Operating System Services for High Throughput Processors

Mark Silberstein
EE, Technion

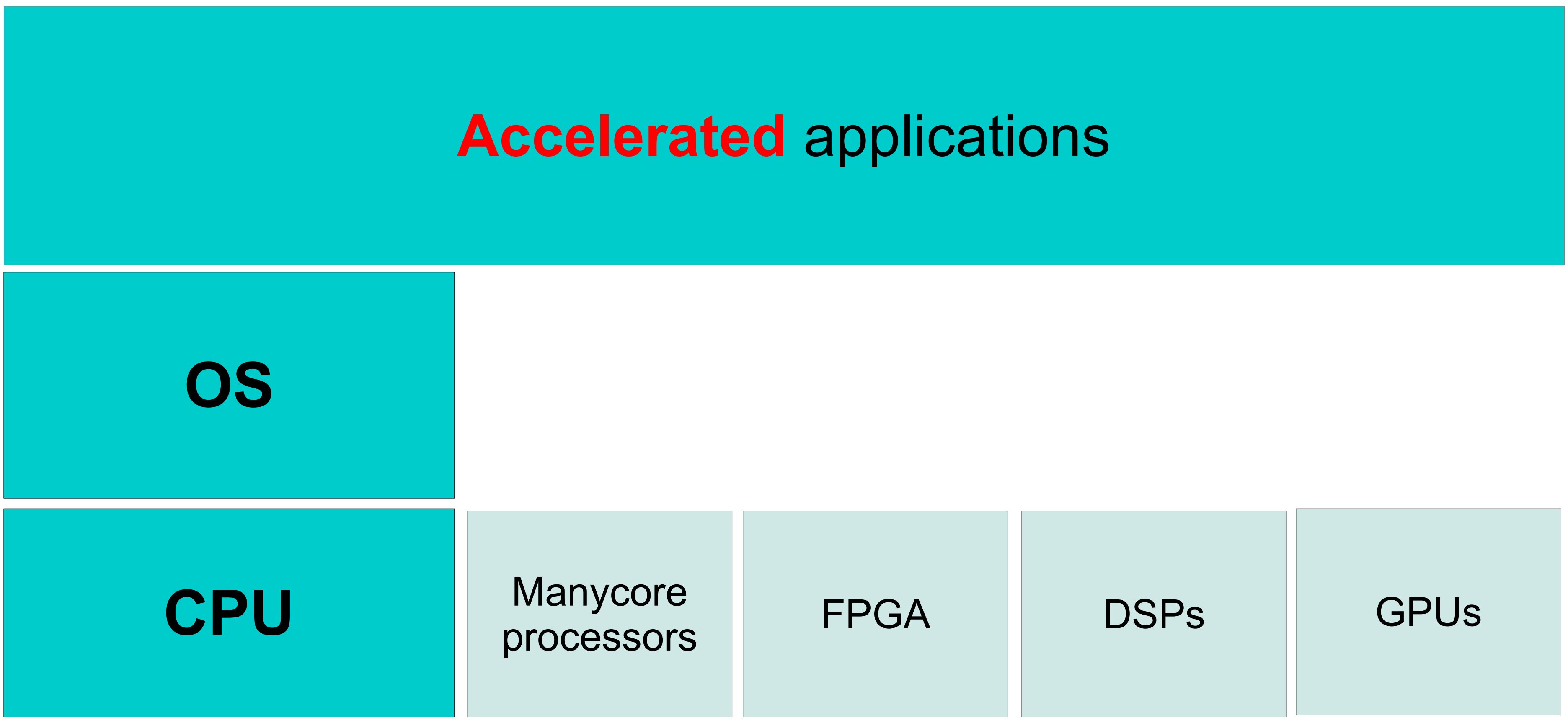
Traditional Systems Software Stack

Applications

OS

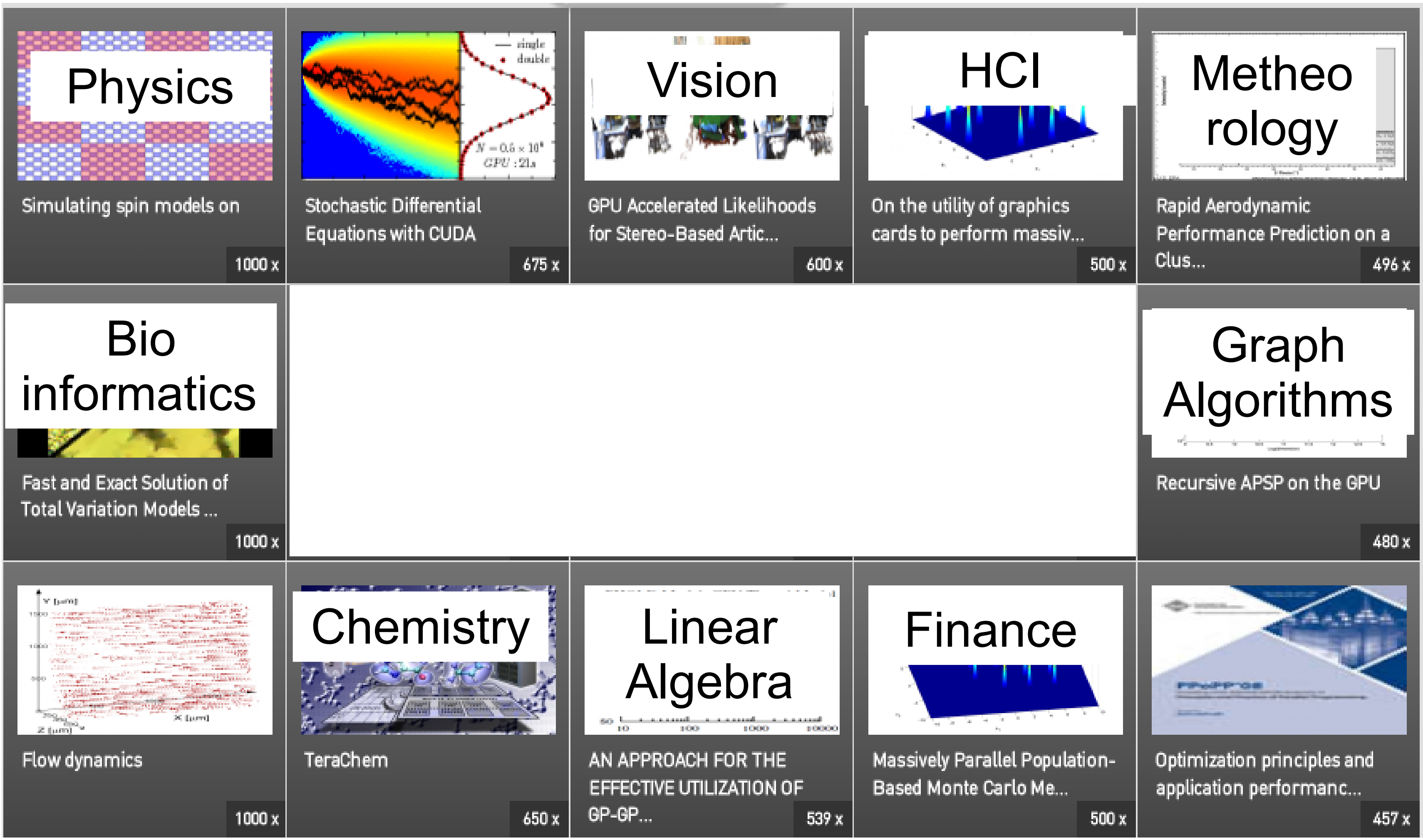
CPU

Modern Systems Software Stack

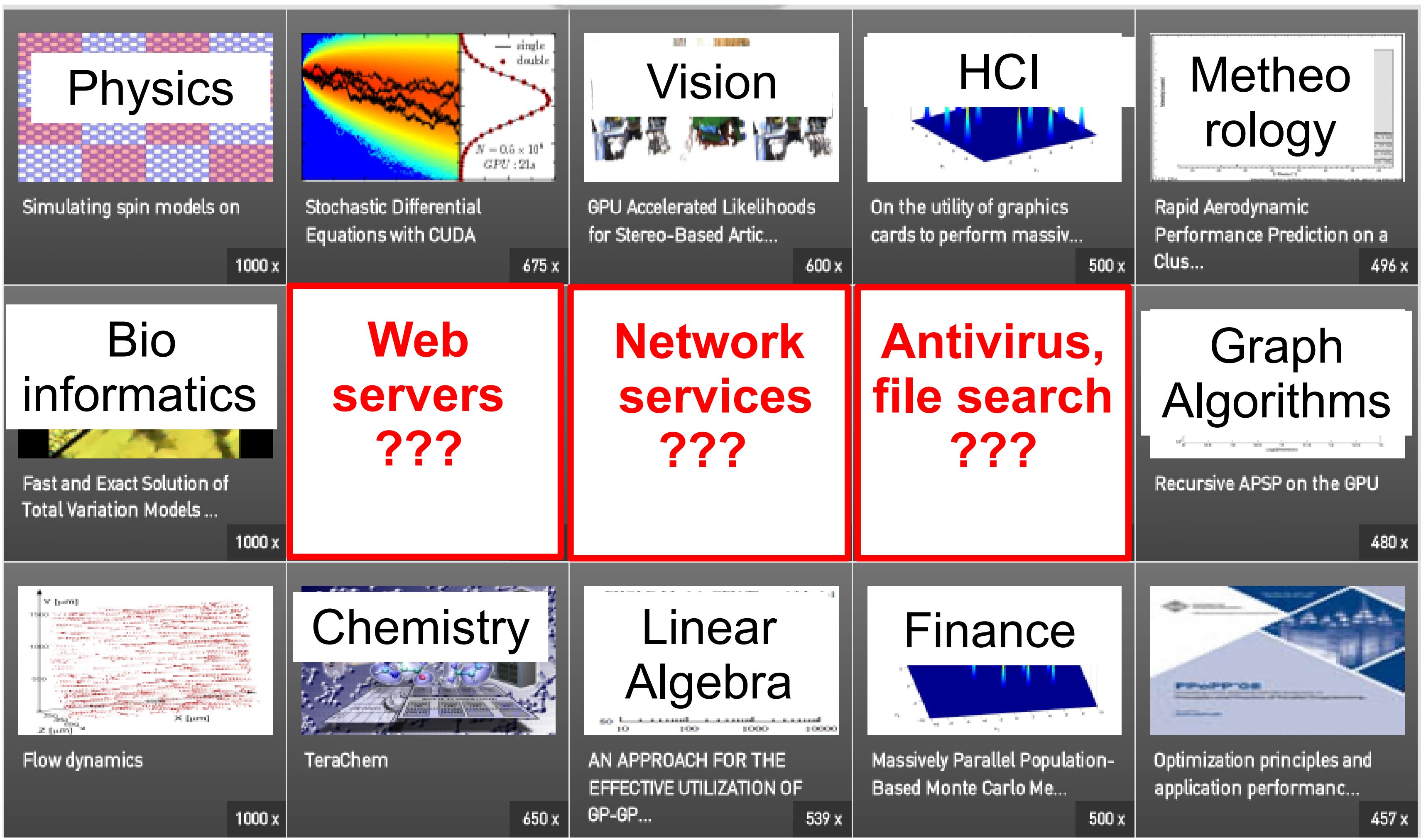


GPUs make a difference...

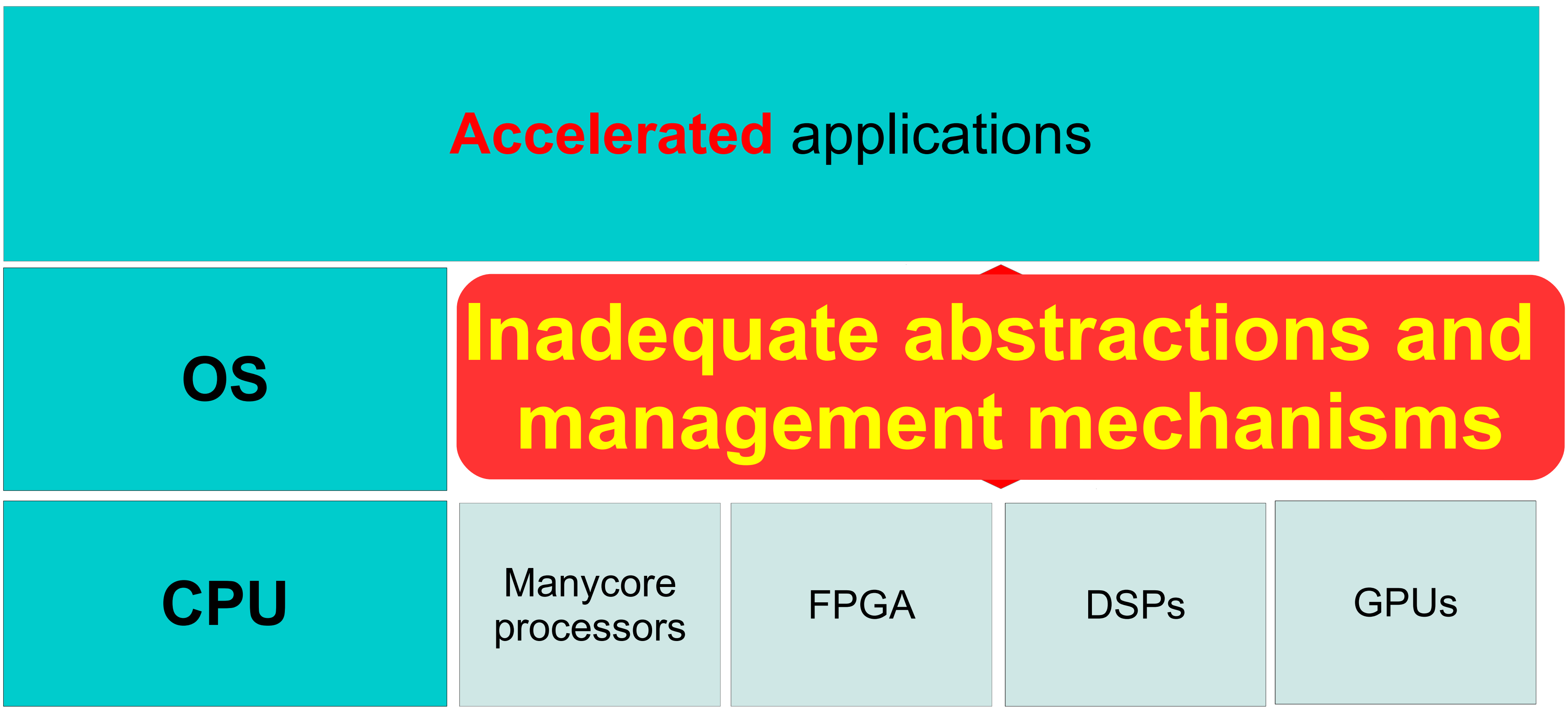
- Top 10 fastest supercomputers use GPUs



GPUs make a difference, but only in HPC!



