) a mechanism for migrating processes between workstations to take advantage of idle machines.

With Sprite users are able to manipulate files in the same way they do on a single machine; the distributed nature of the file system and the techniques used to access remote files remain invisible to users under normal conditions. To system administrators the file system appears as a collection of domains, each containing a structured portion of the overall hierarchy. The domains can be stores on different servers. The domains structure is managed by *prefix-tables*, maintained privately by each client machine kernel and invisible to user processes. When the user initiates an operation on a remote file, the file location is retrieved from the prefix-table

and the required operation is forwarded via a RPC call to the server holding the file.

The Sprite file system is implemented using large caches of recently used file blocks stored in the main memories of both clients (user workstations) and servers (holding the file). The caches allow improved file system performance by eliminating disk access, and they increase system scalability by reducing the load on the network and on the servers. The file caches are maintained similarly to today's Linux implementation. Sprite uses a delayed-write approach in which new data is first written to the local cache and written through to disk only when the block is evicted or 30 seconds have elapsed since the block was last modified.

Sprite guarantees strong consistency; each *read()* will return the most up-to-date data for a file, regardless of how the file is being used around the network. This is done by maintaining the file version. The synchronization point used by Sprite is file-open when the latest file version on the server is compared to the version of the cached data on the client. In the case of concurrent write-sharing, Sprite disables client caching of the file.

Amoeba. The Amoeba [24] operating system was born from a research effort whose goal was the seamless connection of multiple computers. Users are provided with the illusion of a single powerful timesharing system, when, in fact, the system is implemented on a collection of machines, potentially distributed among several countries.

Amoeba is an object-oriented distributed operating system. Objects are abstract data types such as files, directories and processes, and are managed by server processes. A client process carries out operations on an object by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object. Afterwards the server sends a reply message back to the client, which unblocks the client. A Unix emulation service was developed to run existing software on Amoeba.

Amoeba architecture consists of four principal components. First are the workstations, one per user, which run window management software. Second are the pool processors, a group of CPUs that can be dynamically allocated as needed, used, and then returned to the pool. Third are the specialized servers, such as directory, file and block servers, database servers, bank servers, boot servers and various other servers with specialized functions. Fourth are the wide-area network gateways, which are used to link Amoeba systems at different sites in possibly different countries into a single, uniform system.

All the Amoeba machines run the same kernel, which primarily provides communication services. The basic idea was to keep the kernel small, not only to enhance its reliability, but also to allow as much of the operating system as possible to run as user processes, providing for flexibility and experimentation.

Amoeba introduces a new file server that implements an immutable file store, with principal operations of *read-file* and *create-file*. For garbage collection purposes there is also a *delete-file* operation. Immutable files are advantageous in that processes can cache them without inconsistency being a concern. When an application wants to change the file, it reads the complete file into its memory. After making the required changes, a file is created with the new contents. In this operation model applications may write over each other's changes.

MOSIX. MOSIX [8] is a set of enhancements of the Berkeley BSD operating system that supports preemptive process migration for load-balancing and memory ushering in a cluster of PCs. These algorithms are designed to respond dynamically to variations in resource usage among the nodes, by migrating processes from one node to another, preemptively and transparently to the application programs, to improve the overall performance. Each node in MOSIX is capable of operating as an independent system and there are no master-slave relationships between the nodes.

In MOSIX, each user interacts with the multicomputer via the user's "home" node (workstation or server). All the user's processes seem to run at the home node. Processes that migrate to other (remote) nodes use local (in the remote node) resources whenever possible, but interact with the user's environment through the user's home node.

The overall goal is to maximize the performance by efficient utilization of the network-wide resources. In order to efficiently load-balance between the nodes, each node maintains sufficient knowledge about the available resources in other nodes. All of the nodes execute the same algorithms, which continuously attempt to reduce the load differences between pairs of nodes by migrating processes from overloaded nodes to less loaded nodes.

Any user process can be migrated at any time, to any available node, transparently to the user application. The total cost of the process migration includes a fixed cost, to establish a new process frame in the remote site, and an additional cost, proportional to the number of pages copied. In practice, only the page table and the dirty pages of the process are copied.

MOSIX introduces a new protocol for efficient kernel communication that was

specifically designed to reduce the overhead, e.g., of communication between the process and its home node, when it is executing in a remote site, and for process migration.

