Omnix: an accelerator-centric OS architecture for omni-programmable systems

Rethinking the role of CPUs in modern computers

Mark Silberstein

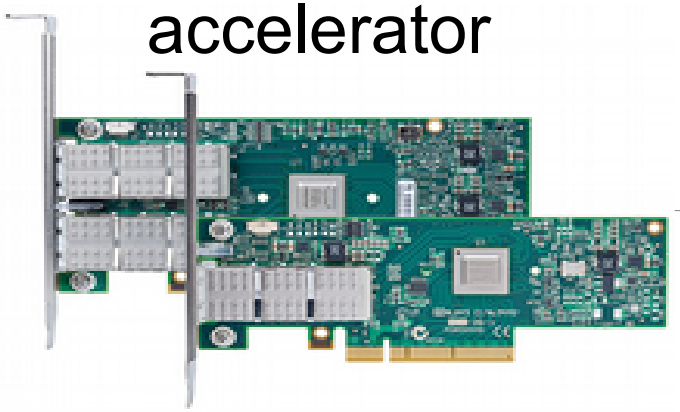
Technion

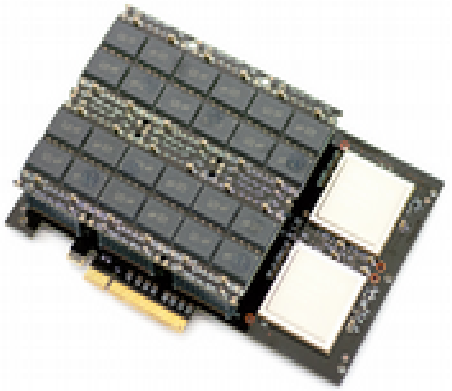
October 2019
EPFL

Computer hardware: circa ~2019

GPU parallel accelerator

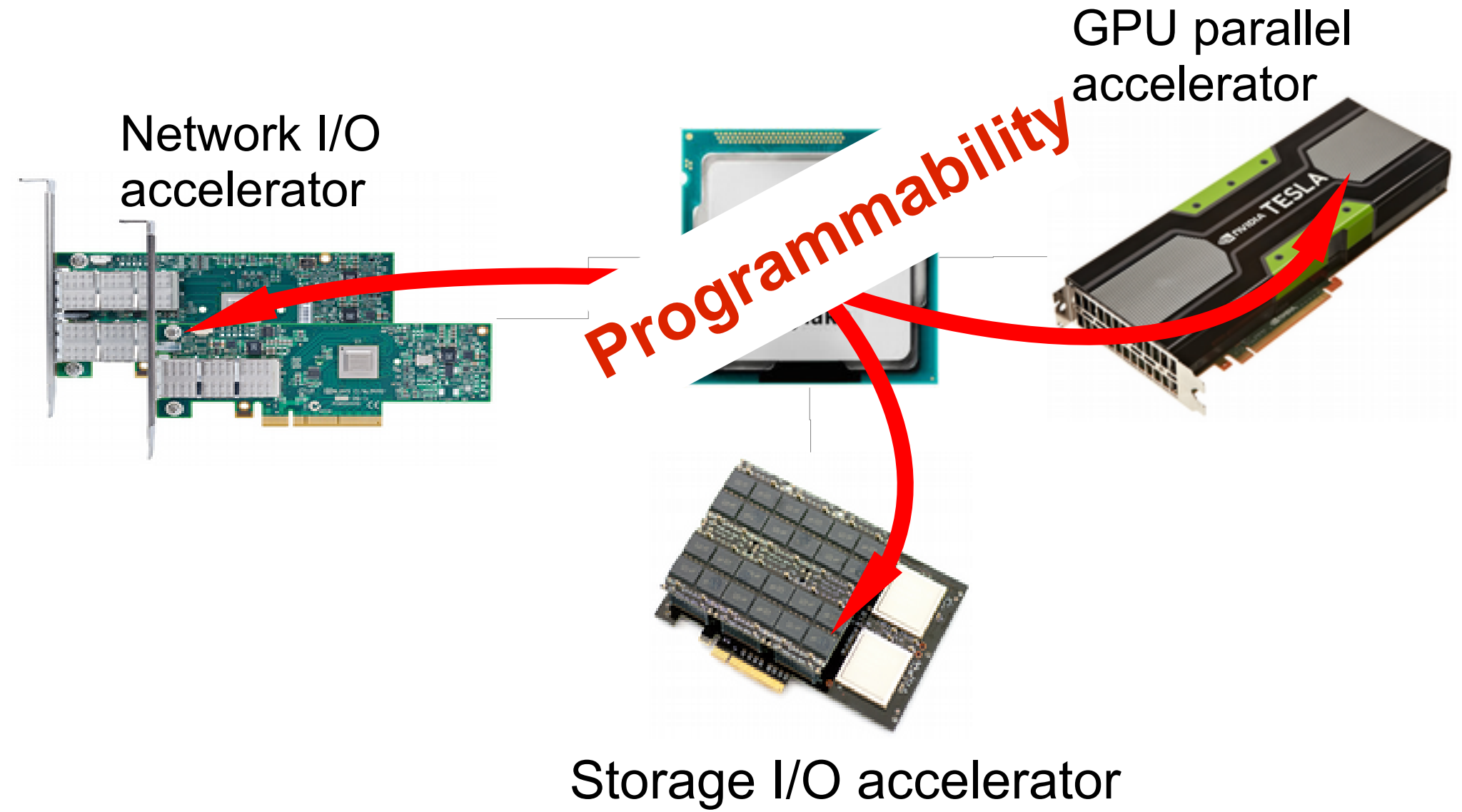
Network I/O accelerator



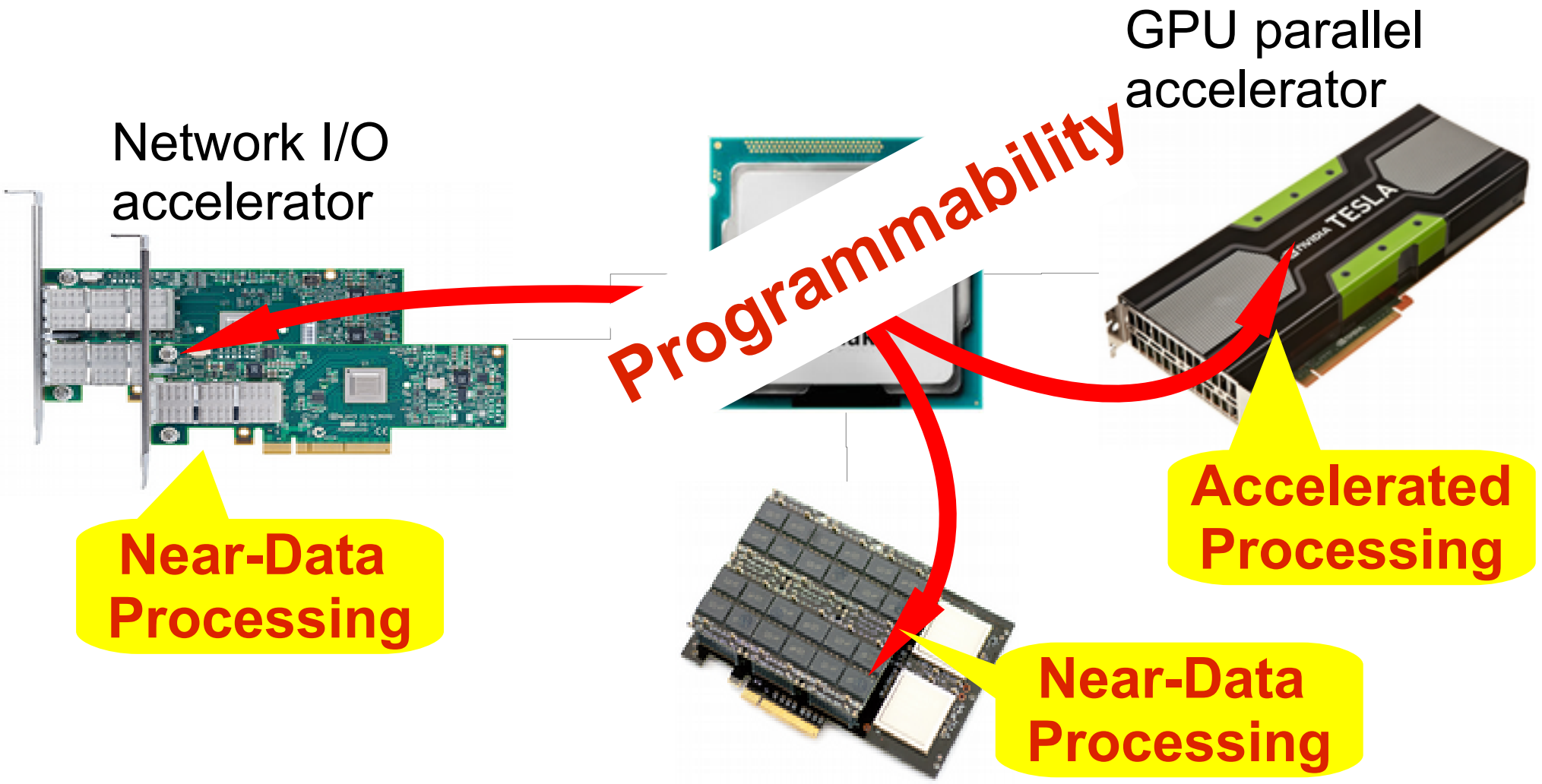


Storage I/O accelerator

Central Processing Units (CPUs) are no longer **Central**

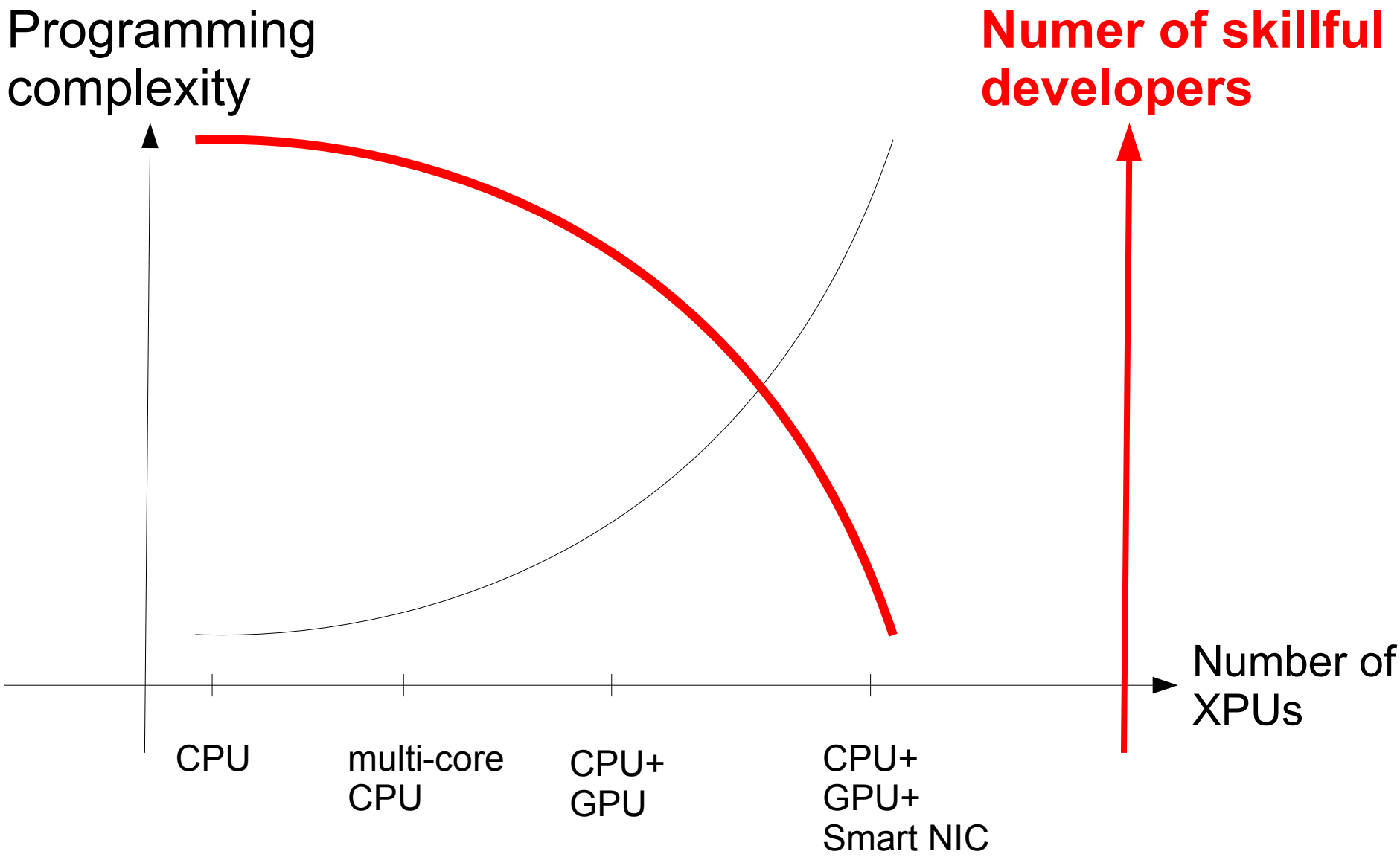


Omni-programmable system
X- Processing Units: XPUs

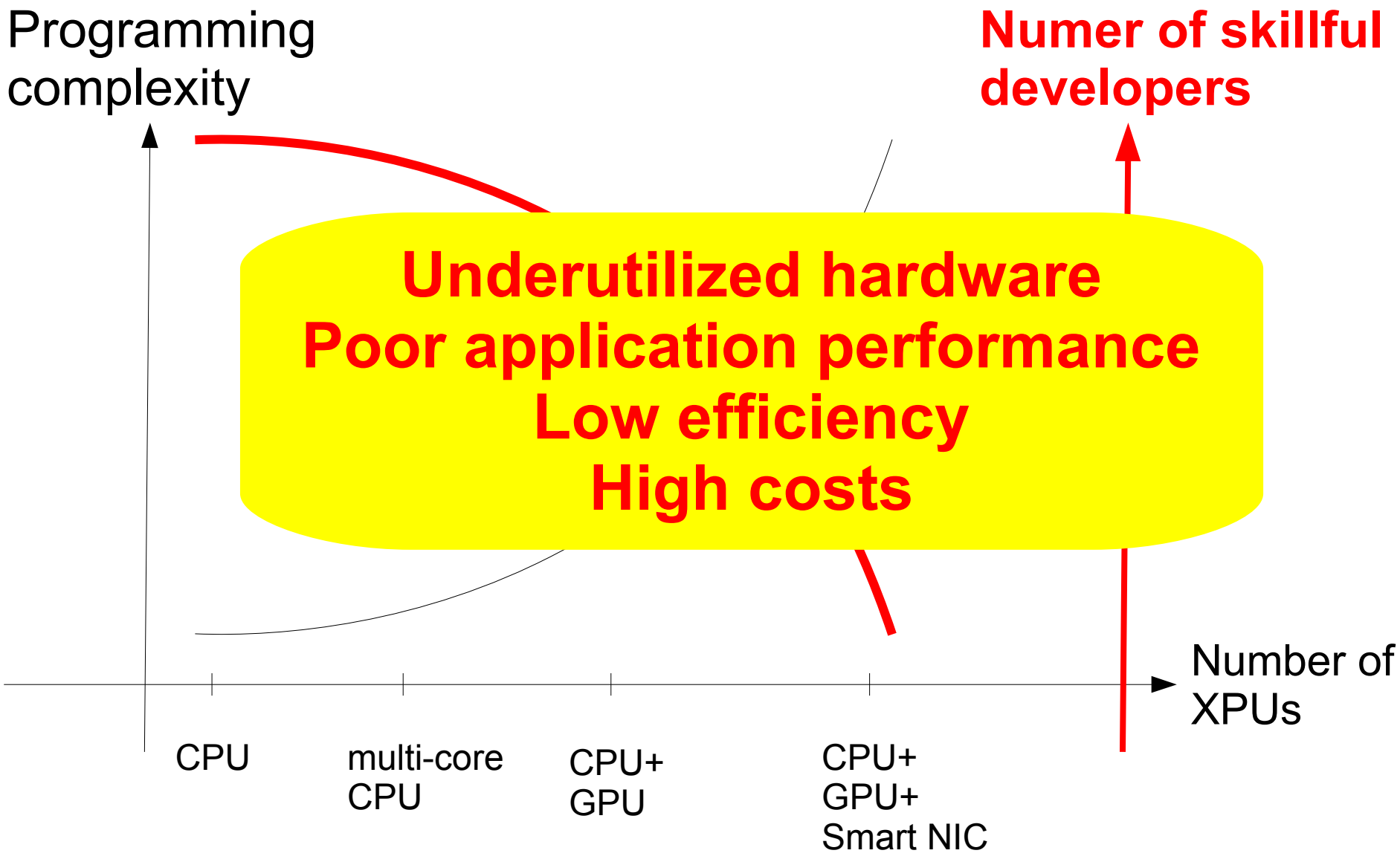


But XPUs also create new walls!

Hard to maintain whole-application efficiency



Hard to maintain whole-application efficiency

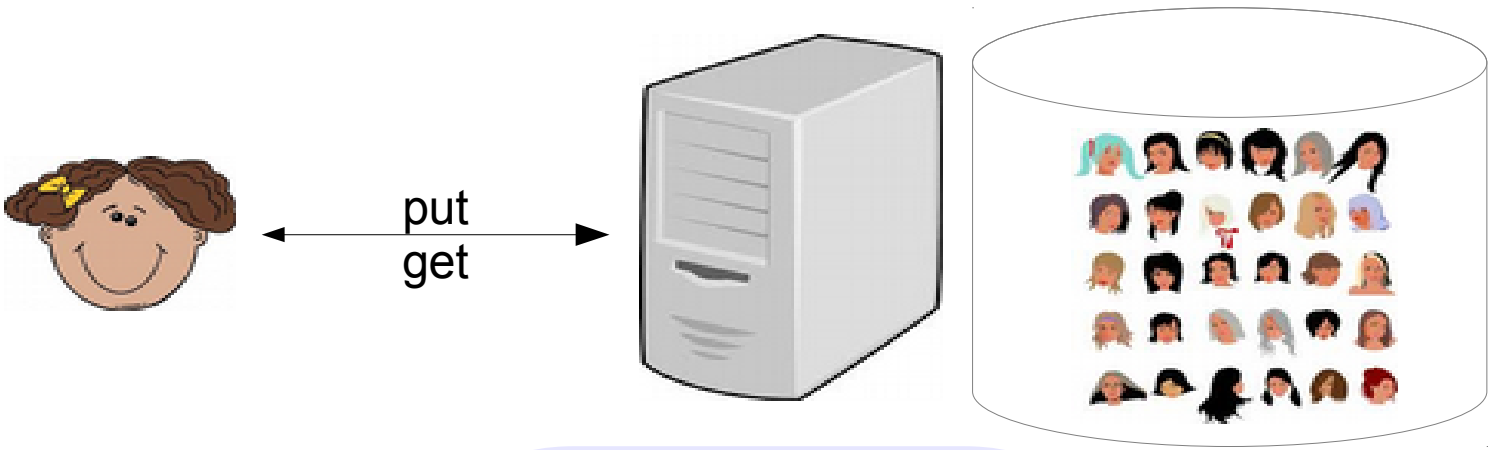


Agenda

- The root cause of the programmability wall
- OmniX: accelerator-centric OS design
  - Principles
  - Examples
  - CPU-less system design
  - OS integration with inline processing

Example: image server

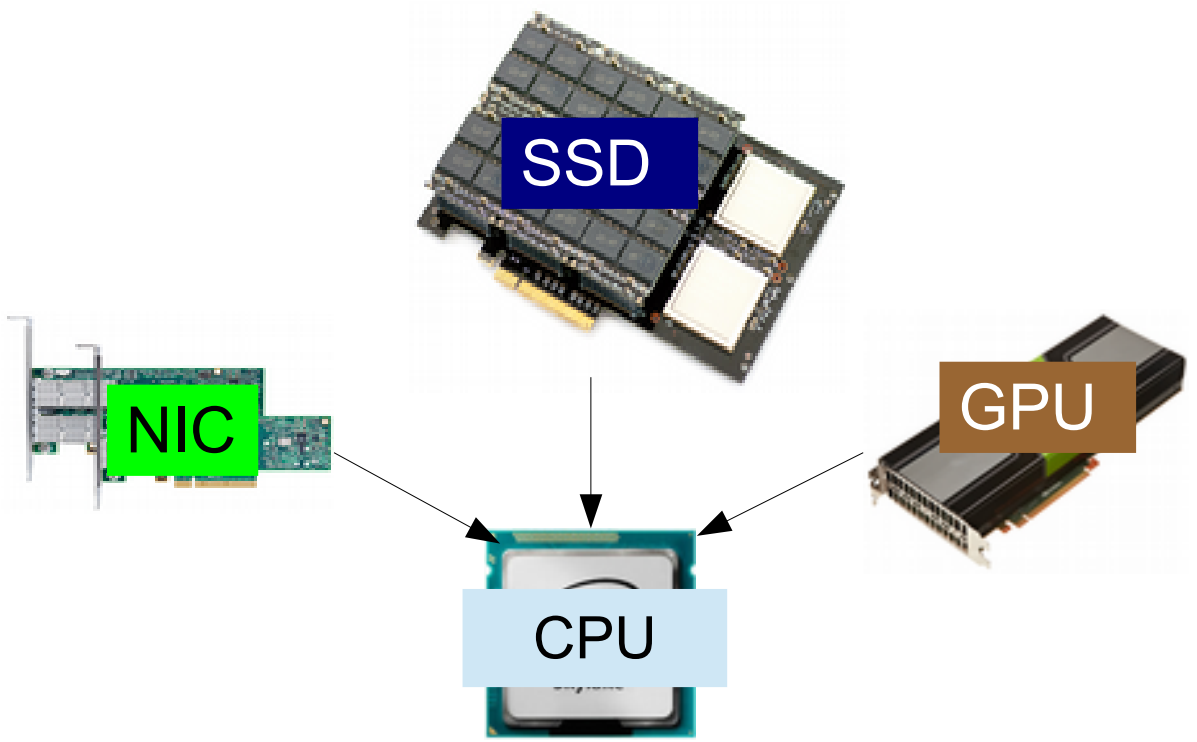
1. put: parse → contrast-enhance → store
2. get: parse → resize → store → marshal



Similar architecture used in Filckr

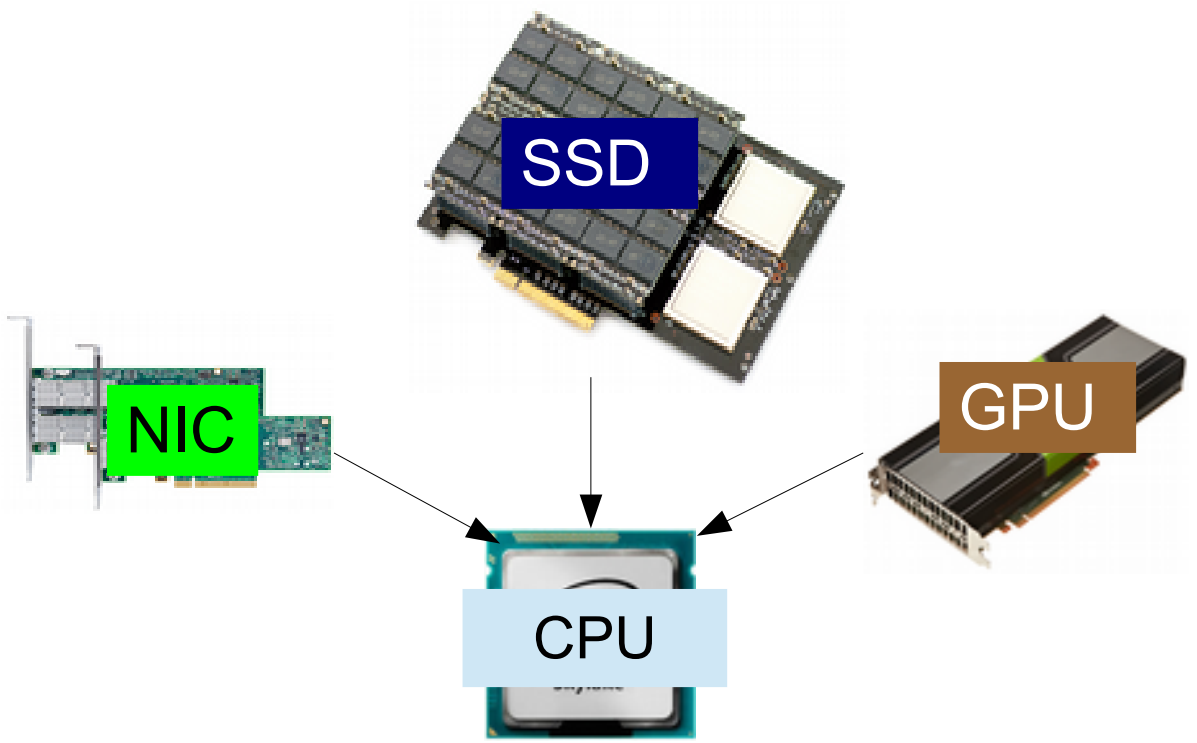
Example: image server

1. put: parse → contrast-enhance → store
2. get: parse → resize → store → marshal