Software-hardware gap is widening

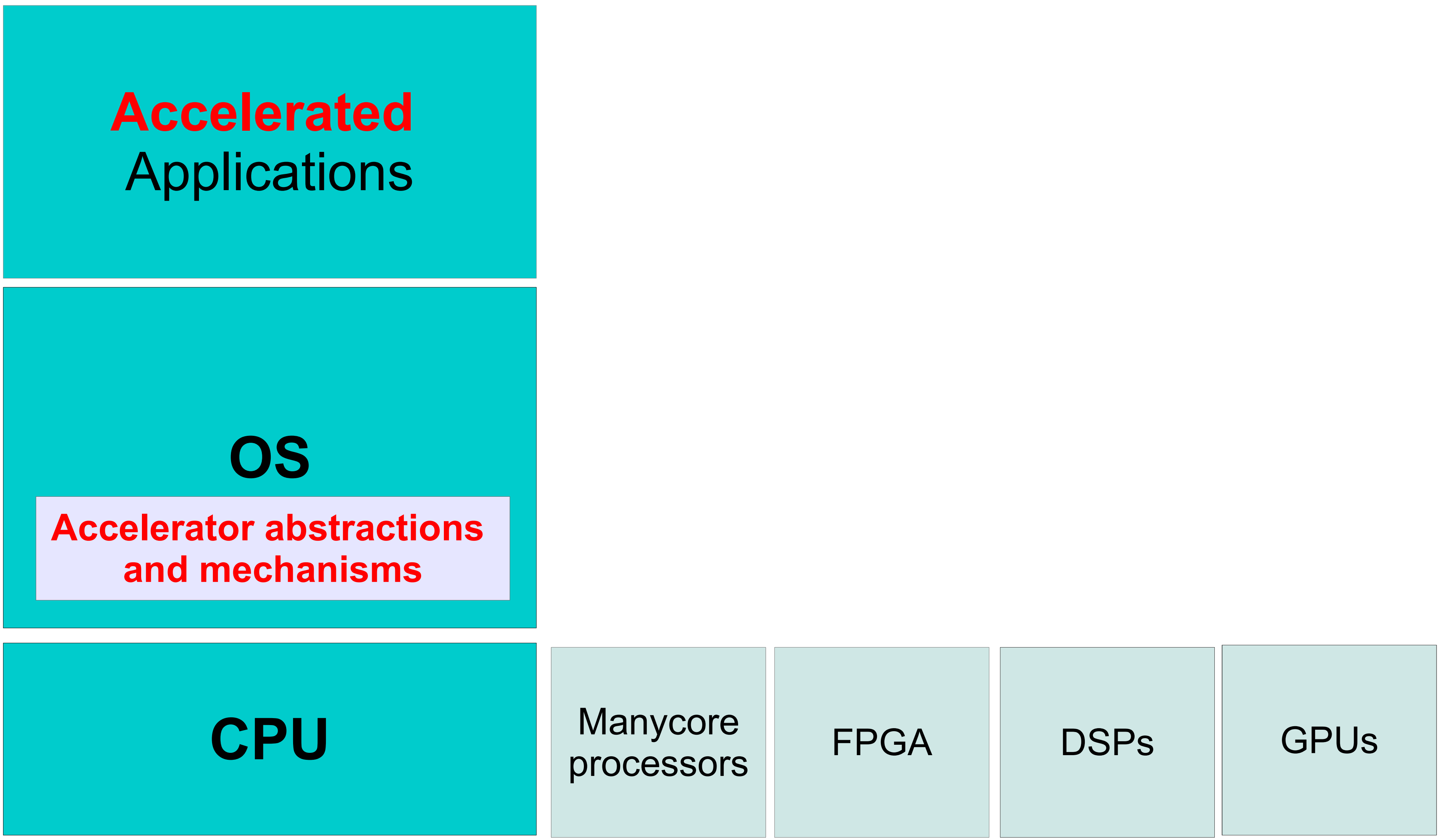


Fundamentals in question

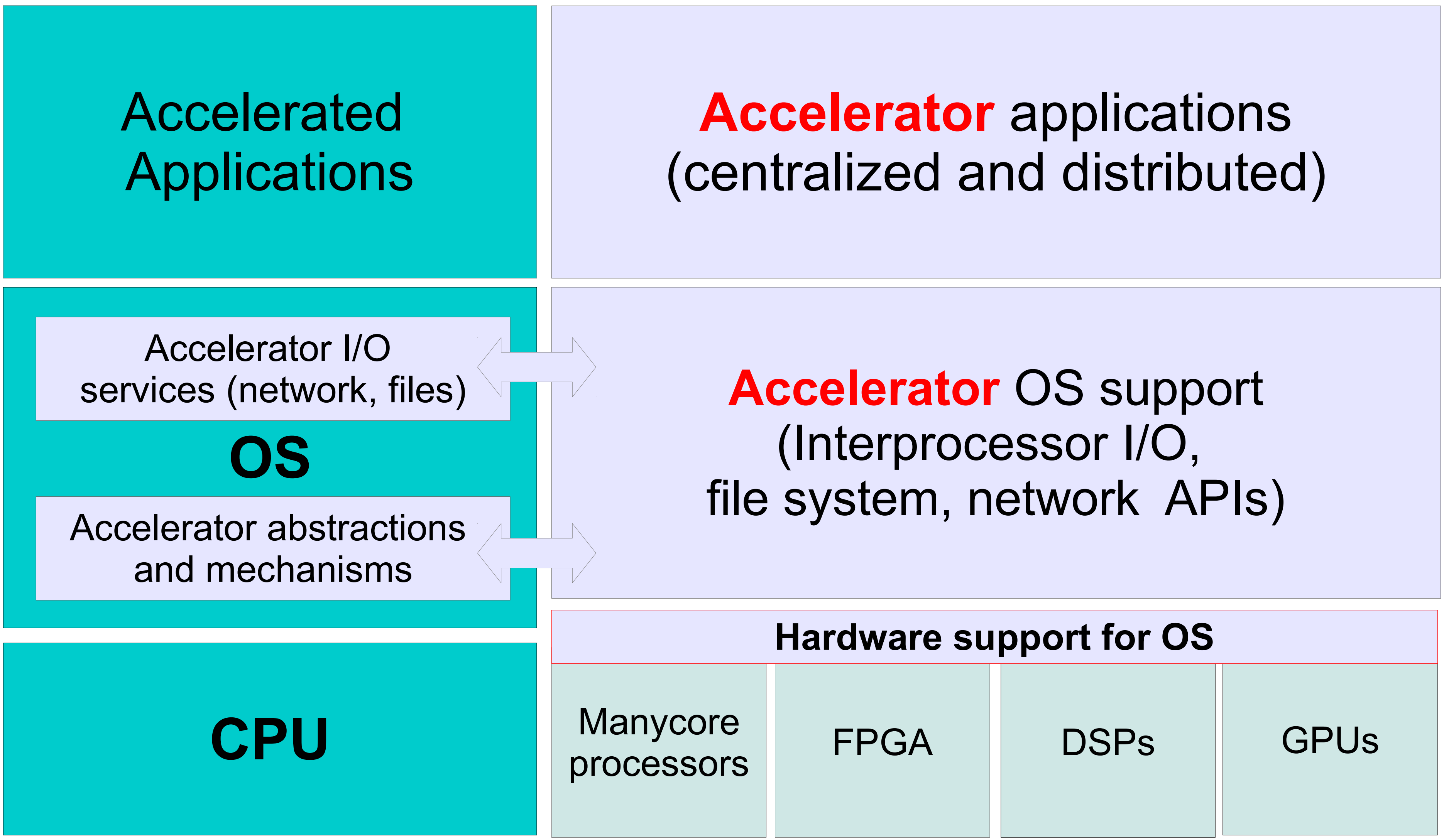
accelerators $\not\equiv$ co-processors

accelerators $\equiv$ peer-processors

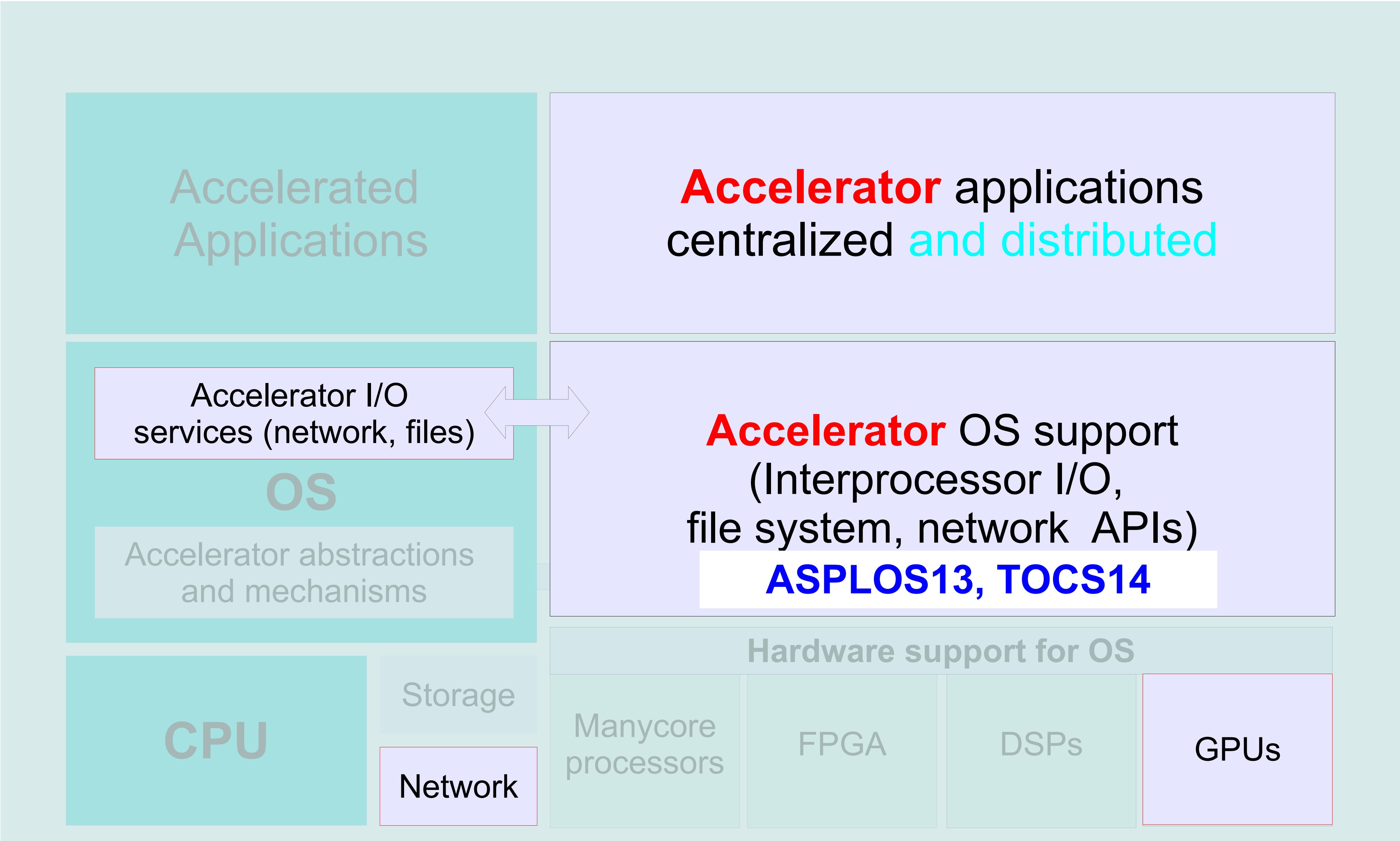
Software stack for accelerat*ed* applications



Software stack for accelerat***or*** applications



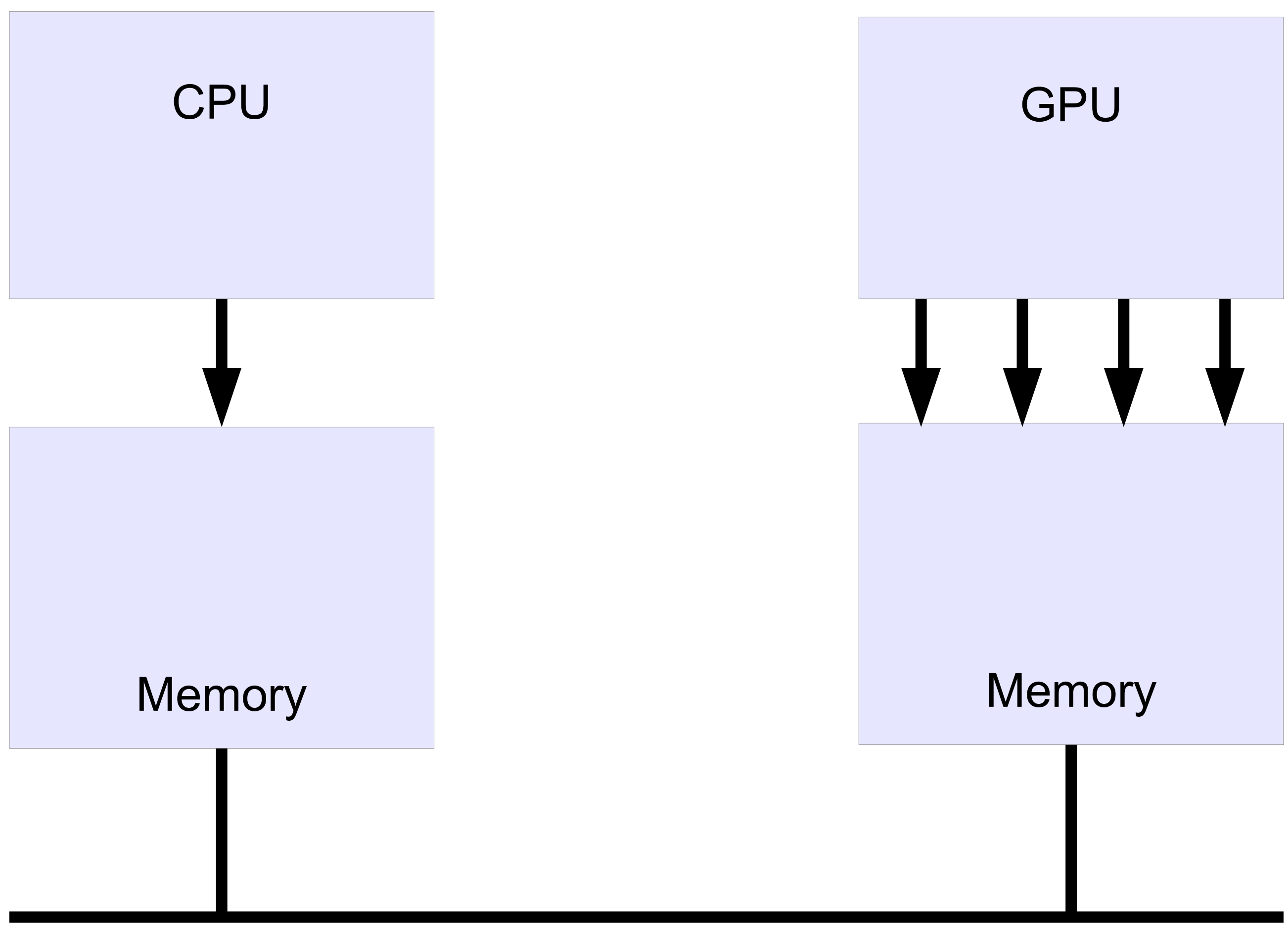
This talk



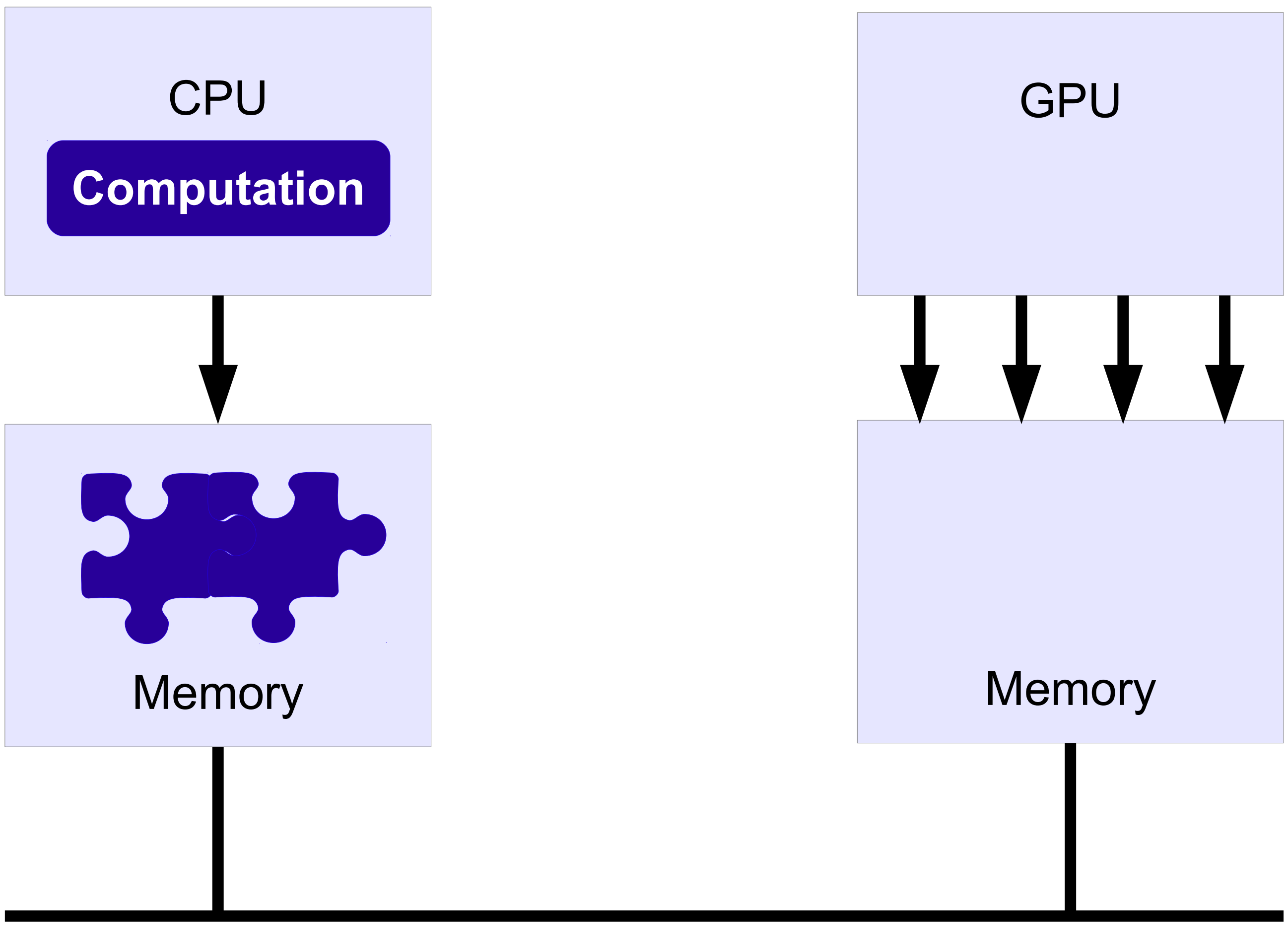
Accelerated Applications
Accelerator applications centralized and distributed
Accelerator I/O services (network, files)
OS
Accelerator abstractions and mechanisms
Accelerator OS support (Interprocessor I/O, file system, network APIs)
ASPLOS13, TOCS14
Hardware support for OS
CPU
Storage
Network
Manycore processors
FPGA
DSPs
GPUs

