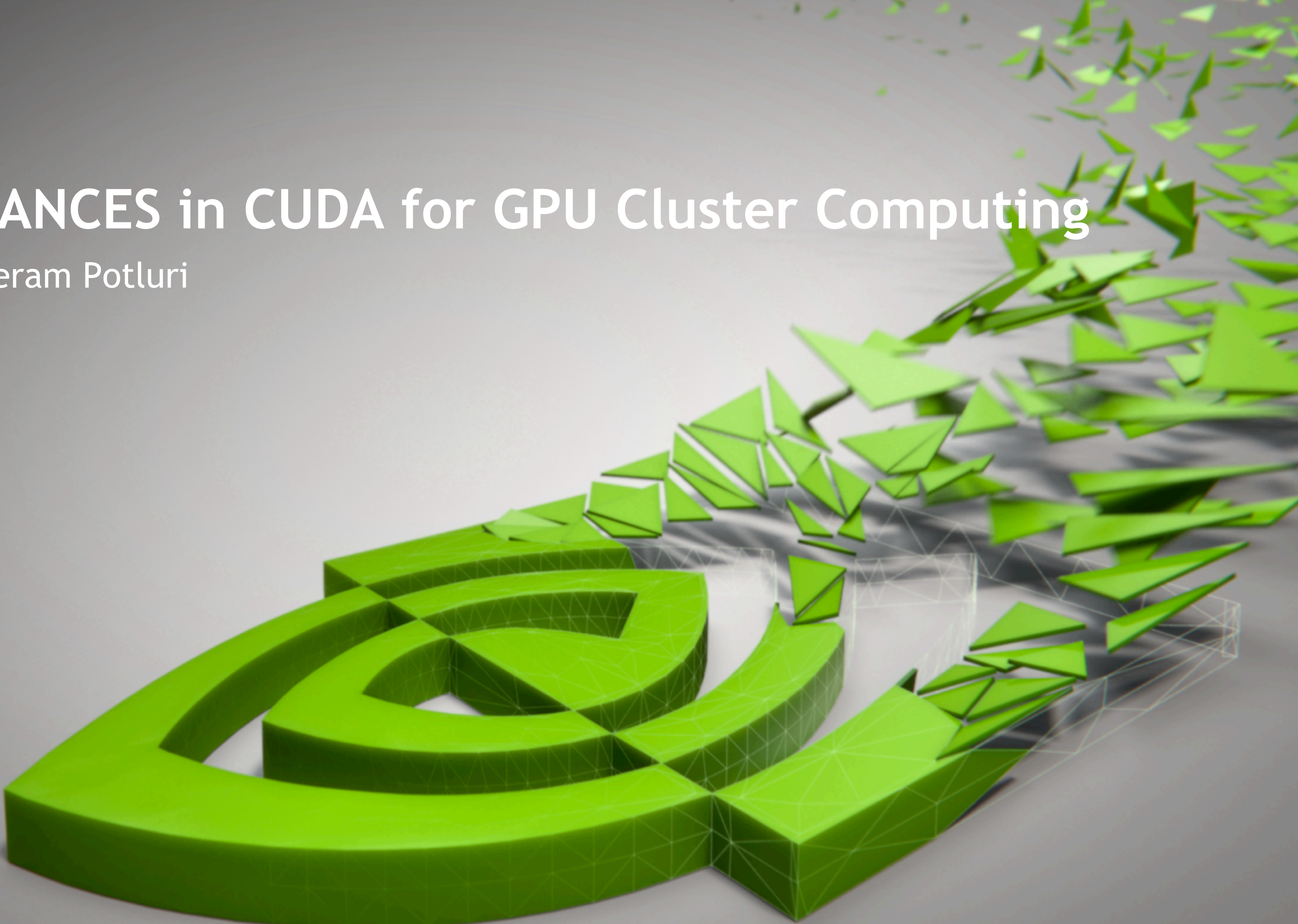
RECENT ADVANCES in CUDA for GPU Cluster Computing

Davide Rossetti, Sreeram Potluri



AGENDA

Introduction

Pascal and NVLink

CUDA 8

GPUDirect Technologies

NCCL

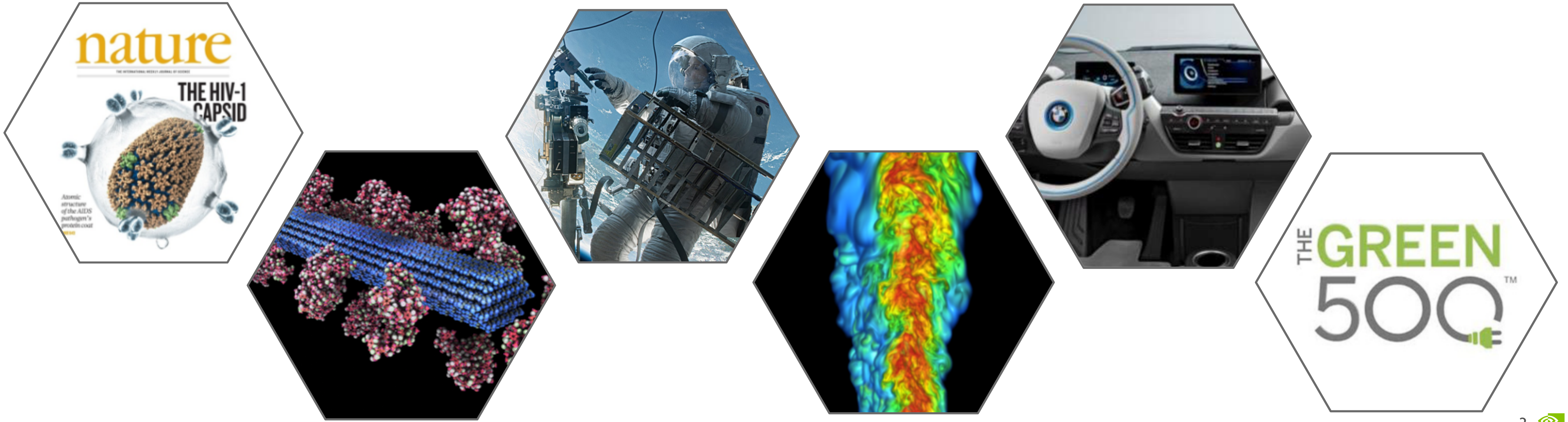
Summary

NVIDIA - INVENTOR OF THE GPU

NVIDIA Invented the GPU in 1999, with over 1 Billion shipped to date.

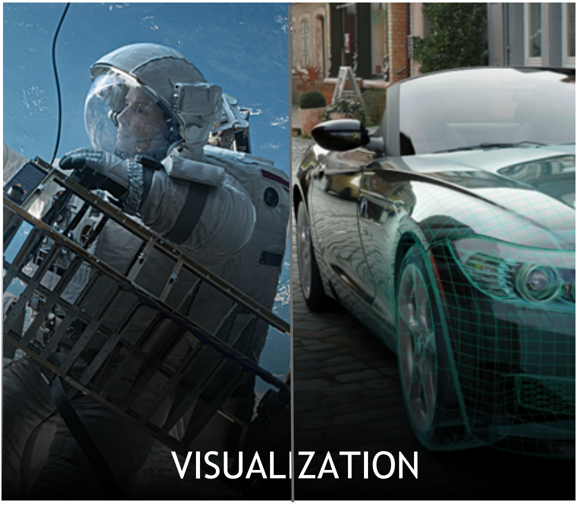
Initially a dedicated a graphics processor for PCs, the GPU’s computational power and energy efficiency led to it quickly being adopted for professional and scientific simulation and visualization.

In 2007, NVIDIA launched the **CUDA®** programming platform, and opened up the general purpose parallel processing capabilities of the GPU.



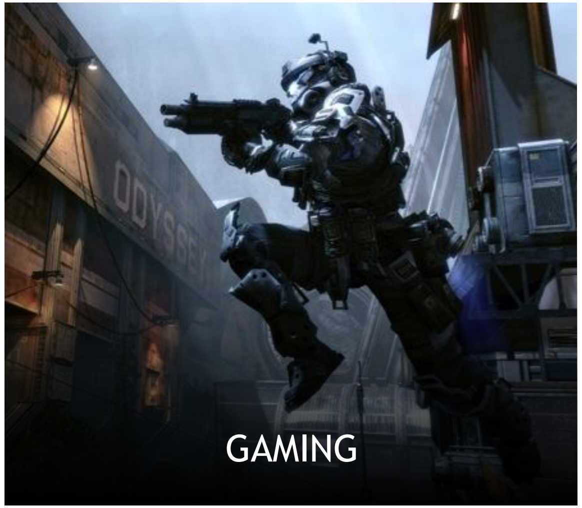
NVIDIA PLATFORM

HPC and DATA CENTER

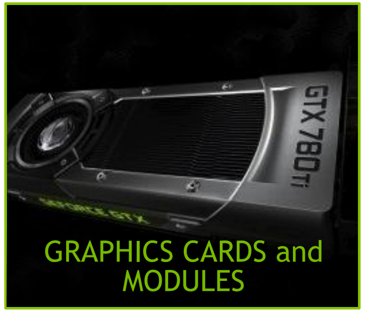


VISUALIZATION

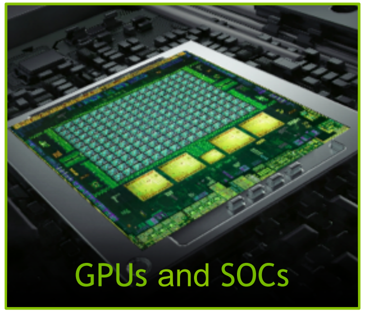
AUTO



GAMING



GRAPHICS CARDS and MODULES



GPUs and SOCs

SYSTEMS

IP

USA - TWO FLAGSHIP SUPERCOMPUTERS

OAK RIDGE National Laboratory

Lawrence Livermore National Laboratory

SUMMIT

150-300 PFLOPS
Peak Performance

SIERRA

> 100 PFLOPS
Peak Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

>40 TFLOPS per Node, >3,400 Nodes

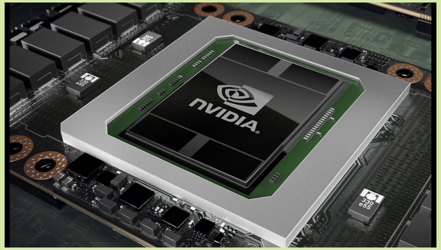
2017

Powered by the NVIDIA VOLTA GPU

INTRODUCING TESLA P100

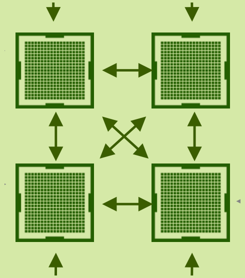
- Extreme performance
  - 5.3 TFLOPS double-precision, 10.6 TFLOPS single-precision, 21.2 TFLOPS half-precision
- NVLink Interconnect
  - 20 GB/sec per lane per direction, 4 lanes per GPU
- HBM2 Memory
  - 3x the memory bandwidth of Maxwell GPU
- Unified Memory
  - Simplifies parallel programming

Pascal Architecture



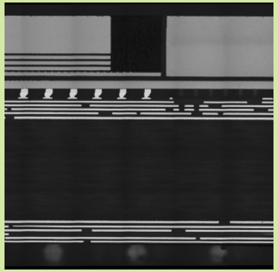
Highest Compute Performance

NVLink



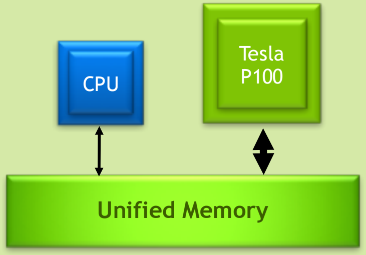
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory



Unifying Compute & Memory in Single Package

Page Migration Engine



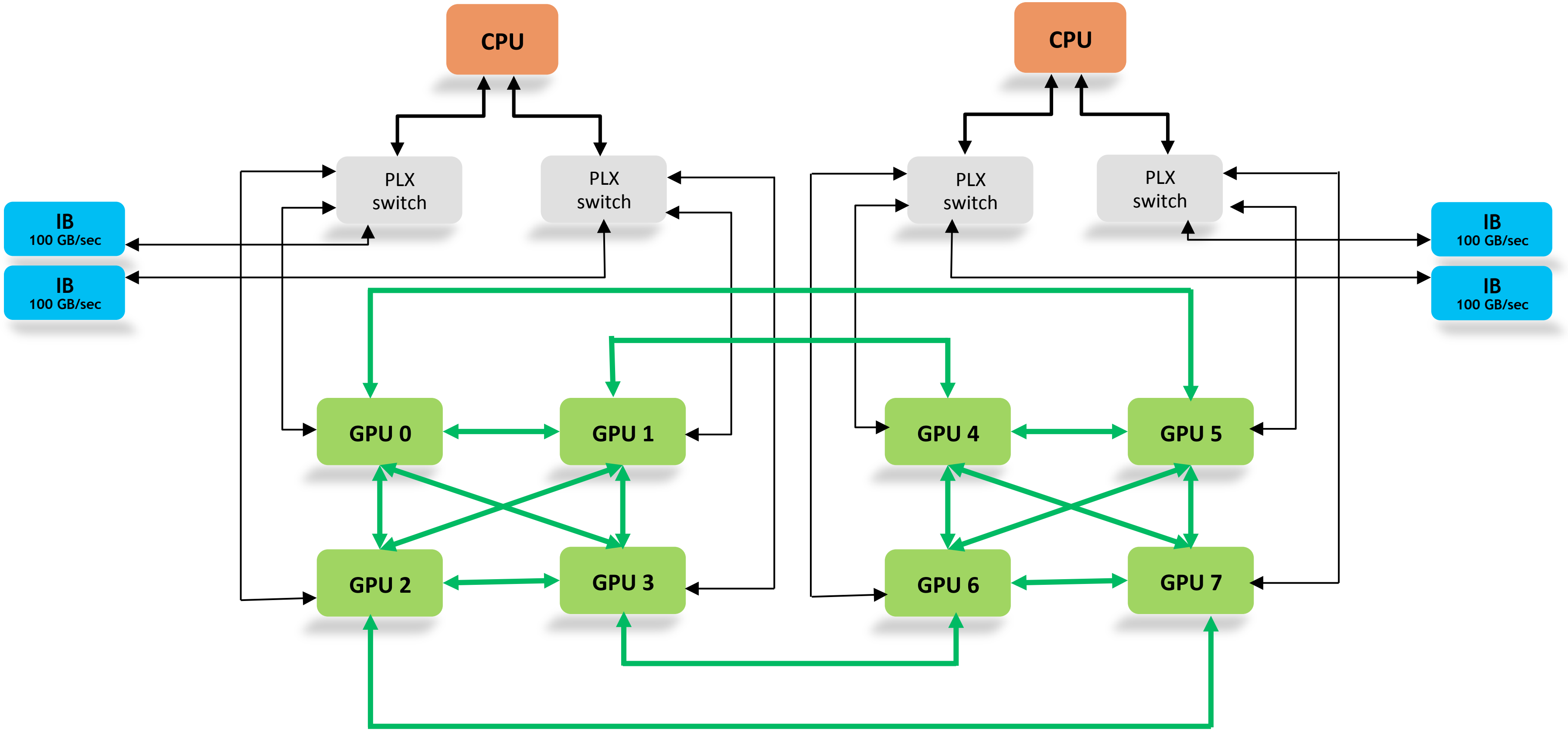
Simple Parallel Programming with 512 TB of Virtual Memory

INTRODUCING NVIDIA DGX-1

AI Supercomputer-in-a-Box

170 TFLOPS | 8x Tesla P100 16GB | NVLink Hybrid Cube Mesh
2x Xeon | 8 TB RAID 0 | Quad IB 100Gbps, Dual 10GbE | 3U — 3200W

DGX-1 TOPOLOGY

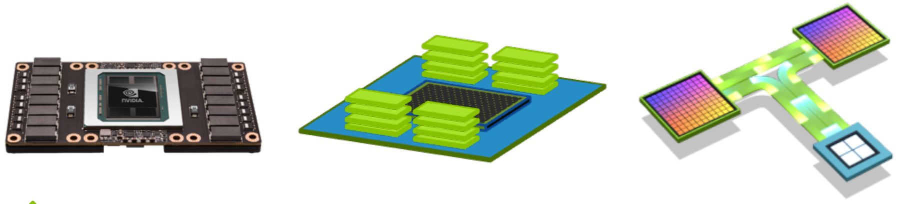


CUDA 8

INTRODUCING CUDA 8

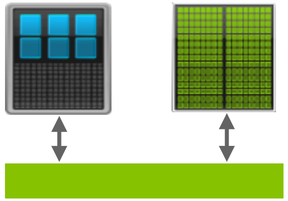
Pascal Support

New Architecture, Stacked Memory, NVLINK



Unified Memory

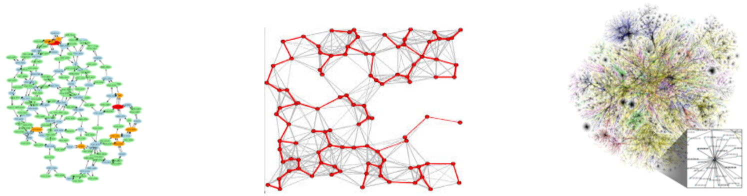
Simple Parallel Programming with large virtual memory



Libraries

nvGRAPH – library for accelerating graph analytics apps

FP16 computation to boost Deep Learning workloads

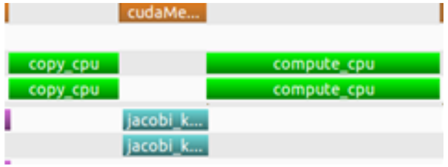


Developer Tools

Critical Path Analysis to speed overall app tuning

OpenACC profiling to optimize directive performance

Single GPU debugging on Pascal

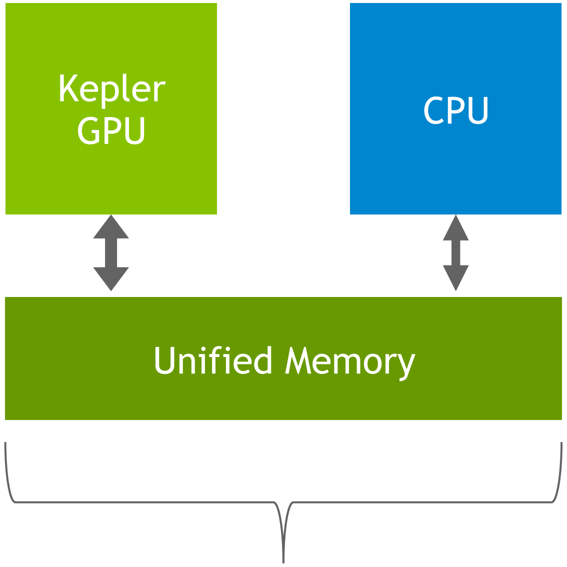


UNIFIED MEMORY

UNIFIED MEMORY

Dramatically Lower Developer Effort

CUDA 6+



Allocate Up To
GPU Memory Size

Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

```c
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

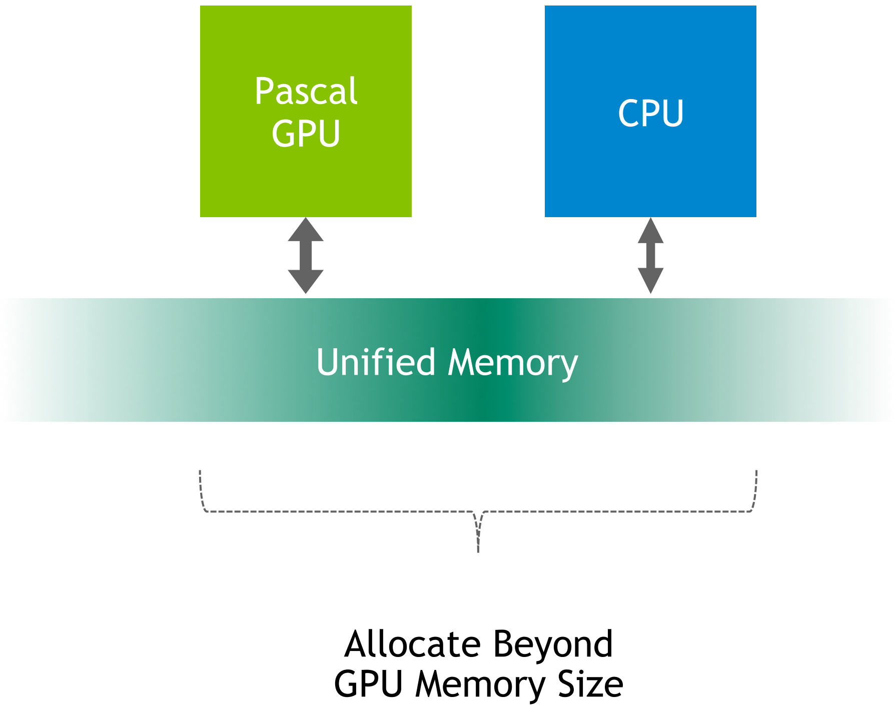
CUDA 6 Code with Unified Memory

```c
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

CUDA 8: UNIFIED MEMORY

Large datasets, simple programming, High Performance

CUDA 8



Enable Large Data Models

- Oversubscribe GPU memory
- Allocate up to system memory size

Simpler Data Accesss

- CPU/GPU Data coherence
- Unified memory atomic operations

Tune Unified Memory Performance

- Usage hints via cudaMemAdvise API
- Explicit prefetching API

UNIFIED MEMORY ON PASCAL

GPU memory oversubscription

```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 32 GB
  char *data;
  size_t size = 32*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

32 GB allocation

Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU when “hot”.

Fails on Kepler/Maxwell

GPU OVERSUBSCRIPTION

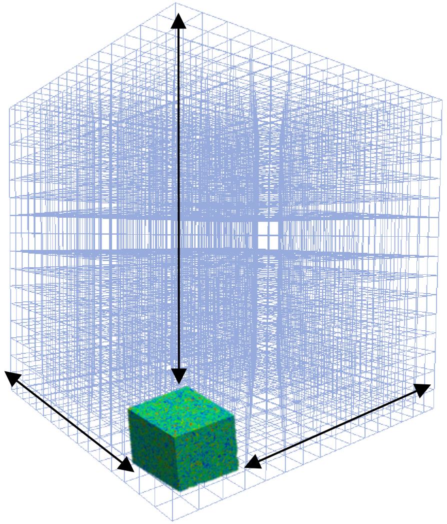
Now possible with Pascal

Many domains would benefit from GPU memory oversubscription:

Combustion – many species to solve for

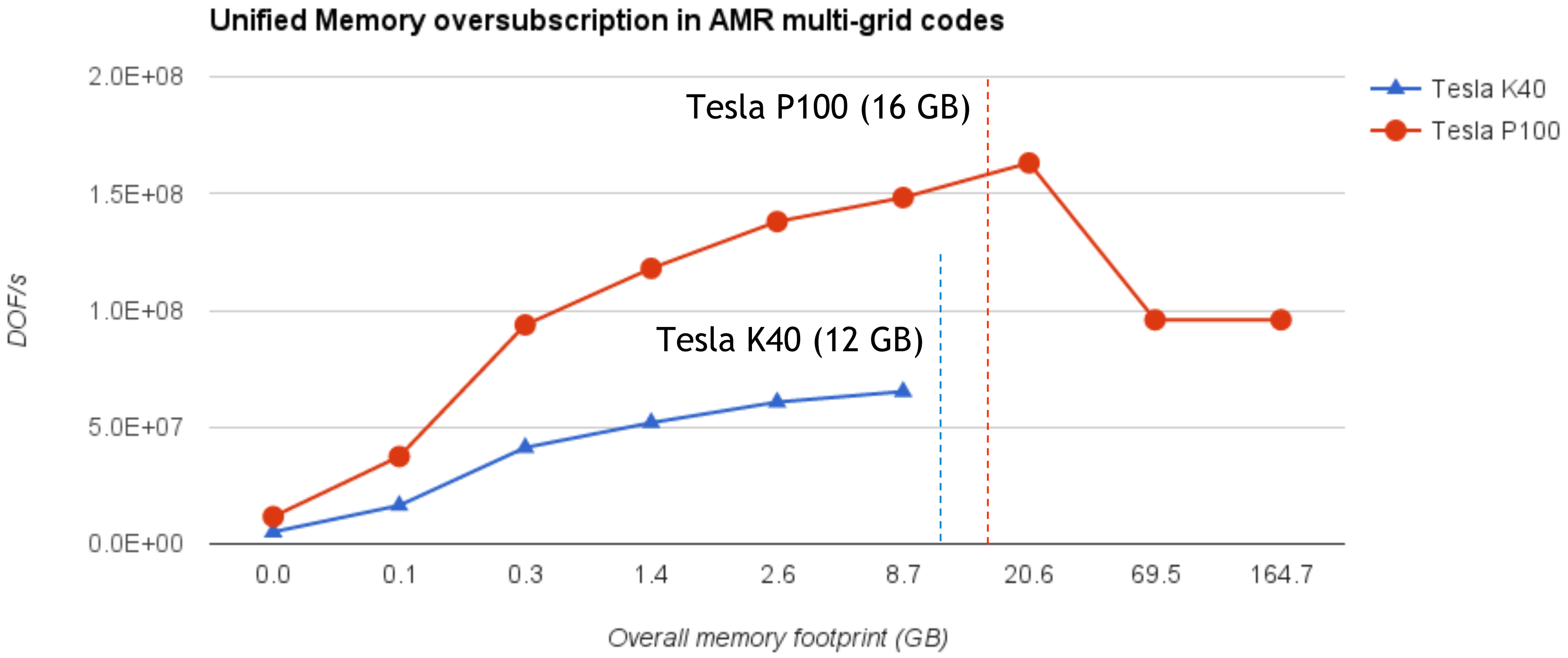
Quantum chemistry – larger systems

Ray tracing - larger scenes to render



GPU OVERSUBSCRIPTION

HPGMG: high-performance multi-grid



*Tesla P100 performance is very early modelling results

UNIFIED MEMORY ON PASCAL

Concurrent CPU/GPU access to managed memory