ActivePointers. The primary goal of ActivePointers [31] 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. The authors implement a pure-software design for virtual memory management by taking a GPU-centric approach. They claim that this design is preferable to the CPU-centric approach taken by NVIDIA Pascal for example, since it doesn't suffer from scalability bottlenecks arising when the CPU has to sustain a potentially high load while handling multiple concurrent page-faults triggered by massively parallel GPU code.

ActivePointers extends GPUfs with a software-managed address translation and page faults, enabling memory mapped files. The design introduces a novel address translation mechanism that caches the mappings in HW registers. In ActivePointers a page can not be swapped out of GPU memory if there is a thread mapping it, requiring the system to keep track of all mapped pages. This design requires the programmer to keep close track of the amount of memory accessed with ActivePointers, so as not to over-commit GPU memory and to release an *apointer* when it is no longer required. This limitation is not present in GAIA. Mapped pages can be swapped out of GPU memory and will be brought back in upon next access, leaving the programmer oblivious to the GPU memory size.

Similarly to GAIA, major page faults in ActivePointers require CPU involvement to read file data from disk. However, ActivePointers handles major page faults by several CPU threads reading the file data in parallel, aggregating several host-to-GPU transfers on the host and issuing a single copy over the PCI. This enhancement is missing in GAIA, where all file reads and host-to-GPU transfers are done in a sequential manner by a single thread.

Unlike GAIA, ActivePointers requires intrusive changes to the GPU kernels, its session semantics is too coarse-grain for our needs, and its page cache is not integrated with the CPU page cache. Therefore does not support peer-caching.

SPIN. SPIN [12] aims to improve the data path between the disk and GPUs, eliminating the redundant copy to CPU memory by using P2P directly to/from disk. It integrates P2P into the standard OS file I/O stack, dynamically activating P2P when appropriate, transparently to the user. When data required by GPU partially resides in CPU page cache, SPIN combines P2P with copy from CPU memory

in order to achieve best performance for the required data retrieval. In SPIN, the GPU is considered to be the destination of the data; the focus is thus on choosing the best path for file I/O to the GPU. GAIA peer-caching utilizes the ideas presented in SPIN, focusing on the CPU as the destination of the required data and choosing the best source for the required data: GPU memory or disk. SPIN doesn't extend the page cache into GPU memory and thus does not support peer-caching as defined in GAIA.

7.3 Memory management in GPUs

NVIDIA Unified Virtual Memory (UVM) and Heterogeneous Memory Management (HMM) [3] 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 [17] is a precursor of UVM that emulates a unified address space between CPUs and GPUs in software.

IBM Power-9. In recent IBM Power-9 CPUs with NVIDIA Volta GPUs, GPU memory is managed as another NUMA node. Memory-mapped files are thus naturally accessible from the GPUs. This system supports hardware cache coherence in cache line granularity. Due to the very small amount of shared memory (64 bytes), false sharing is not an issue in this system and the enhancements offered by GAIA are redundant. However, in commodity x86 architectures that lack CPU-GPU hardware coherence, NUMA abstractions cannot be used for GPU memory: placing CPU-accessible pages in GPU memory will break the expected memory behavior for CPU processes.

DRAGON. DRAGON enhances NVIDIA UVM to enable the GPU access to large data sets residing in NVM by mapping the memory space addressable by the GPU device code directly to NVM devices. Although the implementation is somewhat similar to how *mmap()* was implemented in GAIA with the new MAP_ONTO_GPU flag, there are several conceptual differences. In DRAGON, dragon_map() is designed *solely* for when the data in question has to be brought from NVM to GPU memory for processing by the GPU. Although in the DRAGON implementation the host-allocated pages are inserted into the system page cache, they cannot be used by a regular CPU process in parallel with the GPU. Such behavior is undefined and will not work correctly in the current design. In DRAGON, the system

page cache is used *solely* for the write-back of modified files to NVMe.

DRAGON supports eventual consistency with the NVMe or with other applications wishing to use the same file. If a synchronization is required, it should be performed explicitly by the user by calling dragon_sync(), which will result in flushing *all* of the file data from GPU memory all the way to NVMe (by calling vfs_sync as part of dragon_sync() implementation). In contrast to GAIA, DRAGON does not handle false sharing between multiple processors and is not designed to address this case. The authors of DRAGON do not discuss the idea of establishing a distributed page cache abstraction.

Chapter 8

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.

Bibliography

[1] CUDA toolkit documentation - cudaStreamAddCallback(). https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] Nikolay Sakharnykh. 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.

