Technion
Israel Institute of Technology

ACSL
Accelerated Computing Systems Lab

Accelerator-centric system design

Goals

- Learn the principles of hardware and software design of *programmable* accelerators
- Learn the principles of software-hardware interaction between accelerators and CPUs
- Learn the principles of accelerator-centric systems

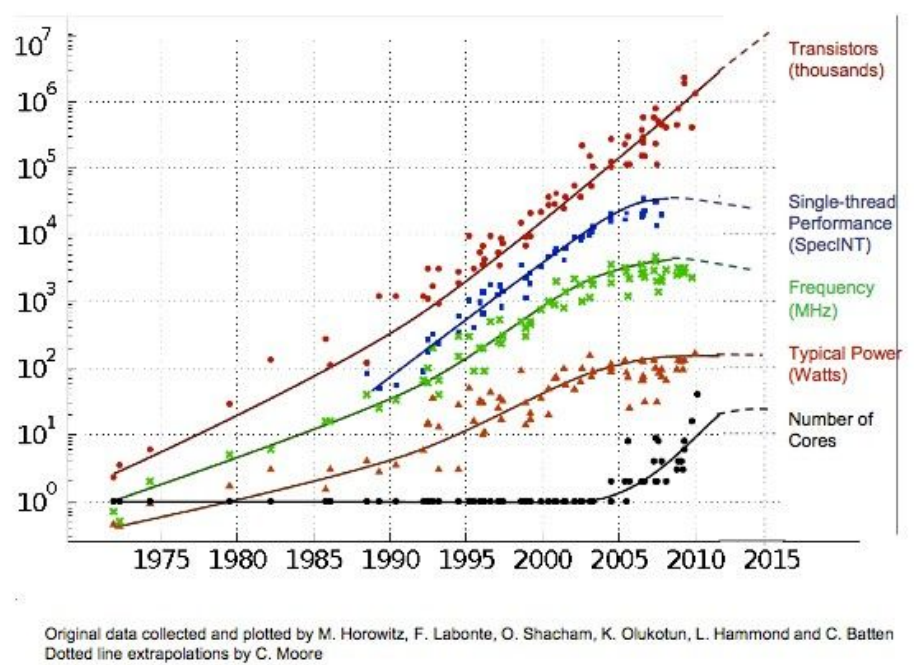
Administration

- Instructor: Mark Silberstein - Technion
- Email: mark@ee.technion.ac.il
- Website: https://marksilberstein.com
- Group Website: https://acsl.group

Introduction

- Why accelerators: trends in computer architecture
- Amdahl's law and the case for multi-accelerator
- Taxonomy of accelerators
- GPU as an example

Moore's law and CPU performance



More transistors != more performance

- Dennard scaling: power density remains constant as we shrink transistors, but they become faster!

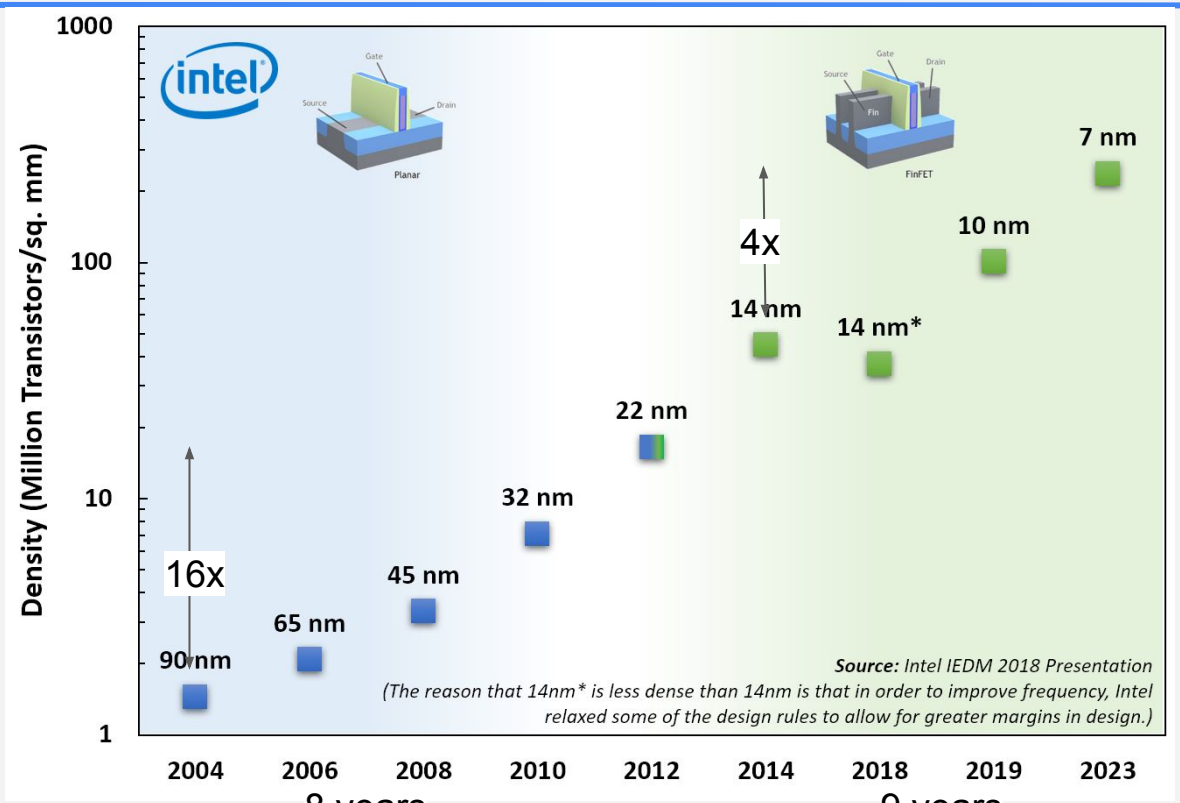
Table 1
Scaling Results for Circuit Performance

Device or Circuit Parameter	Scaling Factor
Device dimension t_{ox}, L, W	$1/\kappa$
Doping concentration N_a	κ
Voltage V	$1/\kappa$
Current I	$1/\kappa$
Capacitance $\epsilon A/t$	$1/\kappa$
Delay time/circuit VC/I	$1/\kappa$
Power dissipation/circuit VI	$1/\kappa^2$
Power density VI/A	1

From: «Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions», 1974

Dynamic leakage killed Dennard scaling @ ~2006

Transistor scaling is much slower



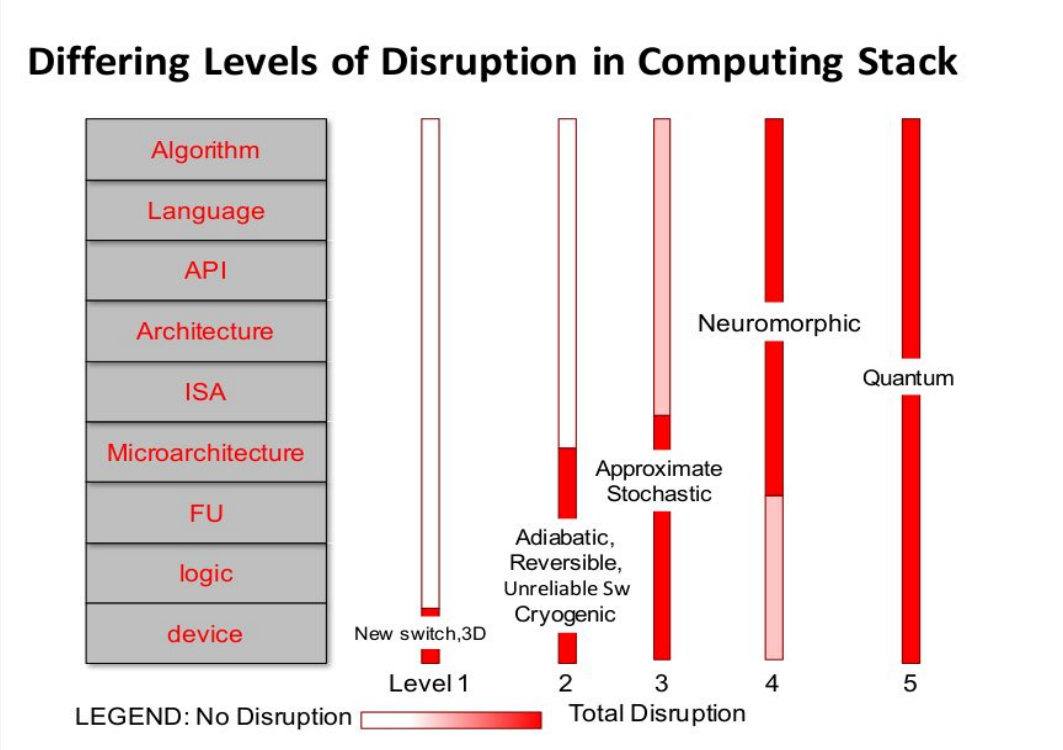
Transistor scaling is non-economical

- 7nm — today (difficult)
- 5nm – not yet available
- 3nm and down – not clear

Looking far beyond CMOS

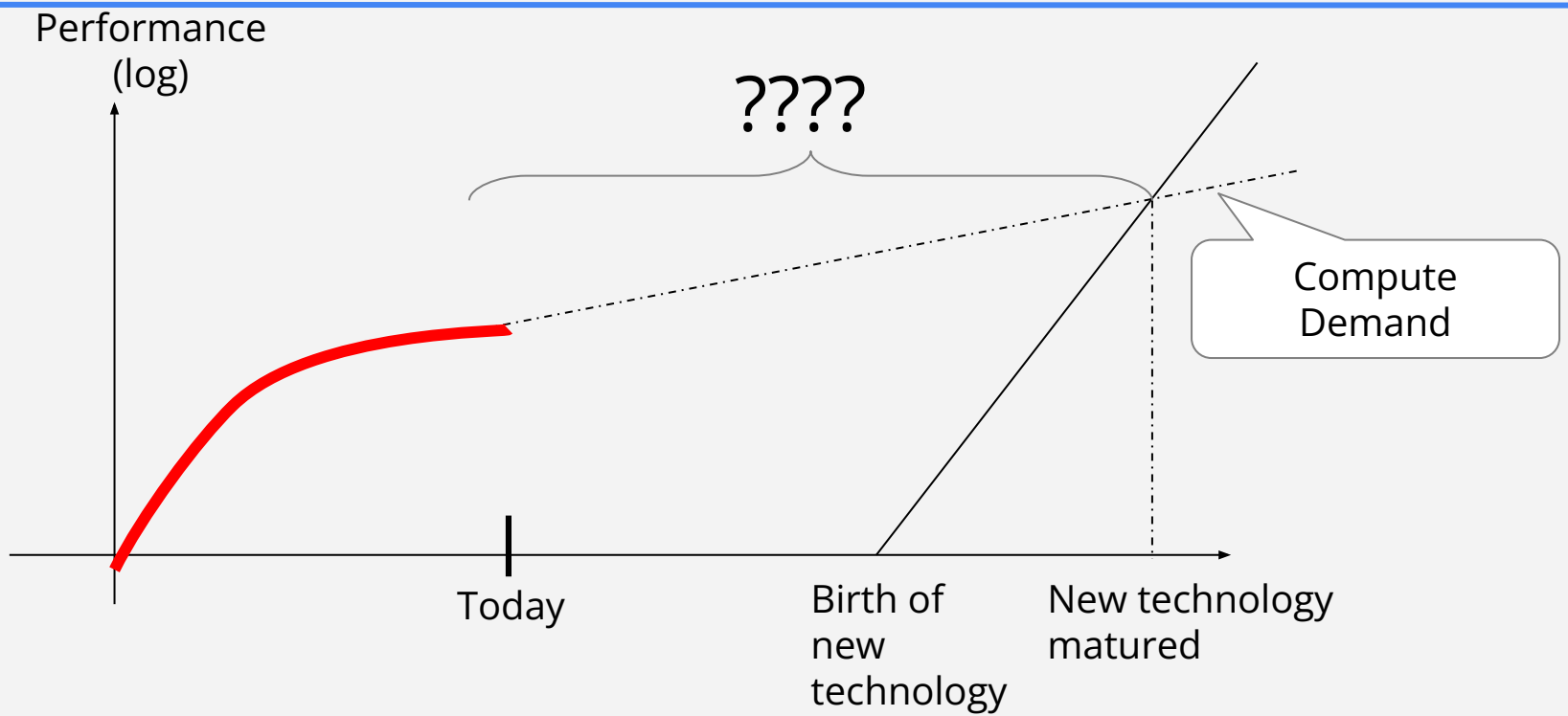
- Cryogenic computing
- Approximate/stochastic computing
- Neuromorphic computing
- Biological computing/storage
- Quantum computing

Looking far beyond CMOS



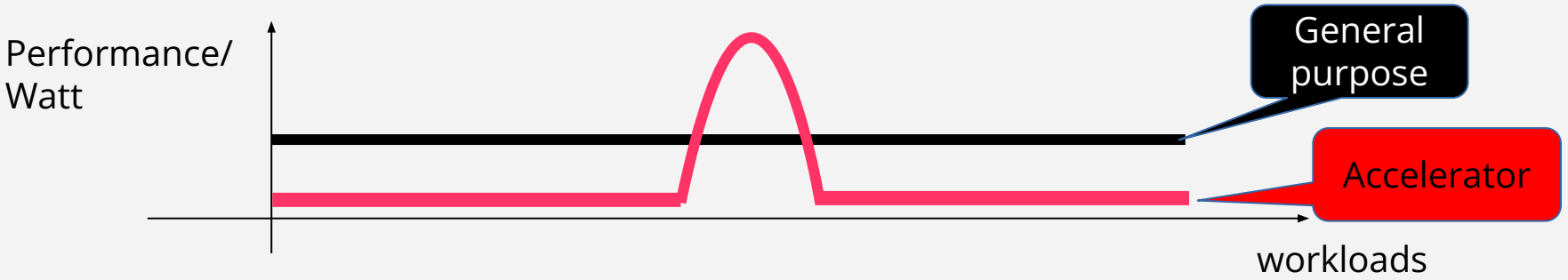
From «IEEE rebooting computing»

What to do until the next revolution?



Accelerators

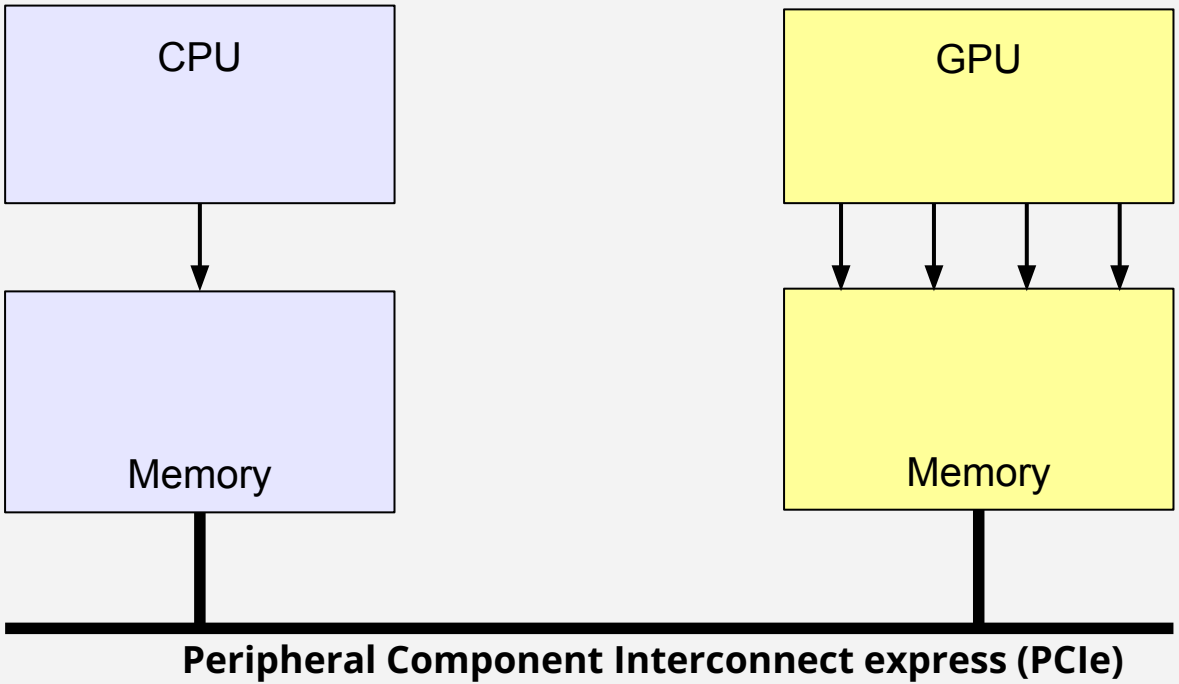
Special-purpose processing units which improve performance of ***specific*** workloads



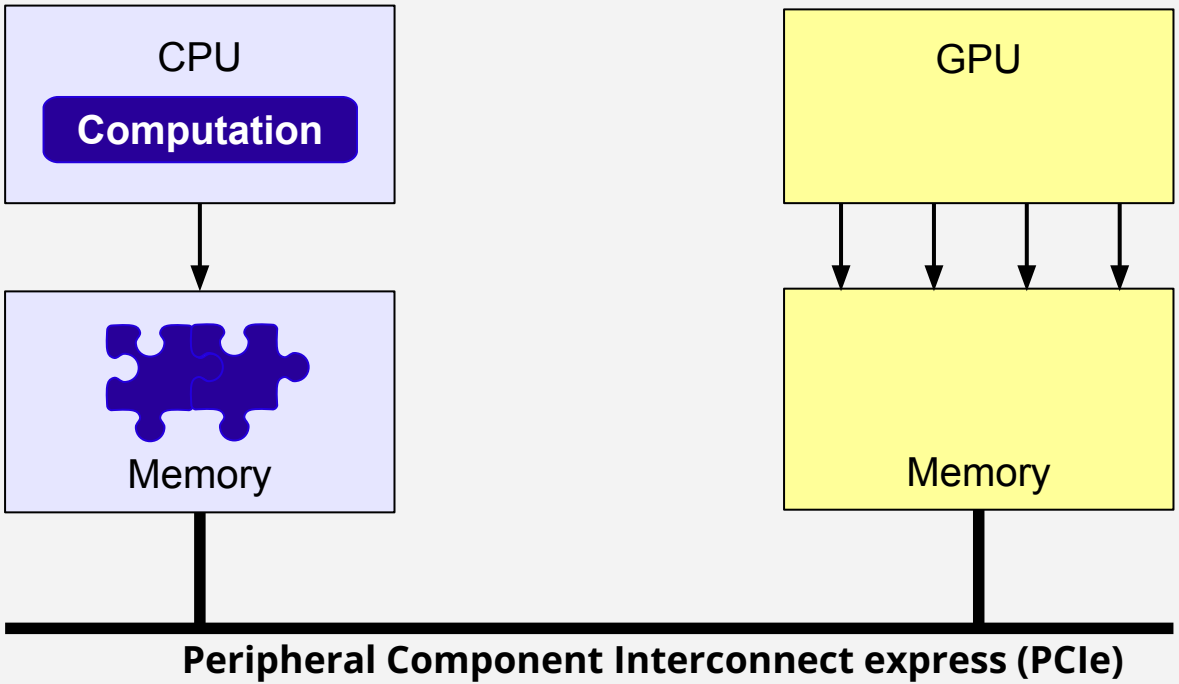
Accelerators: co-processors GPU as an example

- Offloading large parallel tasks for faster execution
- Local state for intermediate results
- Applications run on the host

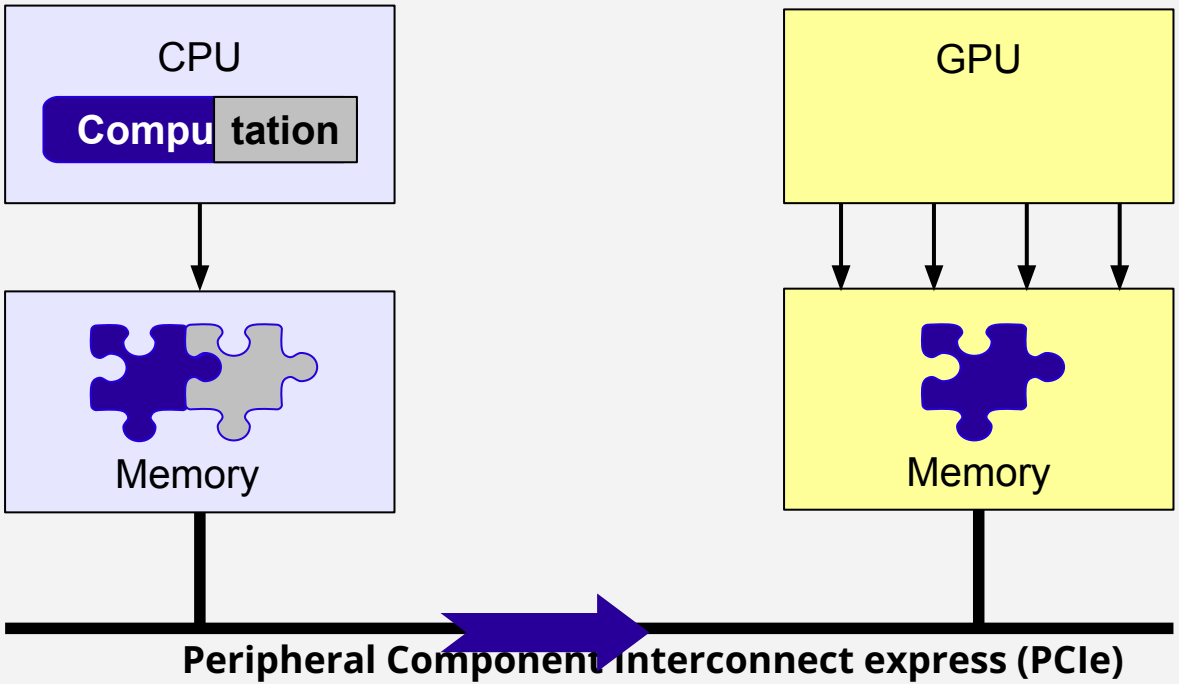
GPU is a co-processor



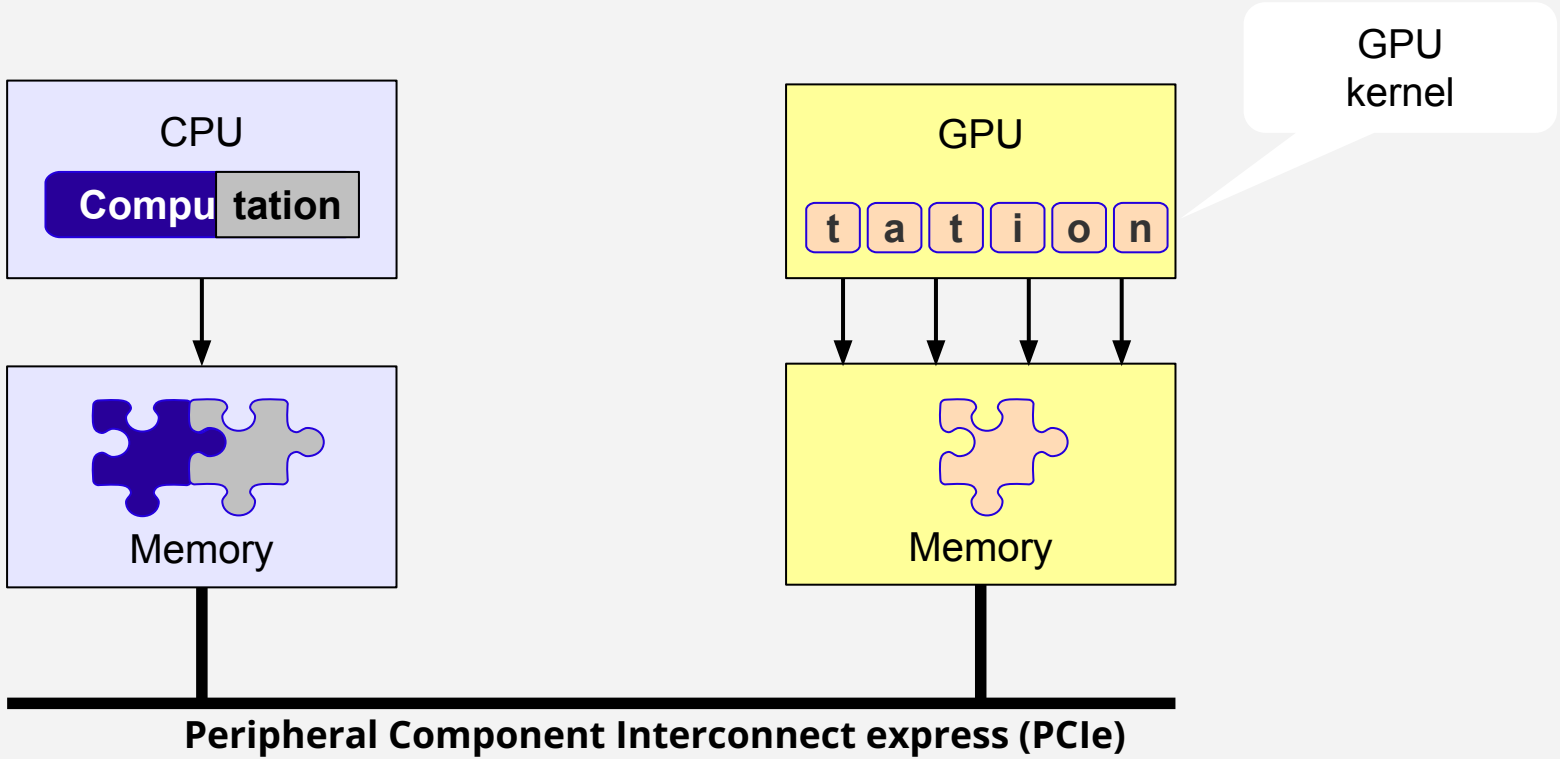
GPU is a co-processor



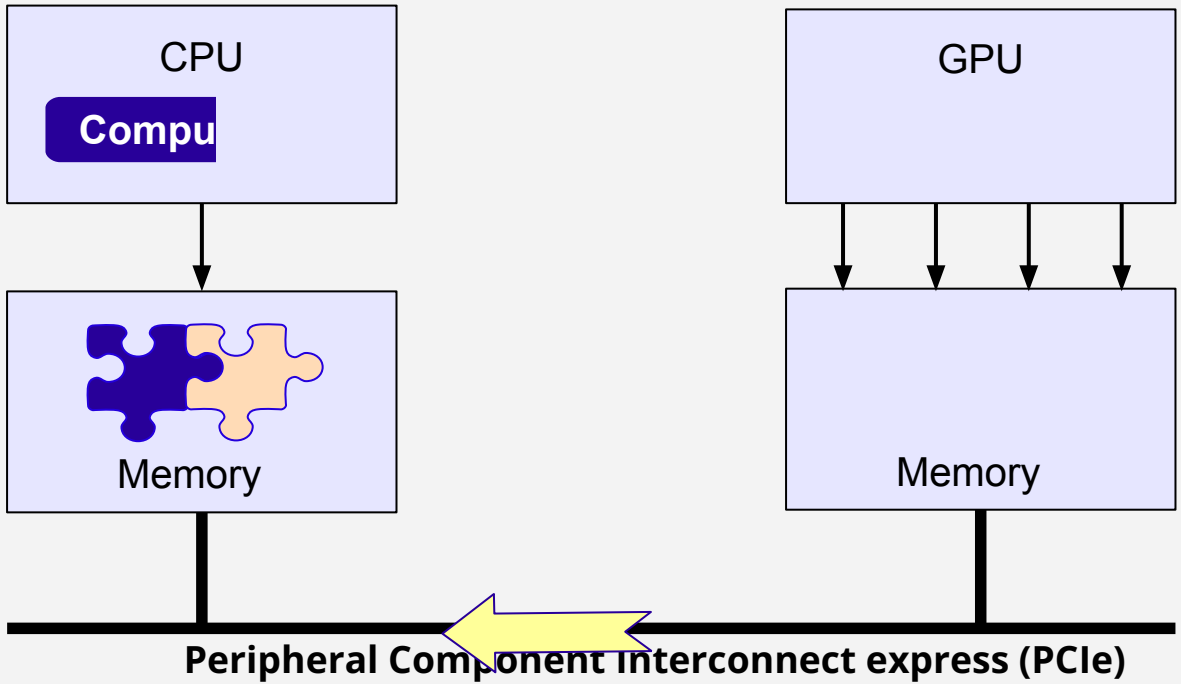
GPU is a co-processor



GPU is a co-processor

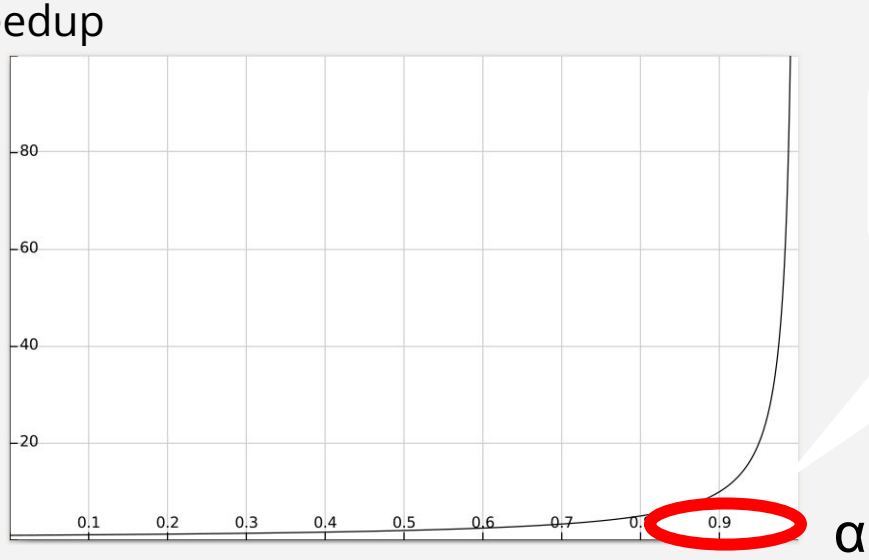


GPU is a co-processor



Bad news: Amdahl's law

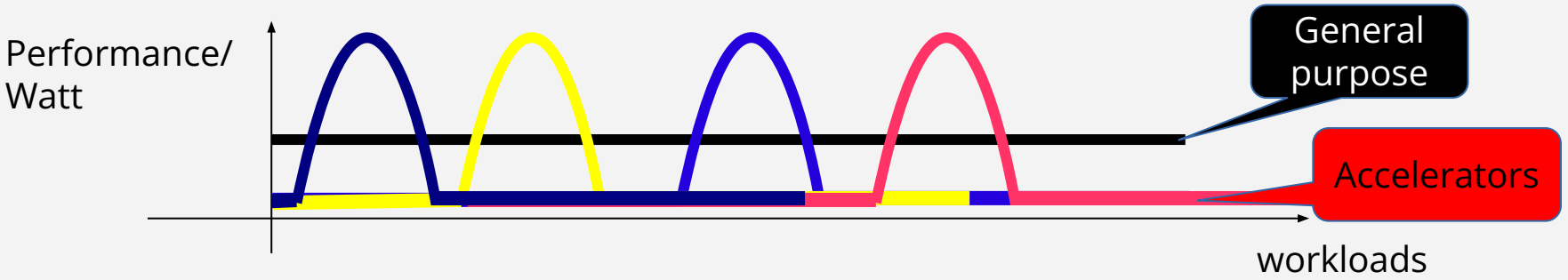
- α — acceleratable part, 1-α — non-acceleratable
- Maximum speedup = 1 / (1-α)



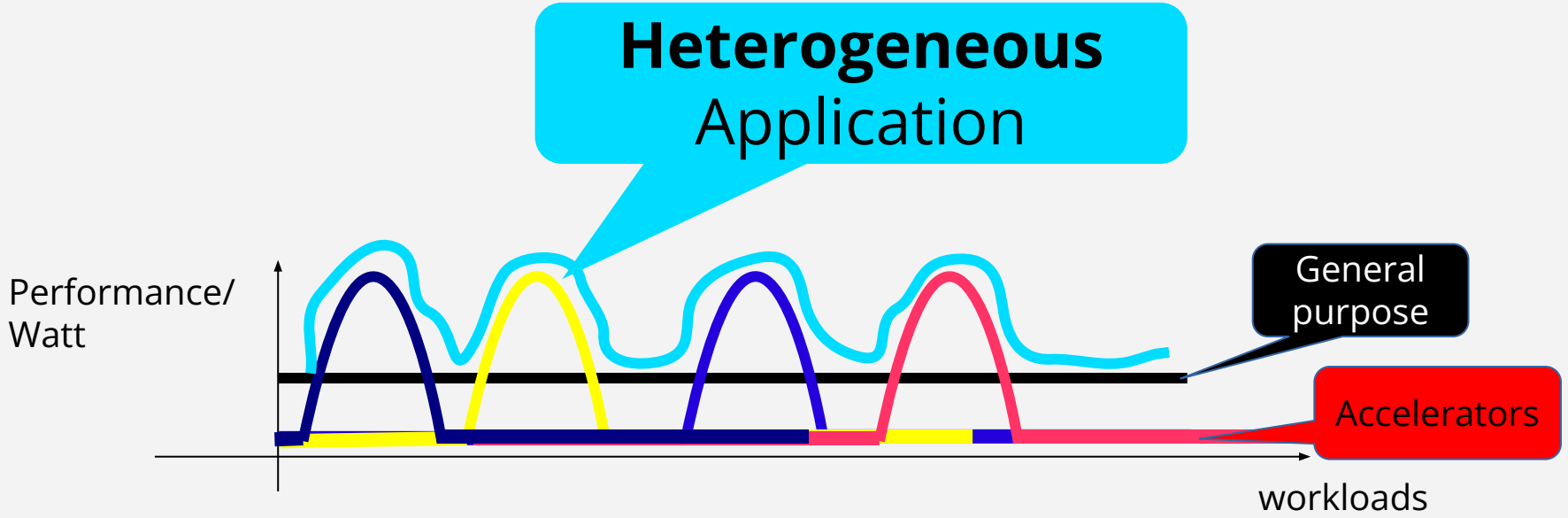
90% acceleratable but speedup at most x10

Solution: multiple accelerators

- Broader applicability
- Accelerators for common tasks



Application developers match tasks with accelerators

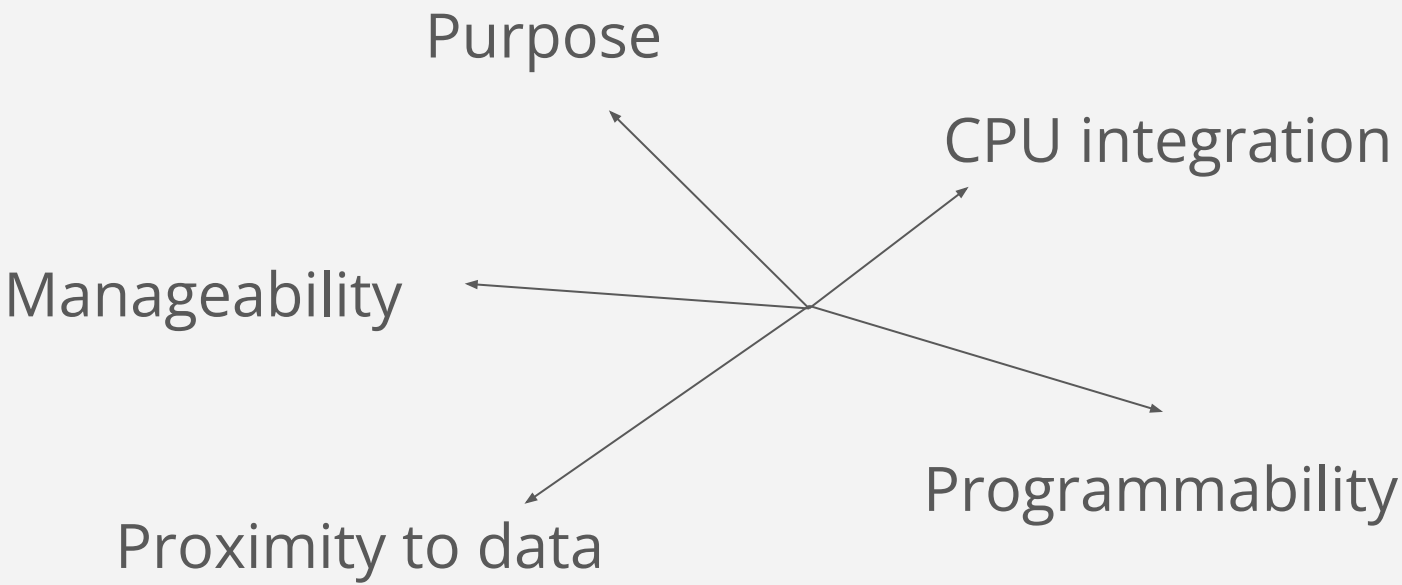


Additional reading

- Amdahl's law in the Multi-Core Era, MD Hill et al
- Many-core vs. Many-thread machines: stay away from the valley, Z. Guz et al
- Rebooting computing: the road ahead, TM Conte
- What is the future of technology scaling, D Brooks
- Validity of the single processor approach to achieving large scale computing capabilities, GM Amdahl
- Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions, R Dennard
- A 30 Year Retrospective on Dennard's MOSFET Scaling Paper, M Bohr

Survey of accelerators

Axes of analysis



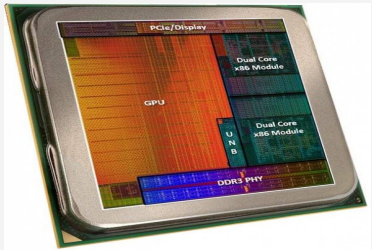
Purposes

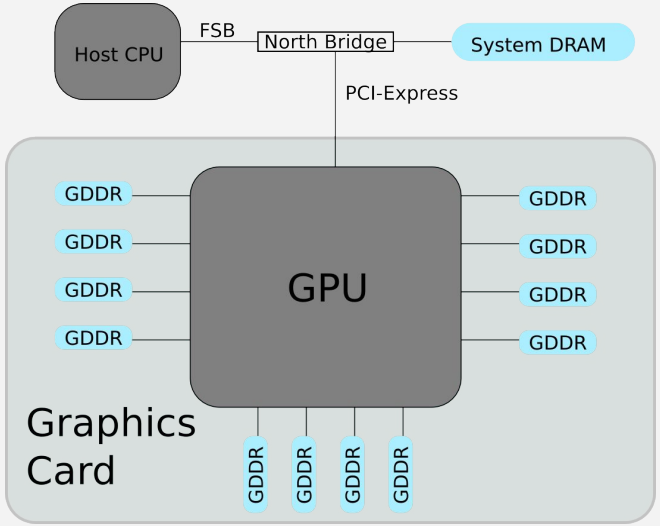
- I/O accelerators
  - E.g., high performance NICs, storage
- Security accelerators
  - E.g., IBM secure processor
- Computational accelerators
  - E.g., GPUs, FPGAs, DSP
- Sensors/media/communication accelerators
  - Codecs, GPS, mobile

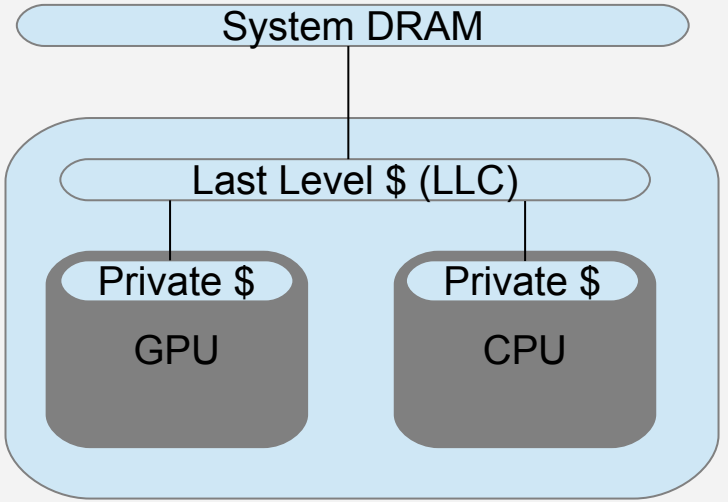
CPU integration

- Discrete accelerators
  - Connected to the host via internal bus
  - Separate local memory
  - Examples: discrete GPUs, NICs
- Integrated accelerators
  - On-die with the host CPU
  - Shared physical (and sometimes virtual) memory
  - Examples: integrated GPUs

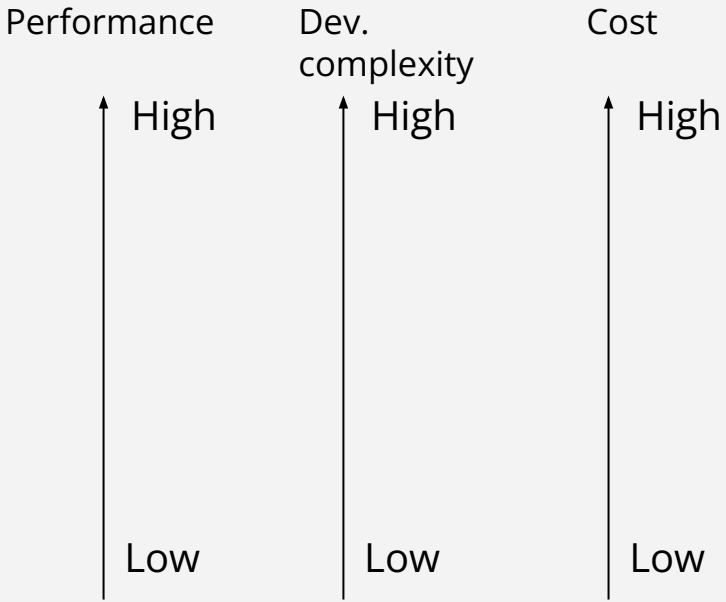
Example: discrete vs. hybrid GPUs







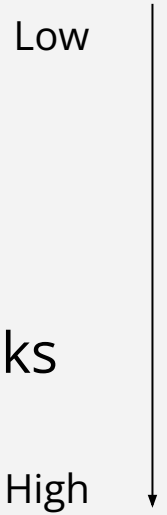
Programmability



ASIC: fixed function

FPGA: software-defined hardware

Programmable: programmable tasks



Example: linear algebra accelerators

- ASICs ~1000x faster/W than CPU
  - Do specific tasks extremely fast
  - But! takes ~$MIns to tape out
- FPGAs ~100x faster/W
  - Non-Von-Neumann architectures
- GPUs ~10x faster/W than CPU
  - No new hardware

Proximity to data

- Near-data accelerators: exploit high bandwidth to data, reduce data movements
  - Processing in storage
    - SSD controller: 16-core ARM processor
  - Processing in memory
    - Micron “automata”
  - Processing in network
    - Programmable Switches
- Compute accelerators: exploit special local memory architecture

Manageability

- By CPU
  - Accelerators do not have privileged mode
    - GPUs
- Self-managed
  - Runs an embedded OS
    - BlueField SmartNIC runs complete Linux
- Combined

Manageability: Invocation and execution

- Inline
  - Bump-in-the-wire SmartNICs: performs processing on every packet without involving CPU to be invoked
- Look-aside
  - GPU: explicitly invoked by the CPU to perform specific tasks

Accelerators vs. ISA extension

- Coarser-grain tasks
- Asynchronous
- Large private state
- Managed by drivers
- Dynamically scheduled
- Preemptable

- Fine-grain tasks
- Mostly synchronous
- State shared w/ CPU
- Invoked directly
- Mapped at compile-time
- Atomic (non-preemptable)

Questions for self-study [1]

1. Why Amdahl's law calls for introducing more accelerators?
2. Is there any inherent tradeoff between programmability and performance?
3. What is the difference between inline accelerator and look-aside accelerator?
4. Why are look-aside accelerators usually most efficient for coarse-grain tasks?