```
__global__ void mykernel(char *data) {
  data[1] = ‘g’;
}

void foo() {
  char *data;
  cudaMallocManaged(&data, 2);

  mykernel<<<...>>>(data);
  // no synchronize here
  data[0] = ‘c’;

  cudaFree(data);
}
```

OK on Pascal: just a page fault

Concurrent CPU access to ‘data’ on previous GPUs caused a fatal segmentation fault

UNIFIED MEMORY ON PASCAL

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

PERFORMANCE TUNING ON PASCAL

Explicit Memory Hints and Prefetching

Advise runtime on known memory access behaviors with `cudaMemAdvise()`

`cudaMemAdviseSetReadMostly`: Specify read duplication
`cudaMemAdviseSetPreferredLocation`: suggest best location
`cudaMemAdviseSetAccessedBy`: initialize a mapping

Explicit prefetching with `cudaMemPrefetchAsync(ptr, length, destDevice, stream)`

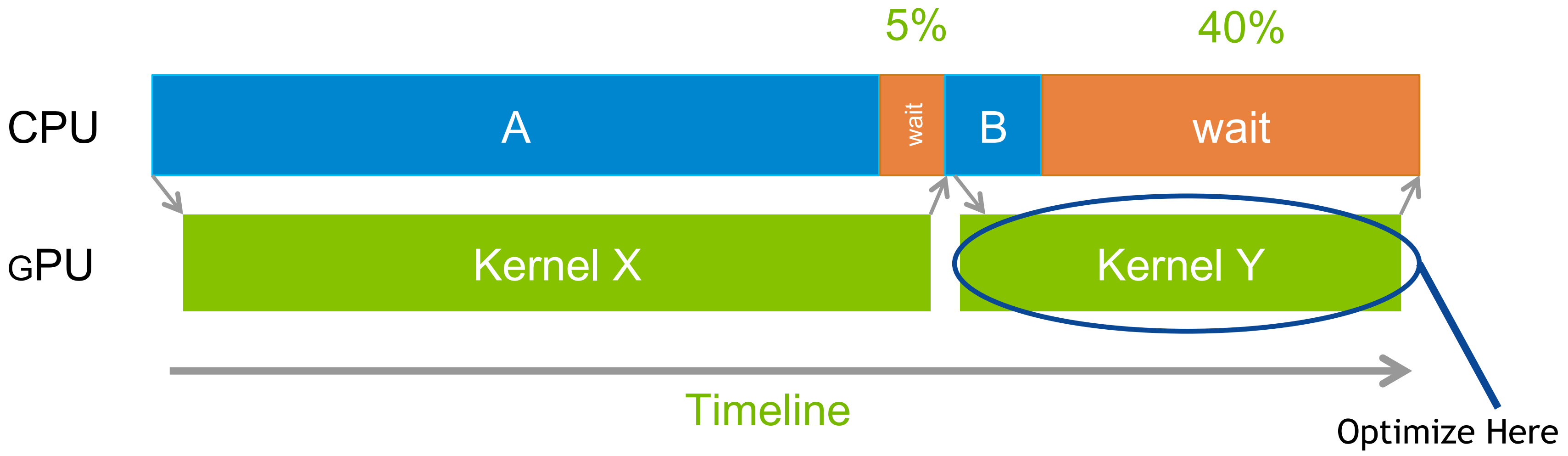
Unified Memory alternative to cudaMemcpyAsync

Asynchronous operation that follows CUDA stream semantics

ENHANCED PROFILING

DEPENDENCY ANALYSIS

Easily Find the Critical Kernel To Optimize



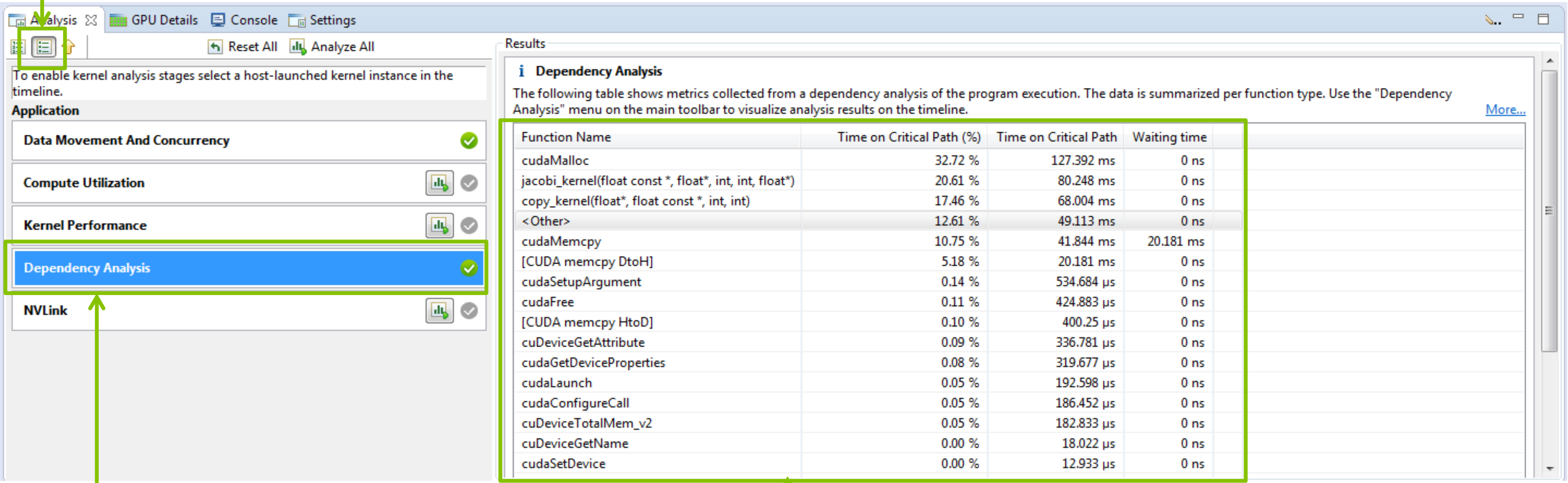
The longest running kernel is not always the most critical optimization target

DEPENDENCY ANALYSIS

Visual Profiler

Unguided Analysis

Generating critical path



To enable kernel analysis stages select a host-launched kernel instance in the timeline.

Application

- Data Movement And Concurrency ✓
- Compute Utilization
- Kernel Performance
- Dependency Analysis ✓
- NVLink

Dependency Analysis

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

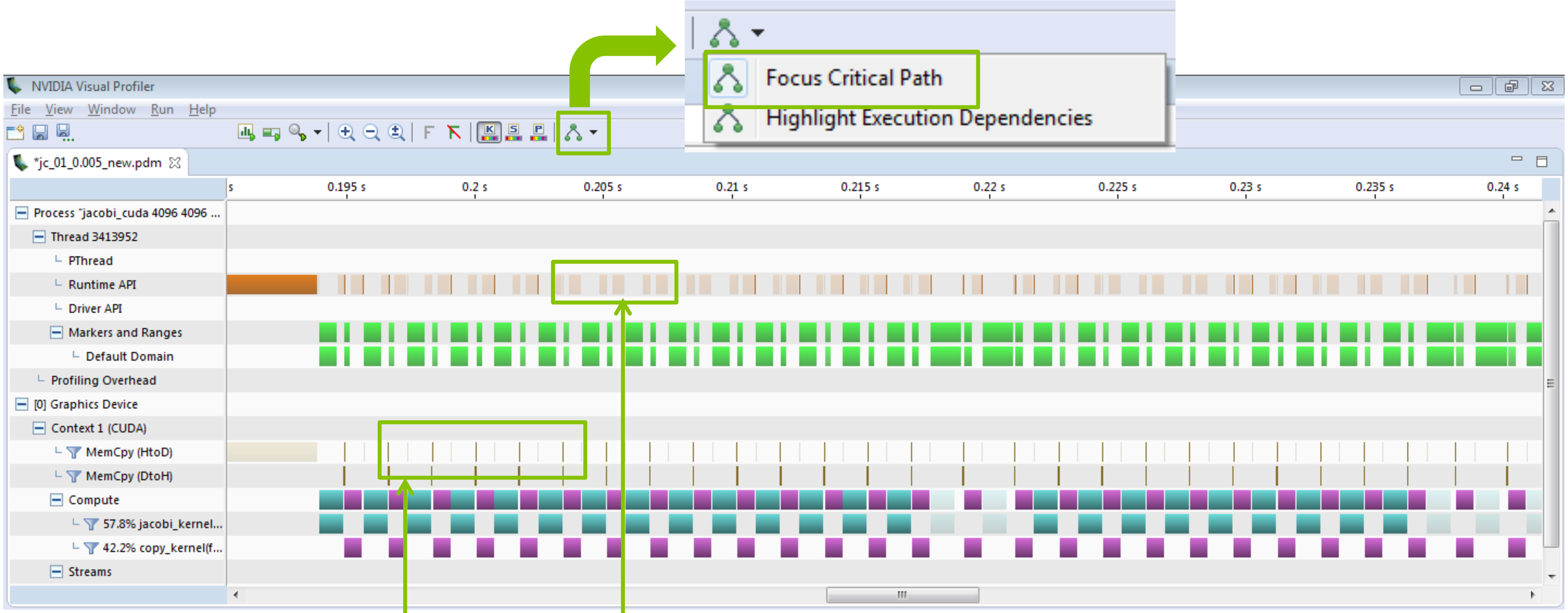
Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
cudaMalloc	32.72 %	127.392 ms	0 ns
jacobi_kernel(float const *, float*, int, int, float*)	20.61 %	80.248 ms	0 ns
copy_kernel(float*, float const *, int, int)	17.46 %	68.004 ms	0 ns
<Other>	12.61 %	49.113 ms	0 ns
cudaMemcpy	10.75 %	41.844 ms	20.181 ms
[CUDA memcpy DtoH]	5.18 %	20.181 ms	0 ns
cudaSetupArgument	0.14 %	534.684 µs	0 ns
cudaFree	0.11 %	424.883 µs	0 ns
[CUDA memcpy HtoD]	0.10 %	400.25 µs	0 ns
cuDeviceGetAttribute	0.09 %	336.781 µs	0 ns
cudaGetDeviceProperties	0.08 %	319.677 µs	0 ns
cudaLaunch	0.05 %	192.598 µs	0 ns
cudaConfigureCall	0.05 %	186.452 µs	0 ns
cuDeviceTotalMem_v2	0.05 %	182.833 µs	0 ns
cuDeviceGetName	0.00 %	18.022 µs	0 ns
cudaSetDevice	0.00 %	12.933 µs	0 ns

Dependency Analysis

Functions on critical path

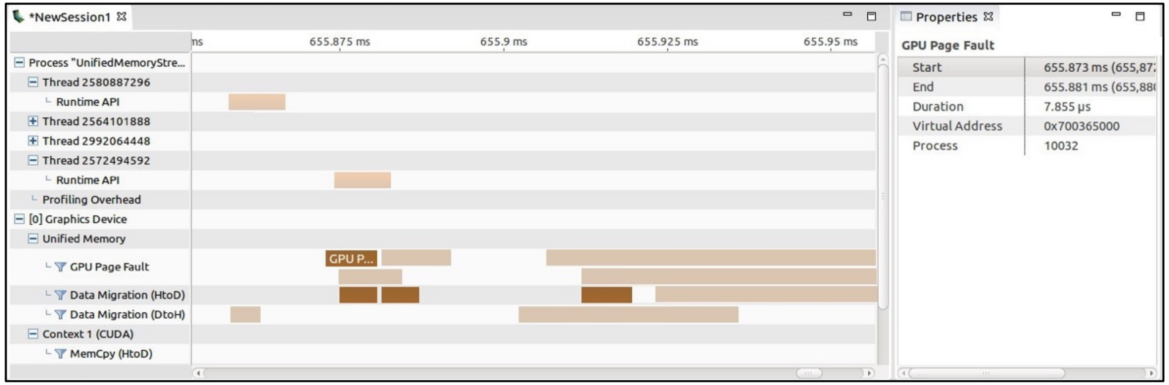
DEPENDENCY ANALYSIS

Visual Profiler



APIs, GPU activities not in critical path are greyed out

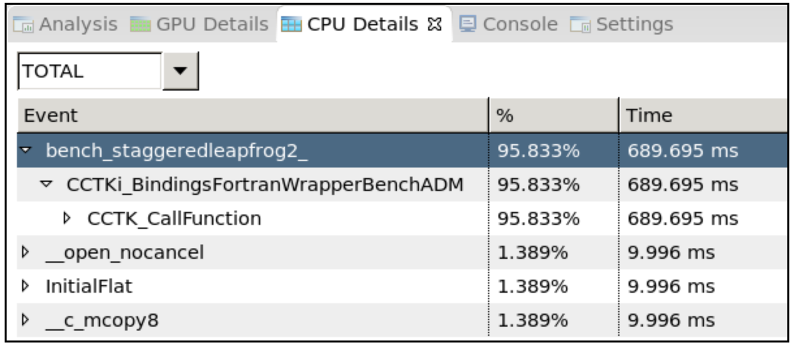
MORE CUDA 8 PROFILER FEATURES



GPU Page Fault

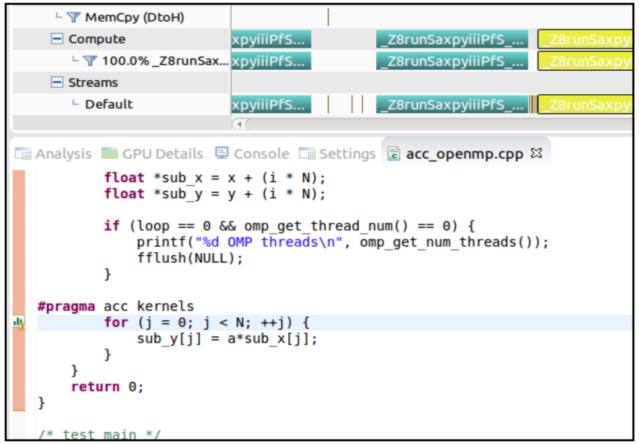
Start	655.873 ms (655,87:
End	655.881 ms (655,88(
Duration	7.855 µs
Virtual Address	0x700365000
Process	10032

Unified Memory Profiling



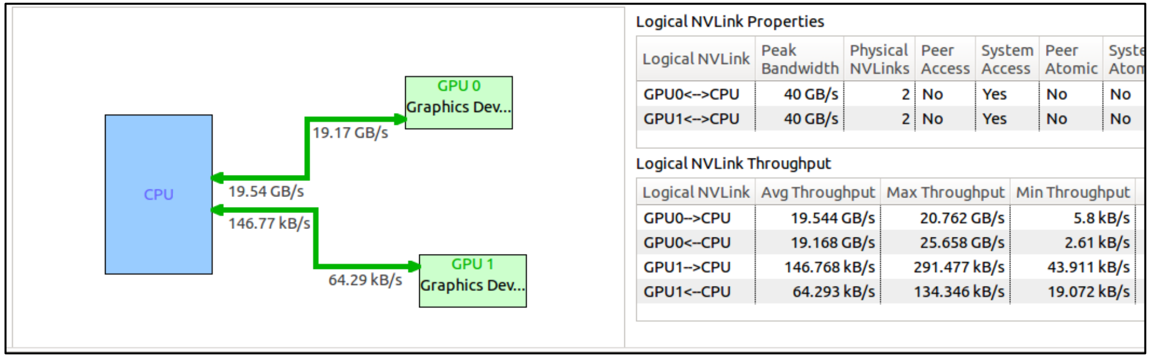
Event	%	Time
bench_staggeredleapfrog2_	95.833%	689.695 ms
CCTKi_BindingsFortranWrapperBenchADM	95.833%	689.695 ms
CCTK_CallFunction	95.833%	689.695 ms
__open_nocancel	1.389%	9.996 ms
InitialFlat	1.389%	9.996 ms
__c_mcopy8	1.389%	9.996 ms

CPU Profiling



