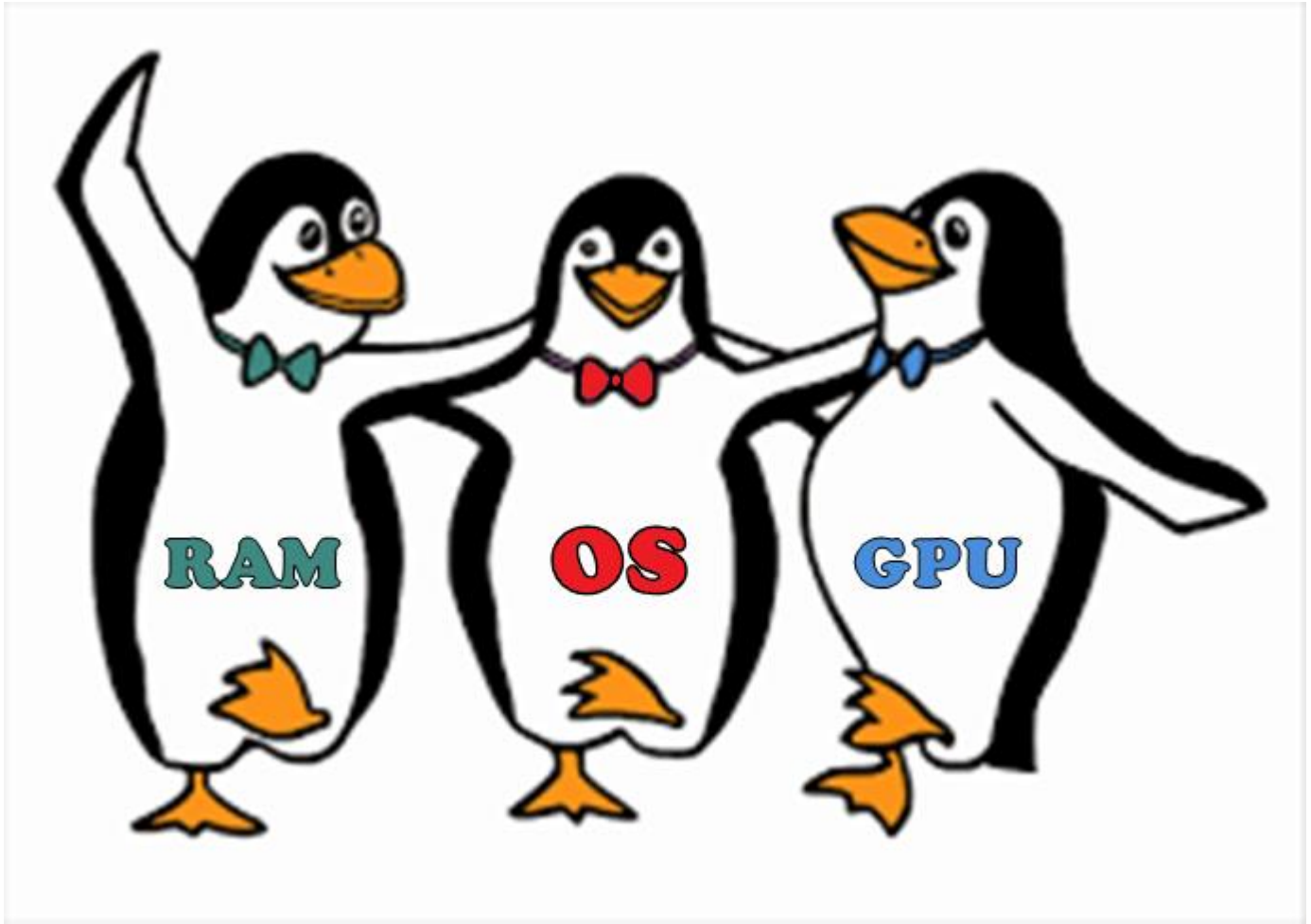


GAIA

Glob**A**l un**I**fied p**A**ge cache for Heterogeneous systems

Tanya Brokhman

Pavel Lifshits

Mark Silberstein

Technion, Israel Institute of Technology

ACSL
Technion Accelerated Computing Systems Lab

THE ANDREW & ERNA VITERBI FACULTY OF ELECTRICAL ENGINEERING

TECHNION
Israel Institute of Technology

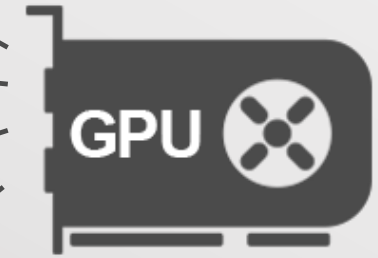
GPUs – its been a long journey…

Fully-programmable

High-performance

Large physical memory

Support for demand paging

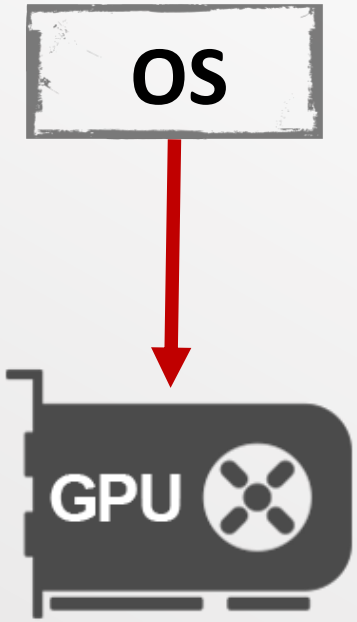


Managed entirely by the GPU driver

No integration with OS memory management

Core services that require OS management of GPU memory

- Access to memory mapped files by GPUs
- Disciplined inter-GPU/CPU file sharing
- Optimized CPU I/O by using GPU memory
- Seamless support for peer-to-peer access to storage from GPU

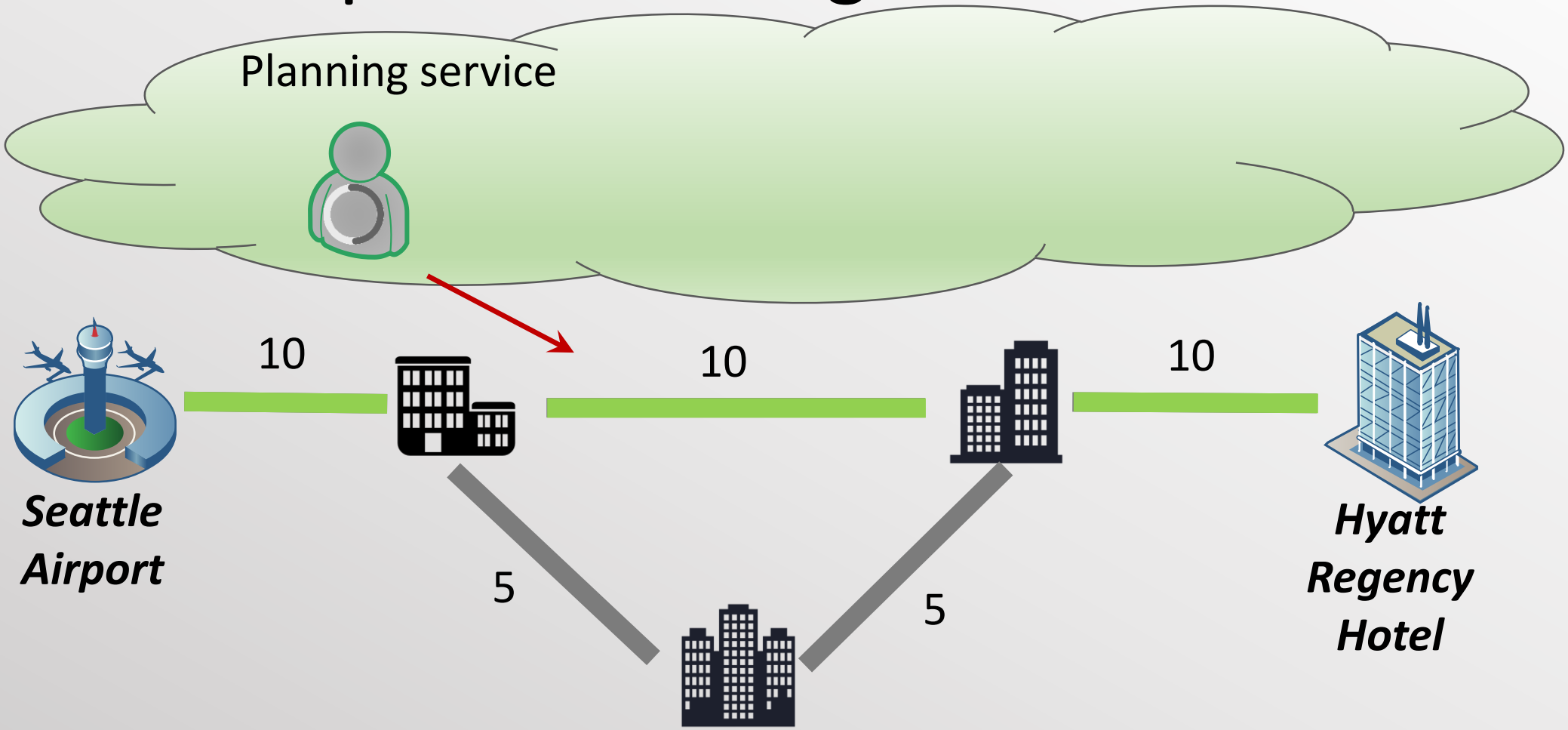


Why do we even need this?