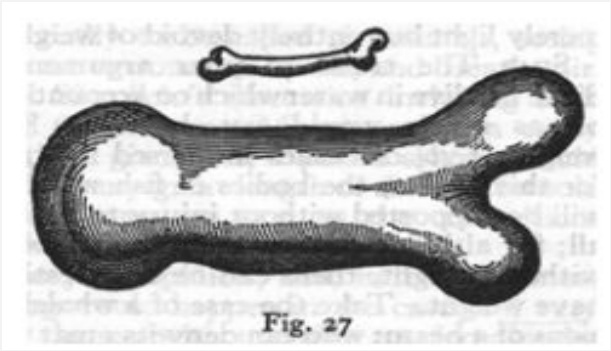
Why building systems with accelerators is hard?

- Incommensurate scaling
- Emerging complexity to lower costs
- Hidden properties

Incommensurate scaling

Different components scale differently

Galileo, 1638



What would happen to a mouse if it grew to the size of an elephant?

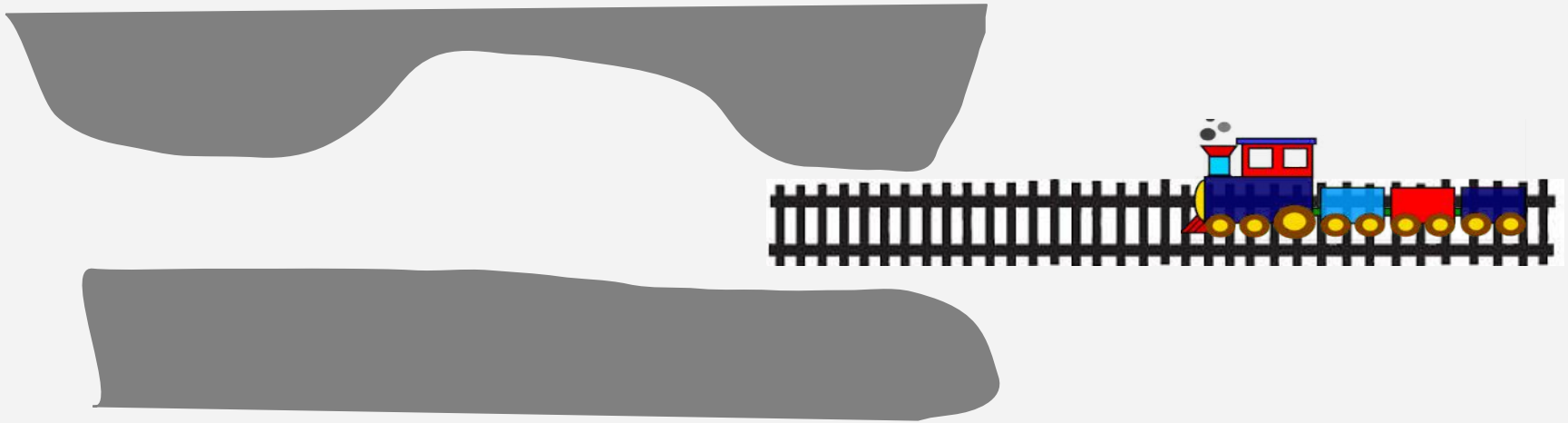
On Being the Right Size (Haldane 1928)

- Scaling mouse to size of an elephant
  - Volume ~ $O(n^3)$
  - Bone strength ~ cross section ~ $O(n^2)$
  - Mouse design will collapse

An elephant needs a different **design** than a mouse

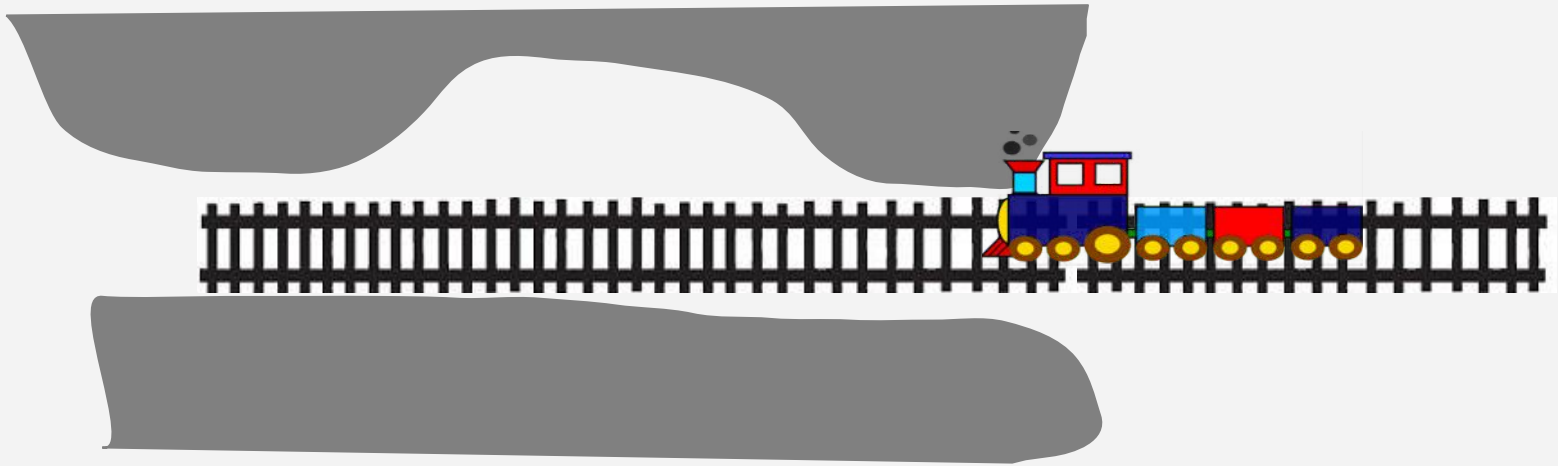
Accelerating part of a system results in incommensurate scaling and exposes new tradeoffs that require **full redesign**

Performance → complexity → unexpected consequences



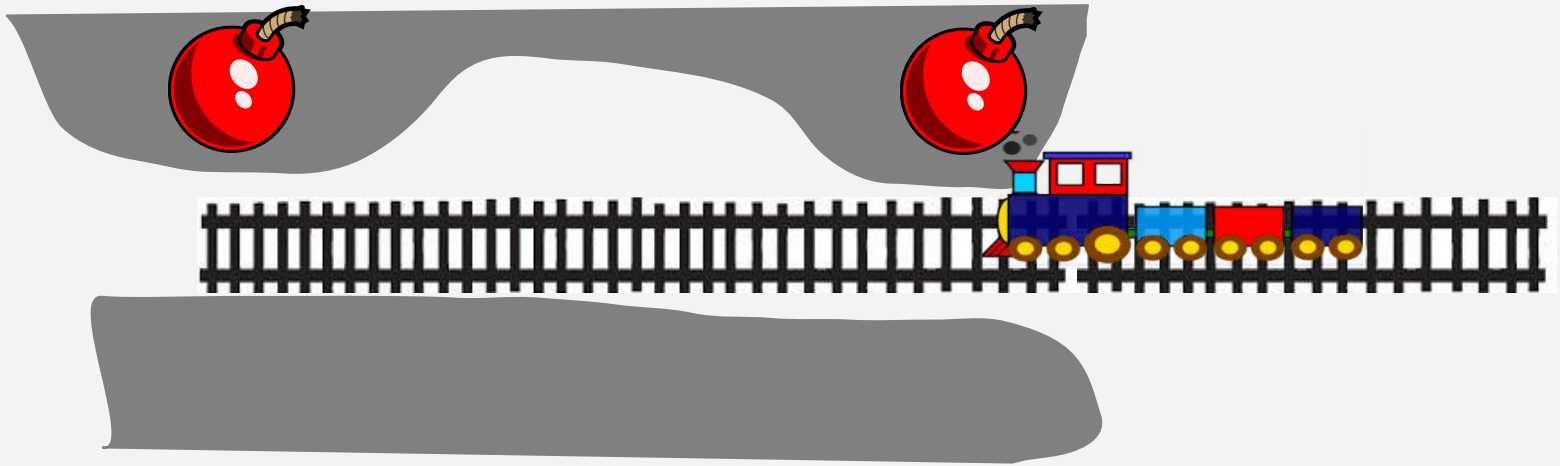
Want to build railway, but mountains interfere

Cheap solution is inefficient



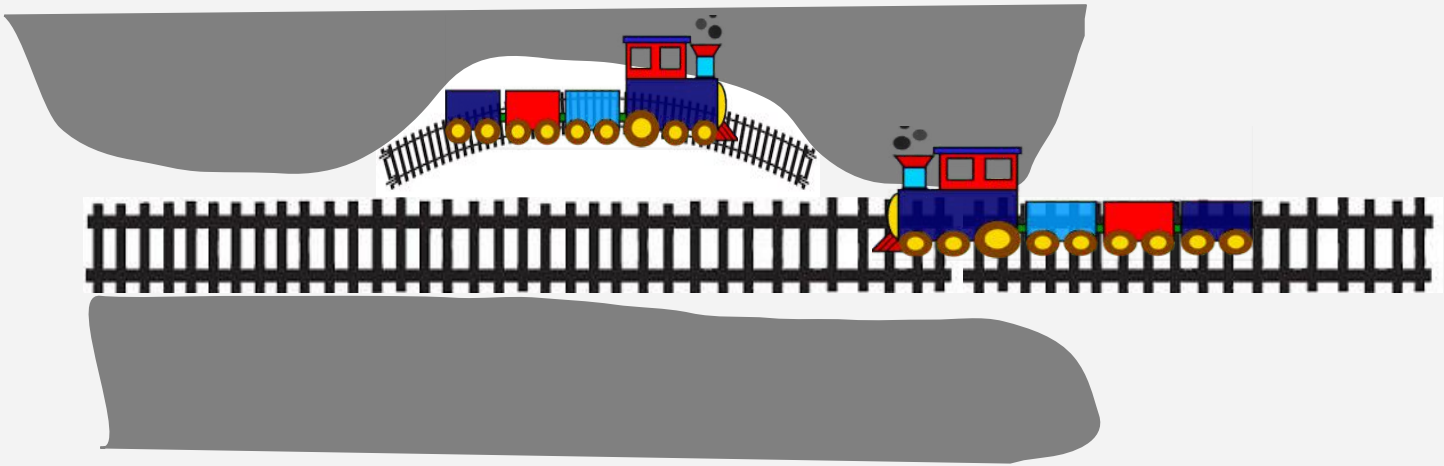
Simple but slow: one train at a time

Can we make it faster?

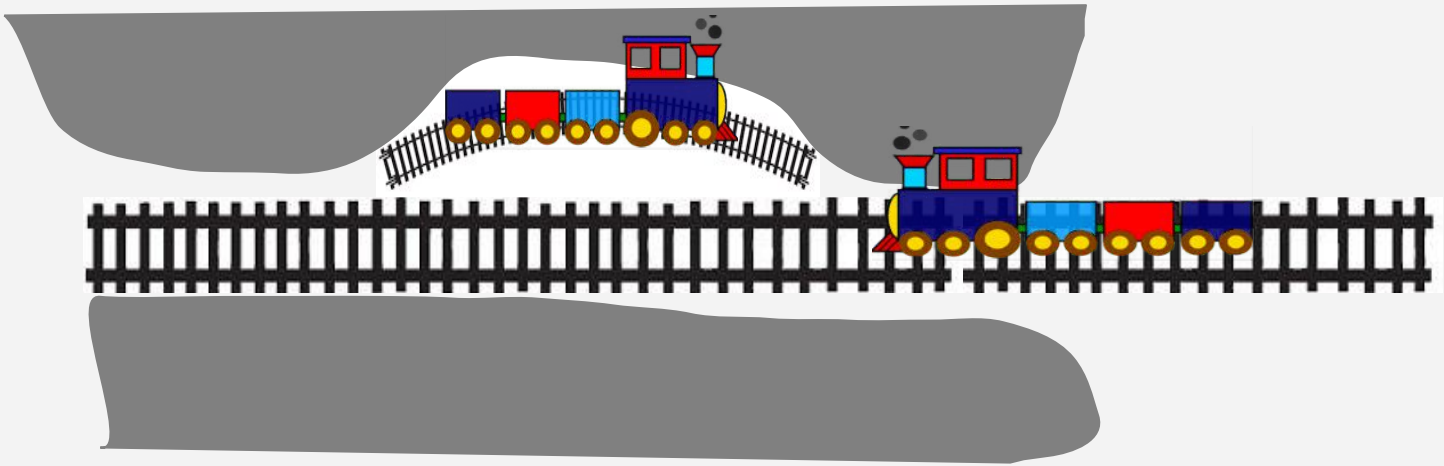


Optimal solution is costly!

Compromise: cheaper than optimal, faster than one track

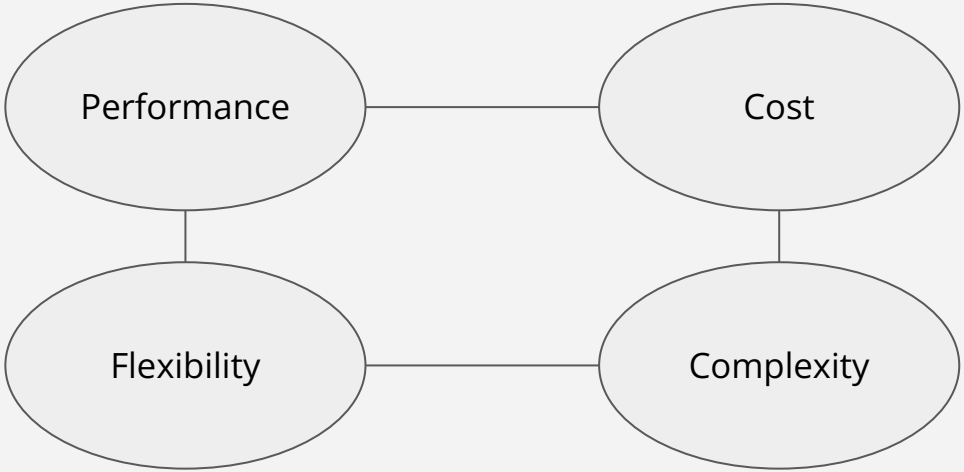


But significantly more complex



One hidden global requirement is introduced!
Which one?

Engineering is the art of trade-offs

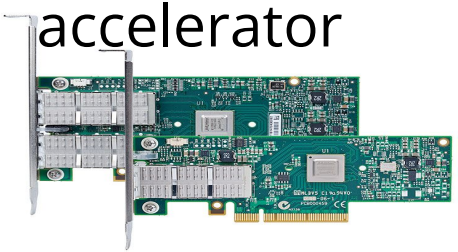


With enough money and time, you would always build a special-purpose ASIC for each task

How do accelerators influence the system design?

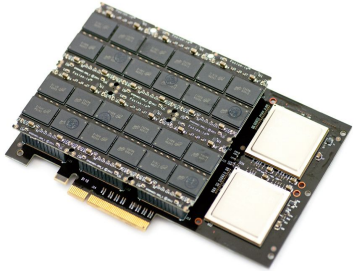
Computer hardware today

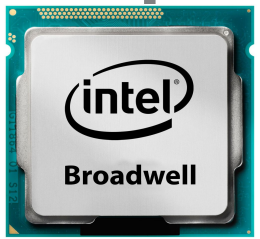
Network I/O accelerator



GPU parallel accelerator

Storage I/O accelerator

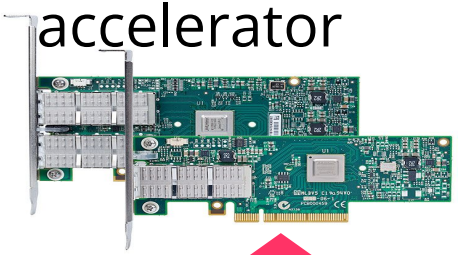




Accelerators for
crypto, media, signal processing....

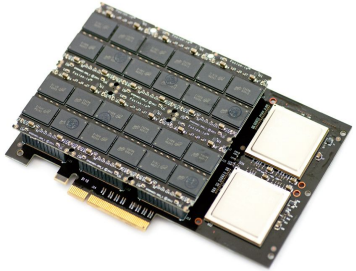
Computer hardware today

Network I/O accelerator



GPU parallel accelerator

Storage I/O accelerator



Power
Performance
Programmability

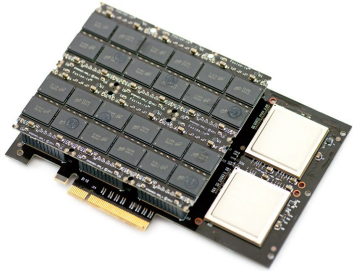
Accelerators for
crypto, media, signal processing....

But the system stack stayed unchanged!

Network I/O accelerator

GPU parallel accelerator

Storage I/O accelerator

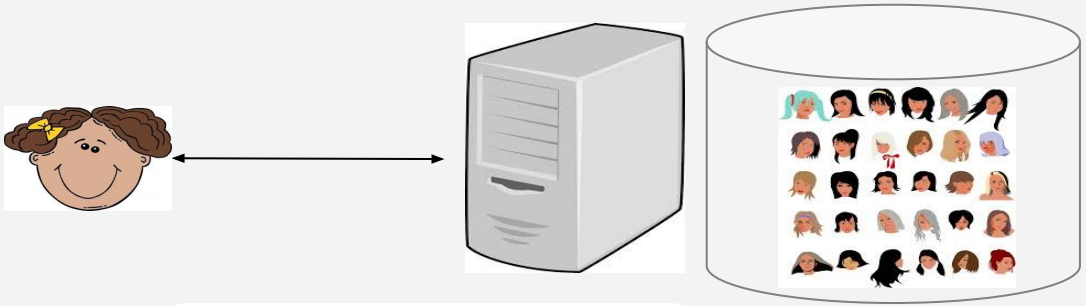


Power
Performance
Programmability

Accelerators for
crypto, media, signal processing....

Example: image server

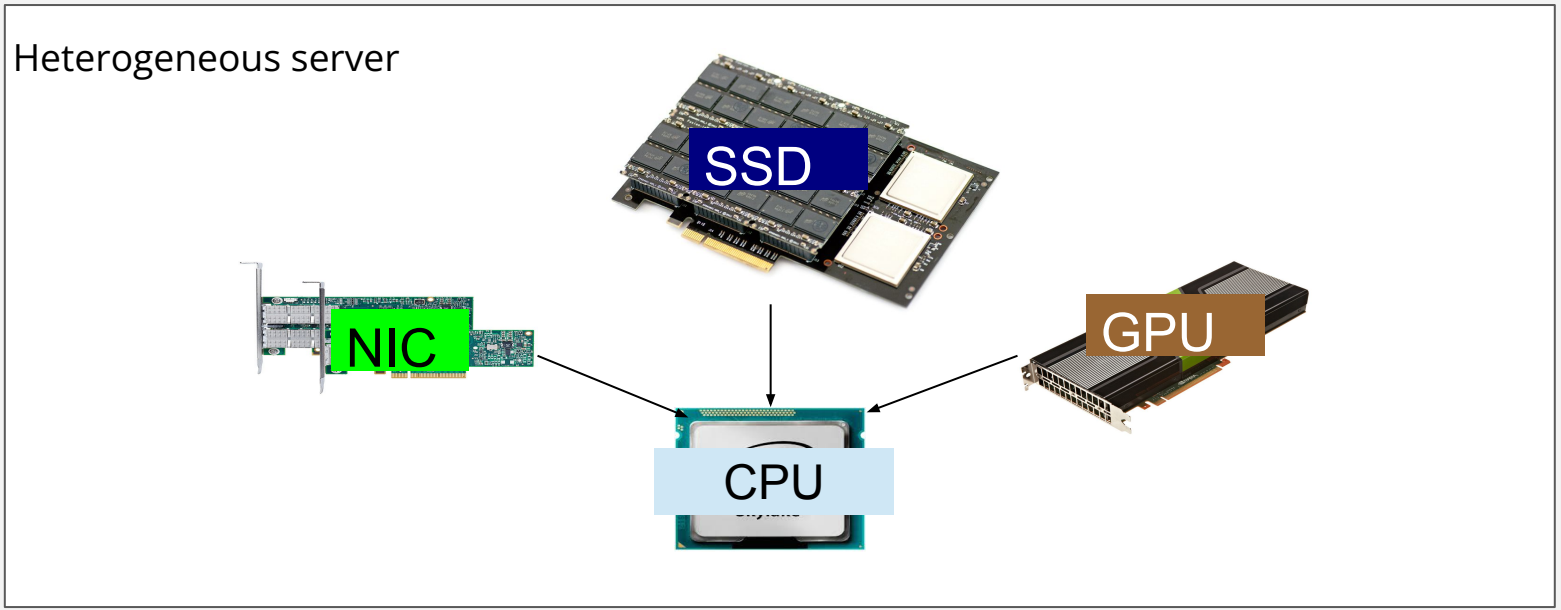
1. put: parse → contrast-enhance → store
2. get: parse → resize → store → serialize



Similar architecture used in Flickr

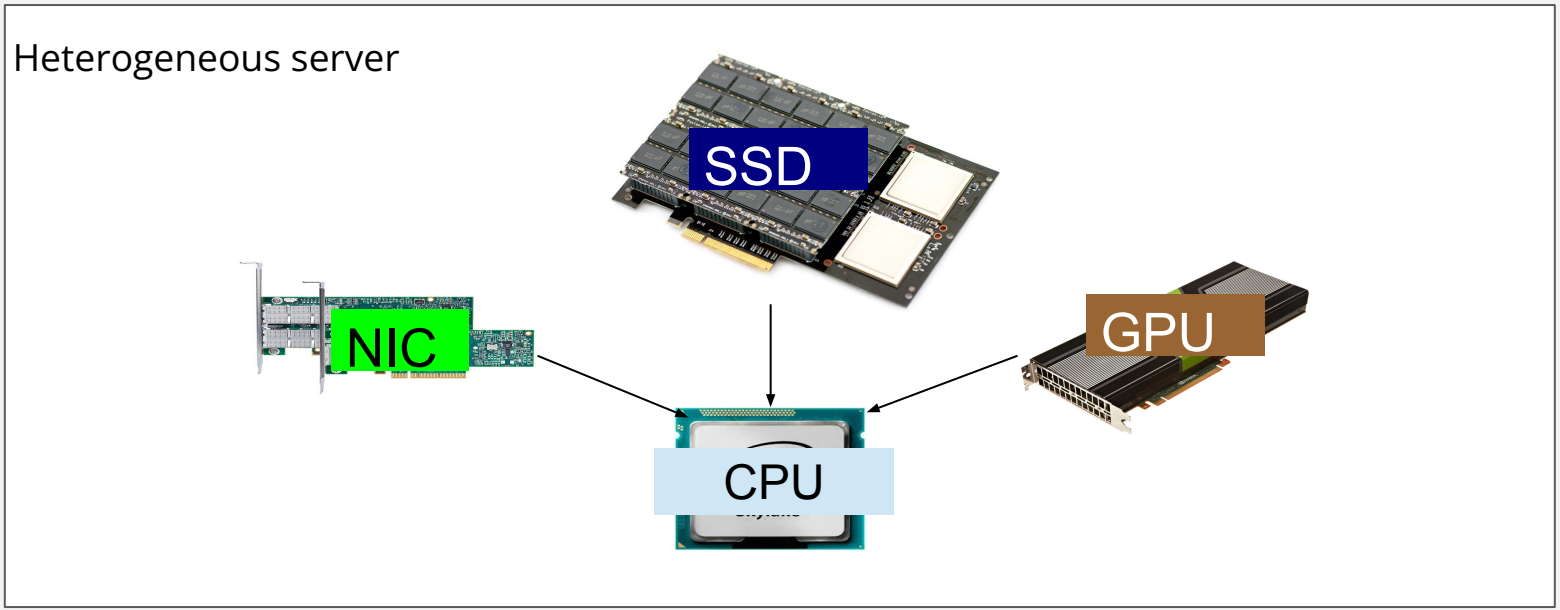
Example: image server

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



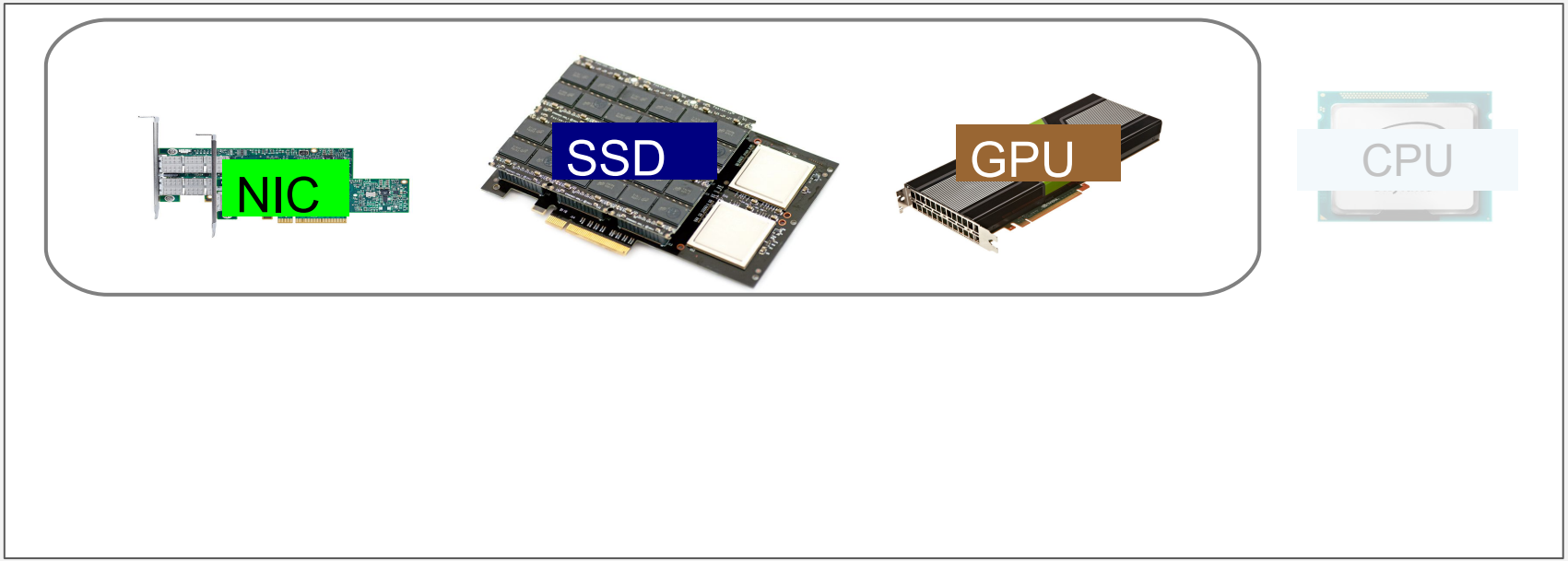
Lets use accelerators!

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



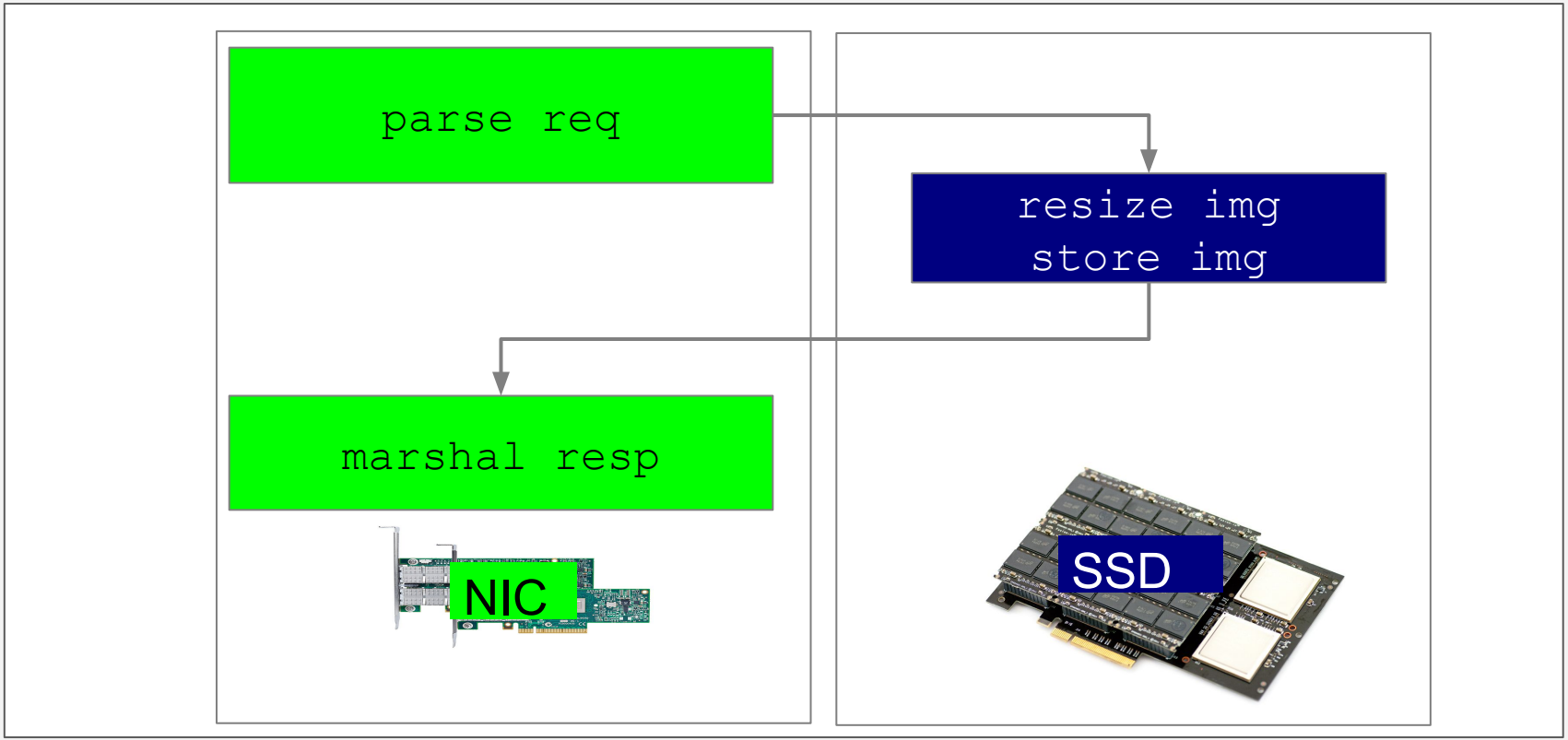
Using accelerators – no need for CPU!