```
        float *sub_x = x + (i * N);
        float *sub_y = y + (i * N);

        if (loop == 0 && omp_get_thread_num() == 0) {
            printf("%d OMP threads\n", omp_get_num_threads());
            fflush(NULL);
        }

#pragma acc kernels
        for (j = 0; j < N; ++j) {
            sub_y[j] = a*sub_x[j];
        }
    }
    return 0;
}

/* test main */
```

OpenACC Profiling



Logical NVLink Properties

Logical NVLink	Peak Bandwidth	Physical NVLinks	Peer Access	System Access	Peer Atomic	Syste Atom
GPU0<-->CPU	40 GB/s	2	No	Yes	No	No
GPU1<-->CPU	40 GB/s	2	No	Yes	No	No

Logical NVLink Throughput

Logical NVLink	Avg Throughput	Max Throughput	Min Throughput
GPU0-->CPU	19.544 GB/s	20.762 GB/s	5.8 kB/s
GPU0<--CPU	19.168 GB/s	25.658 GB/s	2.61 kB/s
GPU1-->CPU	146.768 kB/s	291.477 kB/s	43.911 kB/s
GPU1<--CPU	64.293 kB/s	134.346 kB/s	19.072 kB/s

NVLink Topology and Bandwidth profiling

GPUDIRECT

GPUDIRECT FAMILY[1]

- GPUDirect Shared GPU-Sysmem for optimized inter-node copy
- GPUDirect P2P for intra-node
  - accelerated GPU-GPU memcpy
  - inter-GPU direct load/store access
- GPUDirect RDMA[2] for optimized inter-node communication
- GPUDirect Async for optimized inter-node communication

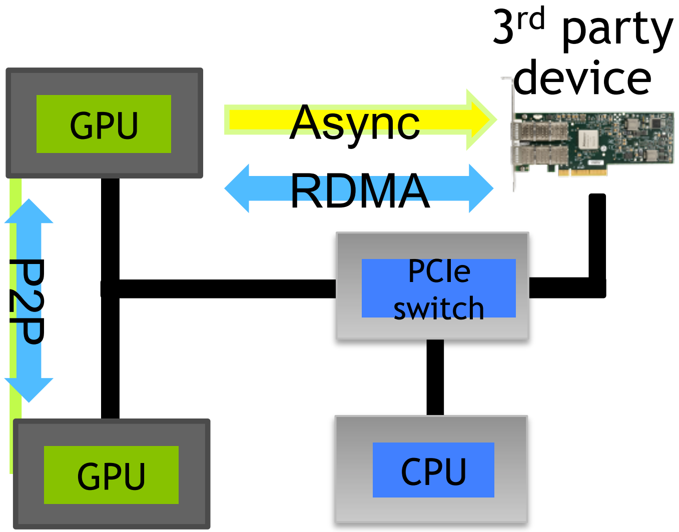
[1] developer info: https://developer.nvidia.com/gpudirect

[2] http://docs.nvidia.com/cuda/gpudirect-rdma

GPUDIRECT

scopes

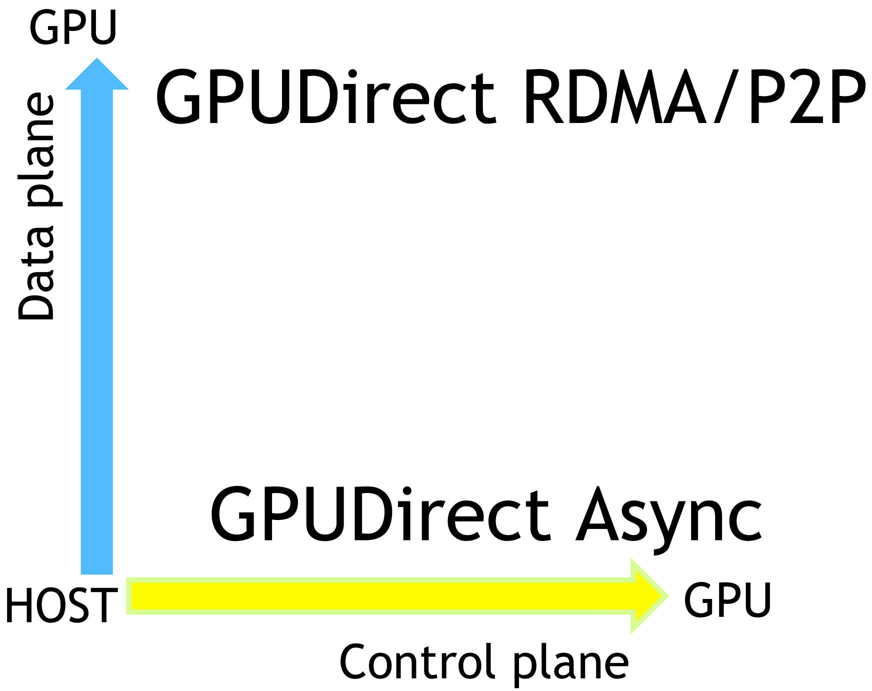
- GPUDirect RDMA & Async
  - over PCIe, for low latency
- GPUDirect P2P
  - over PCIe
  - over NVLink (Pascal+ only)



GPUDIRECT

Scopes (2)

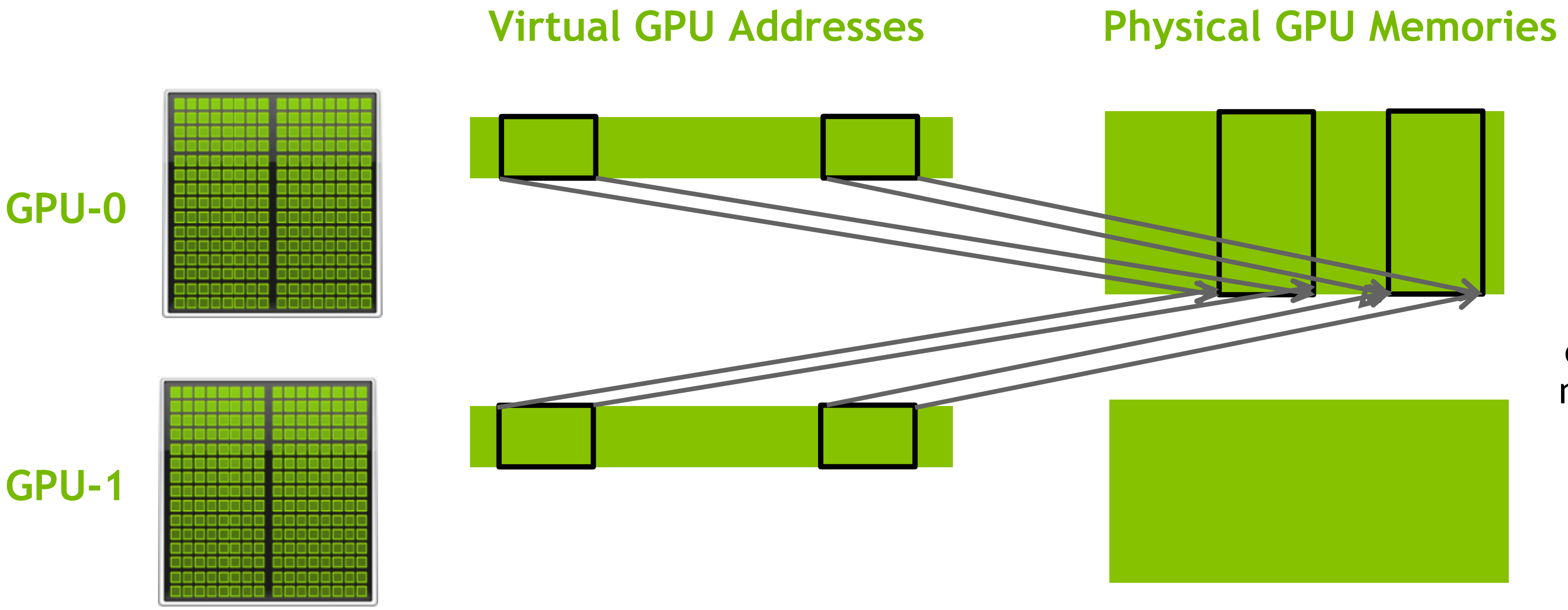
- GPUDirect P2P → data
  - GPUs both master and slave
- GPUDirect RDMA → data
  - GPU slave, 3rd party device master
- GPUDirect Async → control
  - GPU & 3rd party device master & slave



GPUDIRECT P2P

cudaDeviceEnablePeerAccess() mechanics

Using Allocations on Another GPU



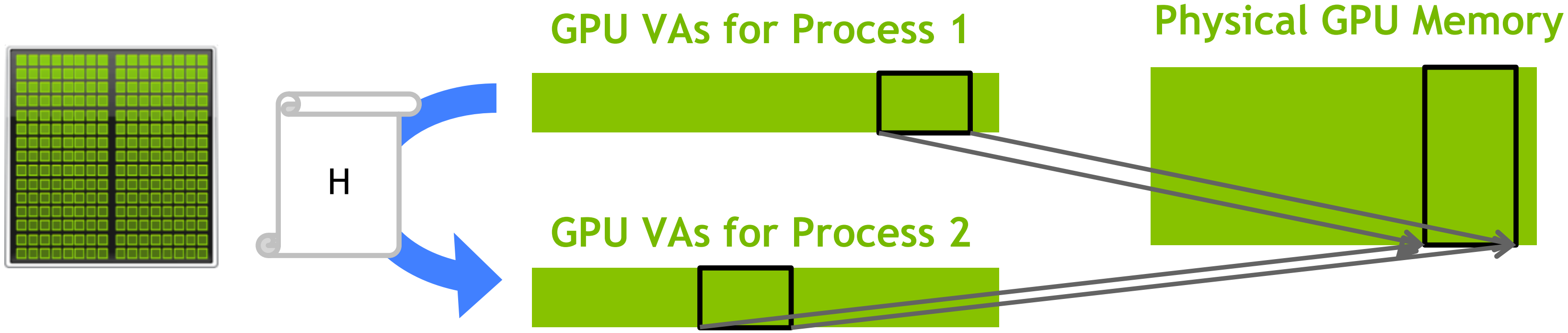
Existing GPU Allocations created through APIs like cudaMalloc