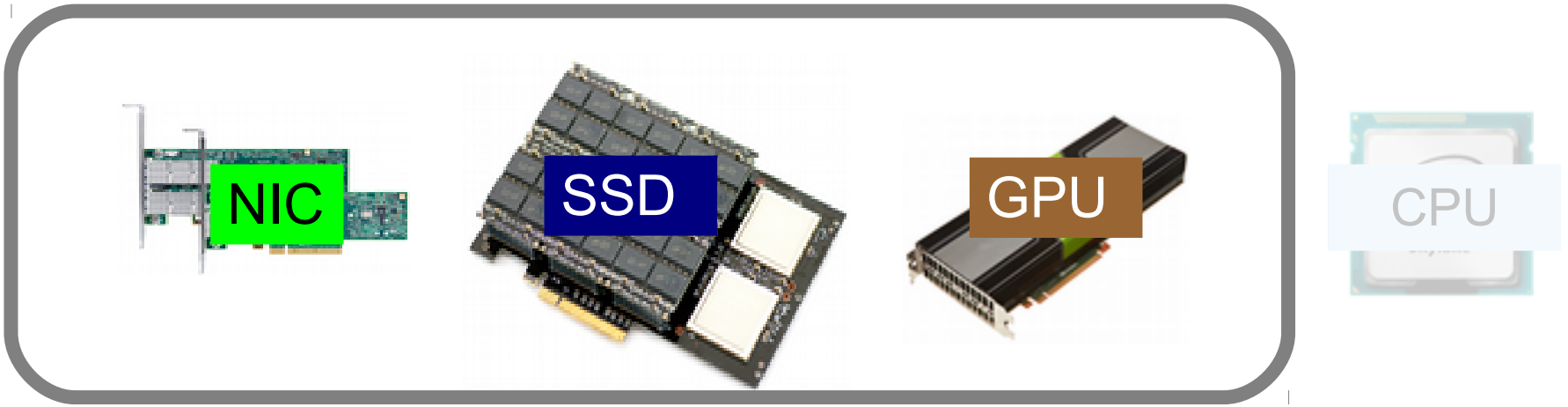
Accelerating with XPUs

1. put: parse → **contrast-enhance** → **store**
2. get: parse → **resize** → **store** → marshal



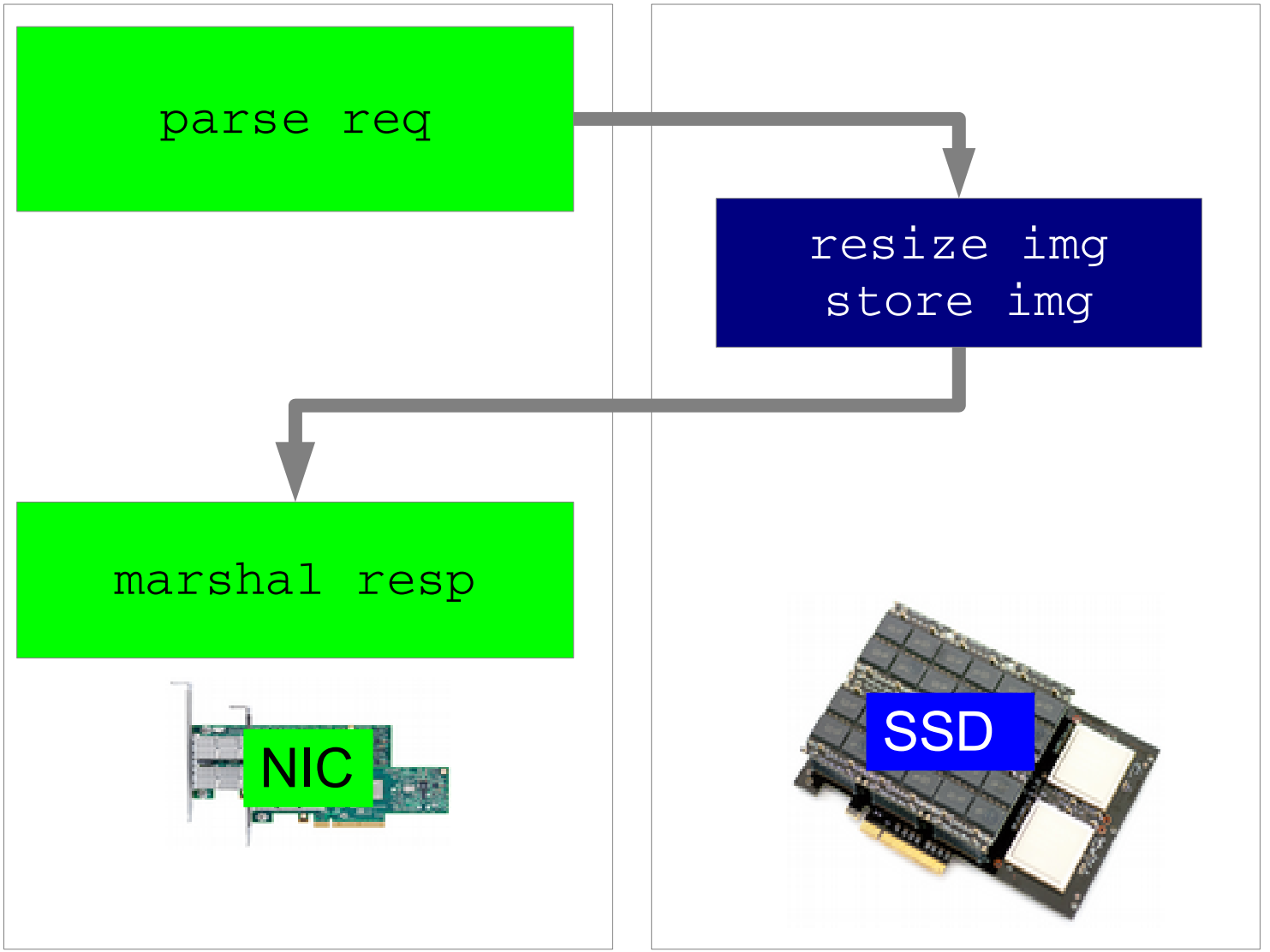
Accelerating with NXUs

1. put: parse → **contrast-enhance** → **store**
2. get: parse → **resize** → **store** → marshal



Closer look at *get*

parse → **resize** → **store** → marshal



OS services run on CPUs

get: parse → **resize** → **store** → marshal

```
parse req
```

```
recv(req)
```

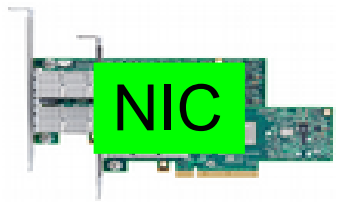
```
read(file,img)
```

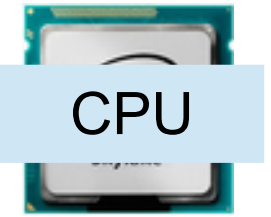
```
resize img
```

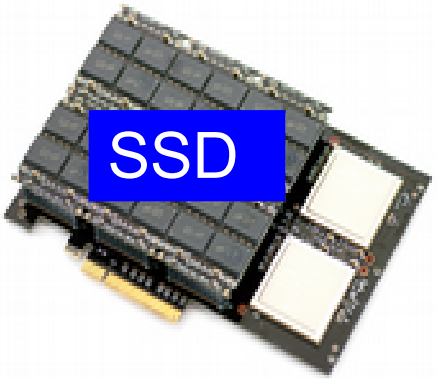
```
marshal resp
```

```
write(file,img)
```

```
send(resp)
```

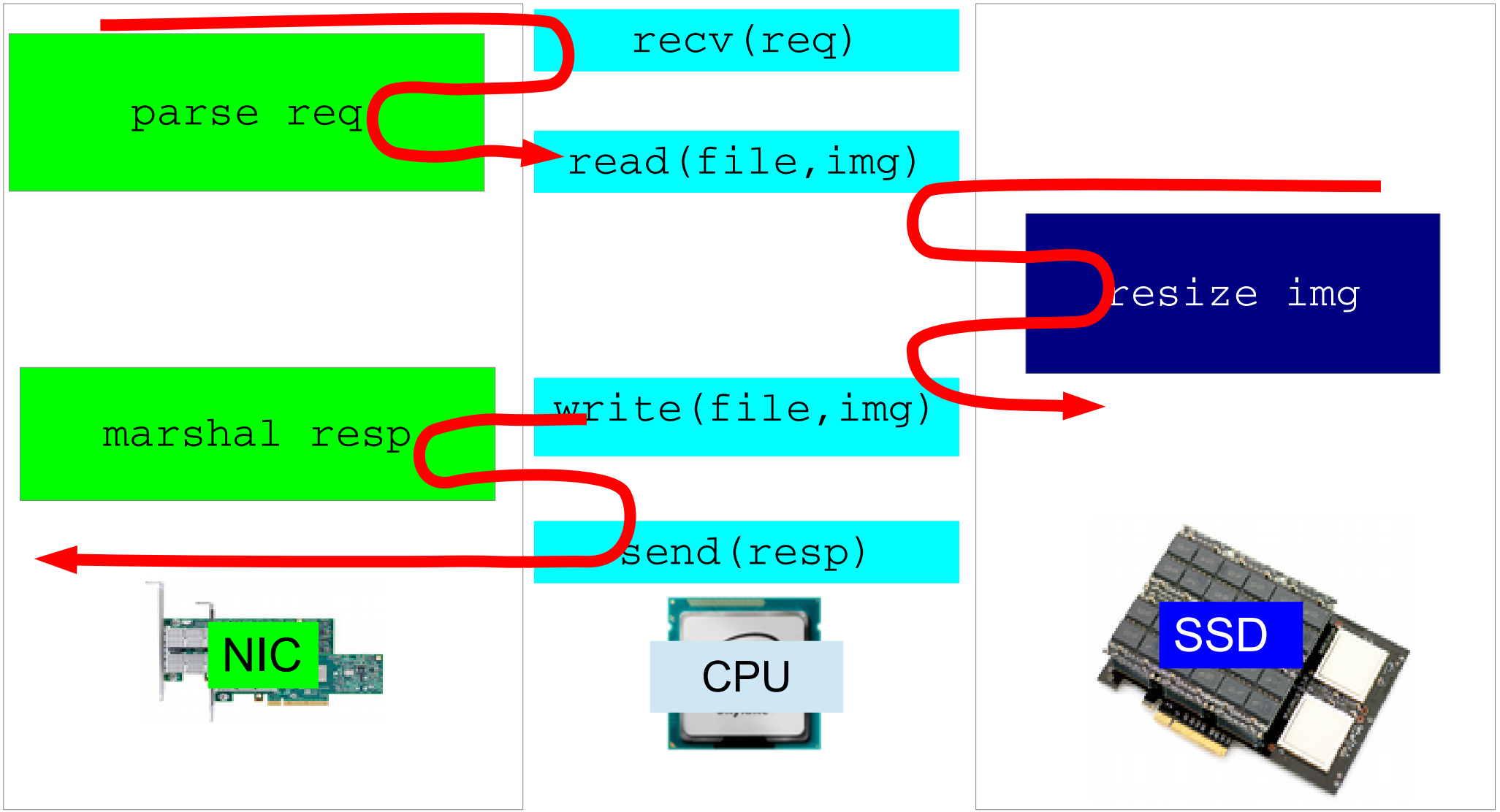






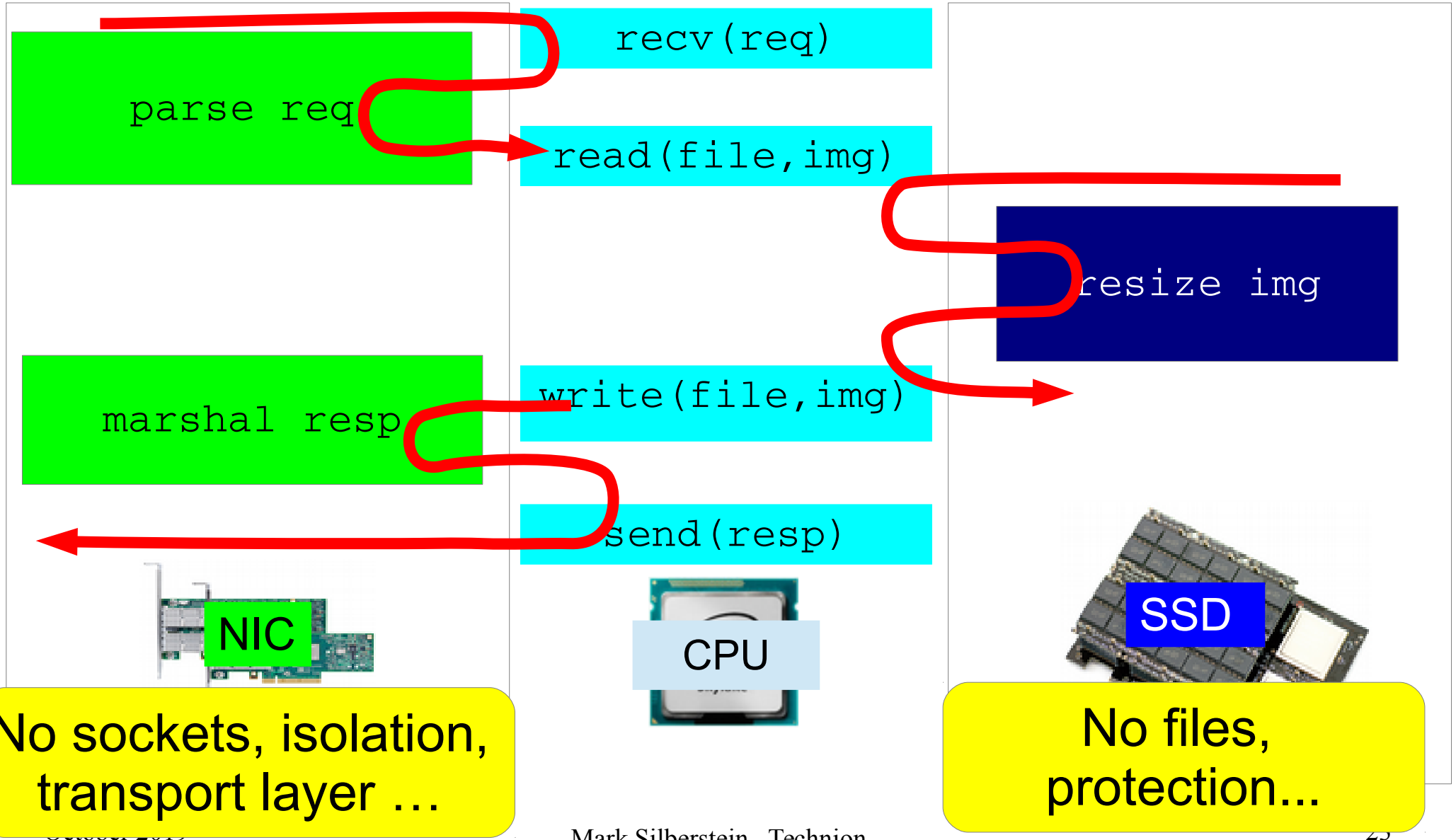
Result: offloading overheads dominate

get: parse → **resize** → **store** → marshal

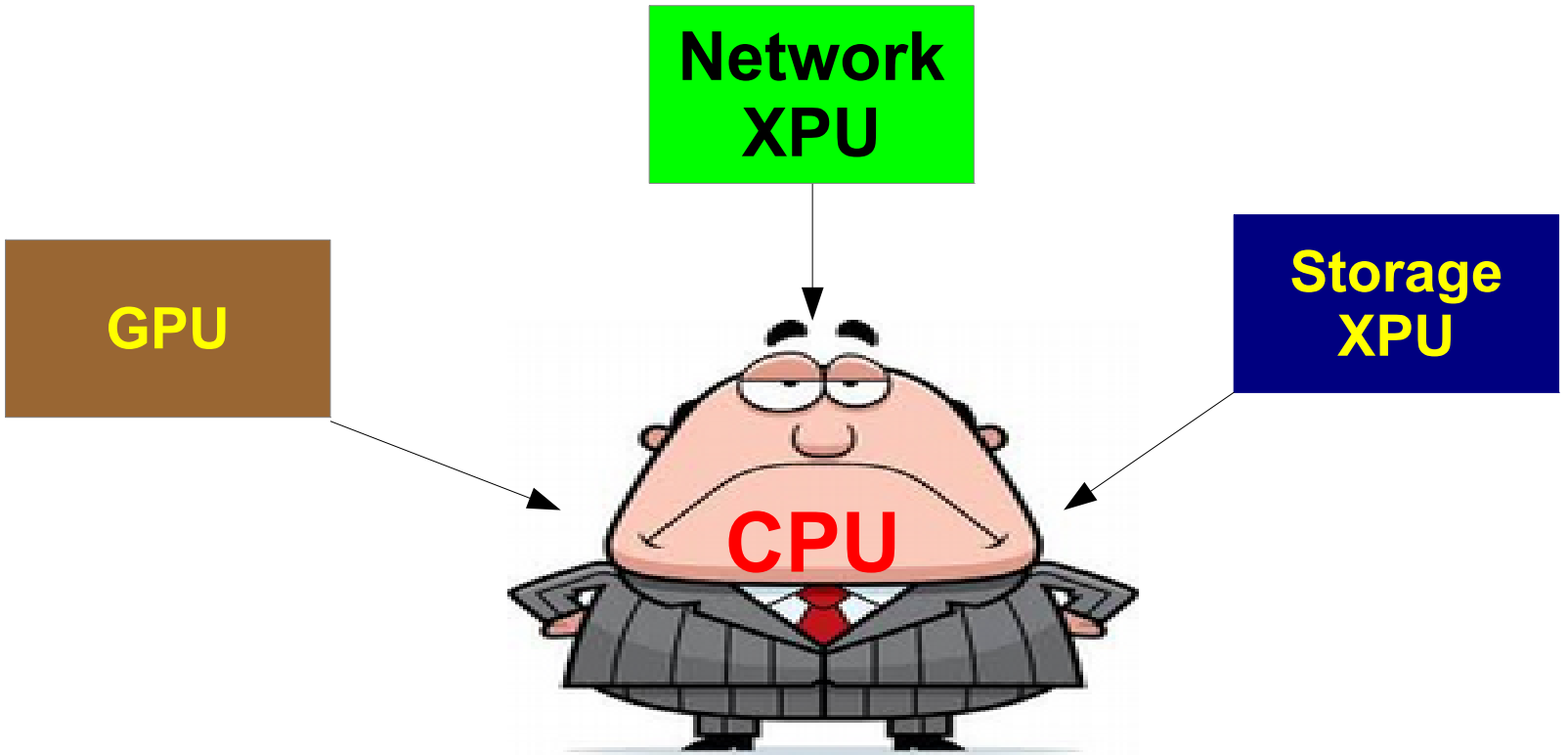


Result: offloading overheads dominate

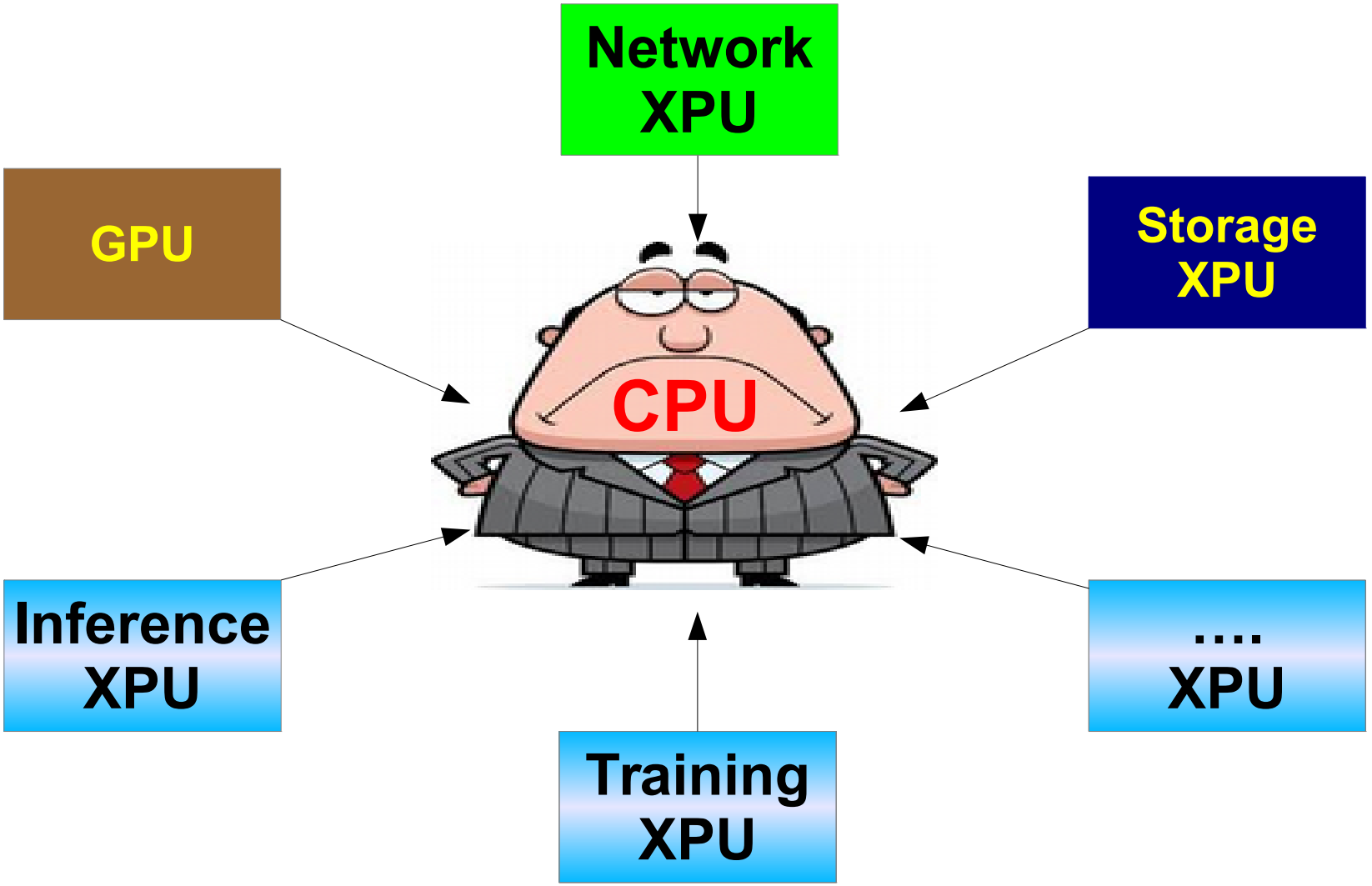
get: parse → **resize** → **store** → marshal