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



Closer look at *get*

parse → **resize** → **store** → serialize

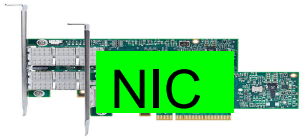


But OS services run on CPUs

get: parse → **resize** → **store** → serialize

parse req

marshal resp

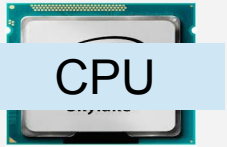


recv(req)

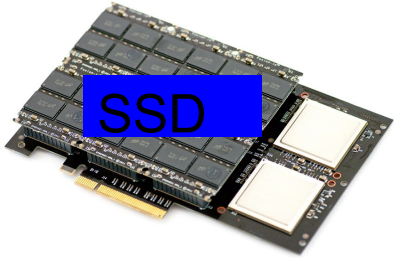
read(file,img)

write(file,img)

send(resp)

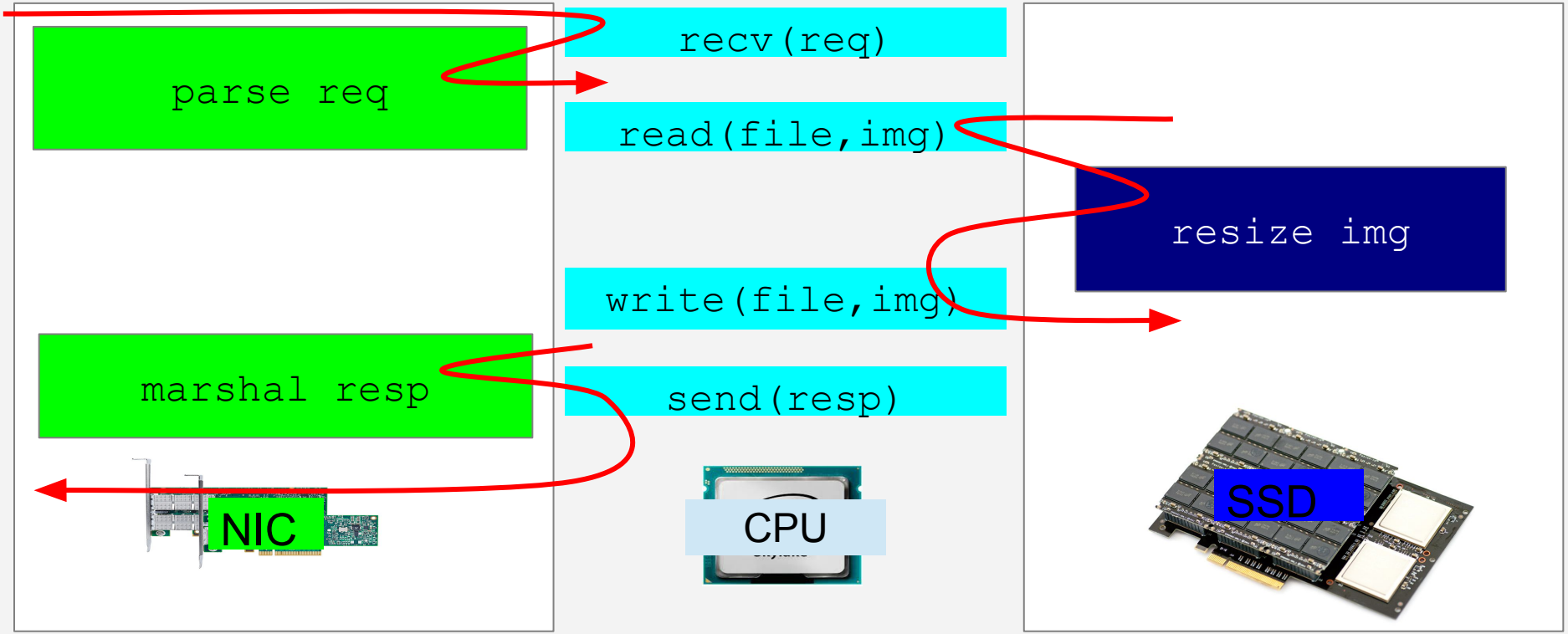


resize img



Offloading overheads dominate

get: parse → **resize** → **store** → serialize



Offloading overheads dominate

get: parse → **resize** → **store** → serialize

parse req

recv(req)

read(file,img)

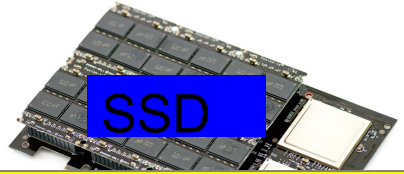
resize img

write(file,img)

marshal resp

send(resp)

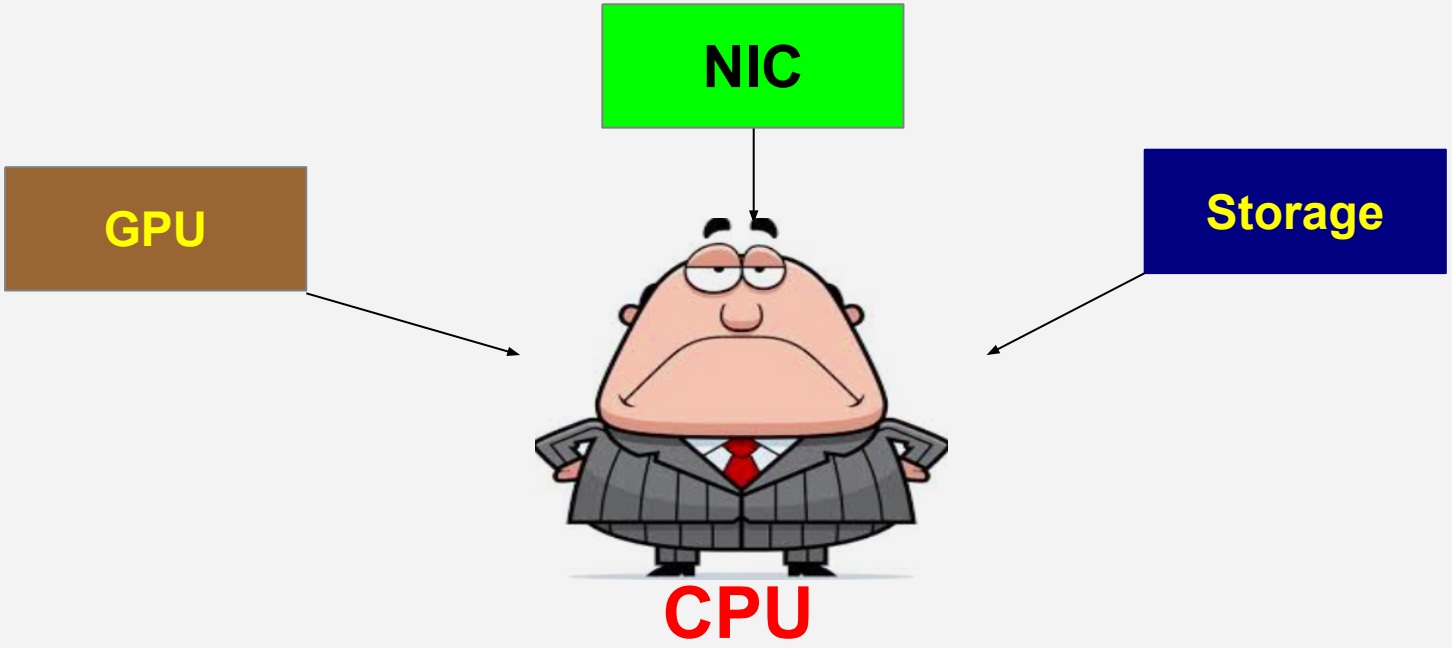
CPU



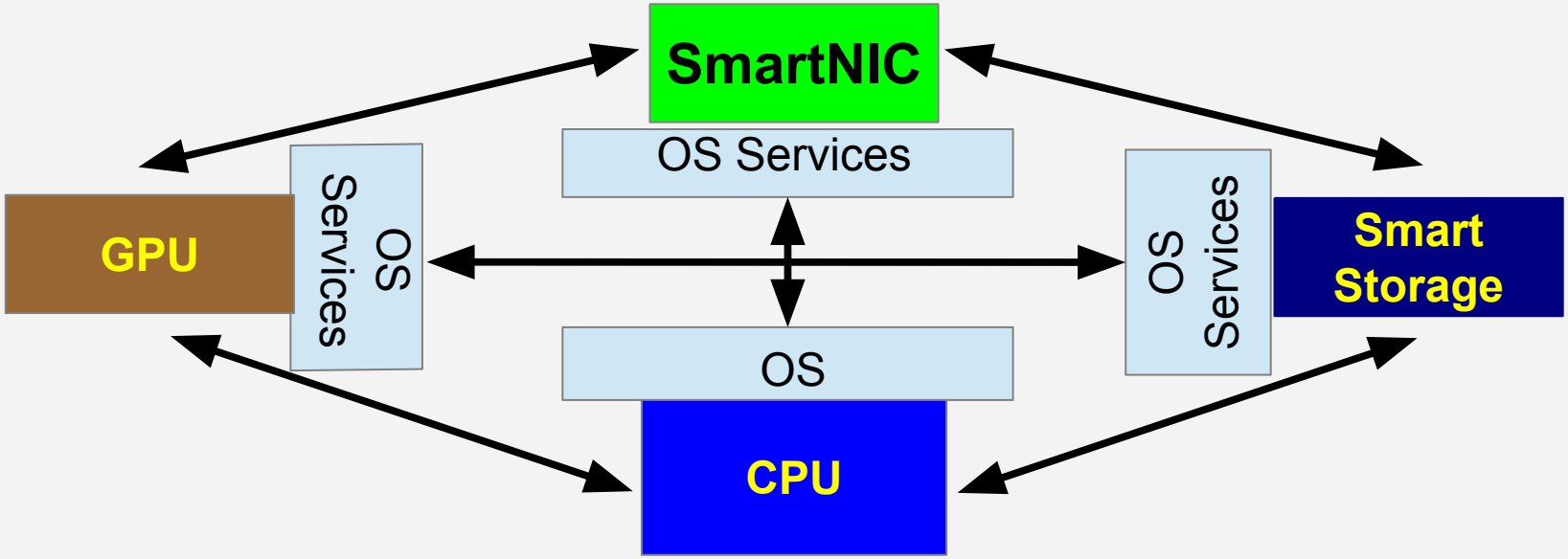
No sockets, isolation, transport layer …

No files, protection...

THE problem: OS architecture is CPU - centric



OmniX: Accelerator-centric OS architecture



This course is about accelerator-centric system design

- Important mechanisms of heterogeneous systems
  - Basics of the co-processor programming model
  - Virtual memory
  - Intra-node networking (PCIe)
  - Memory models
- Accelerator-centric OS design
  - Interaction with I/O accelerators
  - Networking and storage access from accelerators
- Future hardware, heterogeneous and disaggregated data centers

Additional reading

- Principles of Computer System Design: An Introduction, JH Saltzer and MF Kaashoek
- On being the right size, JBS Haldane
- OmniX: an accelerator-centric OS for omni-programmable systems, HotOS'16, M. Silberstein

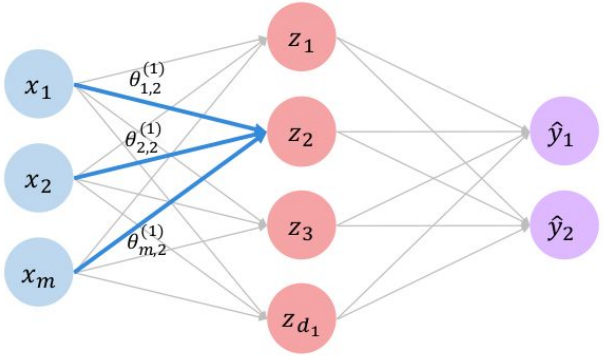
Basics of co-processor accelerator programming

General development flow with programmable accelerators

- Code development in higher-level language using hardware-specific primitives
- CPU-side “driver” development using accelerator runtime API
- Cross-compilation + packaging

Example: GPUs

GPUs in ML – Linear Algebra Accelerators



$$z_2 = \theta^{(1)}_{0,2} + \sum_{j=1}^{m} x_j\, \theta^{(1)}_{j,2}$$
$$= \theta^{(1)}_{0,2} + x_1\, \theta^{(1)}_{1,2} + x_2\, \theta^{(1)}_{2,2} + x_m\, \theta^{(1)}_{m,2}$$

Compute Matrix Product **ON GPU**

How to program a GPU to compute this?

Simple GPU program: exploiting data parallelism

- Idea: same set of operations is applied to different data chunks *in parallel*
- Algorithmic challenge – identify data-parallel tasks

Simple GPU program: exploiting data parallelism

- Idea: same set of operations is applied to different data chunks *in parallel*
- Algorithmic challenge – identify data-parallel tasks
- Implementation
  - Every *thread* runs the same code on different data chunks.
  - GPU concurrently runs thousands of parallel threads

Vector sum C=A+B

- Sequential algorithm

```
For every i
   C[i]=A[i]+B[i]
```

Vector sum C=A+B

- Sequential algorithm

```
For every i
   C[i]=A[i]+B[i]
```

- Parallel algorithm

```
In parallel For every i
   C[i]=A[i]+B[i]
```

Implementation for a vector of length 1024

- GPU **kernel** (this program runs in every thread)

```
C[threadId]=A[threadId]+B[threadId]
```

Per-thread hardware-supplied ID

Implementation for a vector of length 1024

- GPU kernel

```
C[threadId]=A[threadId]+B[threadId]
```

- CPU

1. Allocate arrays in GPU memory
2. Make data accessible to GPU: CPU → GPU copy
3. Invoke GPU kernel with 1024 threads
4. Wait until complete and copy data GPU → CPU

Complete example (sketch)

CPU:

```
void vector_sum(float* A, float* B, float* C, size_t n)
{
     float* gA=cudaMalloc(n); cudaMemcpy(gA,A,CPU_GPU);
     float* gB=cudaMalloc(n); cudaMemcpy(gB,B, CPU_GPU);
     float* gC=cudaMalloc(n);

    vector_sum_kernel<<<n>>>(gA,gB,gC);
    cudaWait();

     cudaMemcpy(C,gC,GPU_CPU);
}
```

GPU:

```
void vector_sum_kernel(float* gA, float* gB, float*gC)
{
      size_t me=cudaGetID();
      gC[me]=gA[me]+gB[me];
}
```

Complete example (sketch)

CPU:

```
void vector_sum(float* A, float* B, float* C, size_t n)
{
     float* gA=cudaMalloc(n); cudaMemcpy(gA,A,CPU_GPU);
     float* gB=cudaMalloc(n); cudaMemcpy(gB,B, CPU_GPU);
     float* gC=cudaMalloc(n);

    vector_sum_kernel<<<n>>>(gA,gB,gC);
    cudaWait();

     cudaMemcpy(C,gC,GPU_CPU);
}
```

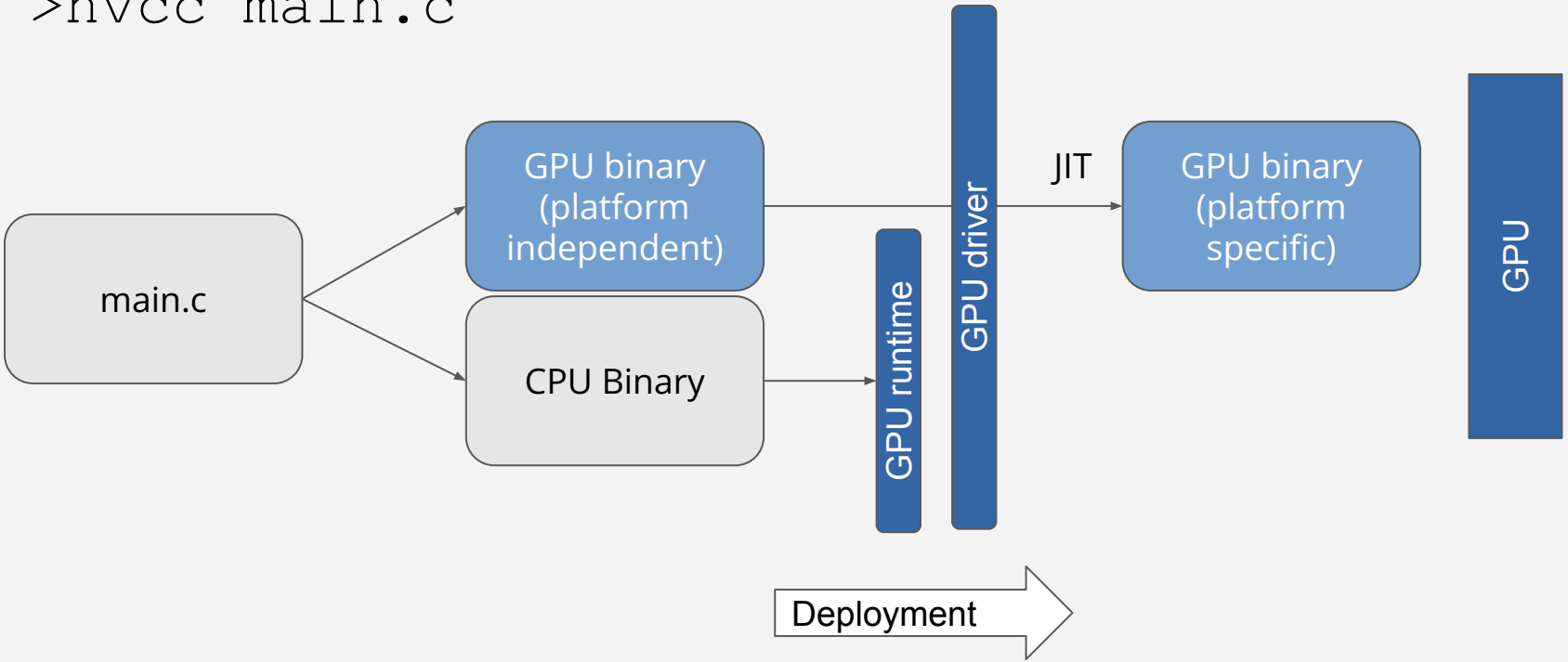
Many hardware-managed lightweight threads

GPU:

```
void vector_sum_kernel(float* gA, float* gB, float*gC)
{
      size_t me=cudaGetID();
      gC[me]=gA[me]+gB[me];
}
```

Compilation and deployment

```
>nvcc main.c
```



Complete example (sketch)

CPU:

```
void vector_sum(float* A, float* B, float* C, size_t n)
{
     float* gA=cudaMalloc(n); cudaMemcpy(gA,A,CPU_GPU);
     float* gB=cudaMalloc(n); cudaMemcpy(gB,B, CPU_GPU);
     float* gC=cudaMalloc(n);

    vector_sum_kernel<<<n>>>(gA,gB,gC);
    cudaWait();

     cudaMemcpy(C,gC,GPU_CPU);
}
```

GPU integration

GPU:

```
void vector_sum_kernel(float* gA, float* gB, float*gC)
{
      size_t me=cudaGetID();
      gC[me]=gA[me]+gB[me];
}
```

GPU programming

Complete example (sketch)

CPU:

```
void vector_sum(float* A, float* B, float* C, size_t n)
{
     float* gA=cudaMalloc(n); cudaMemcpy(gA,A,CPU_GPU);
     float* gB=cudaMalloc(n); cudaMemcpy(gB,B, CPU_GPU);
     float* gC=cudaMalloc(n);

    vector_sum_kernel<<<n>>>(gA,gB,gC);
    cudaWait();

     cudaMemcpy(C,gC,GPU_CPU);
}
```

This is often why things don't work

GPU:

```
void vector_sum_kernel(float* gA, float* gB, flo
{
     size_t me=cudaGetID();
     gC[me]=gA[me]+gB[me];
}
```

That's the fun stuff

Observations

- No operating system on GPU
- CPU manages everything via the GPU driver
  - Memory allocation, code invocation
- CPU program to manage the accelerator might get quite complex

Additional reading

- CUDA Docs (docs.nvidia.com/cuda)
- GPU Programming Course (former Udacity “Intro to Parallel Programming) https://classroom.udacity.com/courses/cs344

Questions for self-study [2]

1. Why the current OS structure precludes efficient use of accelerators
2. Is the following statement correct: “GPUs cannot run an OS because they do not have privilege separation”
3. What is the key algorithmic challenge when programming a GPU
4. Why do GPUs employ Just-In-Time (JIT) compilation when deploying a kernel on a GPU
5. What are the typical tasks done by the CPU when a program is using a GPU

Virtual memory in accelerators

Agenda

- Virtual memory 101: replay from the OS course
- Multiple address spaces
- Globally shared virtual memory
- Hardware page faults and page migration

Virtual Memory 101

- VM abstraction
- Pages, page tables, page cache, swapping, page faults, demand paging
- Page walks, MMU, TLB

What would happen here?

```
__global__ void foo(int* ptr){
    ptr++;
    *ptr=1;
}
```

Runs on the accelerator (GPU)

```
__host__ void run_foo(){
    int h_ptr=malloc(4K);
    int g_ptr=cudaMalloc(4K); // GPU memory

    h_ptr++; *h_ptr=0;
    g_ptr++; *g_ptr=0;

   foo<<<>>>(h_ptr); //invoke GPU kernel
   foo<<<>>>(g_ptr); //invoke GPU kernel
}
```

Runs on the host (CPU)

GPU and CPU have separate address spaces

```
__global__ void foo(int* ptr){
    ptr++;
    *ptr=1;
}
```

crash

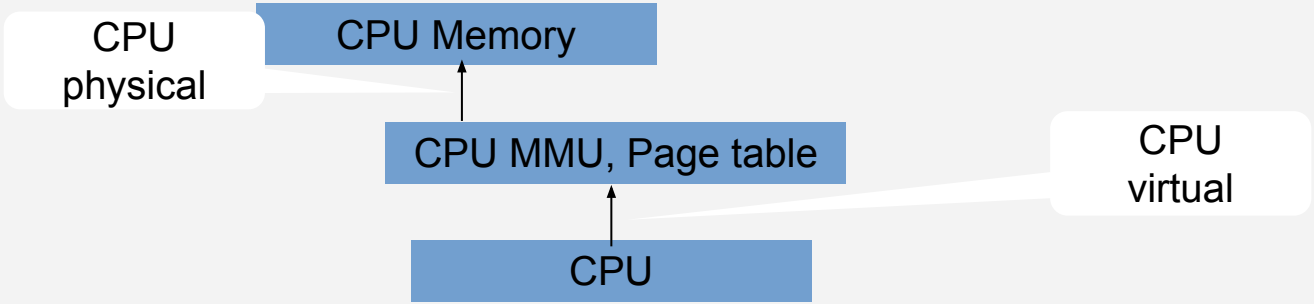
Runs on the accelerator (GPU)

```
__host__ void run_foo(){
    int h_ptr=malloc(4K);
    int g_ptr=cudaMalloc(4K); // GPU memory

    h_ptr++; *h_ptr=0;
    g_ptr++; *g_ptr=0;

   foo<<<>>>(h_ptr); //invoke GPU kernel
   foo<<<>>>(g_ptr); //invoke GPU kernel
}
```

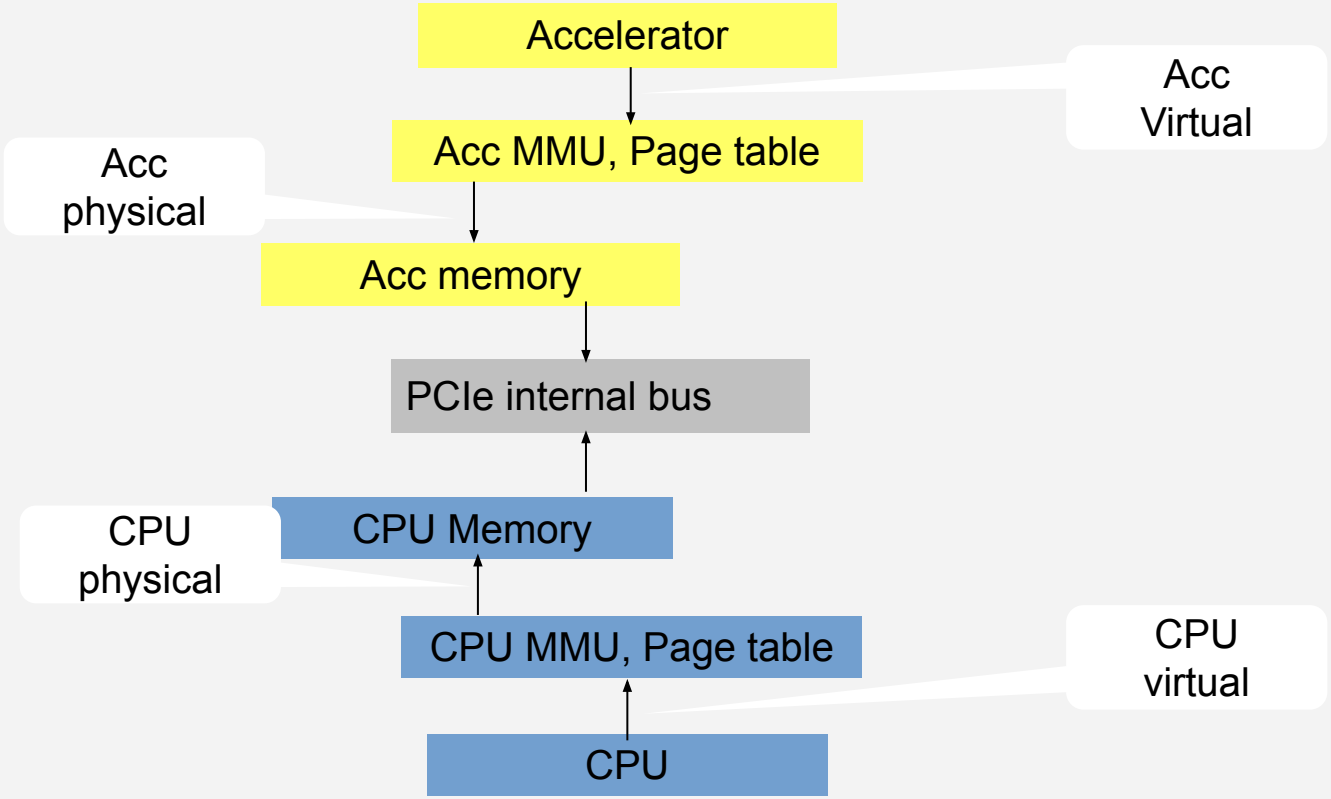
crash

Runs on the host (CPU)

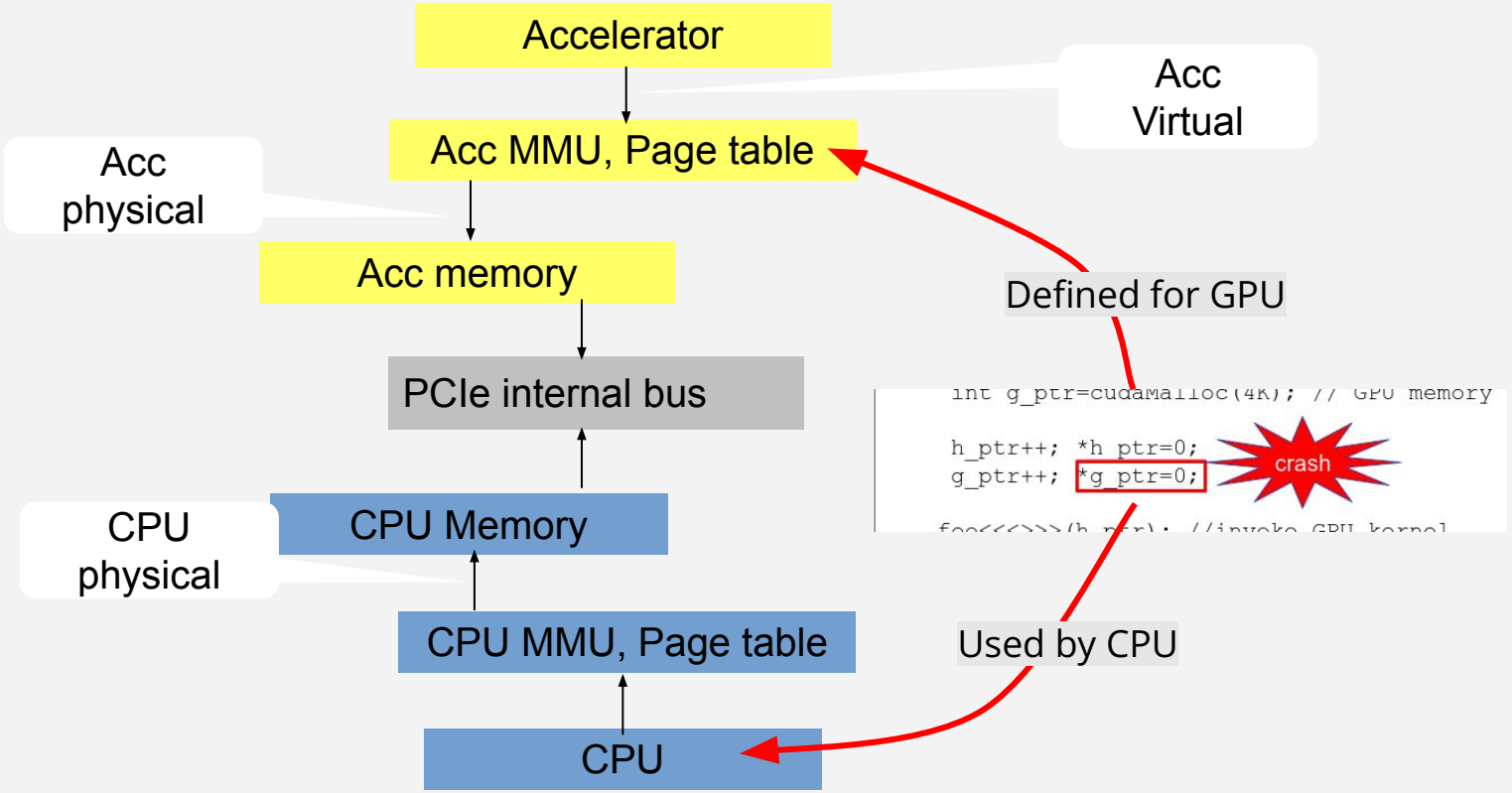
Traditional virtual memory



Inside accelerator virtual memory



Inside accelerator virtual memory



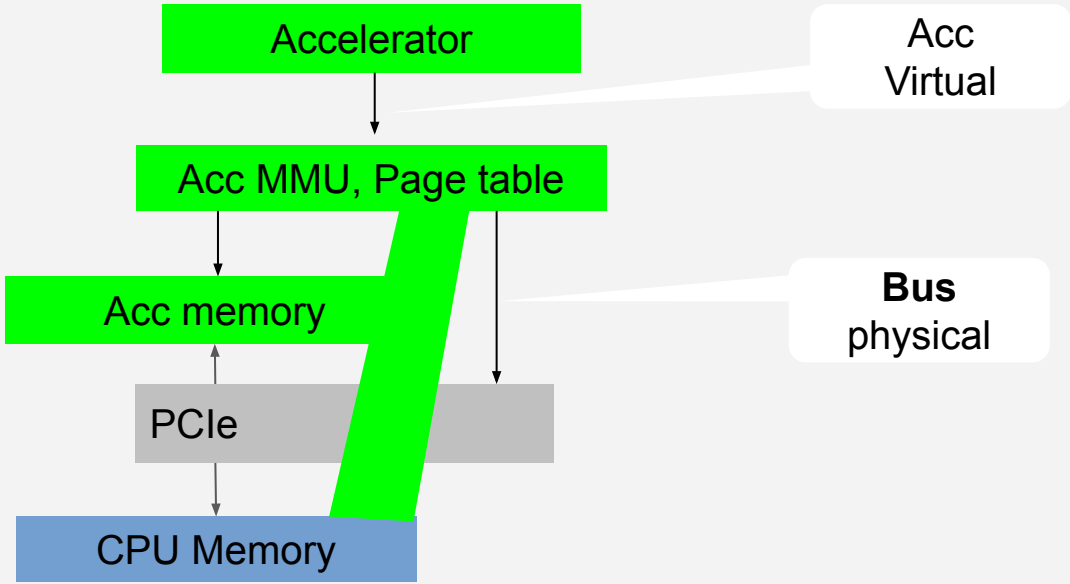
VM in GPUs

- GPU address space managed by CPU driver
  - cudaMemcpy/cudaMalloc operate in GPU address space
- CPU allocates physical memory, updates GPU page table

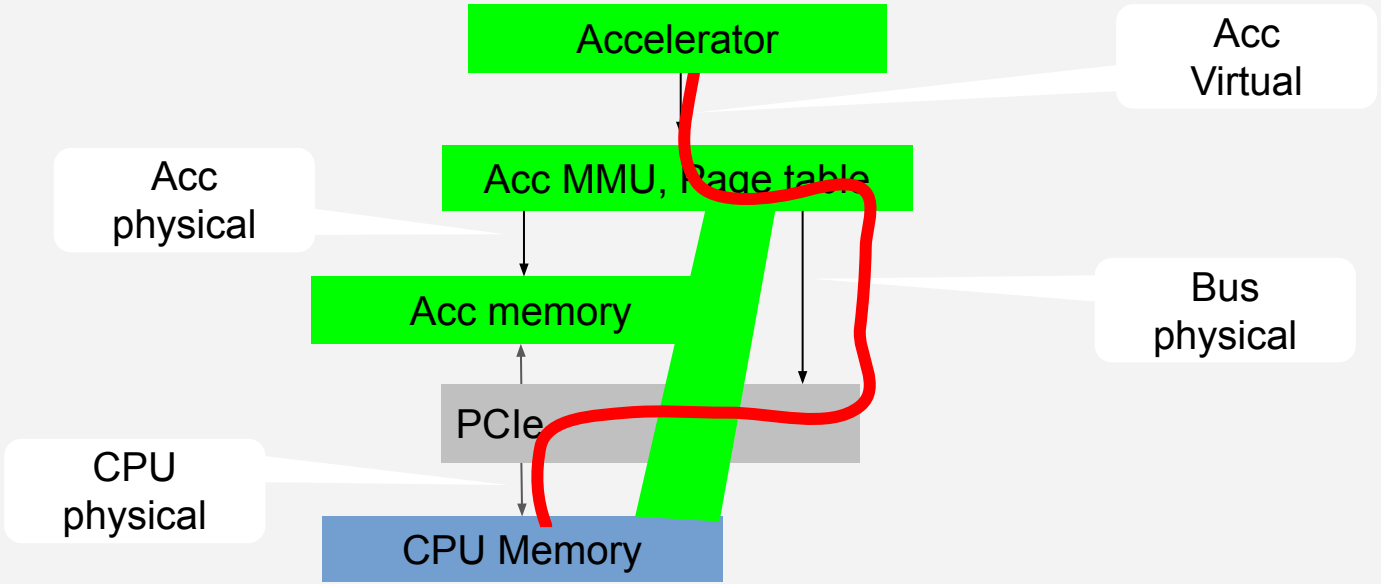
No virtual address space sharing between system processors by default

Can accelerators access CPU memory? (*zero-copy*)

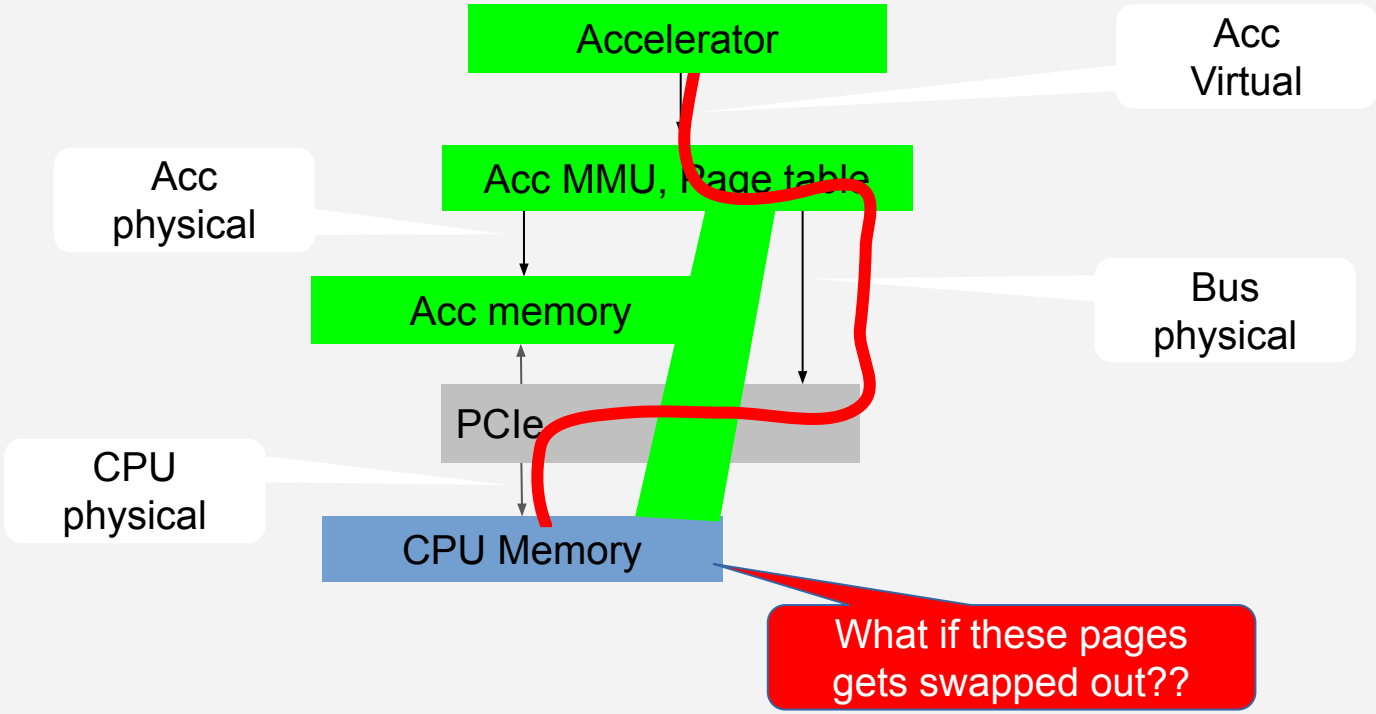
Mapping CPU memory into Acc Virtual address space



Acc access to CPU memory



Acc access to CPU memory



GPU access to CPU memory

- `cudaHostAlloc(MAPPED)`
  - allocates CPU memory to make it accessible to the GPU, and **pins** it in CPU virtual address space
- `cudaGetDevicePointer()`
  - maps CPU memory into GPU address space
  - retrieves the **GPU address** for CPU pointer allocated via cudaHostAlloc

CPU access to GPU memory

- Similar techniques
- In NVIDIA parlance called “GPUDirectRDMA”
- Exposes GPU memory on PCIe (more on that soon)

Multiple address spaces are annoying

- Wouldn't it be great:

```
__global__ void foo(int* ptr){
    ptr++;
    *ptr=1;
}
```

```
__host__ void run_foo(){
    int* u_ptr=unified_malloc(4K);
    u_ptr++; *u_ptr=0;
    foo<<<>>>(u_ptr);
}
```

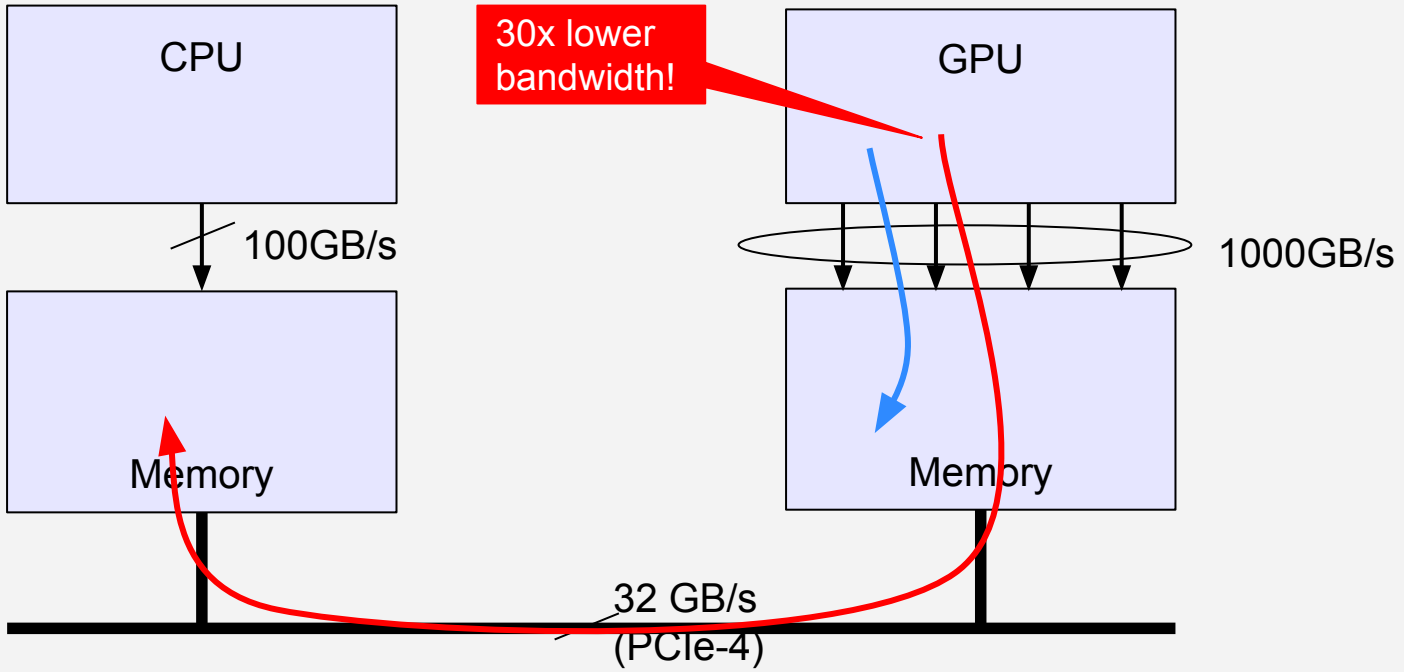
We want shared virtual memory!

Shared address space vs. shared memory

- Shared memory (usually SW): access **the same application data via** potentially **different pointers**
  - Two processes may share memory but map it with different virtual addresses
- Shared address space (usually HW): same (unique) pointer on different processors refers to the same memory

Shared virtual memory =
Shared memory + Shared address space

Problem: Remote access is slow!



