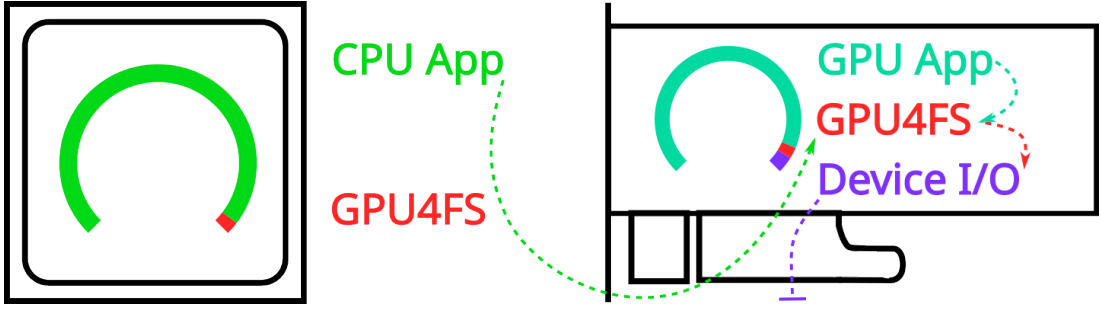
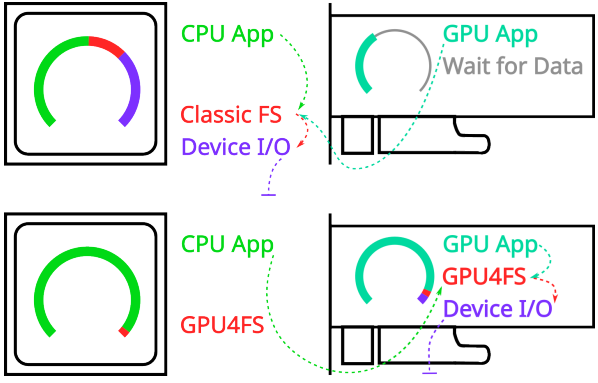
Karlsruhe Institute of Technology

Full-Scale File System Acceleration on GPU

Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, Lukas Werling, Yussuf Khalil, Thorsten Gröninger, Frank Bellosa | March 15, 2024



File System on GPU



Contemporary GPU Data Path

- GPU applications need data from storage
- Request send to and handled by CPU
- Silberstein et al.: file system calls on GPU
- Request handled on CPU

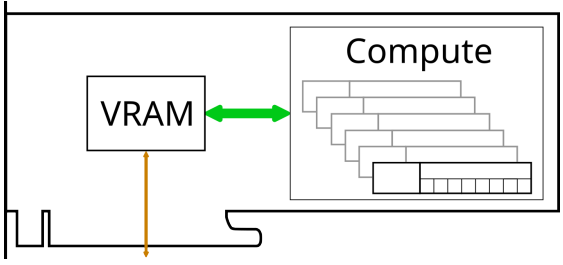
Proposed: GPU4FS

- GPU4FS: file system calls + handling on GPU
- Lower latency for applications on GPU
- More CPU time for applications on CPU

File Systems

- Persistent data store
- Nested folder organization
- Map blocks on drive to files, folders, ...
- Well-understood interface
- Ubiquitous usage in multiple programming languages
- EXT4, BTRFS, ZFS, Nova, WineFS, XFS, ...
- Common acceleration strategies: lookup tables, trees

GPUs



- Massively parallel vector processor
- More compute than CPU

- Dedicated memory area
- Relatively slow interconnect

⇒ Rethink FS datapath, caches

- Bandwidth-optimized, not for pointer-chasing
- Optimized for branchless code

⇒ Indirect loads, esp. trees, difficult

⇒ Rethink full FS structure

Full-Scale File System Acceleration on GPU

Full-Scale File System

Well-understood High-level Interface

- Widespread support
- Developer familiarity

Data Integrity

- RAID
- Checksums
- Crash Consistency

Full-Scale Acceleration on GPU

Latency Reduction

- GPU-App to FS
- Inner-FS

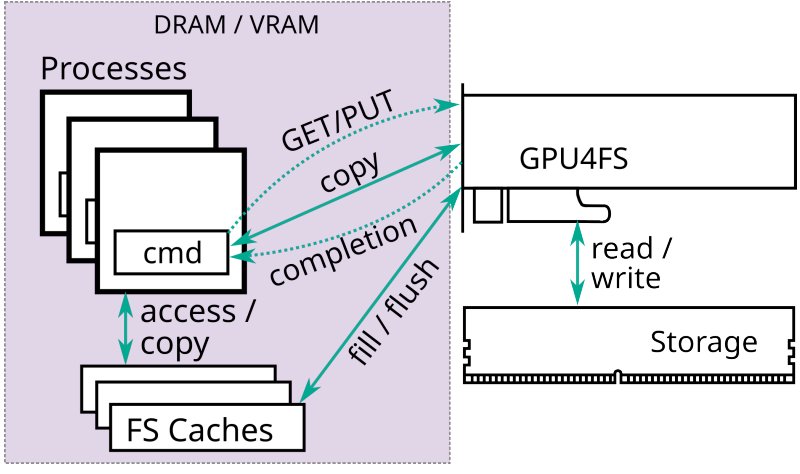
Consistency

- GPU controls progress, completion
- Parallel journaling, CoW, garbage collection