- **GPU 101**
- GPUfs: File I/O support for GPUs
- Future work

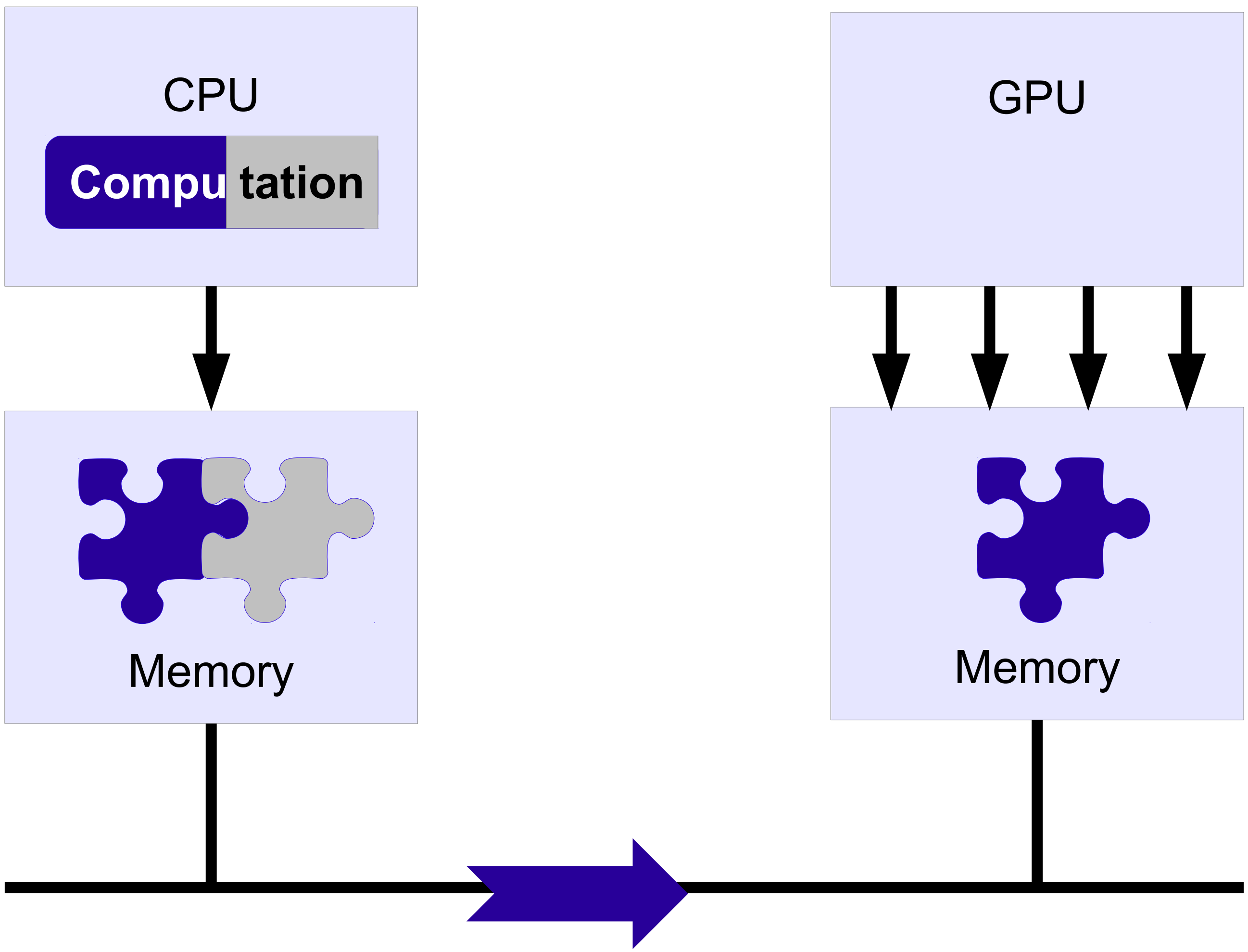
Hybrid GPU-CPU 101 Architecture



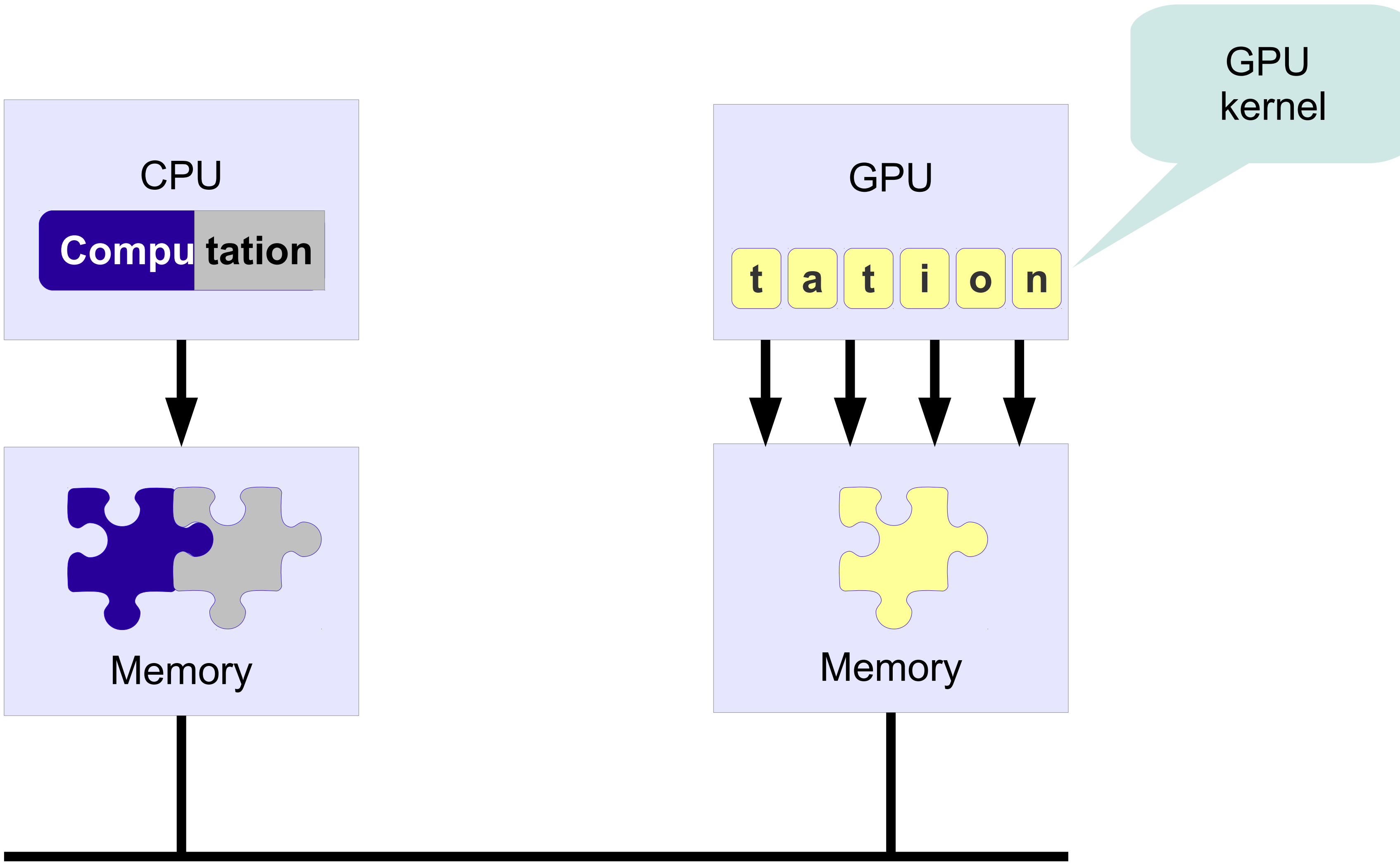
Co-processor model



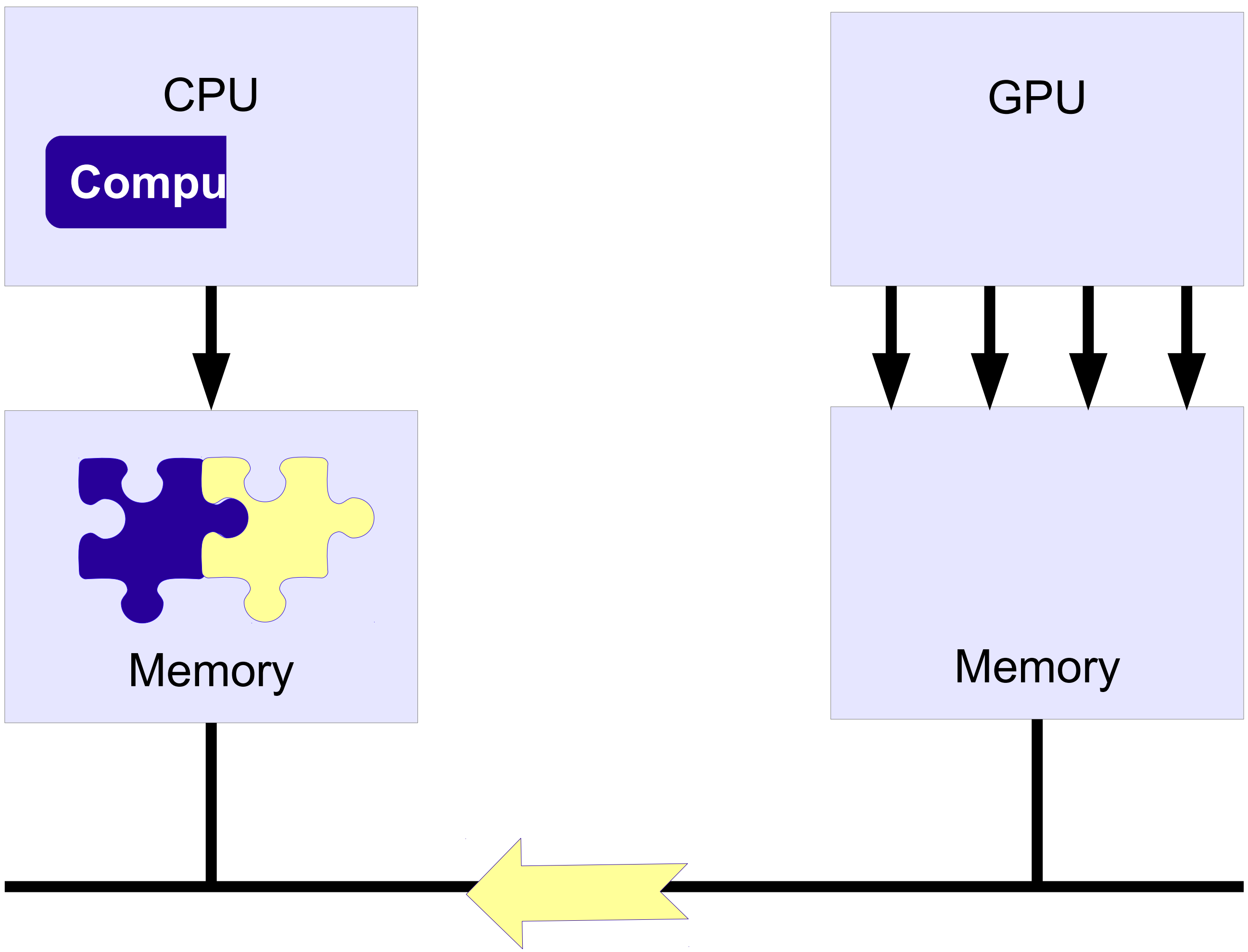
Co-processor model



Co-processor model



Co-processor model



Building systems with GPUs is hard

Why?

GPU kernels are isolated

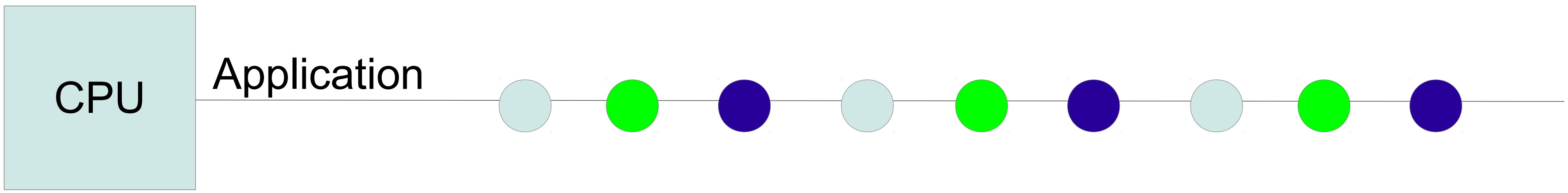
CPU

Data transfers
Invocation
Memory management

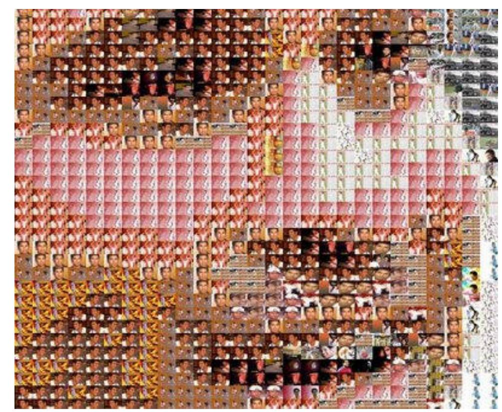
GPU

Parallel
Algorithm

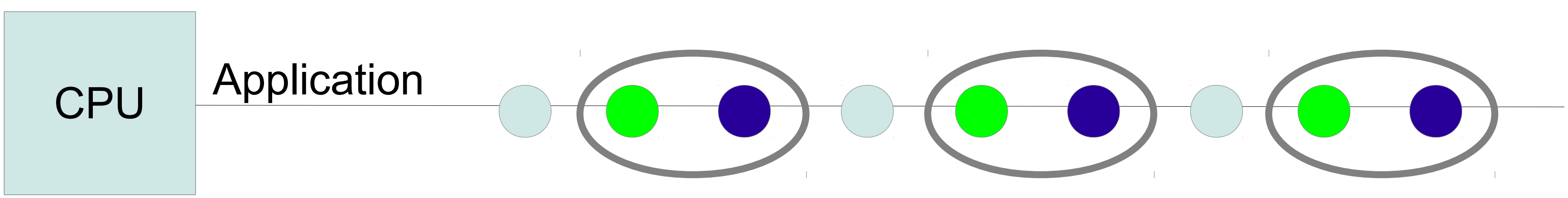
Example: accelerating photo collage




```
While(Unhappy()){
   Read_next_image_file()
   Decide_placement()
   Remove_outliers()
}
```



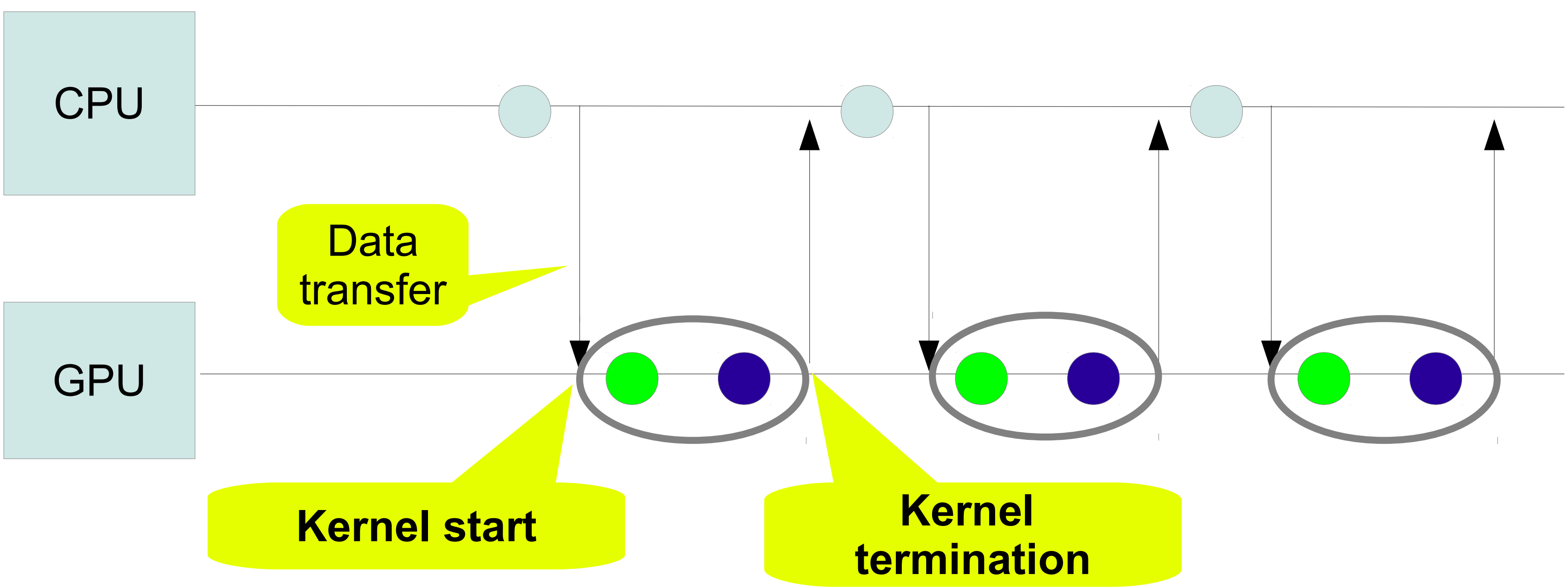
Offloading computations to GPU



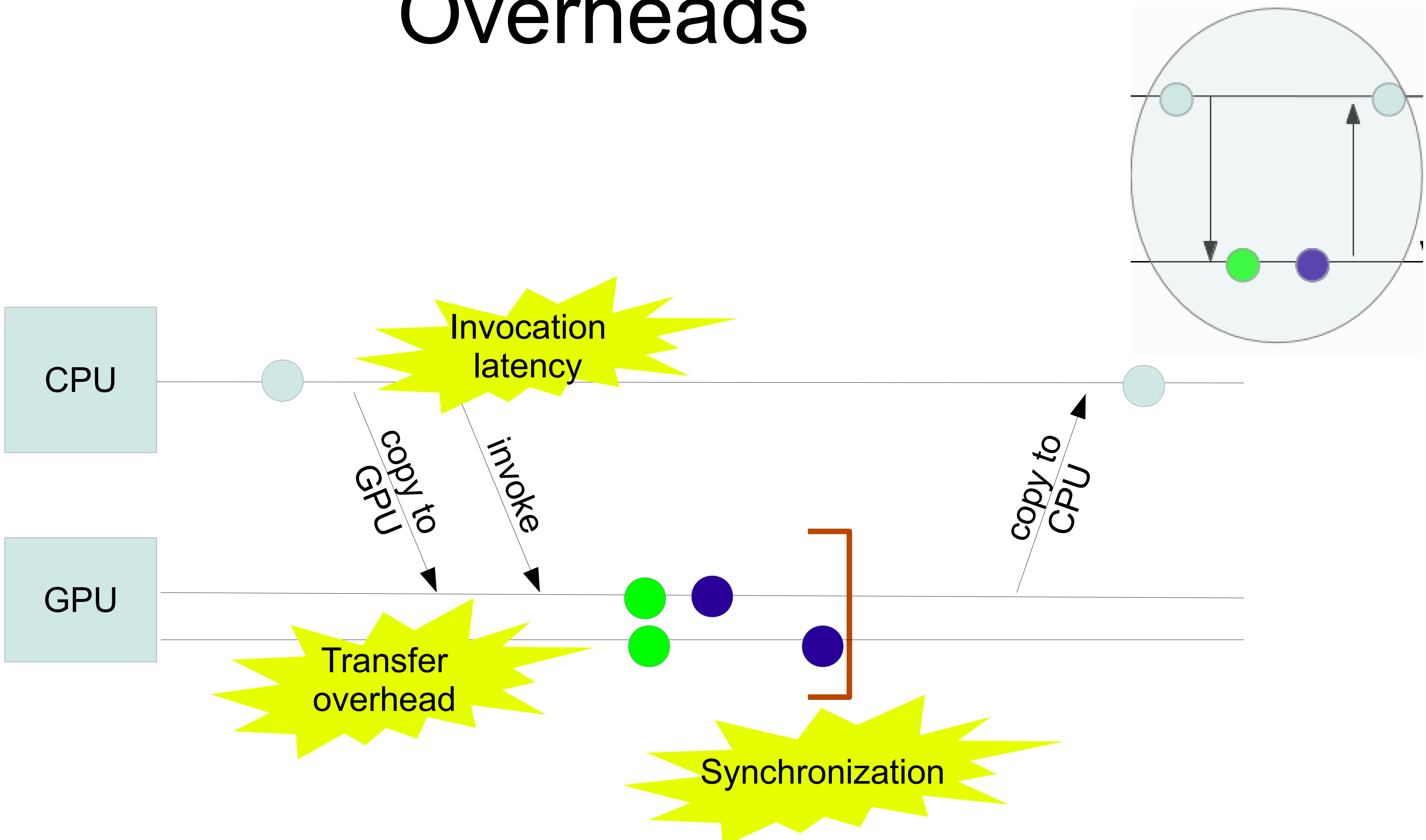
Move to GPU

```
While(Unhappy()){
   Read next image file()
   Decide_placement()
   Remove outliers()
}
```