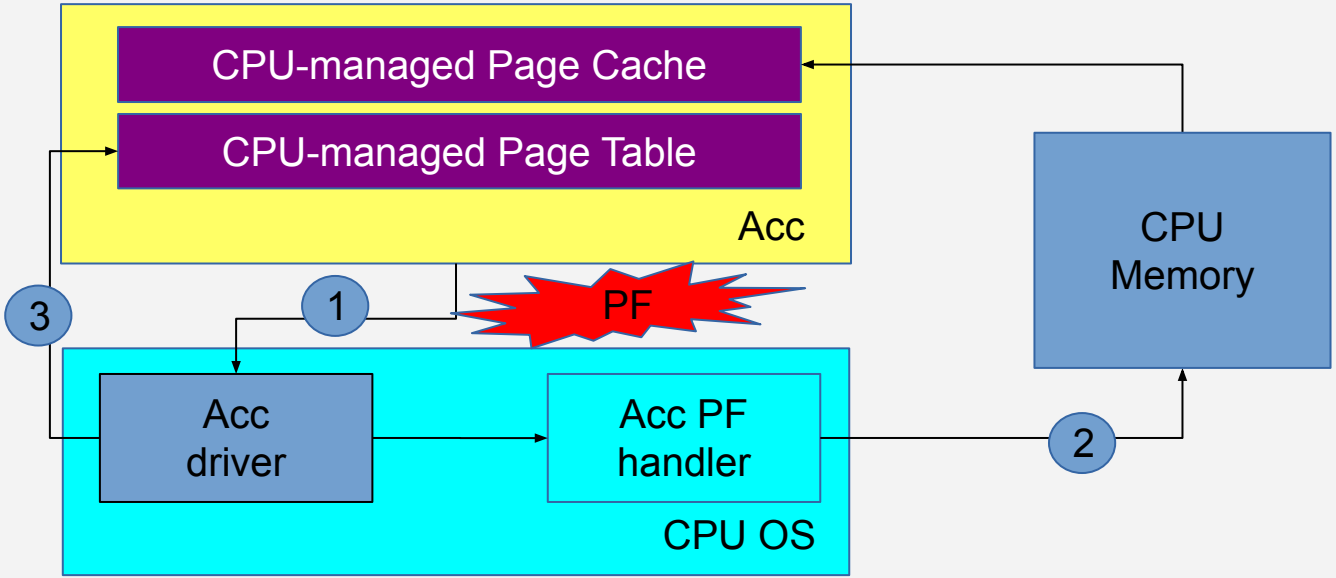
Heterogeneous Shared Memory

- Found in modern NVIDIA discrete GPUs
- Implements an *abstraction* of shared memory
- Optimizes data placement under the hood
- Key idea: move data where it is used
  - Each processor keeps the used data in its memory
  - Migrates on-demand

How to identify data to be used ?

Accelerator page faults

- Found in modern NVIDIA discrete GPUs



Access to shared pages between Acc and CPU

A physical page is mapped on CPU **or** on Acc but never together

- Migrate a page to Acc mem upon Acc page fault
- Migrate a page to CPU mem upon CPU page fault
- Migration granularity: max (CPU page, Acc page)
  - GPU pages are 64KB, so migration is at 64KB granularity

Shared Virtual Memory with GPU PF

```
__host__ void run_foo(){
  int* u_ptr=unified_malloc(4K);

  u_ptr++; *u_ptr=0;

  foo<<<>>>(u_ptr);
```

Allocate virtual space in CPU and GPUs

CPU PF: allocate/map CPU physical page

```
__global__ void foo(int* ptr){
  ptr++;
  *ptr=1;
}
```

GPU PF: unmap page from CPU, allocate physical memory on GPU, migrate page into GPU, map in GPU

What is the consistency model?

E.g., does CPU sees updates if GPU writes to a page

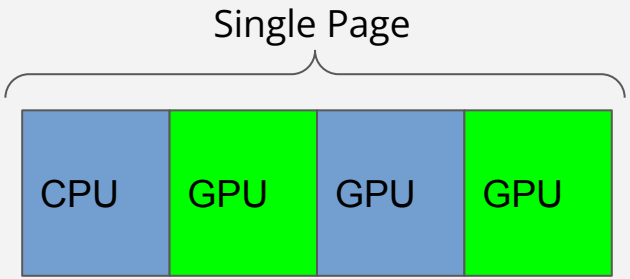
We will discuss memory models in detail later

- Strong consistency: ***single owner***
- Classical Lamport's model

Problems?

False sharing

- Acc and CPU **write**-share the same page, but not the data in the page, so there is no actual data race



- As a result:
  - Page bouncing: a page gets constantly migrated b/w GPU and CPU
  - Extremely slow

Summary

- Acc VM similar/identical to CPU VM
- Acc VM is (usually) managed by CPU
- Slow remote access calls for Virtual Shared Memory abstraction, which might get costly
- False sharing is a problem due to coarse-granular migration

Additional reading

- GAIA: An OS Page Cache for Heterogeneous Systems, USENIX ATC'19, T. Brokhman et al.
- An asymmetric distributed shared memory model for heterogeneous parallel systems, ASPLOS'10, I. Gelado et al.
- Heterogeneous Memory Management (LINUX Kernel Docs)
- ActivePointers: A case for software address translation on GPUs, ISCA'16, S. Shachar et al
- Dragon: Breaking GPU memory capacity limits with direct NVM access, SC'18 P. Markthub

Questions for self-study [3]

1. Why does CPU buffers need to pinned if mapped into accelerator's memory address space
2. If an accelerator can access CPU memory, does it mean that the security of the system is compromised because CPU memory protection mechanisms are not enforced on the accelerator?
3. In a system that does not implement shared virtual memory, a GPU passes a GPU pointer to a CPU. Is a page fault guaranteed to occur on the CPU access to it?
4. Describe a scenario in which strict consistency performance suffers the worst.

Introduction to memory models

What do we expect to be printed here?

A program running in 2 threads
(all shared variables initialized to 0)

Thread 1

```
Data=1;
Flag=1;
```

Thread 2

```
while (Flag!=1);
print Data;
```

What do we expect to be printed here?

A program running in 2 threads
(all shared variables initialized to 0)

Thread 1

```
Data=1;
Flag=1;
```

Thread 2

```
while (Flag!=1);
print Data;
```

Depends on the memory model!
E.g., both 0 and 1 would be valid on a GPU

Memory consistency model

- A contract between a user and a platform
- Defines **permitted reorderings** of memory operations when accessing shared memory from multiple threads

What does it mean “being reordered”

Thread 1

```
Data=1;
Flag=1;
```

Thread 2

```
while (Flag!=1);
print Data;
```

Axiom: *program order* (single thread) is never violated!

T1: W (Data) → W(Flag)

T2: R(Flag) → R(Data)

Question: if Flag is 1, does it imply that Data is 1?

What does it mean “being reordered”

Thread 1

```
Data=1;
Flag=1;
```

Thread 2

```
while (Flag!=1);
print Data;
```

Axiom: *program order* (single thread) is never violated!

T1: W (Data) → W(Flag)

T2: R(Flag) → R(Data)

Question: if Flag is 1, does it imply that Data is 1?

Only if W (Data) → W(Flag) **is enforced globally**

So how can we explain output 0?

Thread 1

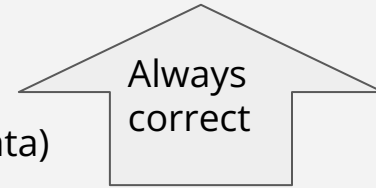
```
Data=1;
Flag=1;
```

Thread 2

```
while (Flag!=1);
print Data;
```

T1: W (Data) → W(Flag)

T2: R(Flag) → R(Data)



So, if Flag is 1 and Data is 0, what was reordered?

For Thread 2 it looks as if W(Flag) → W(Data)
(i.e., these operations were reordered in other threads w.r.t. to program order).

Sequential (strong) consistency: no permitted reorderings

Lamport, 79:

A multiprocessor is **sequentially consistent** if the result of any execution is the same as if the operations of **all the processors were executed in some sequential order**, and the operations of each individual processor appear in this sequence **in the order specified by its program**

```
Data=1;
Flag=1;
```

```
while (Flag!=1);
print Data;
```

Only 1 is permitted as output

Why not sequential consistency

- Limits hardware/software optimizations
- Poor performance

Consider how to ensure that a “store” is visible to all threads?

But more importantly, not always needed!

When the order is not important

Thread 1

```
Data=1
MoreData=2;
Flag=1;
```

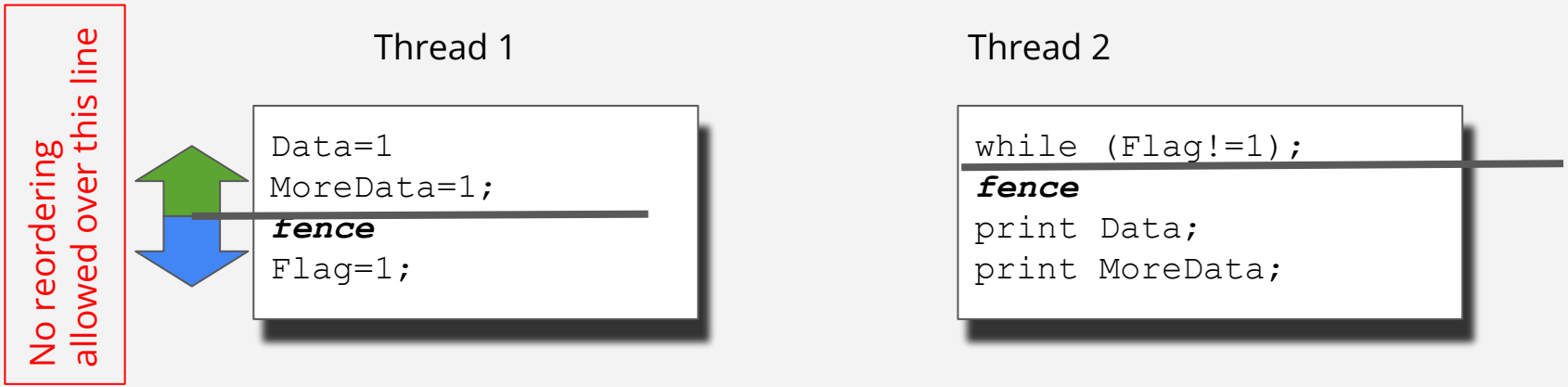
Thread 2

```
while (Flag!=1);
print Data;
print MoreData;
```

Does it matter if Data propagates before MoreData?
No! as long as both propagate before Flag.

Weak consistency

- Hardware allows all permutations across threads
- Puts the burden on ***a programmer***
- Software adds `fence` where *necessary*



Weak consistency

- Hardware allows all permutations across threads
- Puts the burden on ***a programmer***
- Software adds `fence` where *necessary*

Thread 1

```
Data=1
MoreData=1;
fence
Flag=1;
```

Thread 2

```
while (Flag!=1);
fence
print Data;
print MoreData;
```

Why is this fence necessary?

Weak consistency

- Hardware allows all permutations across threads
- Puts the burden on ***a programmer***
- Software adds `fence` where *necessary*

Thread 1

```
Data=1
MoreData=1;
fence
Flag=1;
```

No `fence` needed here

Thread 2

```
while (Flag!=1);
fence
print Data;
print MoreData;
```

Memory models and accelerators

- Architecture-dependent, usually *weak*

- May have different models for inter-accelerator and intra-accelerator (i.e, access to CPU memory)

CUDA GPU memory models

- CUDA supports *C++11 memory model* with small changes
- We will briefly describe C++11 memory model first

C++11 memory model: memory primitives

- Regular variables
- Atomic variables: **std::atomic**<T> a;
  - Support atomic read-modify-write (RMW) operations: Compare And Swap, Exchange, Increment, Assignment,...
  - Usually hardware-supported only for basic types (int, char, float, double..)
- Memory fences: std::atomic_thread_fence();

C++11 memory model: rules

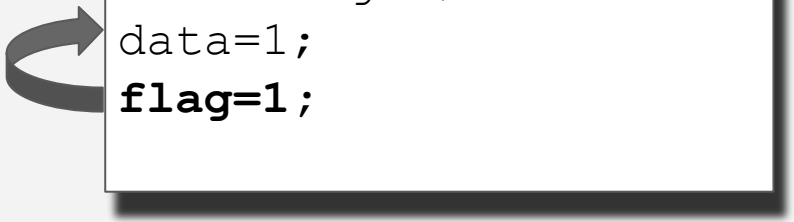
1. All regular variables are **weak**
   a. **No ordering guarantees**

2. Atomic variables can be configured **strong**
   a. Ordered among themselves (by default)
   b. Support atomic RMW operations
   c. Used to enforce order on weak variables

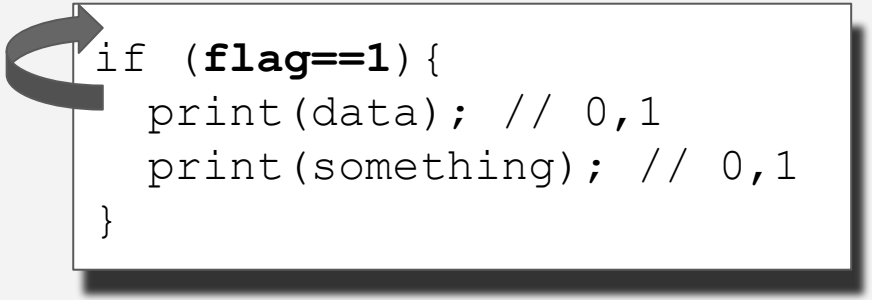
Atomic variables must be used for ordering weak variables

The wild west of weak variables

```
int flag=0, data=0, something=0;
```



```
something=1;
data=1;
flag=1;
```



```
if (flag==1){
  print(data); // 0,1
  print(something); // 0,1
}
```

Atomic variables: seq consistency by default


```
atomic<int> flag, data, moredata;
```

```
data.store(1);
moredata.store(2);
flag.store(1);
```

```
if(flag.load()==1){
  printf(data.load());
  printf(moredata.load());
}
```

Will always print “1” → “2”

Weak var synchronization via atomic variables:

Producer-consumer message passing

For a **given atomic** variable ***SYNC*** **if T2 LOADS the value of *SYNC STORED* by T1, then**

All **weak** loads/stores prior to SYNC.STORE in T1 **are complete** before ***SYNC.STORE***

and

All **weak** loads/stores following SYNC.LOAD in T2 **are performed** after **SYNC.LOAD**

T1

```
data=1;moredata=1; // weak vars
flag.store(1);
```

T2

```
if (flag.load()==1)
  print (data);
  print(moredata);
```

Atomic variables: relaxing default ordering

```
atomic<int> flag, data, moredata;
```

```
data.store(1,memory_order_relaxed);
moredata.store(2,memory_order_relaxed);
```

The use of strong variables should be minimized to avoid unnecessary performance overheads

Implementing a TAS lock?

```
struct mutex{
 _lock=0;
 void  lock(){
    while(TAS(&_lock,1));
 }
  void unlock(){
    _lock=0;
 }
}
```

Reminder:

```
TAS(var,value){
     old=var;
     var=value;
     return old;
}
```

Atomically

This is what we all learned in the OS class

Possible implementation

```
struct mutex{
 atomic<int> _lock(0);
 void  lock(){
    while(_lock.exchange(1)); //TAS
 }
  void unlock(){
    _lock.store(0);
 }
}
```

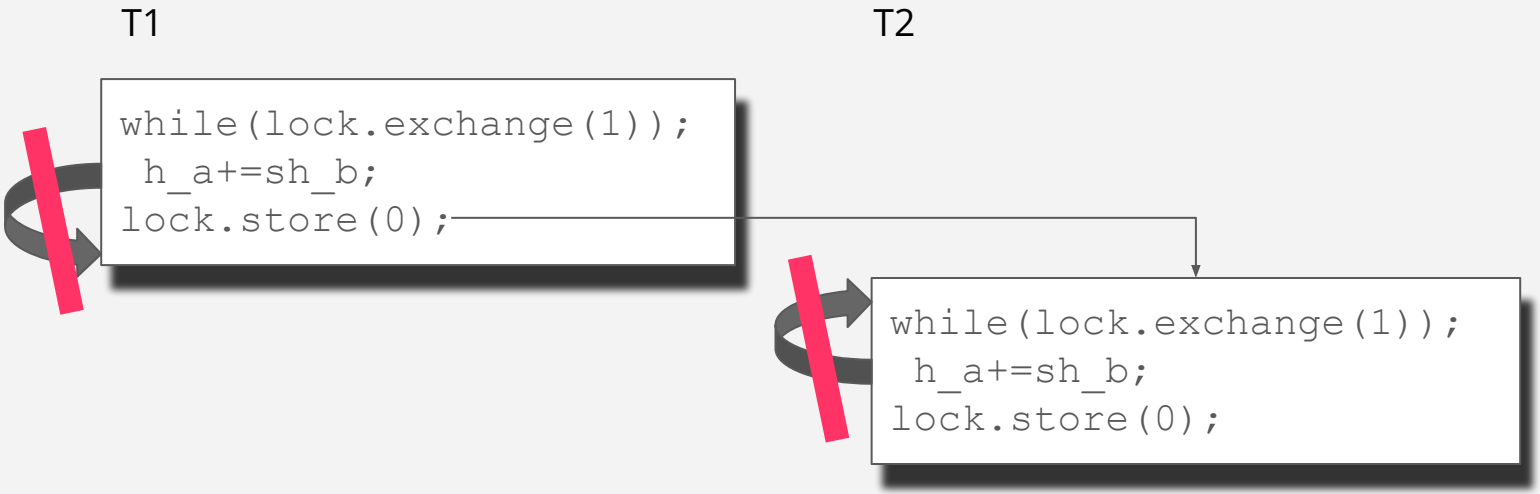
Possible implementation

```
struct mutex{
 atomic<int> _lock(0);
 void  lock(){
    while(_lock.exchange(1));
 }
  void unlock(){
    _lock.store(0);
 }
}
```

Does it guarantee mutual exclusion?

```
mutex.lock()
 h_a+=sh_b;
mutex.unlock();
```

This lock guarantees mutual exclusion



CUDA memory model: C++11 and extensions

```
#include <cuda/atomic>
```

This is the portable way that works on the host and on the device

CUDA introduces scopes to limit the consistency scope

Examples:

- **device**: applies to threads in the same GPU
- **system**: applies to all threads including *other processors* (CPU or GPU)

Scope is a ***type*** of a variable: atomic<int,**thread_scope_block**> var;

Consistency scope

```
atomic<int, memory_scope_system> data_sys, flag_sys;

atomic<int,memory_scope_device> data, flag;
```

Not in the same scope – behave like weak

GPU1

```
data.store(2);
flag.store(1);
```

GPU2

```
if (flag.load())
   print(data.load());
```

In the same scope – behave like atomics

CPU

```
data_sys.store(2);
flag_sys.store(1);
```

GPU1

```
if (flag_sys.load())
   print(data_sys.load());
```

Synchronization over PCIe

- PCIe-3 supports producer-consumer pattern
- PCIe-3 **does not support atomic operations**

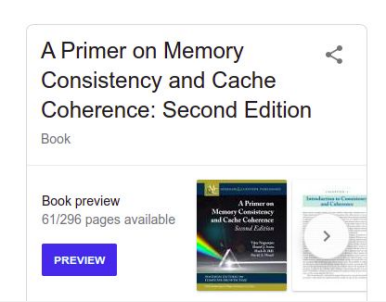
  CPU and GPU updates over PCIe to the same atomic variable **are not atomic**

- These limitations will be removed in CXL/PCIe-5 (most likely)

Summary

- Understanding memory models is essential
  - For building synchronization between devices
  - For ensuring correct software
  - For achieving performance
- CUDA Memory model design and implementation has been a multi-year academic+industrial effort to achieve compliance with C++11
- We touched only a tip of an iceberg

Additional readings



One of the best books I've read on the topic!

- Olivier Giroux on Youtube: "The one decade task: putting std::atomic in CUDA"
- C++11 memory model tutorial(s) on internet. There are many, e.g. https://www.modernescpp.com
- CUDA Documentation
- Watch for tutorials by Daniel Lustig (recently on RISC-V memory models)

Questions for self-study [4]

1. What is the main downside of stronger memory models?
2. Does programmer-added `fence` always introduce performance overheads?
3. Does memory model apply to hardware or also to a compiler?
4. Why does CUDA have scopes on atomic variables?
5. Is it possible to see output “11” on a sequentially-consistent system (all vars are shared and initialized to 0) when running the following:

```
T1: print b; if(t2lock!=1) { a=1; }
T2: t2lock=1; b=1; print a;
```

PCI Express Basics

Perpiheral Component Interconnect express (PCIe)

Used to connect peripherals among themselves and to the main CPU

PCIe = open standard

- By PCISIG (PCI Special Interest Group)
- ISA, PCI, PCI-X, then PCI-Express (PCIe). We'll talk about PCIe only.

Terminology

- **Link**: A path between two devices.
- **Lane**: A send-receive pair within a link
- **Link Width**: #Lanes in a link
  - x1, x2, x4, x8, x16, x32
- 4 generations so far. 5th coming soon?
  - Called “Gen x” (e.g. PCIe 2.0 = PCIe Gen 2)
- Gen 3 is currently most common

PCIe Bandwidth

Link Width	x1	x2	x4	x8	x12	x16	x32
Gen1 Bandwidth (GB /s)	0.5	1	2	4	6	8	16
Gen2 Bandwidth (GB/s)	1	2	4	8	12	16	32
Gen3 Bandwidth (GB/s)	2	4	8	16	24	32	64

(PCI Express Technology, Mike Jackson and Ravi Budruk)

PCIe: Just a network really

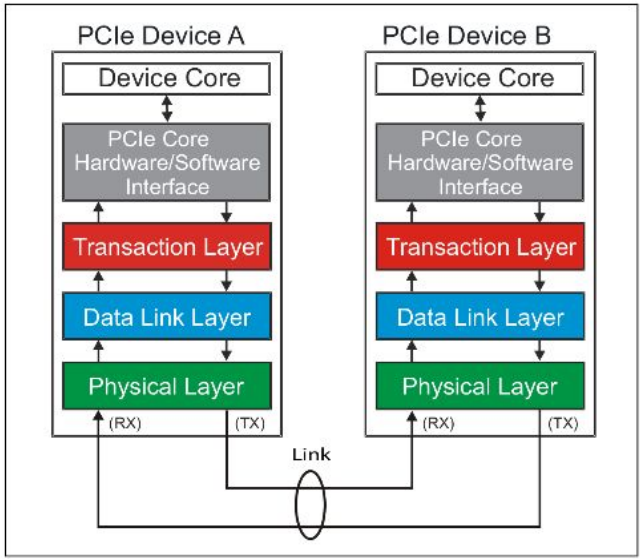
PCIe layer model

Networking model, similar to TCP/IP, OSI

Each layer:

- provides service to the layer on top
- Consumes services from the layer below
- Communicates with the corresponding layer on other nodes

Figure 2-12: PCI Express Device Layers



(PCI Express Technology 3.0, Mike Jackson and Ravi Budruk)

PCIe Layers

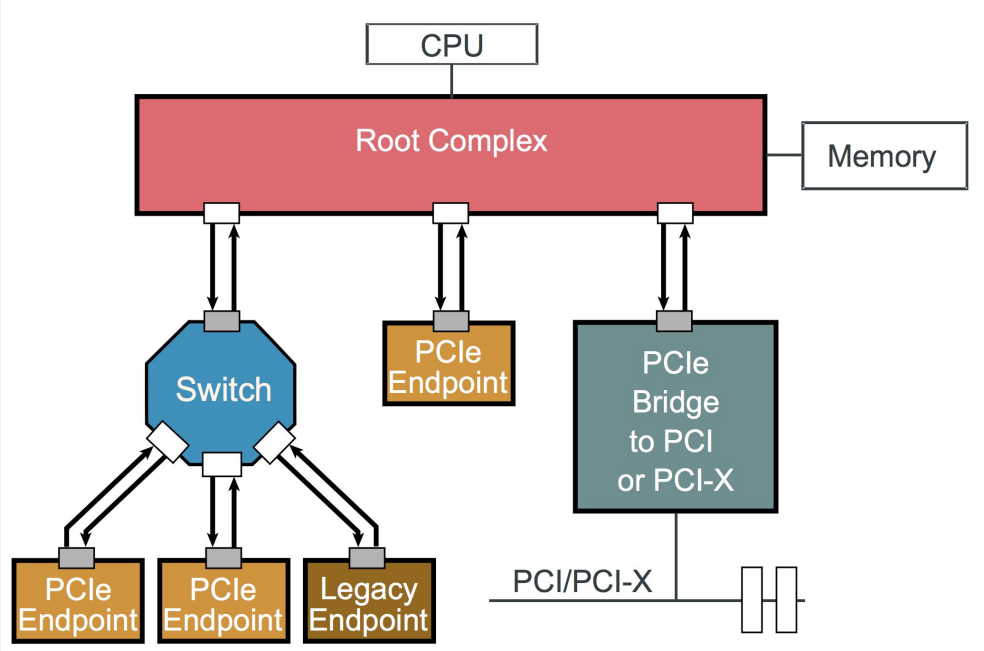
Transaction Layer	TLP, QoS, Flow Control, Ordering
Data Link Layer	DLLP, Error correction
Physical Layer	Byte encoding, Signaling

Topology

- A PCIe link is a **point-to-point** connection between 2 devices.
- Each device is connected to one PCIe link.
- How can we make a device talk to all other devices in the system?

Topology

upstream



downstream

(PCI Express Technology, Mike Jackson and Ravi Budruk)

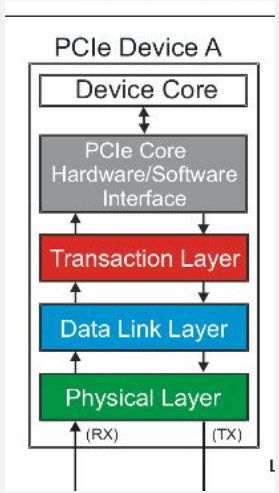
PCIe switches

- A switch is a PCIe component with multiple ports
- Allows building **a tree topology** where every device can talk to every device
- How does it know to which port to forward each packet? Later on this..

PCIe topology must be a tree

Transaction Layer: Application interface

What services do devices need?



- Device accesses host memory (DMA)
- Host accesses device memory/registers
- Interrupts – notify the host about device events
- Configuration

PCIe Transaction Types (TLPs)

- Read/Write (both DMA and MMIO)
- Messages (Interrupts) – also implemented as writes
- Configuration Read/Write
- Others (legacy, extensions, we'll ignore for now)

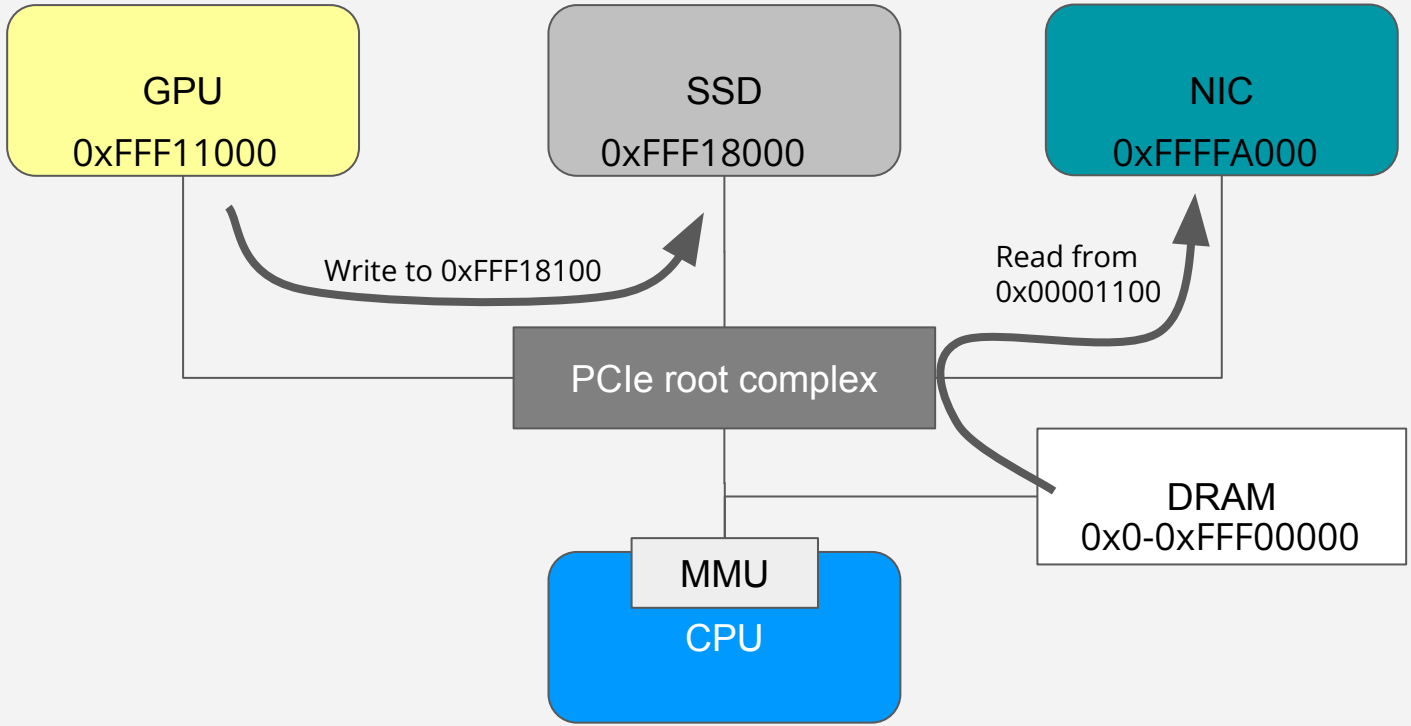
Posted and Non-Posted

- Non-Posted transaction: expects a response *(completion message)*
- Posted: Fire and forget
- Are read transactions posted?

TLP Routing

- Two methods to route transactions:
  - ID routing
    - Target a device with its firmware assigned ID
  - Address routing
    - Each device gets a system-wide unique range of the *physical memory* addresses *(bus addresses)*. This is a **global address space.**

Routing by bus address



BARs – Base Address Registers

- Different devices need different amounts of memory address range. Or even multiple such ranges
- A device tells what size of memory range it **needs via its BARs** (Base Address Registers).
- **Memory address range is reserved** by **the system** and **written to the BARs.**
- **PCIe switches are updated** to span the memory ranges of their devices.

Memory Mapped IO (MMIO)

- Informally, accessing a device address range is called “reading/writing to a BAR”
- BAR of a device can be read from / written to by other device
- A CPU accesses it by *mapping* it into its Virtual Memory

What is the content read/written from/to a device at a given address ?

Use cases

Meaning of reads/writes to the device's memory space is defined by the device.

- A write to disk drive BAR configures a certain encryption algo
- A write to a NIC's BAR at a certain address means: send packet.
- GPU exposes part of its *memory* on its BAR

Reads/writes to device addresses go to the device.
It decides what to do with them.

DMA

- **DMA** = Direct Memory Access
- Done by a DMA engine that is *part of a device* or a CPU. DMA engine must be programmed
  - Example: cudaMemcpy uses GPU's DMA engine to write to/read from CPU memory.
- The memory used by DMA must be pinned (the access is **direct** - bypasses the MMU of the remote device)
- DMA's remote memory can be PCIe BAR of a remote device

DMA vs MMIO

- DMA and MMIO are used **for the same purpose**
  - DMA engines perform the same PCIe reads/writes as MMIO
- However, MMIO is used via load/store instructions in *software*, while DMA is a *hardware* engine
- Tradeoff: DMA for large transfers, MMIO for small
  - DMA is slow to program, but does not waste CPU cycles on memory copies.
  - MMIO requires no programming, but wastes CPU cycles
- MMIO - mostly for control, DMA - mostly for data

Example MMIO: Peer-to-peer GPU0→ GPU1

1. GPU1 (target) exposes its global memory to MMIO via a BAR.

   That is: writing to (BAR + x) writes to global GPU memory with virtual address x. This requires GPU1 to make its PCIe interface handle respective BAR accesses

2. GPU0 maps the GPU1's BAR to its own virtual address space.
3. GPU0 writes to mapped GPU1's BAR.
4. PCIe switch knows how to route the writes to the GPU1's BAR.

DMA: Peer-to-peer GPU0→ GPU1

1. GPU1 (target) exposes its global memory to MMIO via a BAR.
2. CPU programs GPU0's DMA to write to GPU1's BAR,
3. PCIe switch knows how to route the writes to the GPU1's BAR.

Question: is there a solution to use DMA reads?

Transaction Ordering

PCIe used to access memory! Raises issues:

- How to order transactions correctly?
- How to isolate between transactions of differen machines?

Why these rules?

- Deterministic completion
  - Prevent deadlocks
- Preserve the programmer's intentions
  - Producer consumer example
- Maximize performance
  - Relax ordering to allow hardware optimizations

Producer/Consumer example

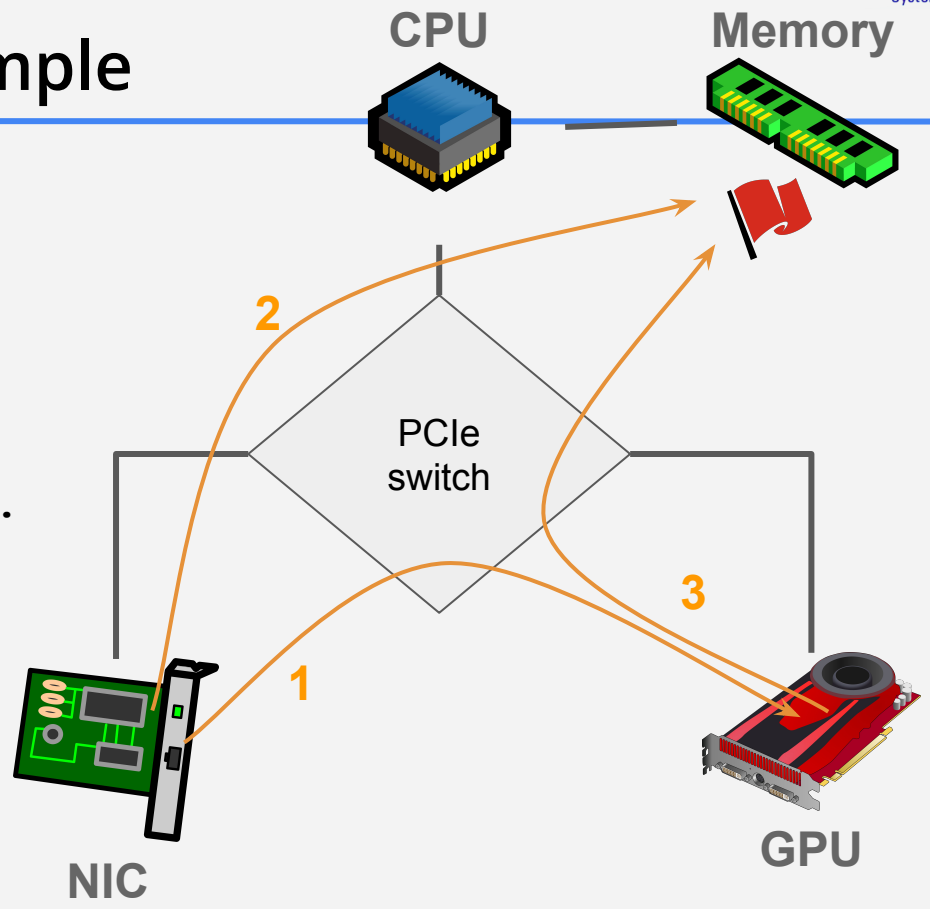
NIC:

1. `write data to GPU`
2. `write flag to memory`

GPU:

3. `do { read flag from mem.`
   `} while (!flag);`
   `access received data`

How to make sure GPU reads correct data?



PCIe ordering semantics

- We define local rules for **each link and each endpoint**
- Switches comply to the rules
- Global ordering behavior **follows** from the **tree structure** of the PCIe topology
  - Dependent transactions will flow via the same links
- The rules are complex, but the most important one:
  - writes are not reordered
  - reads after writes are not reordered, serve as *a barrier*
  - reads preceeding writes must allow reordering to avoid deadlocks

The rules guarateen that the scenario works correctly

Summary

- PCIe is a special-purpose network
- Built to allow device configuration, discovery, data transfer, etc
- Supports transactions (read/write)
- Supports routing according to a globally unique bus addresses
- Enables peer-to-peer architecture-independent inter-device communication
- Serves basis for future inter-device protocols (CXL)

Additional reading

PCI Express System Architecture, T Shanley, D Anderson, R Budruk

Understanding CXL:
https://www.snia.org/educational-library/understanding-compute-express-link-cache-coherent-interconnect-2020

Questions for self-study [5]

1. Why PCIe writes are faster than PCIe reads?
2. Why MMIO is considered less efficient for bulk transfers?
3. Two devices are connected via a PCIe switch, does their communication involve a CPU root complex?
4. Two devices perform DMAs to CPU memory. Do they intefere with each other?
5. A NIC wrote to a GPU BAR. Whose DMA engine was programmed to perform this transfer?
6. What might happen if a CPU reads a random address on the device's BAR?

