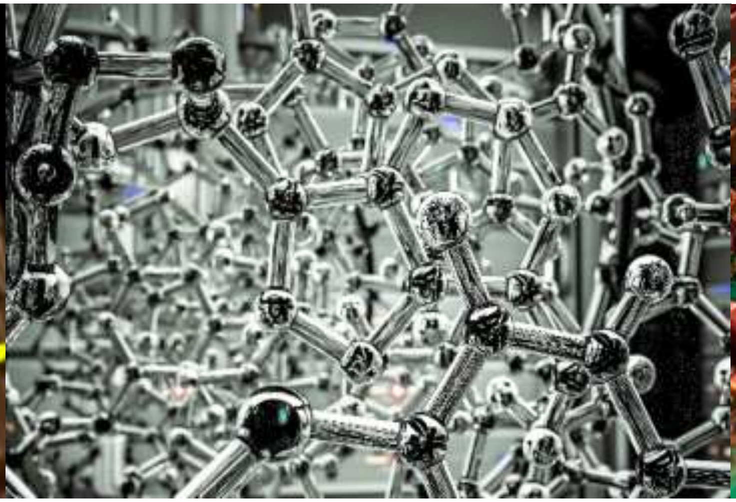
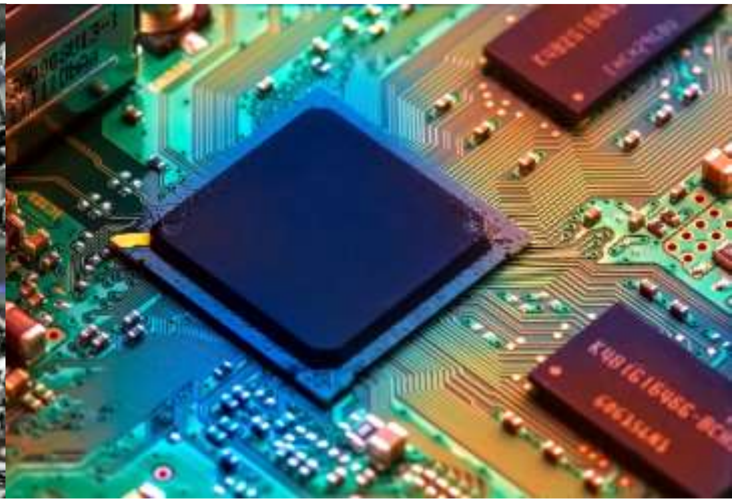
Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects

Carl Pearson[1], Abdul Dakkak[1], Sarah Hashash[1], Cheng Li[1], I-Hsin Chung[2], Jinjun Xiong[2], Wen-Mei Hwu[1]

[1] University of Illinois Urbana-Champaign, Urbana, IL

[2] IBM T. J. Watson Research, Yorktown Heights, NY



ILLINOIS
Electrical & Computer Engineering
COLLEGE OF ENGINEERING

Motivation

CUDA data transfer bandwidth depends on allocation and transfer method → **"comprehensive coverage"**

Microbenchmarks for CUDA communication methods

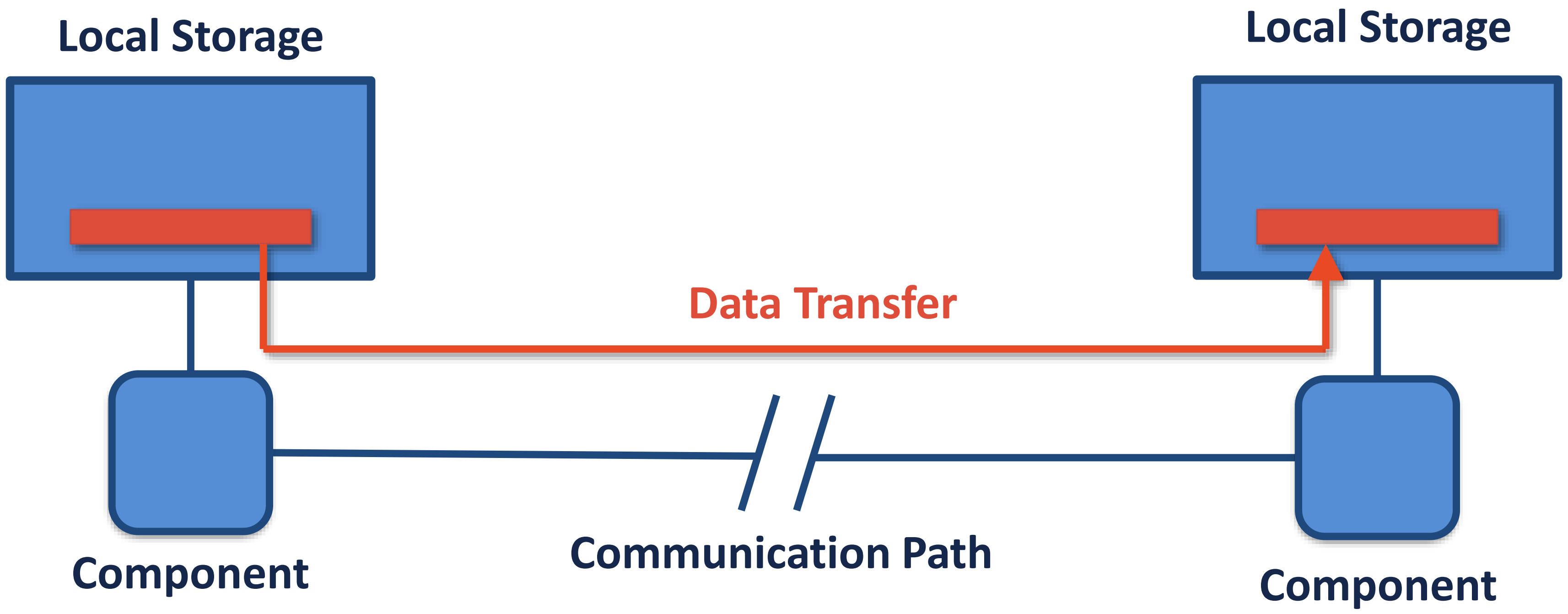
Observed substantial variability in initial measurements → **"common pitfalls"**

Control non-CUDA system parameters during measurements

Avoid synchronization overhead from measurements

Insights about high-performance interconnects?

Comprehensive Coverage of CUDA Bulk Transfers

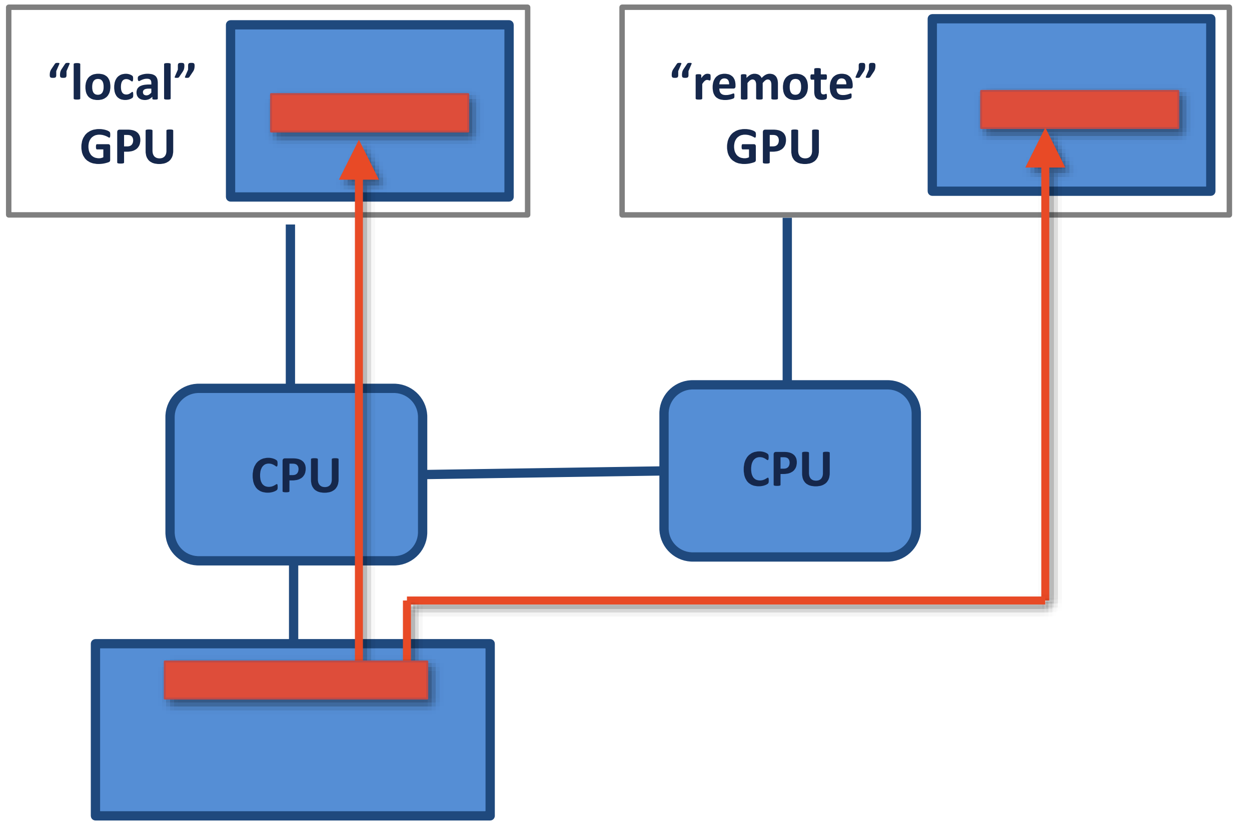


- Unidirectional Operations
- Bidirectional Operations
- NUMA Pinning
- Peer Access
- “Zero-Copy”
- Unified Memory

Non-CUDA Parameter: NUMA Pinning

- Not all cudaMemcpy created equal on high-bandwidth interconnects

Configuration (Limiter)	Theoretical (GB/s)	Observed (GB/s)
AC922 Local (3x NVLink 2)	75	66.6 ± 0.013
AC922 Remote (X-bus)	64	41.3 ± 0.009
S822LC Local (2x NVLink 1)	40	31.9 ± 0.008
S822LC Remote (x-bus)	38.4	29.3 ± 0.013
4029GP Local (PCIe 3)	15.8	12.4 ± 0.0002
4029GP Remote (PCIe 3)	15.8	12.4 ± 0.0002