cudaDeviceEnablePeerAccess creates mapping for another GPU at the same GPU Vas

Use cudaMemcpy to move data between GPUs

cudaIpcGetMemHandle(), cudaIpcOpenMemHandle()

Using GPU memory from another process



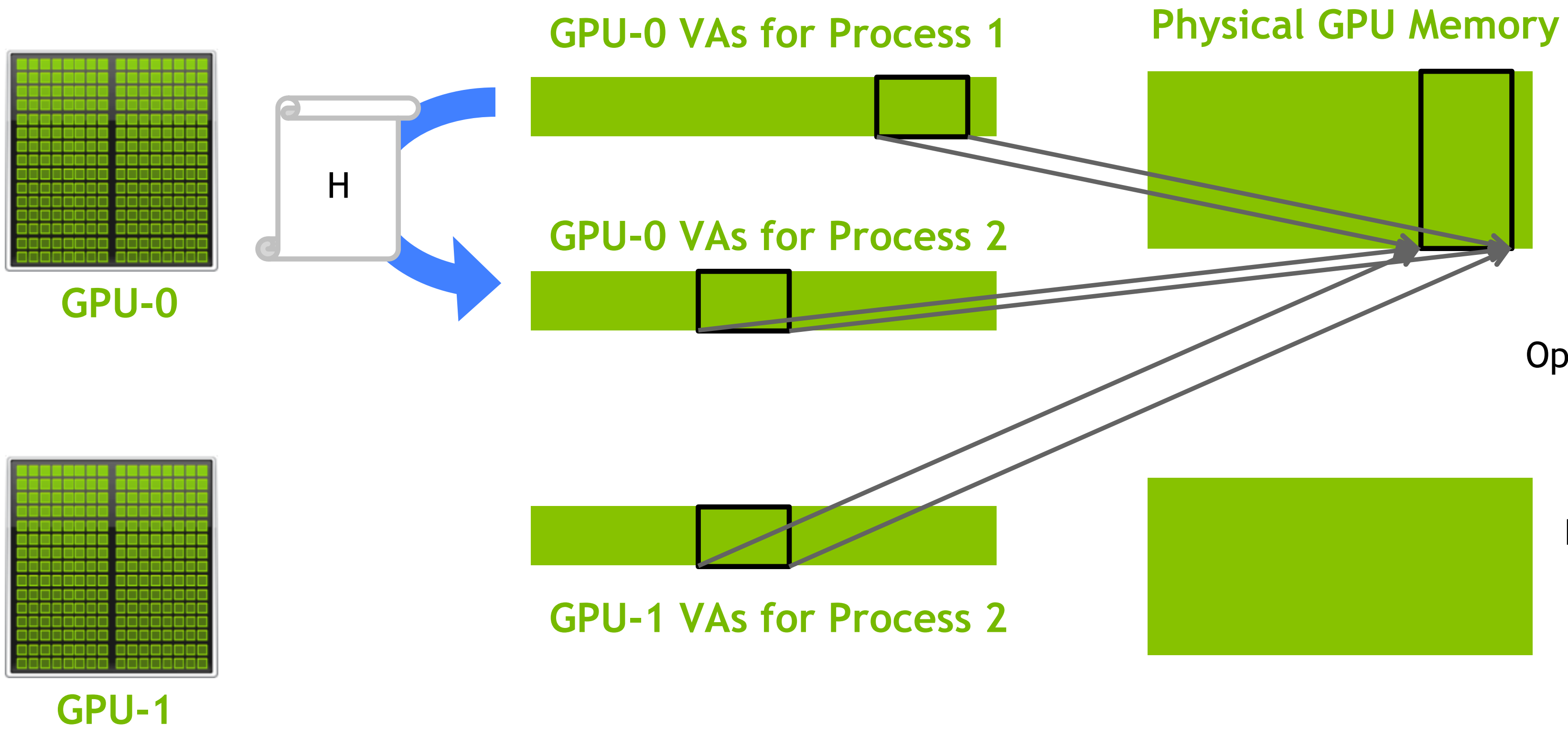
Existing allocation

Get handle and send to other process

Opening handle creates mapping in receiving process

cudaIpcGetMemHandle(), cudaIpcOpenMemHandle() & cudaDeviceEnablePeerAccess()

Using GPU memory from another process, **and from a second GPU**



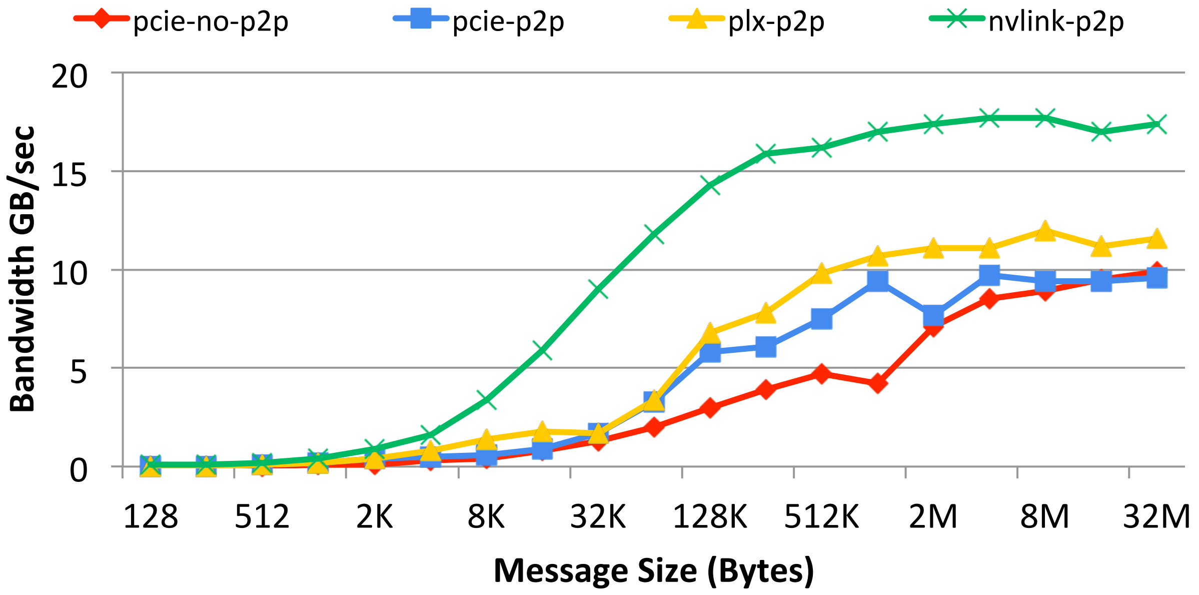
Existing allocation

Get handle and send to other process

Opening handle creates mapping in receiving process

For access by another process AND from a second GPU use peering

GPUDIRECT P2P Performance

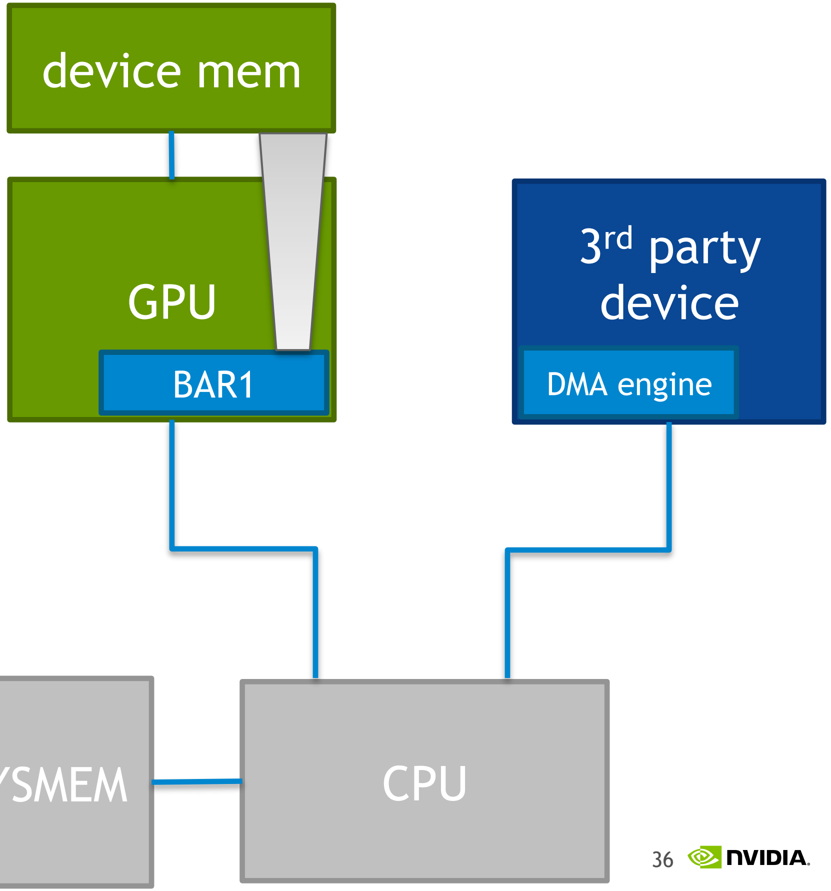


GPUDIRECT RDMA

GPUDirect RDMA

GPUDirect RDMA = PCIe BAR1

- exposing GPU memory (slave)
- used by 3rd party devices (master)
- since Kepler (K10,K20,K40..)
- potential users: network cards, storage devices, FPGAs



GPUDirect RDMA

for better network communication latency

nv_peer_mem plug-in module

- open-source Linux kernel module[1]
- enable NVIDIA GPUDirect RDMA on OpenFabrics stack
- remove cross-dependency

Multiple vendors

- Mellanox[2]: Connect-IB & ConnectX-4, IB/RoCE
- Chelsio[3]: T5, iWARP

[1] https://github.com/Mellanox/nv_peer_memory
[2] http://www.mellanox.com/page/products_dyn?product_family=116
[3] http://www.chelsio.com/gpudirect-rdma

RDMA, PLATFORMS AND TOPOLOGY

design your platform for RDMA!

Platforms supported by NVIDIA GPUs

- x86, e.g. SNB, IVB, HSW, BDW Xeon
- POWER8
- ARM64

RDMA performance varies[1]

- platform and topology matter, more later…

[1] https://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms

GPUDirect RDMA APIs

in CUDA 8.0

new NVIDIA kernel APIs:

```
typedef struct nvidia_p2p_dma_mapping {
    enum nvidia_p2p_page_size_type page_size_type;
    uint32_t  entries;
    uint64_t *dma_addresses;
} nvidia_p2p_dma_mapping;