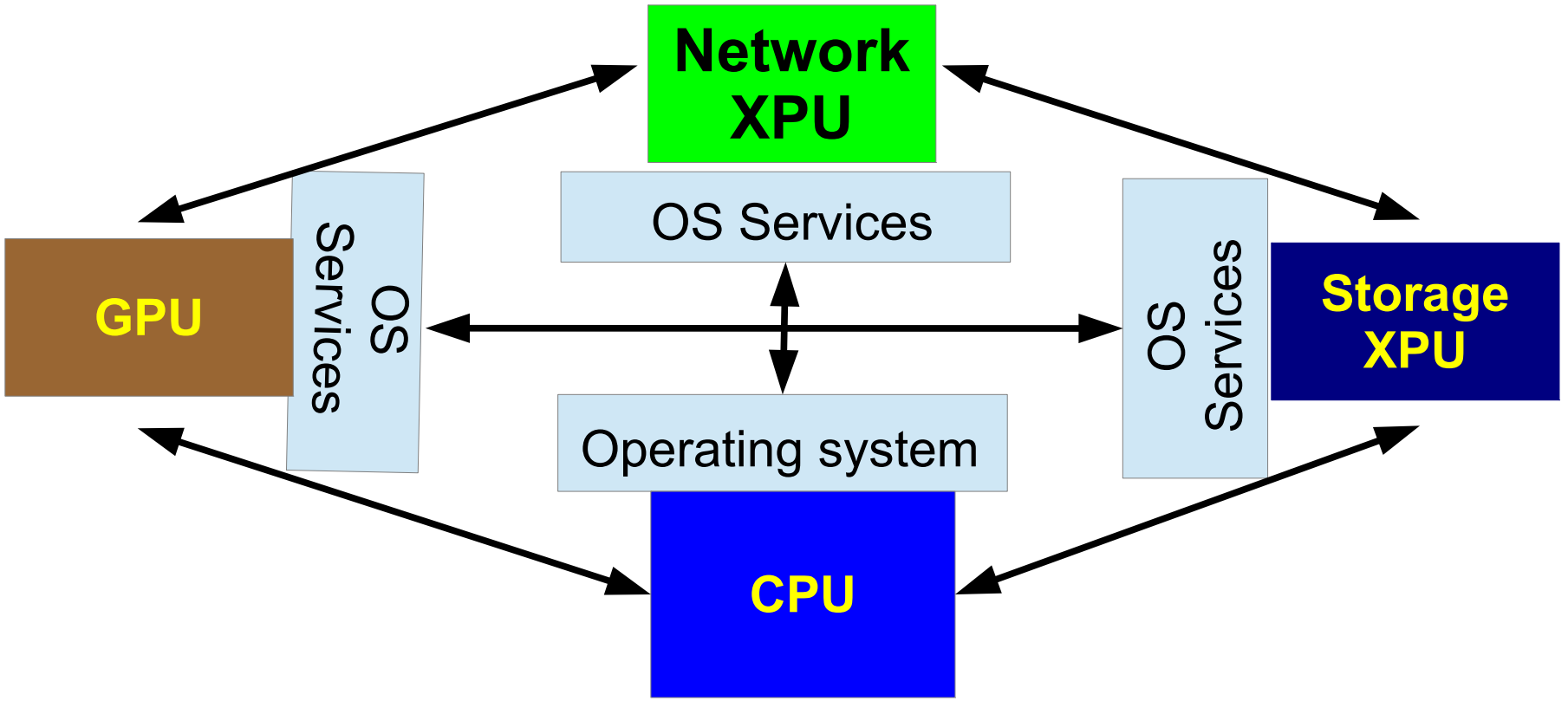
THE problem:
OS architecture is CPU - centric



THE problem ***is general***:
OS architecture is CPU - centric

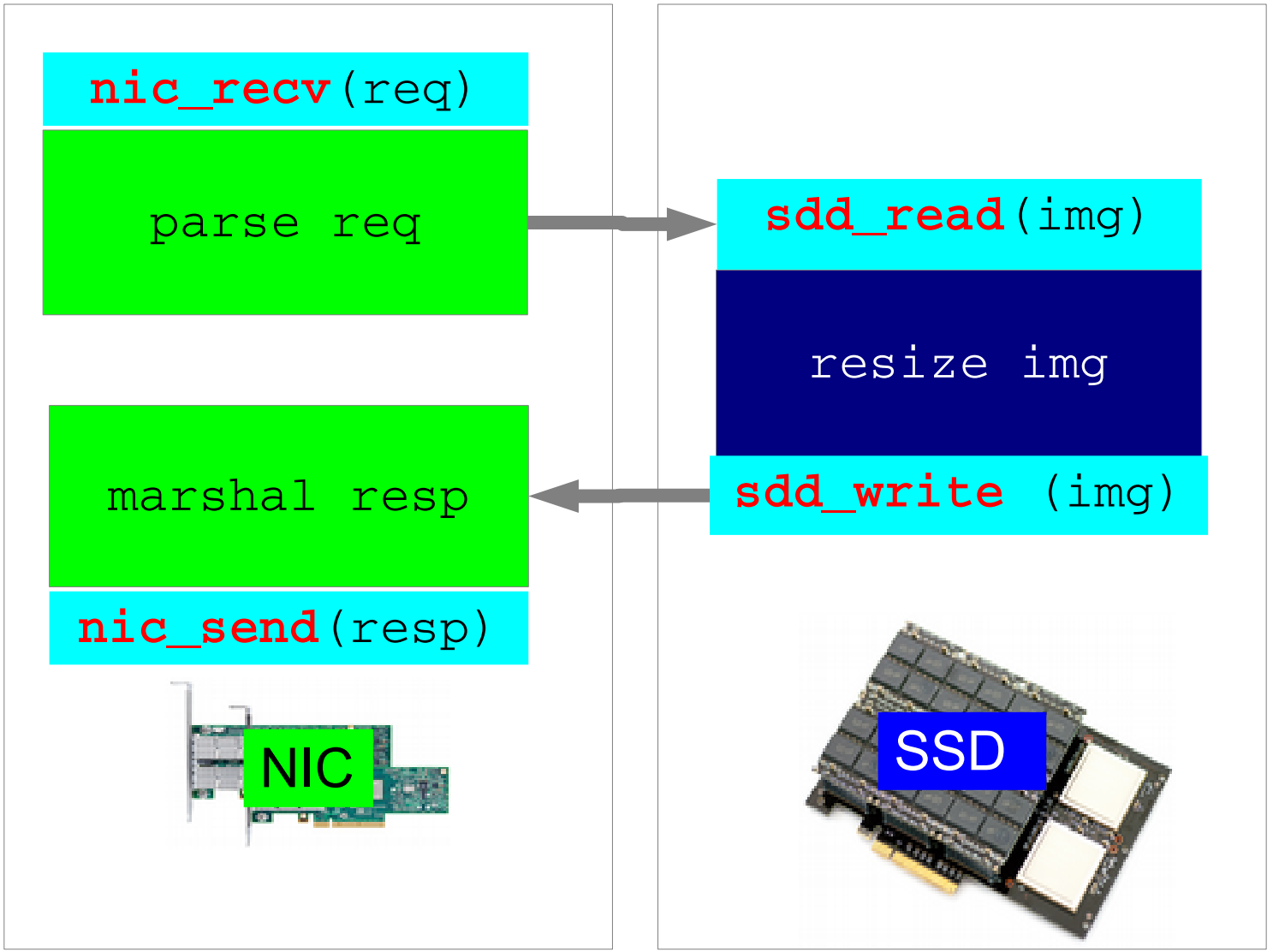


OmniX: accelerator-centric OS architecture



Execution in OmniX

get: parse → **resize** → **store** → marshal



Accelerator-centric OS architecture

Types of OS abstractions for accelerators

Accelerator-centric: no CPU in data/control path

Accelerator-friendly: accelerator-aware host OS

Data-centric: inline near-data processing

Types of OS abstractions for accelerators

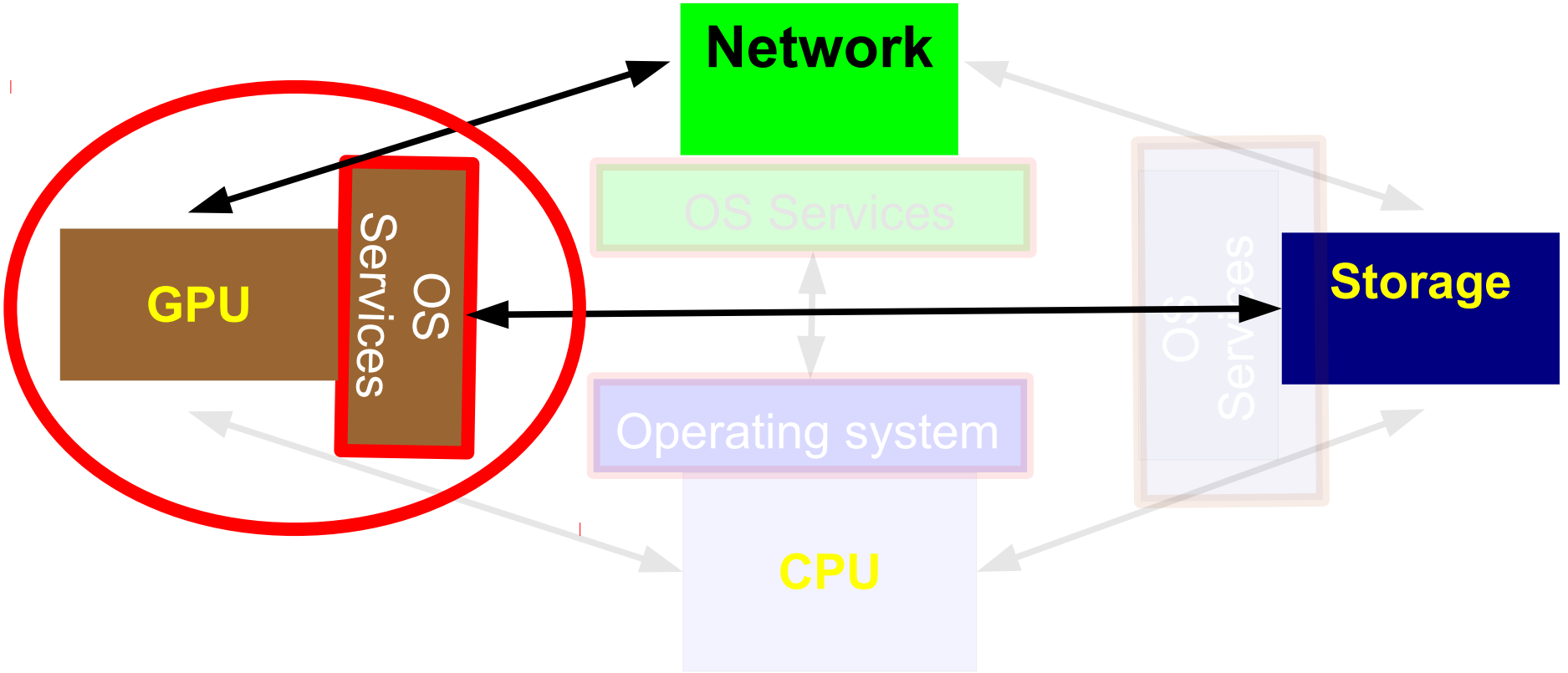
ASPLOS13,TOCS14,OSDI14,TOCS15,ISCA16,SYSTOR16,ROSS16,ATC17,HotOS17,ATC19,PACT19

Accelerator-centric: no CPU in data/control path

Accelerator-friendly: accelerator-aware host OS

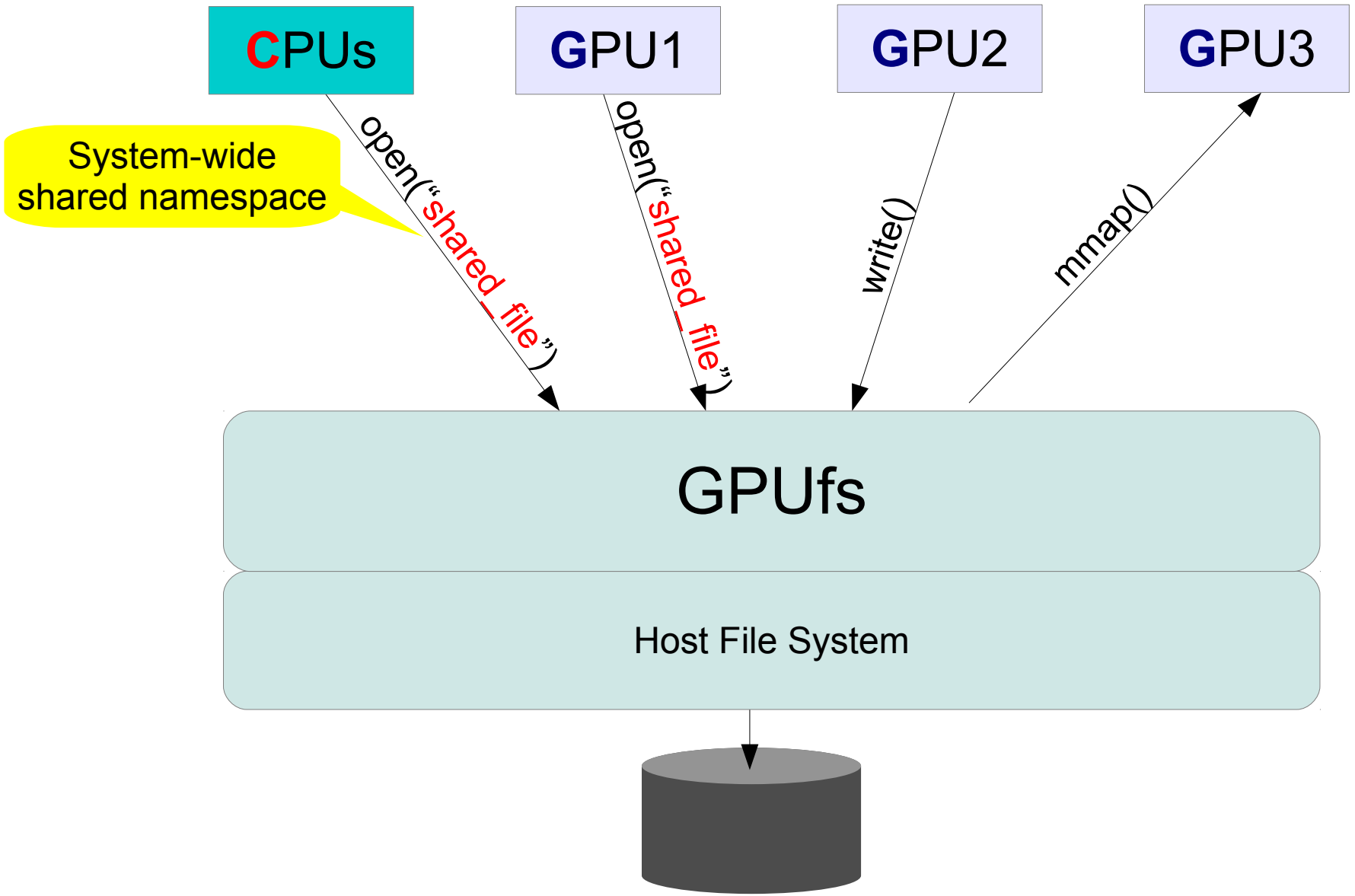
Data-centric: inline near-data processing

GPU I/O services



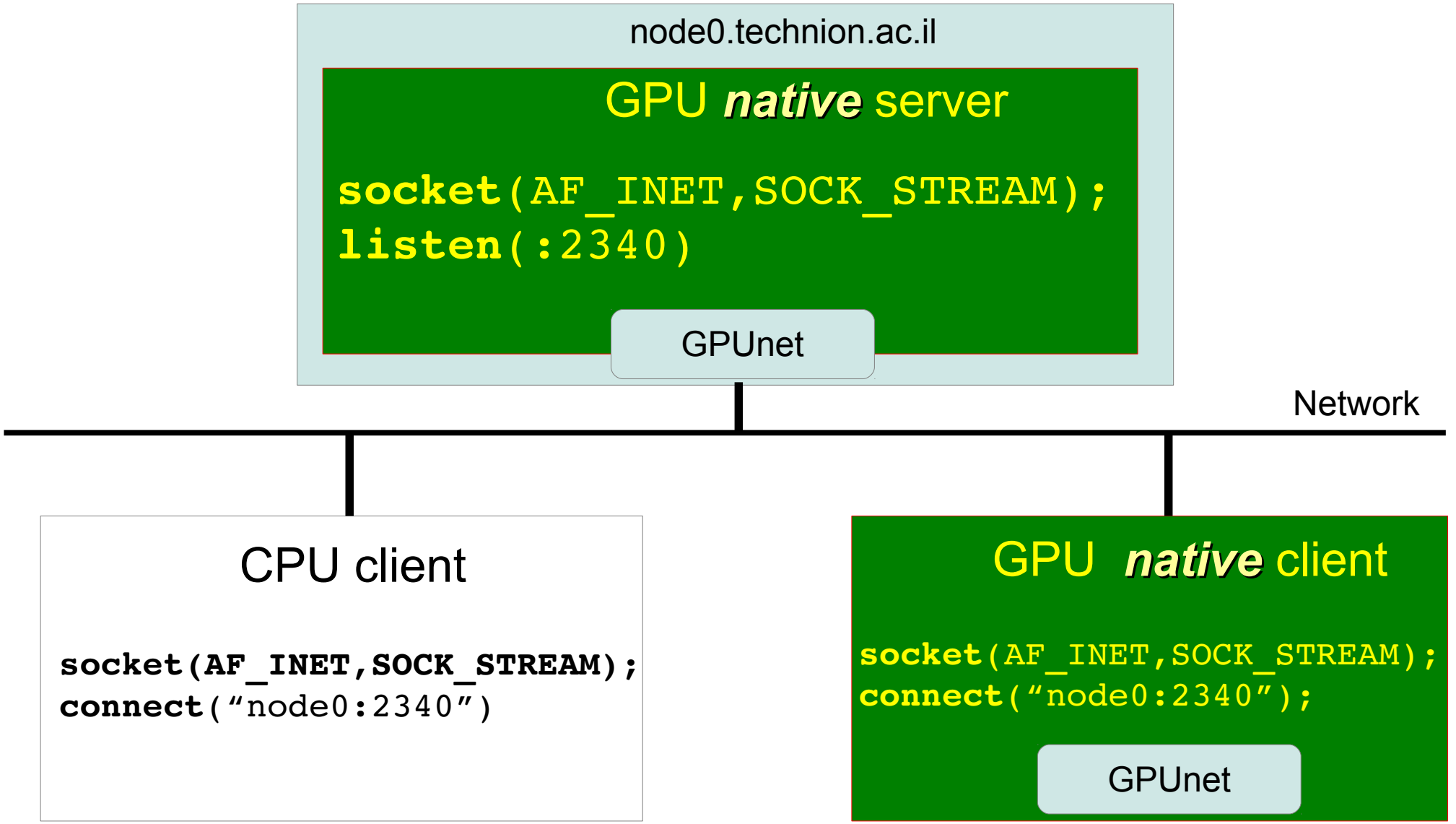
GPUfs: File system library for GPUs

ASPLOS13: S., Keidar, Ford, Witchel



GPUnet: Network library for GPUs

OSDI14



Accelerator in full control over its I/O

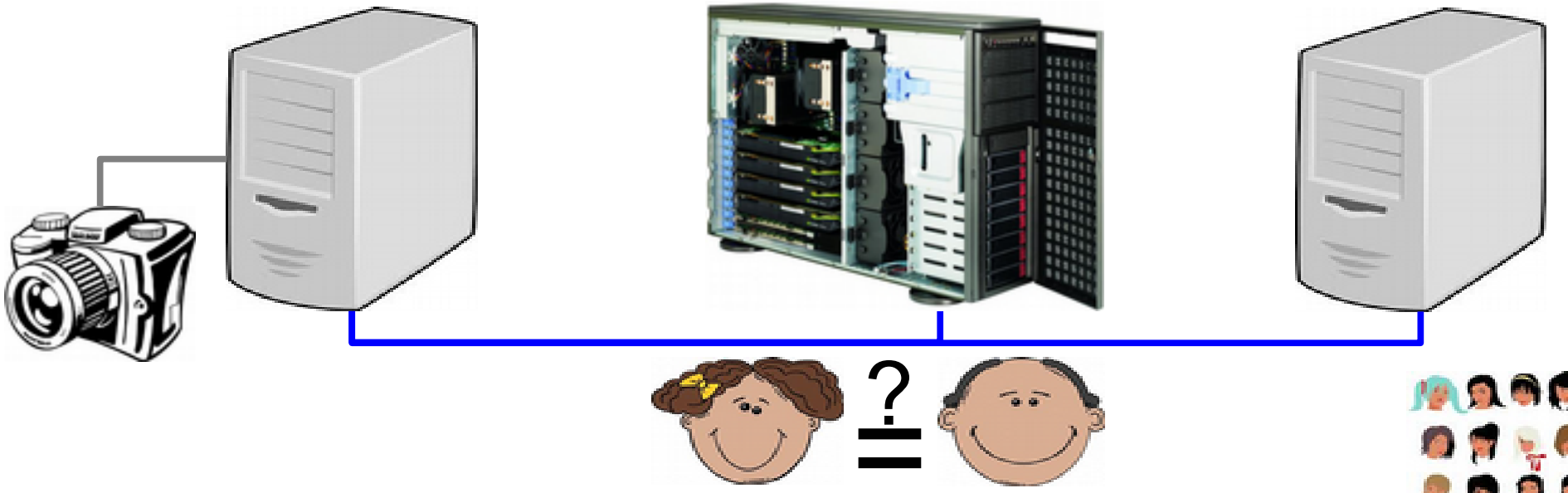
- I/O without «leaving» the GPU kernel
  - Data-driven access to huge DBs
  - Full-blown multi-tier GPU network servers
  - Multi-GPU Map/Reduce (no user CPU code)
- POSIX-like APIs with slightly modified semantics
- Transparency for the rest of the system
- Reduced code complexity
- Unleashed GPU performance potential