1GB pinned host allocation transferred to GPU

Non-CUDA Parameters

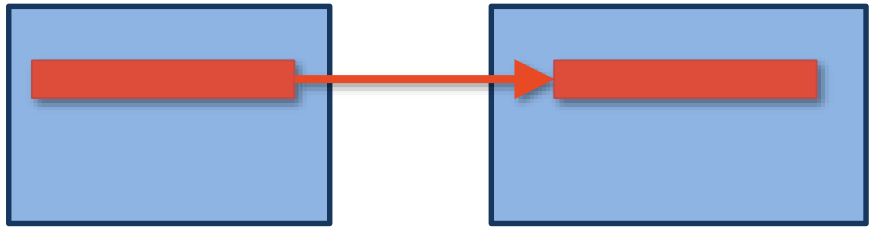
- Variable CPU Clock Speeds
  - cpupower frequency-set --governor performance
- CPU Data Caching

```
// arch/x86/include/asm/special_insns.h

void flush(void *p) {
  asm volatile("clflush %0"
               : "+m"(p)
               : // no inputs
               : // no clobbers
  );
}
```

```
// linux/arch/powerpc/include/asm/cache.h

void flush(void *p) {
  asm volatile("dcbf 0, %0"
               : // no outputs
               : "r"(p)
               : "memory”
  );
}
```

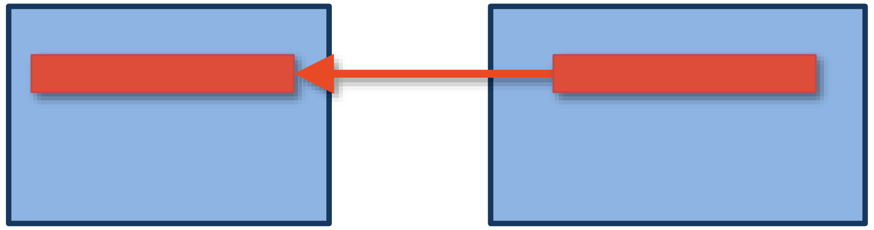
Pinned Allocation and cudaMemcpy

- GPU does DMA to access pinned data on CPU



CPU DRAM GPU Memory

cudaMemcpy(… , cudaMemcpyHostToDevice)

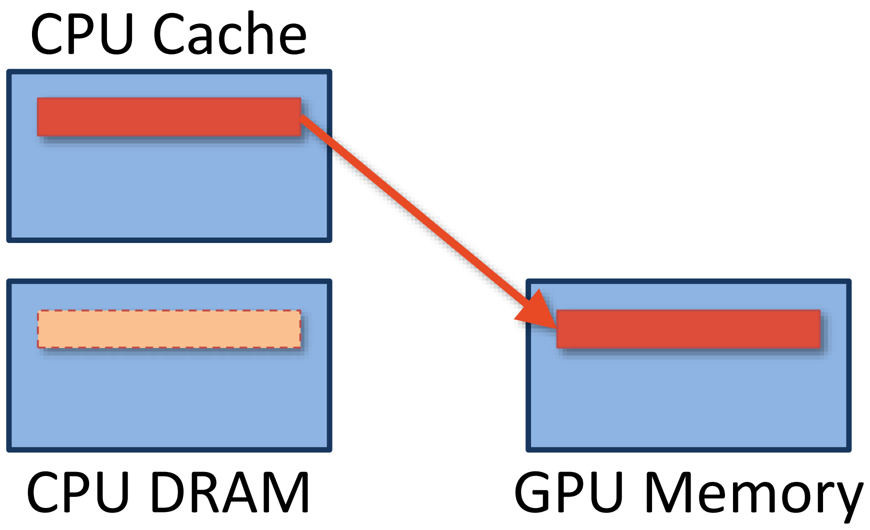


CPU DRAM GPU Memory

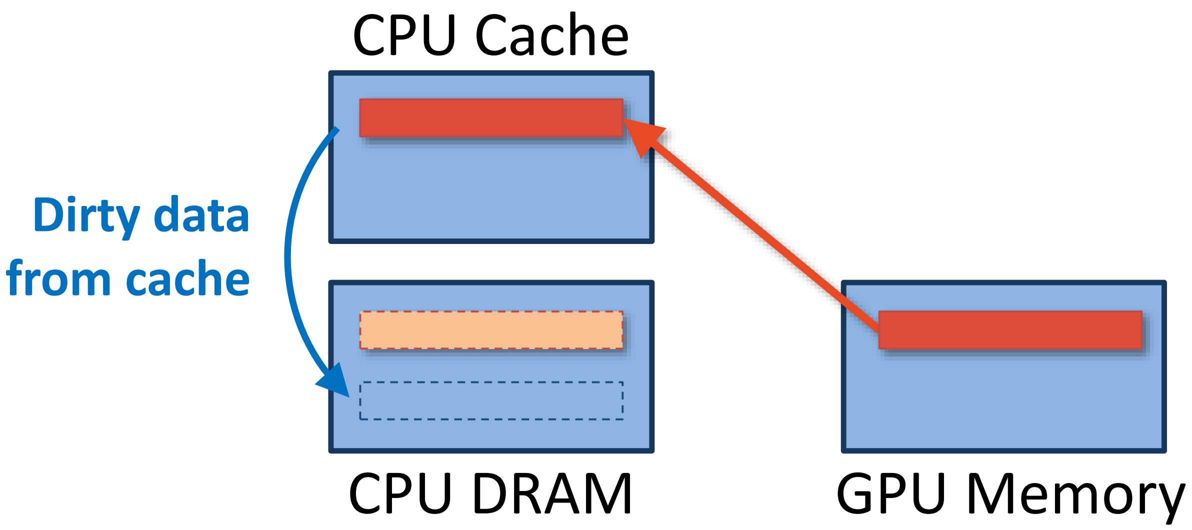
cudaMemcpy(… , cudaMemcpyDeviceToHost)

cudaMemcpy & CPU Cache

- CPU writes values to initialize data
- For small allocations, data may reside entirely in cache



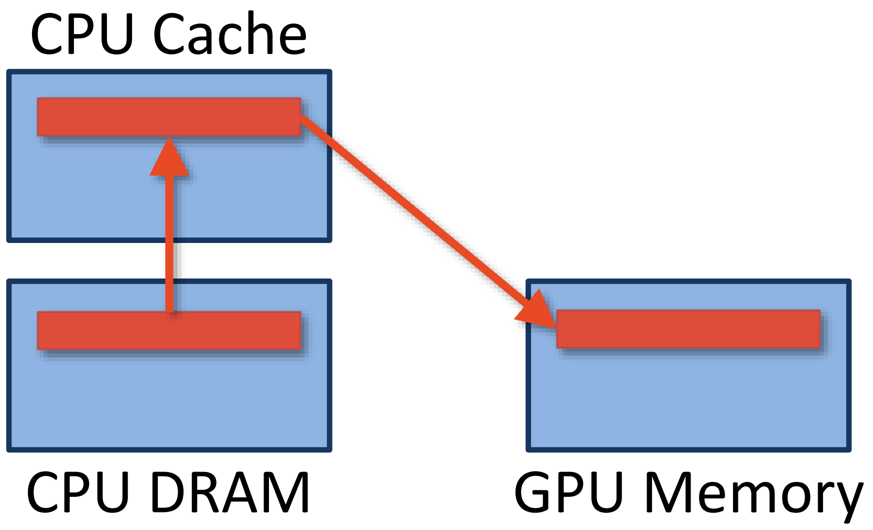
cudaMemcpy(… , cudaMemcpy<u>HostToDevice</u>)



cudaMemcpy(… , cudaMemcpy<u>DeviceToHost</u>)