Offloading computations to GPU

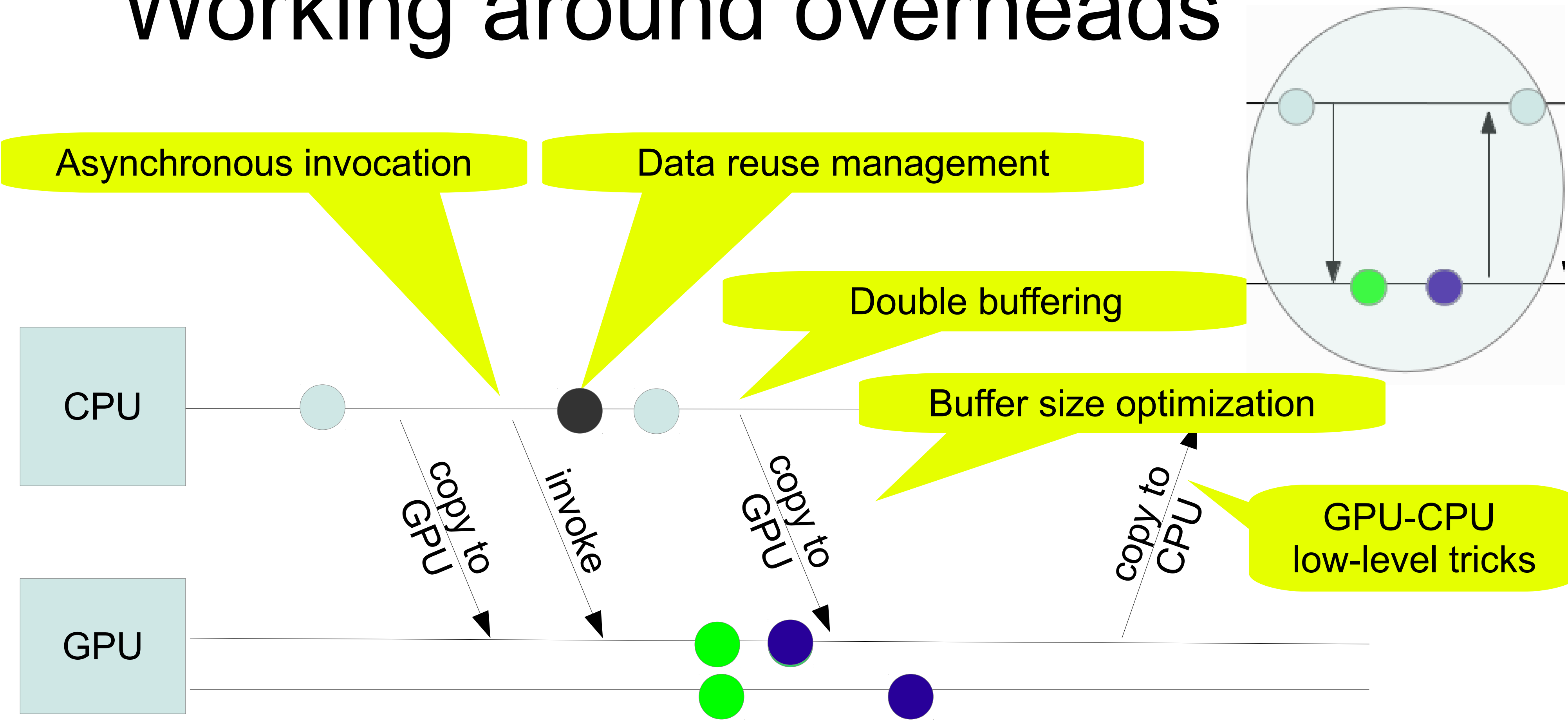


Overheads



CPU
GPU
Invocation latency
copy to GPU
invoke
copy to CPU
Transfer overhead
Synchronization

Working around overheads



Management overhead

Asynchronous invocation

Data reuse management

Double buffering

Buffer size optimization

GPU-CPU
low-level tricks

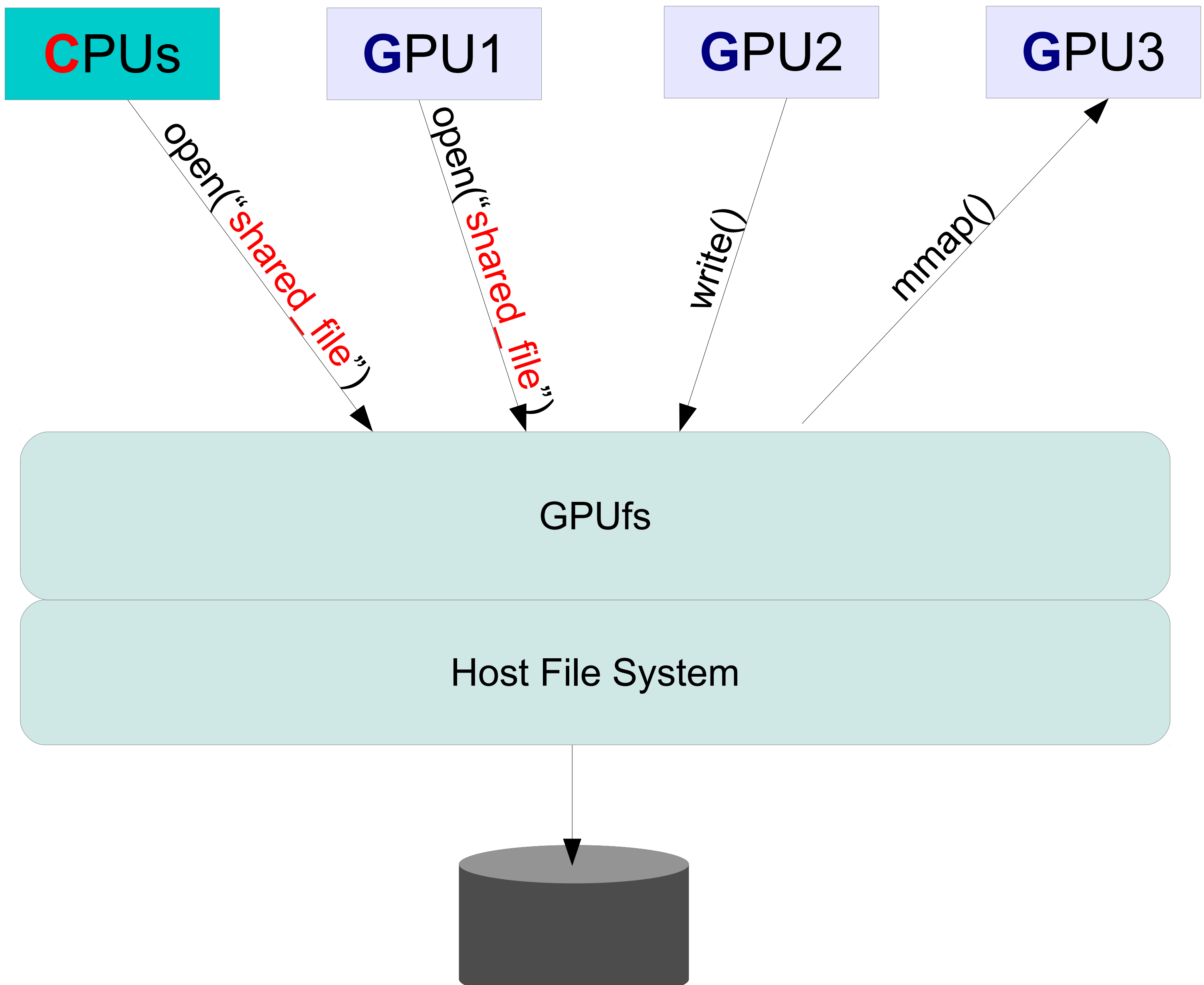
Why do we need to deal with
low-level system details?

The reason is....