Example: face verification server

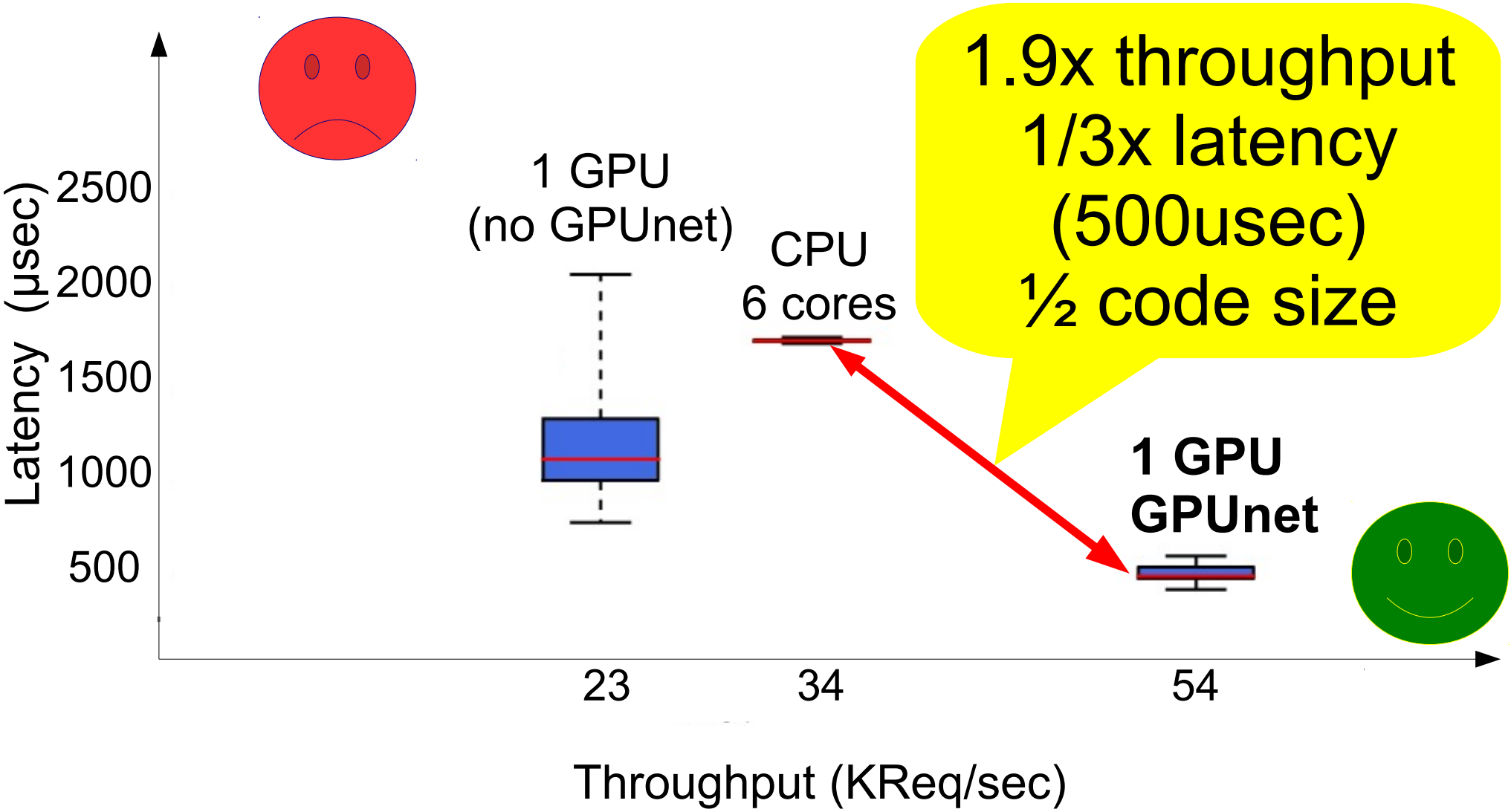
CPU client (unmodified)

GPU server (GPUnet)

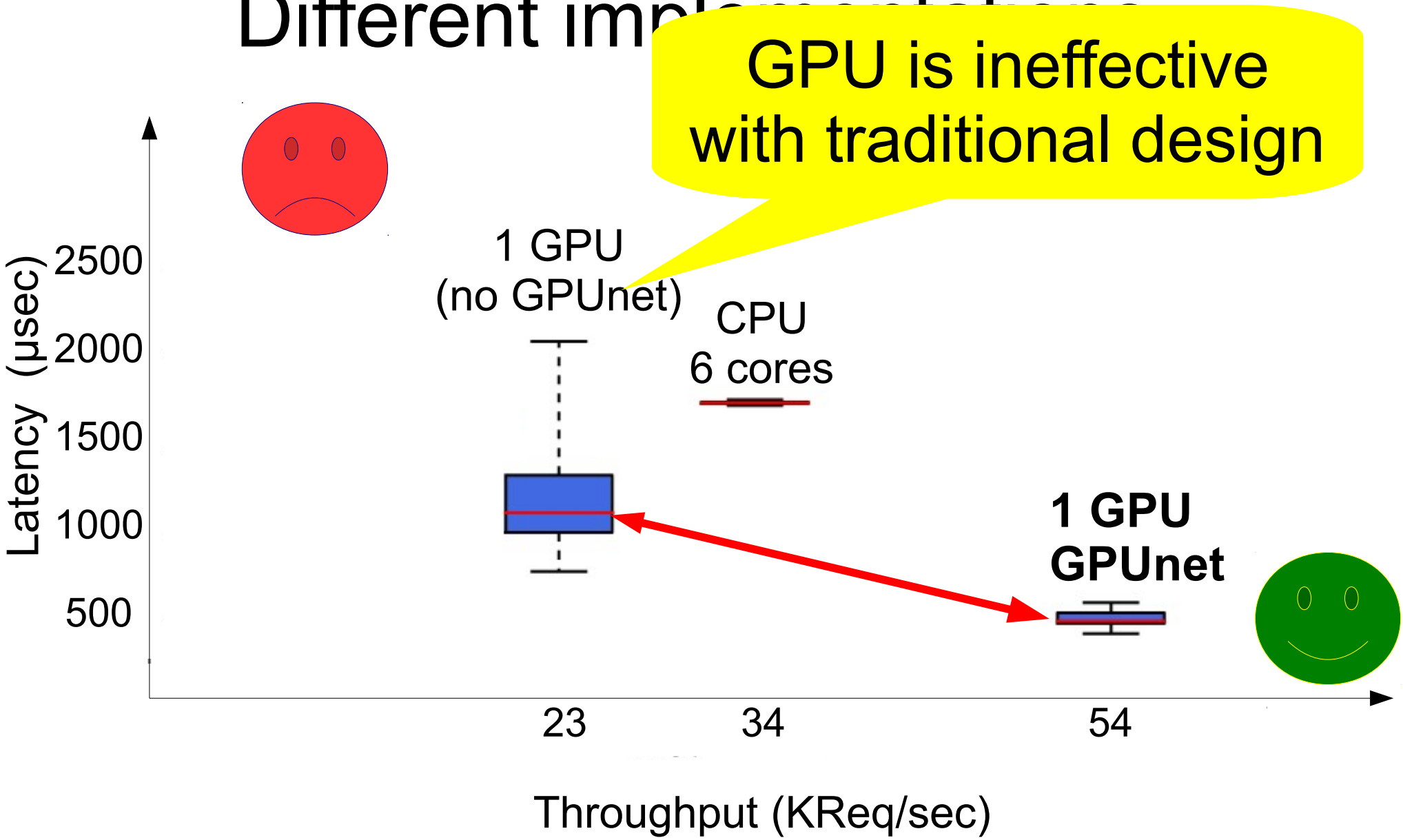
memcached (unmodified)



Face verification: Different implementations



Face verification: Different imp



Main design principles

- Micro-kernel design
  - RPC to File/Network services on the ***CPU***
  - User-land abstraction implementation (libOSes)

Main design principles

- Micro-kernel design
  - RPC to File/Network services on the CPU
  - User-land abstraction implementation (libOSes)
- Single name space with the CPU OS
  - Same socket space, same file name space

Main design principles

- Micro-kernel design
  - RPC to File/Network services on the CPU
  - User-land abstraction implementation (libOSes)
- Single name space with the CPU OS
  - Same socket space, same file name space
- Extensive SW layer on the GPU
  - Handles massive API parallelism
  - Implements consistency model (FS)
  - Implements flow control (sockets)

Main design principles

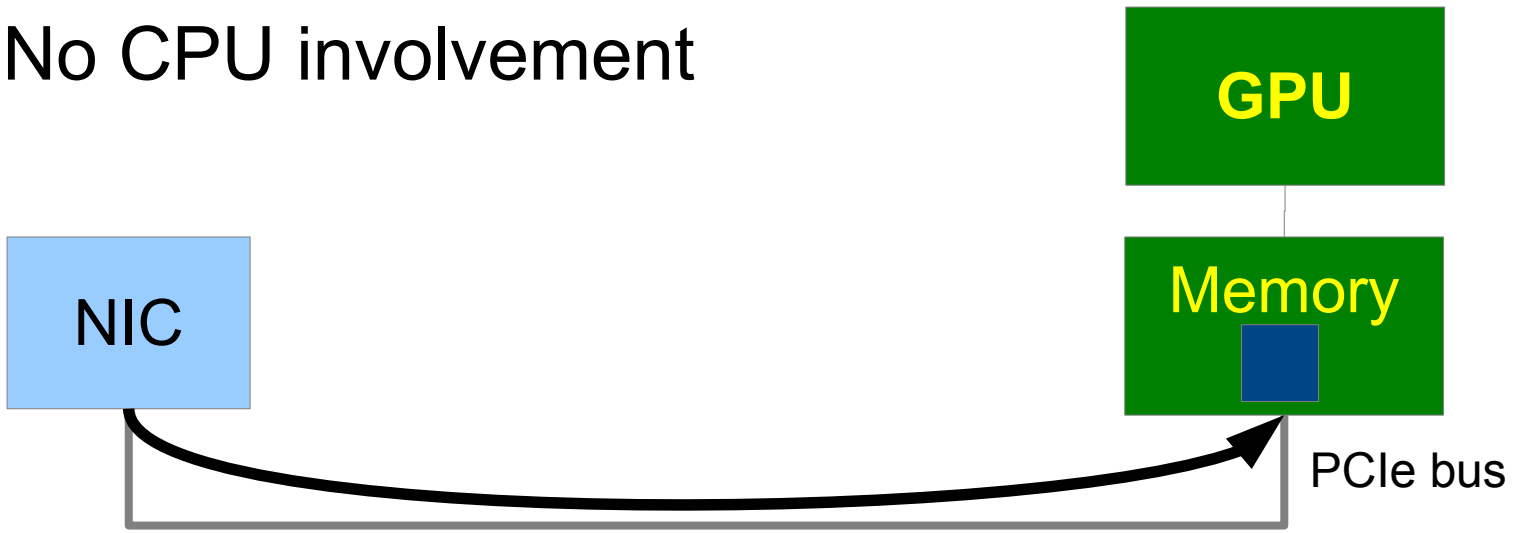
- Micro-kernel design
  - RPC to File/Network services on the CPU
  - User-land abstraction implementation (libOSes)
- Single name space with the CPU OS
  - Same socket space, same file name space
- Extensive SW layer on the GPU
  - Handles massive API parallelism
  - Implements consistency model (FS)
  - Implements flow control (sockets)
- Seamless data path optimization
  - Eliminates CPU from data path
  - Exploits data locality

Main design principles

- Micro-kernel design
  - RPC to File/Network services on the CPU
  - User-land abstraction implementation (libOSes)
- Single name space with the CPU OS
  - Same socket space, same file name space
- Extensive SW layer on the GPU
  - Handles massive API parallelism
  - Implements consistency model (FS)
  - Implements flow control (sockets)
- Seamless data path optimization
  - Eliminates CPU from data path
  - Explits data locality

Optimized I/O: no CPU in data path

- SSD/NIC may perform DMA directly into/from GPU memory without the CPU (P2P DMA)
- Why?
  - Lower latency
  - Less buffering/complexity for thpt
  - No CPU involvement



Optimized I/O: no CPU in data path

- SSD/NIC may perform DMA directly into/from GPU memory without the CPU (P2P DMA)

Challenge: the OS is on the CPU!
I/O device sharing, multiplexing, translation, transport layer

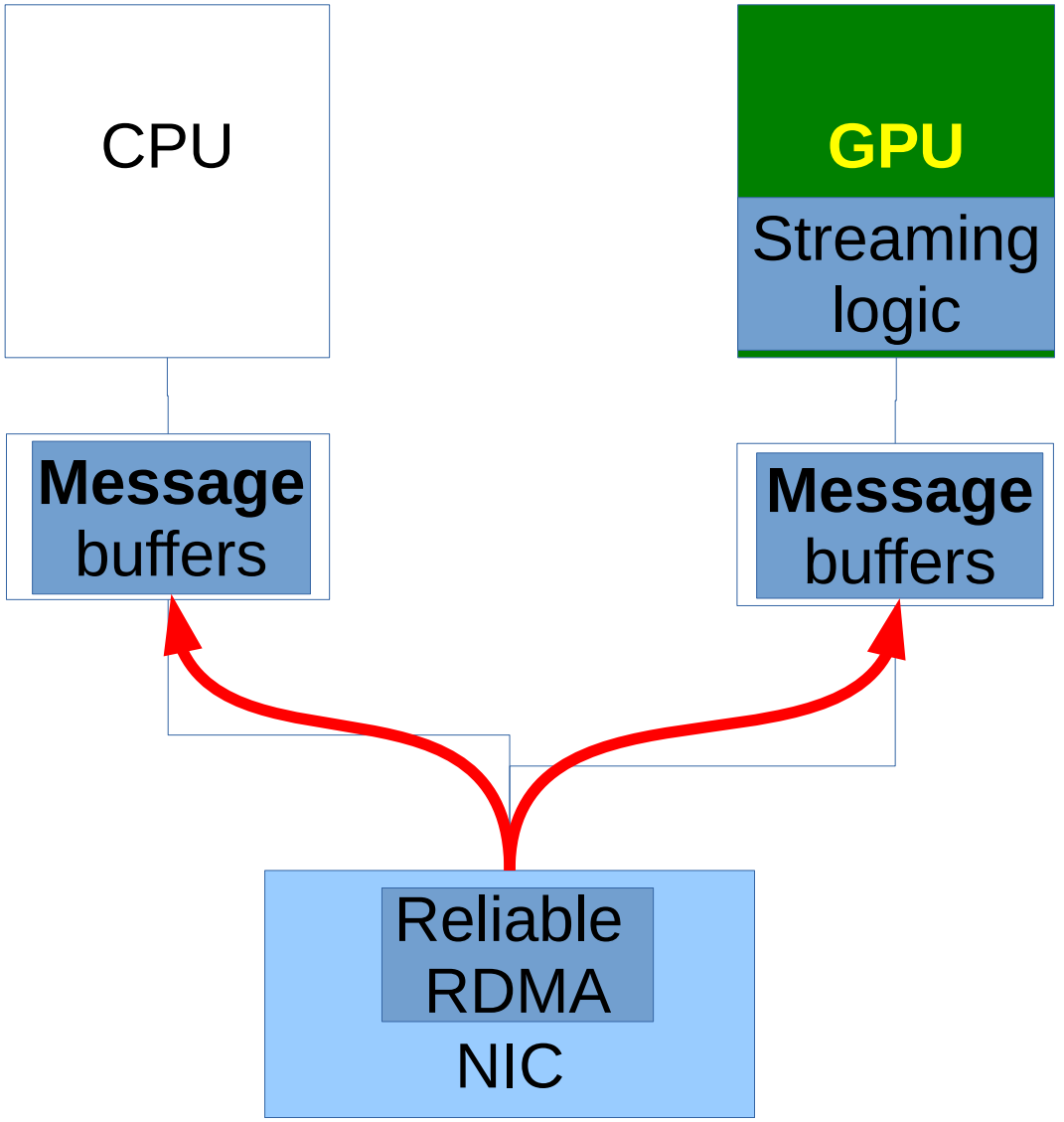
Examples:
- Data from FS could be in the CPU page cache
- GPU and CPU both need to access the network
- TCP on GPU?

Storage: OS integration of P2P between SSD and GPUs

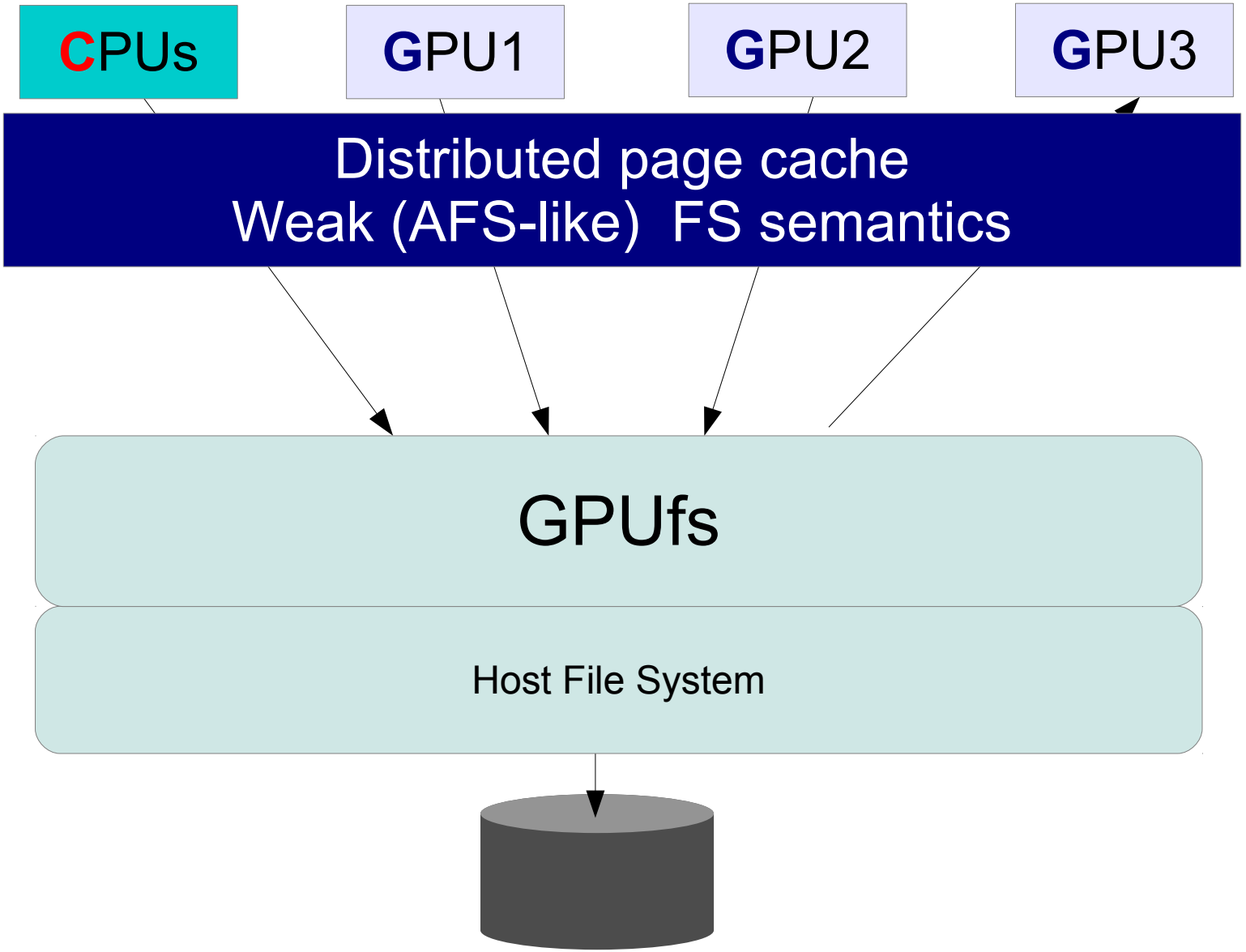
USENIX ATC17, partially adopted by NVIDIA

- Regular OS file APIs may use **GPU** memory buffers
  - mapping GPU memory into CPU address space
- Maintains POSIX FS consistency
- Transparently fetches the page cache or P2P DMA
- Integrates with OS prefetcher
- Compatible with OS block layer (i.e., software RAID)
- Results:
  - 5.2GB/s from SSDs to GPU
  - 2-3x in applications

Networking: offloading transport layer to the NIC (via RDMA)



Transparent locality optimization



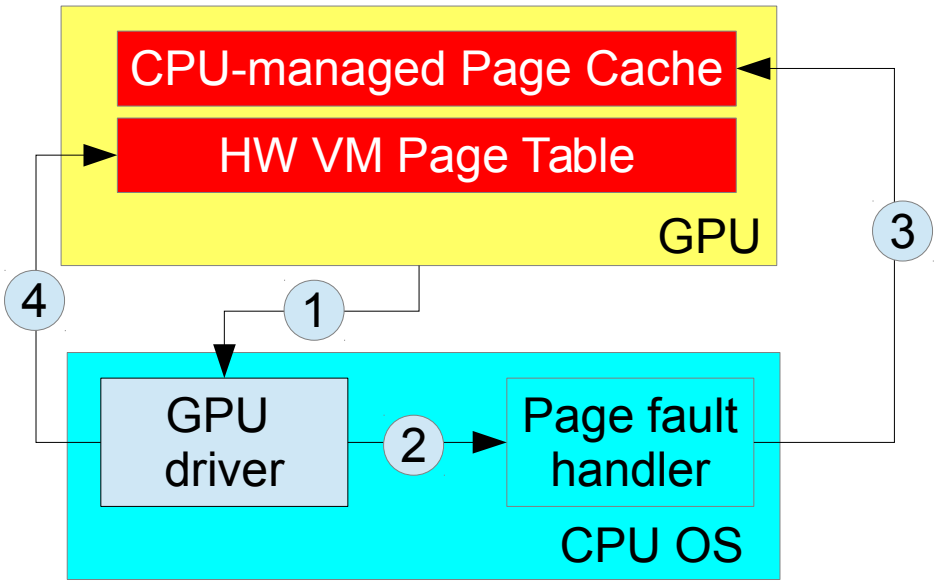
Summary so far...

- Basic GPU-centric I/O services
  - Simplify code development
  - Enable transparent performance optimization
- Can we implement more advanced OS services?

Hmm… what about memory mapped files?

- Useful abstraction
- We already have a page cache on the GPU
- We even have hardware page faults on the GPU, lets use them!

Background:
CPU handling of GPU page faults



GAIA: extending CPU page cache into GPU memory

[USENIX ATC19]

- Enables `mmap` for GPU
- May cache/prefetch file data in GPU memory
- Insights:
  - Slim GPU driver API for enabling host page cache integration
  - Page cache consistency model
  - OS page cache and Linux kernel modifications for consistency support

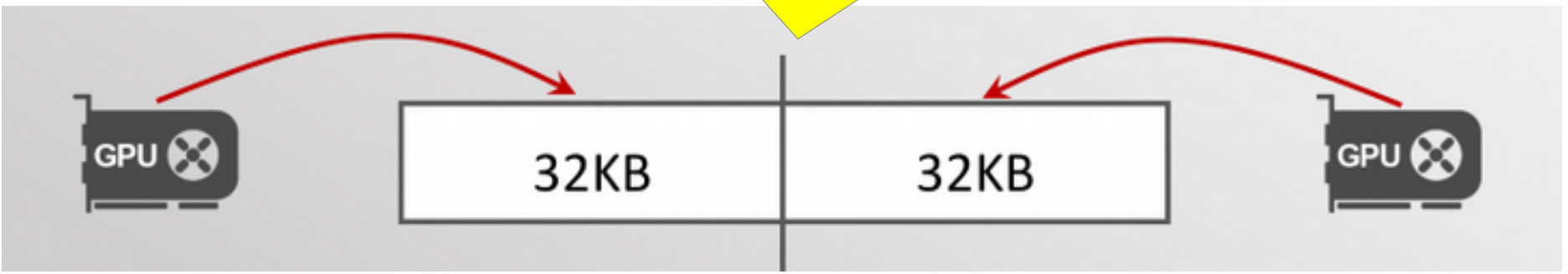
Question: can we use strong consistency in the page cache?