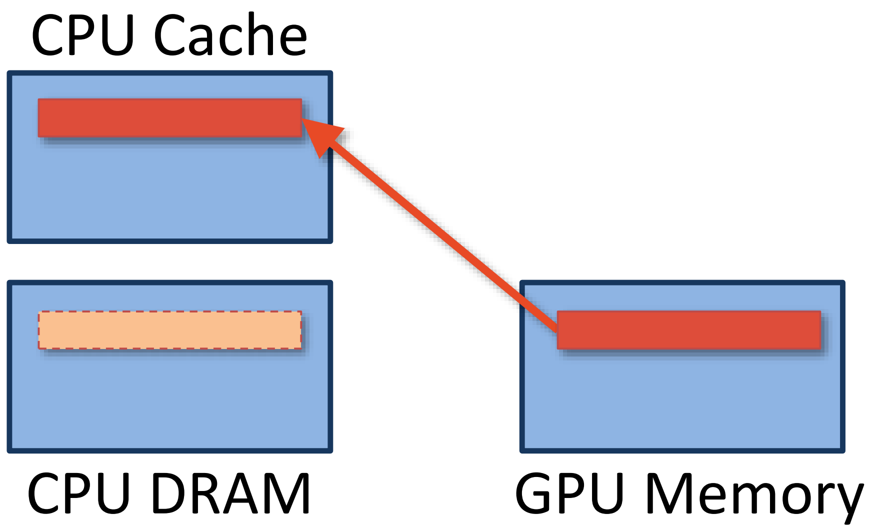
cudaMemcpy & CPU Cache

- Flushing the cache forces data to start in the DRAM

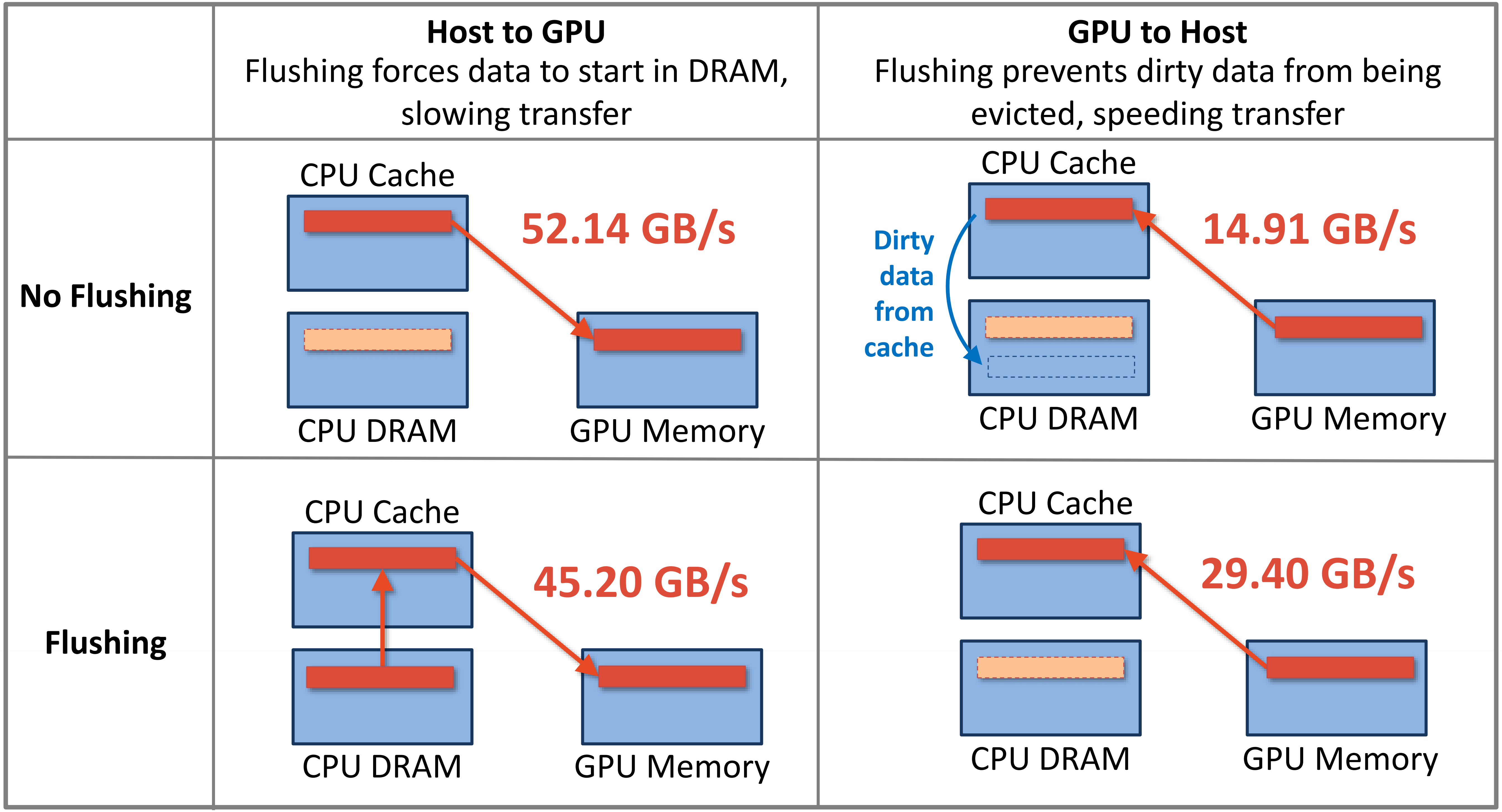


cudaMemcpy(… , cudaMemcpy<u>HostToDevice</u>)

- Flushing the cache prevents write-back of dirty data

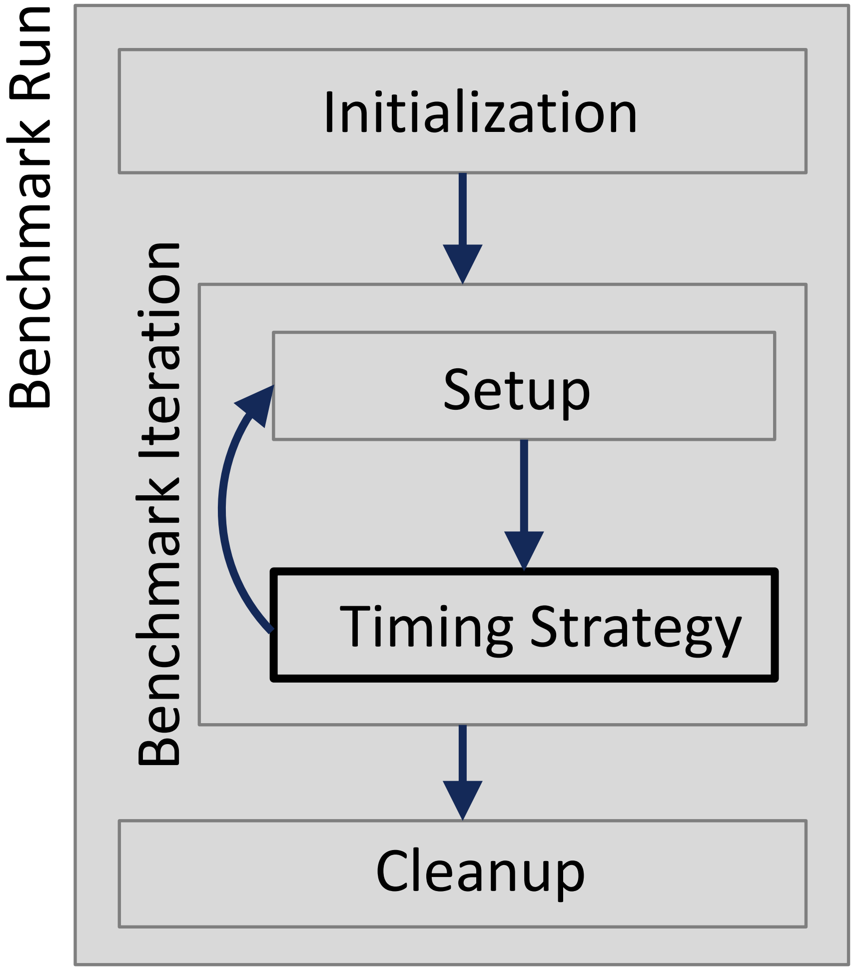


cudaMemcpy(… , cudaMemcpy<u>DeviceToHost</u>)


Host to GPU
Flushing forces data to start in DRAM, slowing transfer
GPU to Host
Flushing prevents dirty data from being evicted, speeding transfer
No Flushing
CPU Cache
CPU DRAM
GPU Memory
52.14 GB/s
CPU Cache
Dirty data from cache
CPU DRAM
GPU Memory
14.91 GB/s
Flushing
CPU Cache
CPU DRAM
GPU Memory
45.20 GB/s
CPU Cache
CPU DRAM
GPU Memory
29.40 GB/s

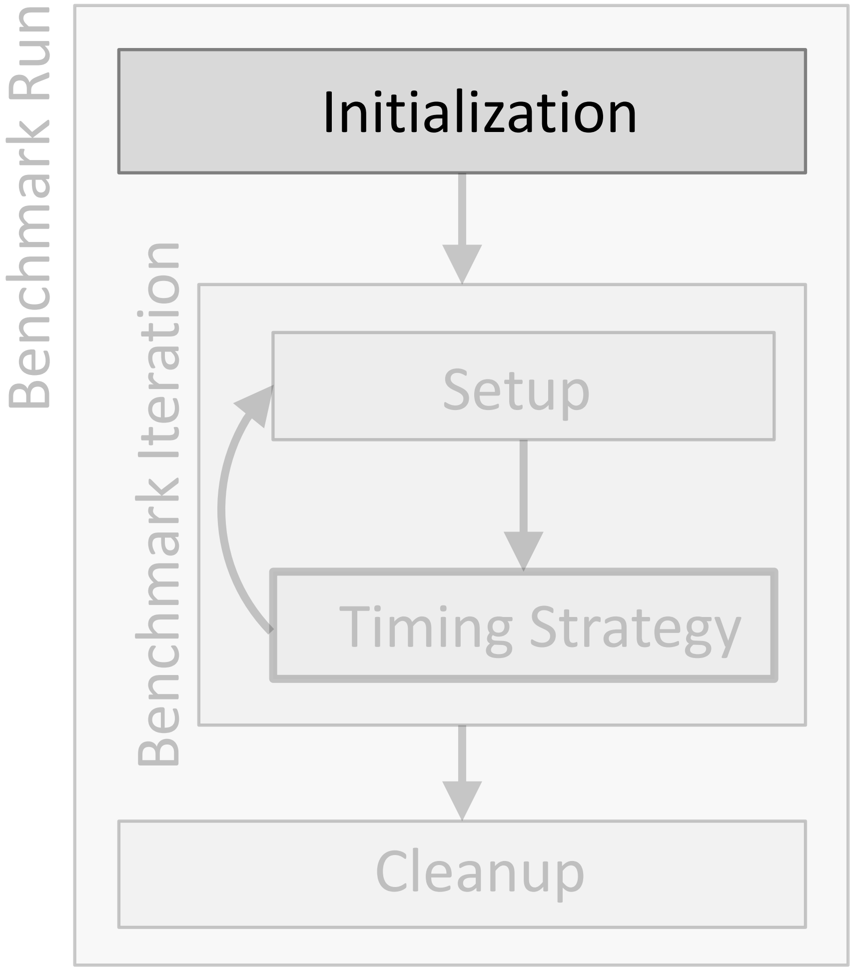
Benchmark Design

- Using Google Benchmark Support Library
  - Each benchmark run consists of some number of iterations
  - The number of iterations is
    1 < n < 1e9 and
    total time under measurement >= 0.5s
- Support synchronous and asynchronous operations
- Report variability across runs