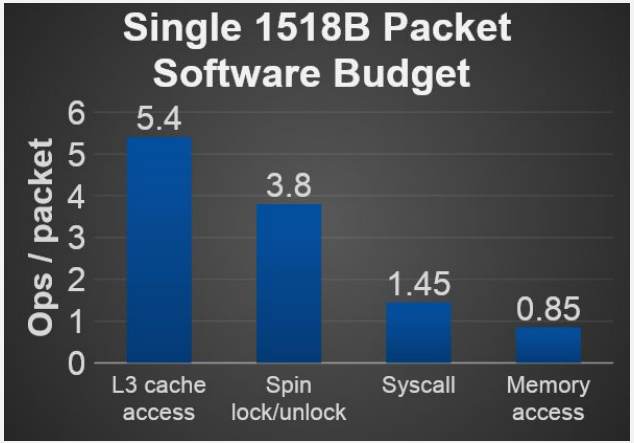
Networking accelerators and accelerator networking

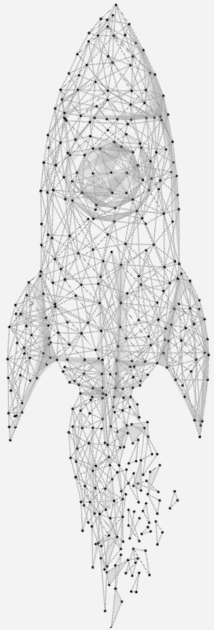
Packets Processing @200GbE is Challenging

Packet Processing Requires Accelerator

1518B Packet

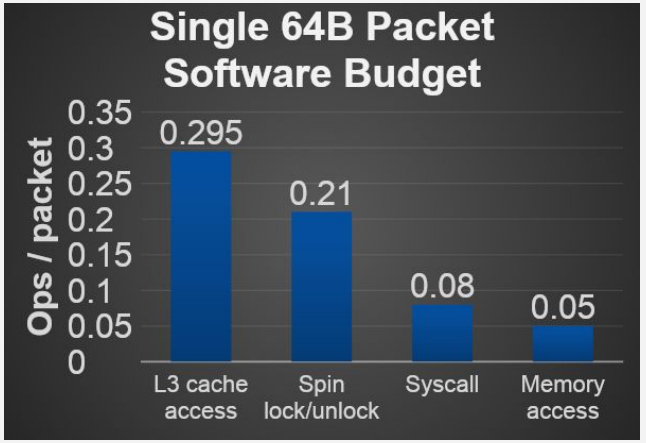
- 16M packets per second
- 62ns/packet





64B Packet

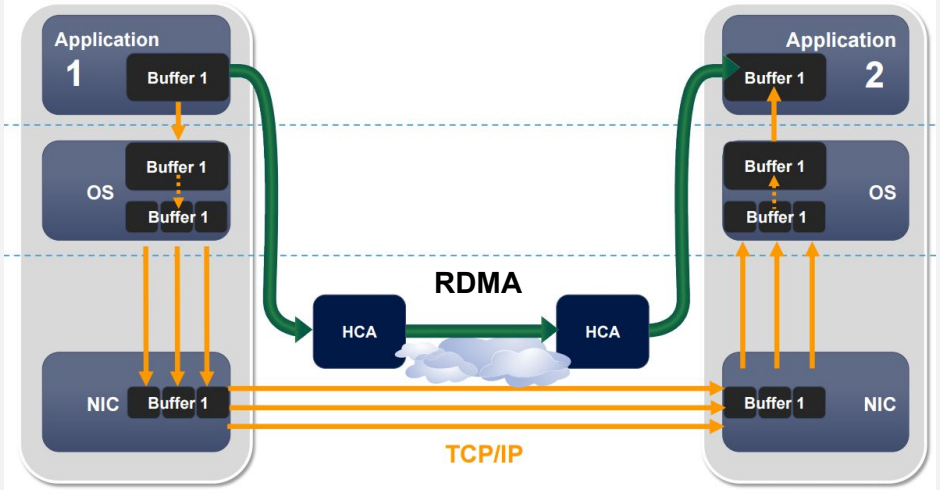
- 298M packets per second
- 3.3ns/packet



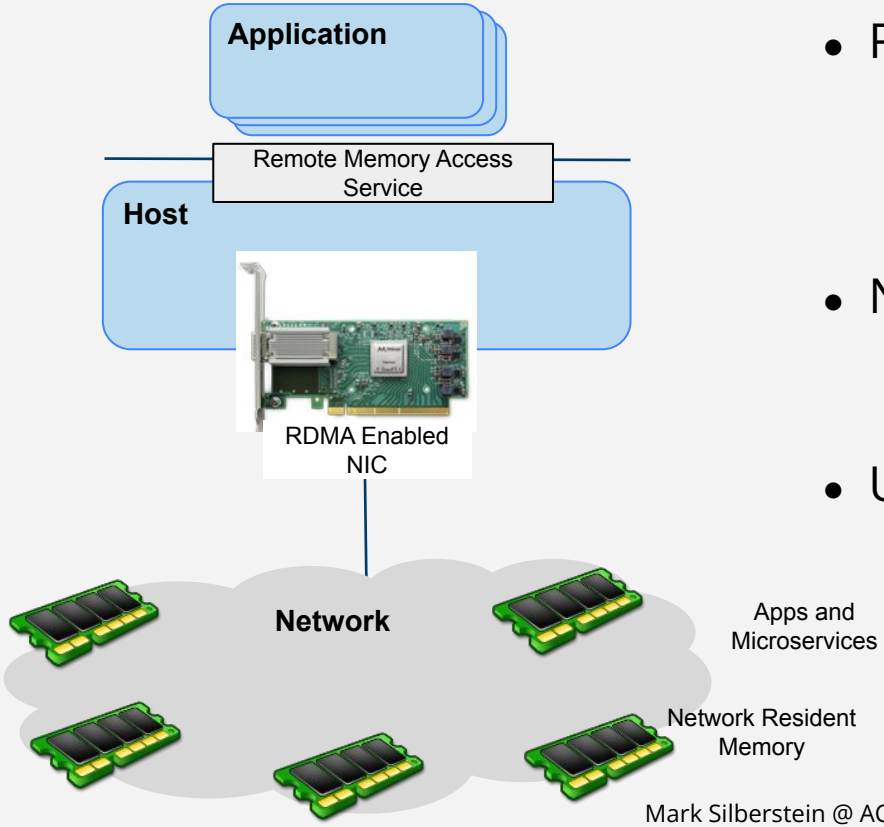
Beyond TCP

- How can we go faster than TCP?
- How can we implement a Networking Accelerator?
- What kind of new interfaces do we need to operate it?

RDMA Protocol for Better Efficiency



RDMA as an Accelerator



- RDMA Service - Network Accelerator
  - Offload packet processing and network overheads
- Network resident Memory
  - Application accessible through RDMA accelerator
- Ultra-low latency

Why do we study RDMA in this course?

- RDMA provides an efficient and generic application interface that is becoming ubiquitous in accelerated systems
  - well beyond networking, i.e., NVMe storage
- RDMA allows direct communication among accelerators
  - We will see how it is used for GPU-GPU communication without the use of CPUs
- RDMA enables building networking accelerator

Main RDMA design principles

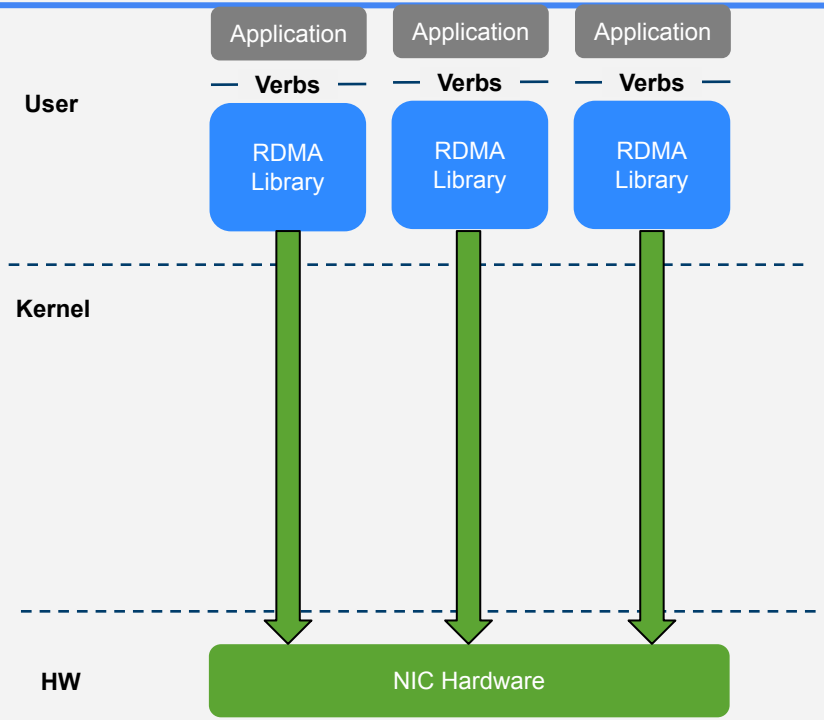
- Get rid of mediators: direct application access to NIC hardware
- Zero copy: application can send/receive data without intermediate copies
- CPU is not involved in data path: hardware-implemented transport and one-sided operations

Separation: data plane– control plane

- Data plane – data-related operations – **on** the critical path
  - Send
  - Receive
  - RDMA
  - Completion Retrieval
  - Request event
- Control plane – configuration and setup – **off** the critical path
  - Resource setup
  - Memory management

Kernel Bypass (Data plane)

- Direct access to hardware by applications
- Hardware roles (offloaded from the OS)
  - Application interface
  - Protocol stack
  - Protection
  - Resource arbitration across apps
  - Memory management (pinning, DMA)

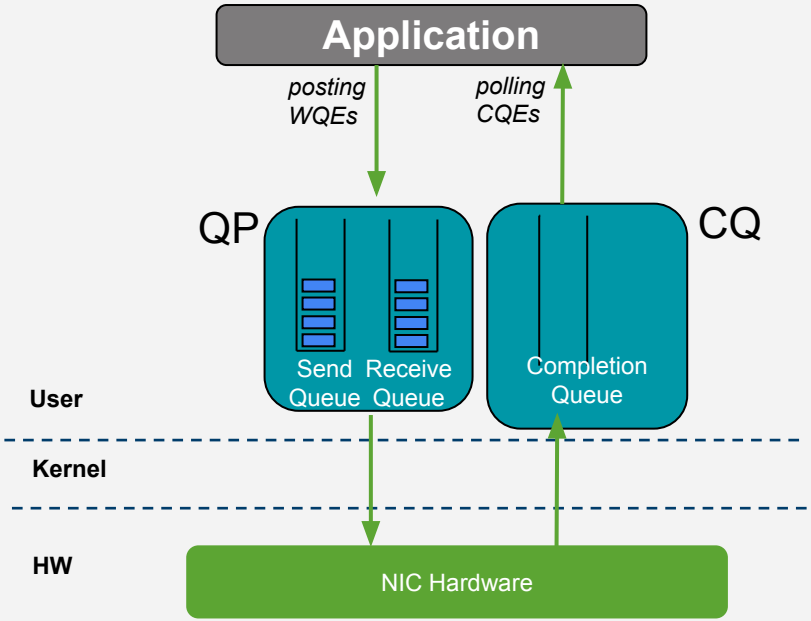


Application Interface: Verbs

- Network operation as a **job**
  - Verb commands that specify "I/O jobs"
- Dedicated queues per logical connection
  - QP (Queue Pair) – send and receive queues
  - CQ (Completion Queue) – completion queue
- Asynchronous interface
  - Enables polling and interrupts/events

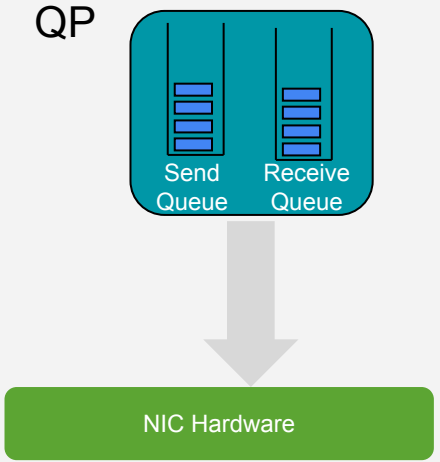
Very similar to interfaces to modern accelerators

Same as the interfaces to high performance storage



Queue Pair (QP) – Transport Endpoint

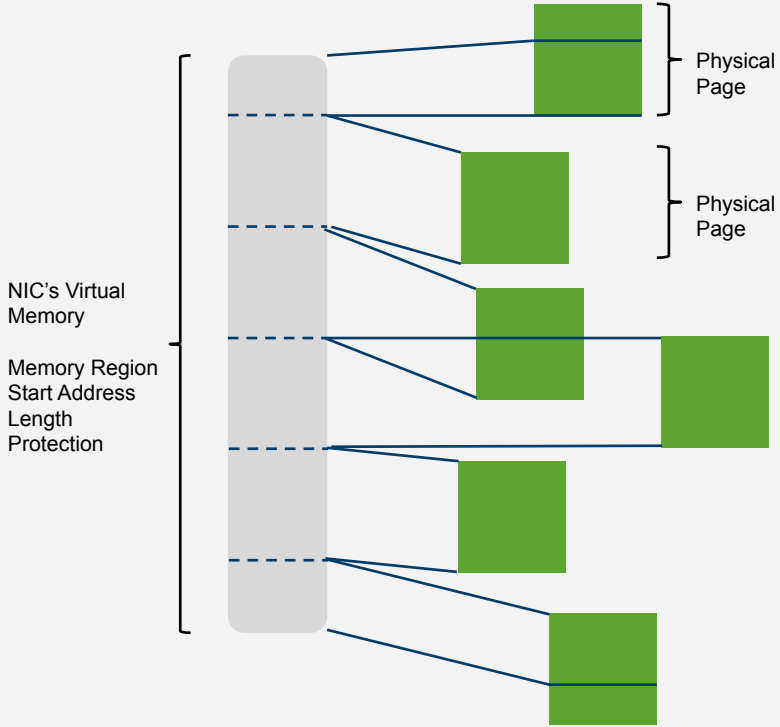
- Send and Receive Queue
- Operations: work requests
  - Send
  - Receive
  - RDMA Read
  - RDMA Write
  - Atomic
- Asynchronous operation
- Transport implemented by the NIC



Data: Memory Registration & Memory Regions

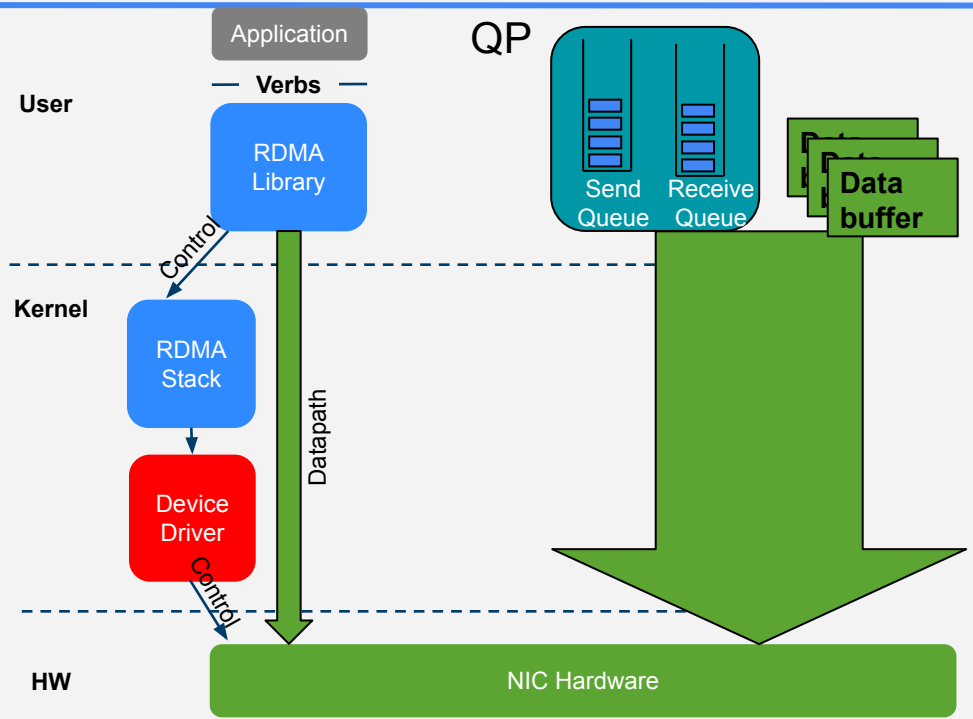
- Protection
  - Byte level range
  - Permission (R/W)
- Translation
  - Page level
- Memory Pinning
  - On demand paging option

- Each region is associated with its **r_key**
  - We use it for securing remote operations on the region



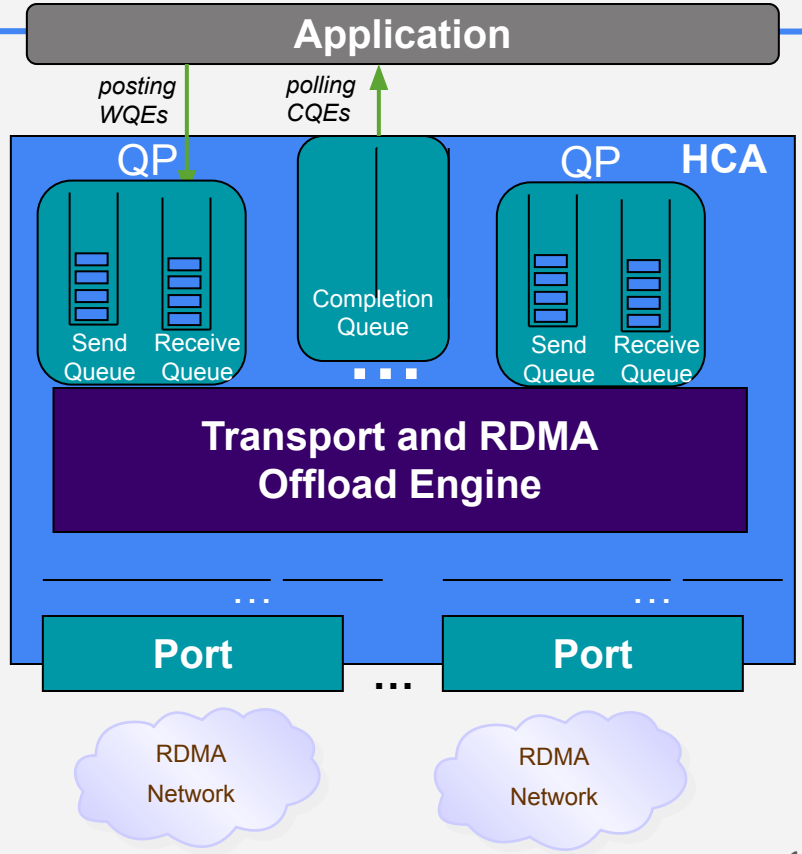
Separation data-control plane

- Separation of Control and Data paths
- Data path
  - Send
  - Receive
  - RDMA
  - Completion Retrieval
  - Request event
- Control path
  - Resource setup
  - Memory management



Transport offload – Host Channel Adapter (HCA) Model

- Asynchronous interface - Verbs
  - Application *posts* work requests
  - HCA processes
  - Application polls completions
- Transport executed by HCA
- I/O channel exposed to the application
  - Kernel bypass
- Polling and interrupt models supported



Summary

- RDMA NICs provide full transport layer offload
- Enable zero-copy transfers thanks to direct DMA into user buffers
- Separate data and control path
- Kernel bypass
- Hardware-based protection

Additional reading

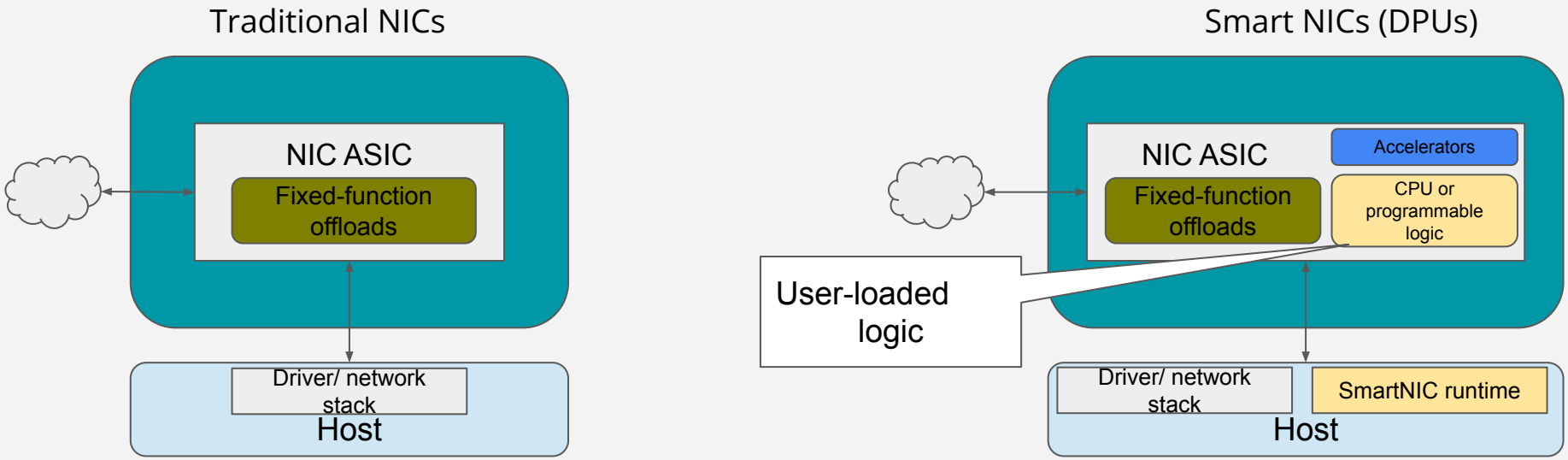
- InfiniBand Network Architecture,T. Shanley
- https://www.rdmamojo.com

Questions for self-study [6]

1. How does RDMA ensure that remote memory is securely exposed to remote party?
2. Why does HCA need to implement memory translation unit?
3. What are the ordering guarantees for RDMA writes performed on the same QP?
4. Can multiple QPs used to connect to the same remote host?
5. What is the role of the OS kernel?

SmartNICs

Architecture at high level

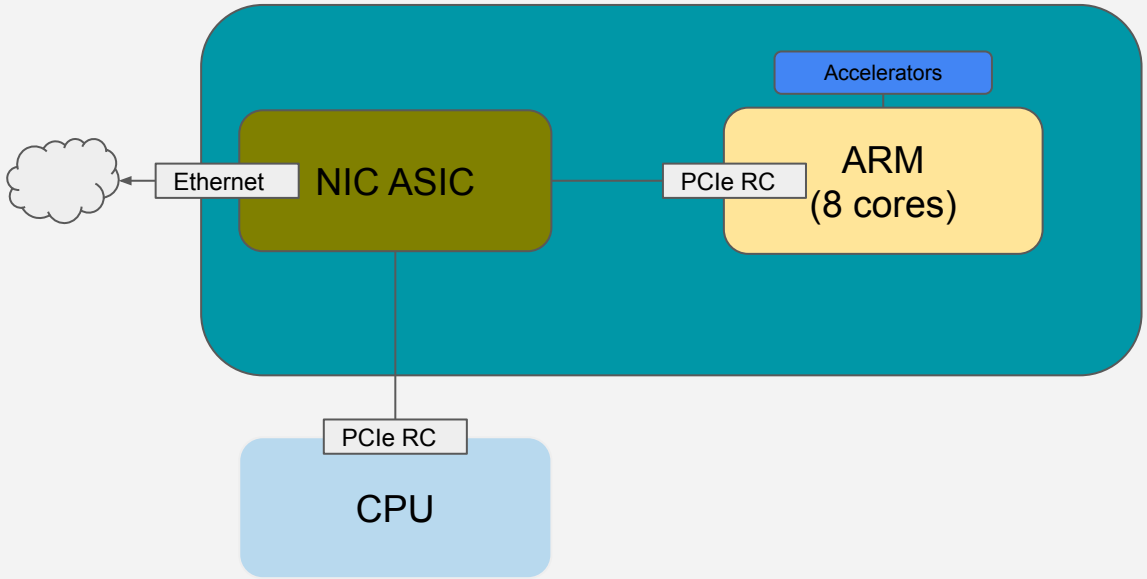


SmartNICs: architectural benefits

- ✓ Reduced PCIe load on the host
  - Ingress – packets can be filtered out without reaching the host
- ✓ Optimized I/O path between the ASIC and the processing logic
  - SoC interconnect under NIC vendor's control
- ✓ Air-gapped, controls all Network IO for the host
  - Host CPU may be blocked from accessing the logic running on the NIC
- ✓ NIC vendors can add custom accelerators to the SoC

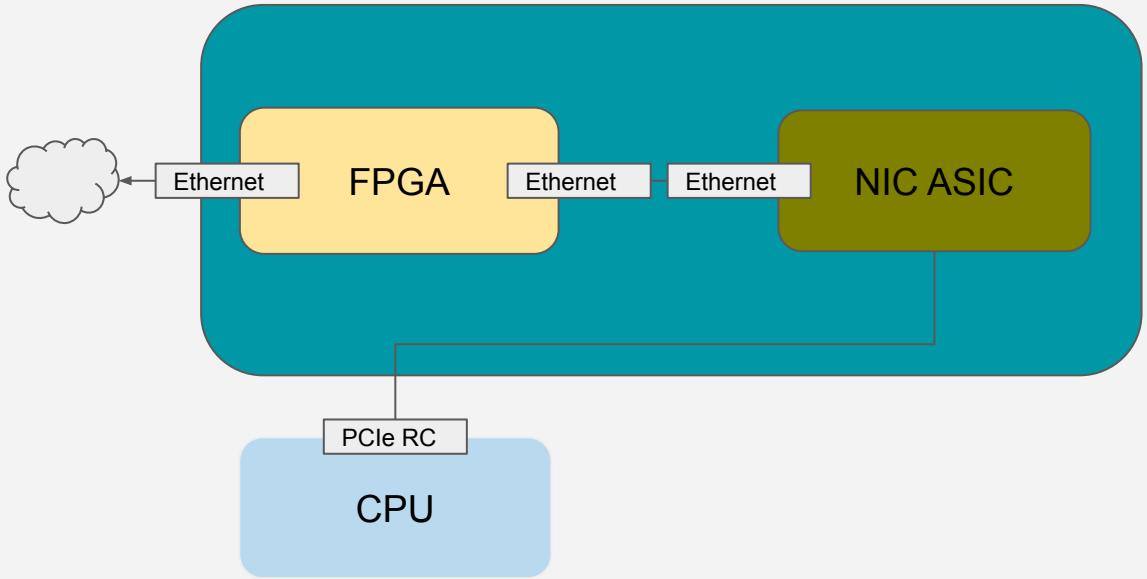
NVIDIA Networking DPU (Bluefield)

- Dual port 25GbE
- ConnectX-5 ASIC NIC
- PCIe Gen 4.0 to ARM
- 8x 64bit ARMv8 A72
- 800 MHz
- HW-accelerated RDMA (RoCE and Infiniband)
- Runs Linux (BlueOS)
- 2 modes
  - Symmetric
  - ARM-controlled



Mellanox Innova (bump-in-the-wire)

- 40GbE
- ConnectX-4 ASIC NIC
- PCIe Gen 3.0
- HW-accelerated RDMA (RoCE and Infiniband)
- Xilinx Kintex UltraScale XCKU060

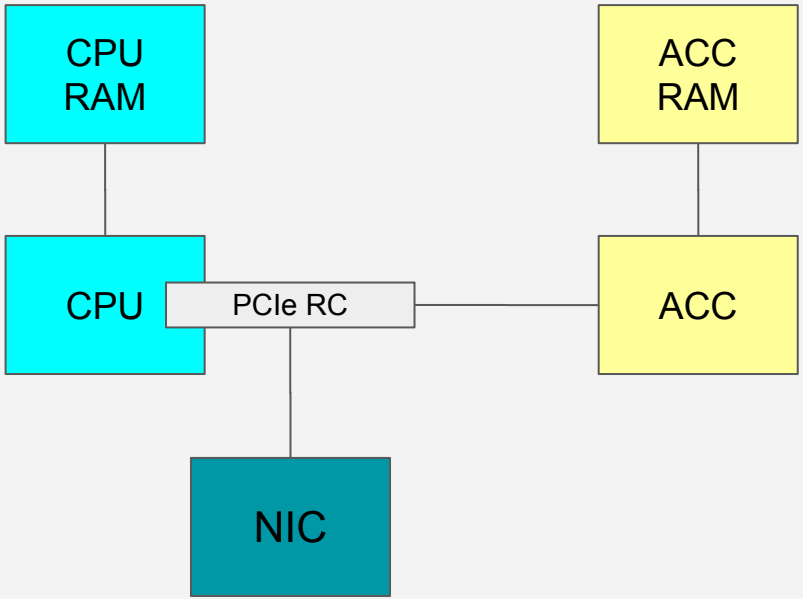


Additional reading

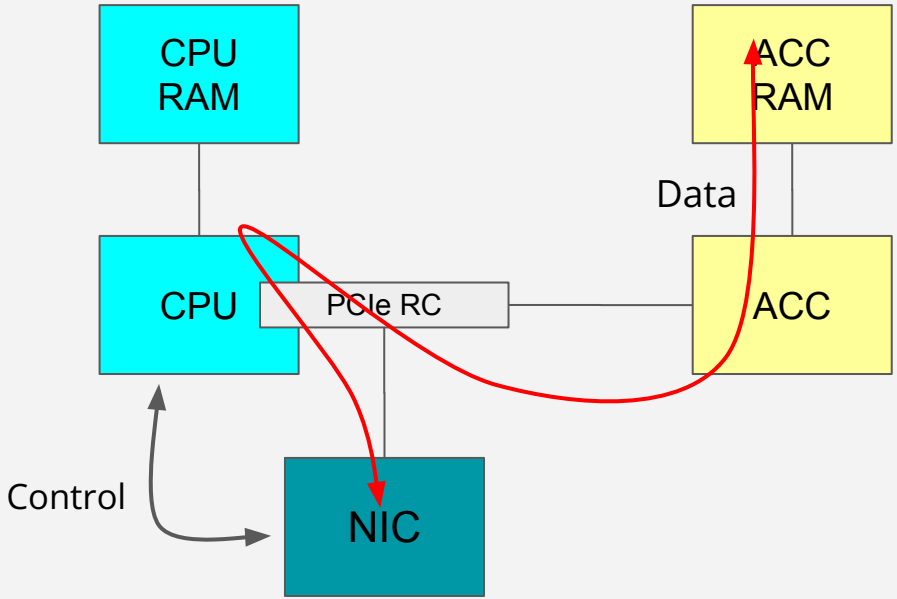
- The new life of SmartNICs
  - https://www.sigarch.org/the-new-life-of-smartnics
- Netronome
  - https://www.netronome.com
- NVIDIA DOCA SDK
  - https://developer.nvidia.com/networking/doca
- Look for my short intro videos about SmartNICs as part of the JWinsight interviews
  - https://jw.ijiwei.com/

Accelerator-native networking

Traditional RDMA I/O path to accelerators

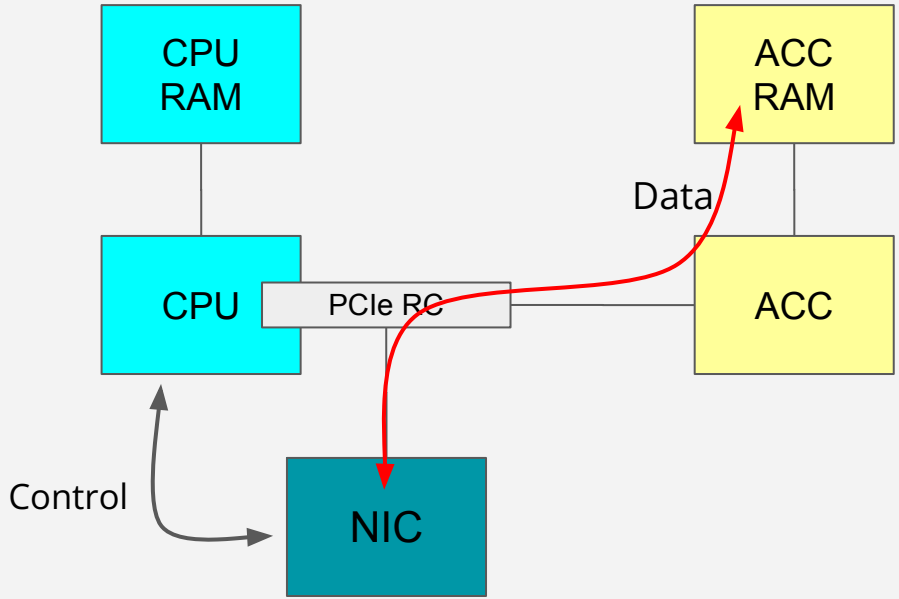


Traditional RDMA I/O path to accelerators

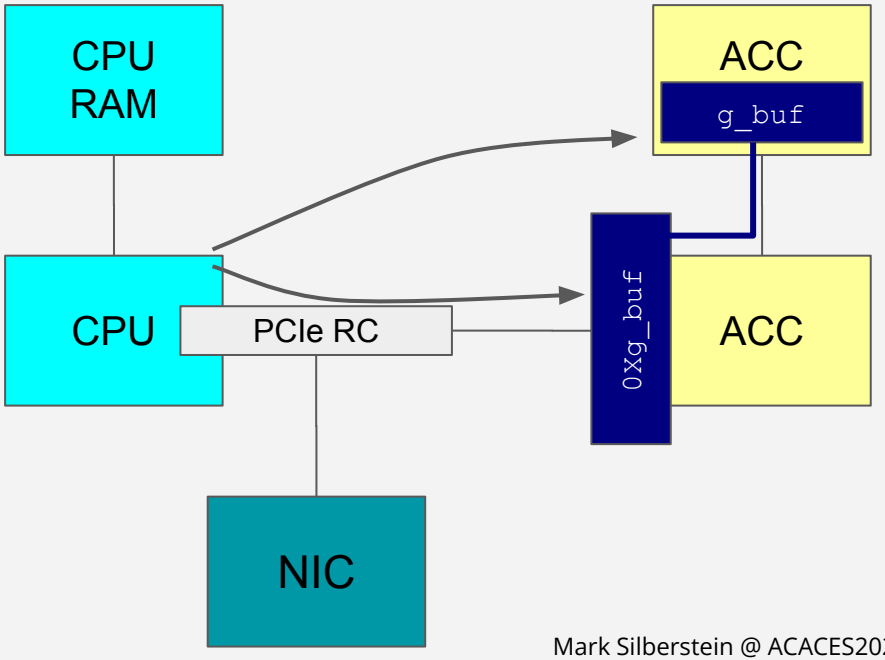


- Unnecessary PCIe load
- CPU involvement in data movements
  - DMA programming
- CPU Cache pollution

Direct data path for RDMA to accelerators



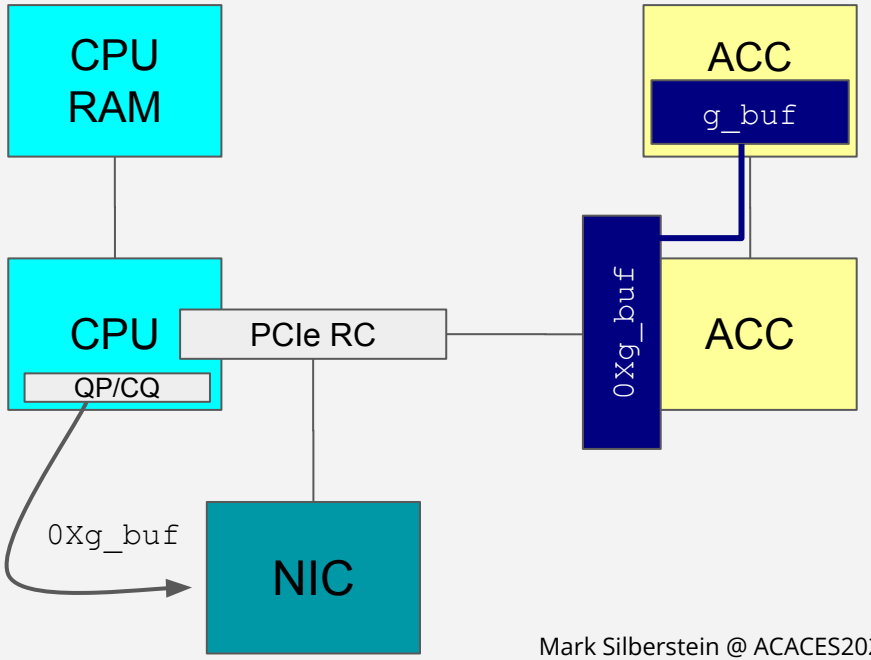
PCIe magic in action



Example: GPUdirectRDMA

- CPU allocates buffers in GPU memory
- GPU exposes these buffers on its PCIe BAR

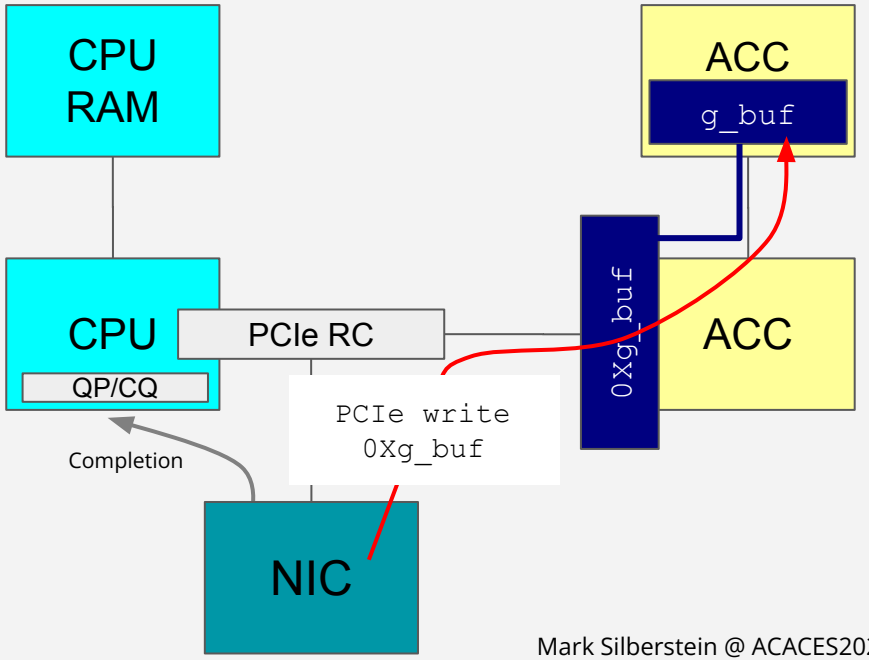
PCIe magic in action



Example: GPUdirectRDMA

- CPU allocates buffers in GPU memory
- GPU exposes these buffers on its PCIe BAR
- CPU configures the NIC to access buffers via bus addresses of the GPU PCIe BAR
- PCIe routes data from the NIC based on the bus address

PCIe magic in action



Example: GPUdirectRDMA

- CPU allocates buffers in ACC memory
- ACC exposes these buffers on its PCIe BAR
- CPU configures the NIC to access buffers via bus addresses of the ACC PCIe BAR
- PCIe routes data from the NIC based on the bus address
- CPU receives interrupted when data arrives
- **PCIe guarantees** that data in ACC arrives in full before CPU is notified

Is this enough? No!