- Current practice in NVIDIA Unified Virtual Memory
- Single owner semantics: the page migrates to the requesting processor

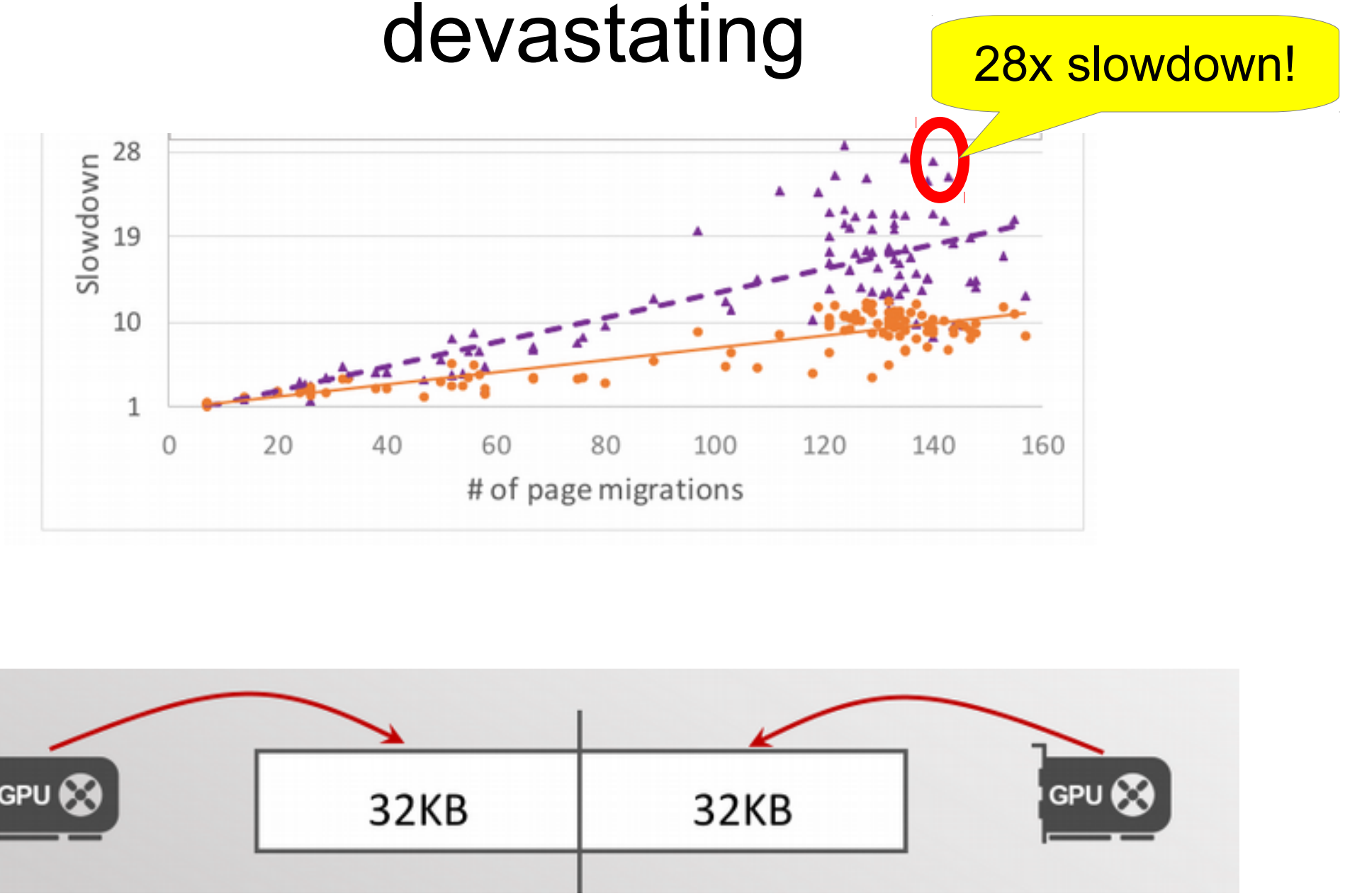
Question: can we use strong consistency in the page cache?

- Current practice in NVIDIA Unified Virtual Memory
- Single owner semantics: the page migrates to the requesting processor

But GPU page is 64KB!
False sharing inevitable
(also in real applications)

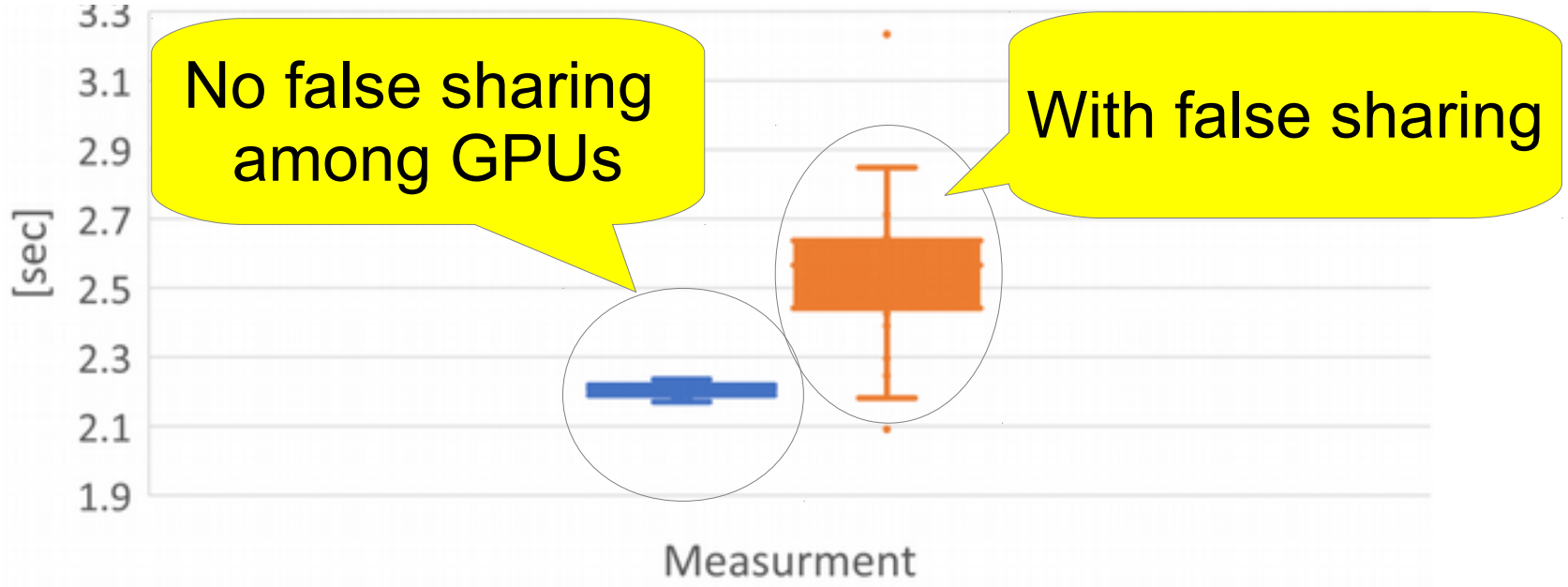


Extreme false sharing is devastating



It even affects CPU processes

CPU-only run of an HPC workload



Lazy Release Consistency to rescue

- Acquire-release to ensure update
- Version vectors for contention detection
- 3-way merge for conflict resolution
- *Transparent for legacy CPU processes*

GPU management code on the CPU

```
int fd=open(«shared_file»);
void* ptr=mmap(…,ON_GPU,fd);
macquire(ptr);
gpu_kernel<<<>>>(ptr);
mrelease(ptr);
```

40% app improvement over strong consistency

GPU HW page faults are good, but…

- CPU is involved in every **GPU** page fault
- CPU is the bottleneck with many page faults
- Requires CPU-GPU coordination for page cache management

GPU HW page faults are good, but…

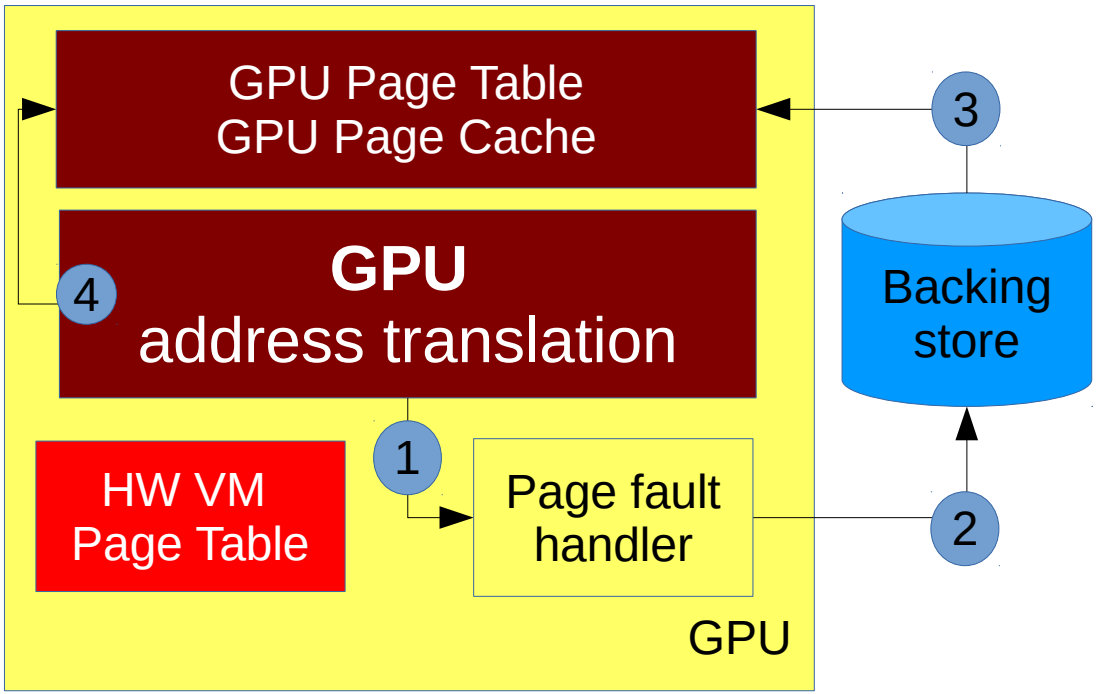
- CPU is involved in every **GPU** page fault
- CPU is the bottleneck with many page faults
- Requires CPU-GPU coordination for page cache management

Can we get rid of the CPU
in the page fault handling path?

GPU-centric Virtual Memory management

ISCA16, Operating Systems Review 2018

- GPU manages its own page tables



Peer-processor

Data and control path
and OS I/O abstractions
by the **GPU**

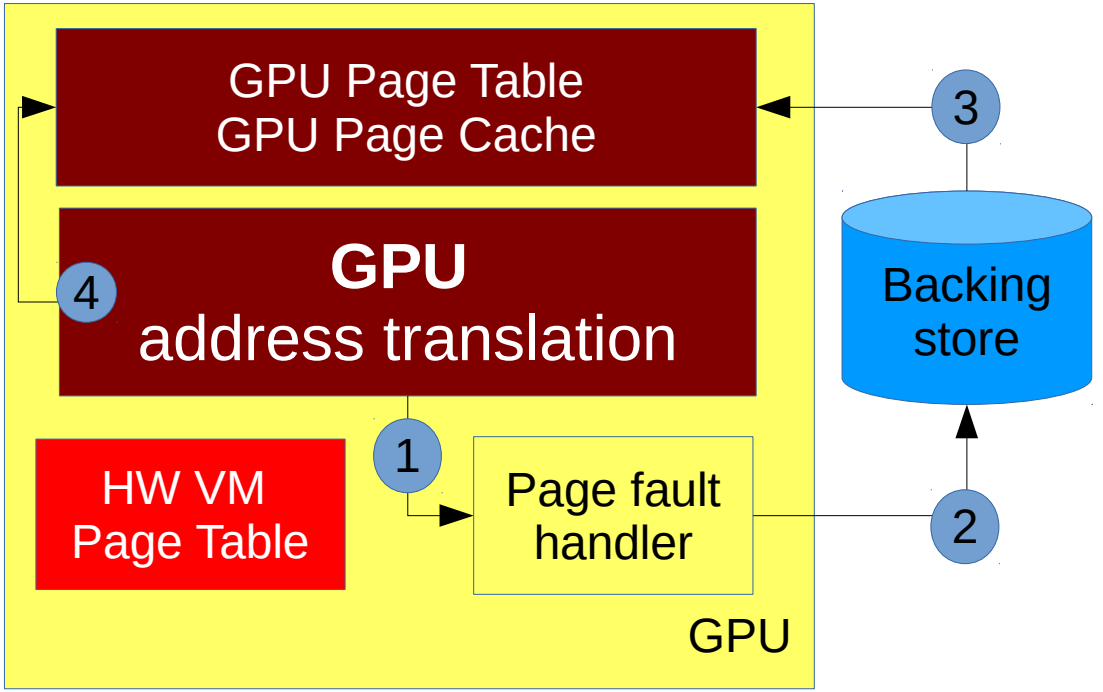
GPU-centric Virtual Memory management

ISCA16, Operating Systems Review 2018

- GPU manages its own page tables

Implementation:

- **Software address translation**
- Overheads hidden thanks to HW multithreading

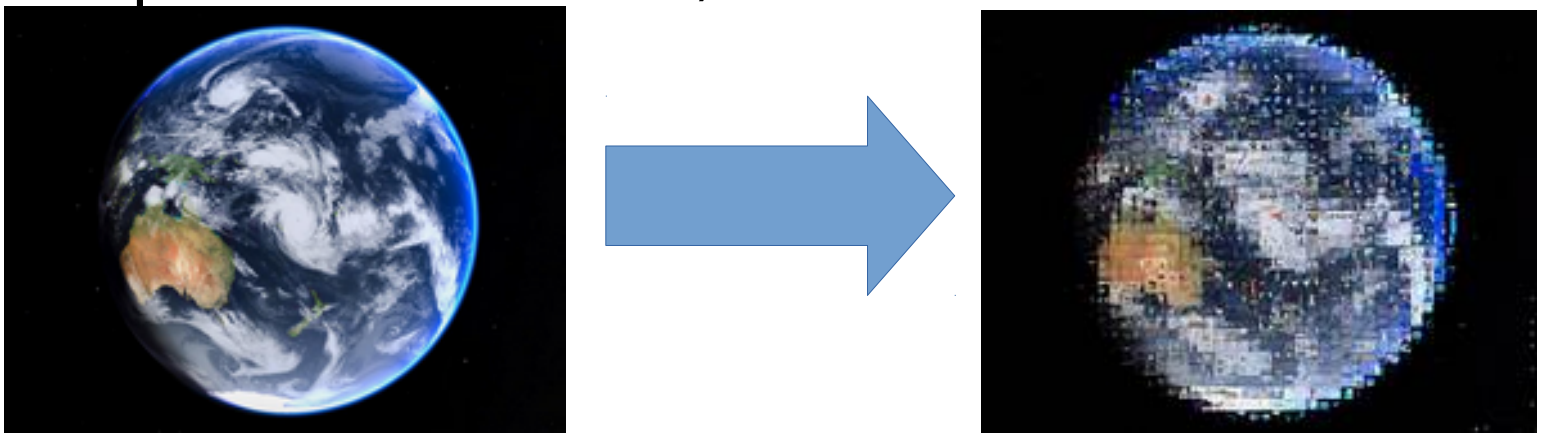


Peer-processor

Data and control path and OS I/O abstractions by the **GPU**

GPU-centric Virtual Memory management

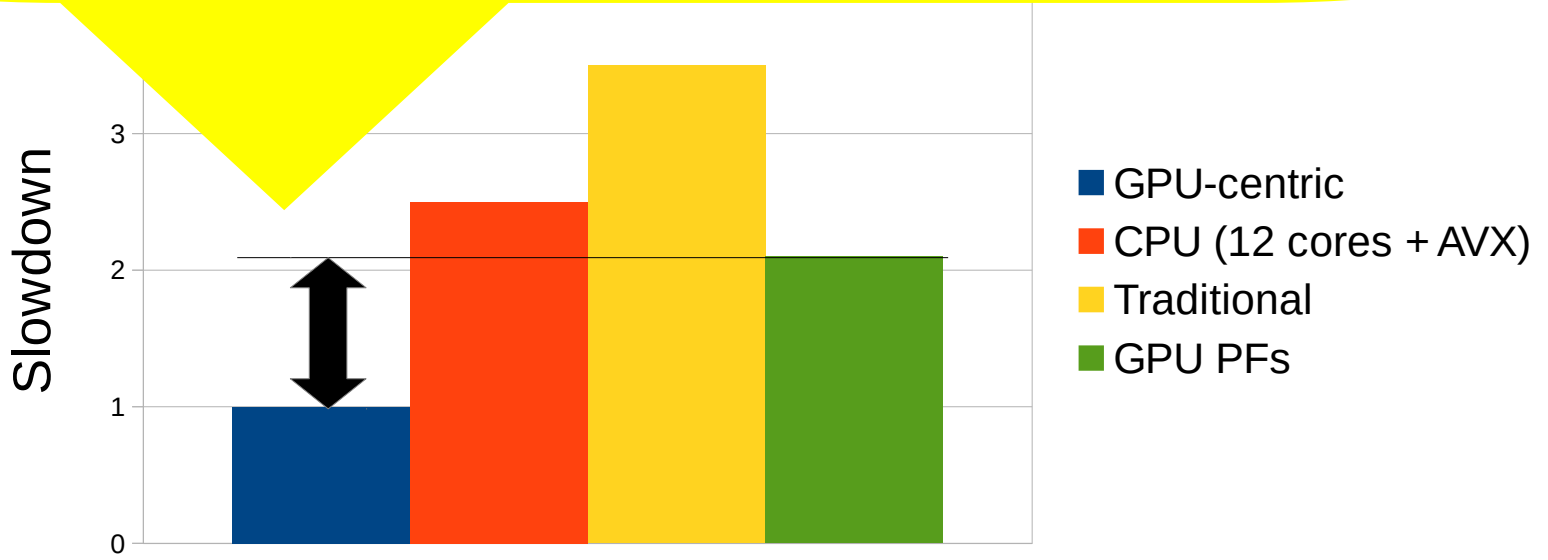
- Application: Image collage
- GPU mmaps **a 40GB DB** file, data-driven accesses



GPU-centric Virtual Memory management

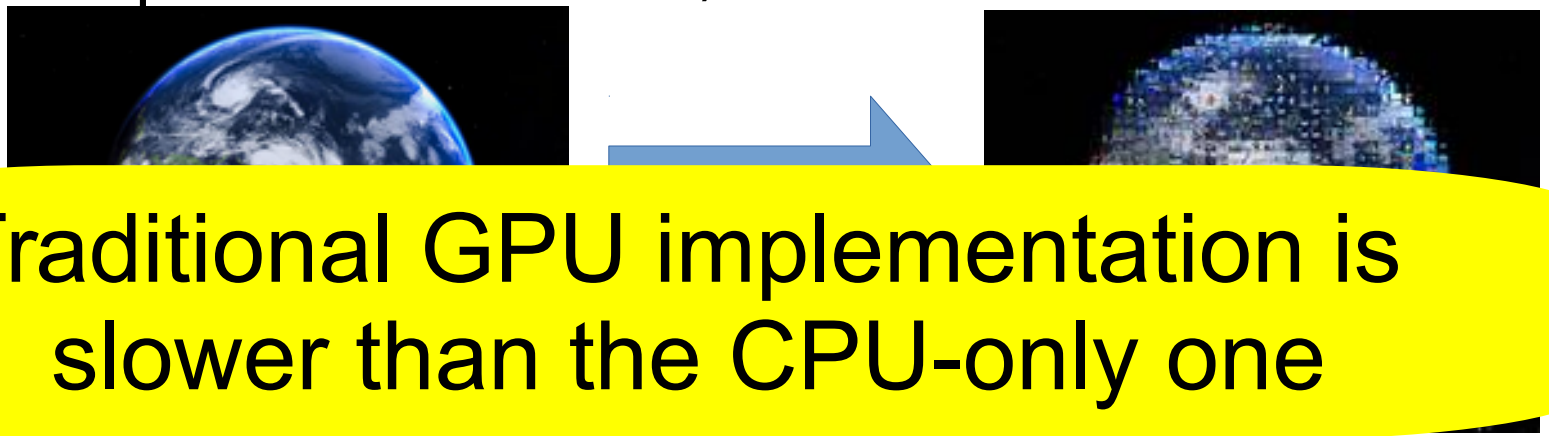
- Application: Image collage
- GPU mmaps **a 40GB DB** file, data-driven accesses

2x faster than GPU HW page faults

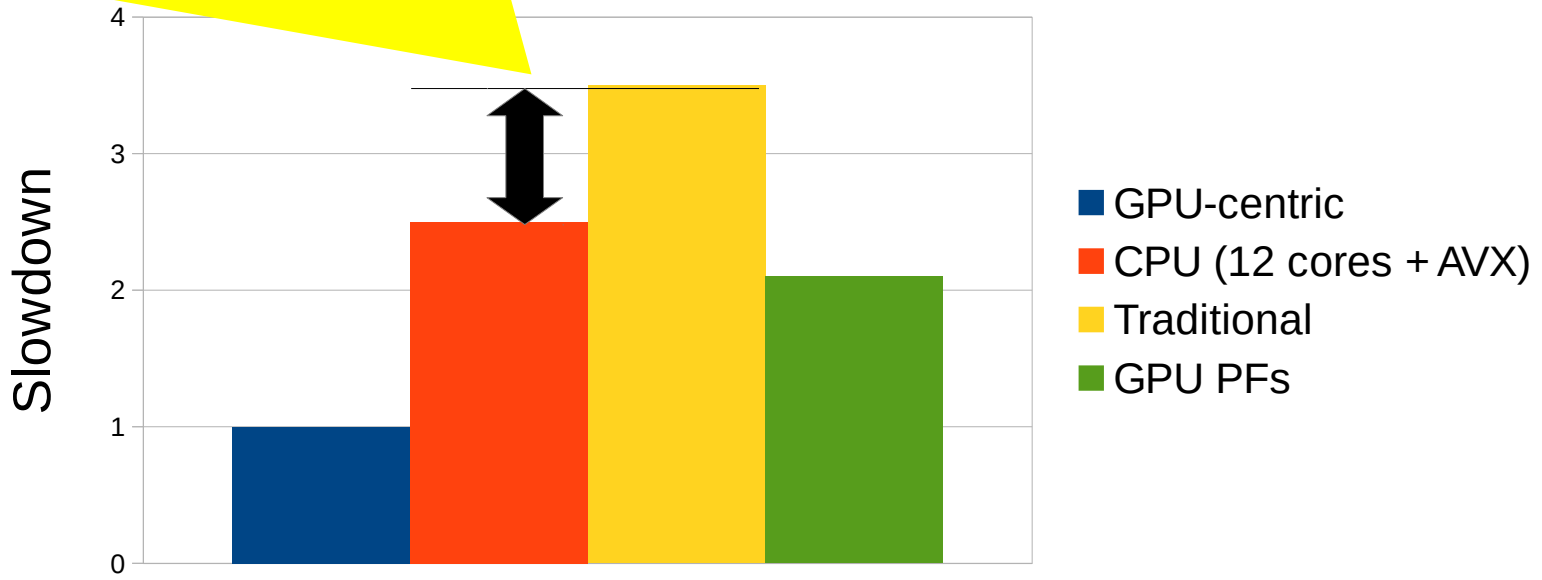


GPU-centric Virtual Memory management

- Application: Image collage
- GPU mmaps **a 40GB DB** file, data-driven accesses



Traditional GPU implementation is slower than the CPU-only one



Summary so far

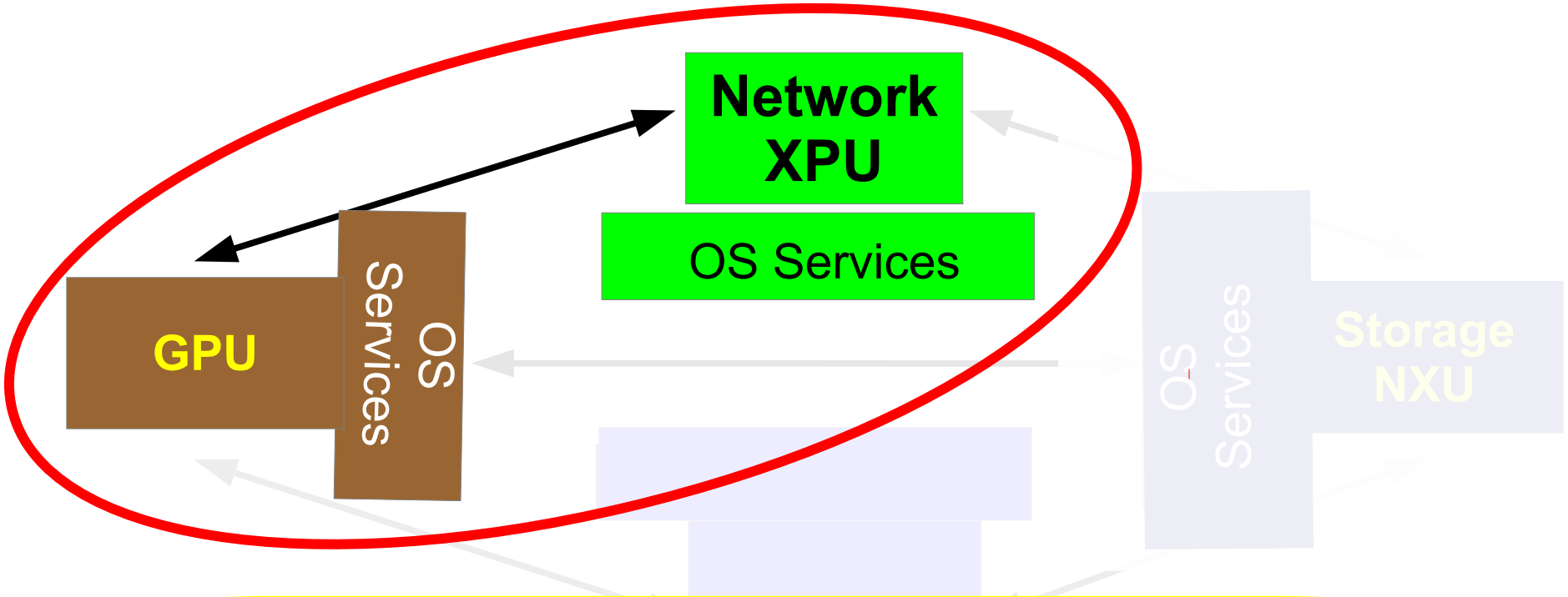
- GPU-centric I/O services
  - Simplify code development
  - Enable transparent performance optimization
  - Leverage new hardware features
- Eliminate CPU from data path

Summary so far

- GPU-centric I/O services
  - Simplify code development
  - Enable transparent performance optimization
  - Leverage new hardware features
- Eliminate CPU from data path

Can we get rid of the CPU in control path too?