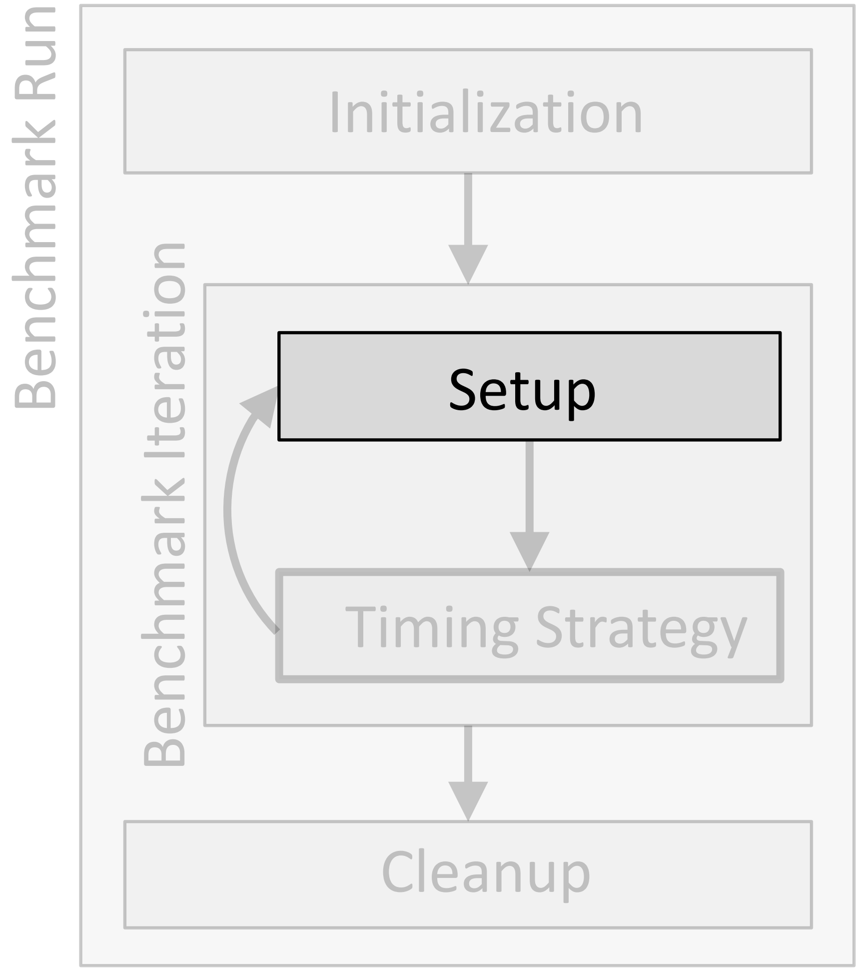
Initialization (as needed)

- Resetting CUDA devices
- NUMA pinning
- Creating allocations
- Creating CUDA streams and events
- Zeroing allocations
- Configure CUDA device peer access



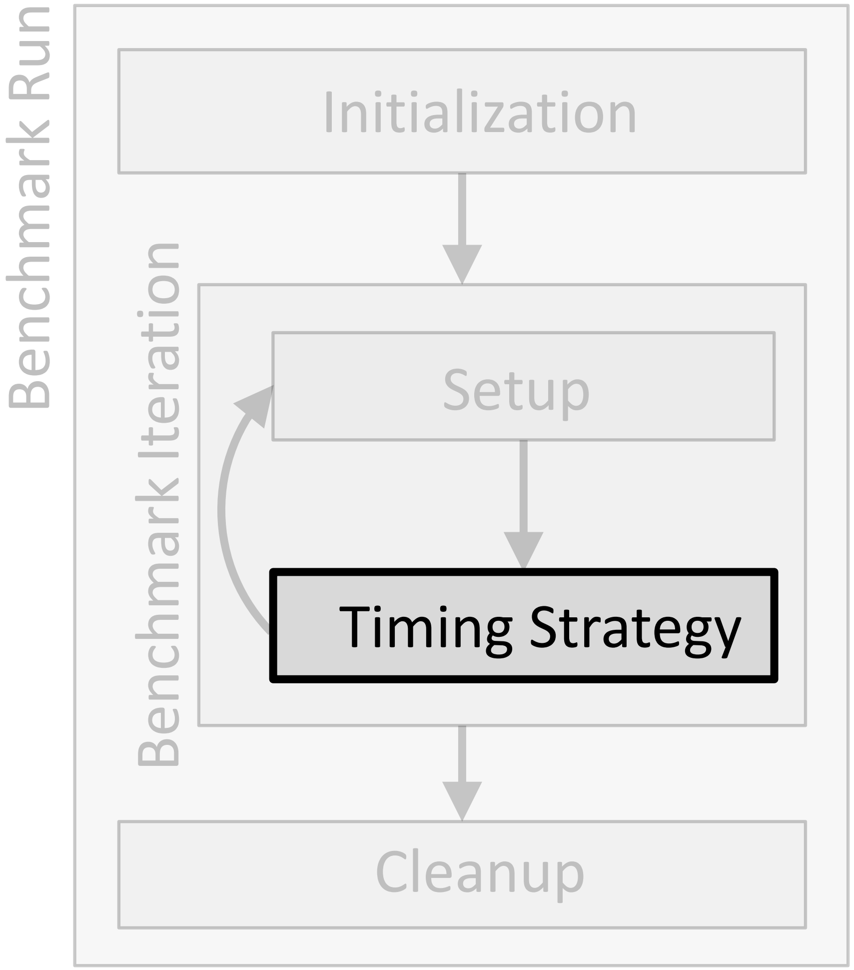
Setup (as needed)

- Moving unified-memory data to a source device
- Flushing caches
- Setting CUDA devices
- Adjusting NUMA pinning



Timing Strategies

- Timing the data transfer operation
- Different approaches for different transfer types:
  - Synchronous
  - Asynchronous
  - Simultaneous



Asynchronous Operations

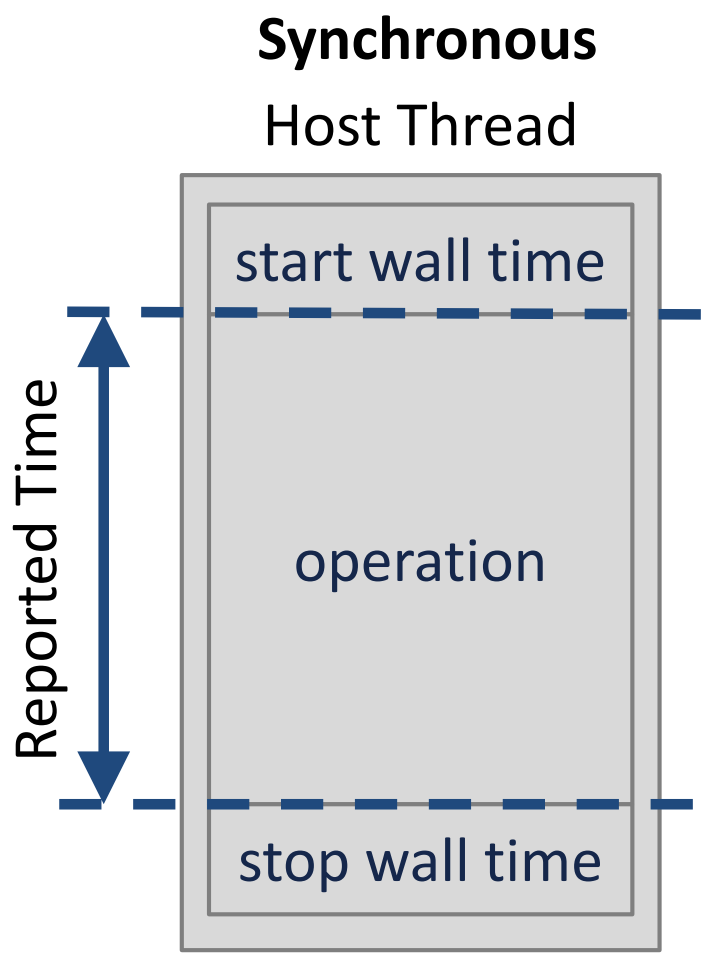
- An operation that may complete at any time (from the perspective of the host)
- CUDA API call may return before the operation is complete

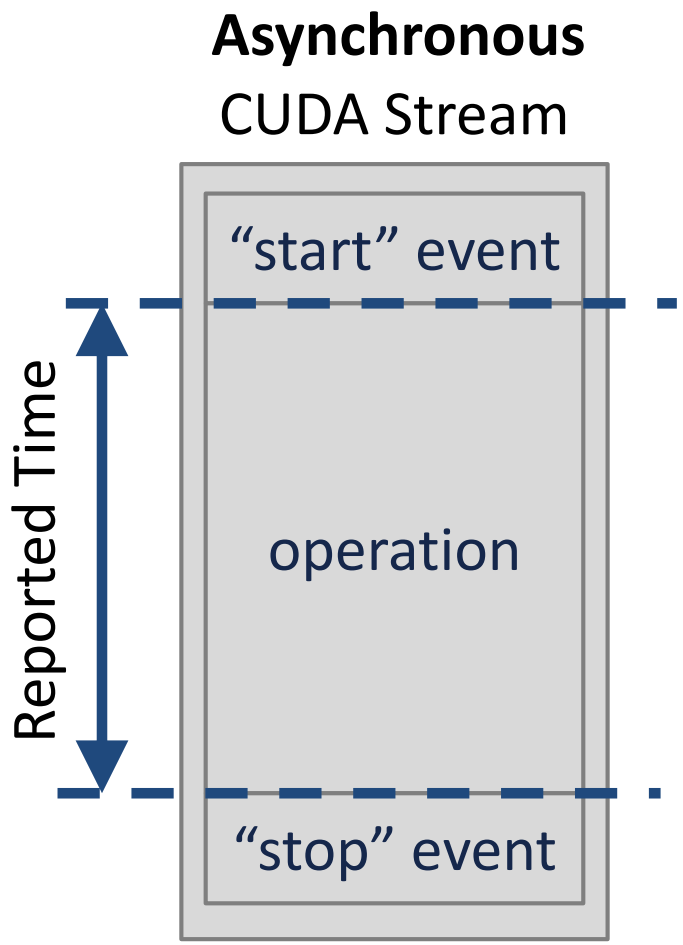
Asynchronous Behavior in Synchronous APIs

- cudaMemcpy
  - CUDA Runtime API §2: “for transfers from pageable host memory to device memory…the function will return once the pageable buffer has been copied to the staging memory, **but the DMA to final destination may not have completed**”

```
// wrong
start = std::chrono::system_clock::now()
cudaMemcpy(..., cudaMemcpyHostToDevice)
end   = std::chrono::system_clock::now()
```