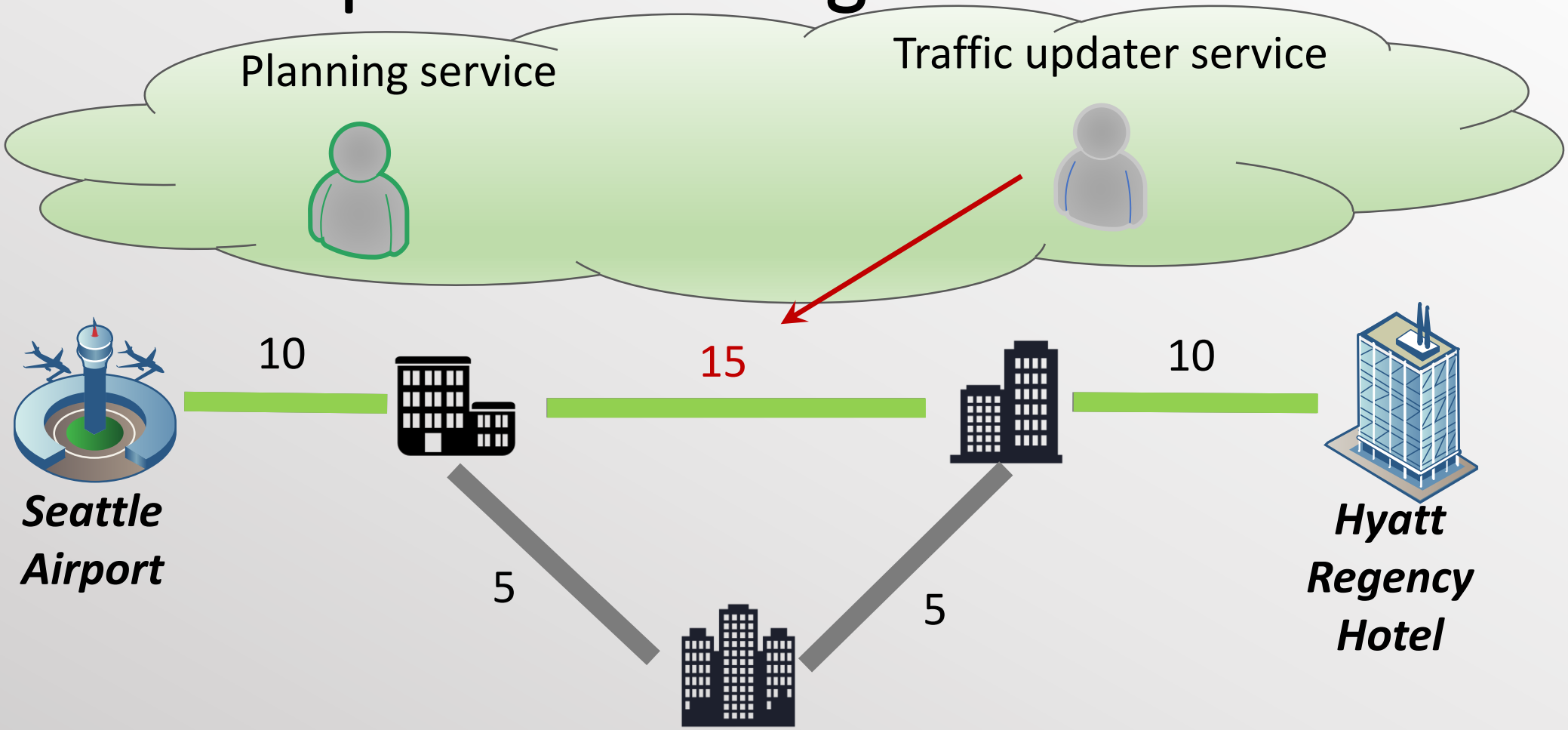
Example: Road navigation service

Road navigation server

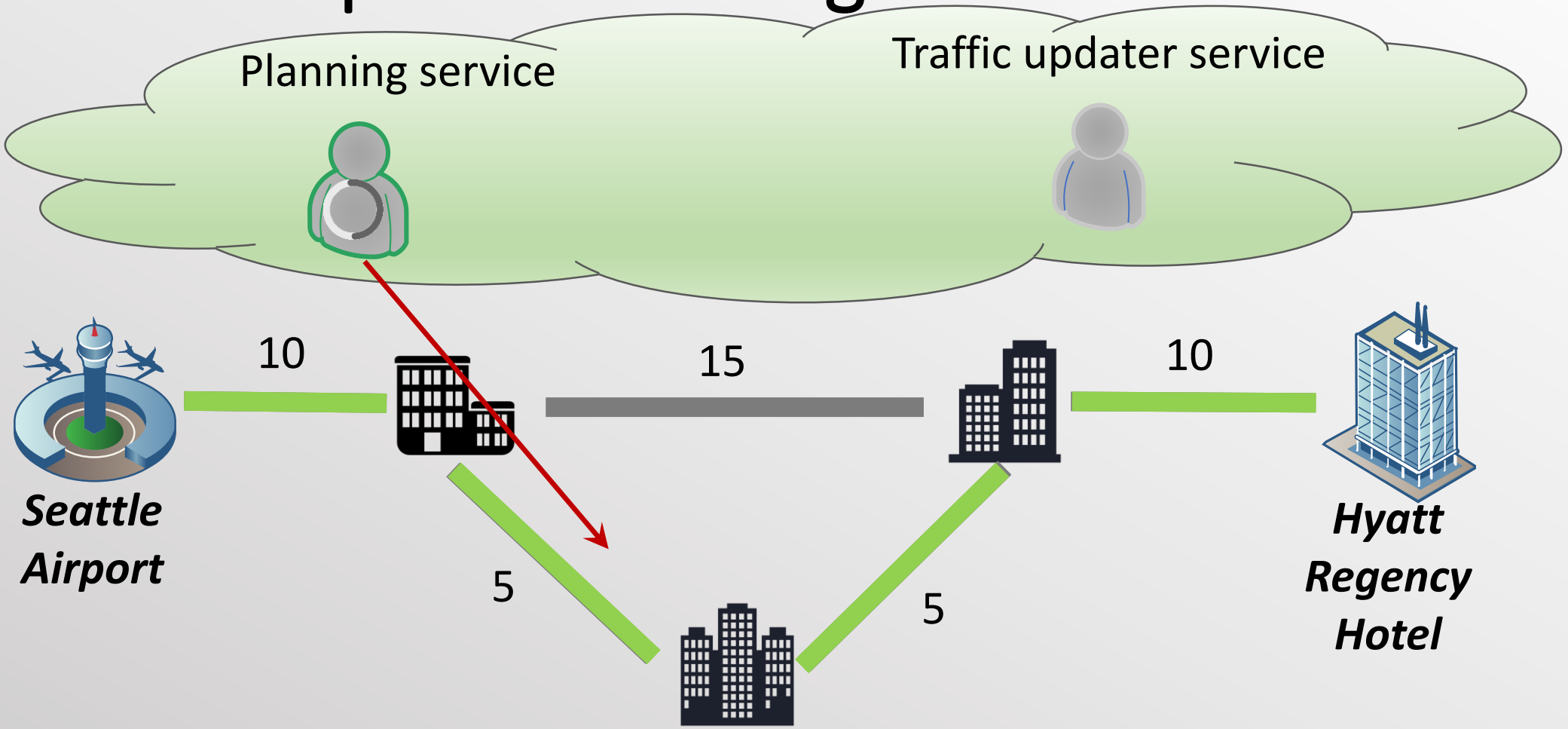
Example: Road navigation service



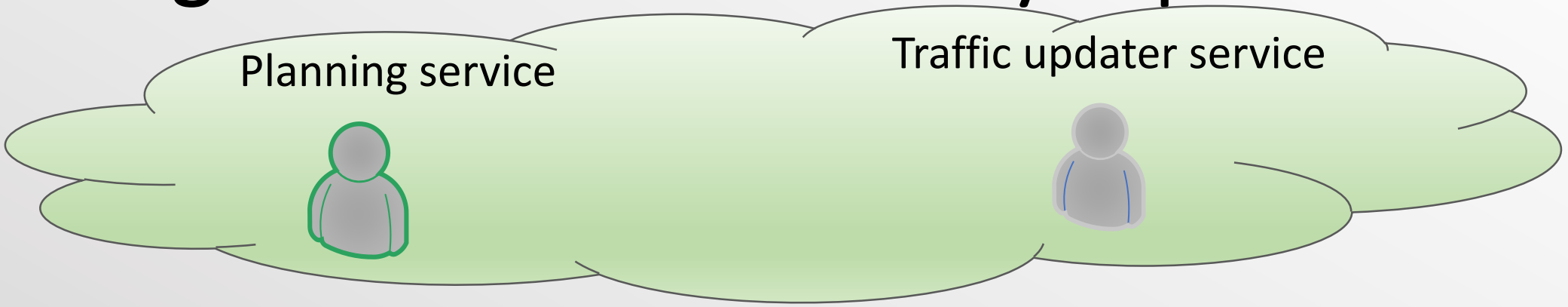
Example: Road navigation service



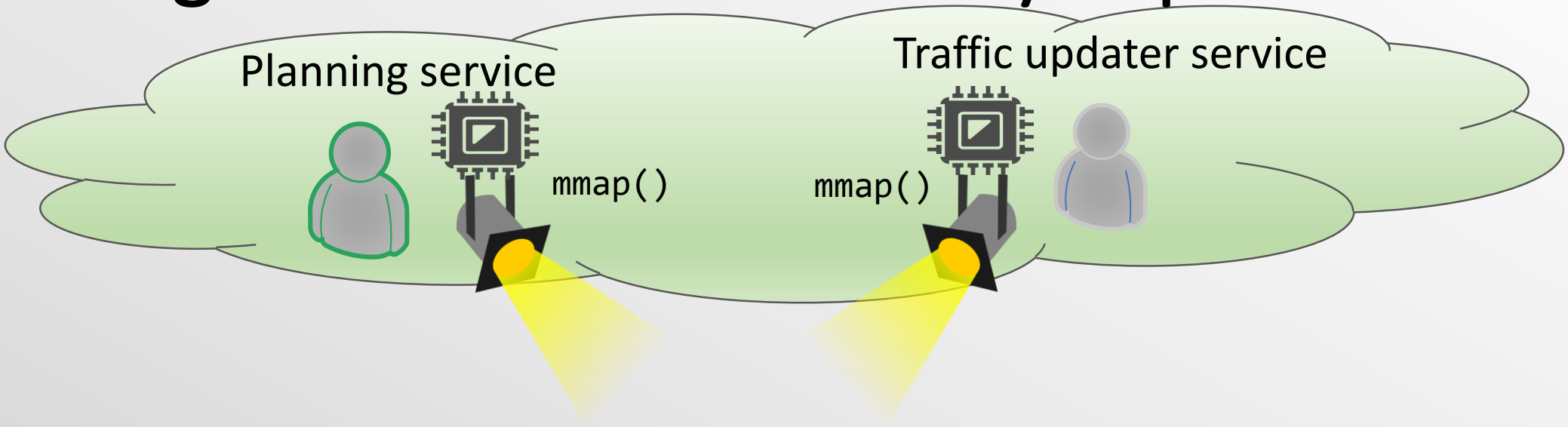
Example: Road navigation service



Road navigation service – CPU only implementation



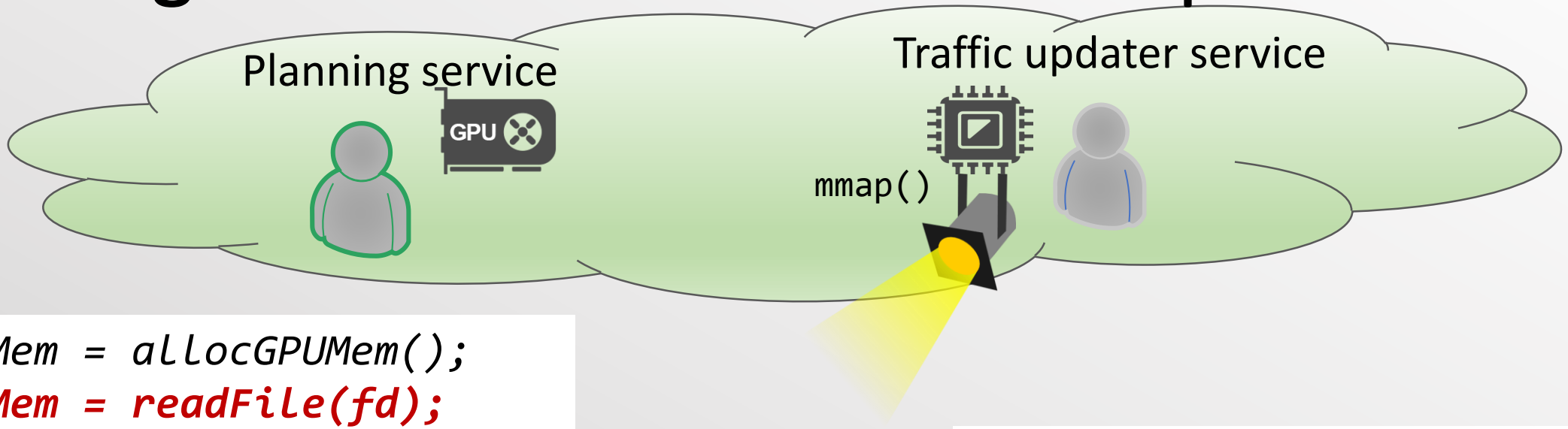
Road navigation service – CPU only implementation




```
mapData = mmap(map.d)
calc_route(mapData)
munmap(mapData)
```


```
mapData = mmap(map.d)
read_data_from_net()
for each road do:
    mapData[road] = newTime
munmap(mapData)
```

Road navigation service – CPU&GPU implementation

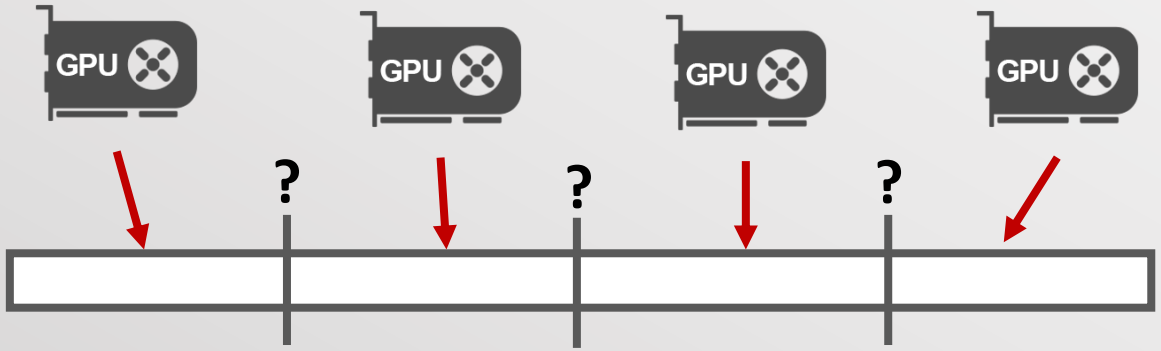



```
gpuMem = allocGPUMem();
cpuMem = readFile(fd);
copyToGPU(cpuMem, gpuMem);
<<calc_route>>(gpuMem);
copyFromGpu(newRoute)
```

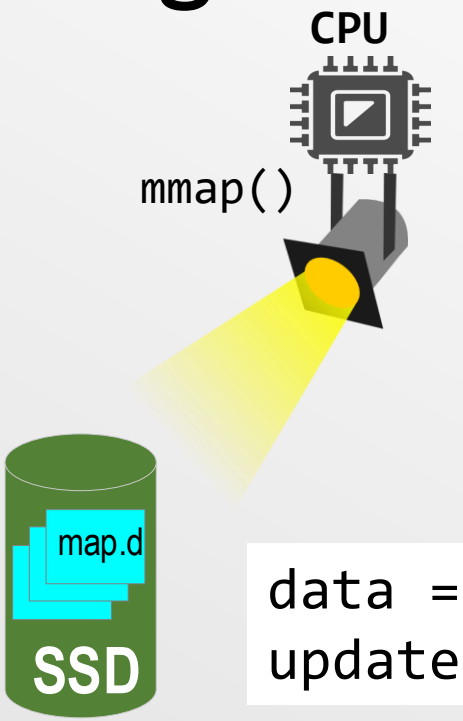
```
mapData = mmap(map.d)
read_data_from_net()
mapData[road] = newTime
munmap(mapData)
```

Whole file is copied!

CPU&GPU file sharing



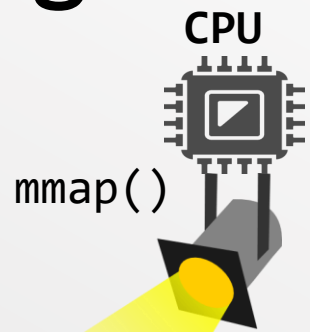

```
cpuMem = readFile(fd);
copyToGPU(cpuMem, gpuMem);
<<gpuKernel>>(gpuMem);
copyFromGpu(gpuMem, cpuMem)
writeFile(cpuMem)
```




```
data = mmap(map.d)
update(data)
```

Whole file is copied 4 times!