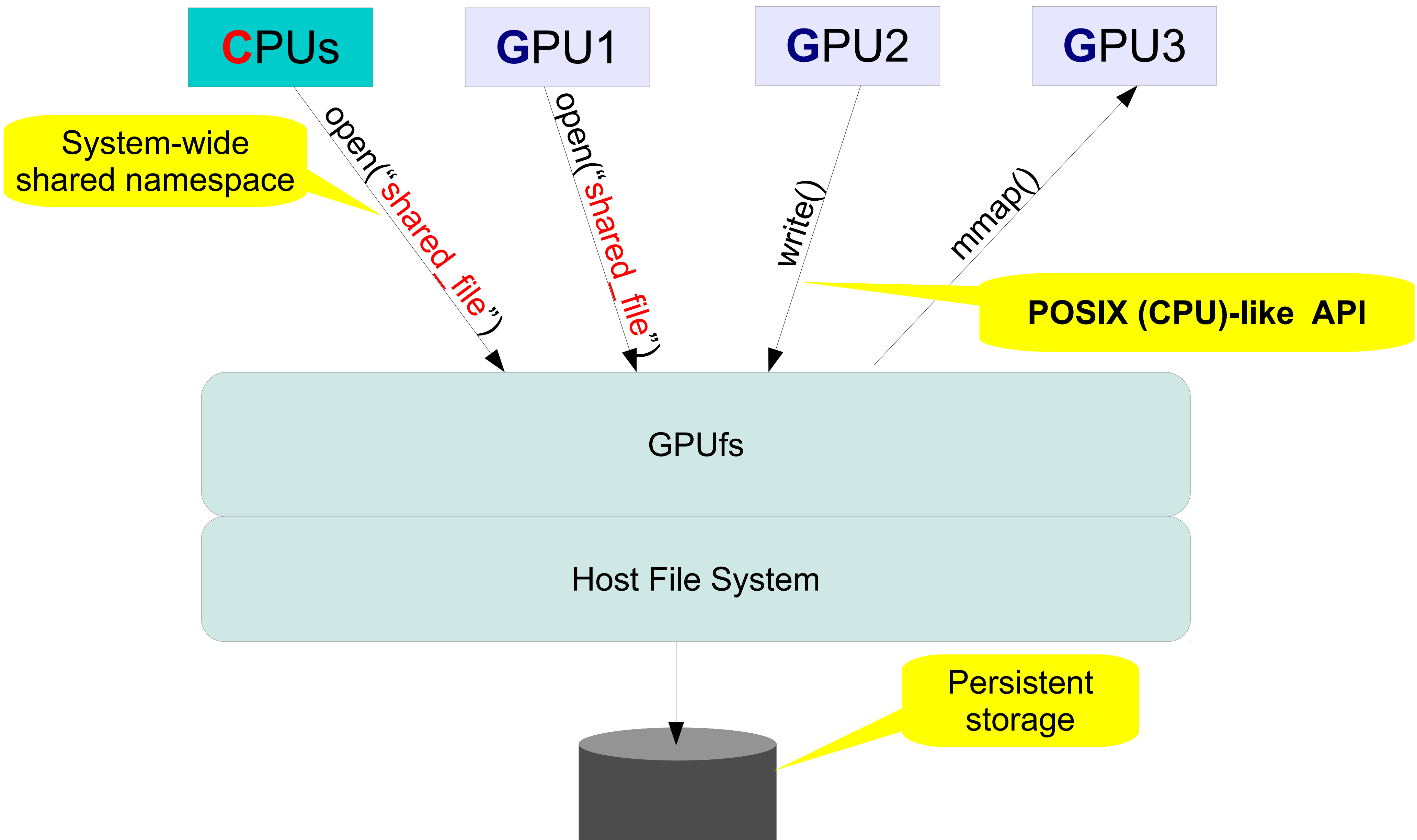
GPUs are peer-processors

They need I/O OS services

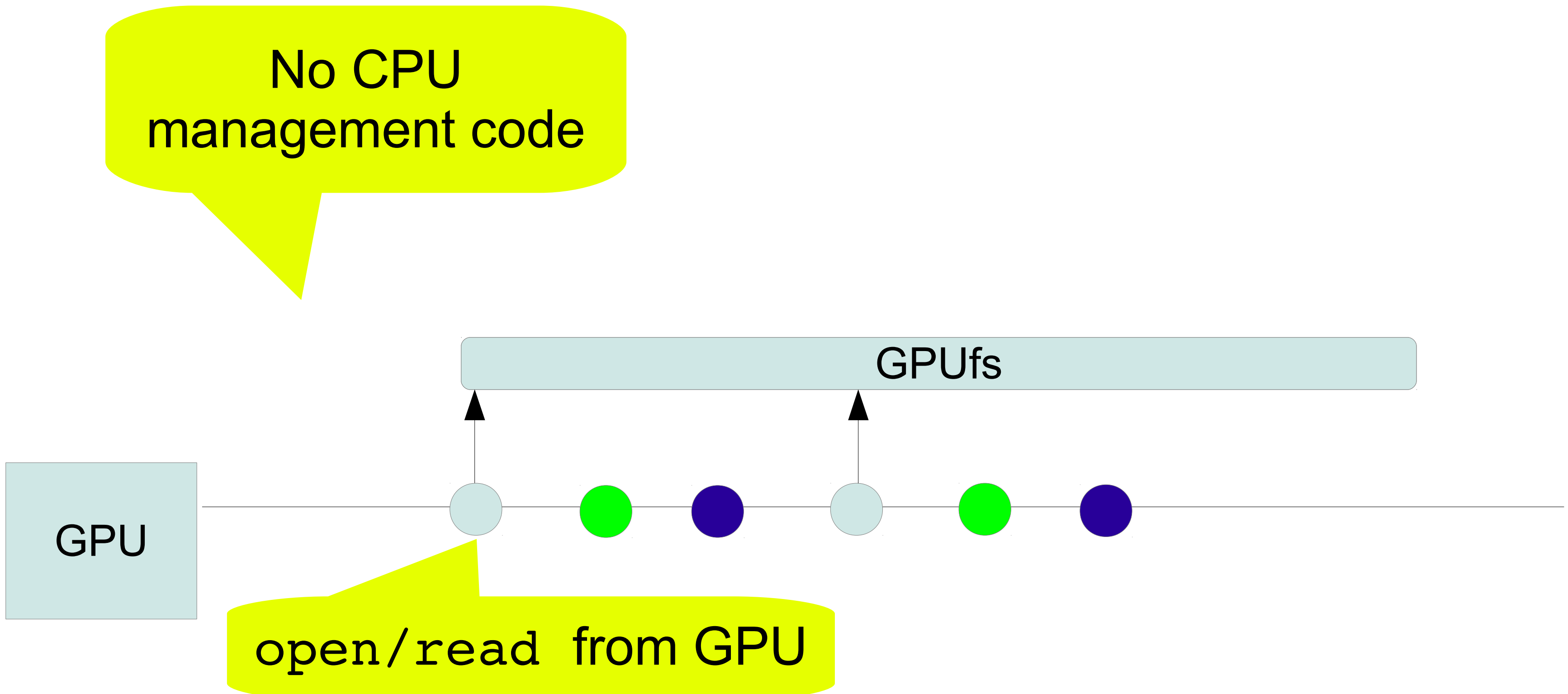
GPUfs: application view



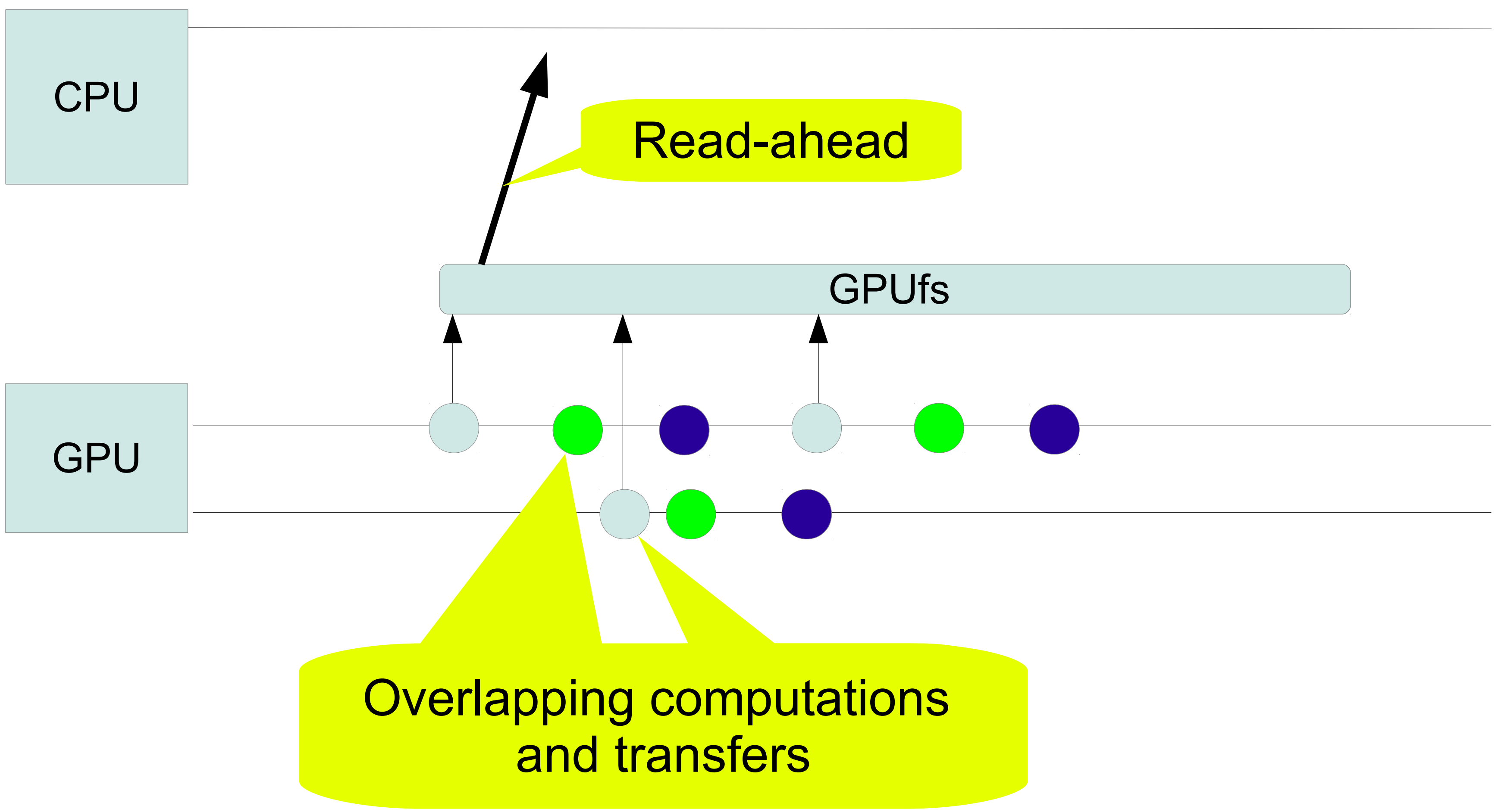
GPUfs: application view



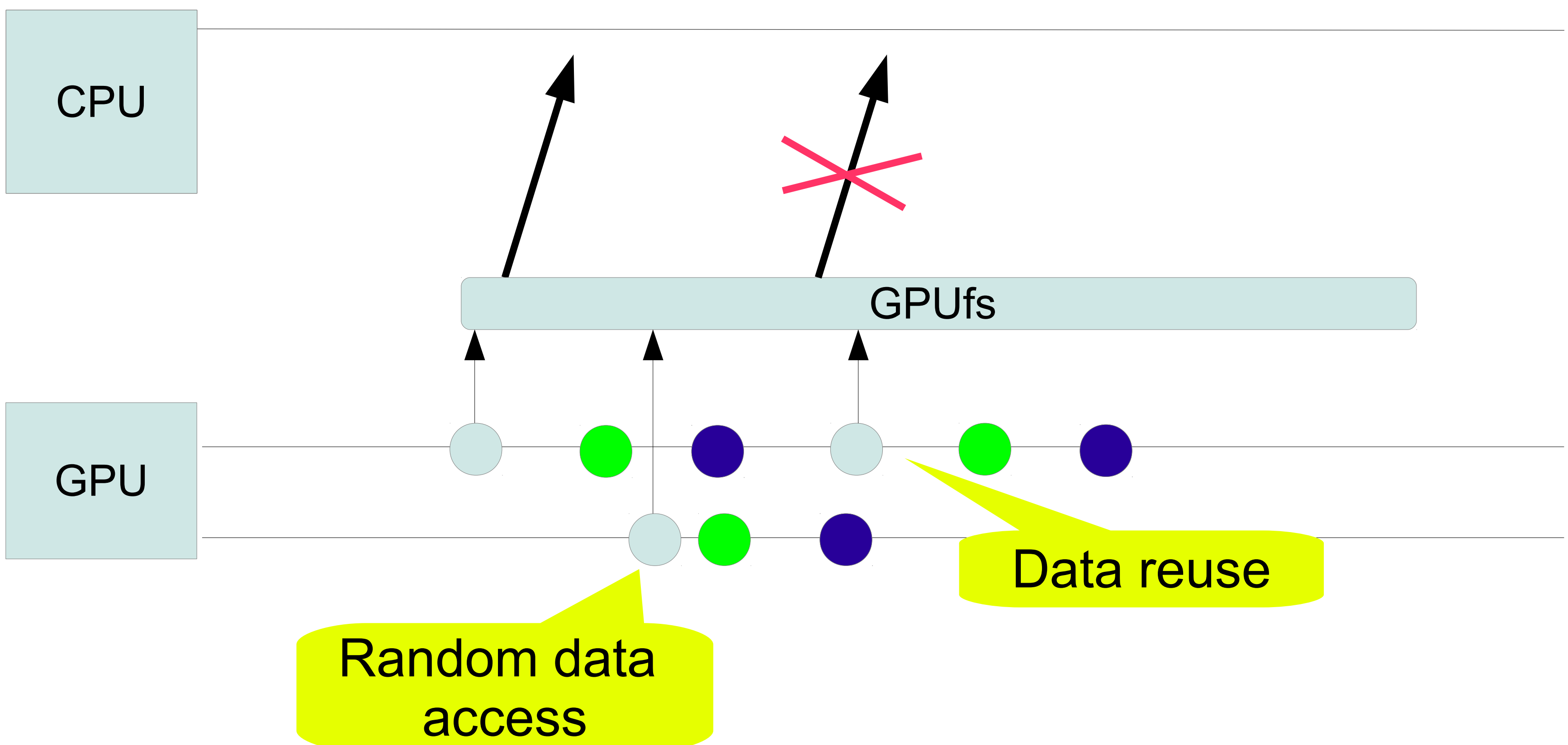
Accelerating collage app with GPUfs



Accelerating collage app with GPUfs



Accelerating collage app with GPUfs



Understanding the hardware

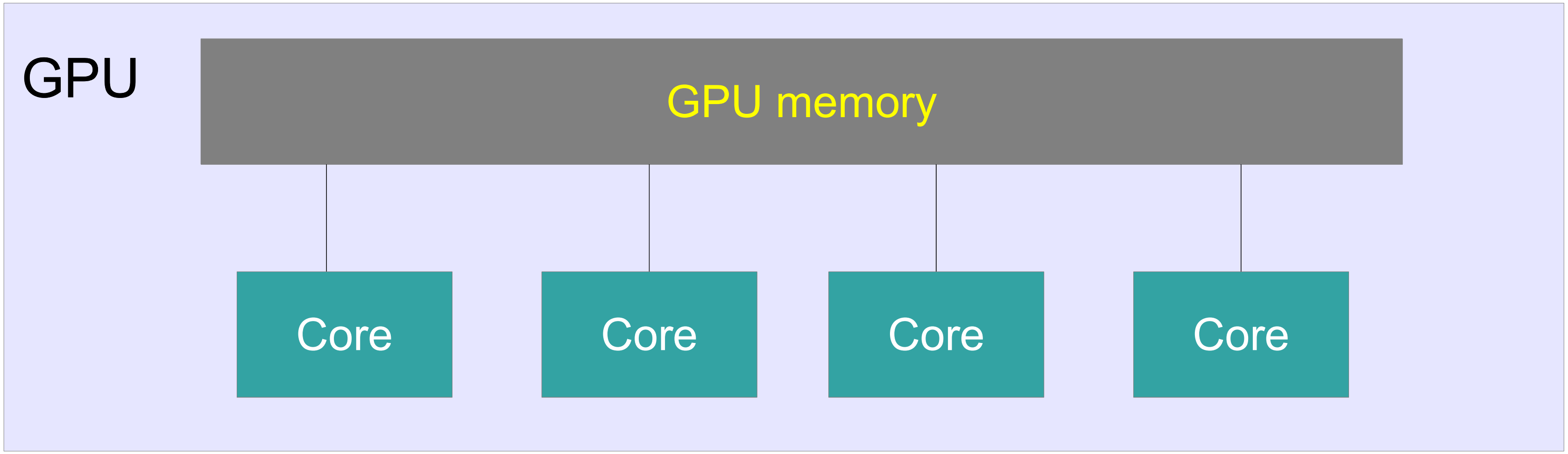
GPU hardware characteristics

Parallelism

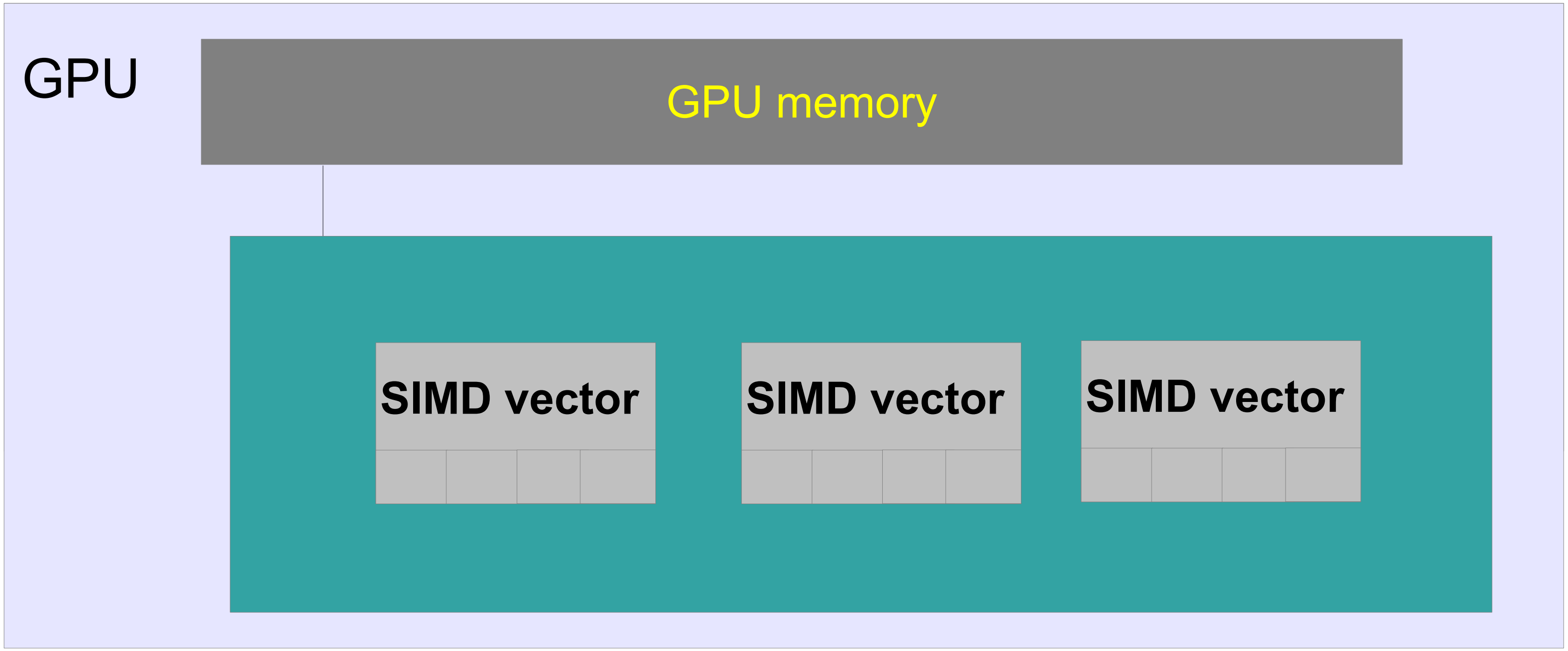
Low serial performance

Heterogeneous memory

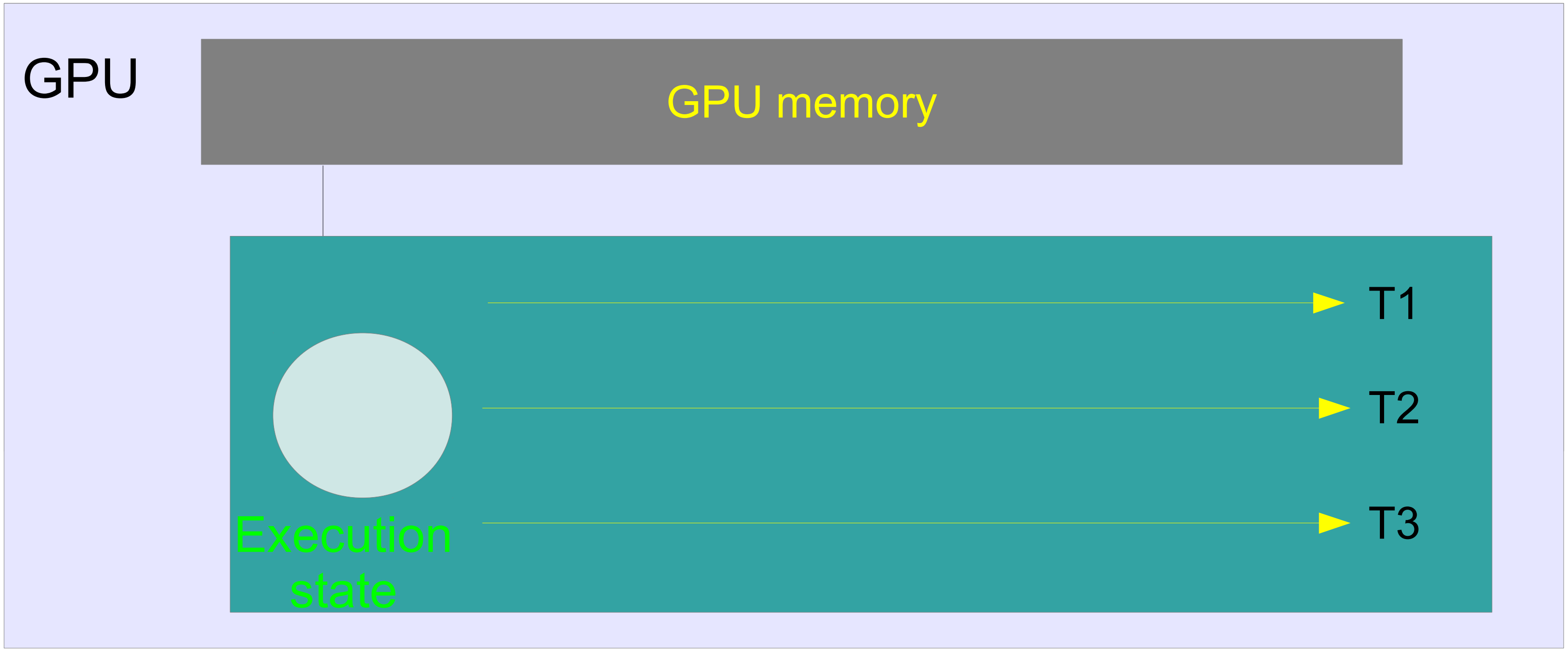
GPU hardware parallelism
1. Multi-core



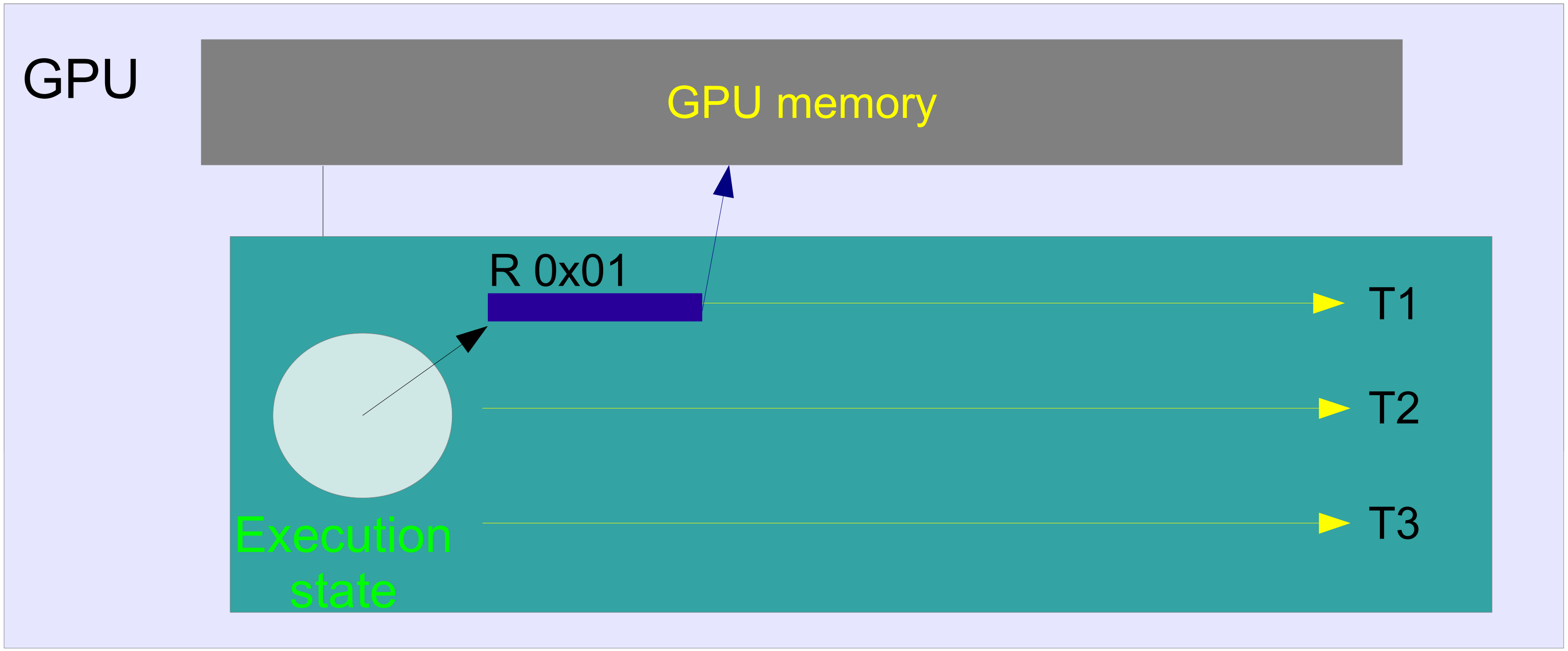
GPU hardware parallelism
2. SIMD



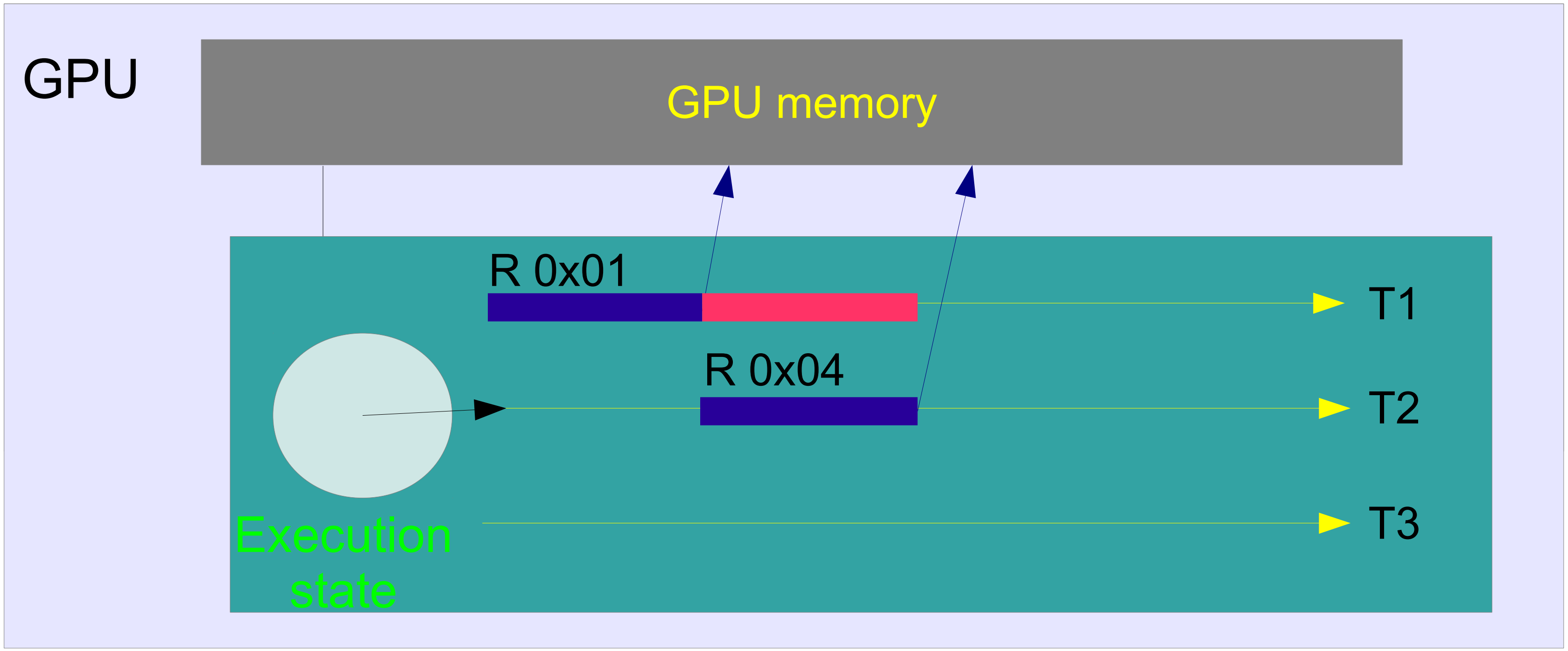
GPU hardware parallelism
3. Parallelism for latency hiding