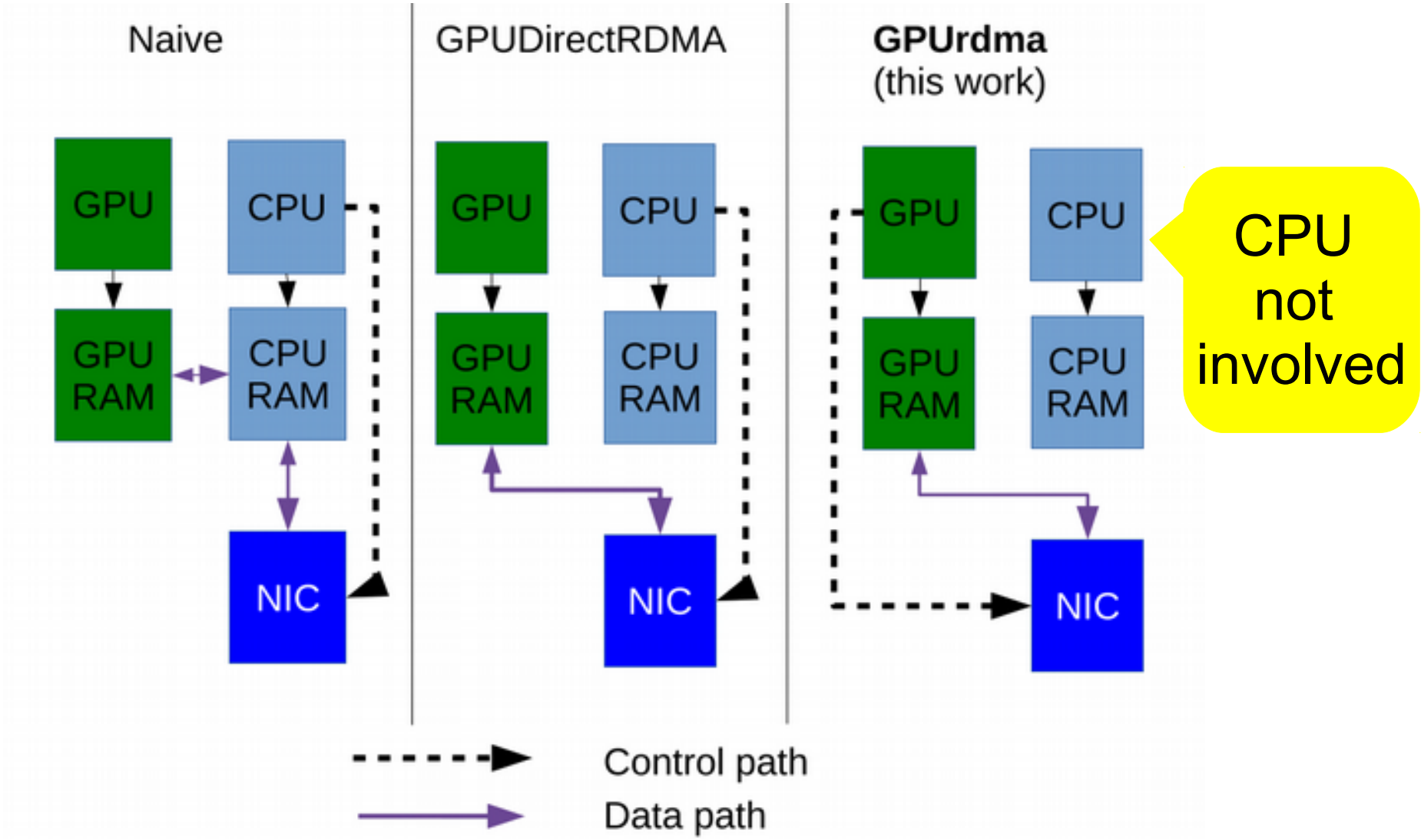
CPU-less design: no CPU in control and data path



Lower latency (no CPU roundtrip)
Better scalability (no CPU load)