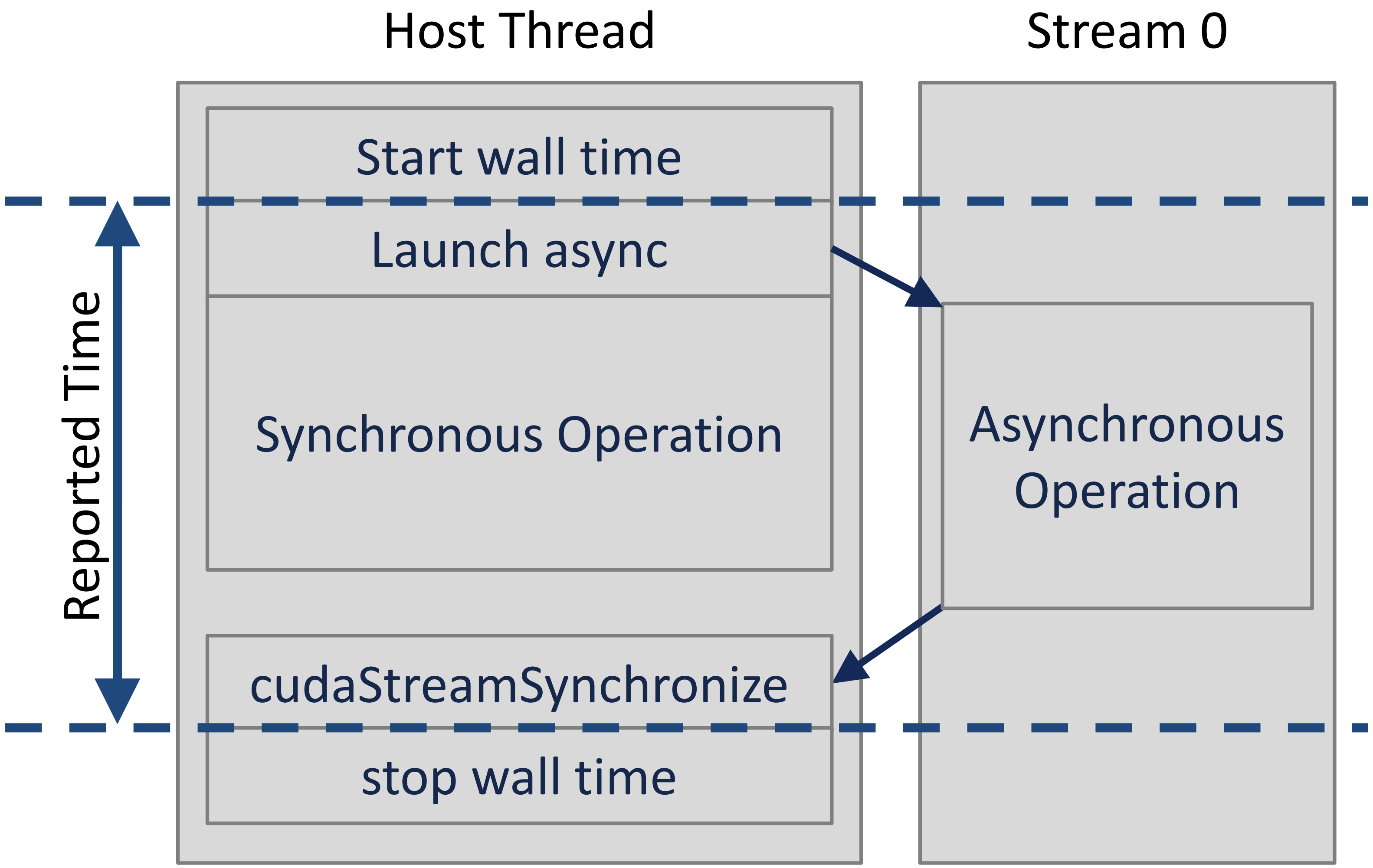
Timing Single Operations





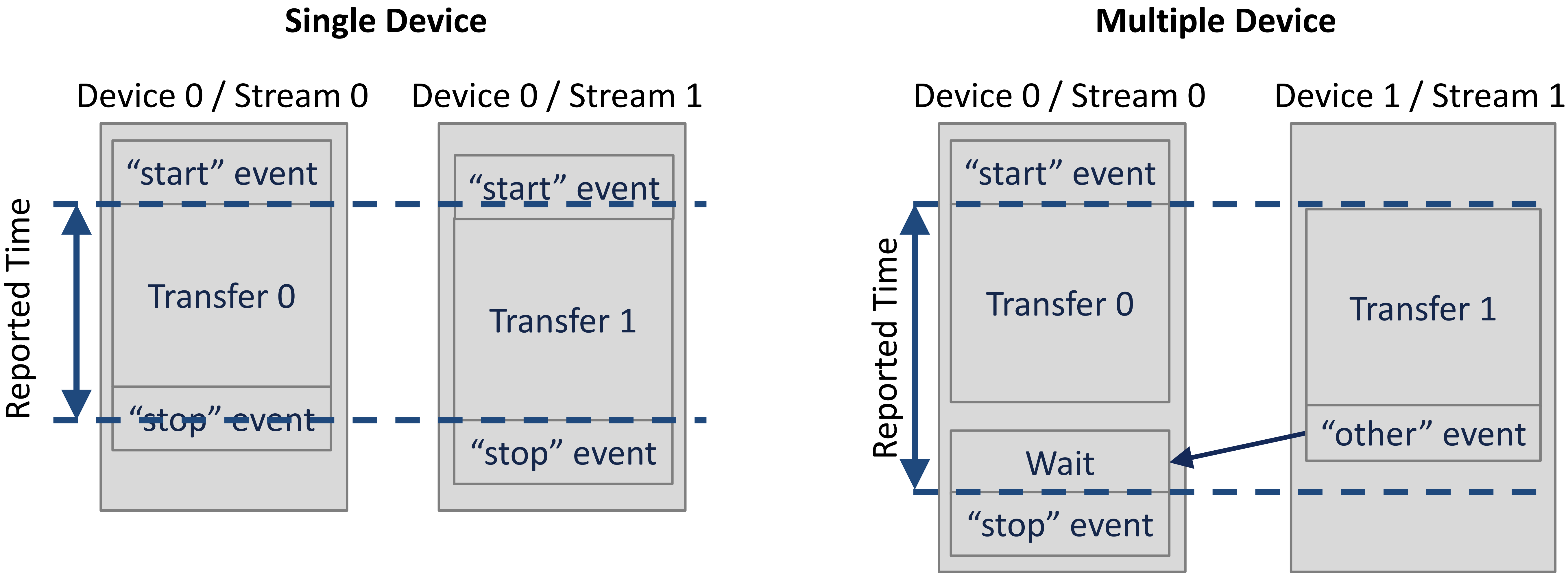
- **No spurious synchronization costs!**

Timing Simultaneous Sync/Async Operations



Unavoidable stream synchronization is measured

Timing Simultaneous Asynchronous Operations

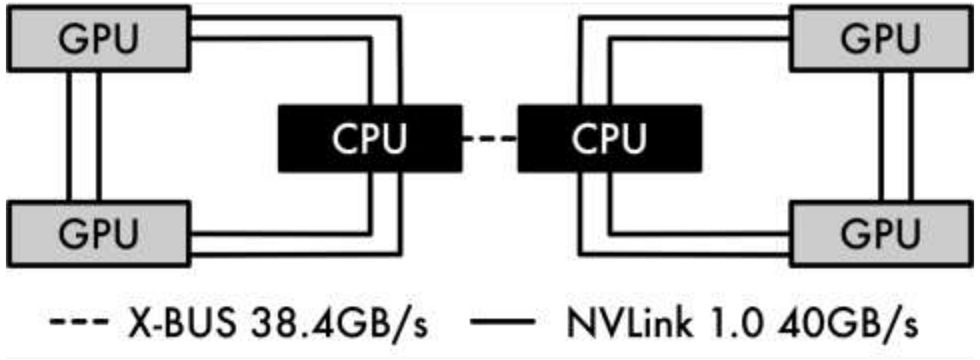


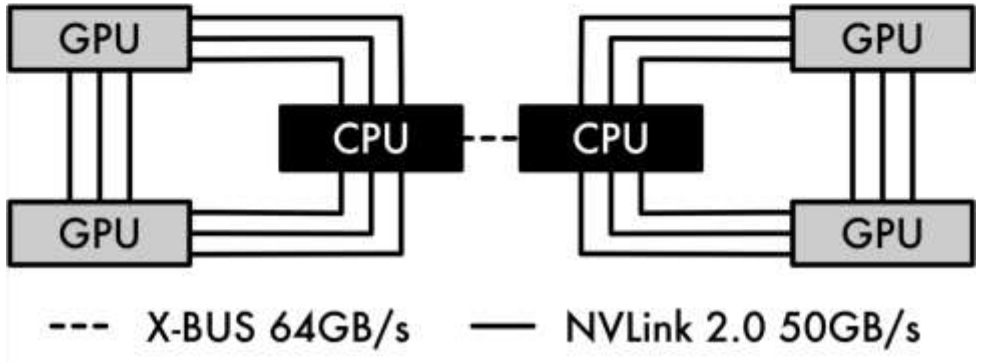
No spurious synchronization costs!

Streams synchronization event measured

IBM S822LC and IBM AC922

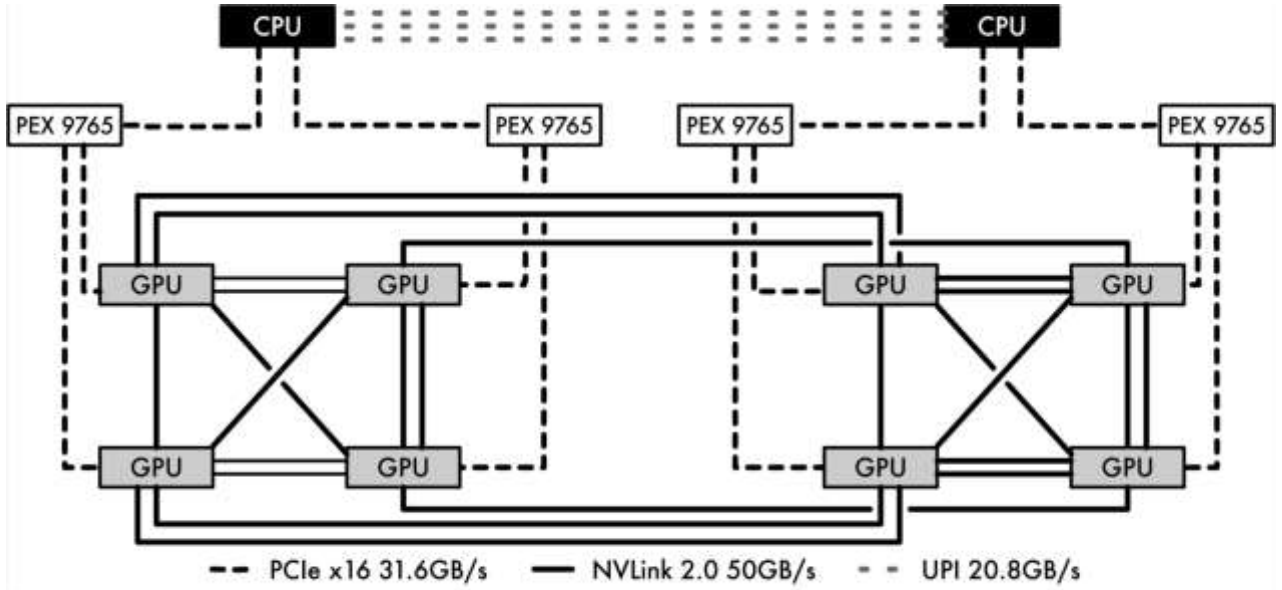
Spec	S822LC	AC922
CPU	2x IBM POWER 8	2x IBM POWER9
GPU	4x Nvidia P100 (Pascal)	4x Nvidia V100 (Volta)
CPU ↔ CPU	X-bus	X-bus
CPU ↔ GPU	2x NVLink 1	3x NVLink 2
GPU ↔ GPU	2x NVLink 1	3x NVLink 2