CPU&GPU file sharing

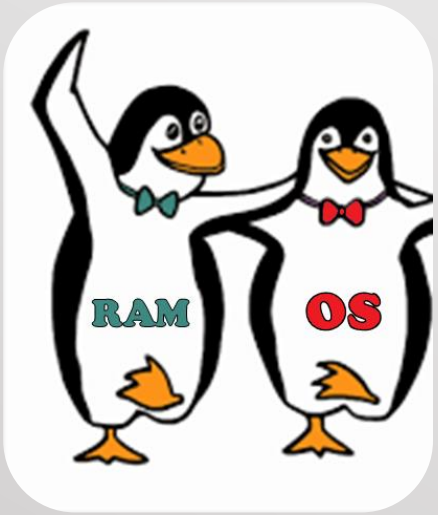


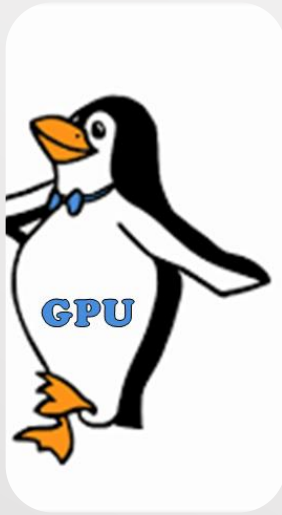
Without OS management:

- ✗ Data-dependent GPU accesses to files
- ✗ Efficient write-sharing between CPU and GPU
- ✗ Repeated copying of the file into GPU memory (no page cache in GPU)

```
data = mmap(map.d)
update(data)
```

Tighter integration of GPU memory into the OS page cache and file I/O mechanisms is required!

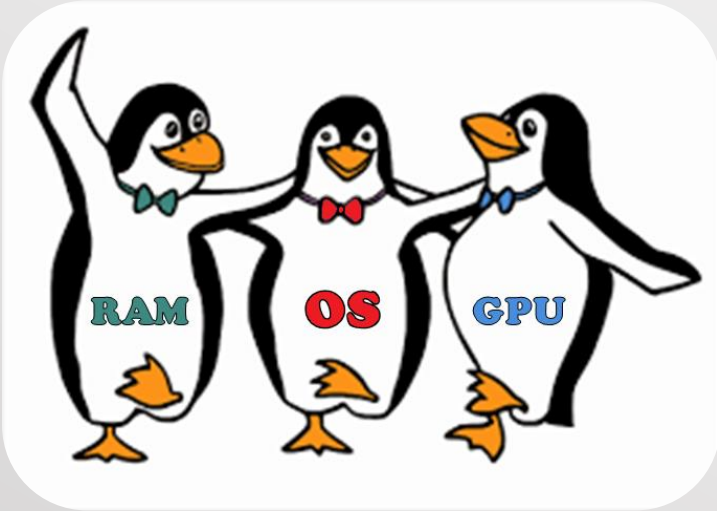




GAIA

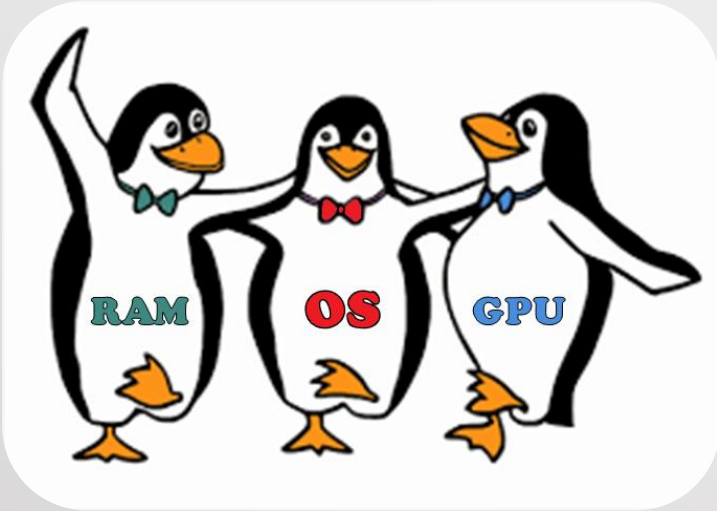
A distributed, weakly-consistent page cache architecture for heterogeneous multi-GPU systems

(GAIA = GlobAl unIfied pAge cache)



GAIA

- ✓mmap for GPU kernels
- ✓Efficient write-sharing between CPU and GPU
- ✓Enable CPU and GPU I/O optimizations



ACSL
Technion Accelerated Computing Systems Lab

Tanya Brokhman, USENIX ATC19

Challenges

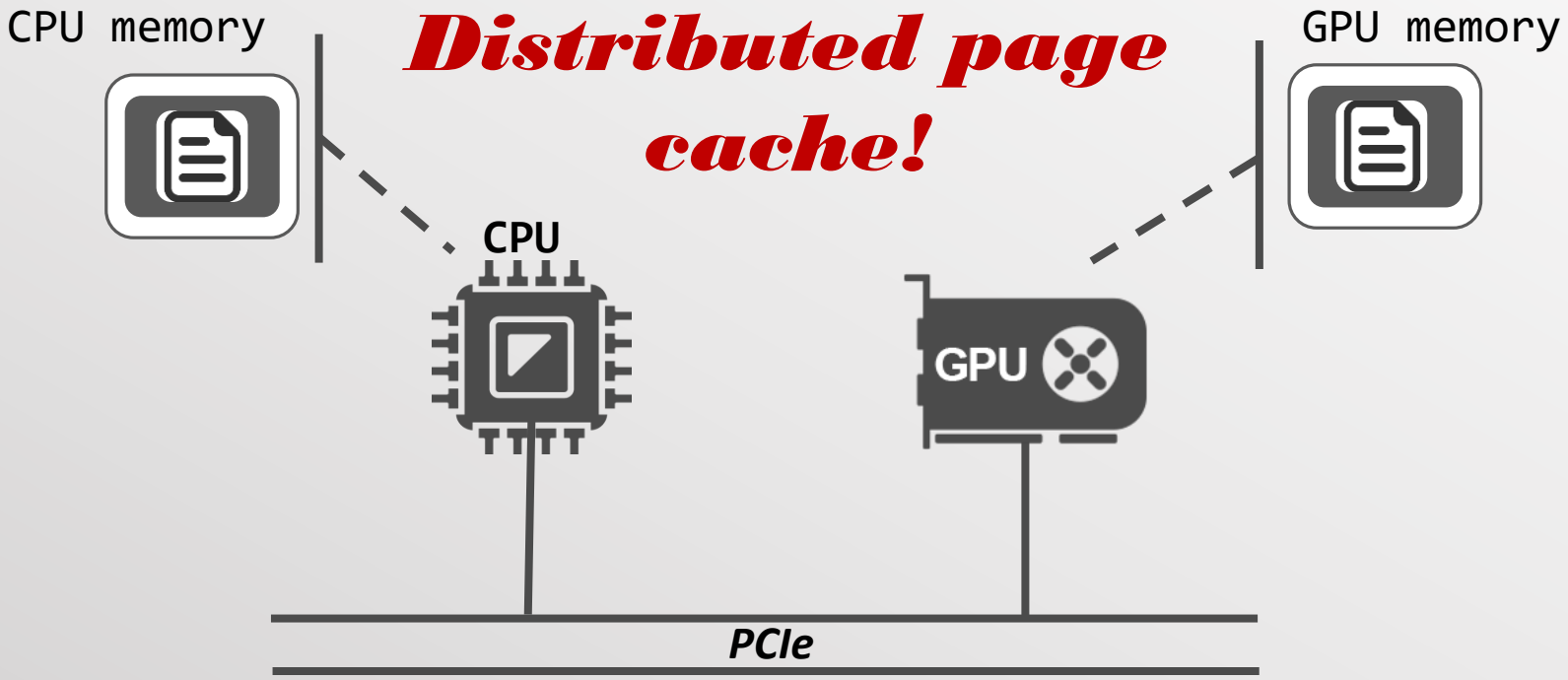
- File system consistency model
- Integration with GPU driver
- Integration with OS page cache and OS prefetcher

Challenges

- File system consistency model
- Integration with GPU driver
- Integration with OS page cache and OS prefetcher

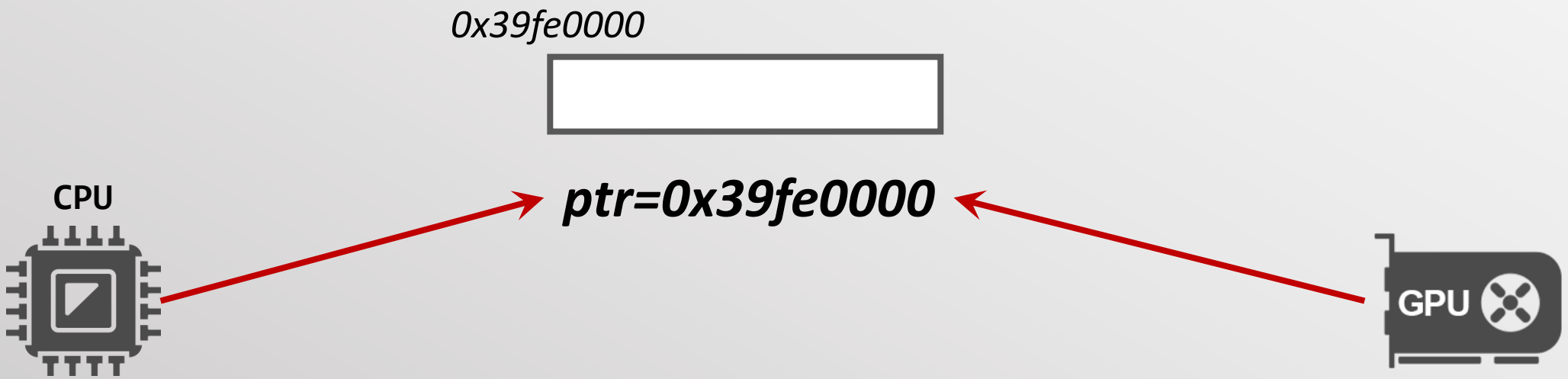
For details see our paper

Consistency model considerations



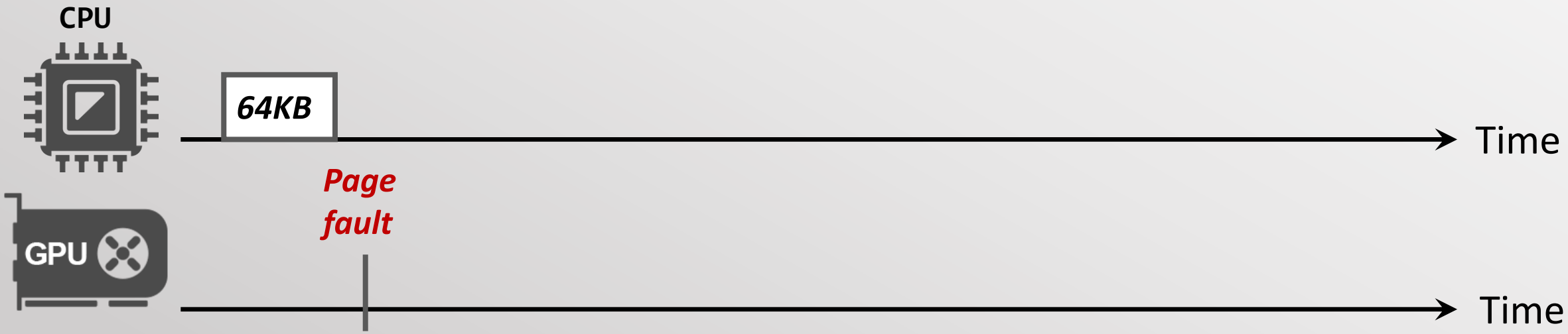
A bit of background: NVIDIA unified memory

- Single pointer accessible from CPU and GPU
- Data transfers occur transparently to programmer as part of page fault handling mechanism



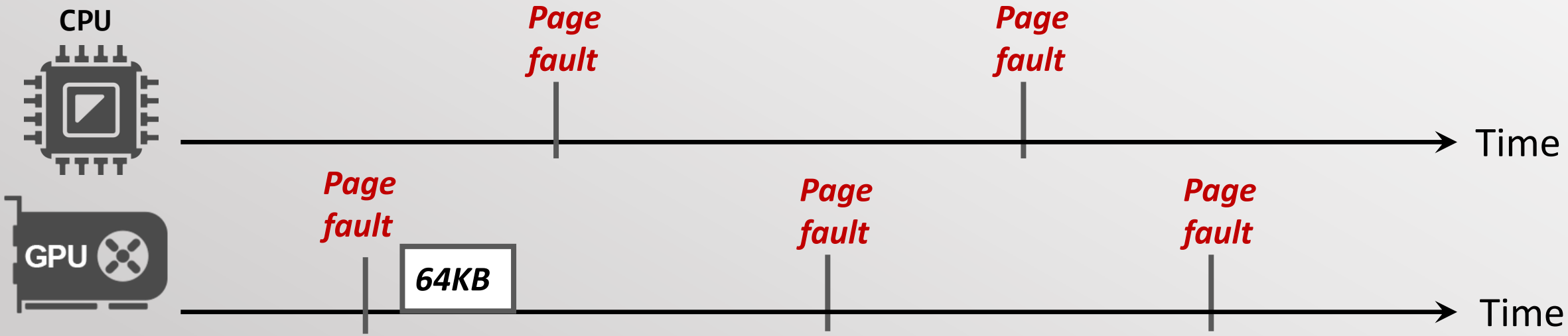
NVIDIA unified memory - Page-level strict coherence