Performance

- Exploit small-scale parallelism
- Allow expensive FS features

Overall Design



Contemporary File System Interface

POSIX

```
fd = open("talk.pdf");
fstat(fd, &st);
buf = malloc(st.st_size);
read(fd, buf, st.st_size);
close(fd);
```

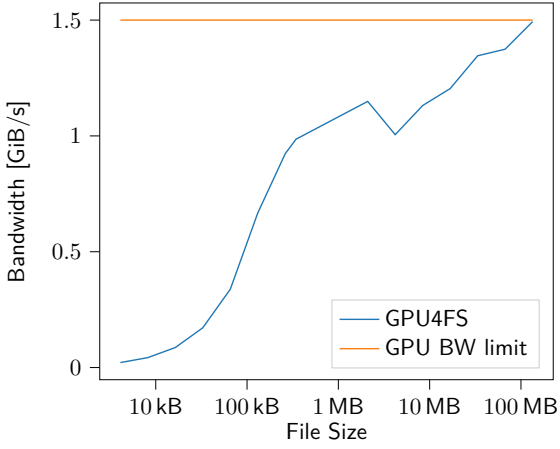
- 👎 Multiple syscalls
- 👎 Pass through Virtual File System (VFS)
- 👎 Racy
- 👎 Not what defines FS

GPU4FS Primary Interface

```
Area result = GET("talk.pdf");
```

- 👍 Read data, metadata atomically
- 👍 Allocate while loading
- 👍 Write data in shared area
- 👍 Freedom to improve on GPU:
  - No legacy code
  - Different requirements

Preliminary Results: No Inherent Bandwidth Limit

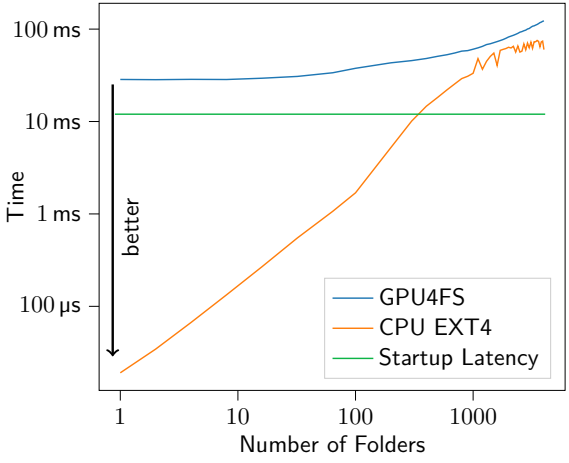


- Question: FS on GPU *inherently* slow?
- Measured: Minimal FS vs max raw GPU access bandwidth
- 1 file written to folder

👎Small files slow ⇒ startup latency

👍Max GPU bandwidth achievable

Interface: Deep Directory Creation



- Implement `mkdir -p a/b/c/d/e/…/zzz`
- Single command on GPU
- Repeated `mkdirat()` in POSIX

👎 Large startup latency

👎 Slower than CPU

👍 Small runtime increase per directory

👍 Comes close to CPU for deep directories

Future Work

Consistency

- 👎 Latency high
- 👎 GPU drivers less stable
- 👎 More application and FS parallelism
  - Global write barriers slow
  - Local `fsync()`/`msync()` meaningless

Disk Allocation / Garbage Collection

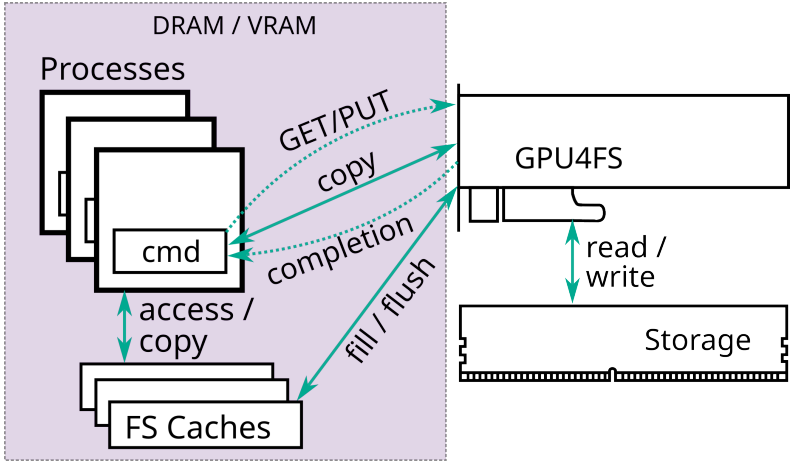
- 👎 Widely different allocation sizes => pre-partitioning difficult
- 👎 Allocation sizes unknown at config time
- 👍 Large allocations infrequent

- Garbage collection as background task
- Consistency implementation informs GC

Goals

- Exploit parallelism
- Reduce latency

Run your FS on GPU!



- GPU latency reduction
- CPU time reduction
- New interface
- Results say: Lets try!