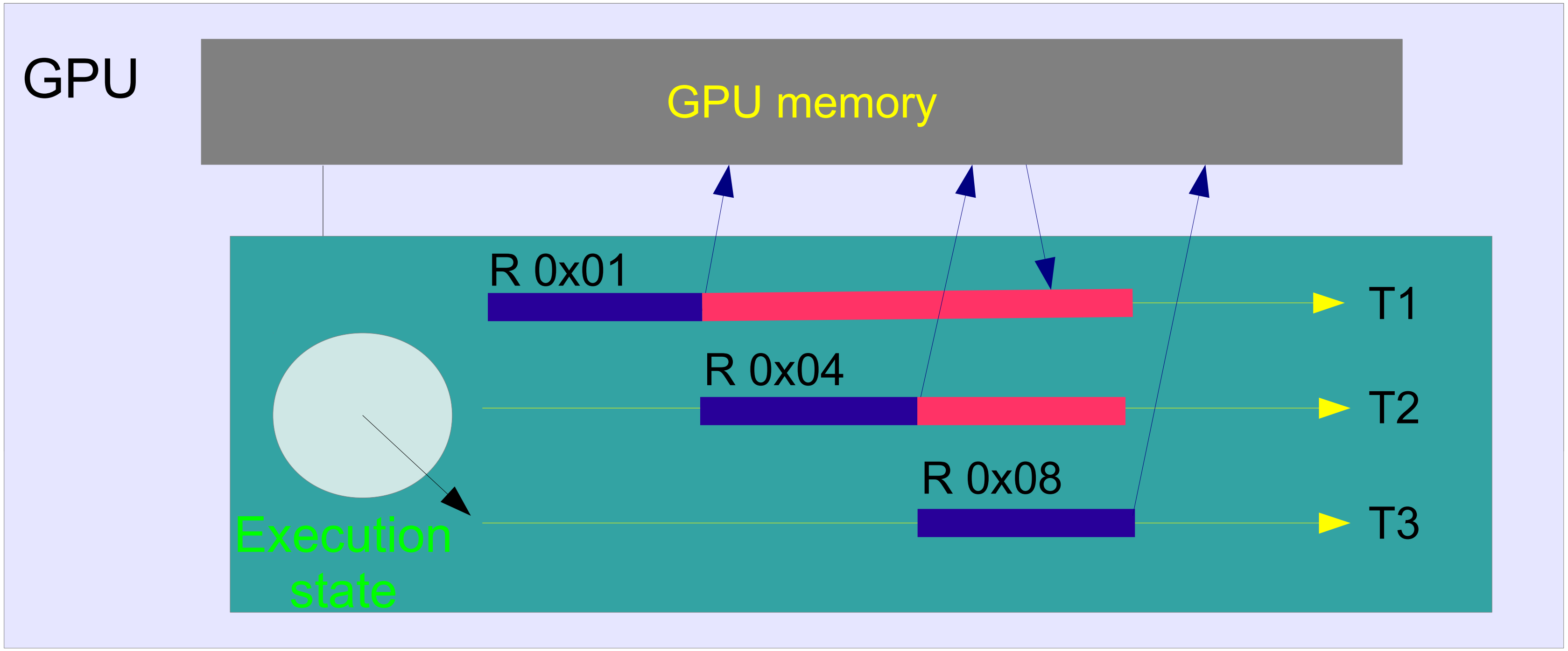
GPU Hardware
3. Parallelism for latency hiding



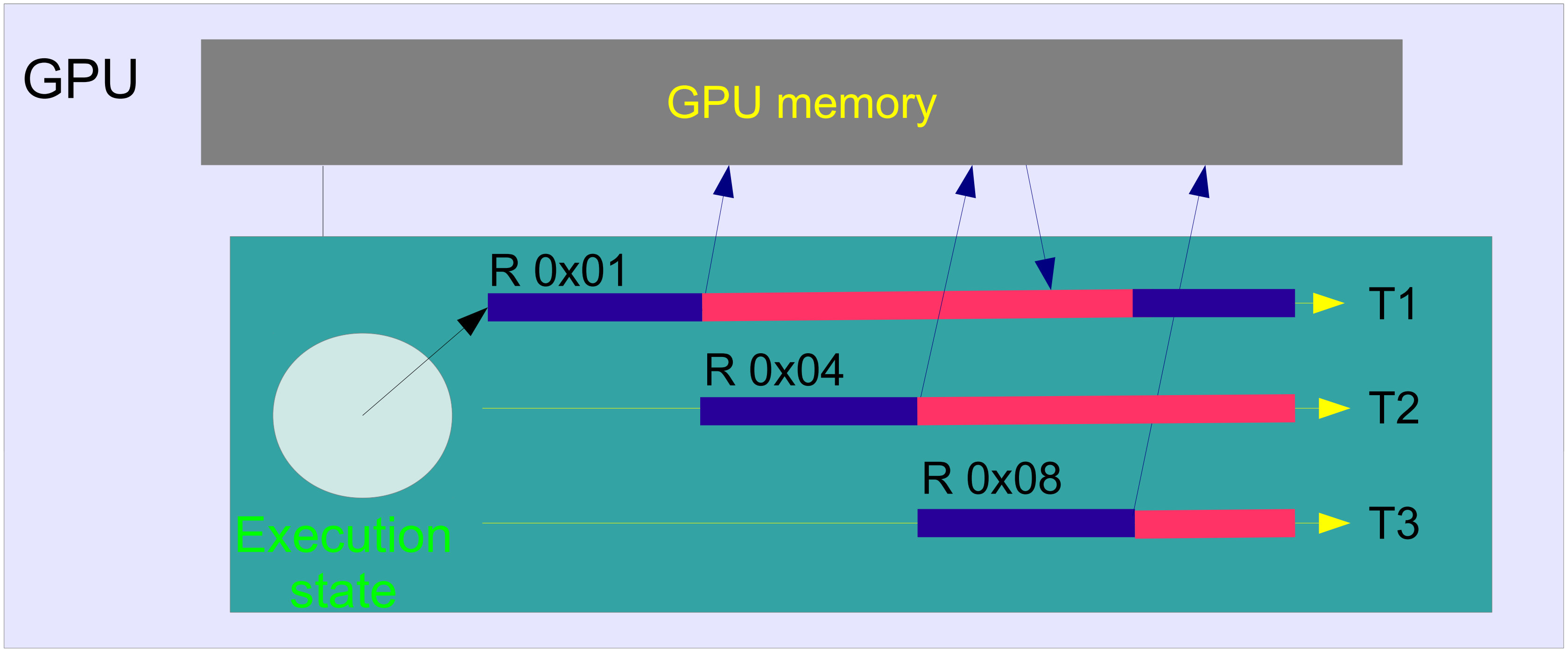
GPU Hardware
3. Parallelism for latency hiding



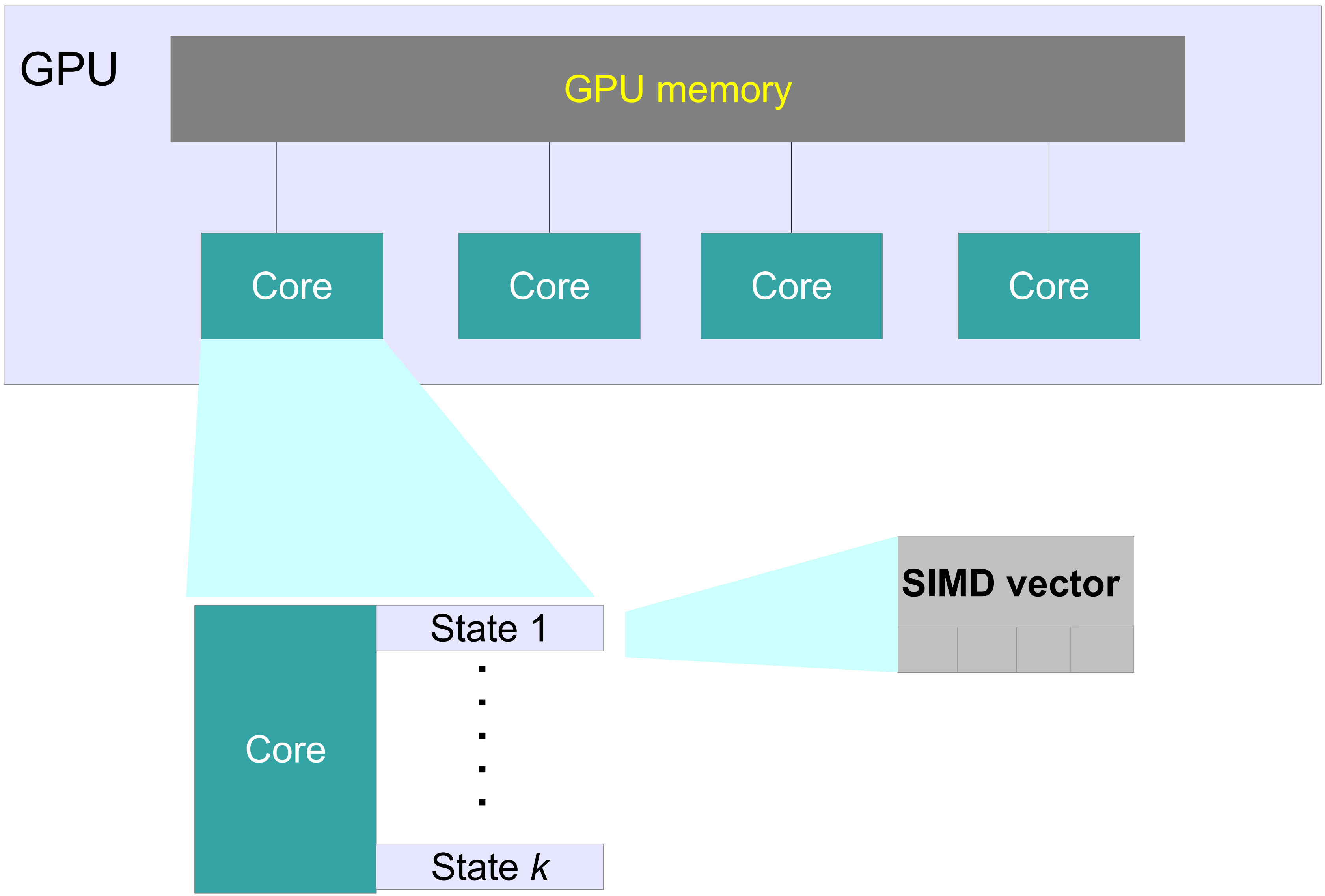
GPU Hardware
3. Parallelism for latency hiding



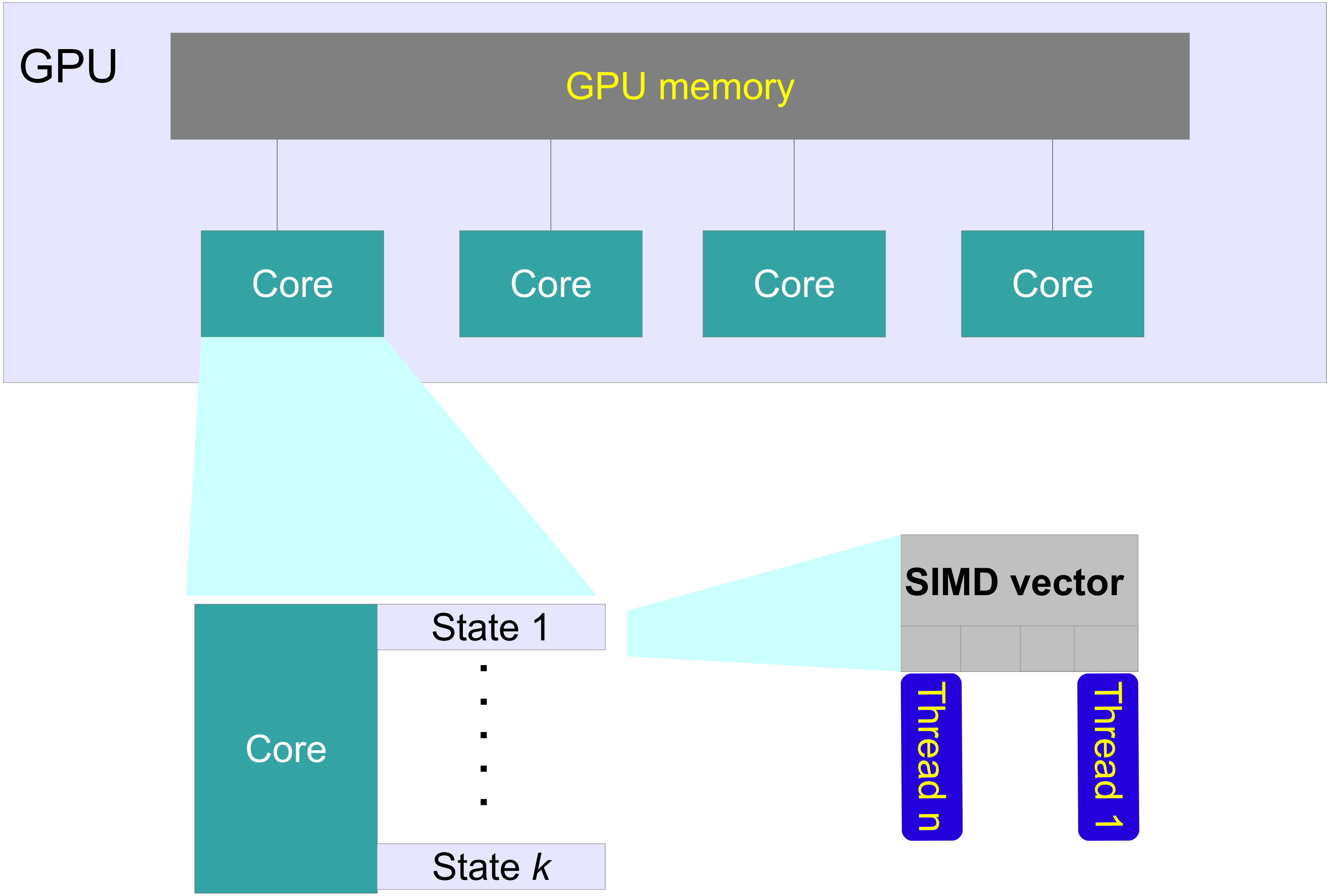
GPU Hardware
3. Parallelism for latency hiding



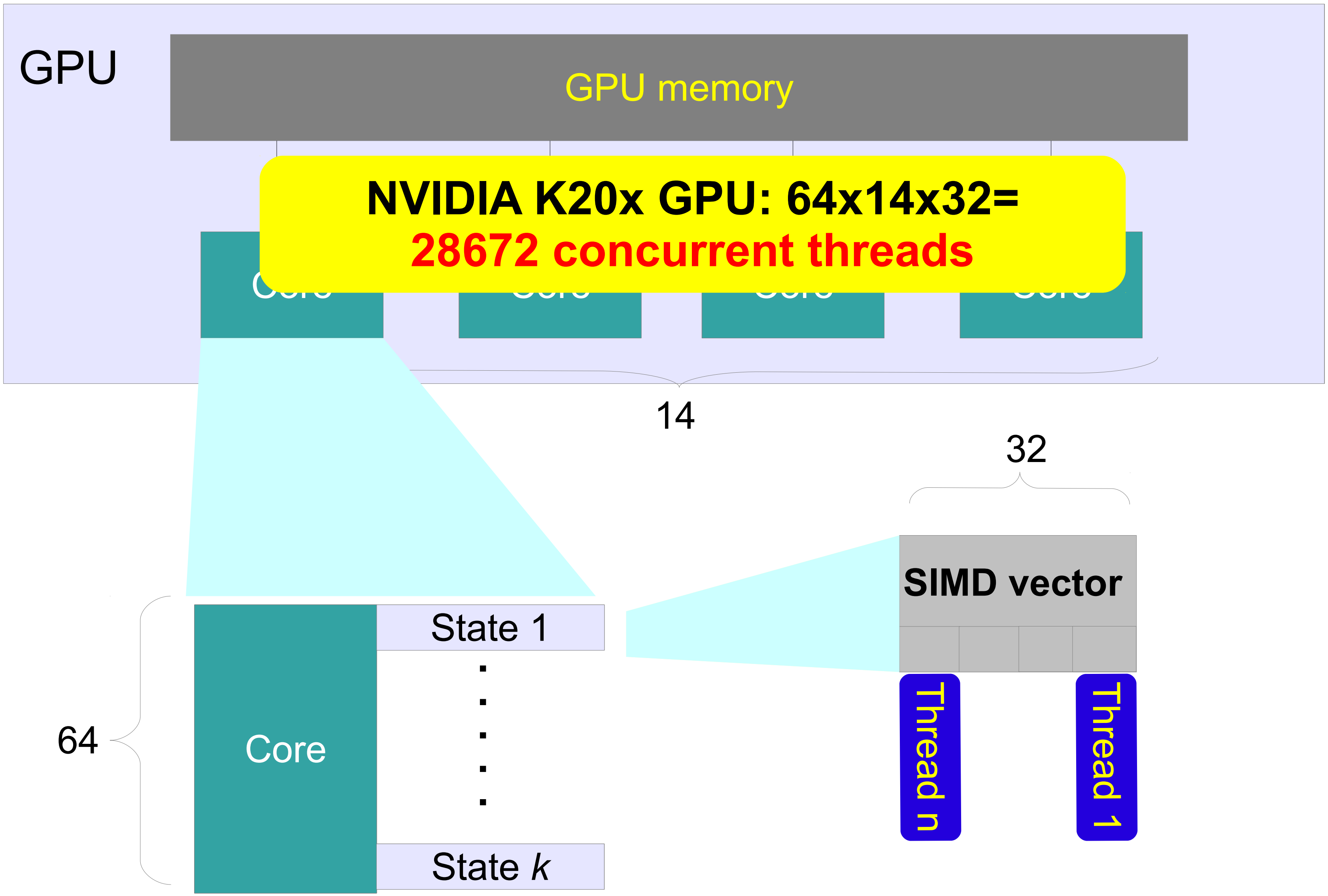
Putting it all together: 3 levels of hardware parallelism