- Strict coherence at the level of a GPU page (64KB = 16 X 4KB)
- A page can be mapped only by one processor
  - Accesses to the shared object are serialized



NVIDIA unified memory - Page-level strict coherence

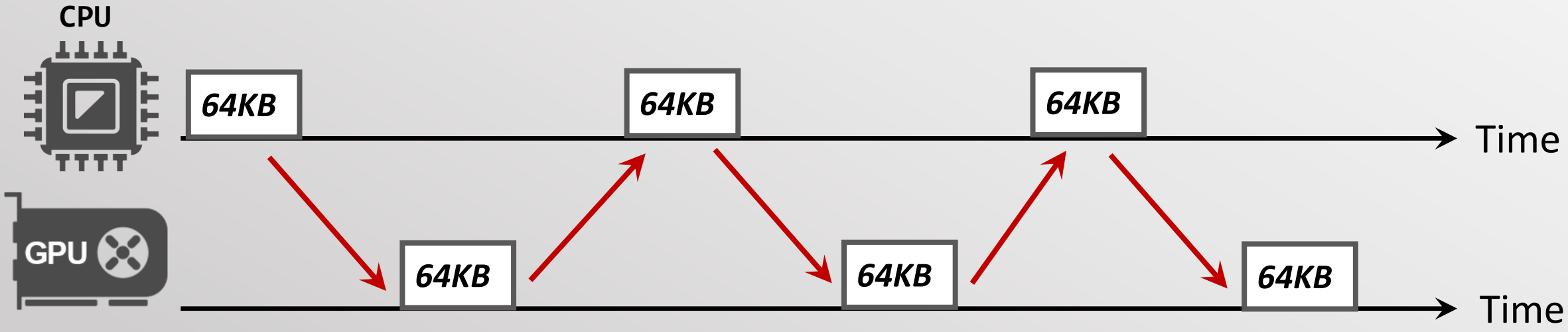
- Strict coherence at the level of a GPU page (64KB = 16 X 4KB)
- A page can be mapped only by one processor
  - Accesses to the shared object are serialized



NVIDIA unified memory - Page-level strict coherence

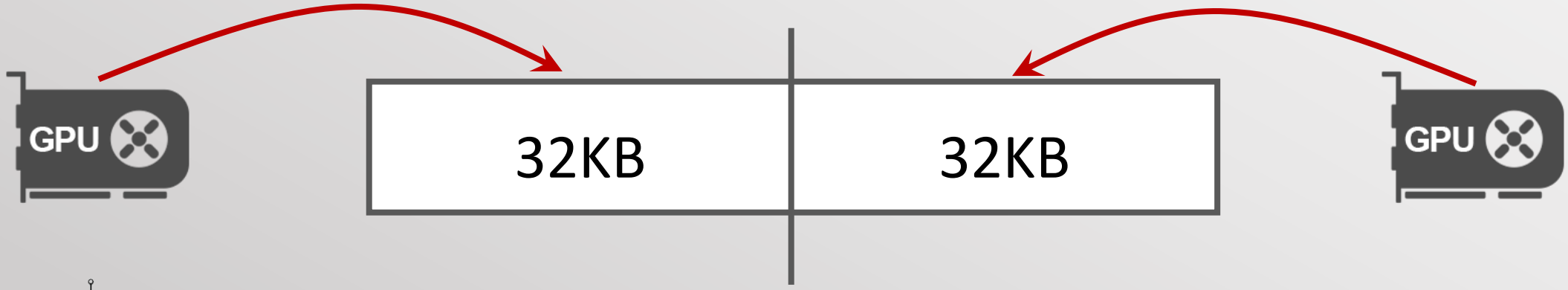
- Strict coherence at the level of a GPU page (64KB = 16 X 4KB)
- A page can be mapped only by one processor
  - Accesses to the shared object are serialized

Write-sharing causes multiple page migrations !

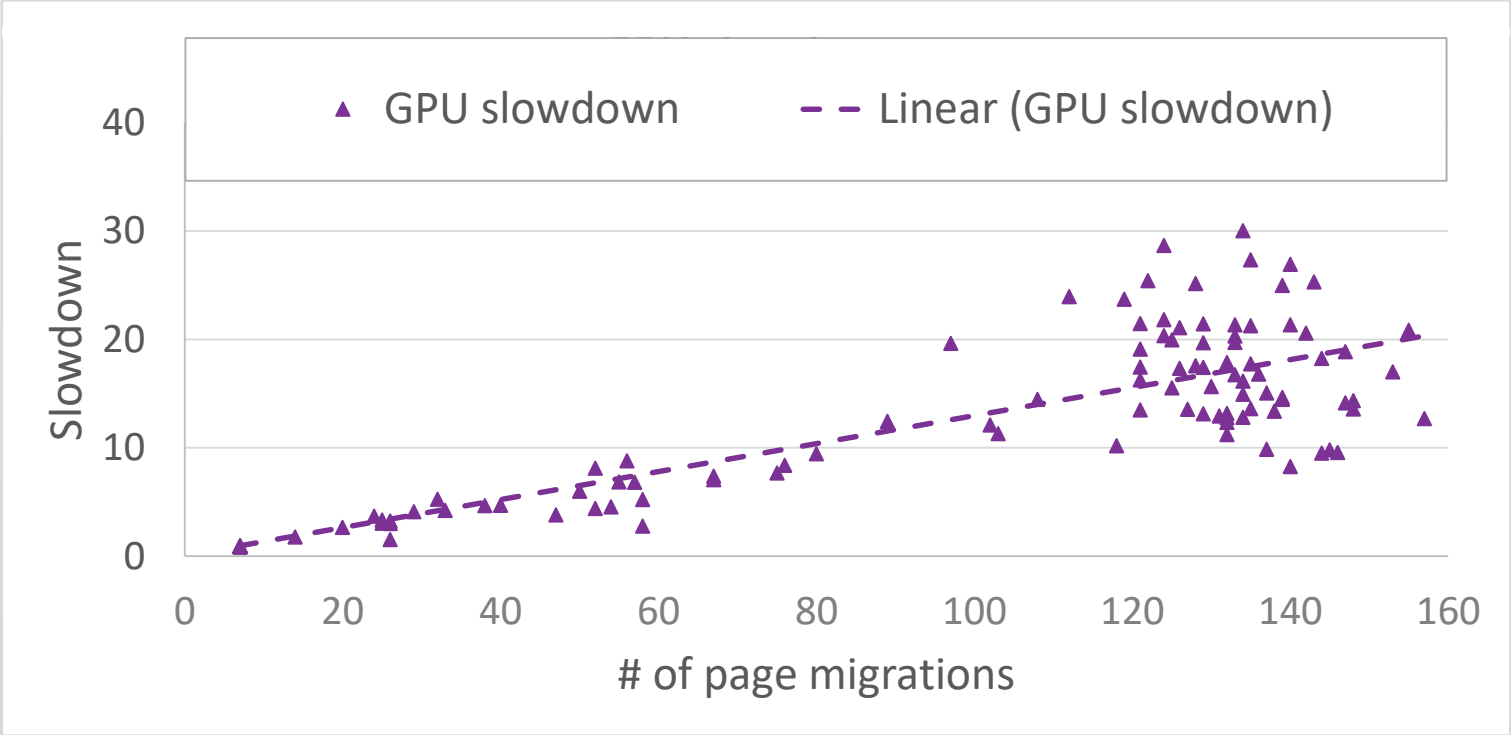


False sharing due to strict coherence

- Two NVIDIA GTX1080 GPUs
- 64KB-buffer (one GPU page) write-shared
  - Each GPU executes read-modify-write
  - Each GPU updates a different portion of the buffer
- Loop iteration per GPU varies