- Accelerator cannot initiate I/O
  - Must be invoked after data arrives
  - Must terminate before sending out
- Network latency affected
- Accelerator local cache flush
- Bulk-synchronous design

CPU code

```
CPU_rdma_read()
InvokeGPUandWait()
CPU_rdma_write()
```

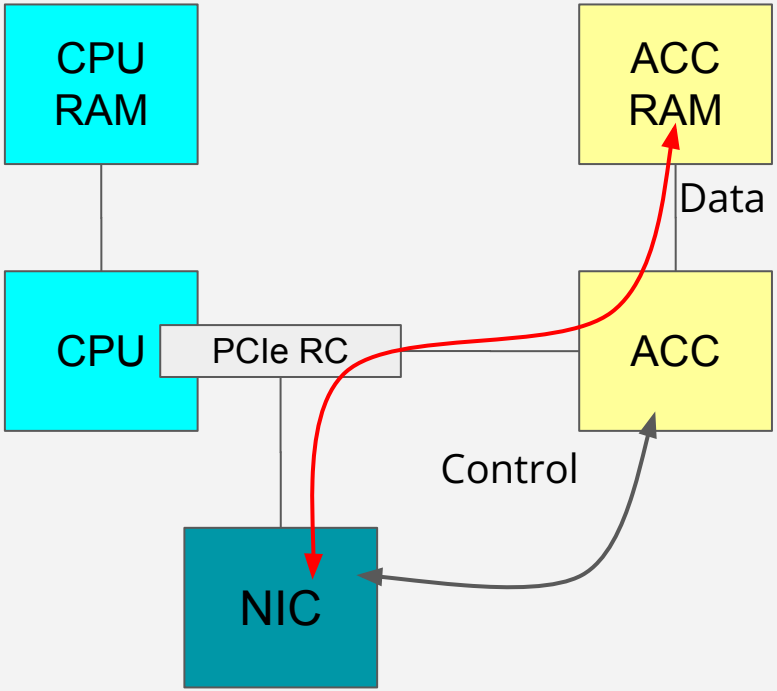
If only ACC were able to perform RDMA operations

- Low latency
- No CPU overheads
- No bulk-synchronous restrictions
- Easy overlap between computations and communications

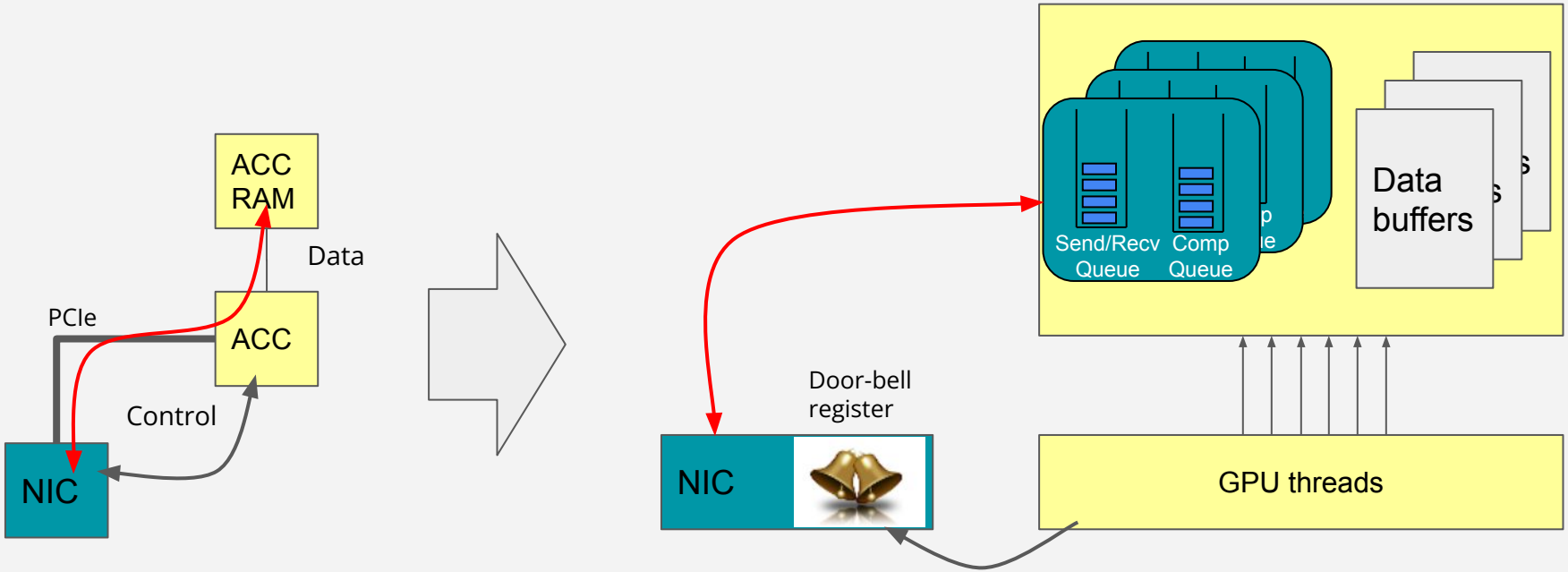
Accelerator code

```
ACC_rdma_read()
compute()
ACC_rdma_write()
```

GPUrdma: access RDMA NIC directly from GPU



All QPs are in GPU memory



Results

- 50Gbps (5% lower than CPU RDMA) for 16K and up
- 4.5x higher throughput for small transfers
- Round-trip 4.8usec latency (5x lower than with RDMA via CPU)
  - Later reduced to 2usec
- Up-to 5x faster kernel execution for synthetic cases

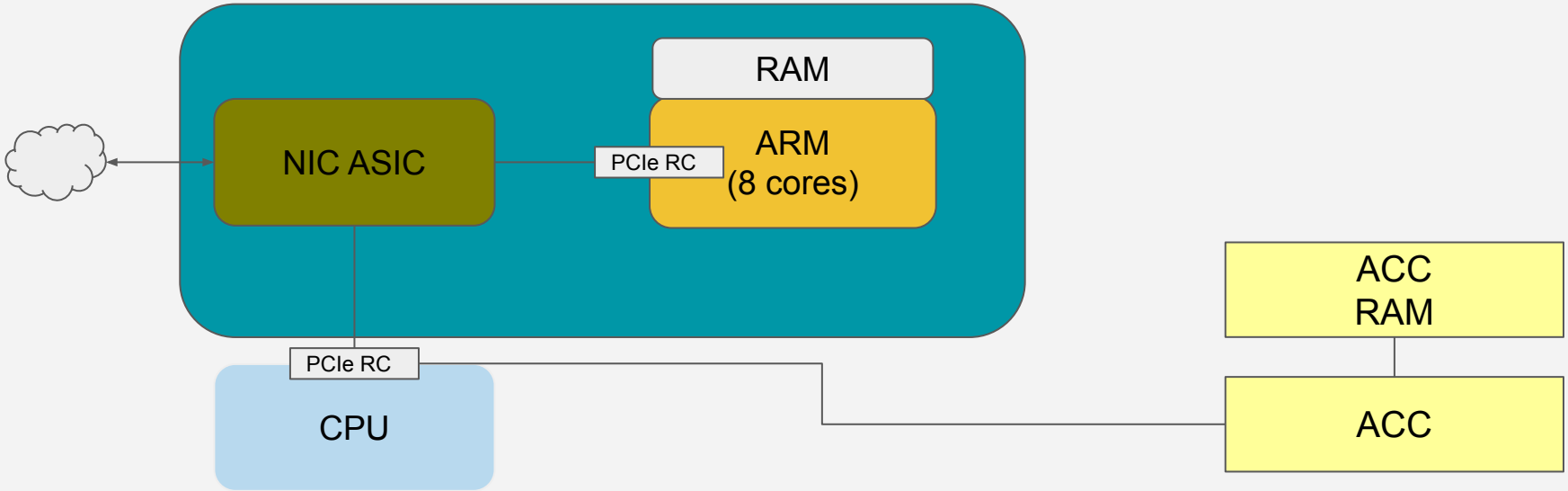
Results

- 50Gbps (5% lower than CPU RDMA) for 16K and up
- 4.5x higher throughput for small transfers
- Round-trip 4.8usec latency (5x lower than with RDMA via CPU)
  - Later reduced to 2usec
- Up-to 5x faster kernel execution for synthetic cases

Problem 1: updates to doorbell registers are blocking GPU kernels!
Problem 2: RDMA VERB creation is too costly!
Problem 3: Only works for RDMA (not for TCP)

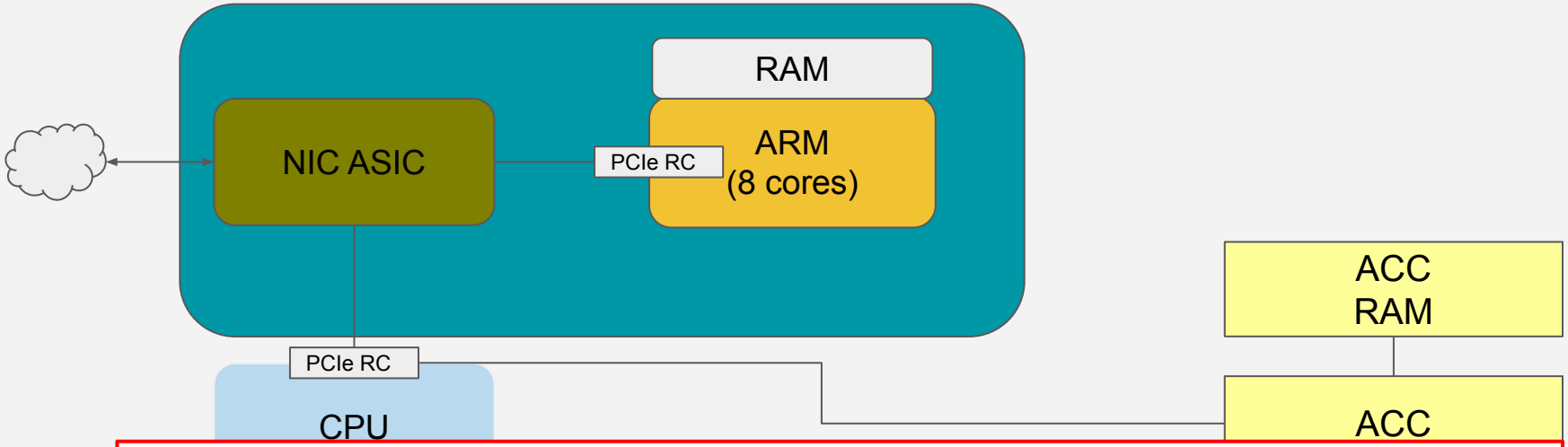
SmartNICs to rescue! But how?

- Goal: enable accelerator networking ***with minimum overheads for accelerators***
- Can we use SmartNICs to offload I/O overheads from accelerators?



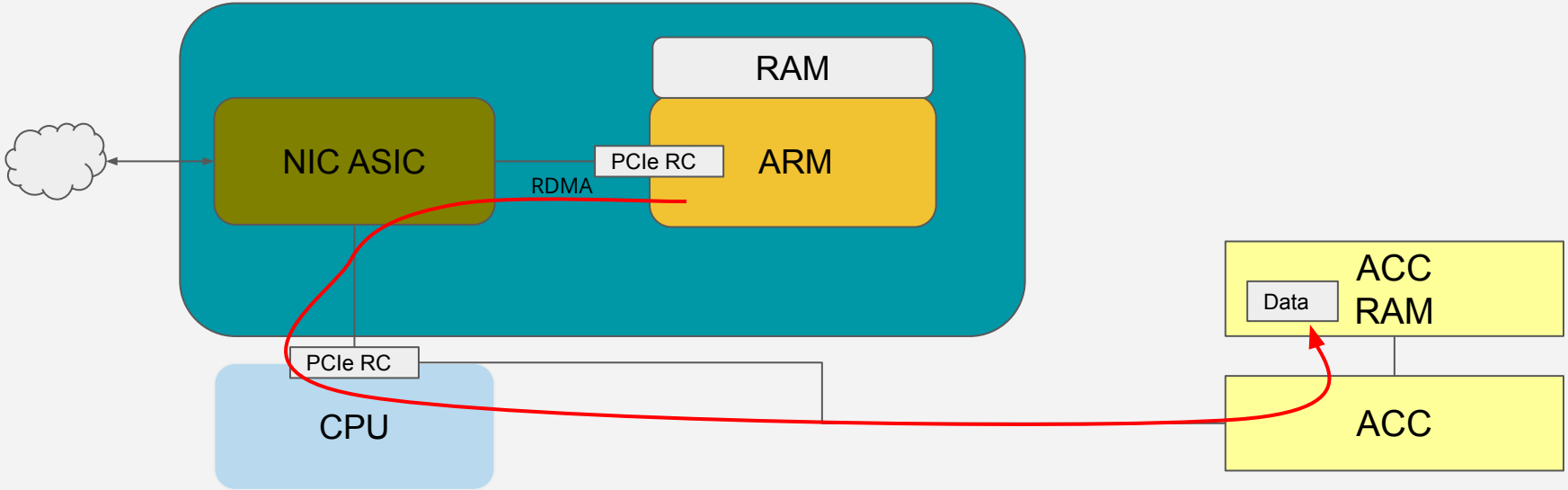
SmartNICs to rescue! But how?

- Goal: enable accelerator networking ***with minimum overheads for accelerators***
- Can we use SmartNICs to offload I/O overheads from accelerators?

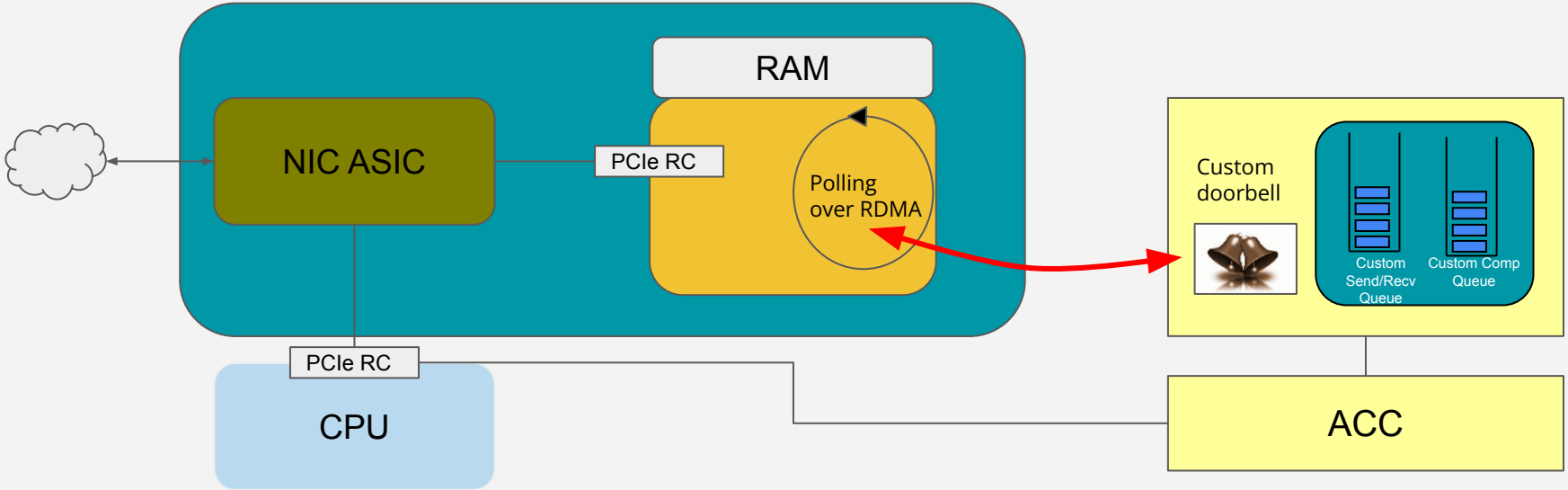


Challenge: ARM on a SmartNIC **cannot** access host's PCIe!

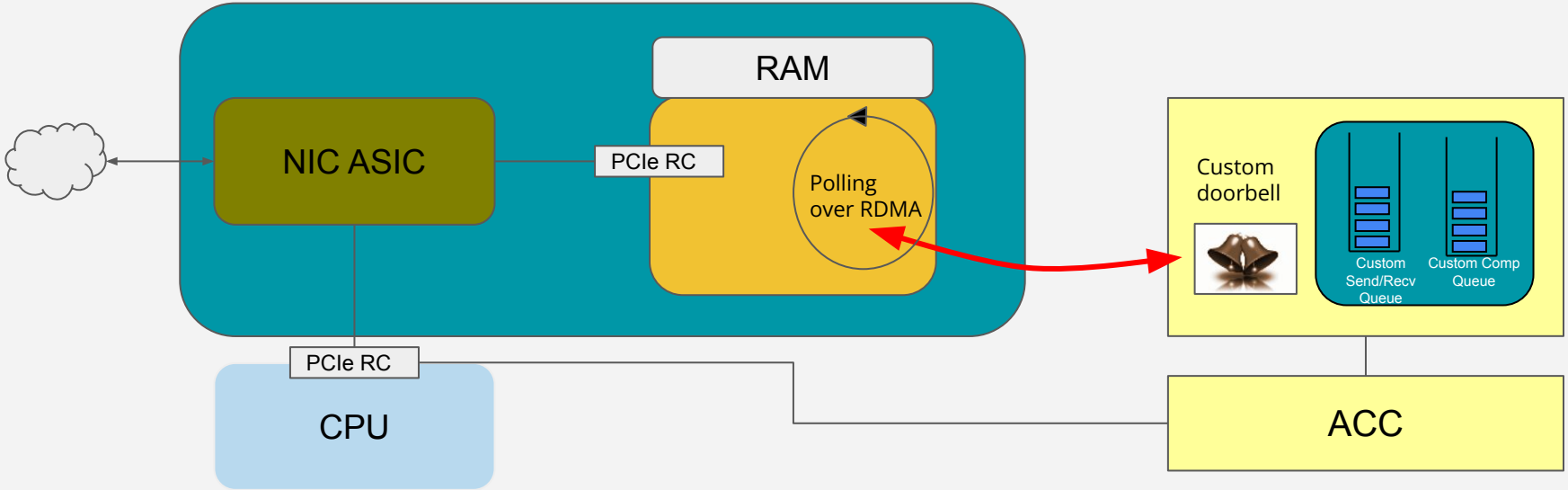
Idea: Use RDMA from ARM to access Accelerator



ARM manages I/O queues in ACC RAM via RDMA

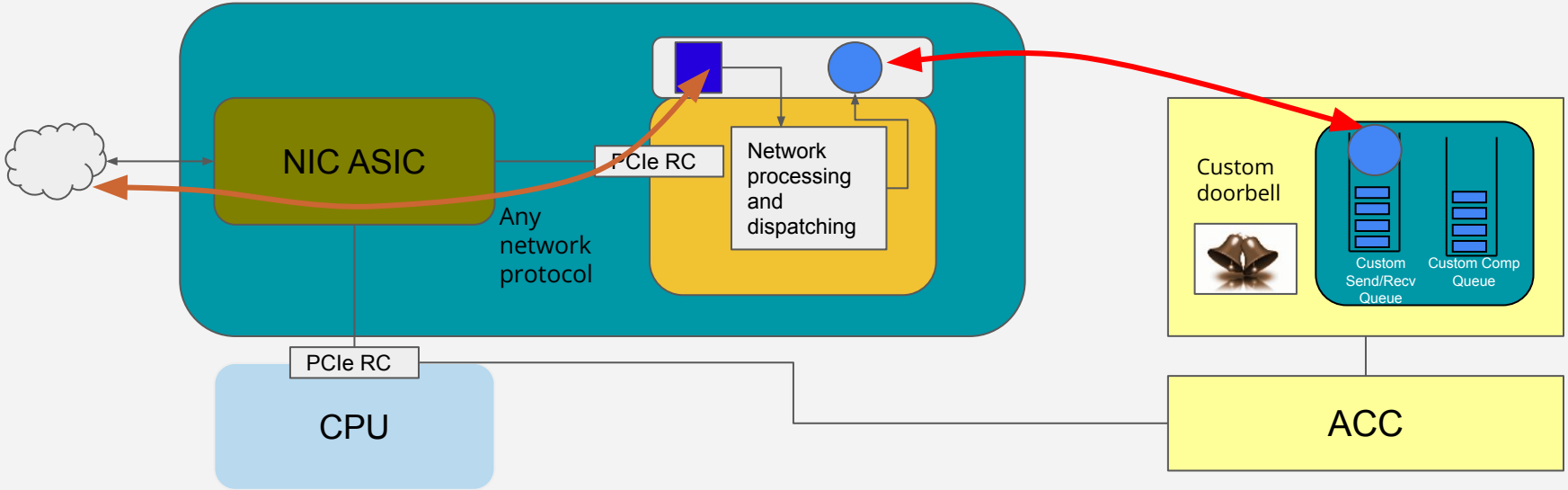


ARM manages I/O queues in ACC RAM via RDMA



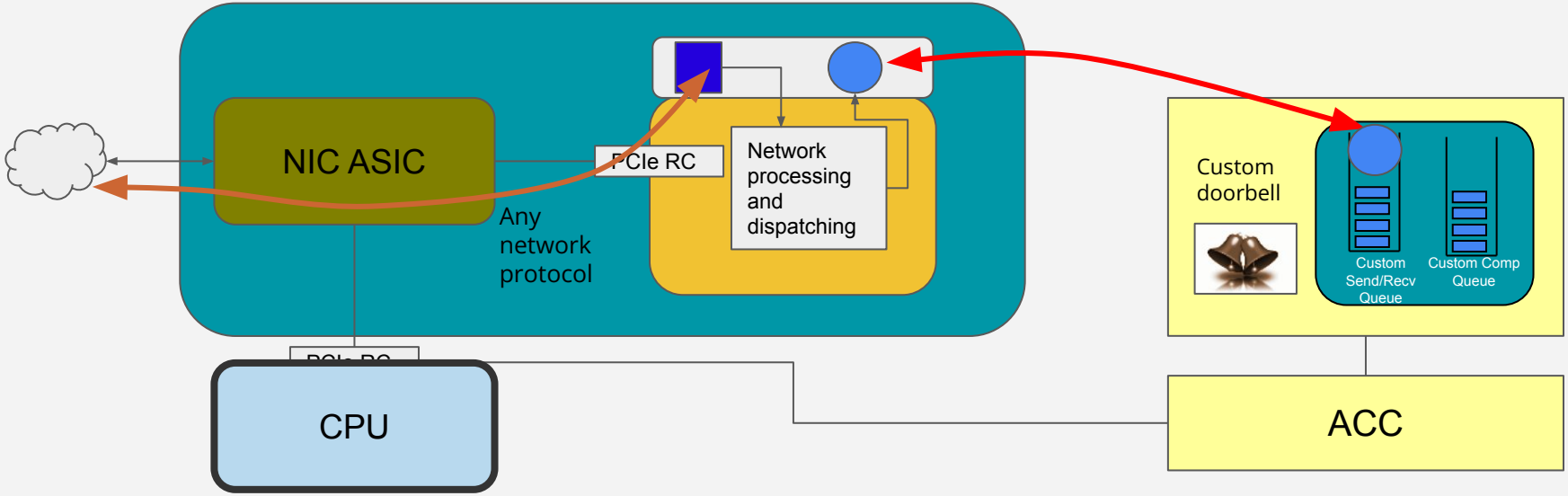
~~Problem 1: updates to doorbell registers are blocking GPU kernels!~~
~~Problem 2: RDMA VERB creation is too costly!~~
Problem 3: Only works for RDMA (not for TCP)

Can support any network protocol



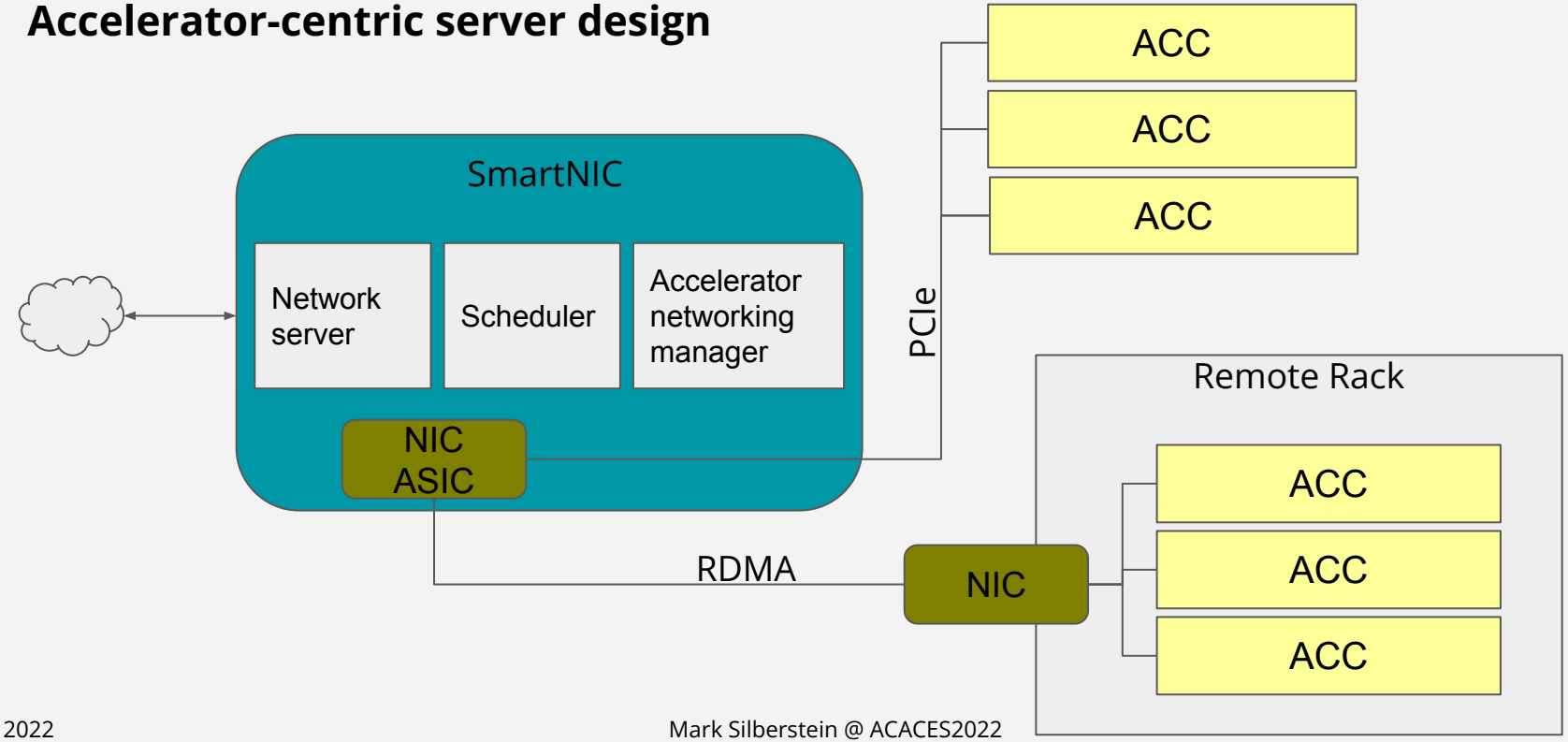
~~Problem 1: updates to doorbell registers are blocking GPU kernels!~~
~~Problem 2: RDMA VERB creation is too costly!~~
~~Problem 3: Only works for RDMA (not for TCP)~~

X86 CPU is idle and can be used for other applications



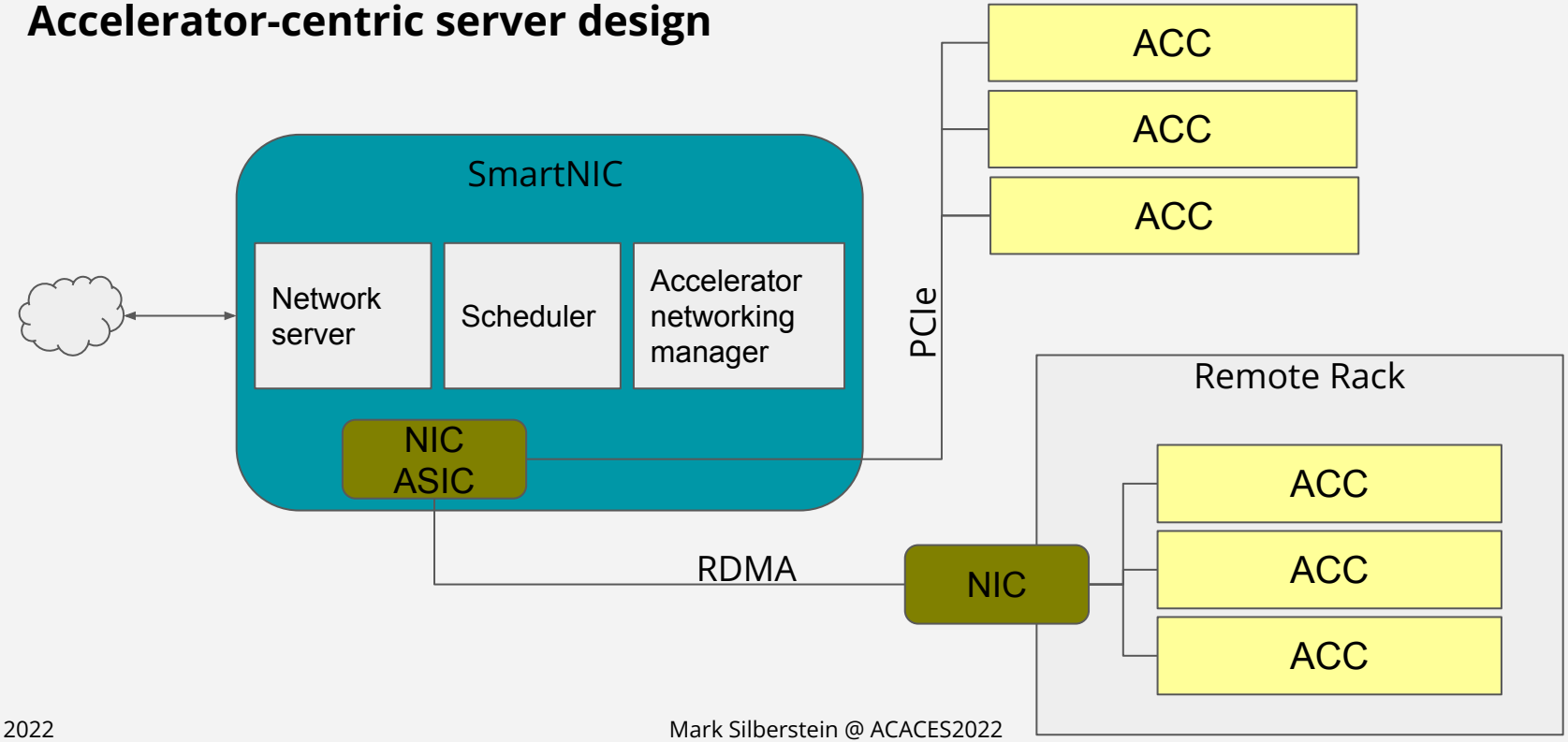
We can use the same design for more!

Accelerator-centric server design



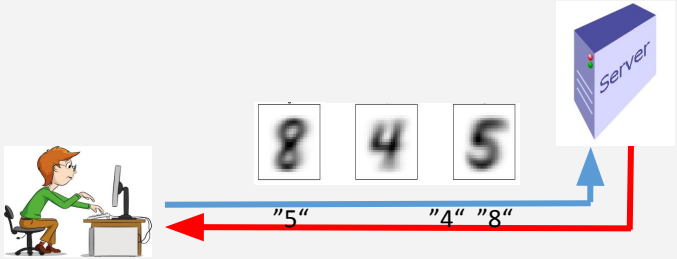
We can use the same design for more!

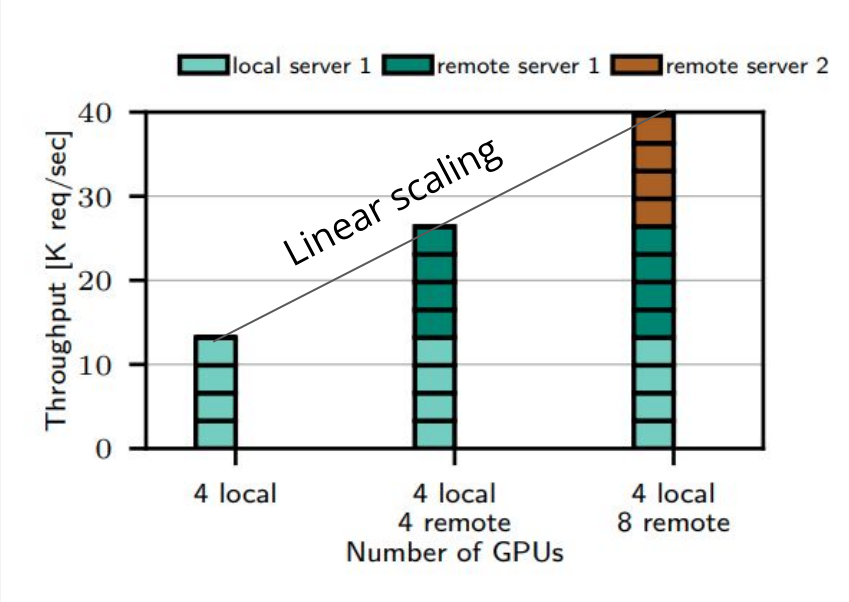
Accelerator-centric server design



Accelerator-centric design in real life

Inference server: MNIST NN running on 4 local and 8 remote GPUs





Projection
A single BlueField SmartNIC can manage up to 100 GPUs (UDP) and 15 GPUs (TCP) for MNIST inference (300 usec, small images)

Summary

- Direct I/O from accelerators is possible
- x86 CPUs can be freed for workloads they are good at → general system efficiency increased
- Requires careful engineering of on-accelerator network layer to avoid prohibitive accelerator overheads
- SmartNICs can be used to offload I/O overheads from accelerators without involving the host CPU

Additional reading

- GPUnet: Networking abstractions for GPU programs (OSDI'14)
  - sockets for GPUs
- GPUrdma: GPUside library for high performance networking (ROSS'16)
  - RDMA for GPus
- Lynx: a SmartNIC-driven accelerator-centric architecture for network servers (ASPLOS'20)

- NICA: An infrastructure for inline acceleration of network applications (USENIX ATC'19)
  - Inline processing framework for SmartNICs with FPGA
- FlexDriver: A network driver for your accelerator (ASPLOS'22)
  - Controlling the NIC ASIC from the accelerator to leverage NIC fixed-function offloads

Questions for self-study [7]

1. What is the main architectural property of BlueField SmartNIC that makes it appropriate for intrastructure offloads, such as server management?
2. Why GPUrdma suffers from higher GPU overheads than SmartNIC-driven design?
3. How is the system design affected by the fact that the ARM processor on the SmartNIC has a PCI Root Complex
4. What are the key performance limitations of the SmartNIC-driven design?
5. Is Acc DMA used in SmartNIC-driven design?

Accelerator-native storage I/O
(we will talk about GPUs)

What if we need to access a file from a GPU?

- Standard mode: co-processor
  - CPU reads from file, copies to GPU memory
  - GPU is invoked
  - CPU copies from GPU memory, writes to file

What if data access pattern is input-driven?