[31] 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.

[32] 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.

[33] 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.

[34] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

מערכות כגון ה-Power-9 של IBM תומכות בcache coherence בחומרה, בין הCPU ל discrete GPUs. בזכות זה הן תומכות בשיתוף זיכרון וירטואלי בין הCPU לGPU [27]. זיכרון GPU מנוהל במערכת ההפעלה כNUMA node נוסף, ולכן מאפשר למערכת ההפעלה לתמוך במיפוי קבצים תוך זיכרון הGPU. למרבה הצער, cache coherence בחומרה בין הCPU וה- GPU אינו נתמך במערכות המבוססות עם x86 ולא ברור אם ומתי תמיכה זו תתווסף.

מעבדים עם GPUs מובנים תומכים בזיכרון וירטואלי משותף קוהרנטי, בניגוד ל discrete GPUs לGPU מובנה בתוך הCPU אין זיכרון פיזי משל עצמו ולכן אין צורך שניהול זיכרון נוסף ע"י מערכת ההפעלה.

בכדי לפתור את המגבלות שתוארו לעיל, אנו מציעים את GAIA, אריכטקטורה חדשה עבור מטמון דפים מבוזר התומכת ב weak consistency. GAIA מרחיבה את מטמון הדפים ומנגנוני I/O של מערכת ההפעלה להכיל גם את הזיכרון של הGPUs במערכת ההטרוגנית החדשה. בעזרת GAIA, תוכניות CPU הרוצות לשתף קבצים עם תוכניות GPU משתמשות במנגנוני I/O הקיימים בדיוק כמו מקודם, ואילו עבור מפתחי GPU, GAIA מרחיבה את פקודת mmap וע"י הוספת דגל חדש MMAP_ONGPU מאפשרת למפות קבצים לתוך זיכרון הGPU. בשביל לממש אב טיפוס למערכת החדשה אנחנו משתמשים במנגנון ה page fault שנתמך בGPUs החדשים של NVIDIA. מכיוון שגישה זו מצריכה אך ורק שינויים בתוך מערכת ההפעלה והדרייבר של ה-GPU, נוכל לממש את הפונקציונאליות הרצויה מבלי שנצטרך לעדכן את תוכניות הGPU הקיימות.

עבודה זו מציגה את החידושים הבאים:

- אנו מאפיינים את התקורה של false-sharing בשיתוף זיכרון בין CPU ל GPU במערכות הקיימות (סעיף 4.1). אנו מציעים מטמון דפים אחיד אשר מתגבר על בעיה זו לחלוטין ע"י שימוש ב lazy release consistency model [19, 7].
- אנו מרחיבים את מטמון הדפים של מערכת ההפעלה כדי שיכיל את הזיכרון הכולל של המערכת (CPU ו- GPU) וממשים אותו (סעיף 5.1). אנו מציגים מנגנון peer-caching ומשלבים אותו בתוך ה readahead prefetcher של מערכת ההפעלה, וע"י כך מאפשרים לכל מעבד להביא את המידע המבוקש מהמיקום הטוב ביותר, בפרט, מזיכרון הGPU (סעיף 7.2.1).
- המימוש שלנו הוא כללי עבור מערכות הפעלה Linux ואינו מובנה עבור שום GPU ספציפי. אב הטיפוס שלנו בנוי עבור Pascal GPU של חברת NVIDIA ע"י כך ששינינו את החלקים של הדרייבר שהם open source ובעזרת אמולציה עבור חלקים שהם closed-source.
- אנו מודדים את הביצועים של GAIA באמצעות אפליקציות אמיתיות כגון (1) סיפריה לעיבוד גרפים בעזרת GPUs – Gunrock [35], (2) תוכנית ליצירת פסיפס מתמונה בעזרת בסיס נתונים גדול של תמונות [32], ו-(3) multi-GPU image stitching [13, 15] המחליט על הדרך האופטימאלית לשילוב תמונות קטנות המרכיבות תמונה אחת גדולה. תוכנות אלו מדגימות את האפקטיביות ונוחות השימוש בGAIA.