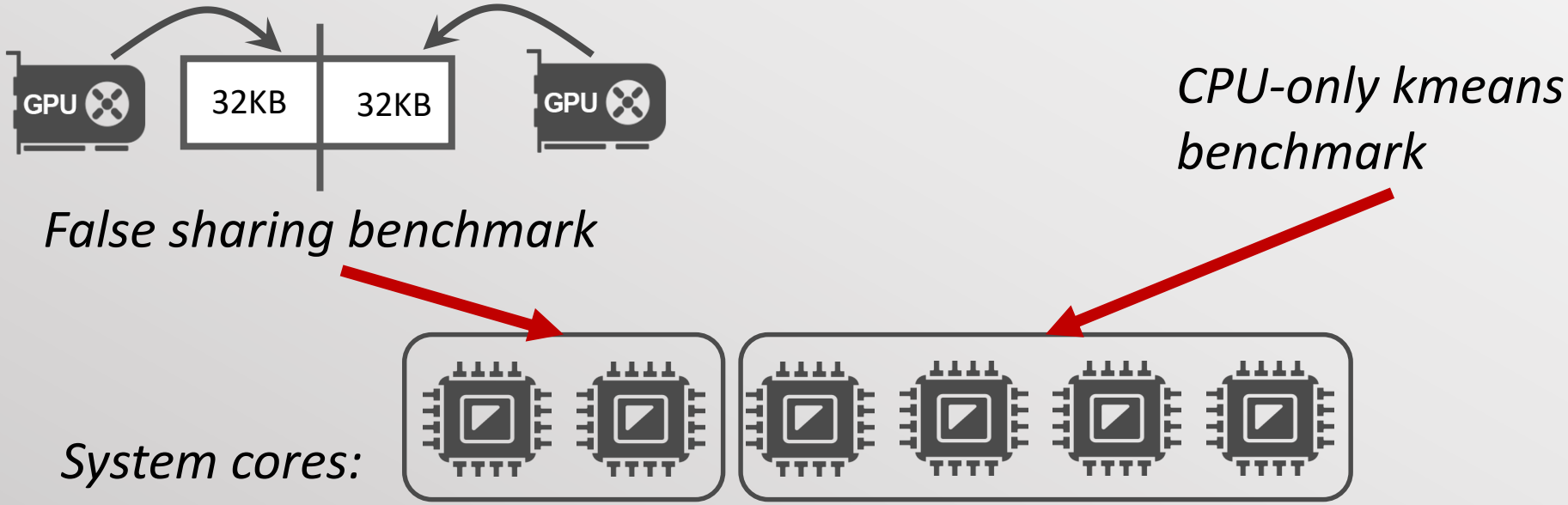


False sharing due to strict coherence

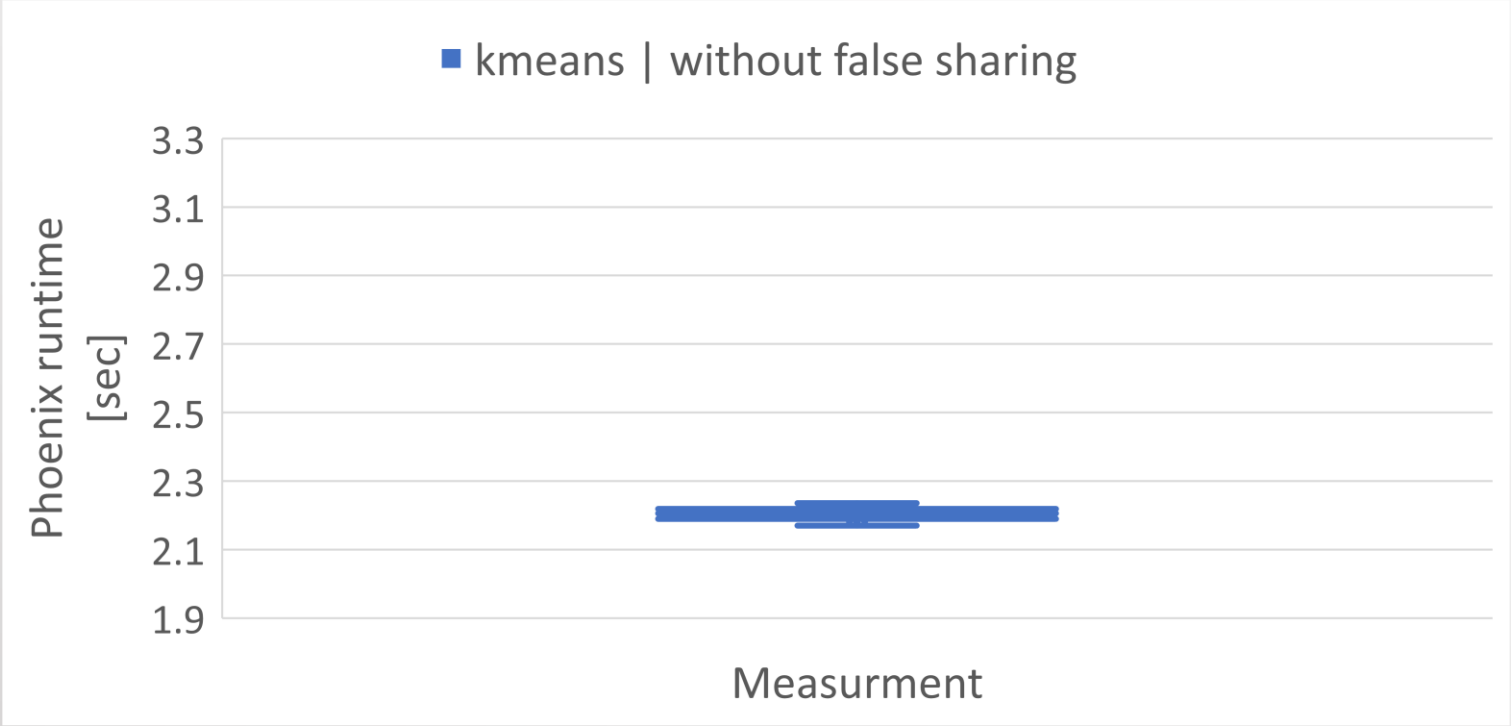


False sharing impact on the system

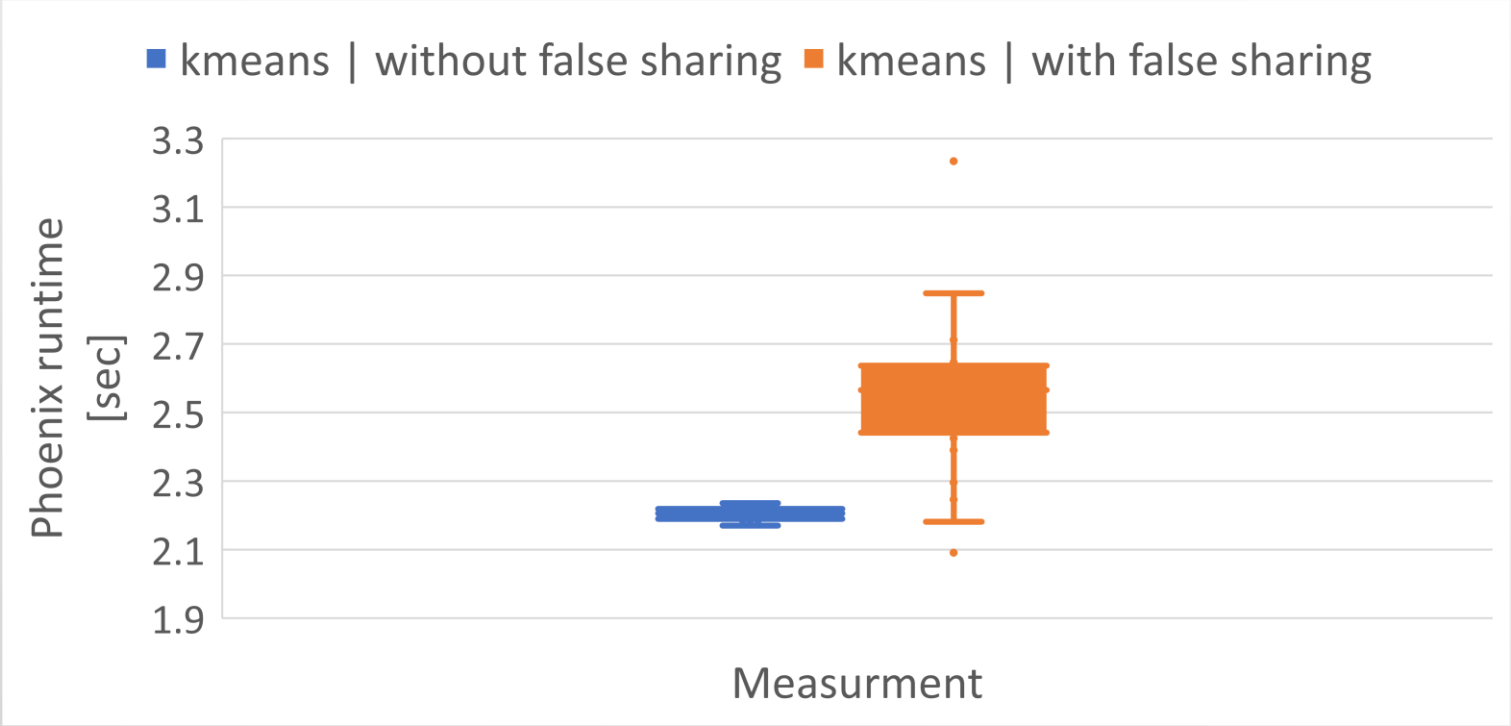
- CPU-only kmeans benchmark limited to specific cores
- False sharing benchmark running on remaining cores



False sharing impact on the system



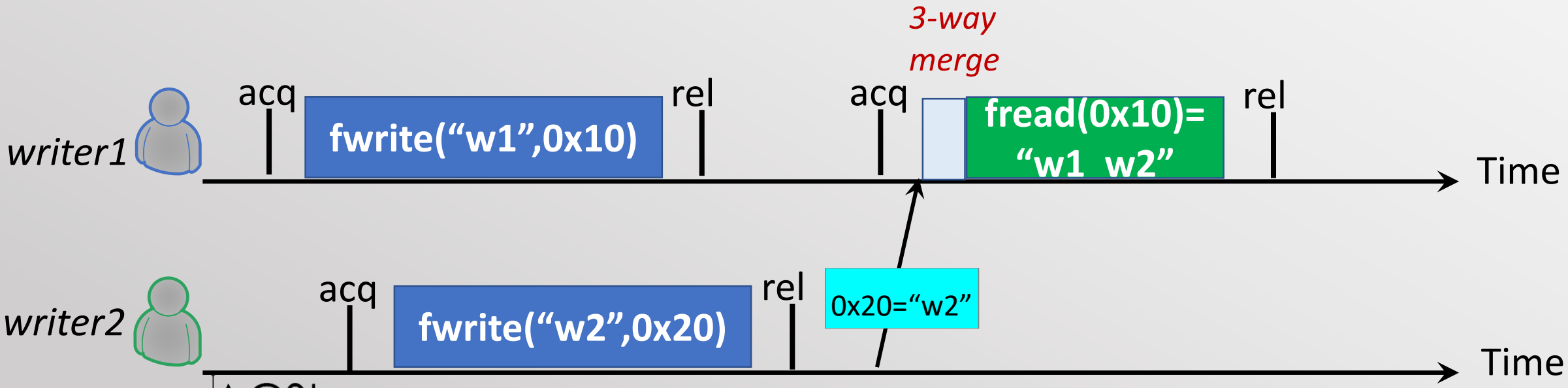
False sharing impact on the system



False sharing affects performance isolation!

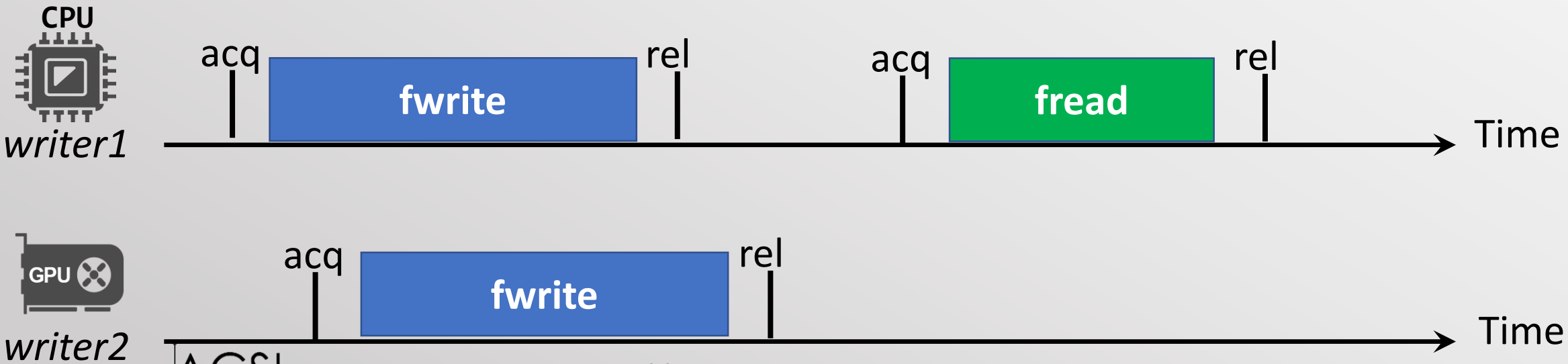
Lazy Release Consistency (LRC)

- Visibility of writes defined by *acquire* and *release* synchronization operations
  - The writes are visible after the writer *release*-s and the reader *acquire*-s
- The propagation of the updates is delayed until *acquire*



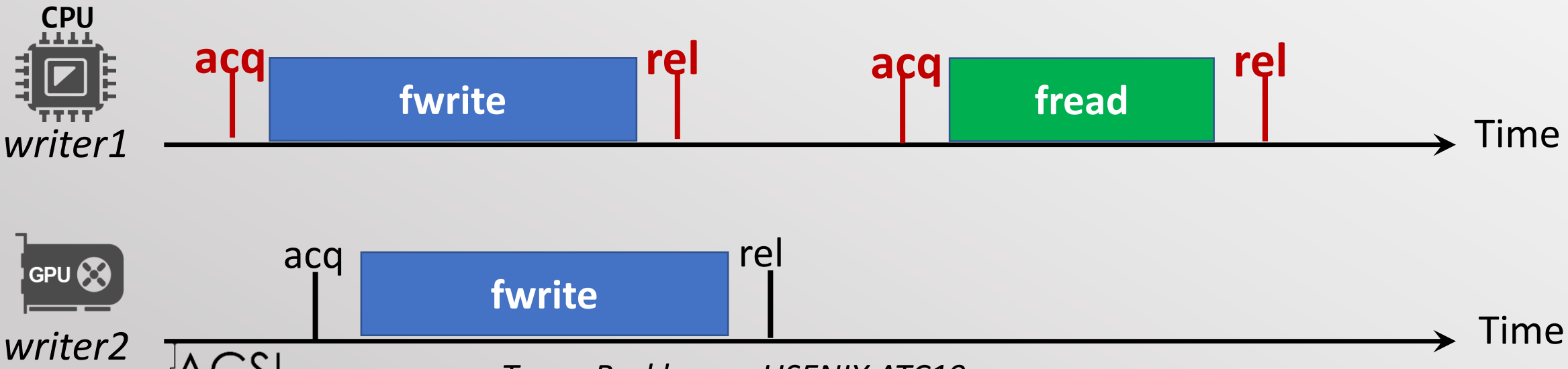
GAIA LRC

- Visibility of writes defined by *acquire* and *release* synchronization operations
  - The writes are visible after the writer *release*-s and the reader *acquire*-s
- The propagation of the updates is delayed until *acquire*



GAIA LRC

Incompatible with CPU legacy applications!



GAIA LRC – Design challenges

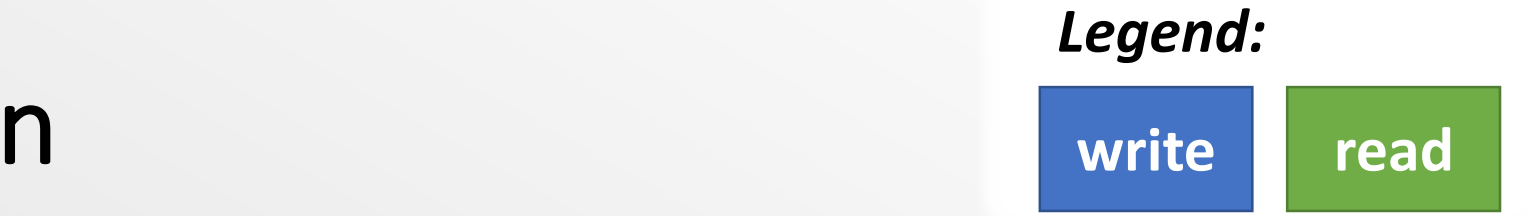
Support unmodified CPU legacy applications!

- CPU applications must be un-aware of the LRC page-cache

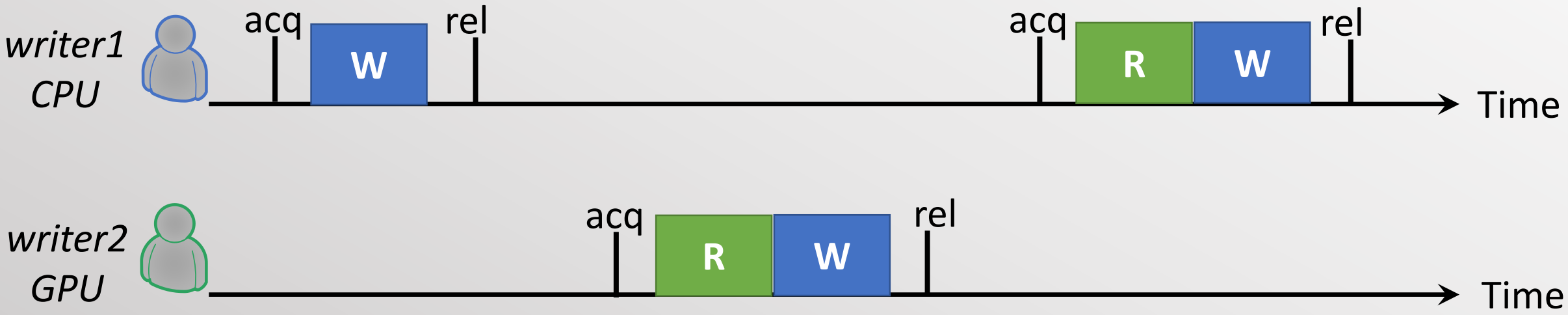
How?