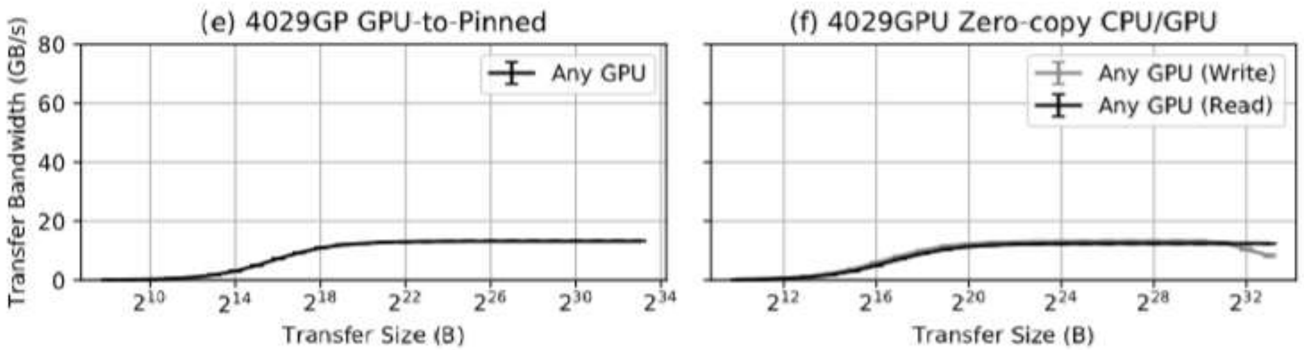


SuperMicro 4029GP-TVRT

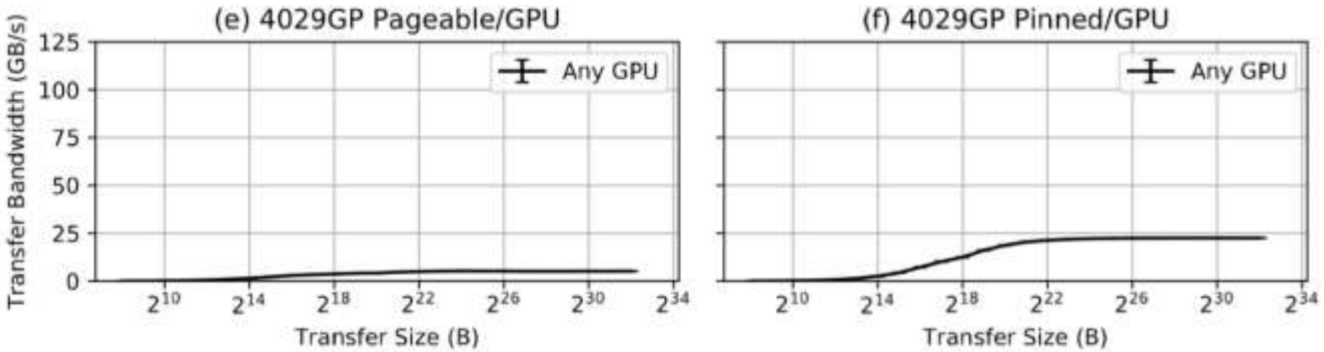
Spec	
CPU	2x Intel Xeon Gold 6148
GPU	8x Nvidia V100 (Volta)
CPU ↔ CPU	Intel UPI
CPU ↔ GPU	PCIe 3.0 x16
GPU ↔ GPU	1x/2x NVLink 2