תקציר

GPUs עברו כברת דרך ארוכה ממאיצי-פונקציות ספציפיות למעבדים כללים בעלי ביצועים גבוהים, הניתנים לתכנות מלא. עם זאת, האינטגרציה בינם לבין מערכת ההפעלה עדיין מוגבלת למדי. בפרט, הזיכרון הפיזי של הGPU, אשר כיום עשוי להגיע עד כ 32GB [6], עדיין מנוהל על ידי הGPU דרייבר, ללא שום התערבות מצד מערכת ההפעלה. אחת ההשלכות הקריטיות של תכנון מערכת בצורה זו היא שמערכת ההפעלה לא יכולה לספק שירותי ליבה לתוכנות GPU, כגון גישה יעילה לקבצים ע"י מיפוי זיכרון וירטואלי, ולא יכולה לייעל את ביצועי I/O עבור תוכנות CPU אשר משתפות קבצים עם הGPU. כדי להתגבר על מגבלות אלו, דרושה אינטגרציה יותר חזקה של זיכרון הGPU לתוך המטמון הדפים (page cache) ומנגנוני I/O של מערכת ההפעלה. אינטגרציה זו היא אחת המטרות העיקריות של עבודה להלן.

מחקרים קודמים הראו כי מיפוי קבצים לזיכרון הGPU כדאיי מכמה היבטים [31, 33, 32]. הוא מאפשר מודל תכנותי אינטואיטיבי המבוסס על מצביעים בגישה לקבצים, משפר ביצועים של תוכנות GPU שעובדות עם קבצים ע"י אופטימיזיות של מערכת ההפעלה ששקופות למפתח כגון data prefetching, ולבסוף, מודל זה הוא נח ואינטואיטיבי לשיתוף מידע בין תוכנות CPU לתוכנות GPU ועבור דפוסי גישה לקבצים התלויים בפעולת החישוב המבוצעת.

הרחבת מטמון הדפים של מערכת ההפעלה כך שיכיל גם את זיכרון הGPU כדאיי אפילו עבור תהליכי CPU שעובדים עם קבצים. שרתים המכילים עד כ8 GPUs נהים יותר ויותר נפוצים ולכן סך כל זיכרון הGPU הזמין (100-200GB) הוא גדול מספיק בכדי לשקול להשתמש בו עבור המערכת כולה, למשל, עבור מטמון הדפים של מערכת ההפעלה. אנו מראים אמפירית, כי גישה לזיכרון הGPU בשביל, במקום גישה ל SSD, עשויה להאיץ את ביצוי I/O עד פי 3 (ראה פרק 7). לבסוף, אם מערכת ההפעלה תנהל את מטמון הדפים גם בזיכרון הGPU נאפשר לתוכנות GPU לקרוא את הקובץ המבוקש ישר מתוך זיכרון הGPU, בלי הצורך לזהם את זיכרון הCPU במידע לא נחוץ [12].

לצערינו מערכות ההפעלה כיום לא מאפשרות אינטגרציה של זיכרון הGPU לתוך מטמון הדפים של מערכת ההפעלה. הכותבים של ActivePointers [31] איפשרו מיפוי קבצים לתוך הזיכרון של הGPU אולם הדבר הצריך שיכתוב גלובאלי של תוכנות הGPU הקיימות ובשל כך בלתי אפשרי לשימוש עבור ספריות סגורות כגון cuBLAS [5]. ה- Unified Virtual Memory (UVM) של NVIDIA וה-Hetrogeneous Memory Management (HMM) של לינוקס, מאפשרים לCPU ולGPU לשתף זיכרון וירטואלי. אולם אף אחד מהם לא מאפשר למפות קבצים לתוך זיכרון הGPU, מה שהופך שתי גישות אלו ללא יעילות בעבודה עם קבצים גדולים (ראה סעיף 7.3.3). מה שחשוב יותר הוא שUVM ו-HMM לא מאפשרים לדף פיזי להיות ממופא ביותר מזיכרון אחד בכל רגע נתון, מה שגורם לפגיעה משמעותית בביצועים של תוכניות המשתפות קובץ לכתיבה בגלל תופעה של false sharing. יתר על כן, ל-false sharing יש השפעה דרסטית על המערכת כולה, כפי שאנו מראים בפרק 4.

המחקר בוצע בפקולטה למדעי המחשב בטכניון בהנחייתו של פרופ. מרק זילברשטיין (מהפקולטה להנדסת חשמל).

אני מודה לטכניון על התמיכה הנדיבה במשך השתלמותי.

מטמון דפים אחיד למערכות הטרוגניות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

טניה ברוכמן

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אדר התשע״ט חיפה מרץ 2019

מטמון דפים אחיד למערכות הטרוגניות

טניה ברוכמן