- Limit the explicit consistency control to the CPU code **invoking GPU kernels**

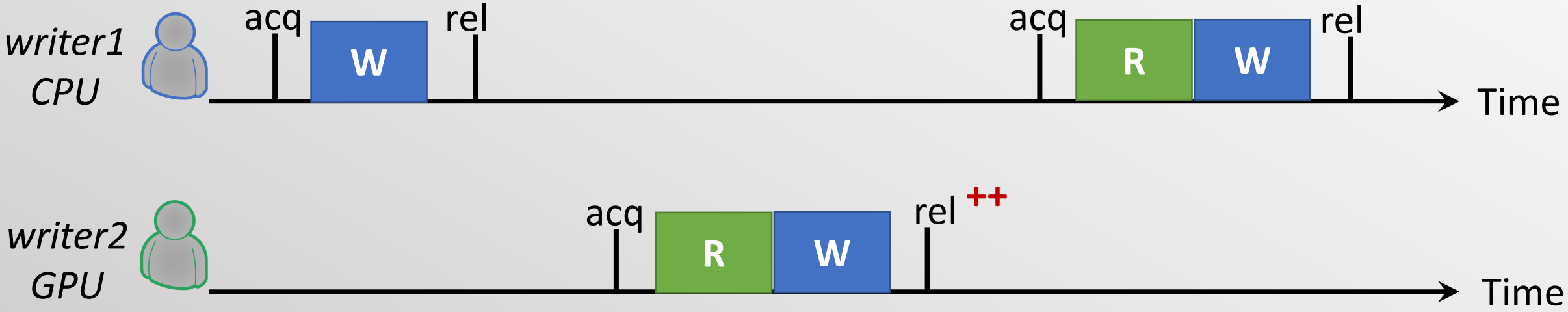
GAIA LRC - Design



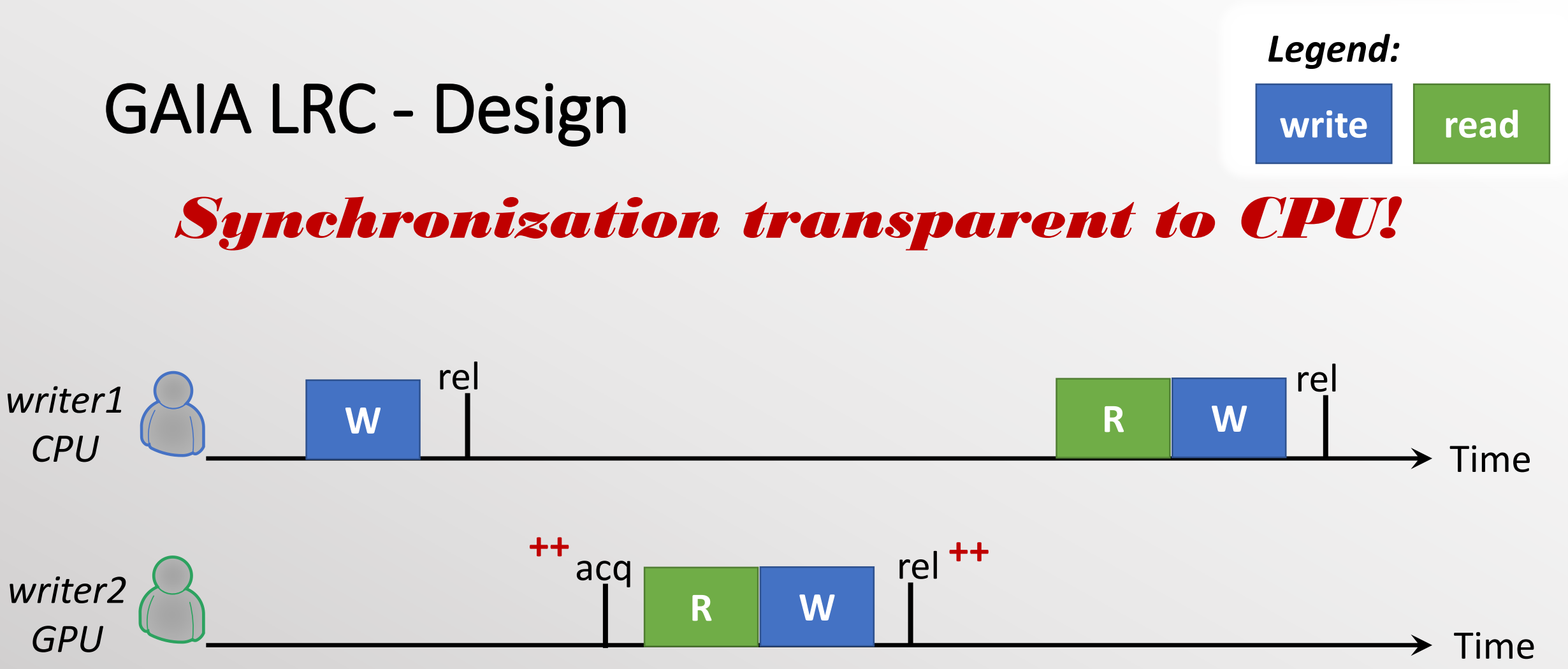
What do we want to happen on release/acquire?



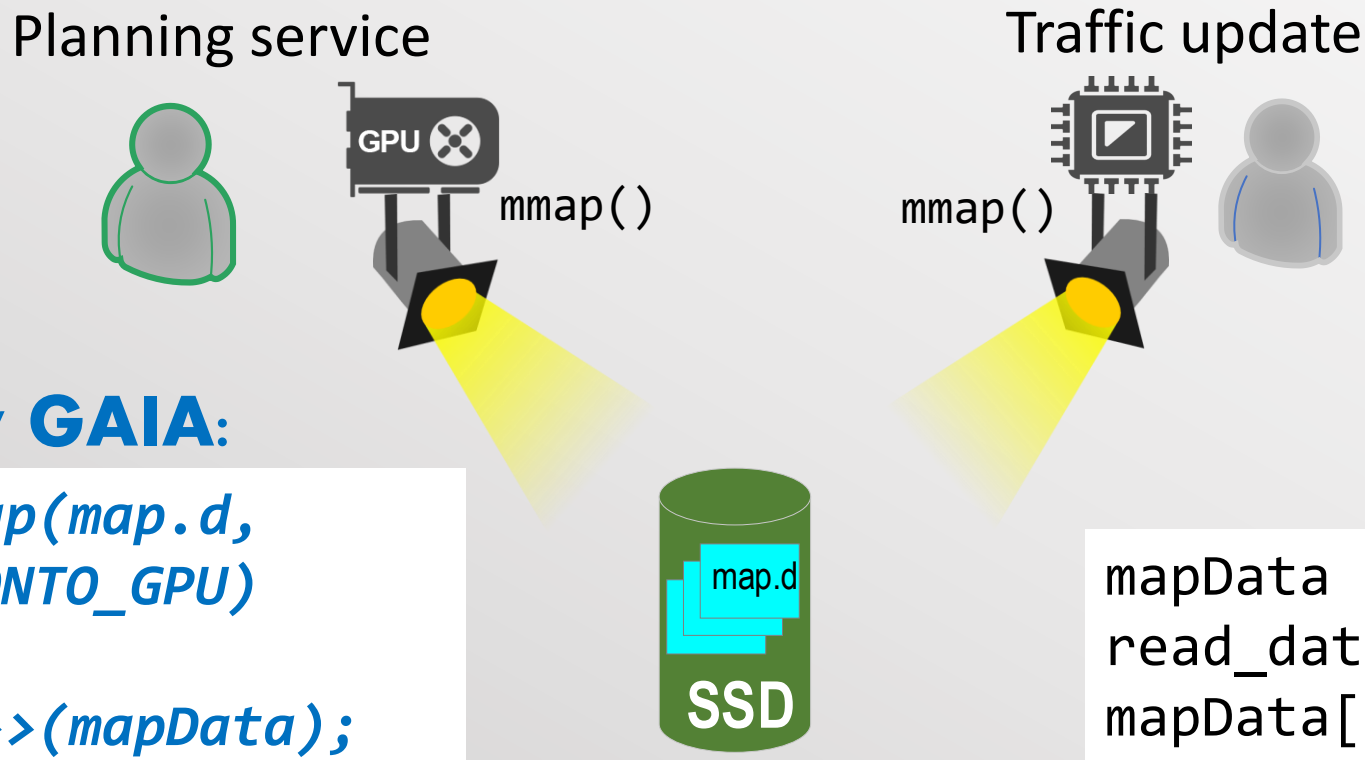
GAIA LRC - Design



GAIA LRC - Design



Putting it all together: Road navigation service – CPU&GPU implementation



w/ GAIA:

```
mapData = mmap(map.d,
               ONTO_GPU)
maquire()
<<calc_route>>(mapData);
mrelease()
munmap(mapData)
```

```
mapData = mmap(map.d)
read_data_from_net()
mapData[road] = newTime
munmap(mapData)
```

More details in the paper…

- Improved CPU and GPU I/O
  - Peer-caching
  - Readahead prefetcher support for GPU I/O
- Implementation details
  - Integration with OS
  - Integration with GPU

Evaluation

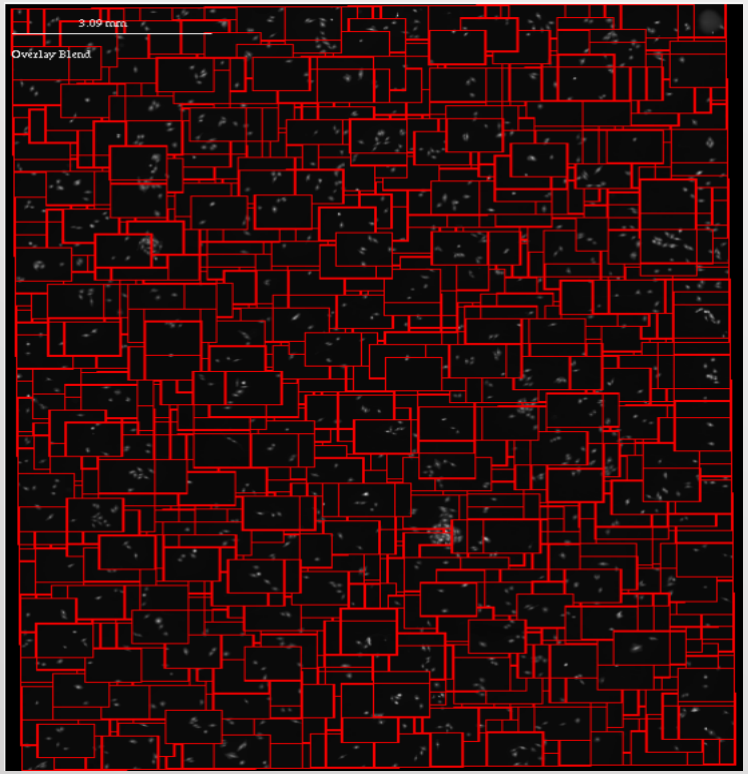
- Overhead analysis
  - Impact on CPU I/O
  - Memory overheads
- Microbenchmarks
  - Benefits of peer-caching
  - False sharing
  - Streaming read performance
- Applications
  - Performance of on-demand data I/O with image collage
  - Effects of false sharing in image stitching
  - Dynamic graph processing with Gunrock

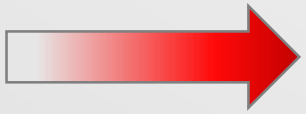
Evaluation

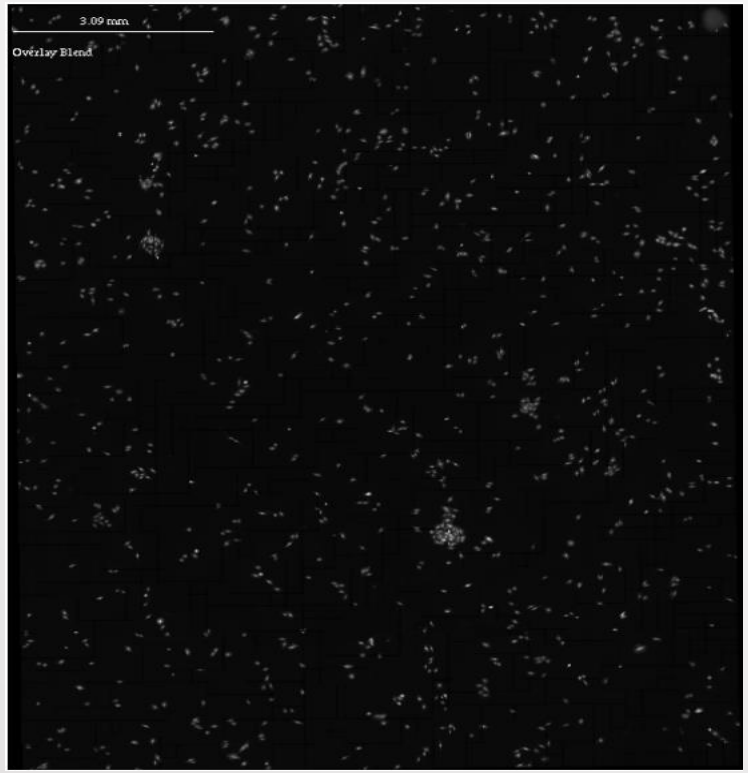
- Overhead analysis
  - Impact on CPU I/O
  - Memory overheads
- Microbenchmarks
  - Benefits of peer-caching
  - False sharing
  - Streaming read performance
- Applications
  - Performance of on-demand data I/O with image collage
  - Effects of false sharing in image stitching
  - Dynamic graph processing with Gunrock