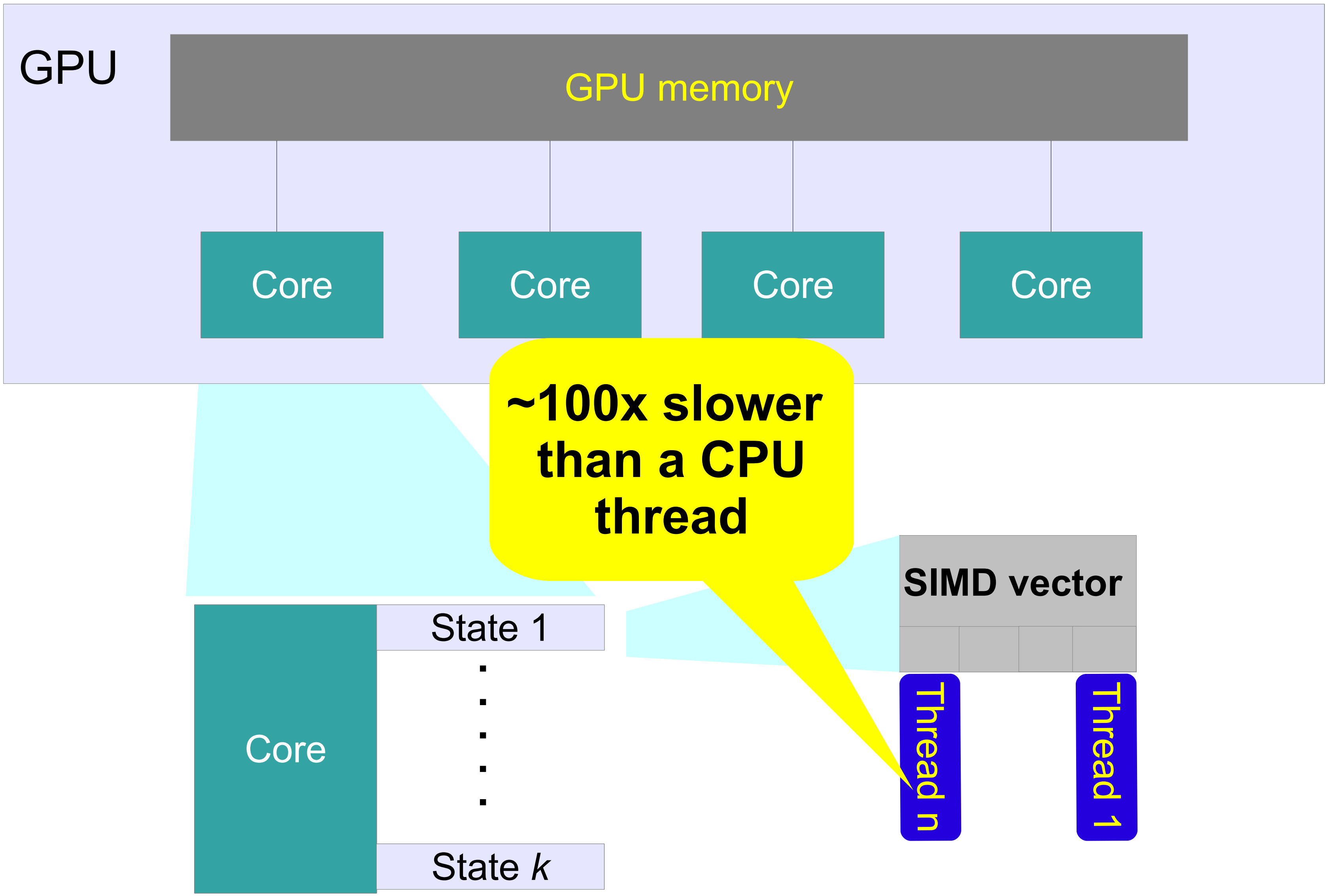
Software-Hardware mapping



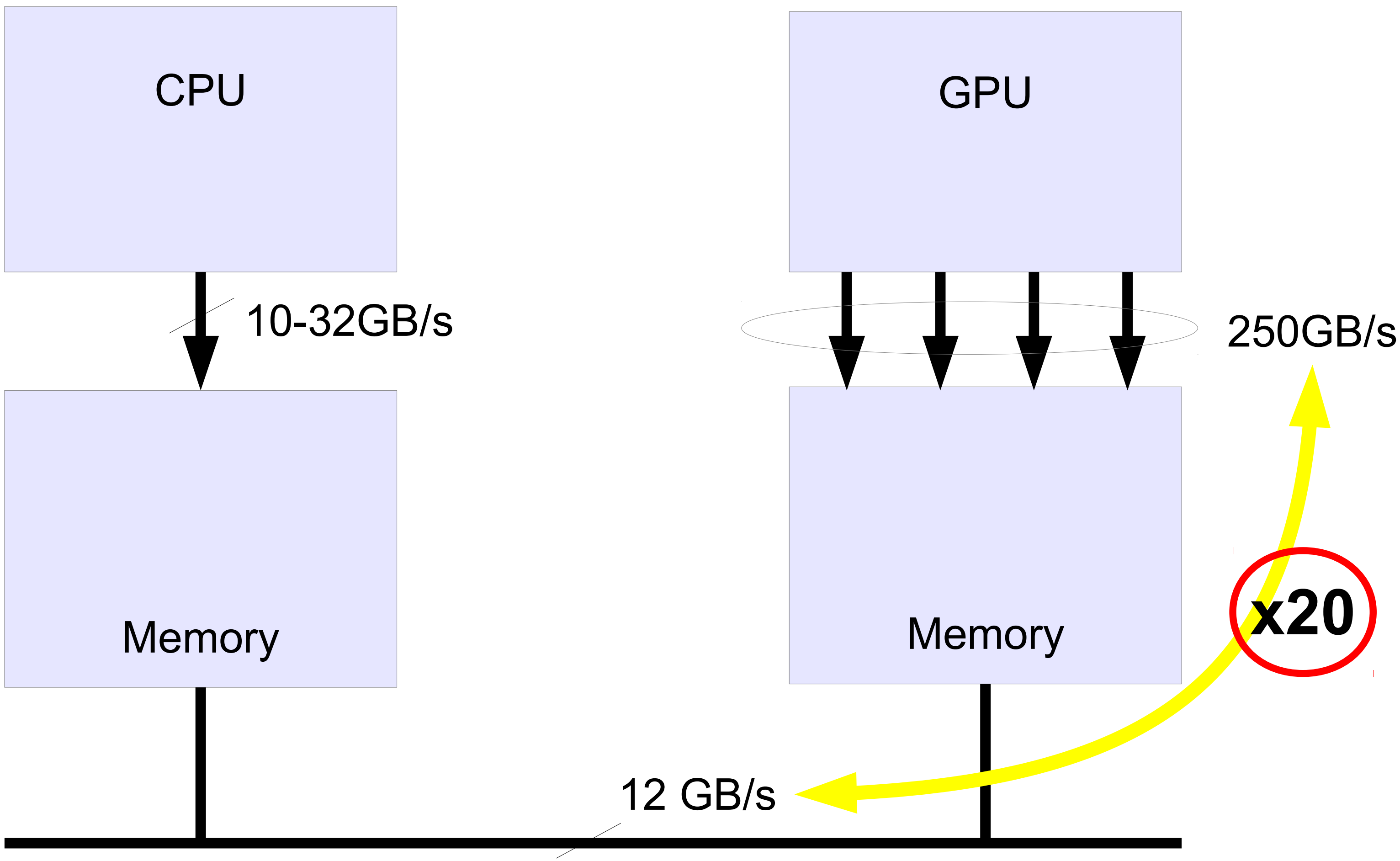
(1) 10,000-s of concurrent threads!



(2) Each thread is slow



(3) Heterogeneous memory



GPUfs: file system layer for GPUs

Joint work with
Bryan Ford, Idit Keidar, Emmett Witchel
[ASPLOS13, TOCS14]

GPUfs: principled **redesign** of the **whole** file system stack

- **Modified FS API semantics** for massive parallelism
- **Relaxed distributed FS consistency** for non-uniform memory
- **GPU-specific implementation** of synchronization primitives, read-optimized data structures, memory allocation, ….

GPU program using GPUfs

```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,buffer);

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

GPU program using GPUfs

```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,buffer);

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

Supporting GPU programming idioms

GPU program using GPUfs

```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,buffer);

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

Parallel API calls: hundreds of threads perform the *same* call in lockstep

GPU program using GPUfs

```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,buffer);

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

open is cached on GPU

GPU program using GPUfs

```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

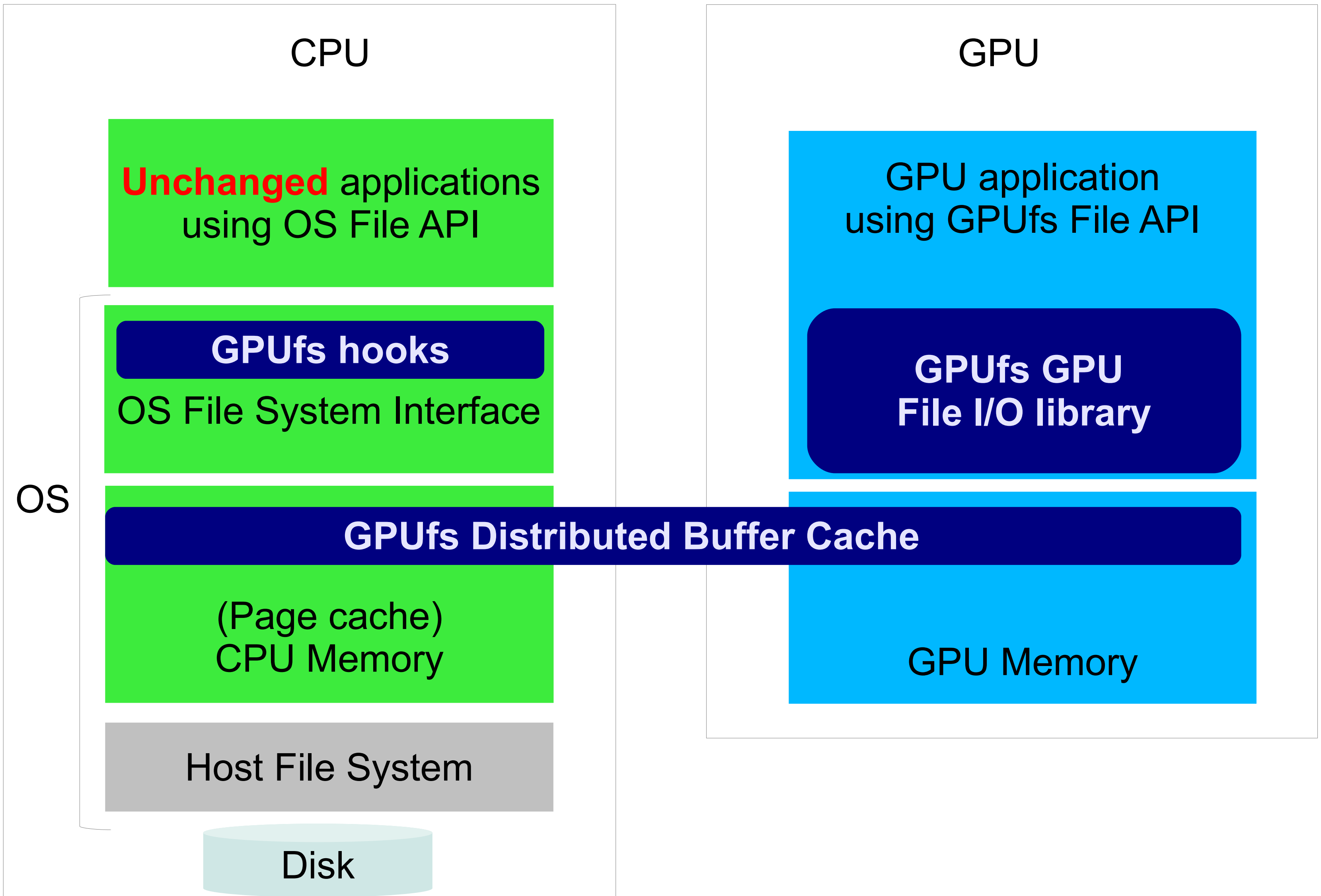
read/write:
explicit offsets to
for parallel access
and low contention

GPU program using GPUfs

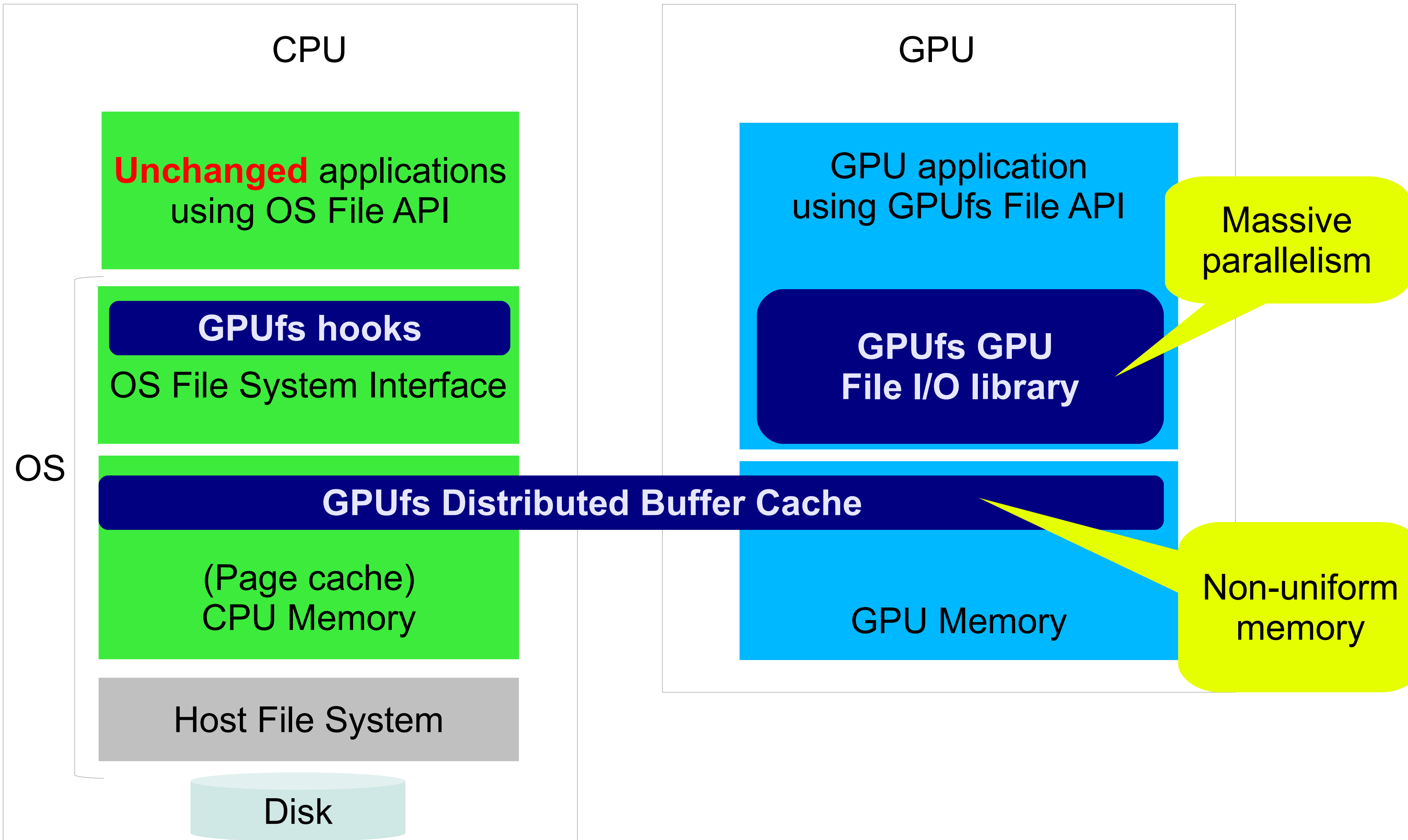
```
__shared__ float buffer[1024];

int fd=gopen(filename,O_GRDWR);

gread(fd,offset,1024*4,buffer);

buffer[myId]=compute(buffer[myId]);// parallel compute

gwrite(fd,offset,1024*4,buffer);