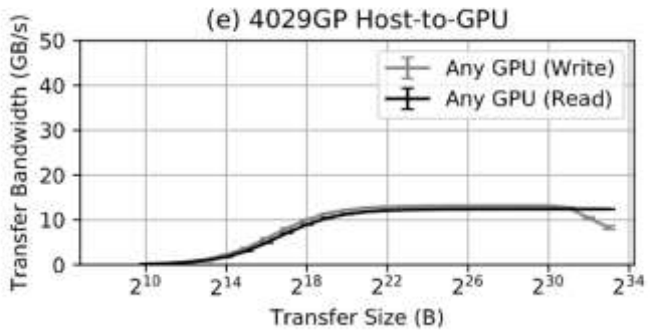
No Locality or Anisotropy on PCIe



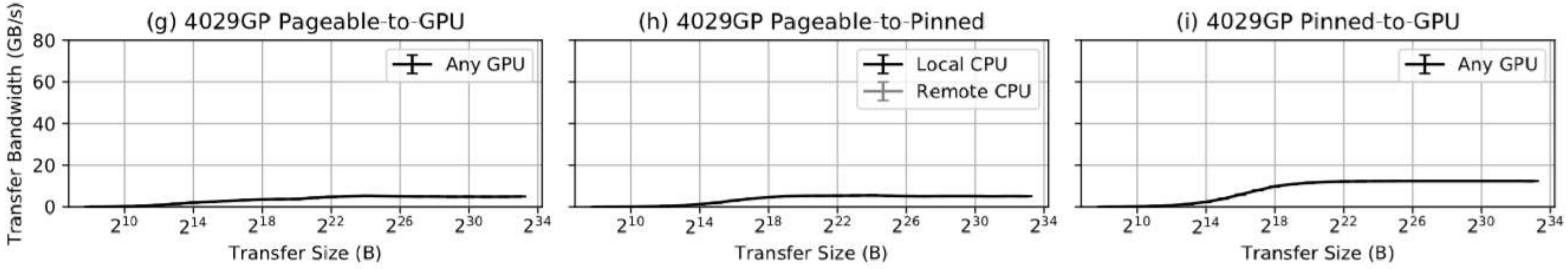
cudaMemcpyAsync vs zero-copy CPU/GPU



cudaMemcpyAsync vs zero-copy CPU/GPU



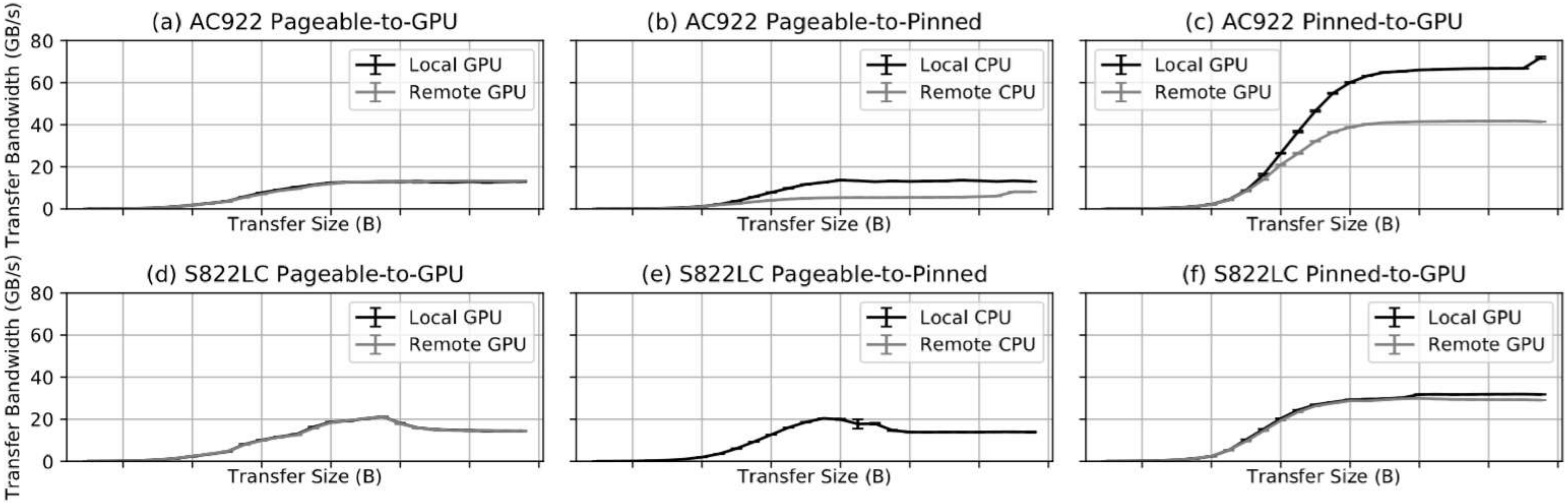
Unified memory demand transfers



cudaMemcpyAsync

- Low bandwidth PCIe 3.0 on 4029GP hides interesting behavior

Pageable Host Allocations and Fast Interconnects



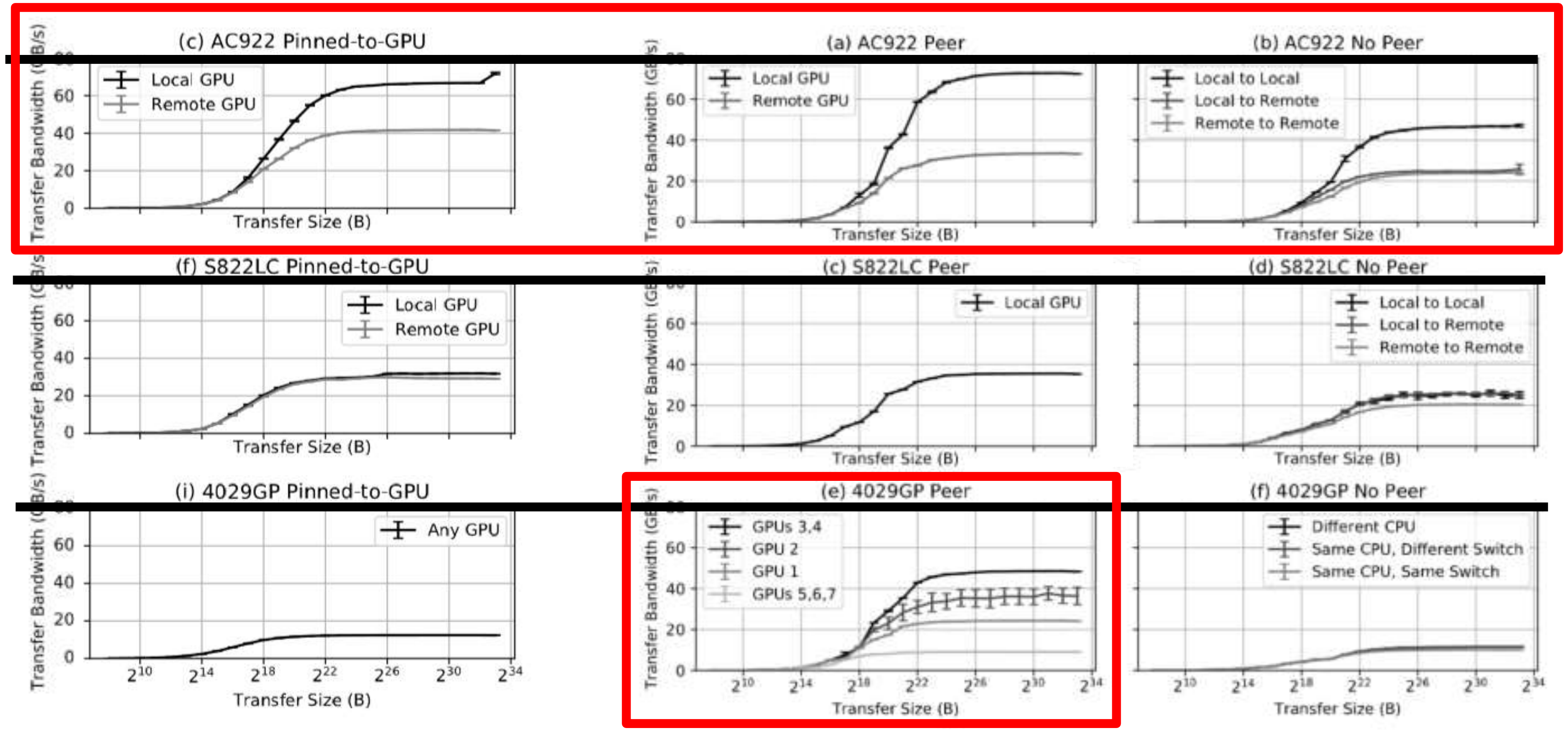
- The implicit pageable-to-pinned copy prevents exploiting fast interconnects
- Multiple threads should speed up pageable-pinned copy
  - Application could use simultaneous transfers
  - CUDA runtime could use multiple worker threads

Strong Locality with High Bandwidth Configurations

80 GB/s

80 GB/s

80 GB/s



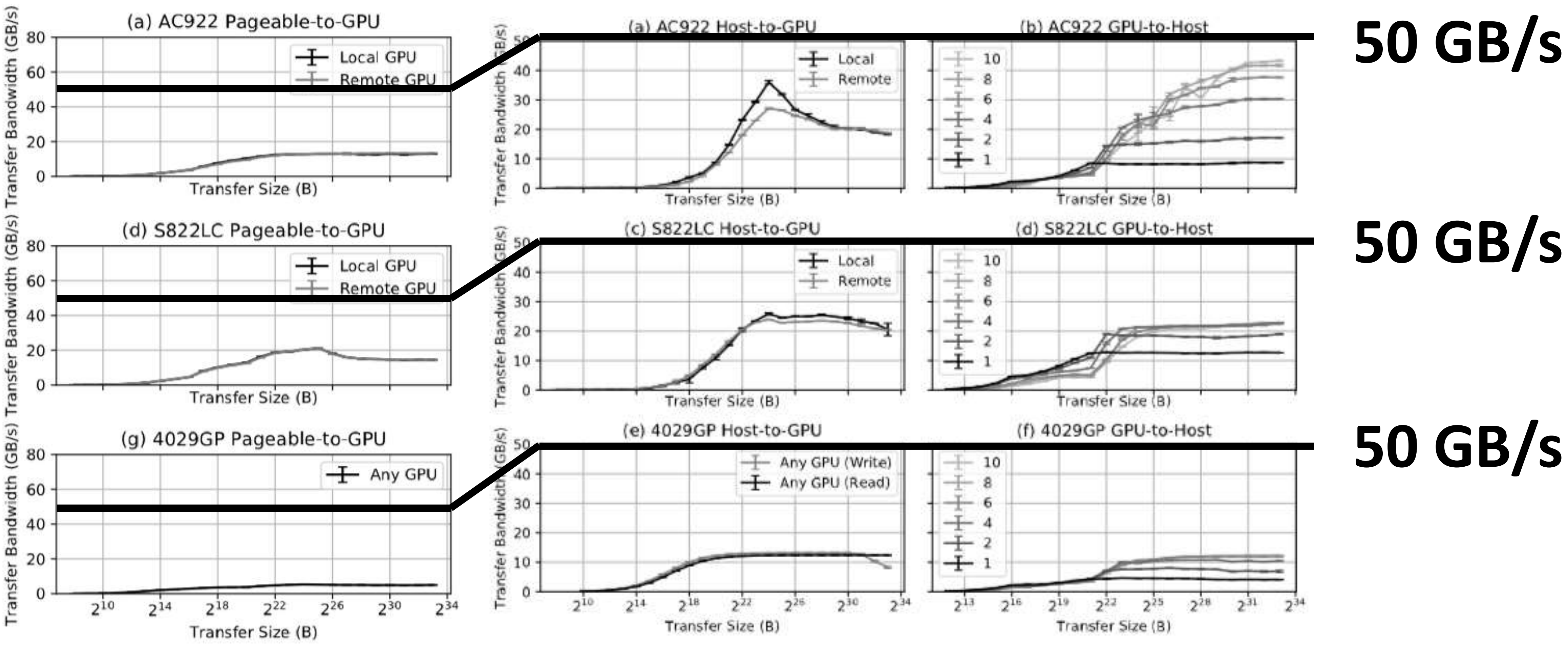
cudaMemcpyAsync CPU-GPU

cudaMemcpyAsync GPU-GPU

Transfers across NVLink 2 show strong locality effects

ECE ILLINOIS

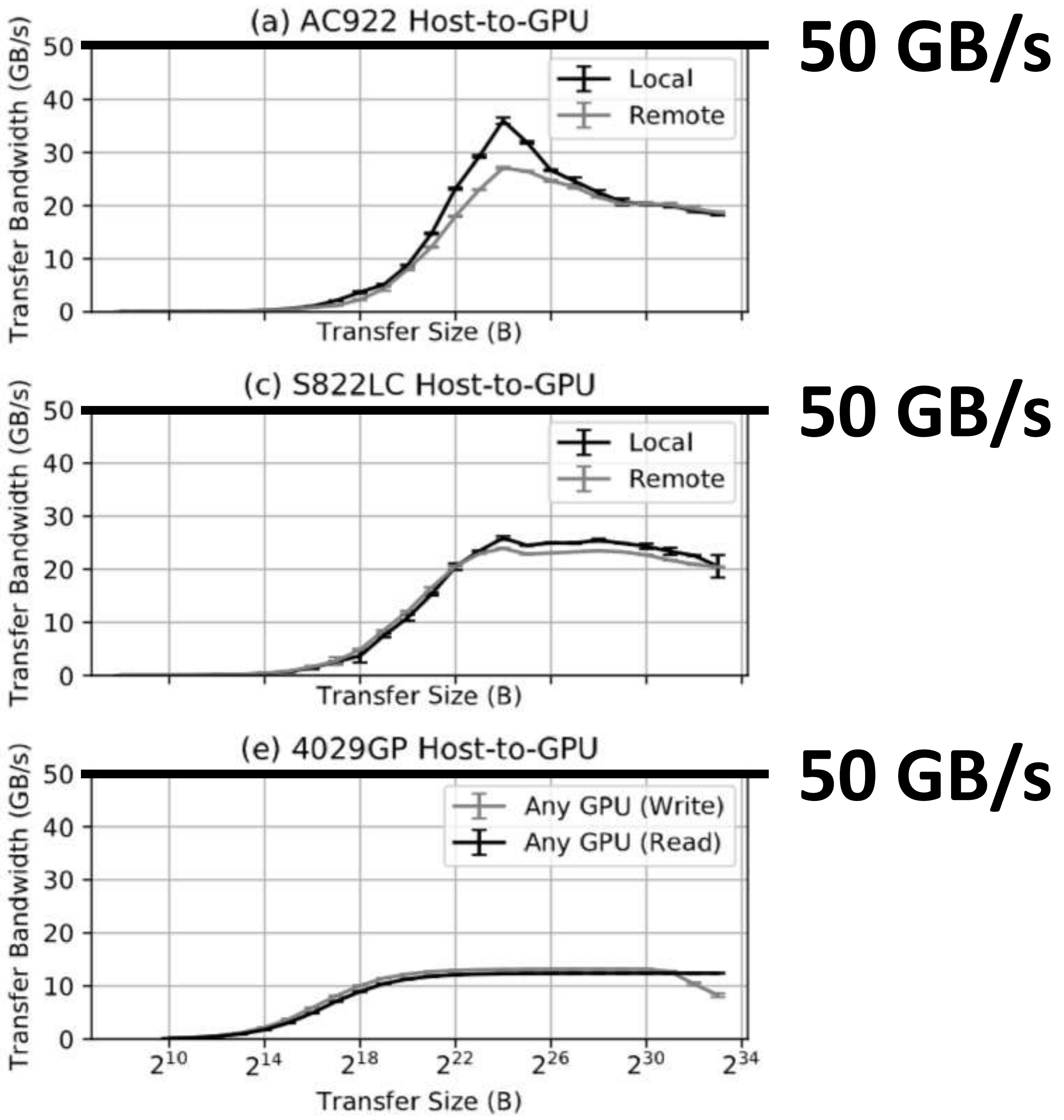
Demand Page Migration vs Explicit Trnasfer