For details see our paper

A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images





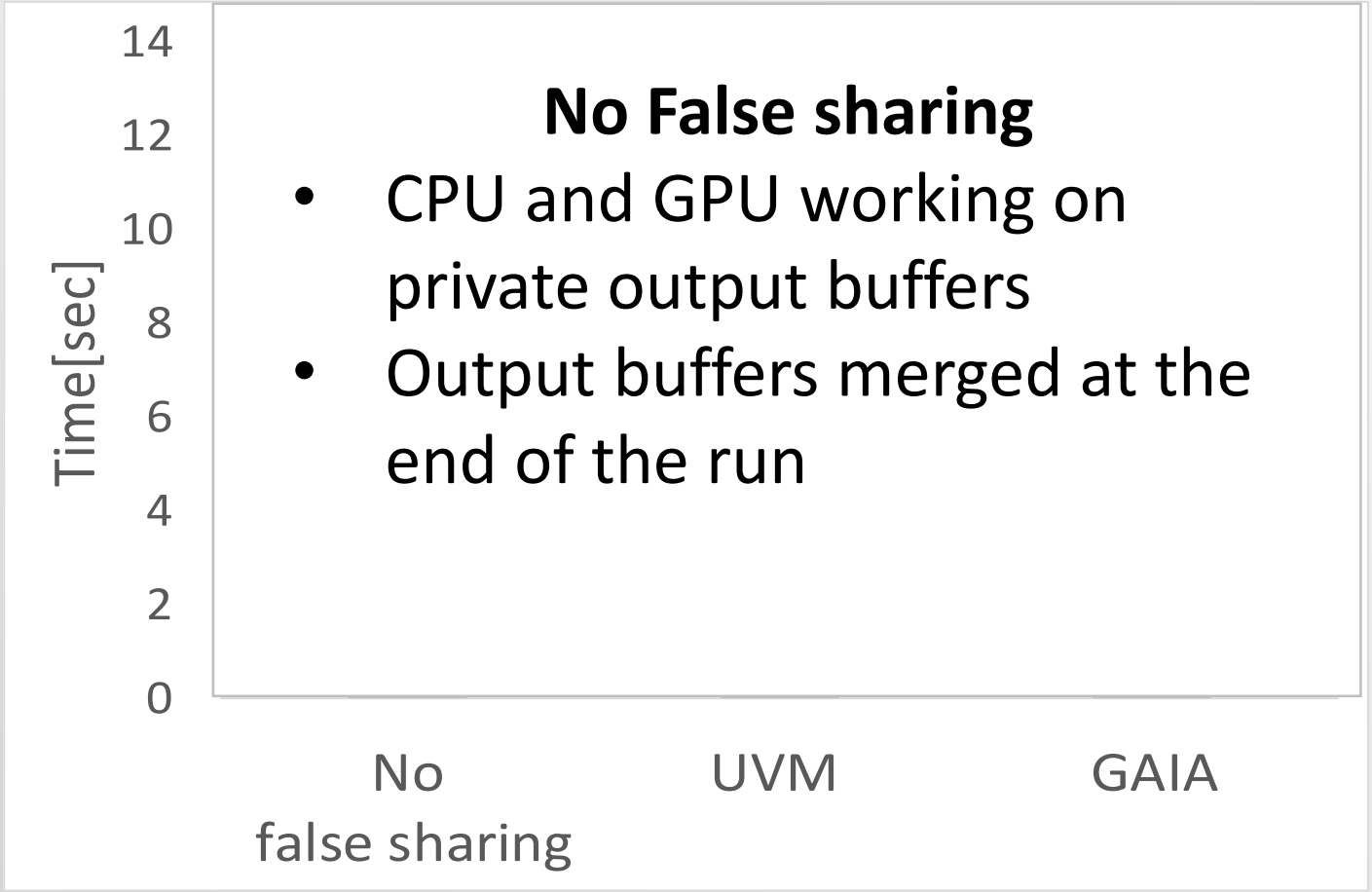


Timothy Blattner, Walid Keyrouz, Joe Chalfoun, Bertrand Stivalet, Mary Brady, and Shujia Zhou. "A HybridCPU-GPUSystemforStitchingLargeScaleOptical Microscopy Images." In 2014 43rd International Conference on Parallel Processing, pages 1–9, Sept 2014.

Tanya Brokhman, USENIX ATC19

ACSL
Technion Accelerated Computing Systems Lab

A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images

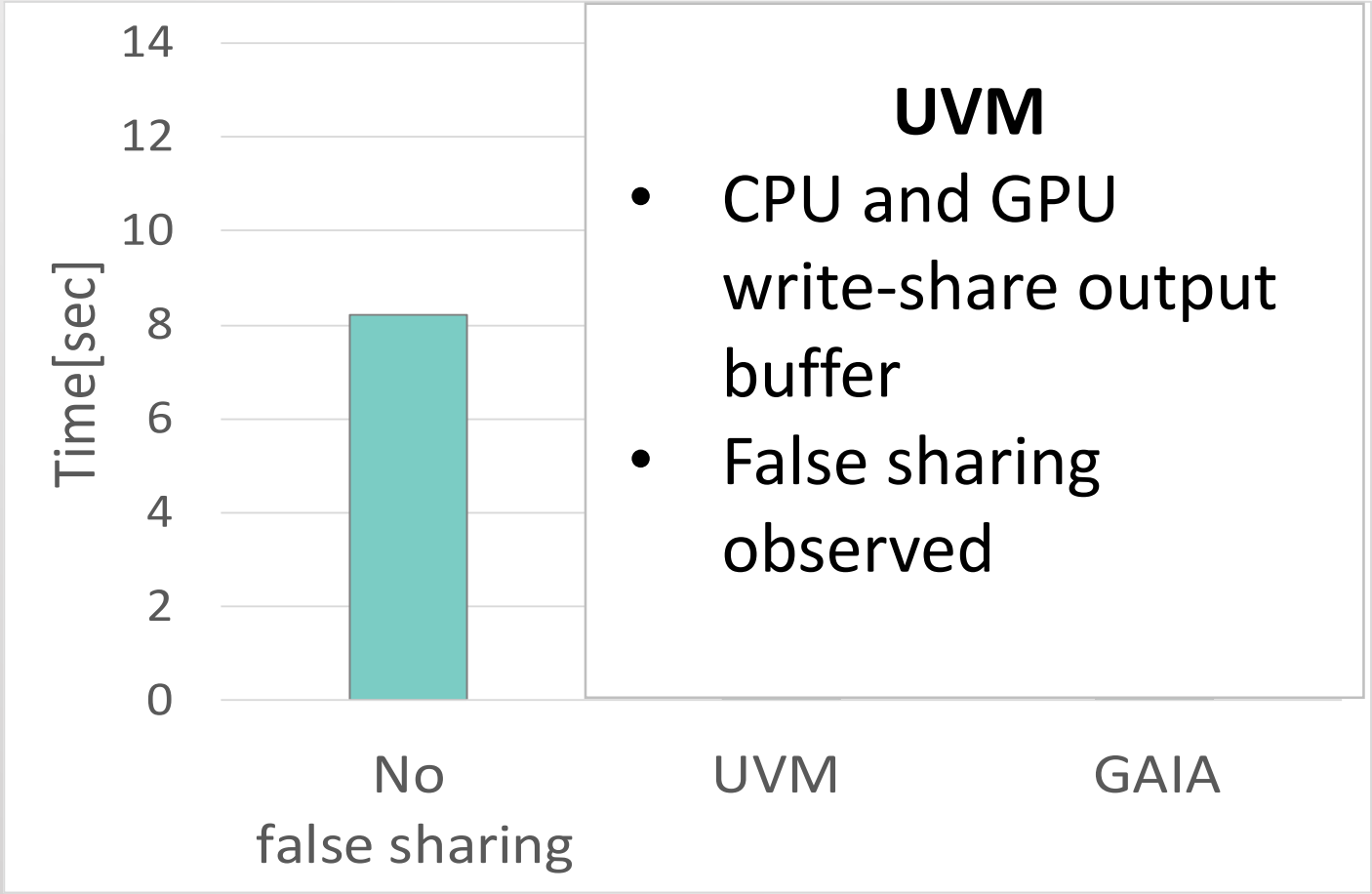


No False sharing

- CPU and GPU working on private output buffers
- Output buffers merged at the end of the run