I/O intensive applications
Example:Image collage

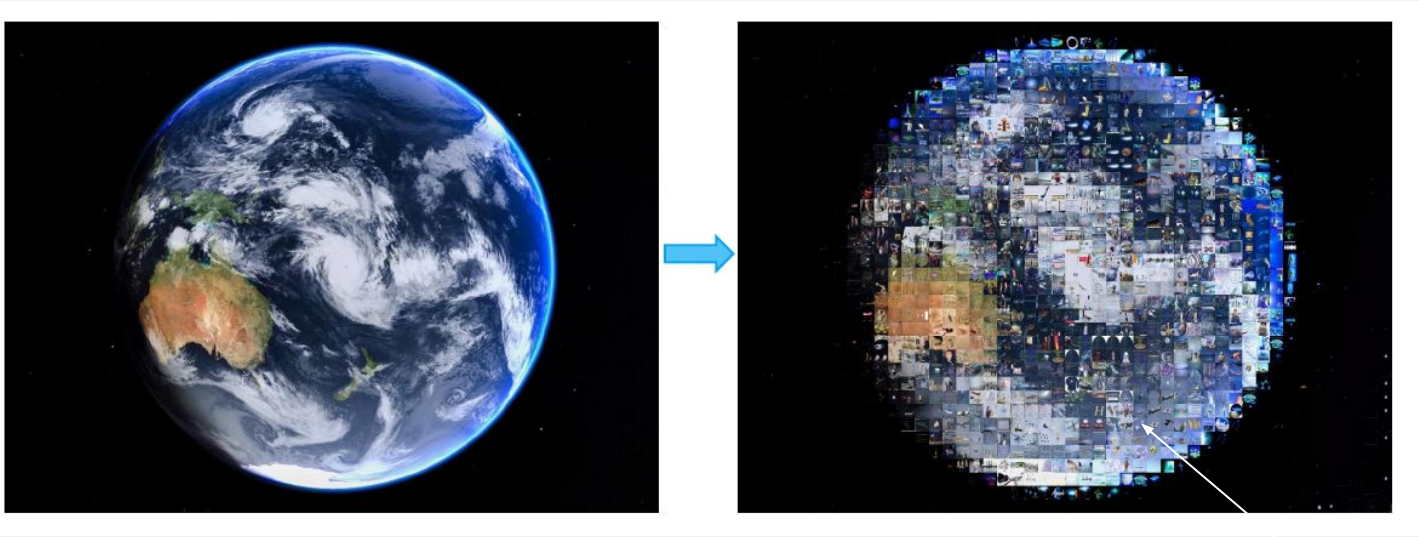
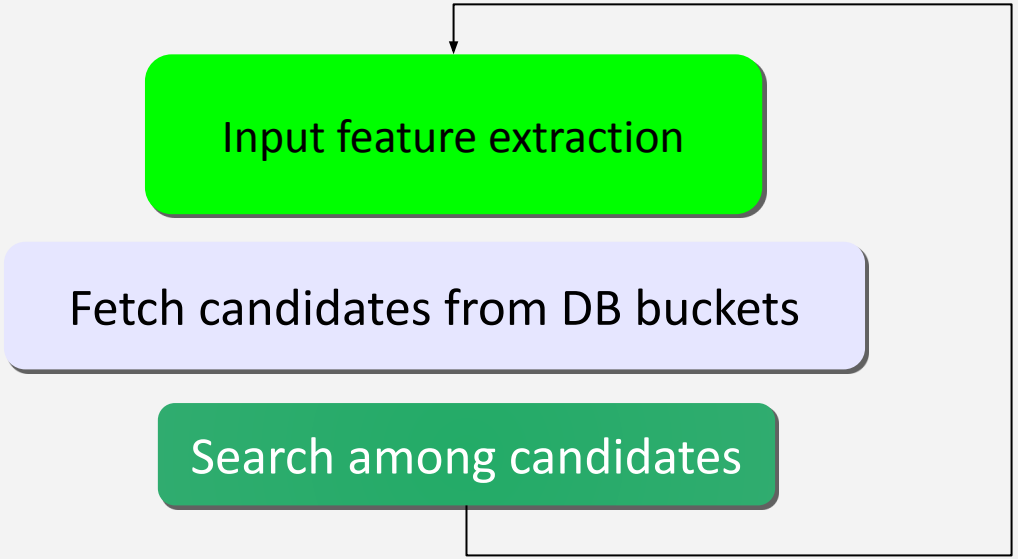


Image dataset: 40 GB of data

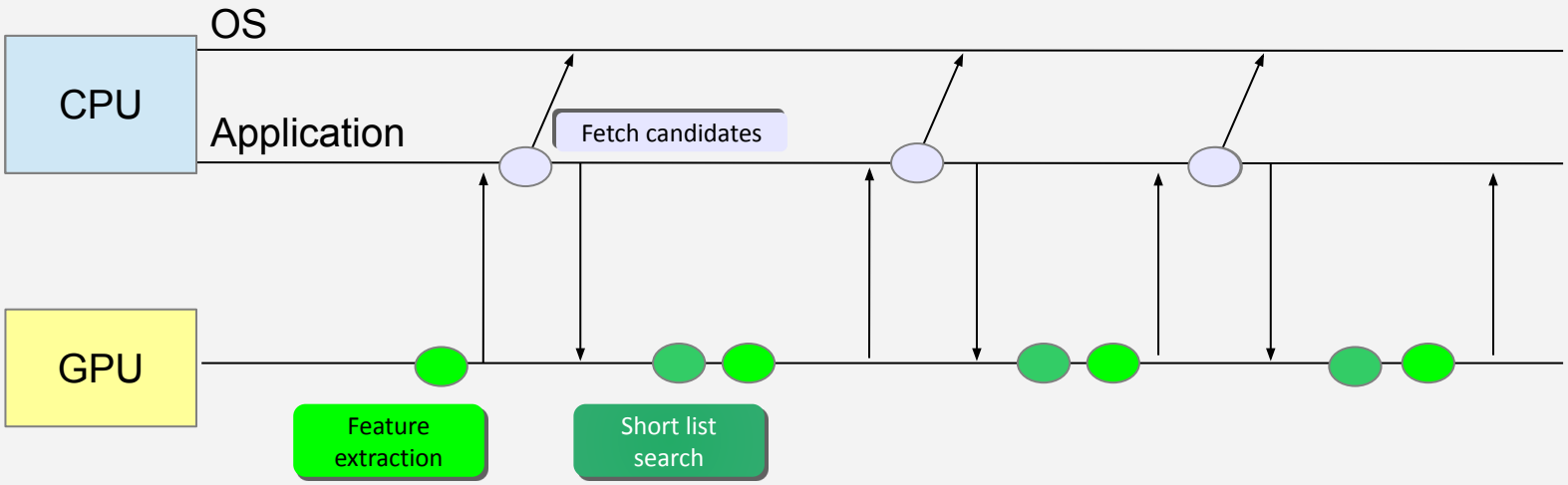
Algorithm: locality sensitive hashing

Preprocessing: group the dataset images into “buckets” according to some features, place them in a DB

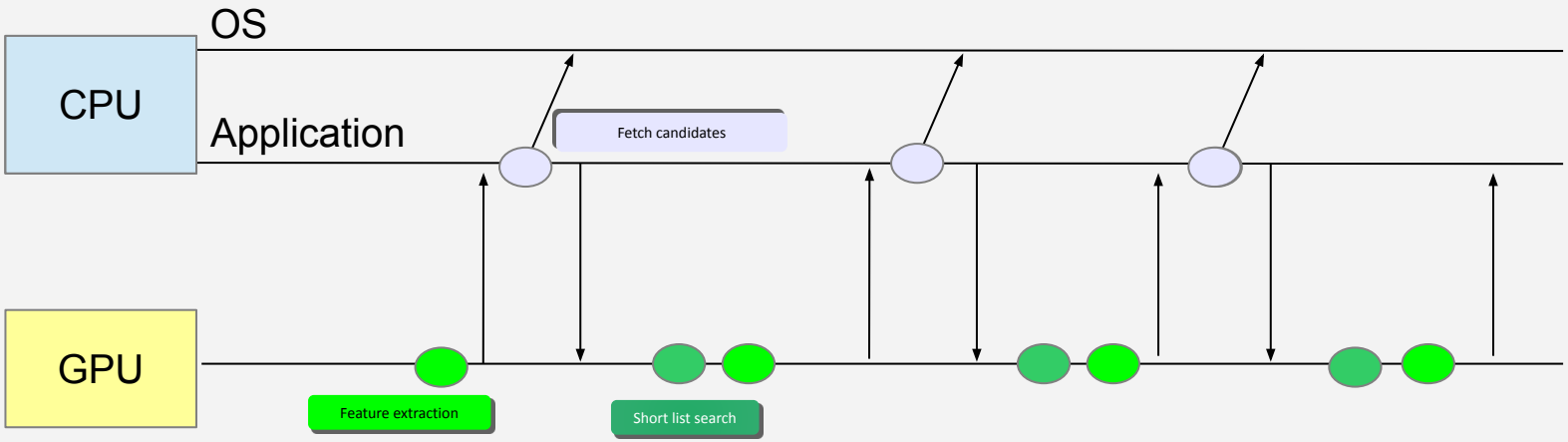


For every block
in the input image

Offloading computations to GPU



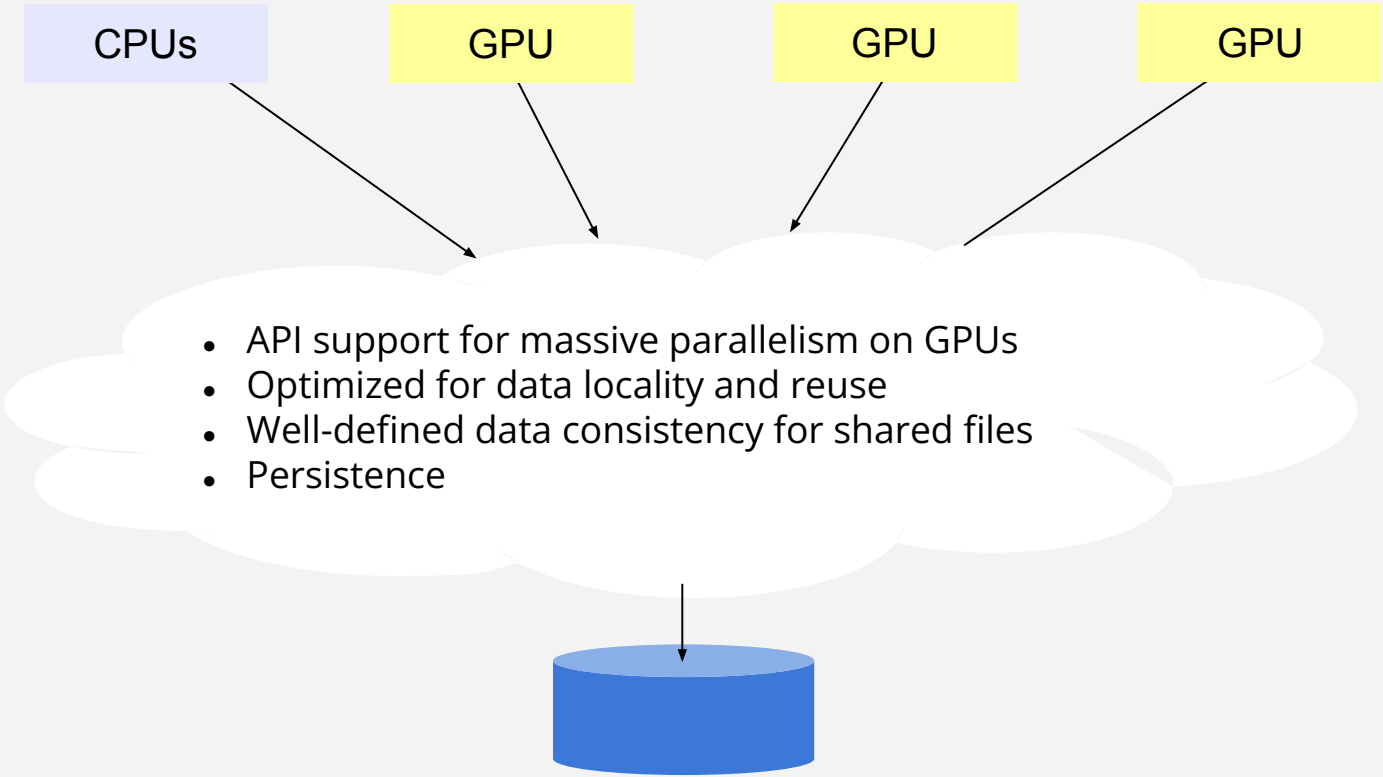
Offloading computations to GPU



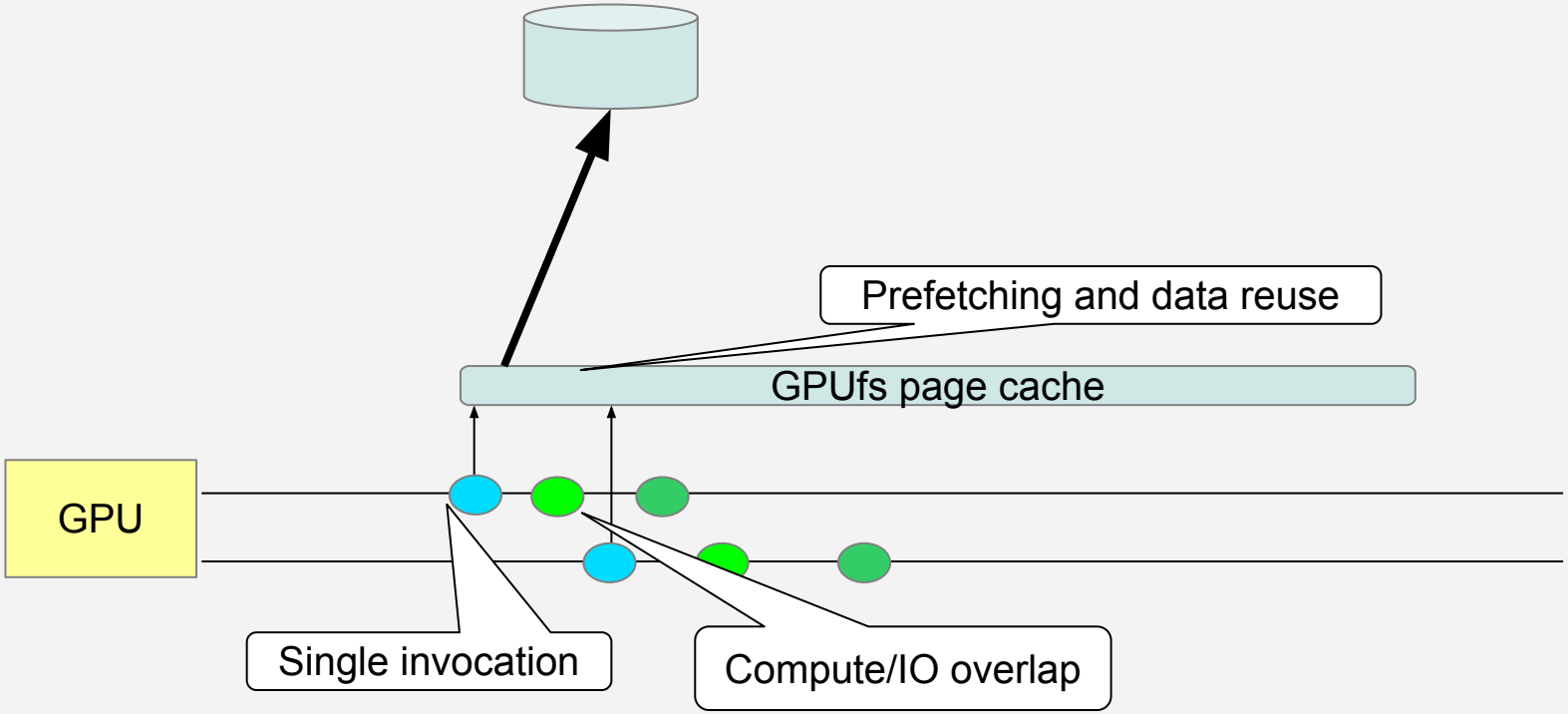
This is highly inefficient:

1. Low granularity of processing and GPU kernel invocation overheads
2. Manual data reuse management
3. CPU-mediated data path

GPUfs: accessing files from GPU programs



GPU execution with GPUfs



What GPUfs can do for GPU programmers

- Simpler CPU-like development
- Dynamic working sets
- Support for large data sets
- Portability and forward compatibility
- Interoperability with legacy programs
- Coordination with peer GPUs and CPUs

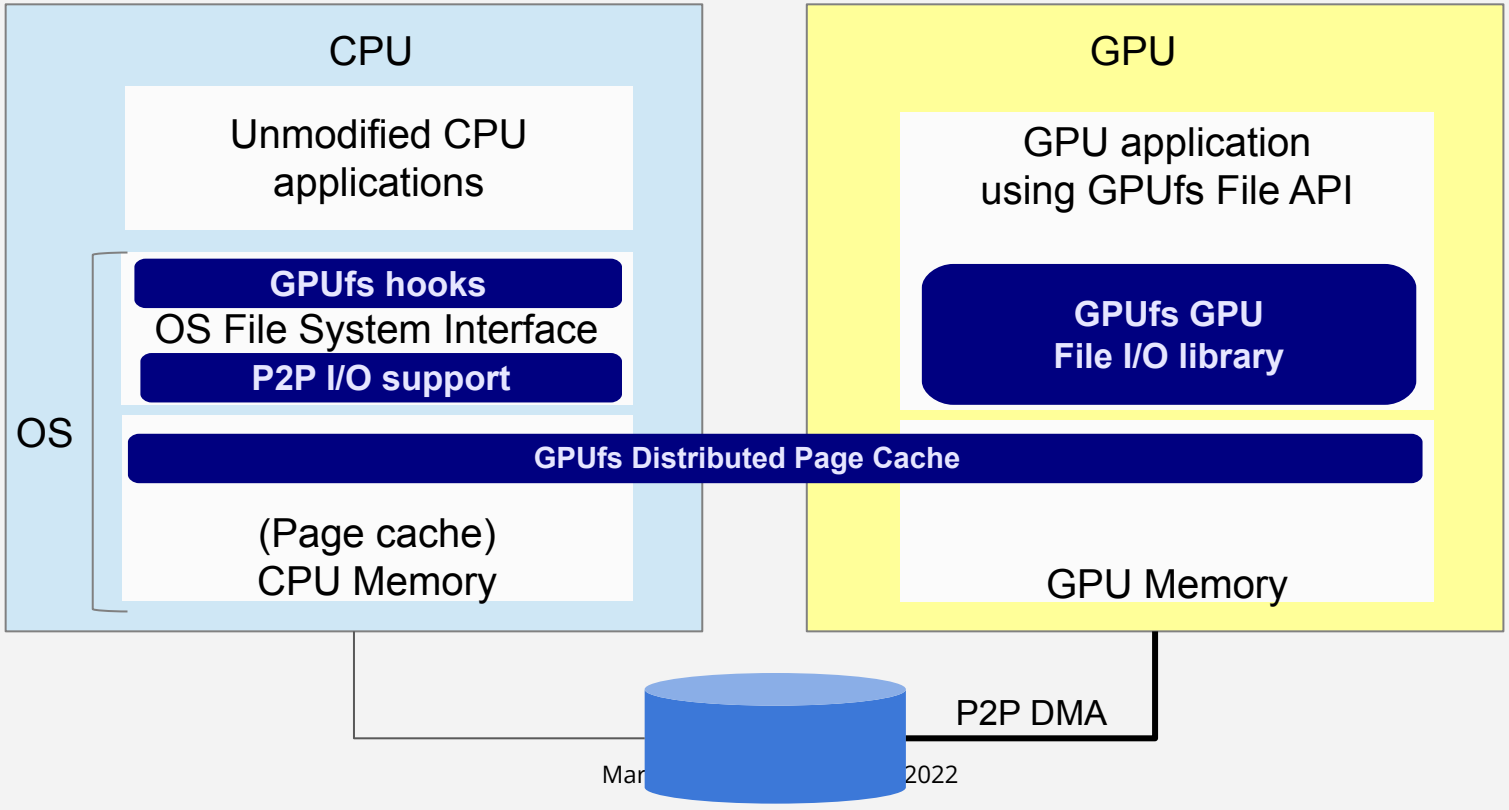
Do we need CPU in the game?

- Needed: **direct data path to files from GPU (without CPU)**

BUT!

- We want to allow a single shared FS, therefore..
  - File system runs on the CPU OS, so need to access files *via CPU* to resolve file offsets to disk block and enforce file permissions
  - Files are shared with the CPU, so need to take care of file caching in the page cache

GPUfs design



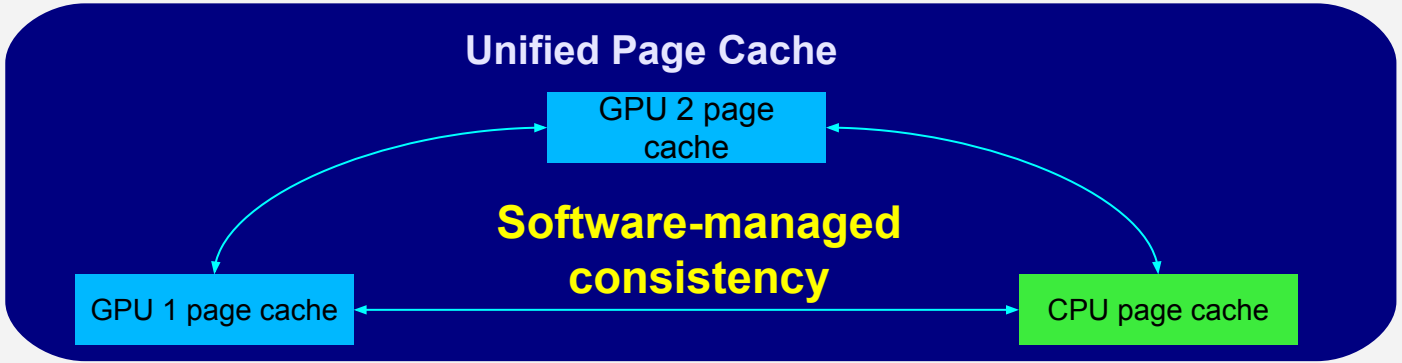
GPU program using GPUfs

```
__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);
```

This code runs in all GPU threads

System-wide page cache

- Logical (user) view: ***cross-application*** page cache spanning all GPUs and CPUs
- Physical (system) view: private cache per device



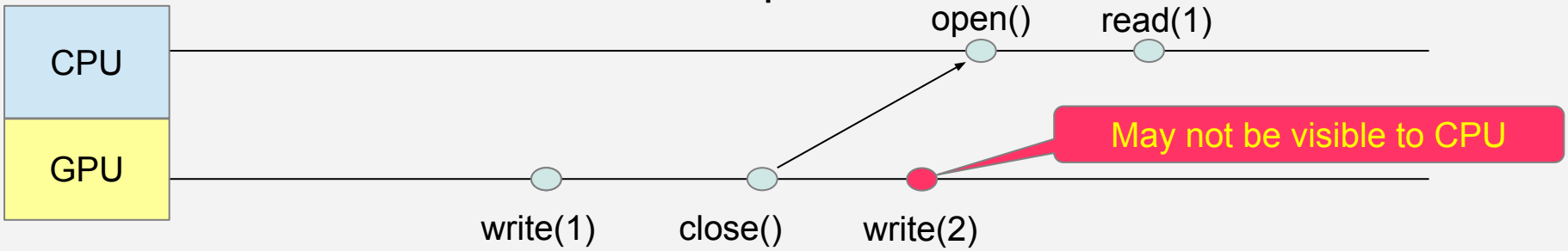
Reminder: page cache

- Page cache is managed in memory pages
- Holds radix trees of data that belongs to a particular file indexed by file offset
- When accessing a file, first check in page cache

What if a file gets accessed by CPU and GPU (and also cached in both)?
How to define memory consistency?

Weak data consistency to minimize inter-processor synchronization

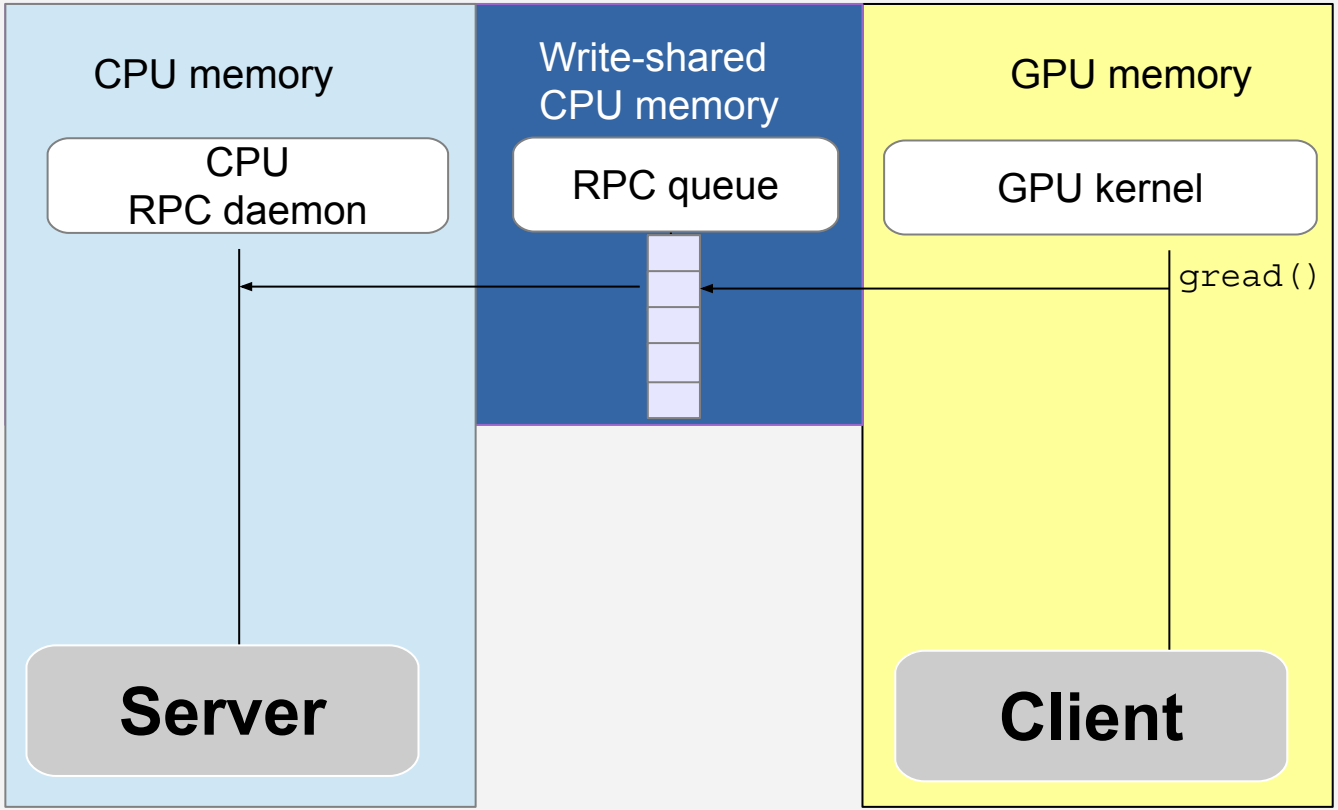
- Session semantics: close-to-open



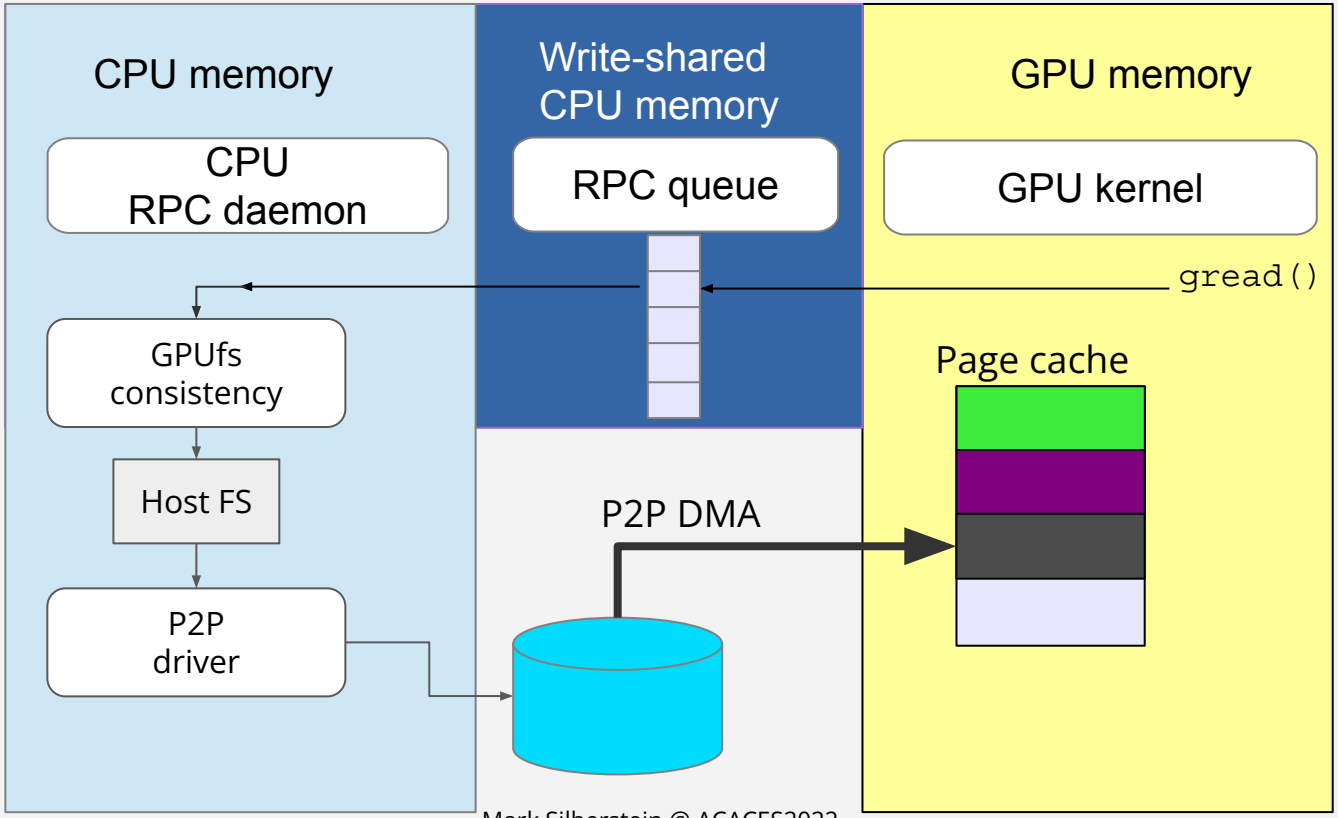
Good enough for the cases when different processors do not work on the same data in parallel

- Consider running a GPU-accelerated `grep` on your file system
- Now consider editing a file on CPU and re-running `grep`
- Will the result include the updated data?

CPU-GPU RPC for on-demand data transfers



Integration with consistency and P2P DMA



Some other things to make it work

- Using GPU-optimized lock-free data structures
  - GPU memory consistency model offers many optimization opportunities
- P2P GPU-SSD data transfers need to guarantee disk-to-page cache data consistency
  - This is unrelated to GPUfs consistency
- Distributed page cache is susceptible to false-sharing, but can be *updated concurrently*
  - Requires multi-way merge

But not everything is great…

- High GPU cost of FS-related operations
- Memory overheads due to internal FS data structures
- Required change in GPU programming optimizations
- Required change in GPU programs to access data

End-result: impactful concept, not really used

Can we do better? Yes (with new hardware)!

- Reminder: memory mapped files

```
int fd=open("file.txt")
char* data=(char*)mmap(fd);
data[2]='a'; // write to file at offset 2
unmap(data);
```

- First access to the mapped pointer causes page fault
- OS brings the file data into the page cache
- Maps the page cache page into the application VM

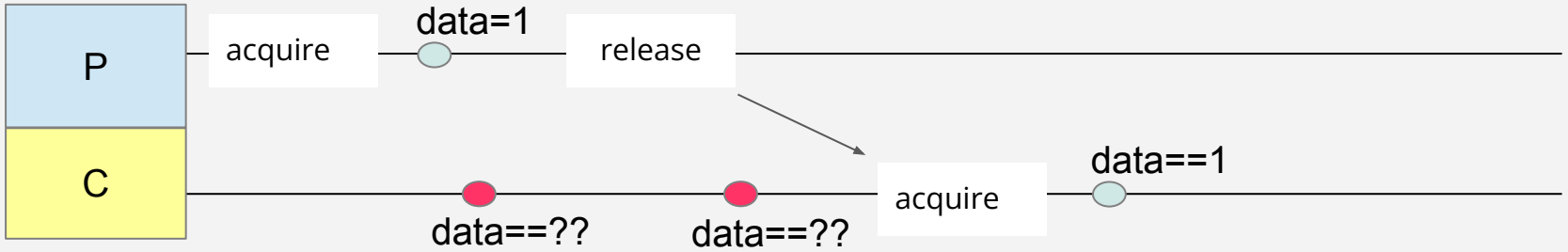
`mmap` a file into a GPU memory

- mmap is issued on a CPU before the kernel is invoked
- Accessing a file on a GPU triggers a **GPU** page fault
- Pros:
  - No file system management overhead on GPU
  - No need to change GPU kernels (supports closed-source)
  - Easy to use

But what if a file is in the CPU page cache?
GPUfs open-close memory consistency is too coarse-grain

Idea: Page cache with release consistency

- Similar to open-close consistency model
- For Producer (P) and Consumer (C), C is guaranteed to observe P's updates if
  - P updates are followed by **release**
  - C reads are preceeded by **acquire** issued after P's **release**



How to use this idea?

CPU code

```
fd=open("file"); // file at offset 0 = "a"
ptr=mmap(...size,MAP_ONGPU,fd);
ptr[1]=1+ptr[0];
macquire(ptr, size, GPU0);
// ensure that GPU0 will see consistent data
invoke_gpu0_kernel(ptr);
wait_for_gpu_completion();
mrelease(ptr, size,GPU0);
// notify that GPU0 does no longer touch ptr
printf("%c\n",ptr[2]);
munmap(ptr); close(fd);
```

GPU code

```
// GPU kernel

void kernel(char* data){
  char var=data[1];
  data[2]=var+1;

}
```

What do we expect to be printed by CPU?
What do we expect to in file at offset 2?

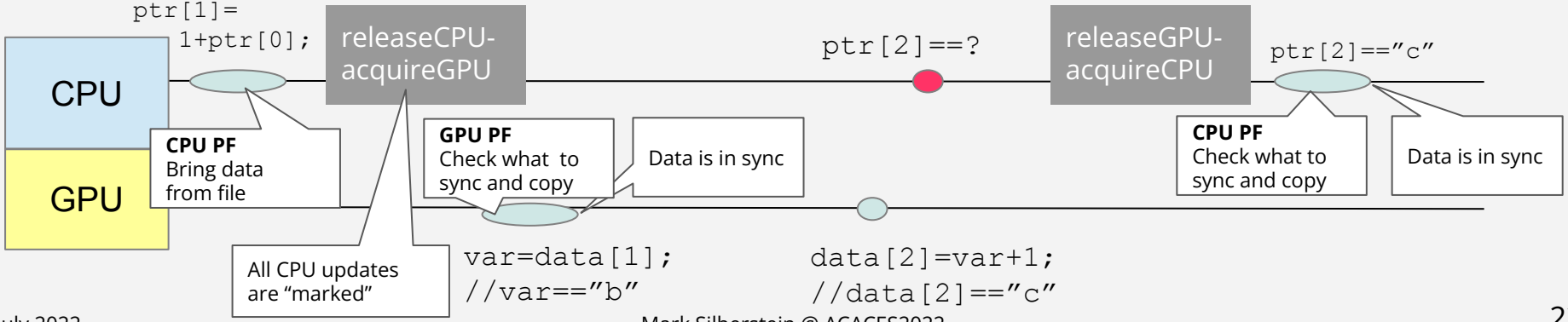
What's going on under the hood?

CPU code

```
fd=open("file"); // file at offset 0 = "a"
ptr=mmap(...size,MAP_ONGPU,fd);
ptr[1]=1+ptr[0];
macquire(ptr, size, GPU0);
// ensure that GPU0 will see consistent data
invoke_gpu0_kernel(ptr);
wait_for_gpu_completion();
mrelease(ptr, size,GPU0);
// notify that GPU0 does no longer touch ptr
printf("%c\n",ptr[2]);
munmap(ptr); close(fd);
```

GPU code

```
// GPU kernel

void kernel(char* data){
  char var=data[1];
  data[2]=var+1;

}
```



Summary

- File system access from accelerators is crucial for data-driven applications
- Distributed page cache is an essential part of the FS support for accelerators
  - *Requires* weak consistency
- FS layer implementation might be costly for accelerators, and not always possible to implement
- The use of page faults in accelerators can alleviate the FS layer overheads, but requires optimizing accelerator PF mechanisms

Additional reading

- GPUfs: Integrating a File System with GPUs, ASPLOS'13, M.S. et al
  - First paper on FS API and Distributed Page Cache for GPUs
- ActivePointers: A Case for Software Address Translation on GPUs, ISCA'16, S Shachar et al
  - Pre-GPU Page fault support for mmap on GPUs
- SPIN: Seamless OS Integration of peer-to-peer DMA between SSDs and GPUs, USENIX ATC'17, S Bergman et al
- GAIA: An OS Page Cache for Heterogeneous Systems, USENIX ATC'19, T Brokhman
  - Shared page cache leveraging GPU page faults

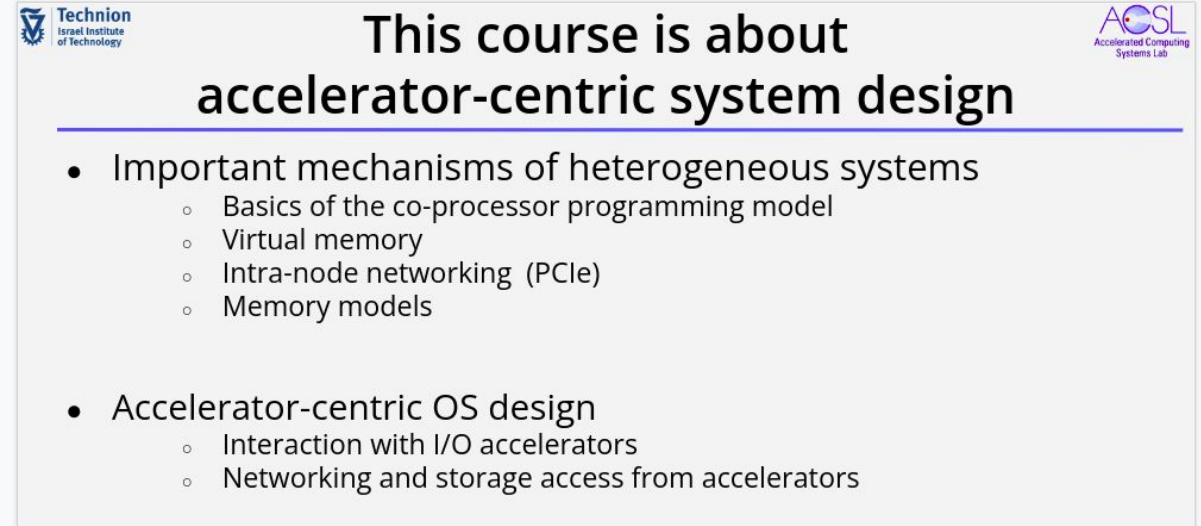
- NVIDIA GPUDirectStorage
- Dragon: Breaking GPU Memory Capacity Limits with direct NVM access, SC'18, P. Markthub et al.
- BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage, Z Qureshi et al.

Questions for self-study [8]

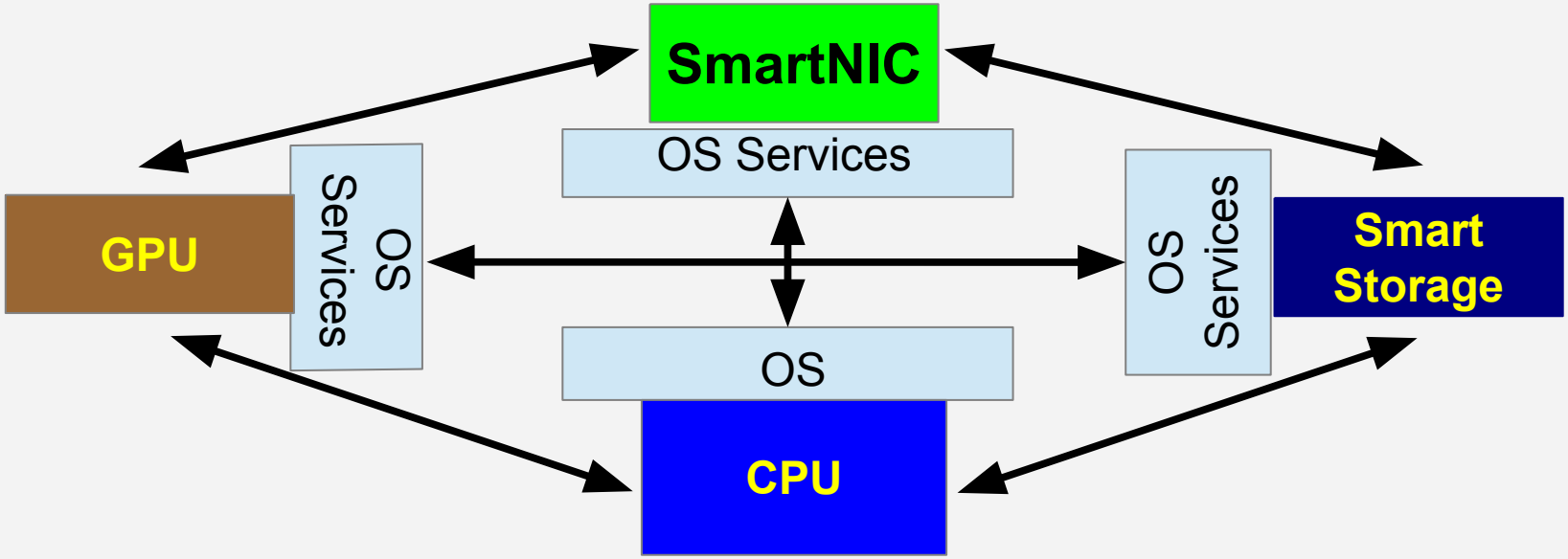
1. Accessing files is usually quite slow and is likely to become a bottleneck anyway. Why then File I/O for accelerators may improve efficiency?
2. What happens if a page cache exceeds its allocated memory, and there is a need to evict pages?
3. How crash consistency is handled?
4. What are the benefits of release consistency compared to open-close semantics of GPUfs?
5. Does release consistency help with false-sharing?

Summary and Future outlook

This is what you learned so far



Reminder: accelerator-centric OS architecture (OmniX)



What are the principles of the accelerator-centric OS design?