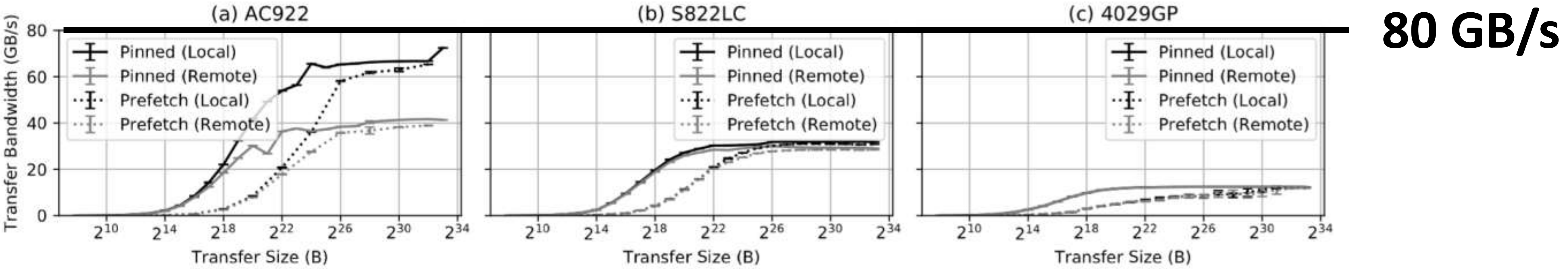
- Multiple host threads are needed to make UM faster

Demand Page Migration

- CUDA system software limits performance available in hardware
  - – Page faults
  - – Per-page driver heuristics
- Underlying interconnect performance not so important



Unified Memory Prefetch vs Explicit



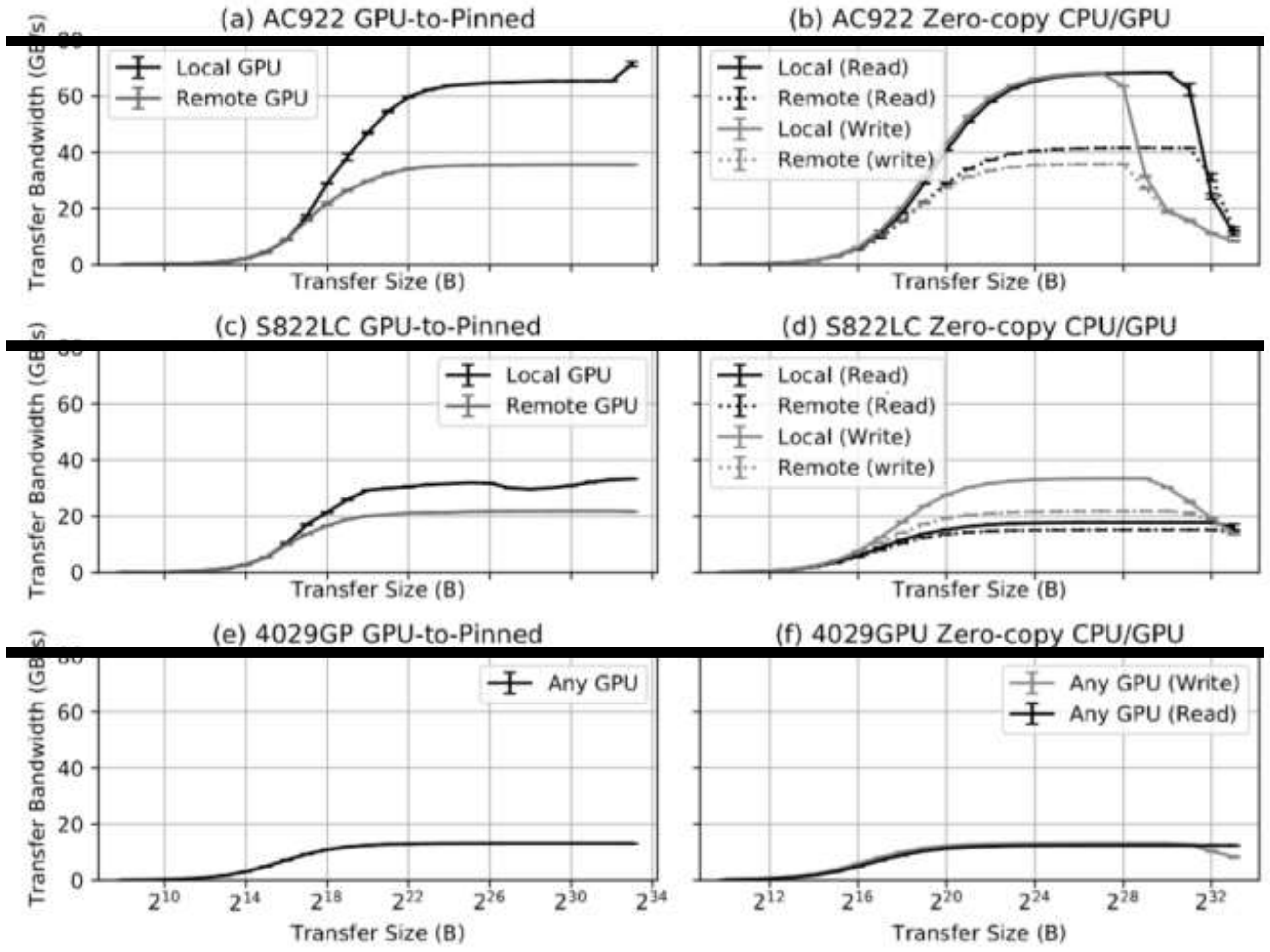
- Unified memory prefetch is slow at intermediate sizes

Zero-Copy

80 GB/s

80 GB/s

80 GB/s



- Implicit, like unified memory
- Unlike unified memory, can achieve near interconnect theoretical bandwidth

Open-source & Docker

- v0.7.3 released April 8th
- Github: c3sr/comm_scope
- Docker: c3sr/comm_scope

- CUDA 8.0+, CMake 3.12+
- x86 and POWER
- Apache 2.0 license

Future Work

- Unified Memory Microbenchmarks
  - Access patterns & driver heuristics
- System-aware CPU/GPU and GPU/GPU data structures
  - How to allocate and move data depending on who produces and who consumes
    - Hints from application or records from previous executions
- System health status
  - Sanity check during system firmware development or system bring-up

Conclusion

- Comprehensive coverage of CUDA communication methods
- Bandwidth affected by CUDA APIs, non-CUDA system knobs, system topology
- High-bandwidth interconnects expose interesting behavior of hardware/software system
- Open-source, cross-platform, artifact evaluation stamp

Thank you / Questions

pearson@illinois.edu

https://cwpearson.github.io

Other C3SR System Performance Research Projects

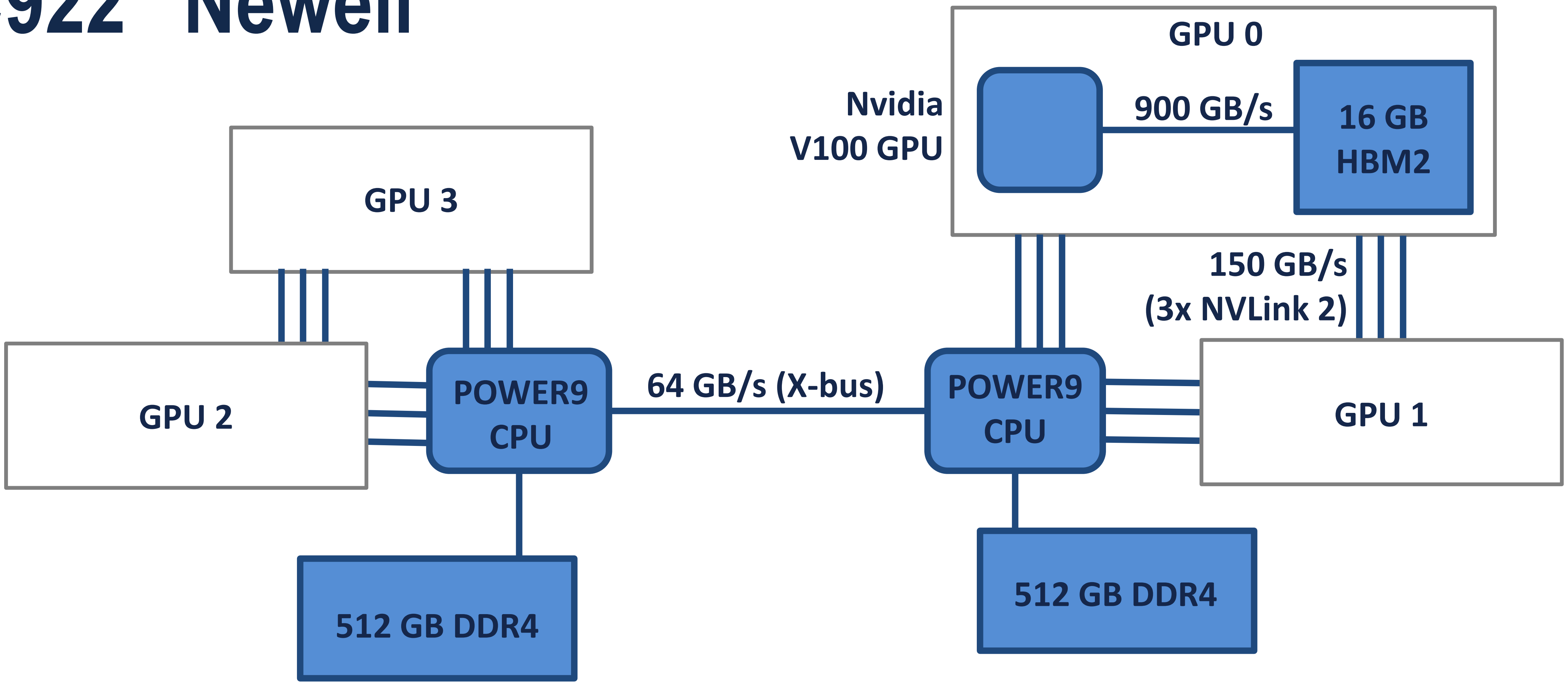
System microbenchmarks: `https://scope.c3sr.com`

Full-stack machine learning with tracing: `https://mlmodelscope.org`

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation award OCI-0725070 and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications

IBM AC922 “Newell”



- IBM AC922 “Newell”

CUDA Events

- CUDA C Programming Guide §3.2.6.3
  - cudaEventRecord() will fail if the input event and stream are associated with two different devices
  - cudaEventElapsedTime will fail if the two input events are associated with different devices

```
// wrong
// ... create one stream for each device
// place a start event, a kernel, and a stop even in each stream
// compare the earliest start with the latest stop to get total time
```