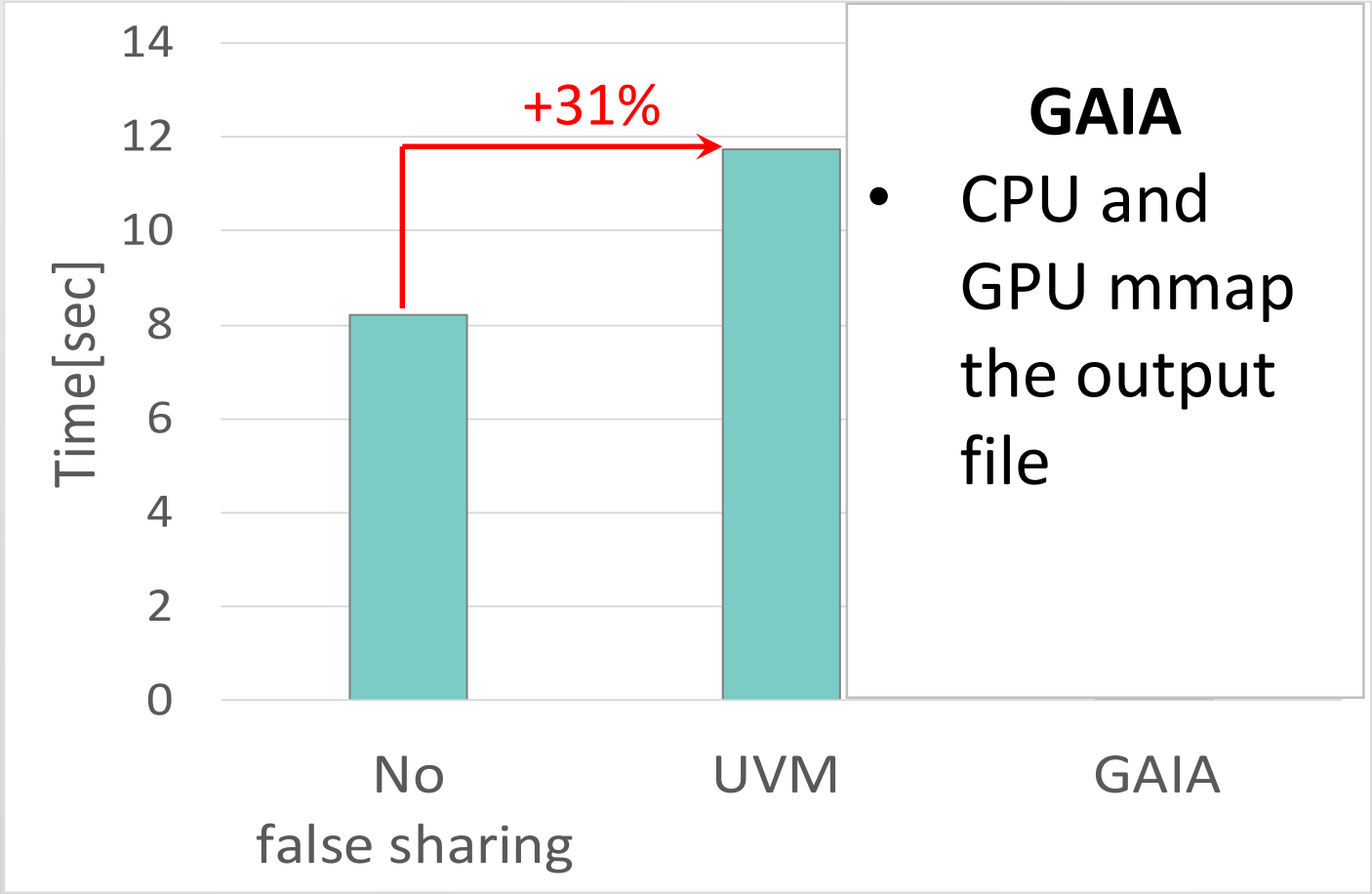
Output file size = 5.3GB

A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images



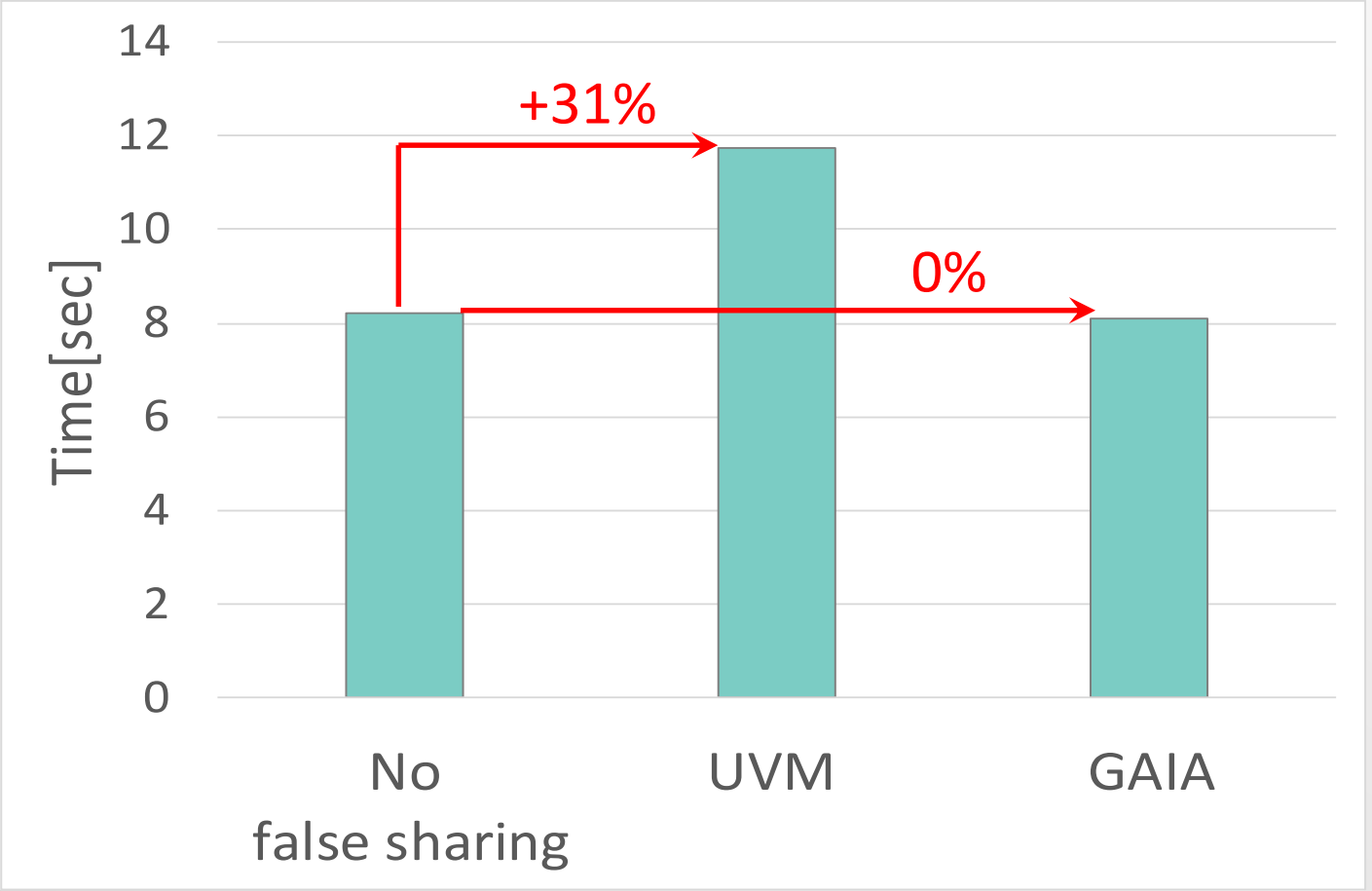
Output file size = 5.3GB

A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images



Output file size = 5.3GB

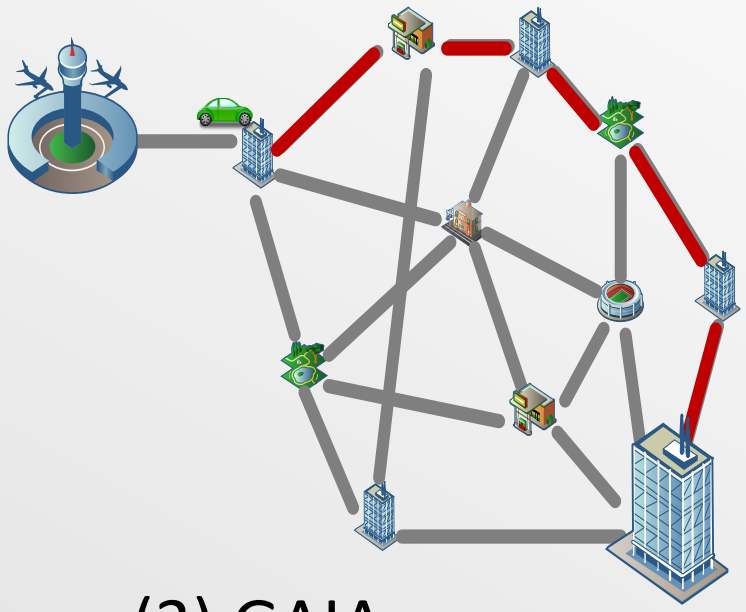
A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images



Output file size = 5.3GB

Road navigation service

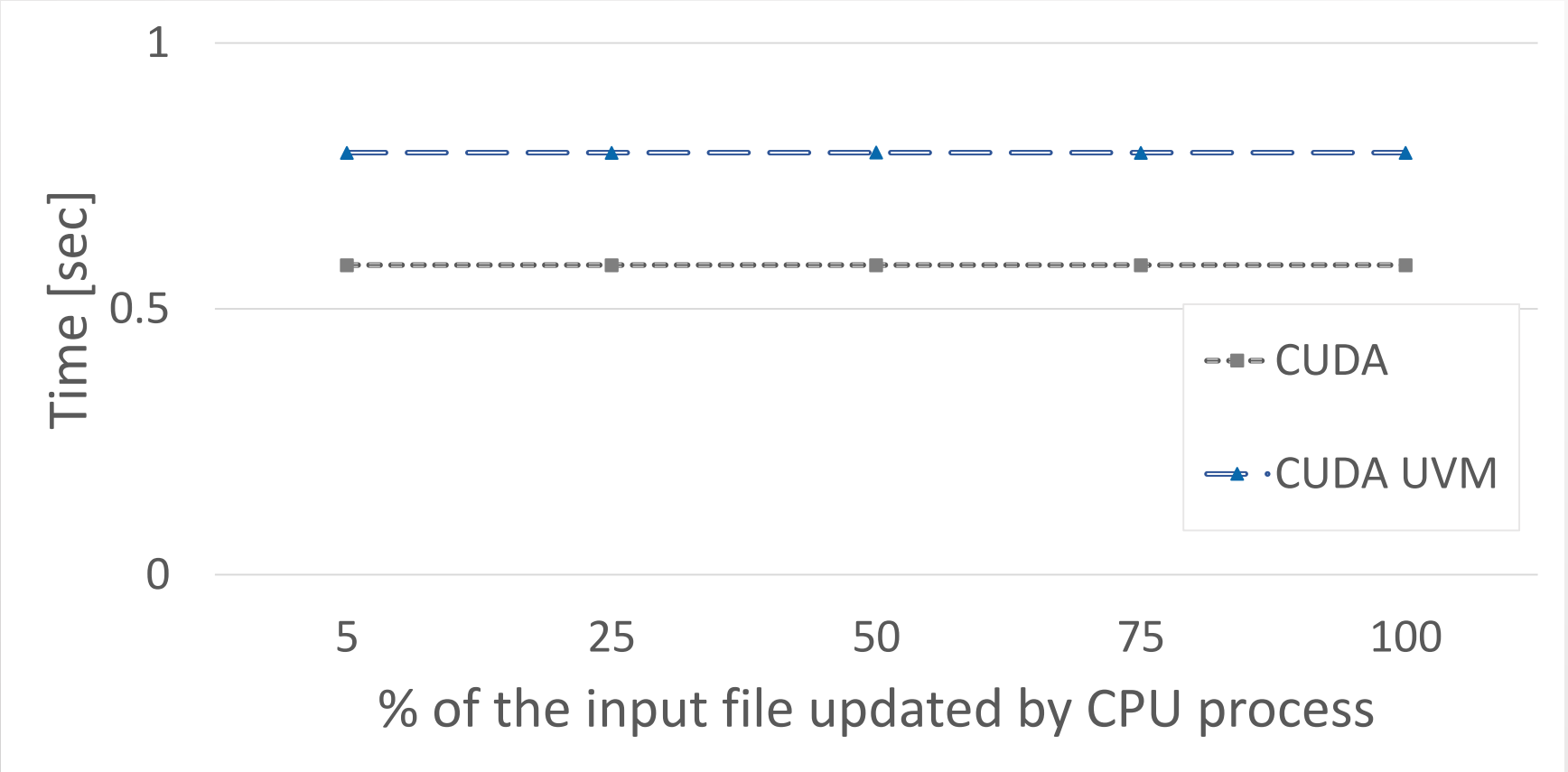
- Gunrock: CUDA library for graph-processing designed for the GPU
  - *Unmodified* Single Source Shortest Path application
- Legacy CPU process updates the input file
- Updates and graph computations are interleaved



- Compared implementations:

(1) CUDA (original) (2) UVM (3) GAIA

Dynamic graph processing with Gunrock

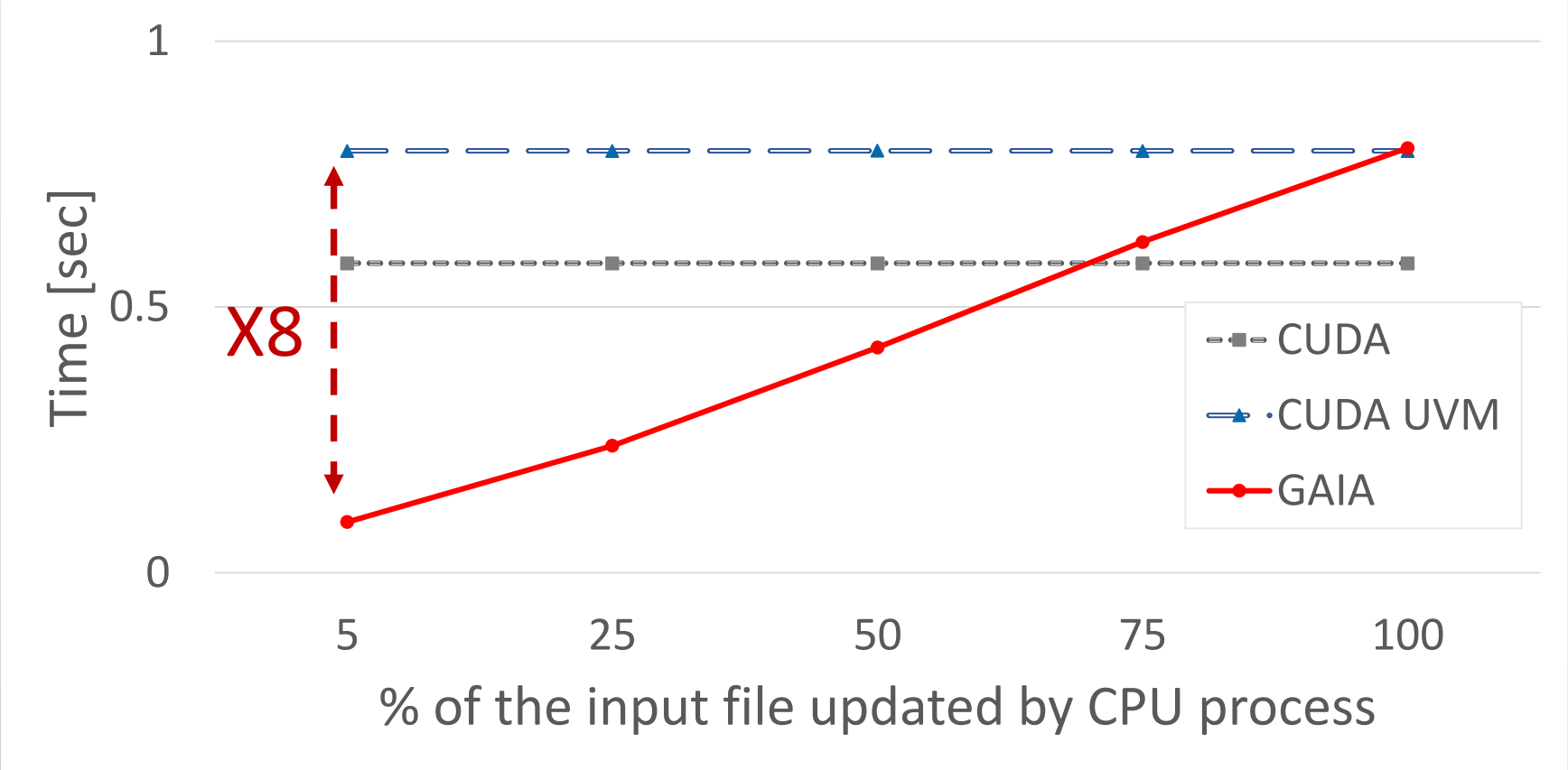


ACSL Technion Accelerated Computing Systems Lab

Tanya Brokhman, USENIX ATC19

Input file size = 5.2 GB

Dynamic graph processing with Gunrock



Tanya Brokhman, USENIX ATC19

Input file size = 5.2 GB

GAIA - Conclusion

- Scalable weakly consistent page cache abstraction extended to GPU memory
  - Demonstrating the benefit of LRC for write-shared workloads
  - Support mapping large files into GPU address space, enabling on-demand I/O
- Backward compatible with legacy CPU and *unmodified* GPU kernels
  - Transparent consistency support for legacy CPU applications
- IO optimizations for legacy CPU applications

http://github.com/acsl-technion/gaia

GAIA - Conclusion

- Scalable weakly consistent page cache abstraction extended to GPU memory
  - Demonstrating the benefit of LRC for write-shared workloads
  - Support mapping large files into GPU address space, enabling on-demand I/O
- Backward compatible with legacy CPU and *unmodified* GPU kernels
  - Transparent consistency support for legacy CPU applications
- IO optimizations for legacy CPU applications

Questions?