int nvidia_p2p_dma_map_pages   (struct pci_dev *peer,
                                struct nvidia_p2p_page_table *page_table,
                                struct nvidia_p2p_dma_mapping **dma_mapping);
int nvidia_p2p_dma_unmap_pages (struct pci_dev *peer,
                                 struct nvidia_p2p_page_table *page_table,
                                 struct nvidia_p2p_dma_mapping *dma_mapping);
```

needed on some platforms

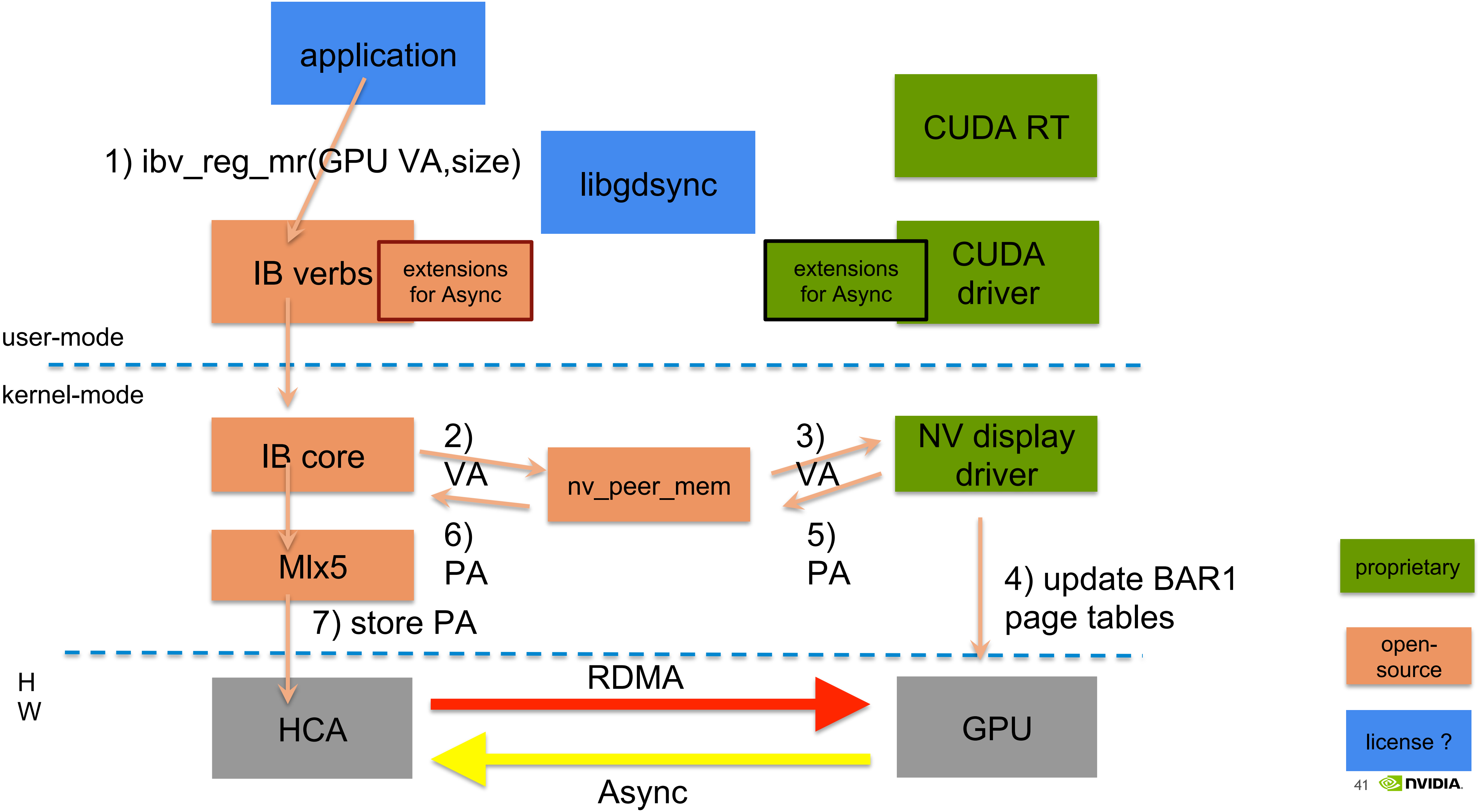
leveraged by nv_peer_memory plug-in module (update needed)

GPUDIRECT RDMA ON MELLANOX IB

NVIDIA, Mellanox partnership

- GPUDirect RDMA support
  - early prototype started in Feb 2013
  - presented at GTC 2013 on MVAPICH2 1.9
- rework for Mellanox SW stack
  - May 2013 - Dec 2013
  - showcased at SC’13 at Mellanox booth
  - shipping since 1/2014 MOFED 2.1
  - OpenMPI >=1.7.4, MVAPICH2 >=2.0

RDMA: MAPPING OF GPU MEMORY ON HCA



application

1) ibv_reg_mr(GPU VA,size)

libgdsync

CUDA RT

IB verbs

extensions for Async

extensions for Async

CUDA driver

user-mode

kernel-mode

IB core

2) VA

nv_peer_mem

3) VA

NV display driver

6) PA

5) PA

Mlx5

4) update BAR1 page tables

7) store PA

HW

HCA

RDMA

Async

GPU

proprietary

open-source

license ?

GPUDIRECT RDMA + INFINIBAND + CUDA-AWARE MPI

Less overhead = lower latency and more effective bandwidth