gclose(fd);
```

Asynchronous close

High-level design



High-level design

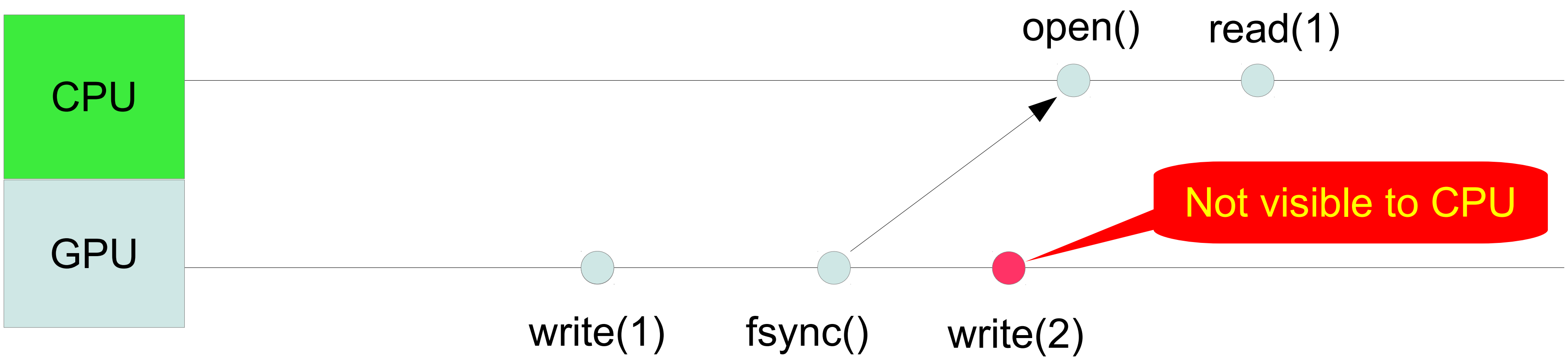


Buffer cache semantics

Local or Distributed file system data consistency?

Weak data consistency model

- close(sync)-to-open semantics (AFS)



Reason

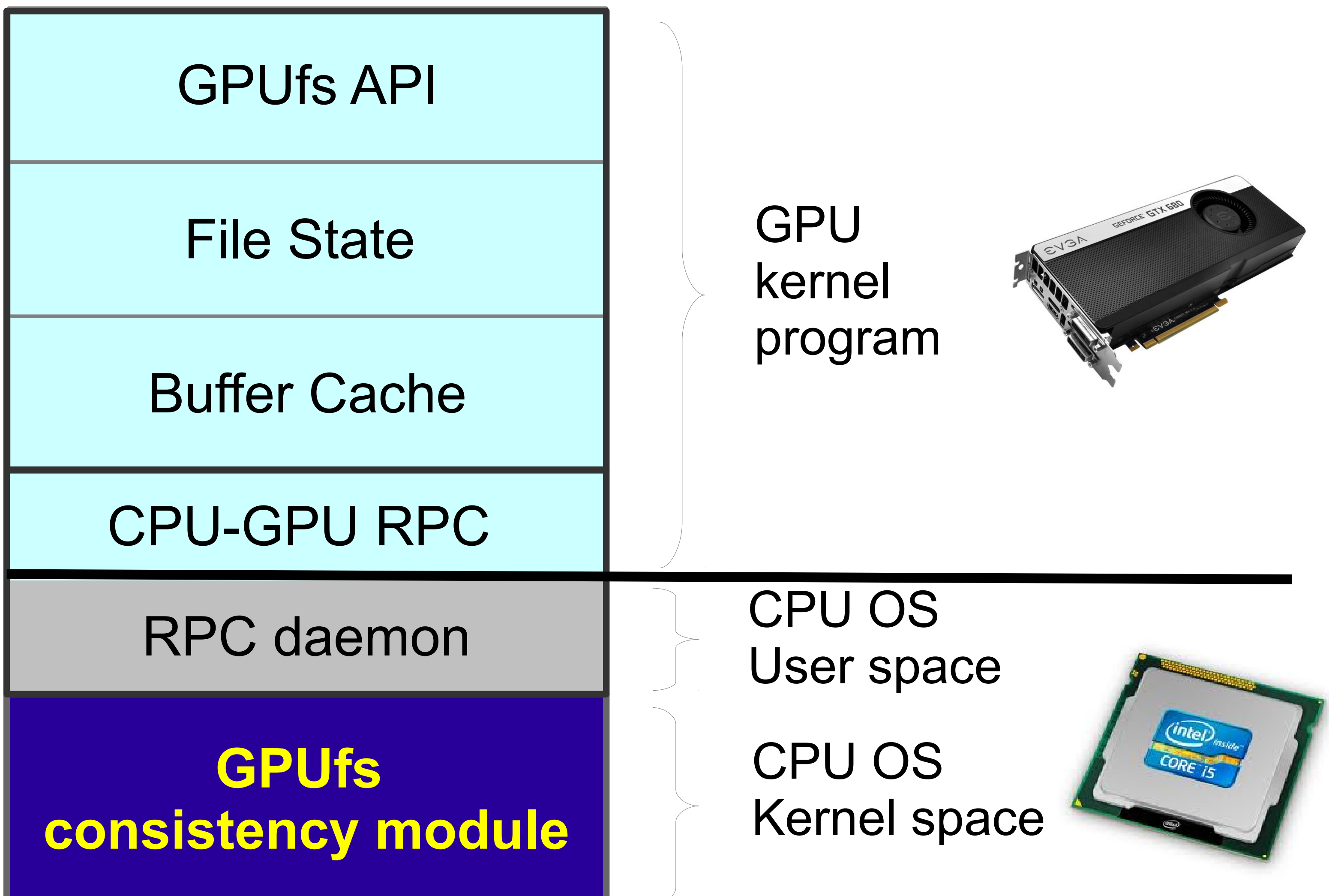
Minimize inter-processor synchronization

Implications

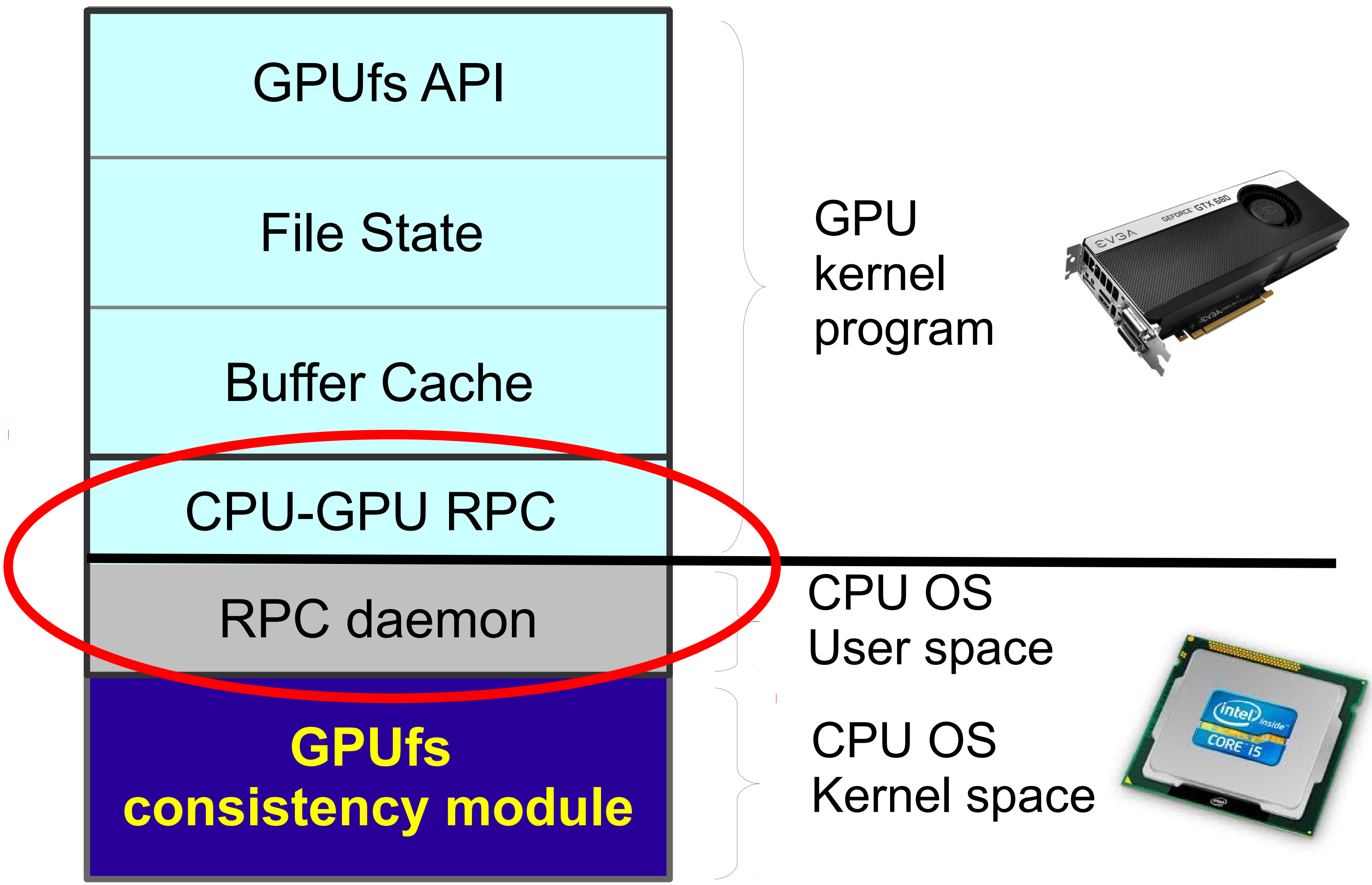
- Overlapping writes
- Cache page false sharing
- Consistency protocol

Implementation bits

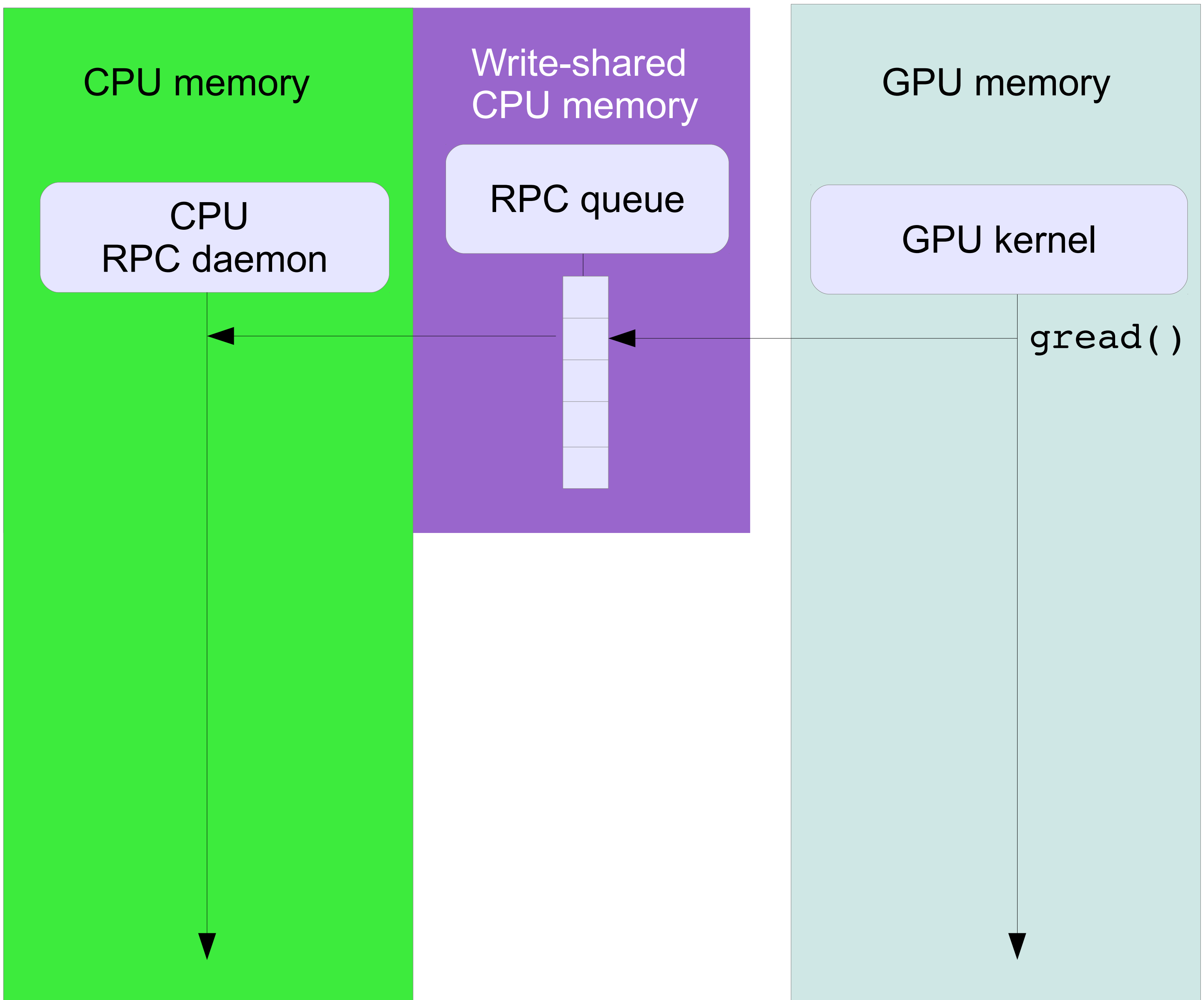
GPUfs prototype



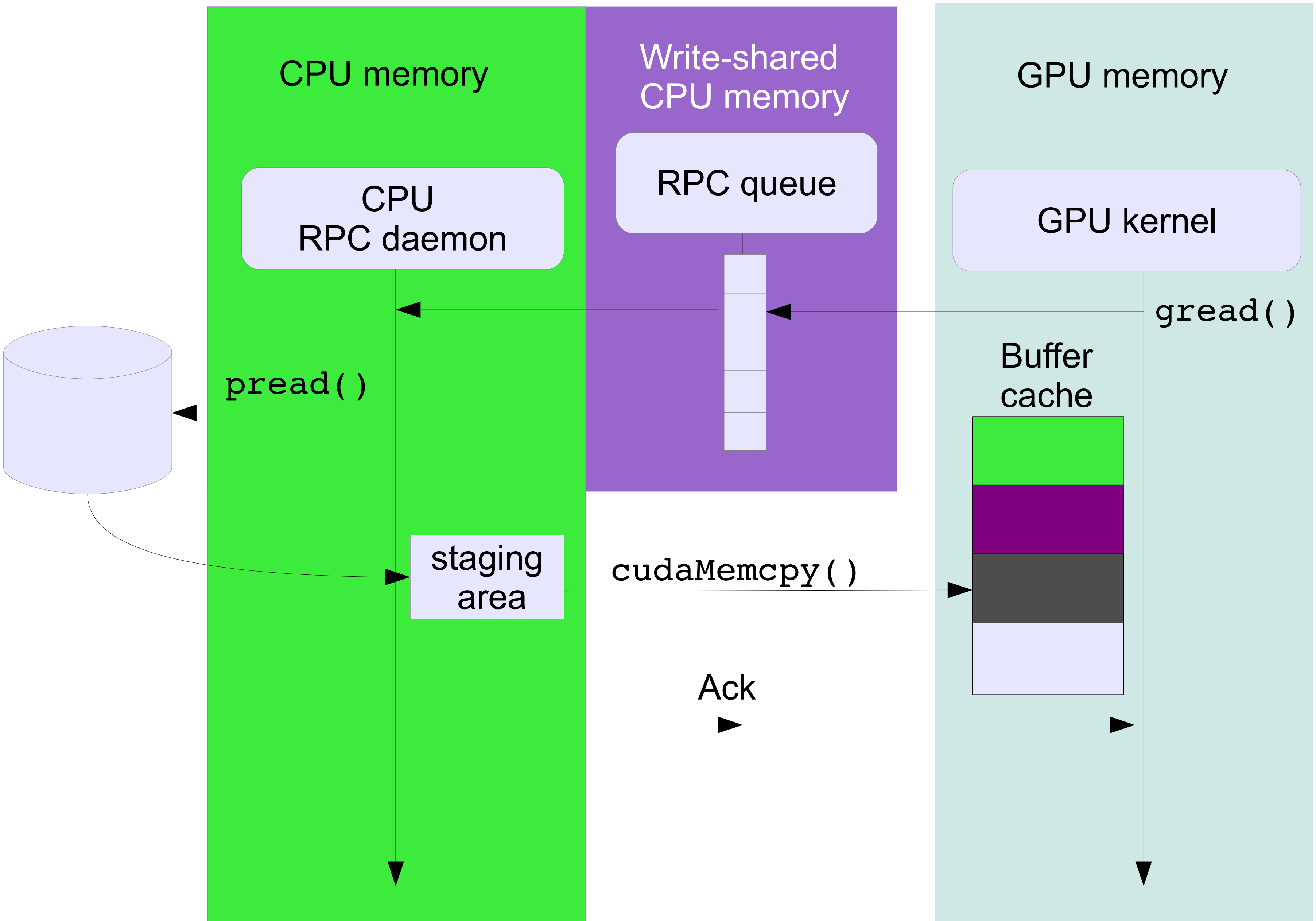
GPUfs prototype



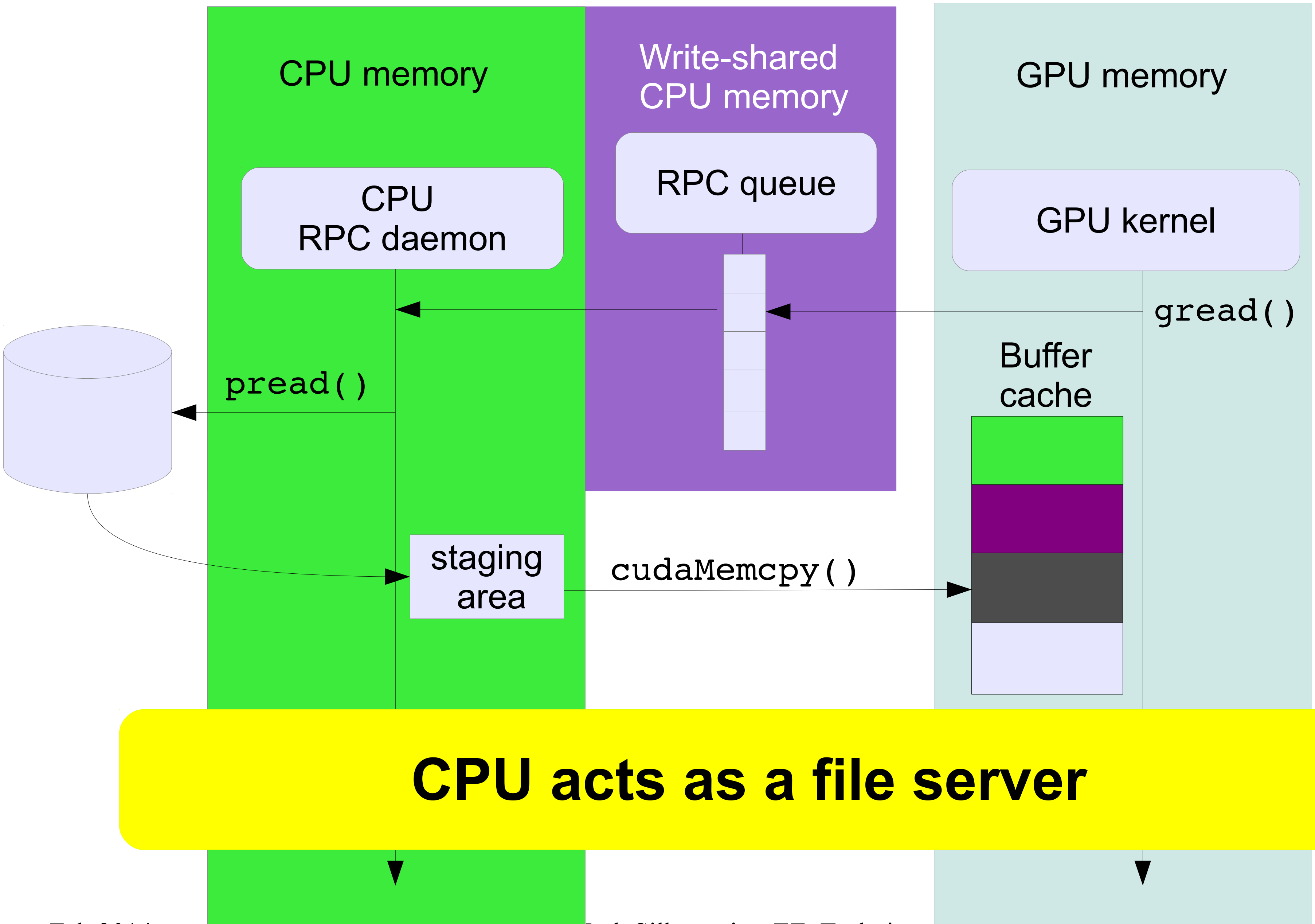
On-demand data transfer



On-demand data transfer



On-demand data transfer



CPU acts as a file server

More implementation challenges

In the paper

- Paging
- Dynamic data structures and memory allocators
- Lock-less read-optimized radix tree
- Inter-processor consistency

GPUfs impact on GPU programs

- Memory overhead
- Very little CPU involvement

Pay-as-you-go design

Evaluation

All benchmarks are written with a GPU self-contained kernel – **no CPU part**

Real applications

- Approximate image matching
  - 4 GPUs – 6-9**x** faster than 8 CPU cores

- String matching in Linux kernel tree: 33,000 files
  - 1 GPU – **6 - 7x** faster than 8 CPU cores
  - GPUfs overhead = 7%

Summary - GPUfs

GPUfs is a first system to provide I/O for GPUs

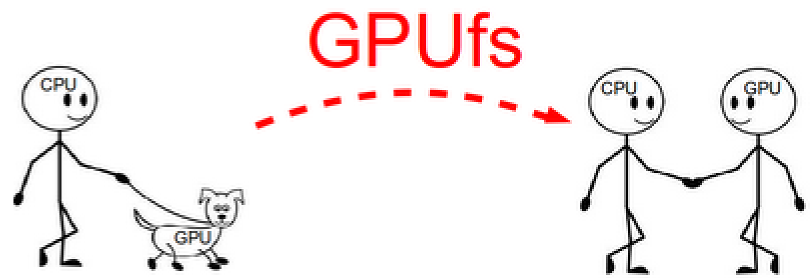
Open issues

- Buffer cache: consistency, CPU page cache interaction, page faults, mmap
- Direct access to storage devices
- Optimizing file naming mechanisms
- Applications
  - Image format readers, git-grep
- Other accelerators – FPGAs, Xeon-Phi, DSP
- GPU networking

Summary

- System performance will rely on accelerators
- Programmable accelerators are peer-processors (not co-processors)
- They need I/O abstractions and OS services
- GPUnet, GPUfs – first step in this direction

Set GPUs free!



Interested in a project? Talk to me

046274: Spring 2014, Mon 16.30
GPU-accelerated systems

mark@ee.technion.ac.il