Design principles

- Lightweight OS layer for each device
- Direct access to storage and networking I/O from accelerators
- Global shared namespace
  - File names, networking address space
- Shared virtual address space
  - No cache coherence, no location transparency
- Seamless data-path optimizations
- Devices may directly invoke tasks on their peers
  - Recall accelerator-centric server design
- Host CPU used for setup, configuration and scheduling
  - Not in data path, and even not in control path
- Relaxed data consistency where possible

Why I/O abstractions on accelerator

- Data-driven access support
- Portability
- System-level optimizations
- Reduced code complexity
- Unleash performance potential

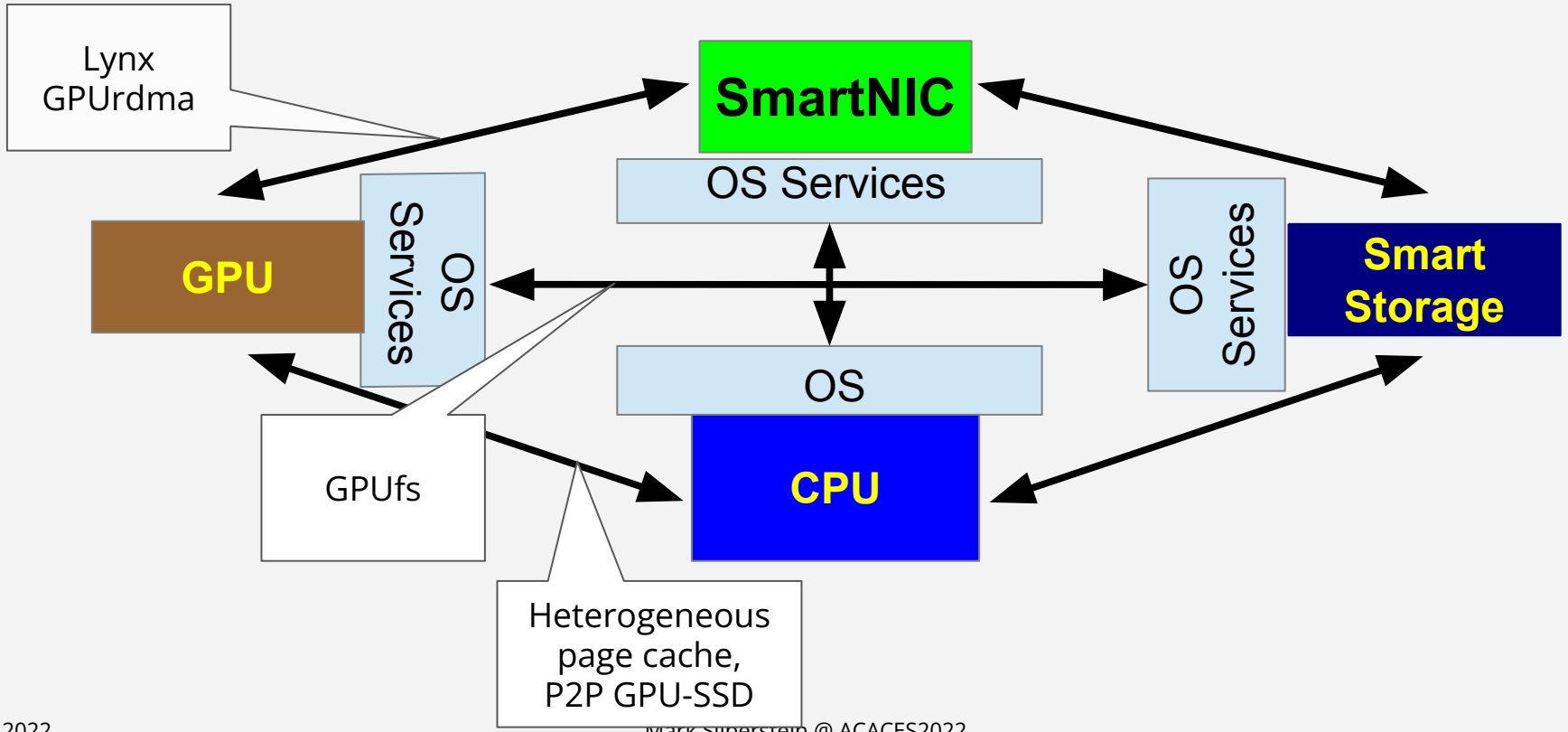
Why CPU mediation is bad?

- Higher latency
- CPU becomes the bottleneck
- Poor scalability
- Poor performance isolation

Main implementation ideas

- Micro-kernel OS approach on the host
  - Only necessary privileged functionality runs on the host
  - Servse as a relay to access priliveged services
- Accelerators run *unprivileged* libOSes to expose services
- Reduce system software overheads for accelerators
  - Offload to I/O accelerators where possible
- Take advantage of PCIe and its unique address routing
- QP/CQ are ubiquitous and easily implementable with weak memory models

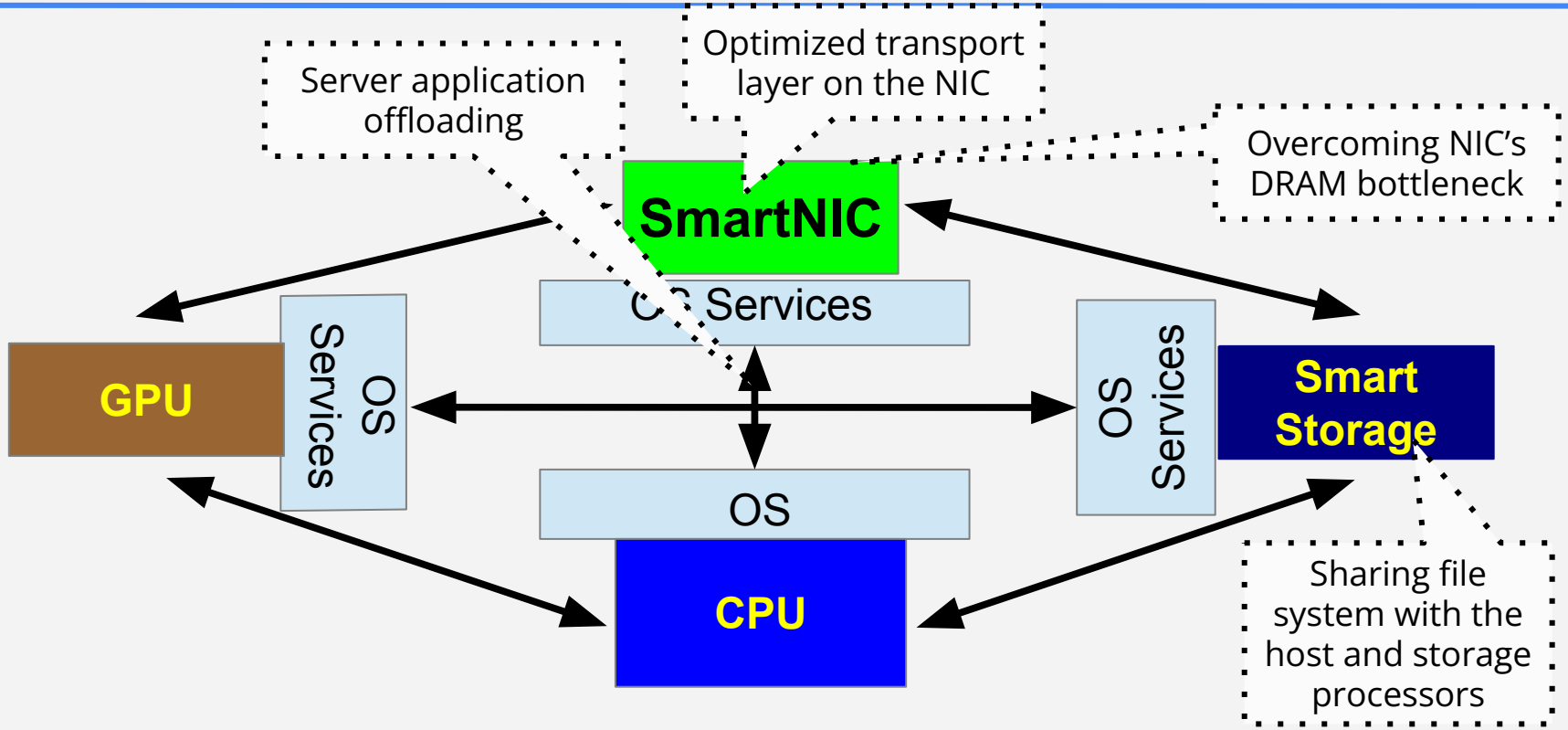
How to fit what we've learned



CPU role

- Not really
- Handle exceptions, first access to resources (files, sockets), cleanup, any privileged operations
- Runs the main program

Quite a few questions are still open…



Open challenges

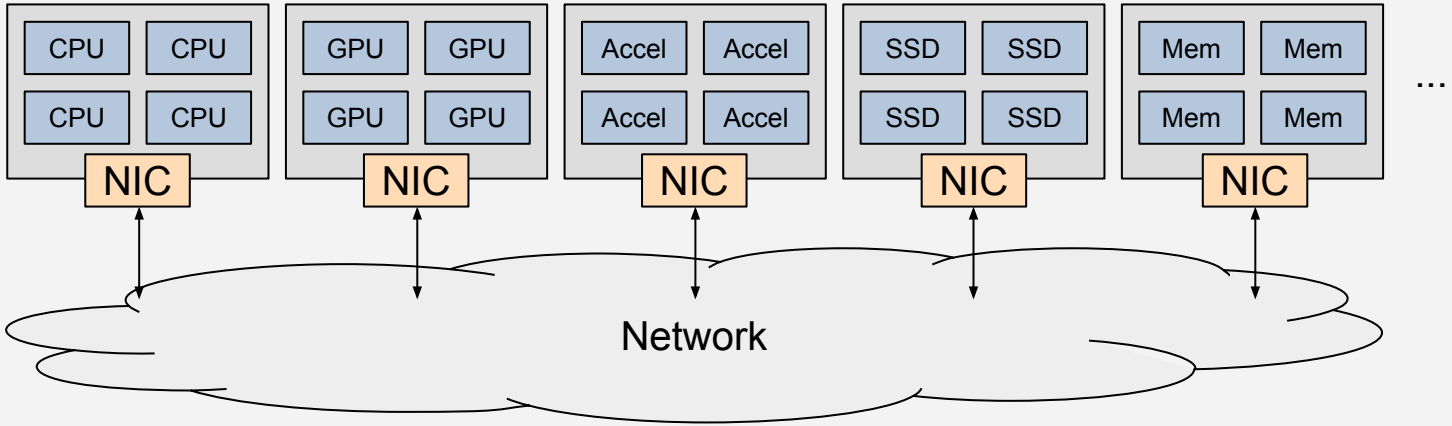
- What is the best programming model
  - Distributed vs. centralized vs. event-driven vs. data flow?
- Efficiency
  - How to get rid of polling? How to leverage heterogeneity to optimize for power at the OS level?
- Multi-tenancy
  - How to schedule and enforce fair-sharing
- Security and confidentiality
  - Integrating Trusted Execution Environments on peripherals
- Incorporating new hardware
  - Persistent memory, Near-memory, In-memory, In-storage computing

But wait… what about data centers?

- Majority of computing today is done in data centers
- Data center is a large-scale computer
- Is accelerator-centric design still relevant? Yes! More than ever

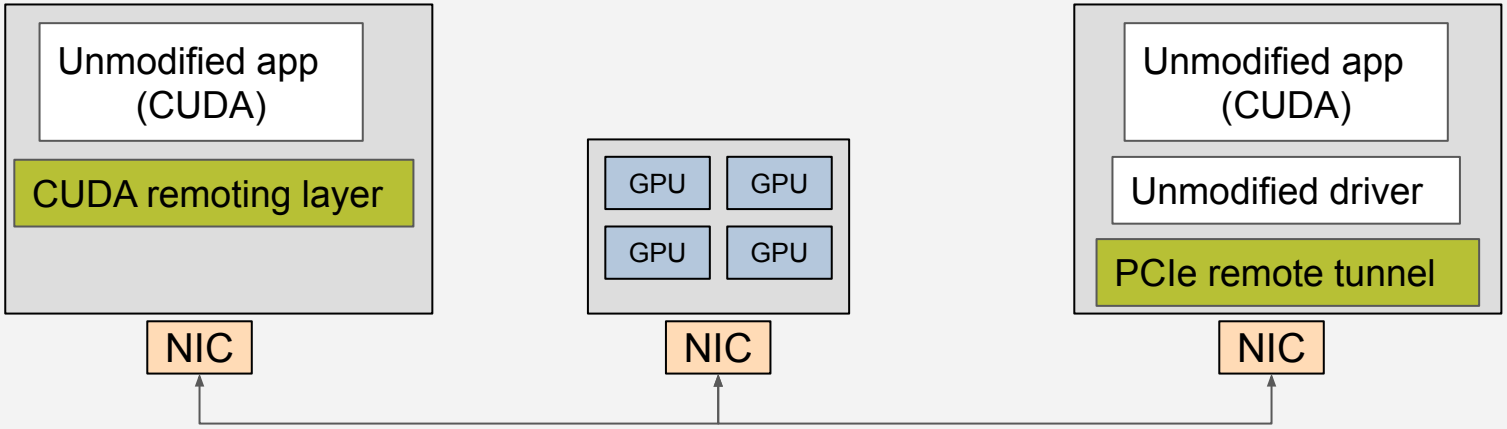
Data Center Architecture Trends

- Resource disaggregation
- High benefits in TCO and utilization, but what about performance?

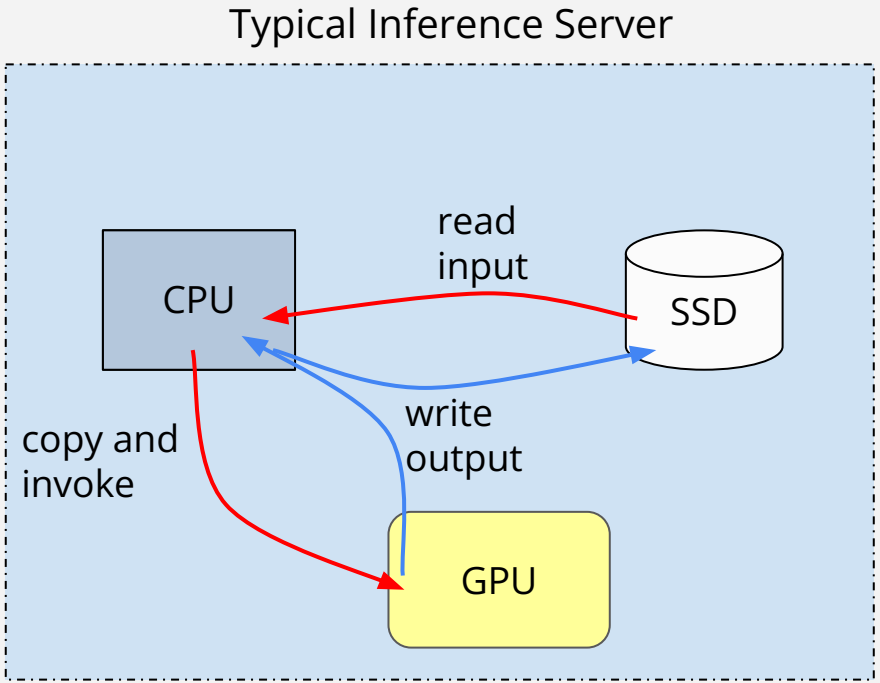


Most common approach: transparent disaggregation

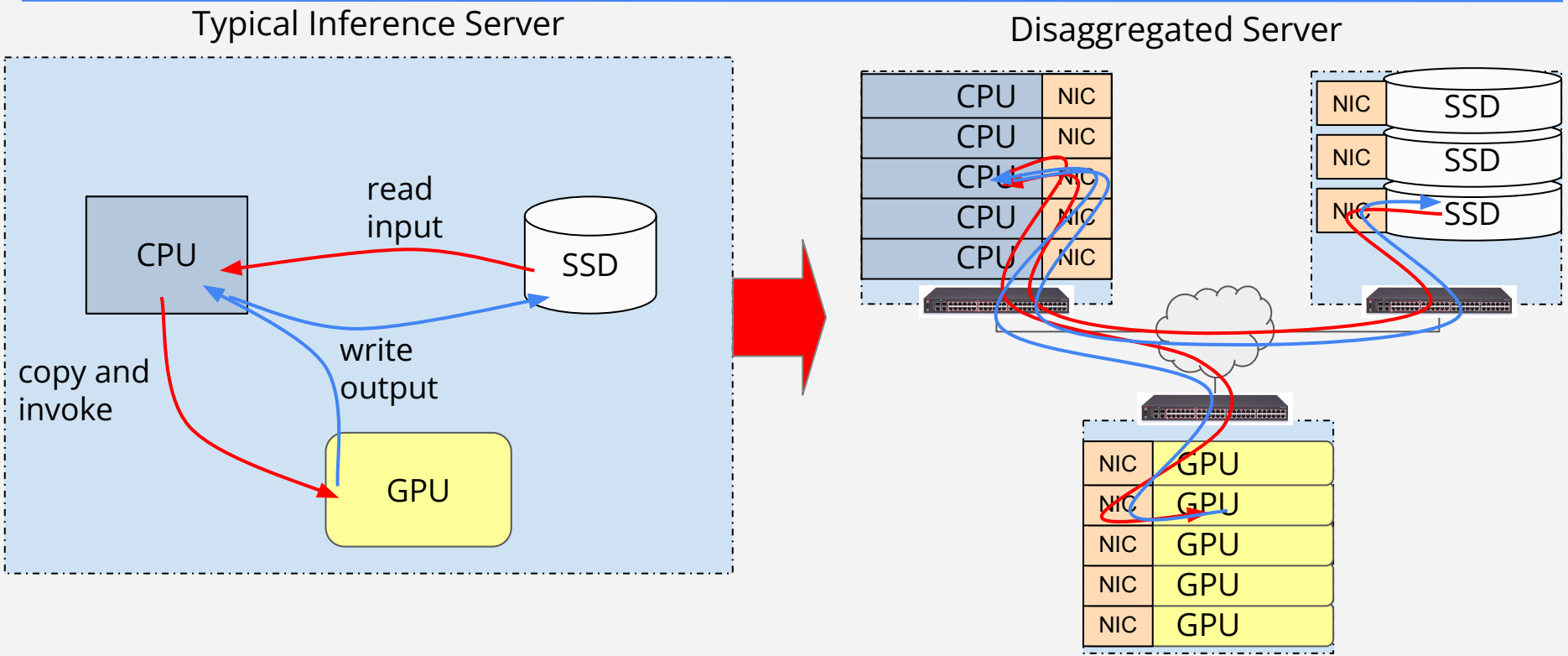
Example: two approaches to remote GPU access



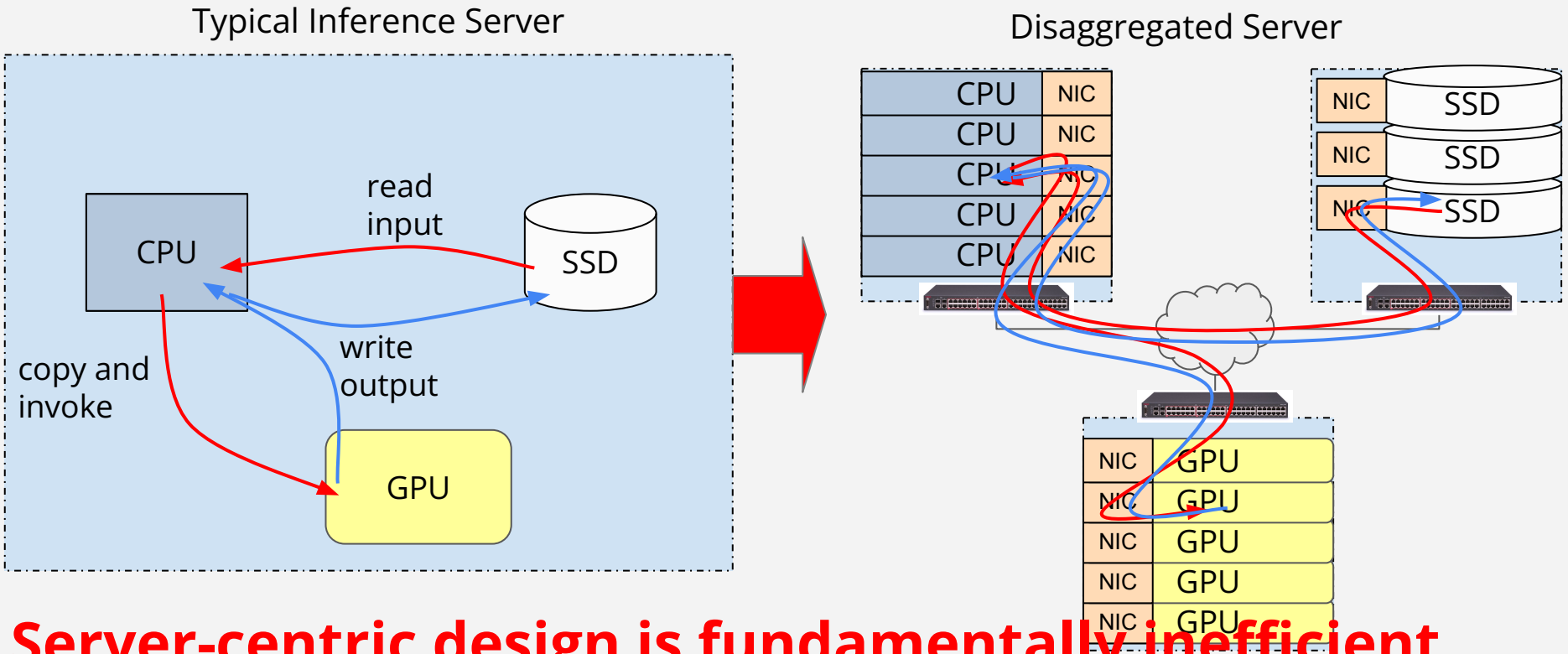
But does it work in a multi-resource setting?



But does it work in a multi-resource setting?



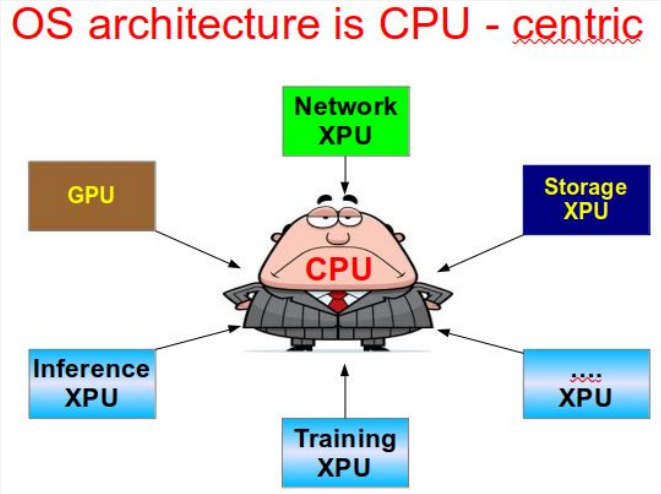
But does it work in a multi-resource setting?



Server-centric design is fundamentally inefficient

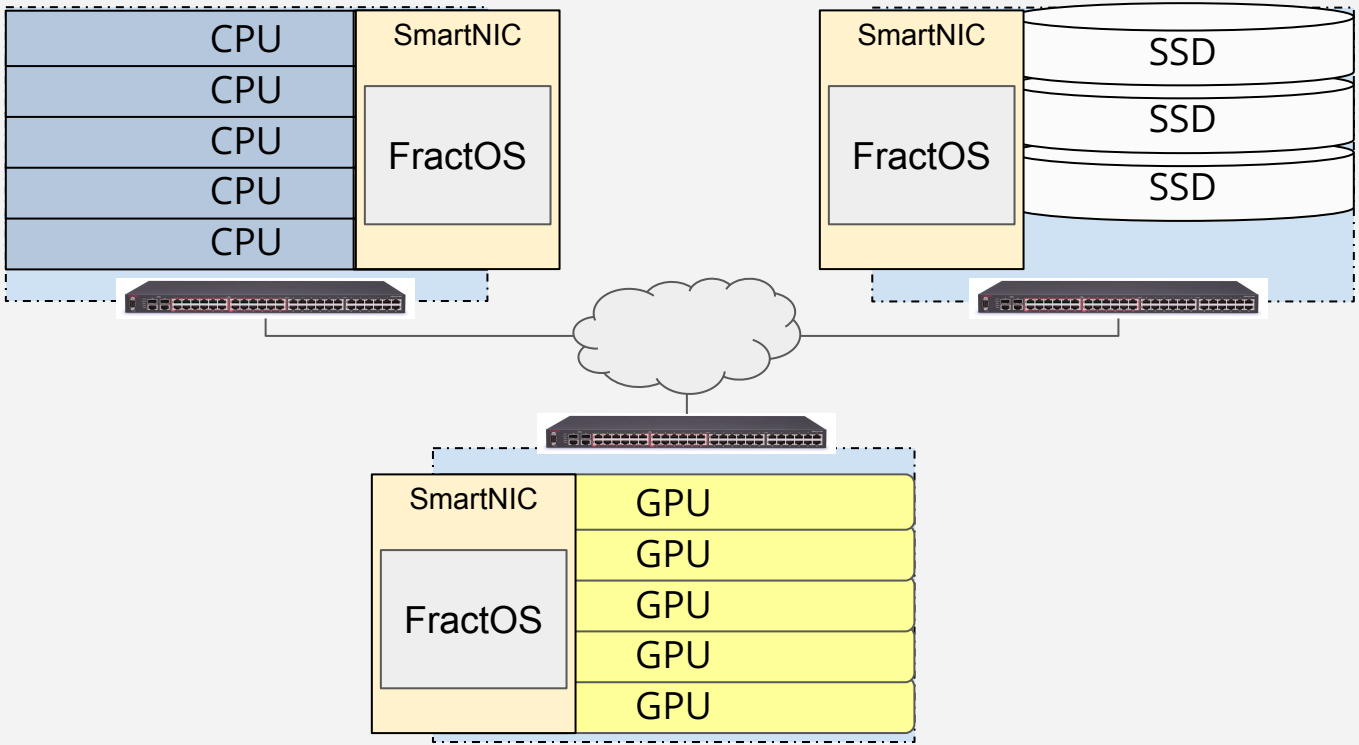
Cons of the server-centric design imposed by transparent disaggregation

- A centralized OS is a control/data bottleneck
- I/O devices and accelerators are *slaves*
- Application control and data planes are centralized
- Sounds familiar?

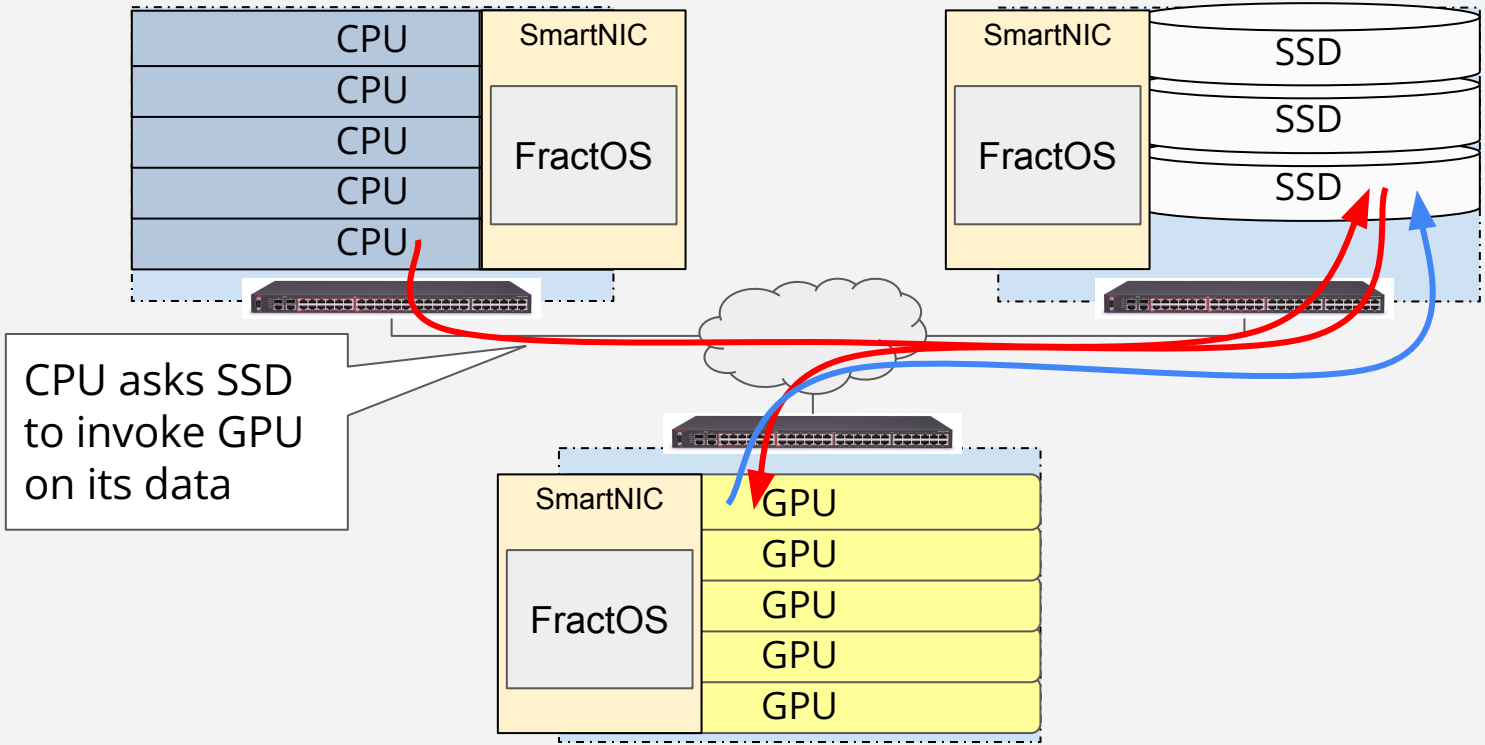


Needed disaggregation-native OS design

FractOS: extending OmniX principles to disaggregation



FractOS enables decentralized execution and direct data/control path among devices



FractOS vs. OmniX

- Avoid CPU in data/control path
- Devices as first-class citizens
- Direct interaction among devices
- Transparent data-path optimizations

OmniX

- Decentralized capability management
- Decentralized task graph execution
- Unified software/hardware interfaces

Additional reading

- Slashing the disaggregation tax in heterogeneous data centers with FractOS, EuroSys 2022, L Vilanova et al

- LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation, OSDI'18, Y. Shan et al.
- Network Requirements for Resource Disaggregation, OSDI'16, PX Gao et al.
- HeteroOS: OS design for heterogeneous memory management in Datacenter, ISCA'17, S. Kannan et al.
- rCUDA https://www.rcuda.net
- SmartIO: Zero-overhead Device Sharing through PCIe Networking, ACM TOCS'2020, J Markussen et al.
- AvA: Accelerated Virtualization of Accelerators, ASPLOS'20, H Yu et al

A few words about ACSL@Technion

https://acsl.group

ACSL group circa 2020
(COVID-inspired group meeting)

Looking forward to reading your papers about accelerator-centric systems!

Thank you!

Answers to questions for self-study

[1]

1. One accelerator can only accelerate the amount of work proportional to the acceleratable part α. Higher speedups require high α, which is often unrealistic. Multiple accelerators may *together* be able to accelerate a large part of the application.
2. Yes. Full programmability implies that a processor must spend resources on transforming instructions into actions and doing so in an efficient way. Further, many CPUs suffer from the von Neumann bottleneck. A fixed-function ASIC has no such issues.
3. A look-aside accelerator is explicitly invoked by some controlling unit. It is fed with data and is more similar to a remote machine that is requested to perform certain operations by Remote Procedure Calls. An inline accelerator is implicitly invoked when data are passing through it (hence *inline*). It does not have any explicit invocation controlling unit, and determines which data to process and how. Such accelerators are driven by data elements, and are located “on the way” of data transfers.
4. A look-aside accelerator involves control and data movement overheads, which need be amortized by increasing the granularity of the invoked tasks, i.e., via batching.

[2]

1. Accelerators are used as co-processors. This would have been OK for fixed-function units. However for programmable accelerators there are cases where a user program needs access I/O or invoke tasks on other devices. Accelerators today are isolated from the rest of the system from the programmer perspective, and they do not expose appropriate OS abstractions. Therefore, developers have to develop complex programs on a central CPU to perform accelerator management and data transfers.
2. The term “OS” is traditionally associated with monolithic OSes such as Linux and Windows. However, in a more general context, an OS is a group of inter-operating services that offer simple programming abstractions that hide system complexity, and perform seamless performance optimizations for applications. From that perspective, an OS can be a runtime library that carries out such functions.
3. GPU kernels must be highly-parallel, and must achieve high memory access efficiency.
4. JIT-compiling a kernel allows GPU vendors to modify hardware architecture quite dramatically in newer generations, while maintaining backward compatibility with the existing software. The compiler generates an intermediate assembly, for which the backward compatibility is mostly guaranteed. The JIT-compiled kernel is cached for later reuse, i.e., the JIT-compilation is performed only once for a given GPU.
5. A CPU fully manages the device: it allocates memory, copies data to and from the GPU, configures and controls the GPU execution, handles exceptions and cleans up.

[3]

1. Accelerator's access to CPU memory does not pass through the CPU MMU, therefore the CPU virtual page originally mapped into the accelerator VM could have been swapped out and the physical memory content would be inconsistent. Pinning the page on the CPU prevents the page from being swapped out.
2. In general, the answer is yes – peripheral devices may access CPU memory directly making the CPU VM protection irrelevant. However, at a system level, memory mappings are performed by the accelerator's driver, which is a trusted and privileged software component. The driver's responsibility is to validate that a particular application has permissions to access the CPU memory region being mapped. But rogue drivers or rogue devices may breach system security severely (see DMA attacks)
3. No: a GPU memory pointer might be perfectly valid on a CPU if, coincidentally, a GPU had allocated virtual memory addresses that happened to be also allocated by a CPU. But they refer to a different buffer and thus such accesses would be to a wrong data.
4. This is the scenario described in detail in the false-sharing case. The same page is being accessed by multiple processors and thus being constantly migrated. Worse, due to the different memory page size in different processors, the migration is performed at the granularity of the largest page among all processors. Thus, an application might not even be aware that it suffers from false sharing.

[4]

1. Stronger memory models are potentially more costly (performance-wise) to use because they disallow or limit optimizations in the cases where strong ordering is not necessary, leading to unintended (and hard to debug) performance degradation. On the other hand, weaker models are harder to use (and so the chance of complex correctness bugs is higher) but can be much more efficient.
2. No. When the processor already implements a stronger consistency model (i.e., TSO in x86), some orderings are enforced by hardware, hence fences are replaced with NOPs by a (smart) compiler if they are deemed unnecessary for the specific processor.
3. Memory model expresses a contract between a developer and a system. If a developer develops for hardware directly (assembly) then she writes for the model defined for that hardware. Using C++11 and later versions allows a developer to build *portable* programs assuming memory consistency defined by the C++11 spec. It is the job of the compiler to optimize a portable program for a particular hardware memory model. Java has a different memory model.
4. Scopes enable more fine-grain programmer control over ordering-related overheads.
5. No. Prints can be reordered/delayed in the output buffer by the OS, so let's consider two scenarios
   a. if b is printed first and is 1, then T1 read b==1. t2lock is 1 due to the program order of T2. Due to SC this order is guaranteed globally. Thus T1 will observe t2lock==1 and must read a== 0.
   b. If a is printed first and is 1, then t2lock was 0 according to T1 when b was loaded by T1. But since T1 observes writes to t2lock and to b in the same order as T2, then if t2lock is 0, b must have been 0 too. So T1 would read b==0

[5]

1. Writes are posted (non-blocking) and reads are non-posted (blocking). But how can a device know that its write is complete? It must read from where it wrote, and since reads are not reordered with the preceding writes, such reads must return the written values when the write has been actually completed.
2. MMIO involves software execution a CPU (or any other programmable device). For example, to transfer a buffer from a CPU to a GPU via MMIO, the source data is loaded via the CPU load instruction into a CPU register from CPU memory, and then stored via the CPU store instruction in the GPU on its BAR. Thus, performing bulk transfers using MMIO is inefficient.
3. No, a PCIe switch is connected (potentially through other switches) to the CPU Root Complex, so the devices interact directly. This is the case, for example, for dual-GPUs such as NVIDIA K80.
4. They do not interfere on the PCIe, because the CPU has effectively a point-to-point PCIe connection to each device via its Root Complex. However, CPU DRAM my become a bottleneck.
5. NIC initiates the transfer hence NIC's DMA is programmed to write into GPU's BAR
6. The behavior is undefined and depends on how a device implements reads from a particular address on its BAR. The fact that the CPU reads (and not writes) does not mean that there will be no side-effects from this operation. For example, an incorrect read might reset a device or cause malfunction.

[6]

1. r_key is passed to the NIC by the remote party to allow access to a particular memory region. This r_key is unique, and must match the one used when registering the memory region to the NIC locally. Note that r_key is passed in the clear (not encrypted), hence this protocol is insecure if the network is untrusted.
2. HCA needs to access registered memory regions, which might comprise multiple physically non-contiguous pages. The memory registration mechanism updates internal translation tables on the NIC so that the NIC can find the physical pages upon remote access.
3. RDMA memory ordering semantics are defined by the standard. In particular, writes to the same QP cannot be reordered.
4. Yes, QP is a logical communication end-point similar to socket, hence there are no restrictions on the number of QPs to use and how they are used. However, each QP consumes resources on the host and on the NIC, and these must be minimized to ensure high performance.
5. The OS kernel driver is a privileged software that ensures that an application registers only memory it has access to. The driver mediates all control-plane operations.

[7]

1. BF can serve as a management accelerator because it supports the mode of operation where it fully controls the host's network I/O, and the host cannot reset or interfere with BF's operations
2. GPUrdma requires the GPU to create and issue RDMA VERBs, and the GPU must perform PCIe writes to the NIC's doorbell register. PCIe writes, despite being asynchronous, incur high latency when performed from the GPU code. The SmartNIC-driven design has no such limitations as it allows the accelerator to perform I/O by maintaining a local doorbell register polled by the SmartNIC, and also allows to optimize the structure of the local QPs in accelerator memory to make it convenient for the accelerator to access.
3. As PCIe RC is located on the ARM processor, the code running on the SmartNIC cannot access the host's PCIe bus directly, neither it can directly access host's DRAM. As a result, the current design requires RDMA transactions to access host's memory from the SmartNIC, which in turn puts unnecessary load on the ASIC NIC and internal buses. This limitation will likely be alleviated in the future versions of the SmartNIC (via non-transparent PCIe bridge)
4. First, all the data transferred to/from the accelerator must be put/read from the ARM's DRAM. The memory subsystem may quickly become the bottleneck. Second, the ARM processor is quite weak, so if the application needs to perform pre-processing on the data before invoking the accelerator, then ARM would become the bottleneck. This is why SmartNICs are packed with accelerators that can be used by programs running on the ARM CPU.
5. No. Acc DMA is not used because the data is passed to the Acc by RDMA accesses on the Acc PCIe BAR.

[8]

1. Indeed, for certain applications which are bottlenecked by storage bandwidth or are purely compute-bound, there is little to no performance benefits when using accelerator-native FIle I/O. Still, one major benefit is that it is easier to use, as writing accelerator programs, while having the CPU managing their I/O requests, forces changing a natural program flow. Second, certain files might still reside in the CPU page cache, so accessing them would copy their contents from CPU memory. Third, the native File I/O implementation can be explicitly optimized for accelerators, i.e., by issuing multiple I/O requests in parallel, via prefetcher, and by implementing peer-to-peer DMA. Last, multi-accelerator case requires dealing with synchronizing the file caches, which is better be done by the OS infrastructure rather than a programmer.
2. Page cache eviction requires finding the right candidate page to evict. Traditional recency-based algorithms might not be applicable, as they require running a background process (such as pdflush in Linux) that keeps track of the pages accesses and evicts less used pages in the background. Using CPU to perform page eviction on accelerators might not be possible due to the need to lock a page when evicting it. This is not possible today because of PCIe limitations. GPUfs introduced a mechanism where a GPU selects the pages for eviction following “least recently allocated” pages, and then lets the CPU migrate them to the disk.
3. File data updated and stored in the accelerator page cache might not be persisted if the accelerator program crashes. This problem has no simple solution in GPUs.
4. Release consistency gives the programmer fine-grain control of the file synchronization among the system processors, without coupling synchronization and file operations (open/close).
5. Release consistency entirely eliminates the false sharing-induced pingpong of a page observed in a single-owner model. That is because each processor can concurrently update the page in its own memory, thus creating multiple versions of that page. The page is then merged at the synchronization time. See more about 3-way merge in the paper on Heterogeneous Page Cache from ATC’19.