When GDRDMA is effective

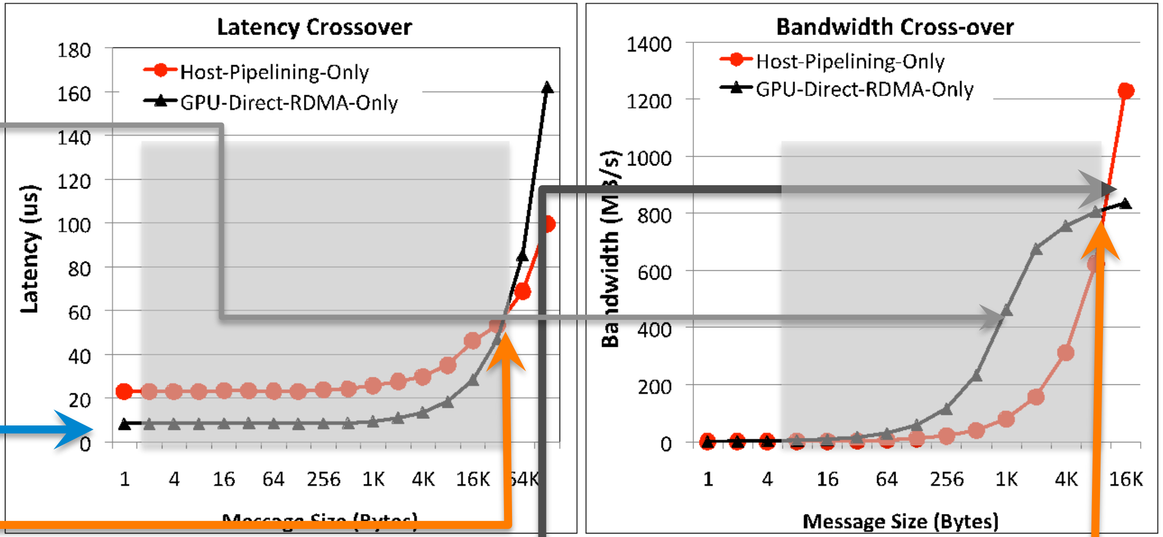
- 3x more BW
- ~70% less latency

Not always good:

- Experimentally, cross-over behavior at 8-16KB
- GDRDMA reading BW cap at 800MB/s on SNB
- GDRDMA writing BW saturates.

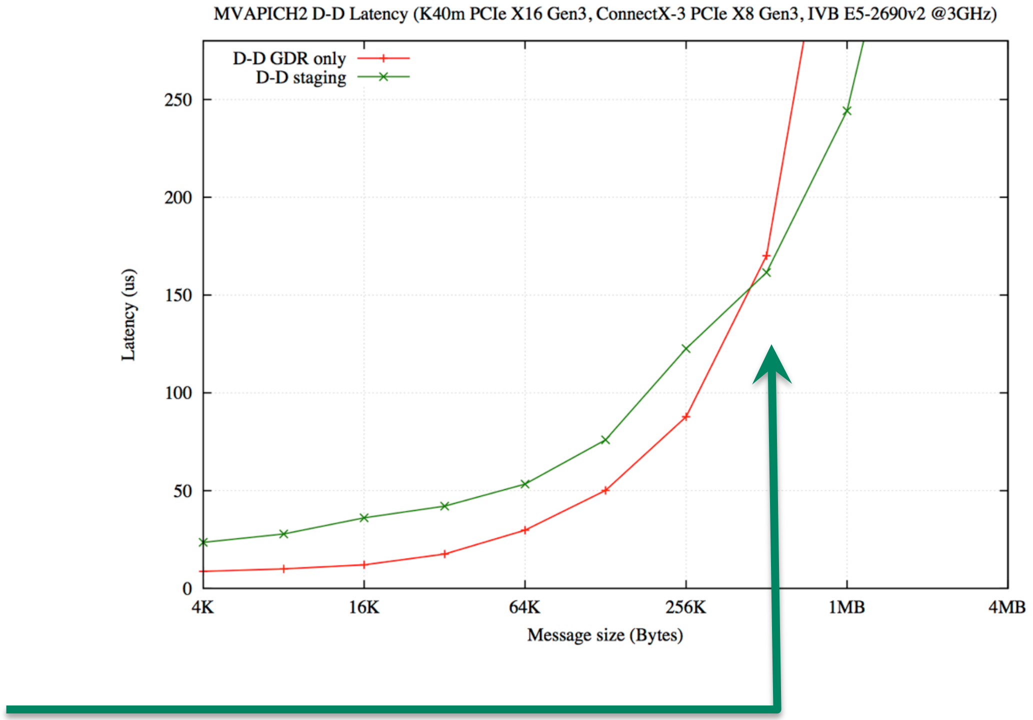
Preliminary Performance of MVAPICH2 with GPU-Direct-RDMA

Cross-over Between GPU-Direct RDMA and Host Pipelining

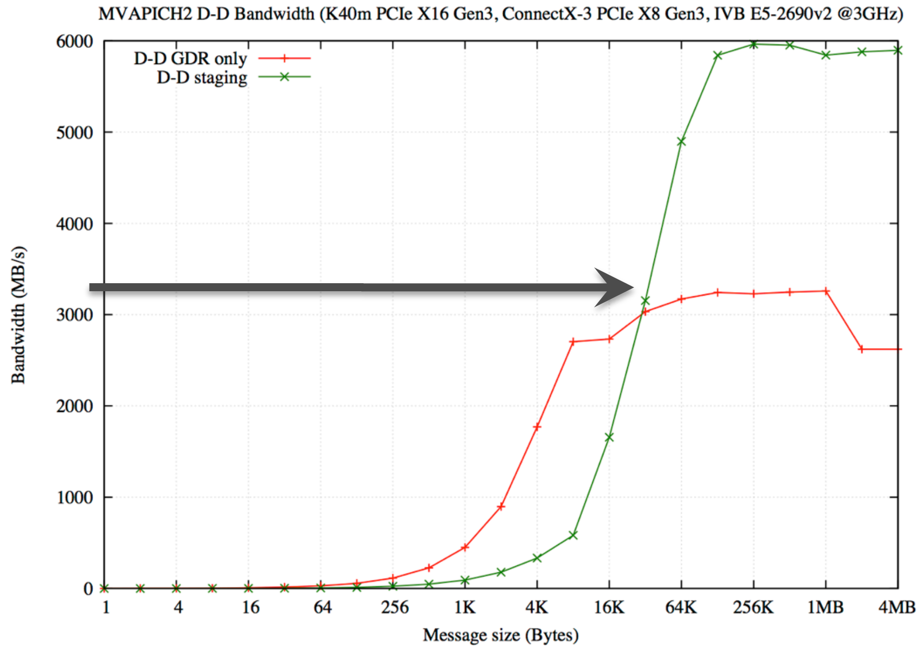


2 nodes MPI benchmarks
with and without GPUDirect RDMA
Intel SNB, K20c, MLNX FDR PCIe X8 gen3

ON IVY BRIDGE XEON



MVAPICH2 D-D Latency (K40m PCIe X16 Gen3, ConnectX-3 PCIe X8 Gen3, IVB E5-2690v2 @3GHz)

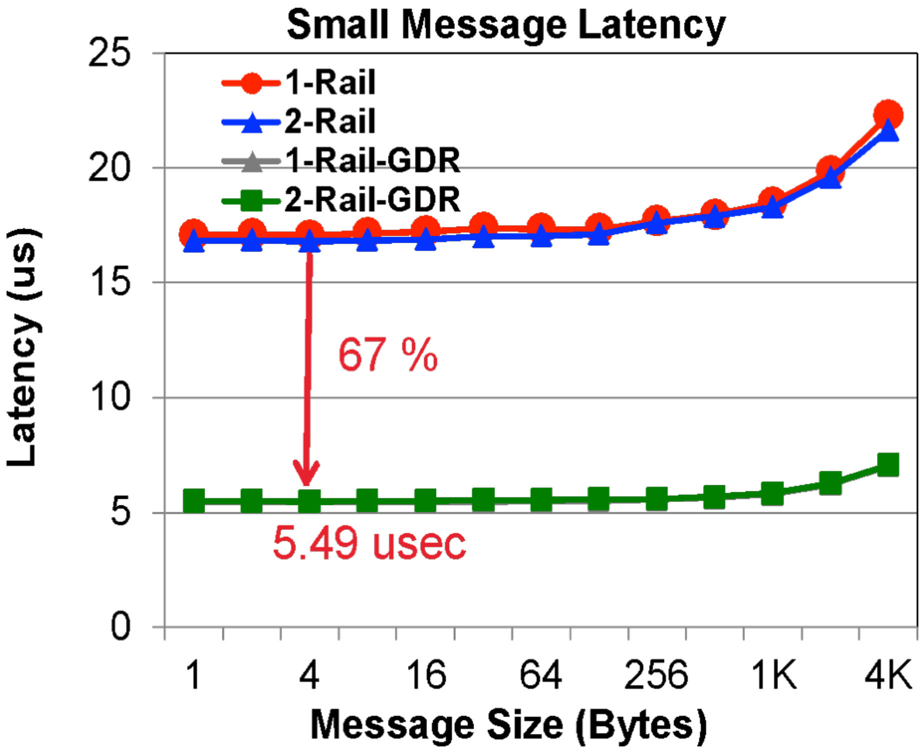


MVAPICH2 D-D Bandwidth (K40m PCIe X16 Gen3, ConnectX-3 PCIe X8 Gen3, IVB E5-2690v2 @3GHz)

MPI USING GPUDIRECT RDMA

back in GTC14 time…

any space for improvement ?



PLATFORM LIMITATIONS

Platform-related limitations

PCIE topology and NUMA effects:

Number and type of traversed chipsets/bridges/switches

- GPU memory reading BW most affected
  - Sandy Bridge Xeon severely limited
- On older chipsets, writing BW affected too

Crossing inter-CPU bus huge bottleneck

- currently allowed though

NUMA effects and PCIE topology

- GPU BAR1 reading can be half PCIE peak, by design
  - **Fixed in Pascal**

- GPUDirect RDMA performance may suffer from bottlenecks