GPUrdma: direct GPU access to RDMA

ROSS16, partially adopted by NVIDIA



GPUrdma: direct GPU access to RDMA

GPU-to-GPU roundtrip latency via Infiniband

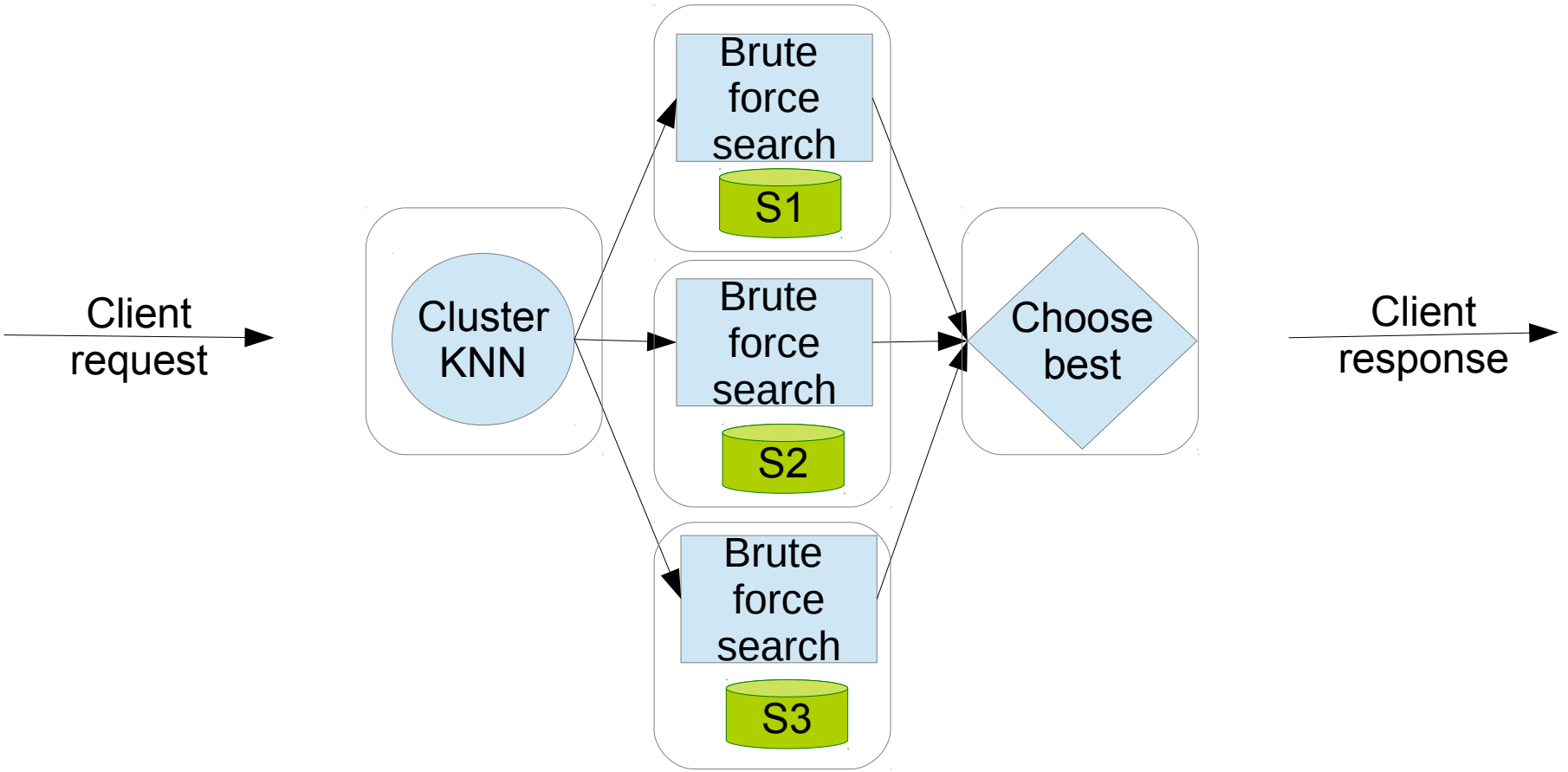
GPUnet (CPU-mediated RDMA): 50 usec

GPUrdma (NIC controlled by GPU): 5 usec

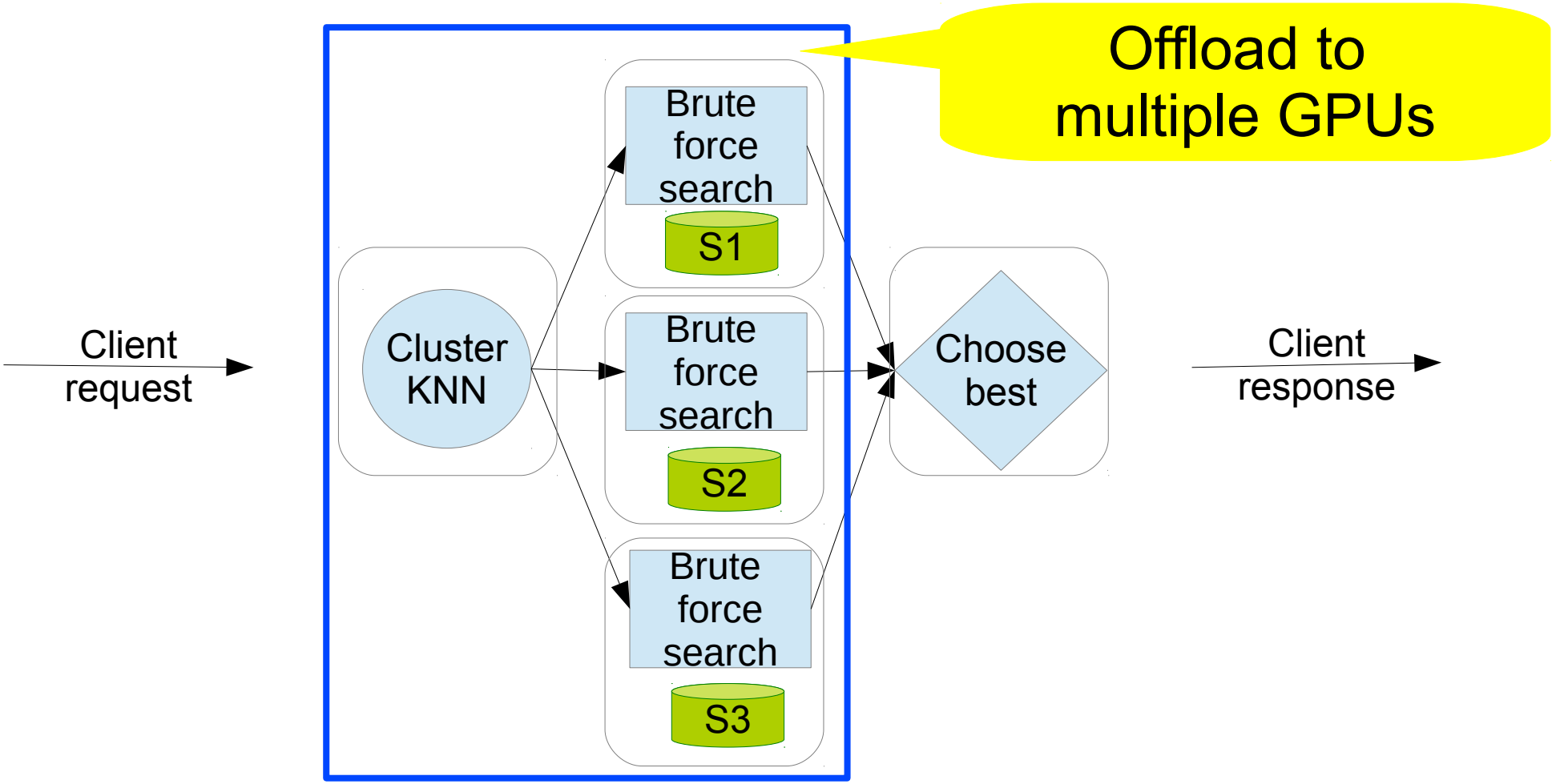
CPU-less design: lower latency

The case for CPU-less multi-GPU server design

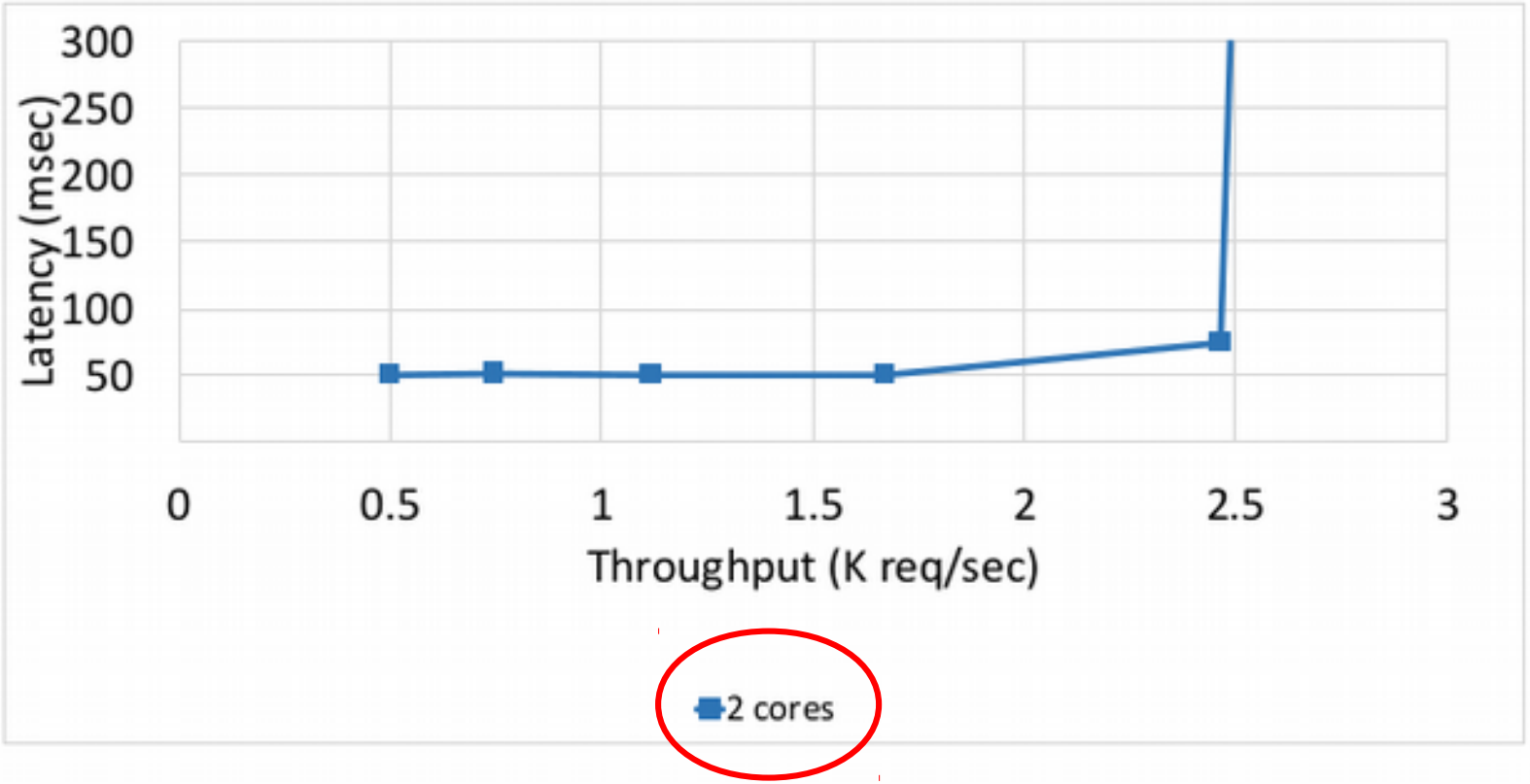
Image Similarity Search



Traditional design: CPU controls GPU invocation and data movements

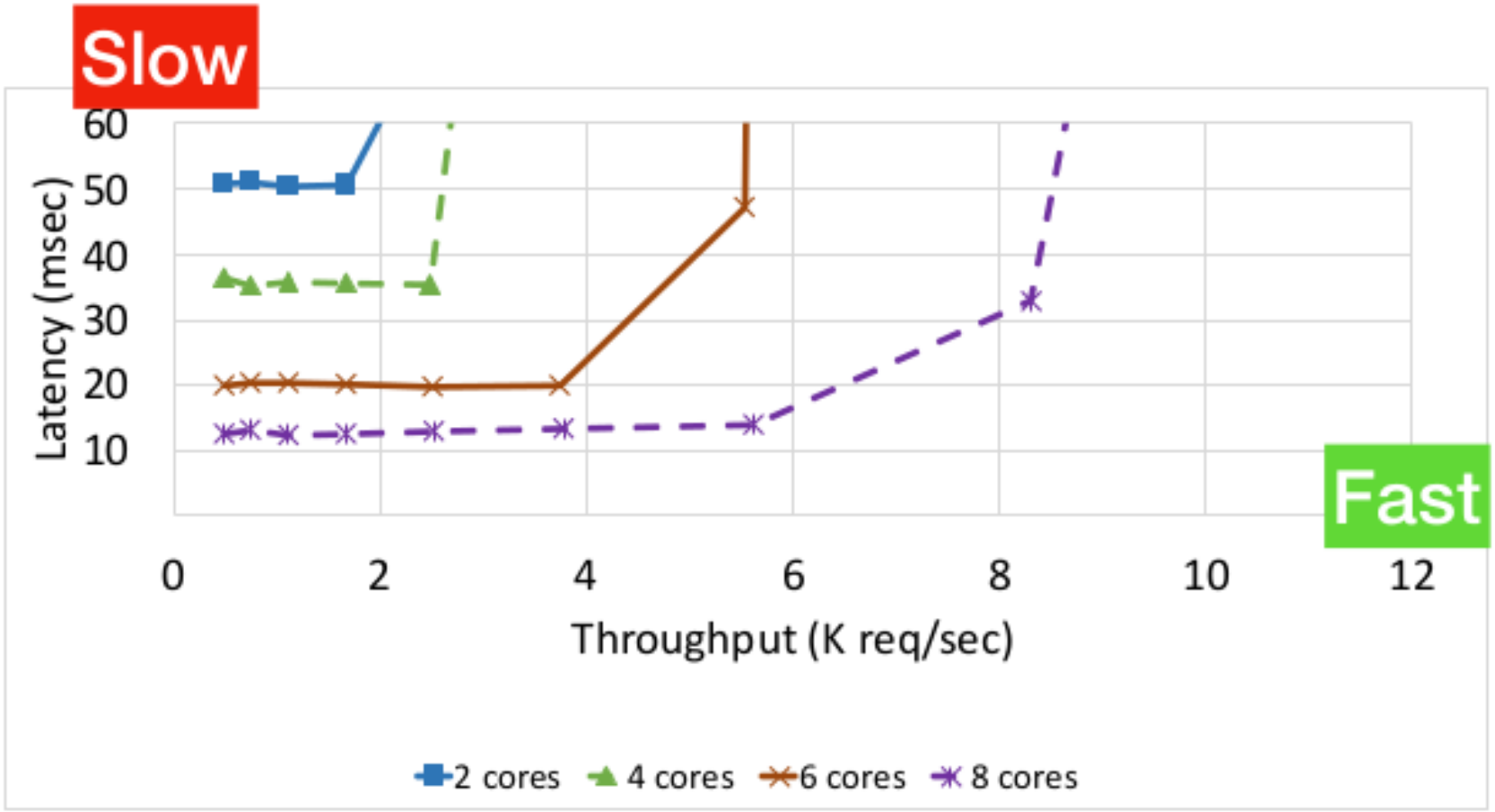


Traditional design: CPU controls GPU invocation and data movements

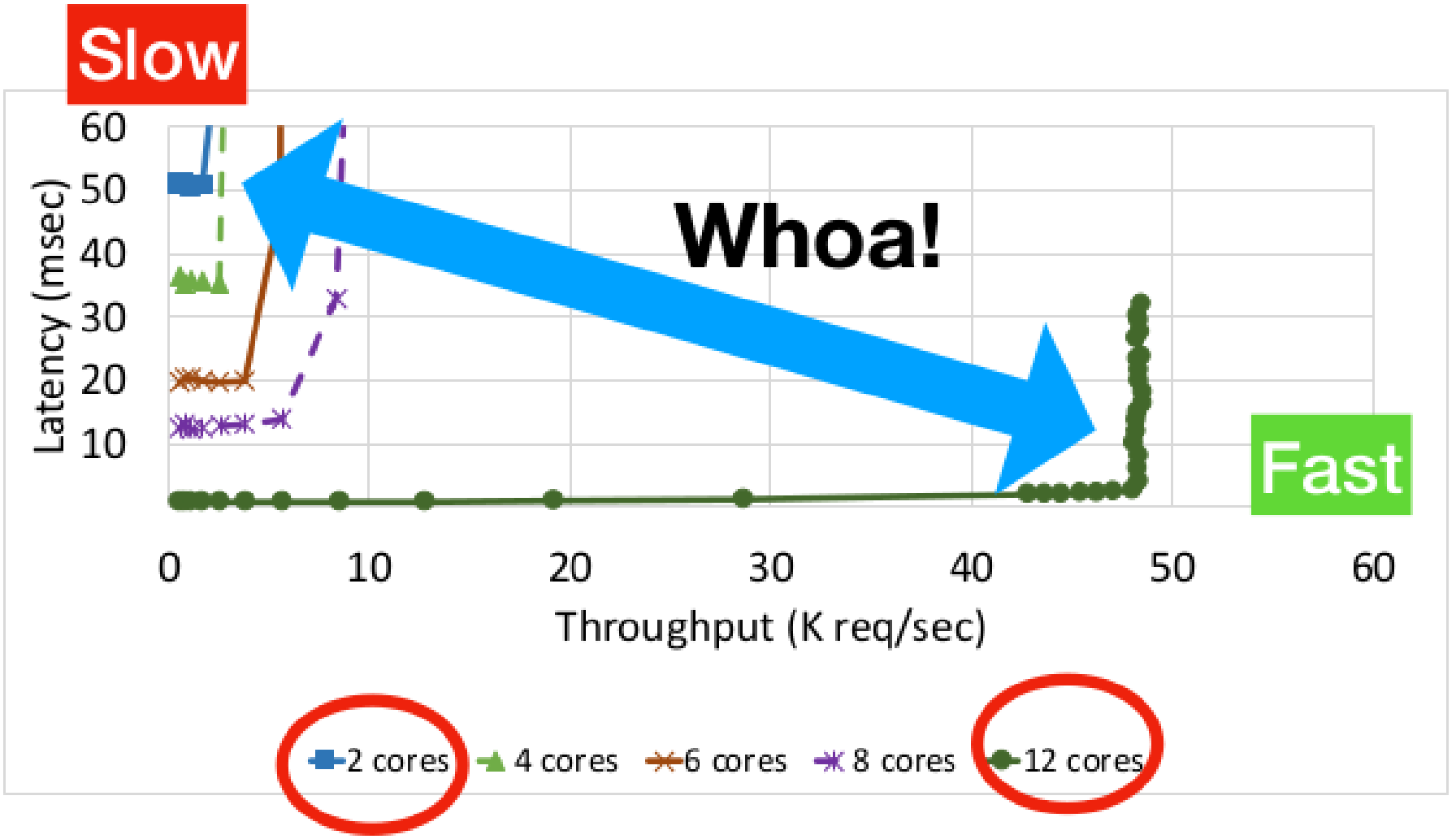


6 GPUs

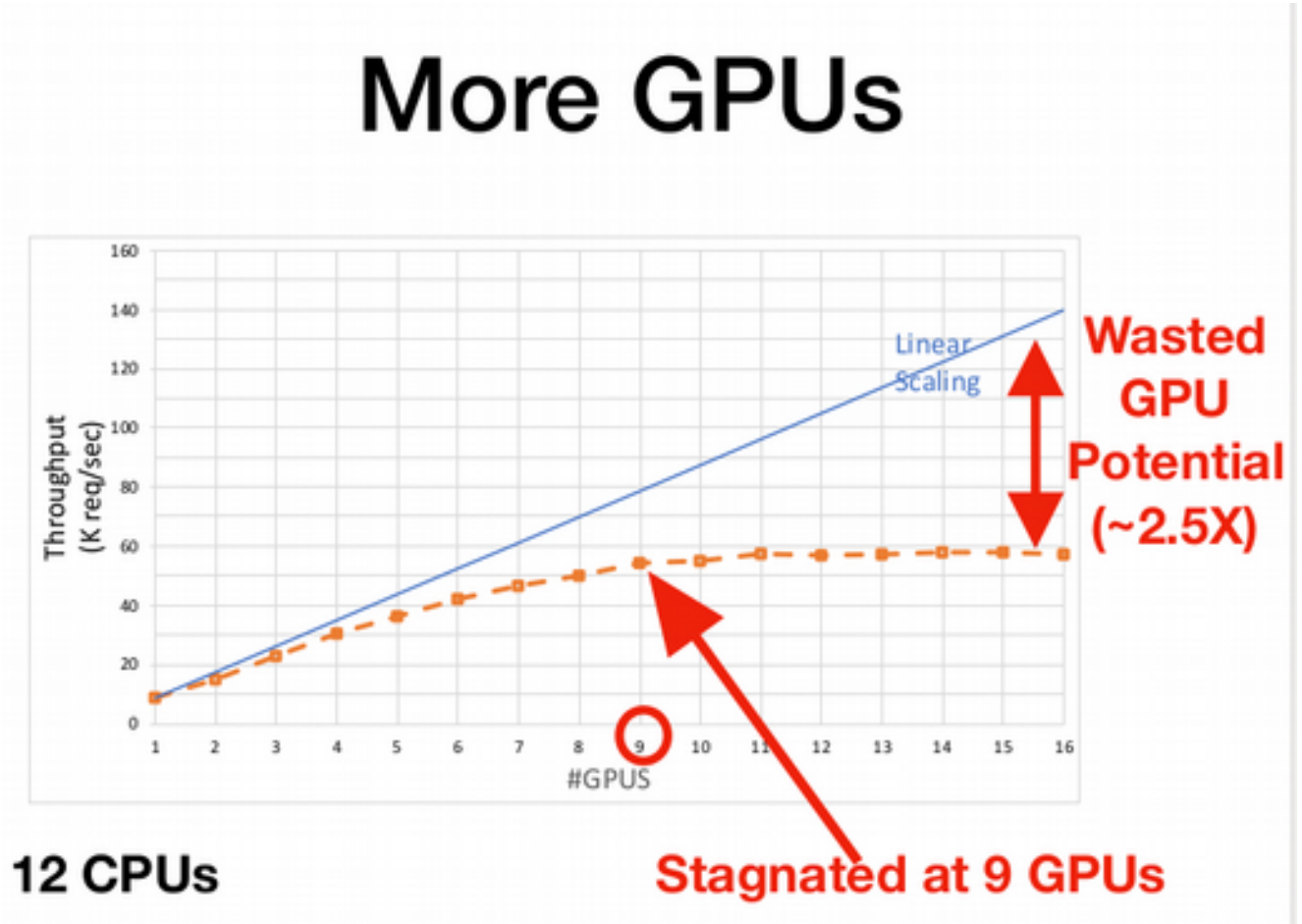
Lets add more CPU cores



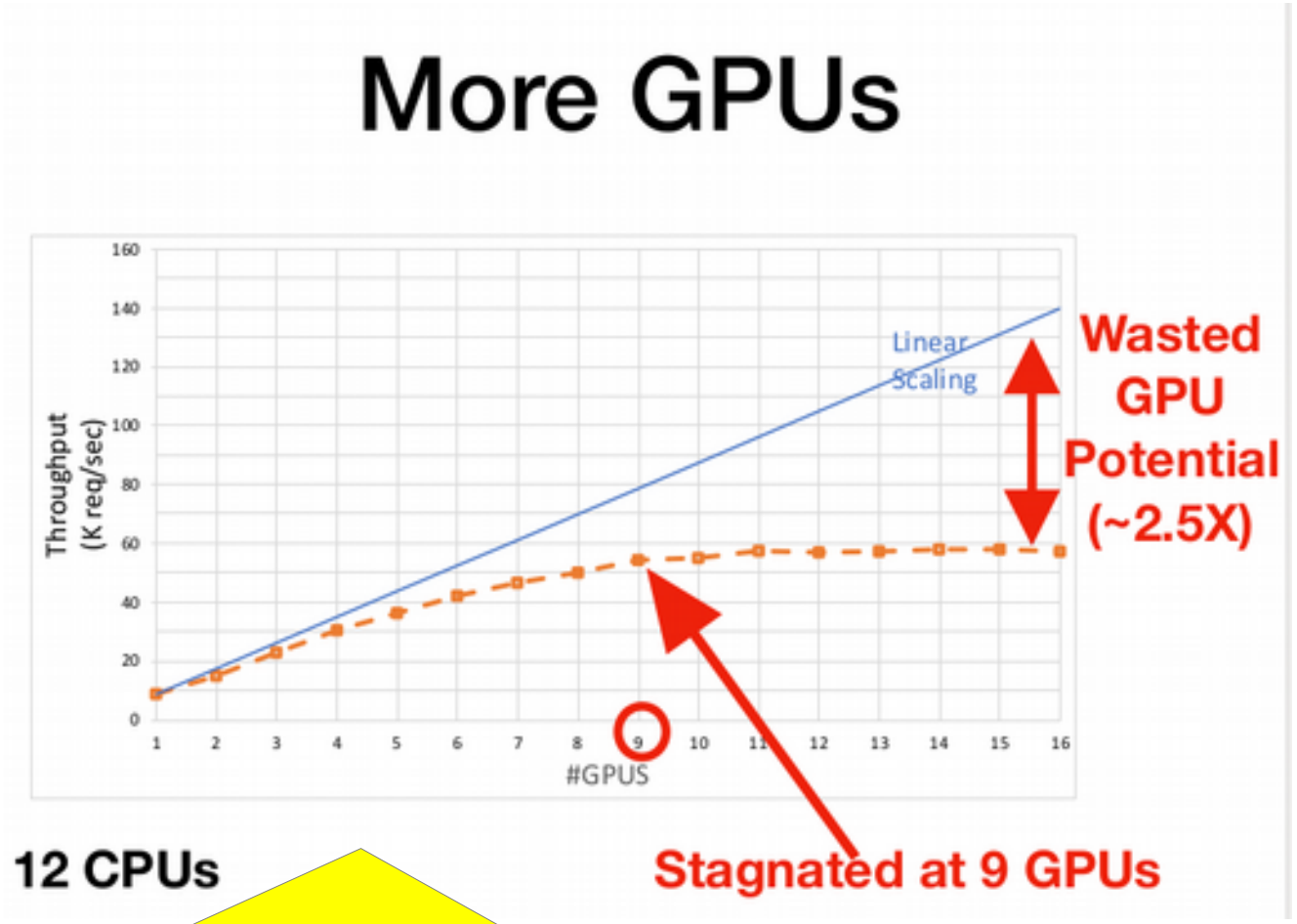
12 CPU cores needed!



12 CPUs are not enough to scale!



12 CPUs are not enough to scale!



The problem is inherent in the CPU-driven design [PACT19]

NVIDIA's DGX-2 GPU servers are CPU beasts!

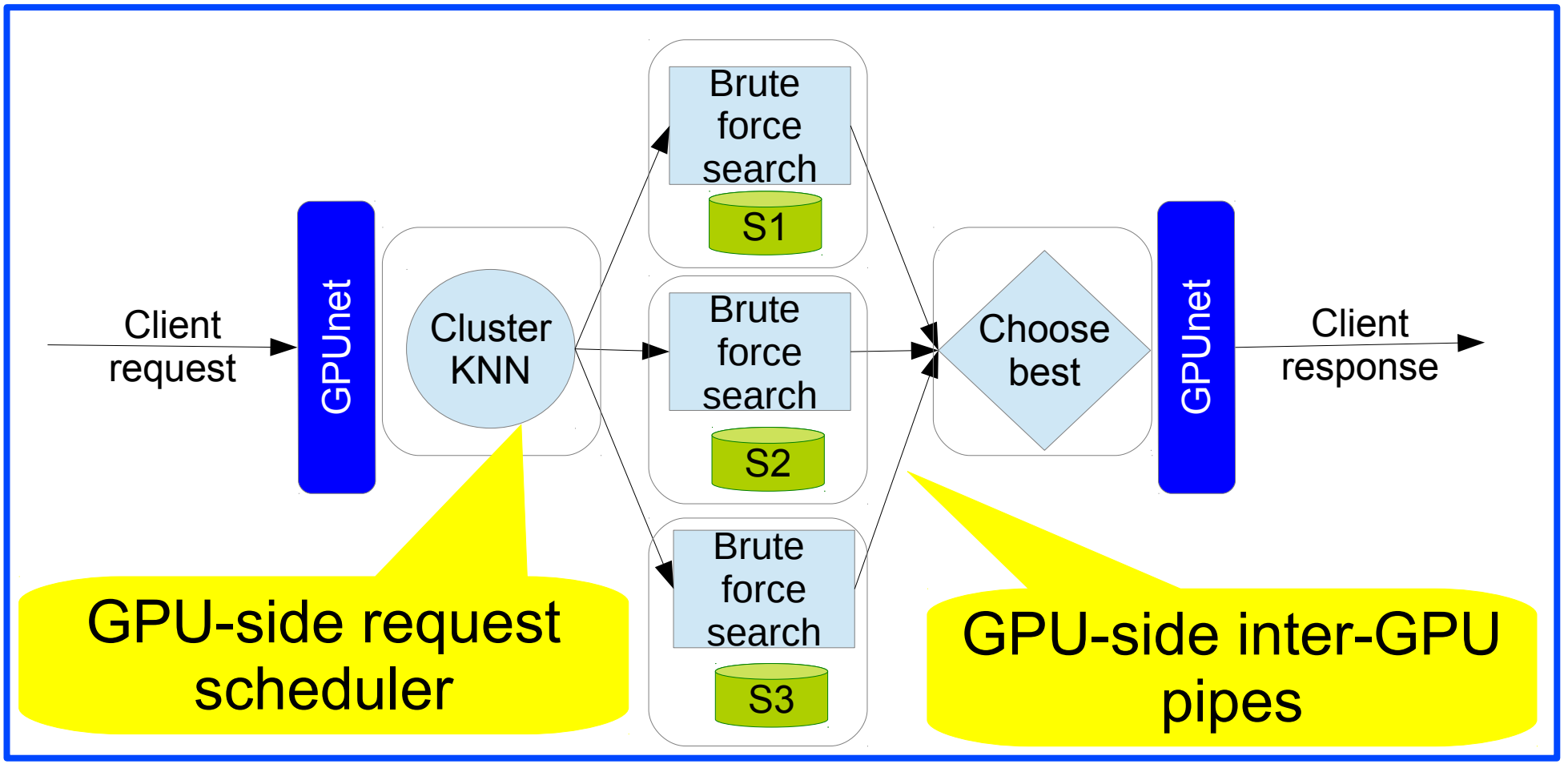
48 CPUs
vs.
16 GPUs
!!

SYSTEM SPECIFICATIONS

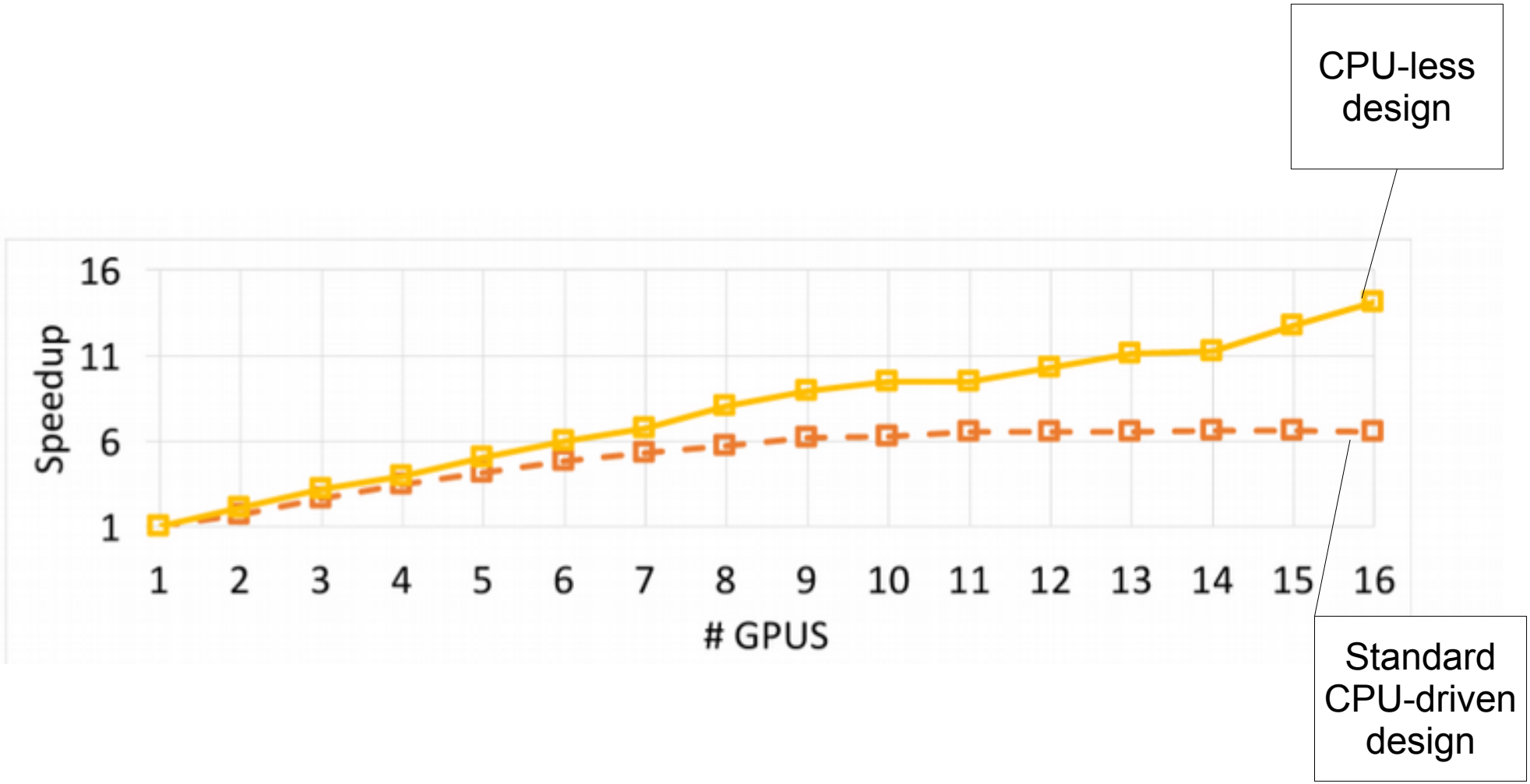
GPUs	**16X NVIDIA® Tesla® V100**
GPU Memory	**512GB total**
Performance	**2 petaFLOPS**
NVIDIA CUDA® Cores	**81920**
NVIDIA Tensor Cores	**10240**
NVSwitches	**12**
Maximum Power Usage	**10kW**
CPU	**Dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores**
System Memory	**1.5TB**

The case for **CPU-less** multi-GPU server design

PACT 19



CPU-less Multi-GPU network server



CPU-less design: better scaling

CPU's role

Do the setup
Then leave

Summary so far

- GPU-centric I/O services
  - Simplify code development
  - Enable transparent performance optimization
  - Can leverage new hardware features
- Eliminate CPU from data path
- Eliminate CPU from control path
  - it does not make sense in all cases though

But.. there are still issues

- GPUs are bad at system software
  - Function calls are extremely slow
  - Large code base causes slowdowns
  - No preemption, even not software-only
- GPU access to PCIe is ***slow***
  - even writes block multiple threads for a few usec

But.. there are still issues

- GPUs are bad at system software
  - Function calls are extremely slow
  - Large code base causes slowdowns
  - No preemption, even not software-only
- GPU access to PCIe is ***slow***
  - even writes block multiple threads for a few usec

We have a very cool solution, but.. [anonymized]

Types of OS abstractions for accelerators

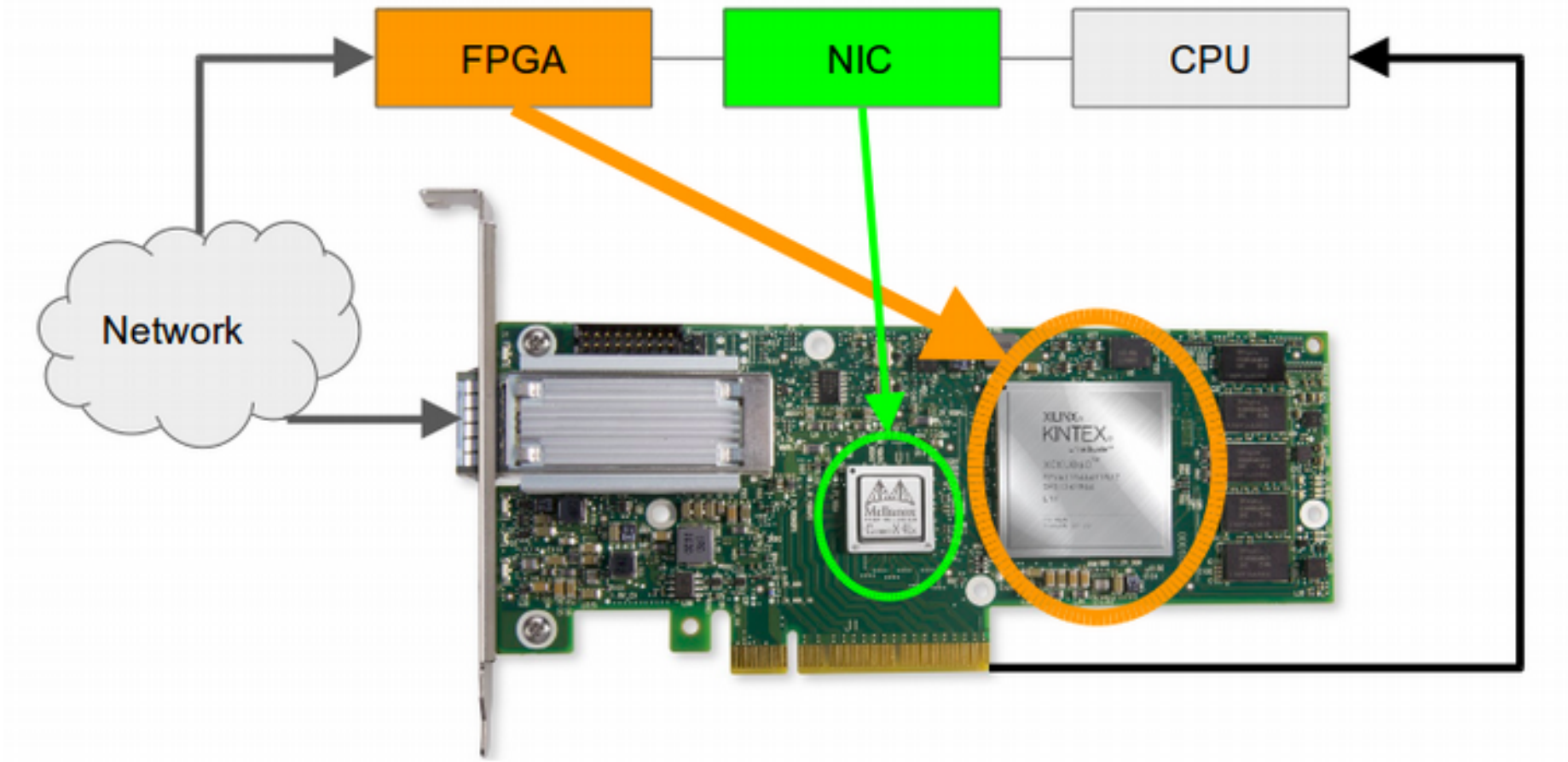
~~Accelerator-centric: no CPU in data/control path~~

~~Accelerator-friendly: accelerator-aware host OS~~

SFMA17, SFMA18, FCCM19, USENIX ATC19

Data-centric: inline near-data processing

Smart Network Adapters



Inline processing at wire speed!

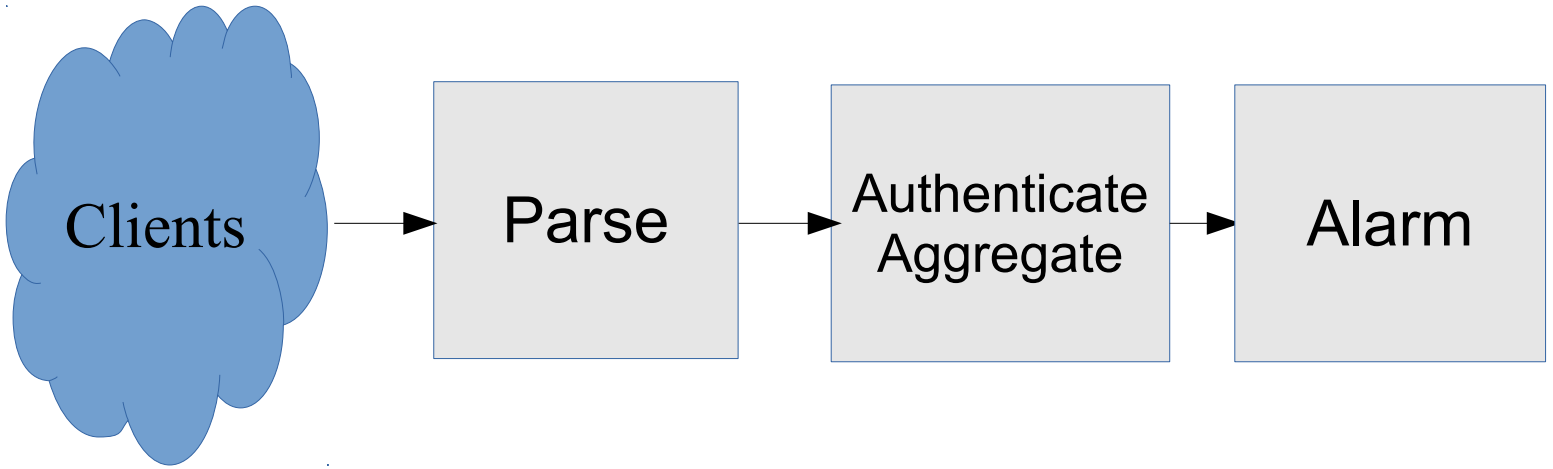
Main applications today

- Network infrastructure offloads
  - virtual networking, encryption, compression, SDN, NFV
- Stand-alone application accelerators
  - DNNWeaver, Bing, BrainWave

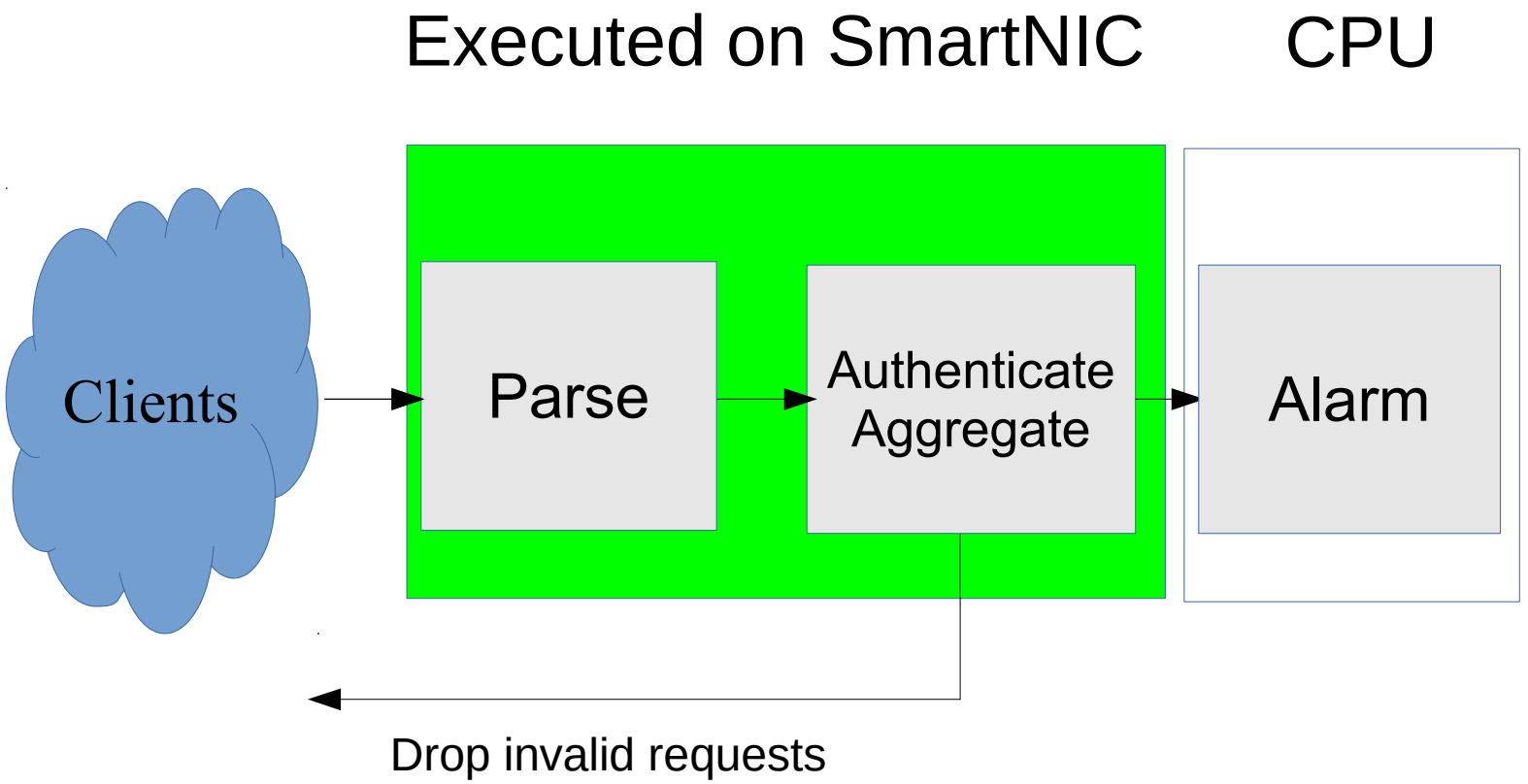
Our goal:
Server logic acceleration for cloud tenants

Example: Edge IoT server

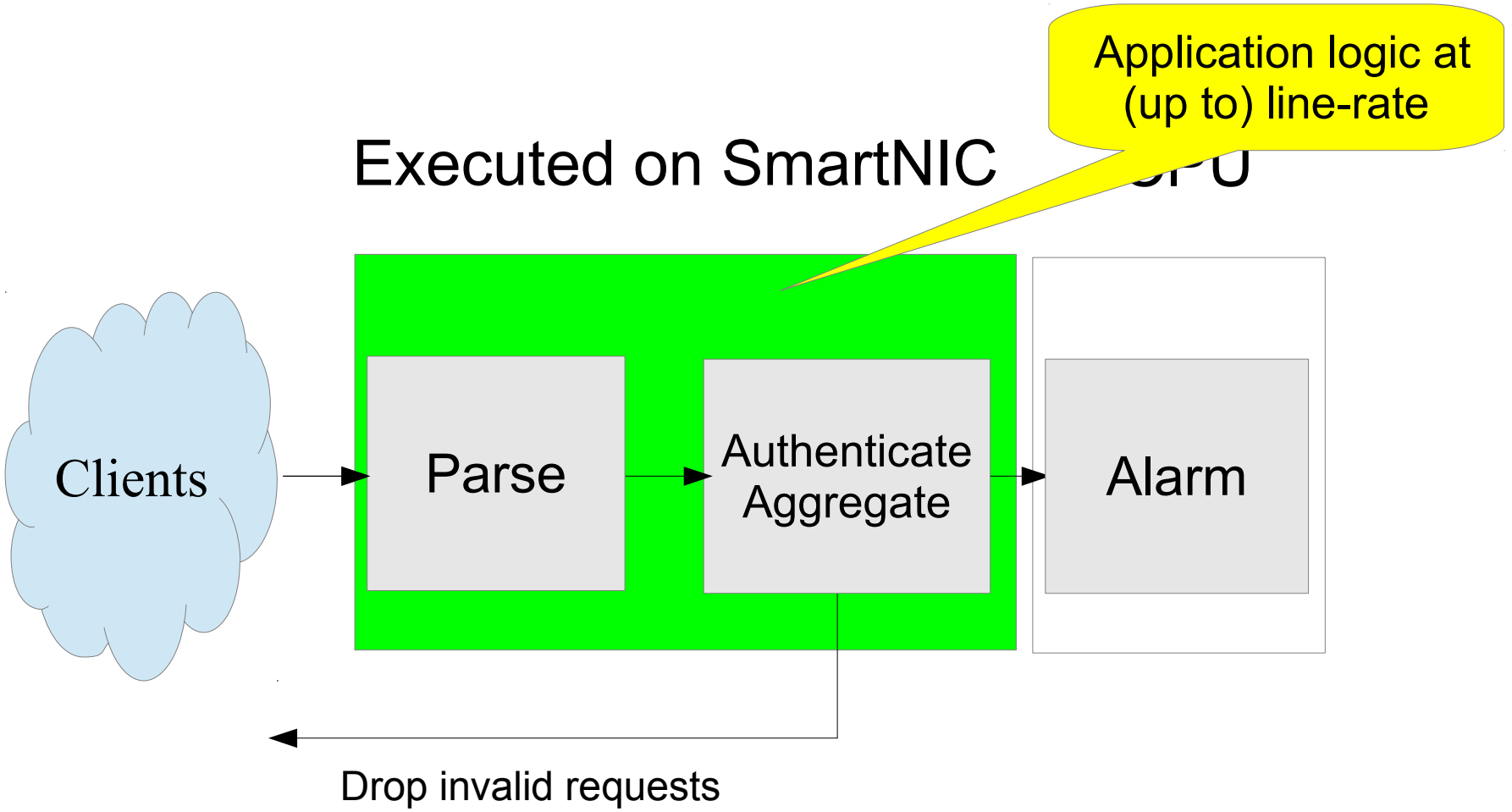
- Aggregate data from sensors, warn on anomaly



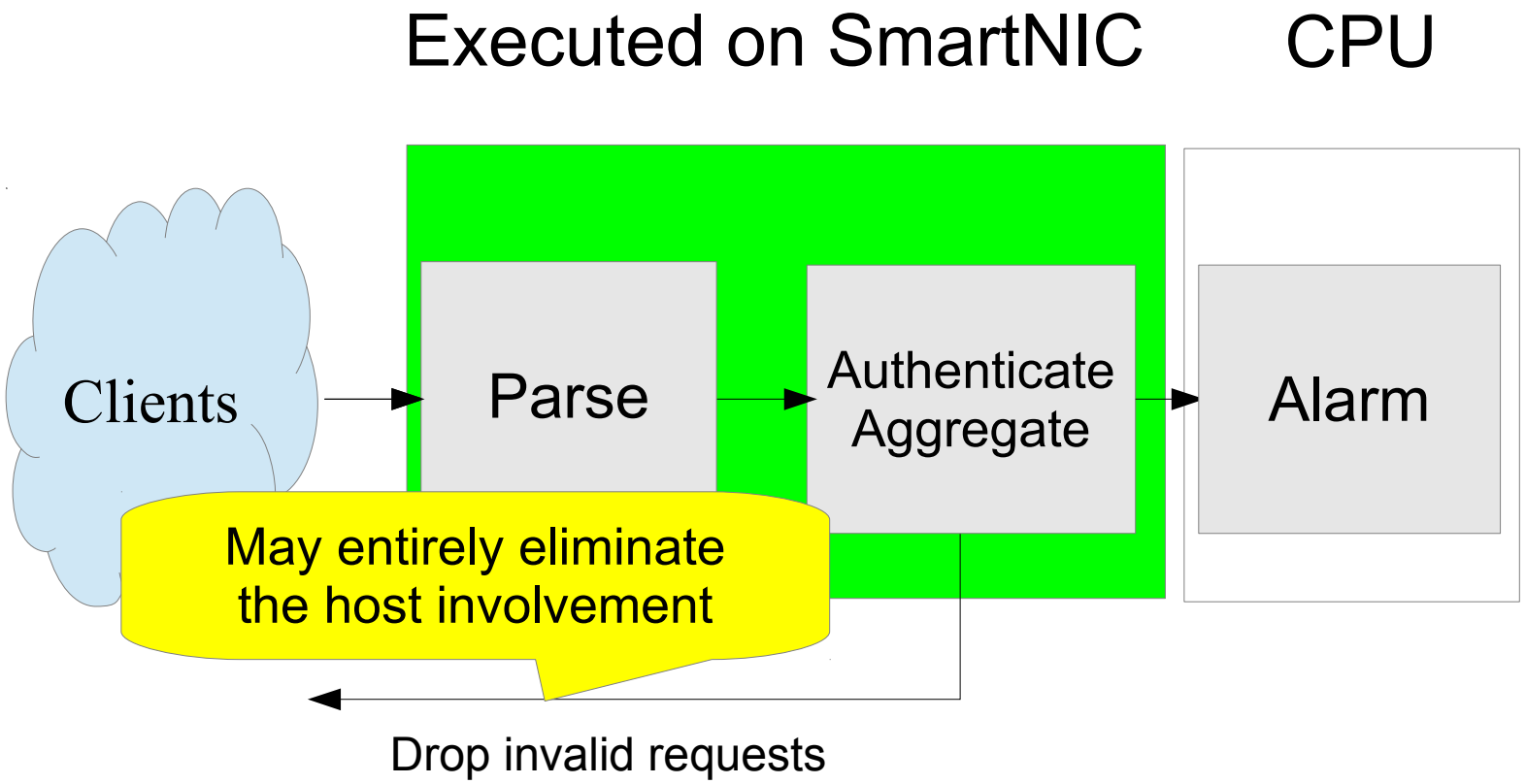
Example: IoT server **inline application** offload



Example: IoT server **inline application** offload



Example: IoT server **inline application** offload

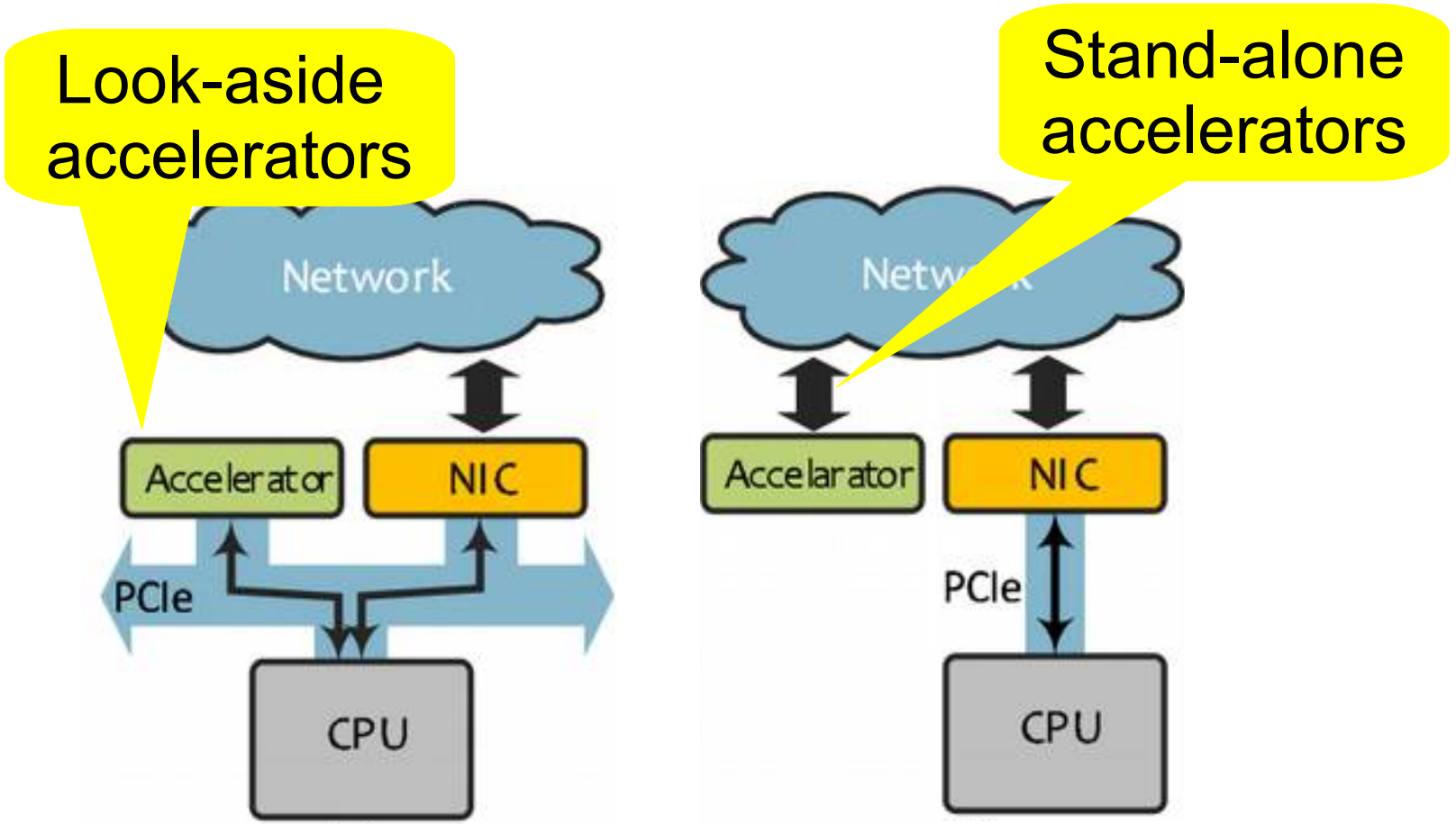


Goal: application offload

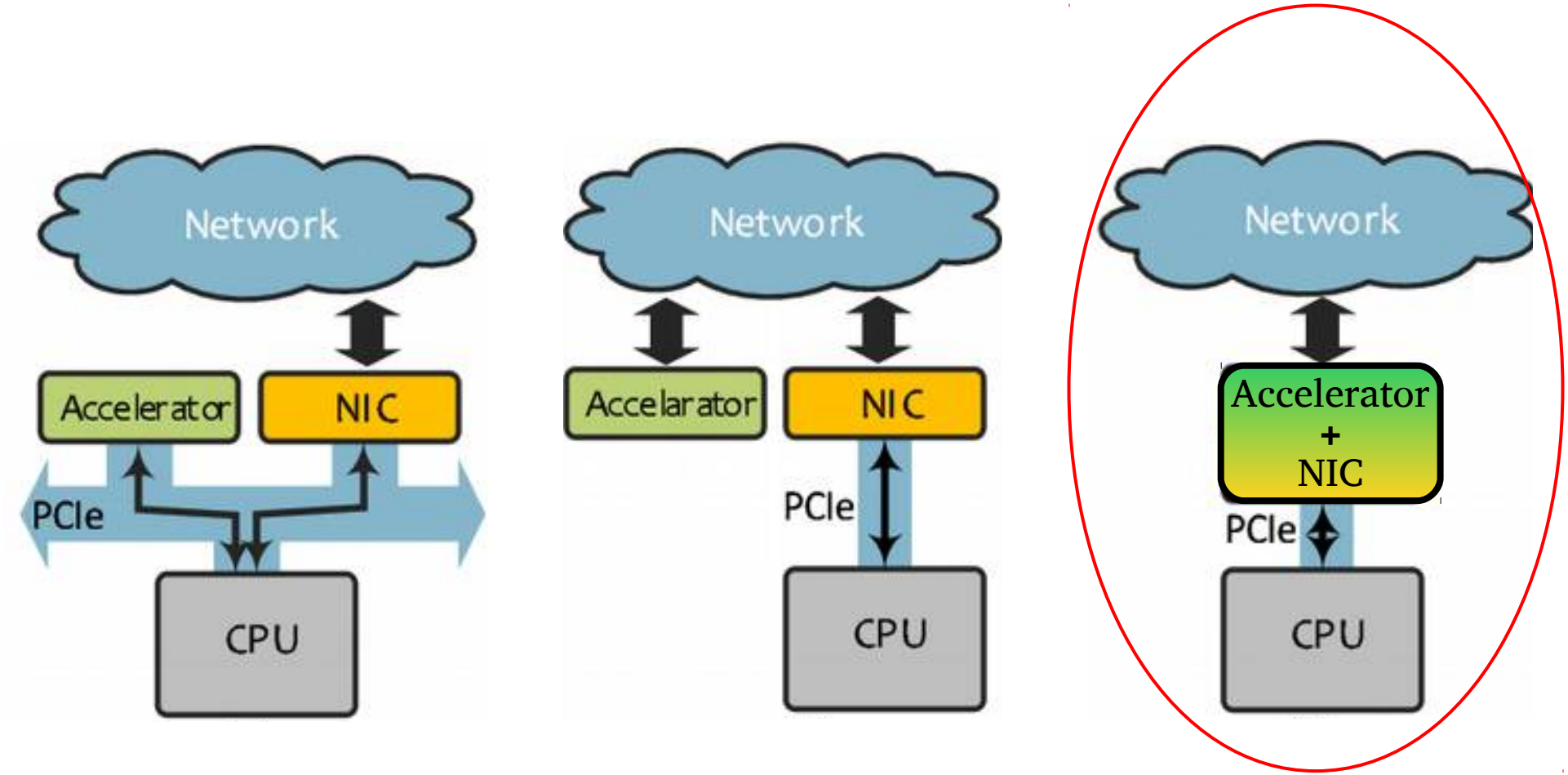
- Low, predictable latency
- High power efficiency
- Free CPU (and its caches) for other tasks
- Multi-tenancy support

These goals are essential for data center applications

Systems we discussed so far...



Inline acceleration is different!



***Transparent inline* acceleration**

Challenges

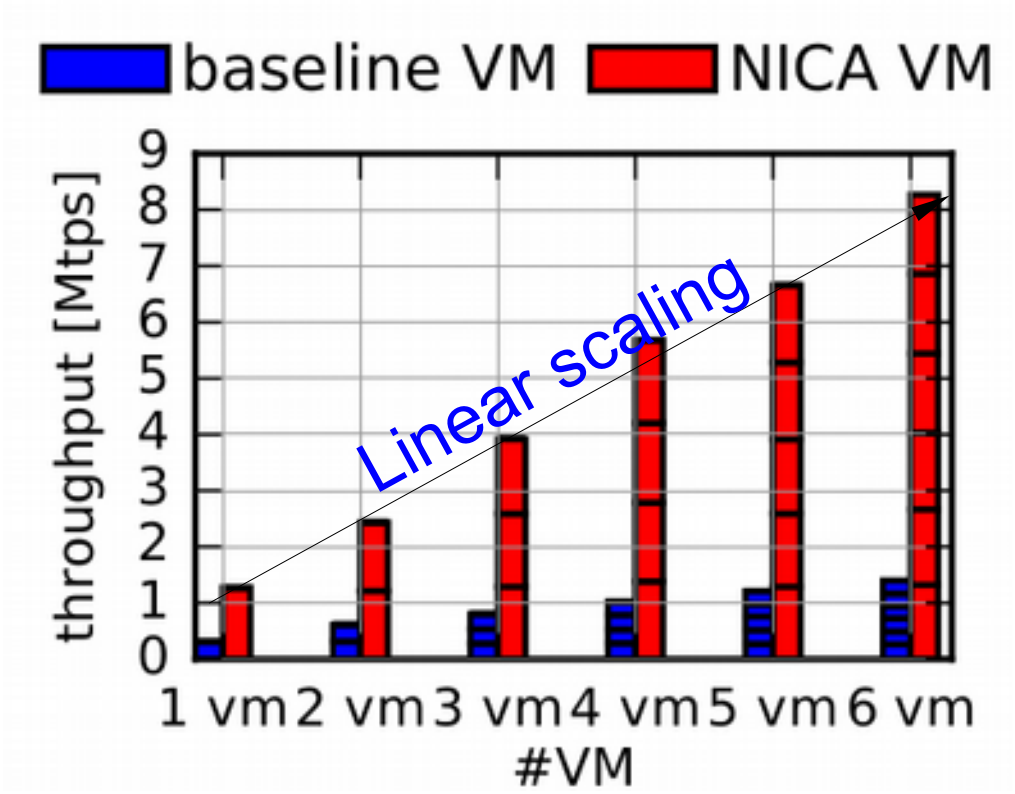
- Processing needs to be managed by the host
- No OS abstractions
- No protection, isolation, virtualization
- No integration with other accelerators

NICA: Application offload to SmartNICs in data centers

USENIX ATC19

- Insight:
  - Applications are complex, offload only performance-critical logic
- Our abstraction enables ***partial*** offloading
- Programming model tightly integrates with *sockets API*
- In-NIC runtime
- Virtualization support for performance isolation

Virtualized KVS cache in the NIC



- Only **130 lines** of code changed in `memcached`
- 2.1usec hit latency (20x faster than CPU)
- 5x speedup at 90% hit rate

Summary

- Future omni-programmable systems face **programmability wall**
- **Accelerator-centric OS architecture** simplifies programming and improves performance
- It exposes **OS abstractions on accelerators**
- Tightly integrates new abstractions **with the host OS**

Summary

- Future omni-programmable systems face **programmability wall**
- **Accelerator-centric OS architecture** simplifies programming and improves performance
- It exposes **OS abstractions on accelerators**
- Tightly integrates new abstractions **with the host OS**

Same principles are useful for SGX [Eurosys17,USENIX ATC19] and disaggregated data centers [SFMA19, ongoing]

Code available @ https://github.com/acsl-technion

OmniX is an ongoing work in

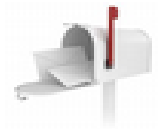
Haggai Eran, Amir Watad, Shai Bergman, Tanya Brokhman, Lior Zeno, Maroun Tork, Meni Orenbach, Lev Rosenblit, Alon Rashelbach, Pavel Lifshits, Gabi Malka, Lina Maudlej

OmniX is an ongoing work in

Haggai Eran, Amir Watad, Shai Bergman, Tanya Brokhman, Lior Zeno, Maroun Tork, Meni Orenbach, Lev Rosenblit, Alon Rashelbach, Pavel Lifshits, Gabi Malka, Lina Maudlej

But only if you are ready to climb higher!

Thank you!



mark@ee.technion.ac.il
https://sites.google.com/site/silbersteinmark