  - GPU memory reading most affected
  - old chipsets, writing too
  - IVB Xeon PCIE much better

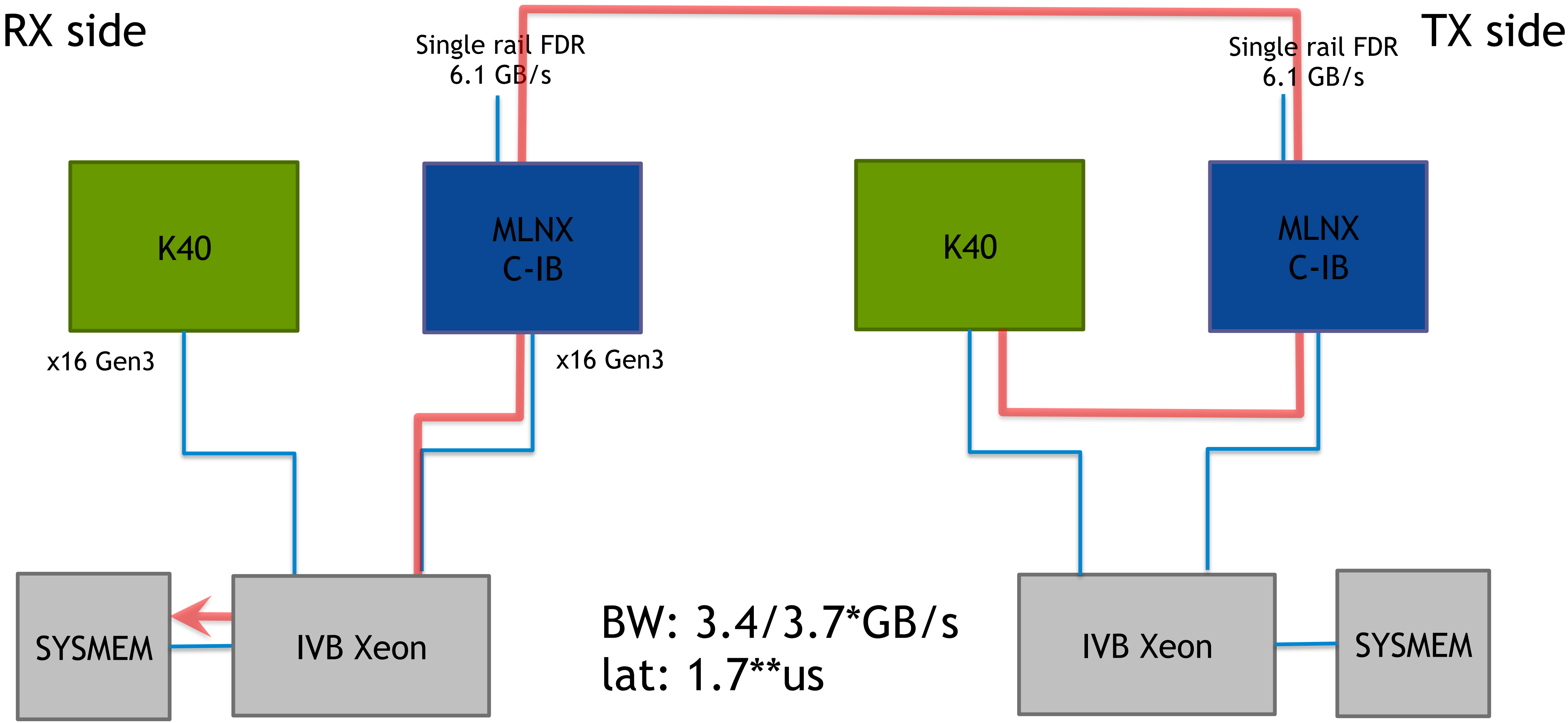
MPI strategy

- use GPUDirectRDMA for small/mid size buffers
  - threshold depends on platform
  - and on NUMA (e.g. crossing QPI)
- don’t tax BAR1 too much
  - bookkeeping needed [being implemented]
- revert to pipelined staging through SYSMEM
  - GPU CE latency >= 6us
- no Eager-send for G-to-G/H
  - excessive inter-node sync
  - use MPI3 one-sided in apps

Platform & Benchmarks

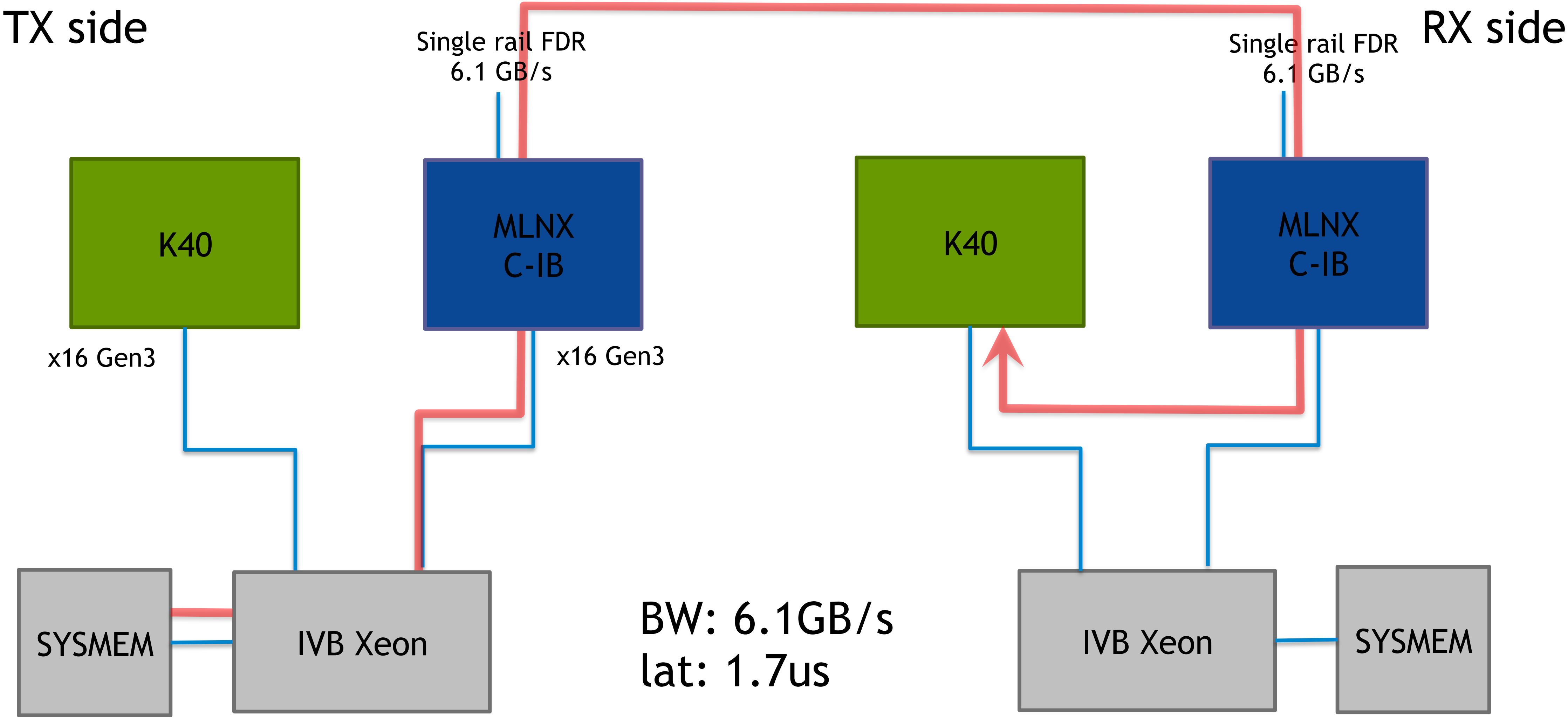
- SMC 2U SYS-2027GR-TRFH
  - IVB Xeon
  - PEX 8747 PCIE switch, on a raiser-card
- SMC 1U SYS-1027GR-TRF
- NVIDIA K40m
- Mellanox dual-port Connect-IB
- Benchmarks:
  - GPU-extended ibv_ud_pingpong
  - GPU-extended ib_rdma_bw

GPU to Host (Ivy Bridge)



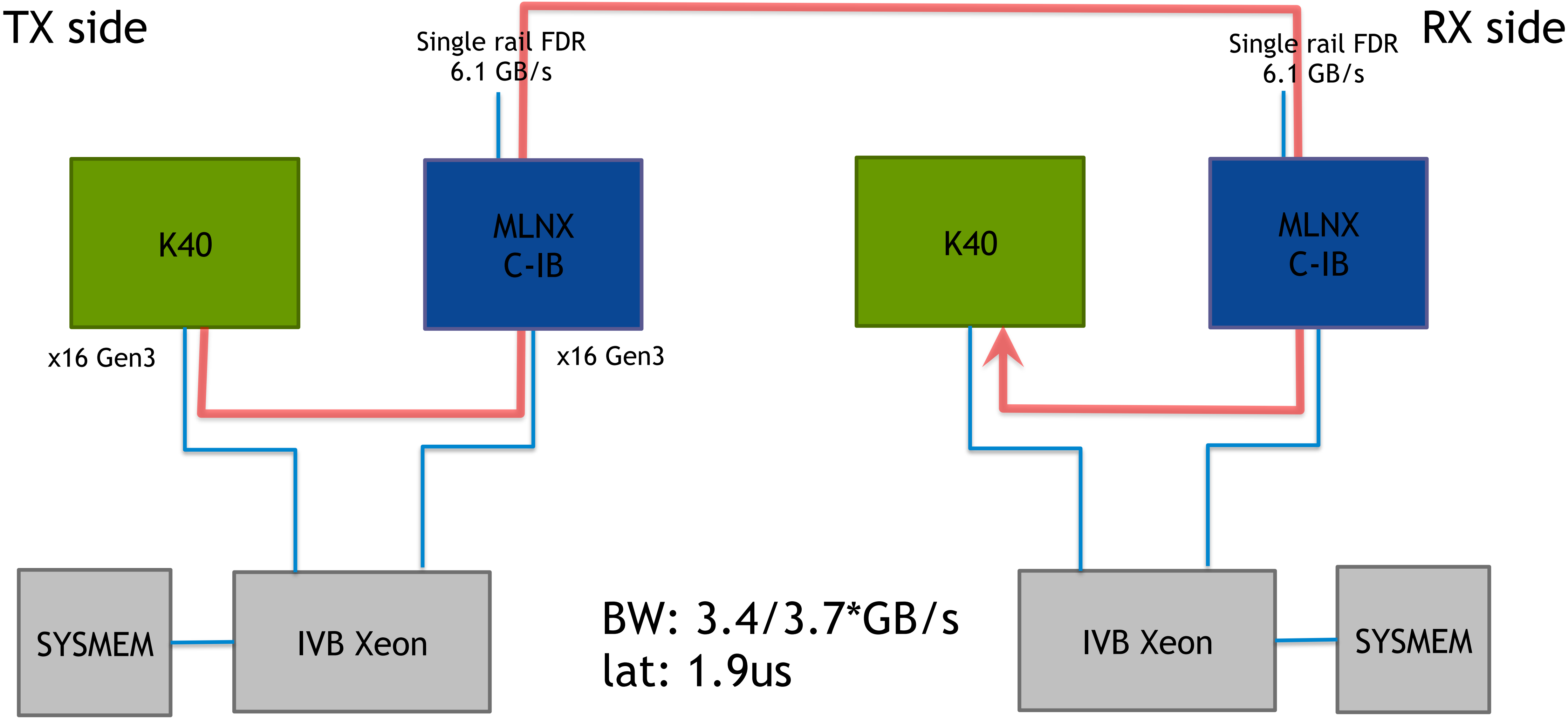
BW: 3.4/3.7*GB/s
lat: 1.7**us

Host to GPU (Ivy Bridge)



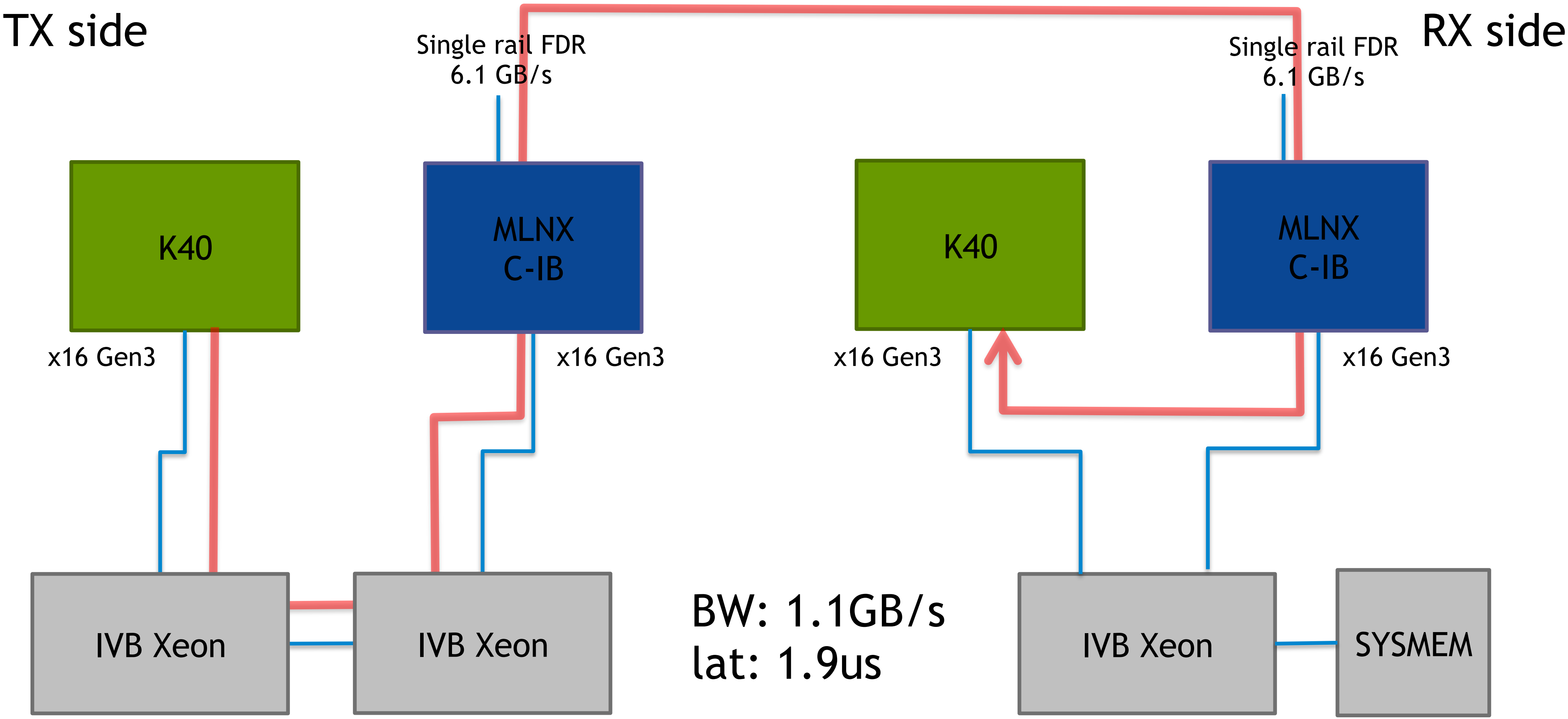
BW: 6.1GB/s
lat: 1.7us

GPU to GPU (Ivy Bridge)



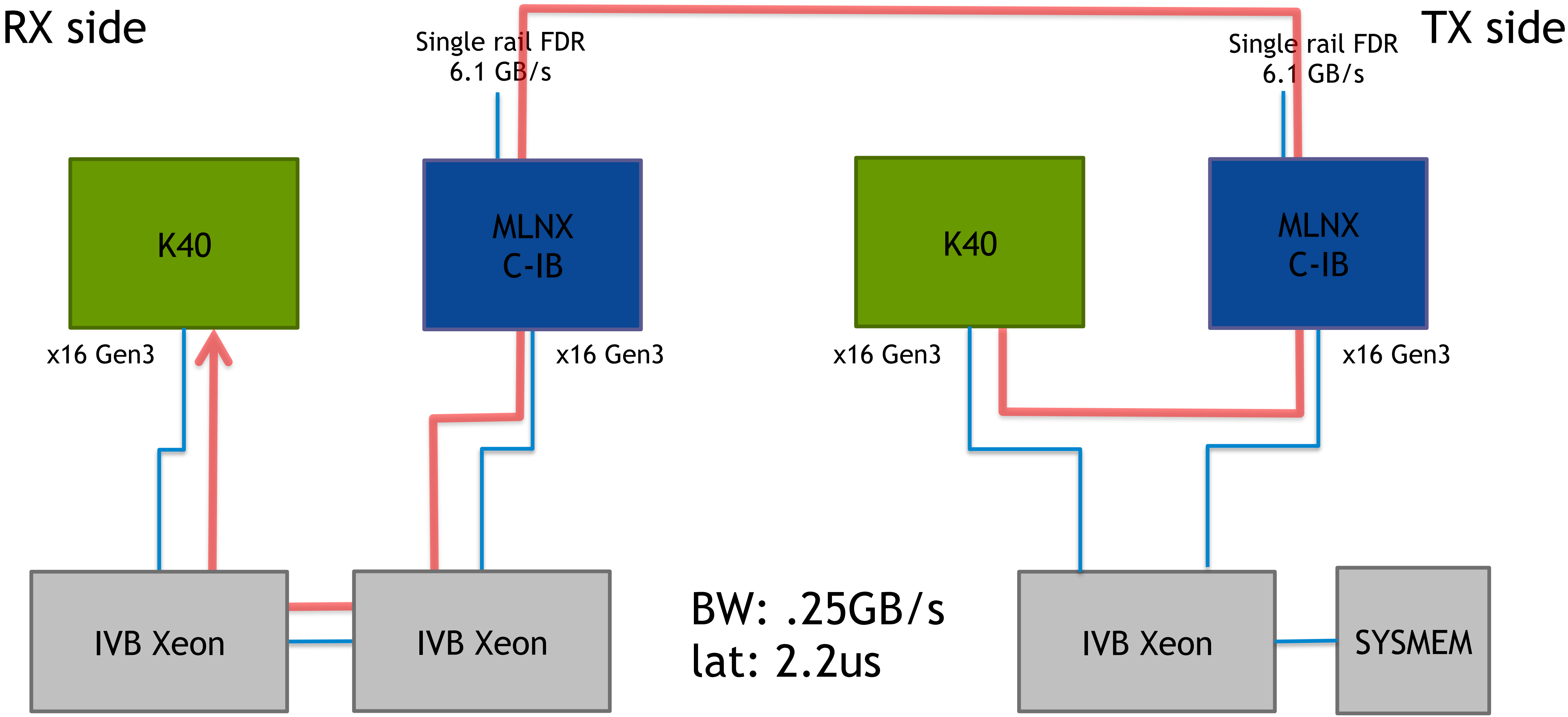
BW: 3.4/3.7*GB/s
lat: 1.9us

GPU to GPU, TX across QPI (Ivy Bridge)



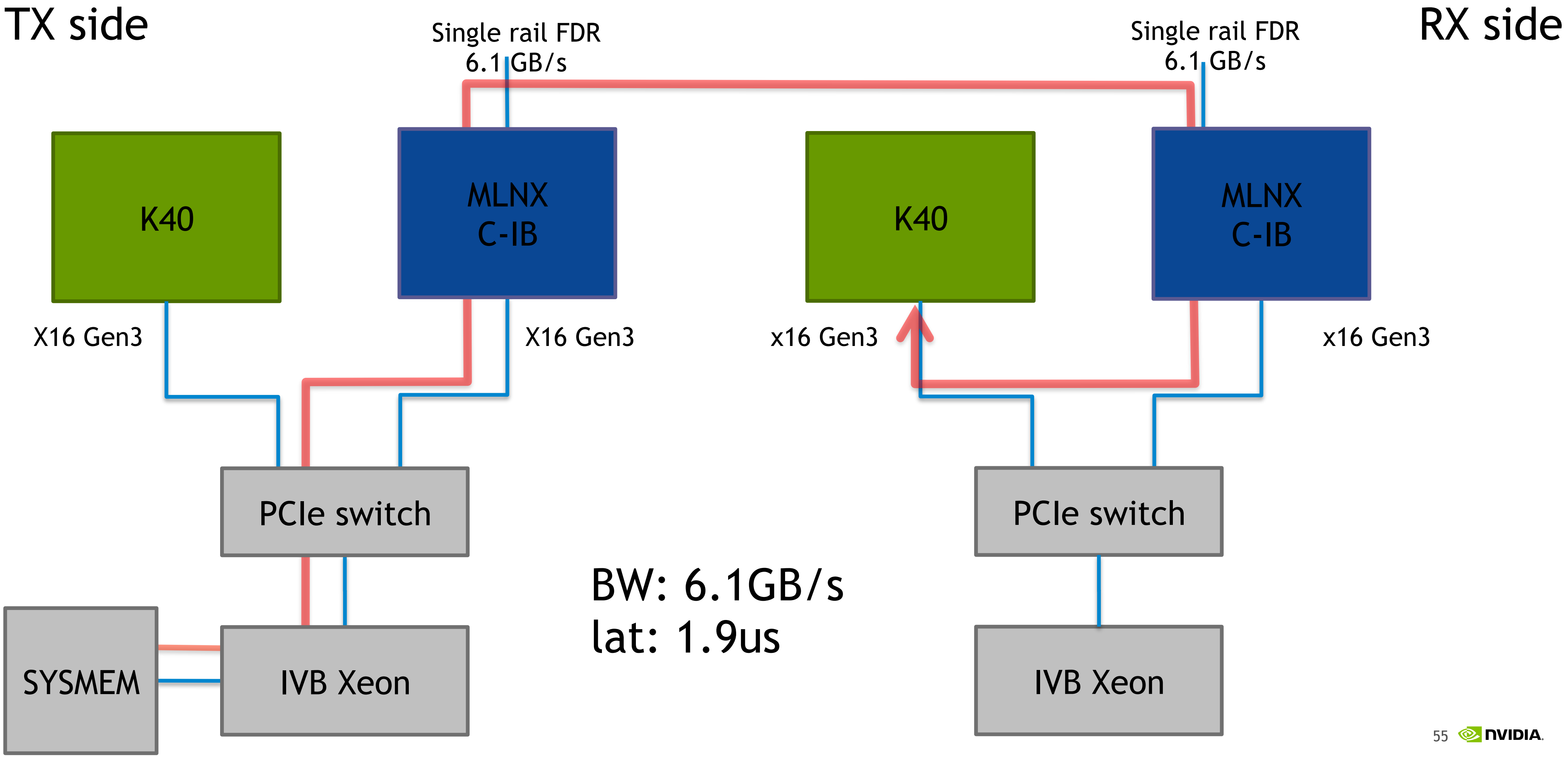
BW: 1.1GB/s
lat: 1.9us

GPU to GPU, RX across QPI (Ivy Bridge)



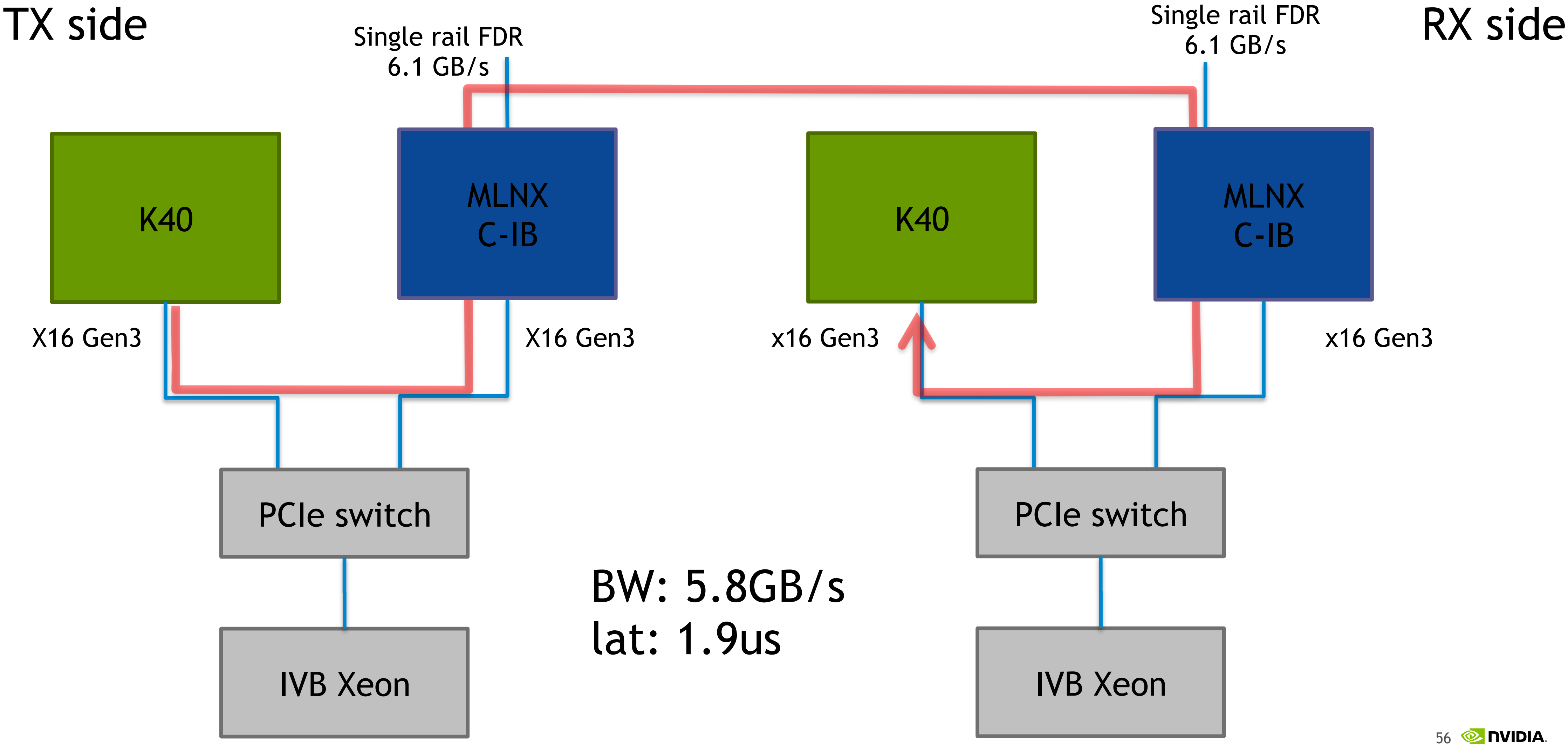
BW: .25GB/s
lat: 2.2us

GPU to Host, RX across PCIe switch (Ivy Bridge)



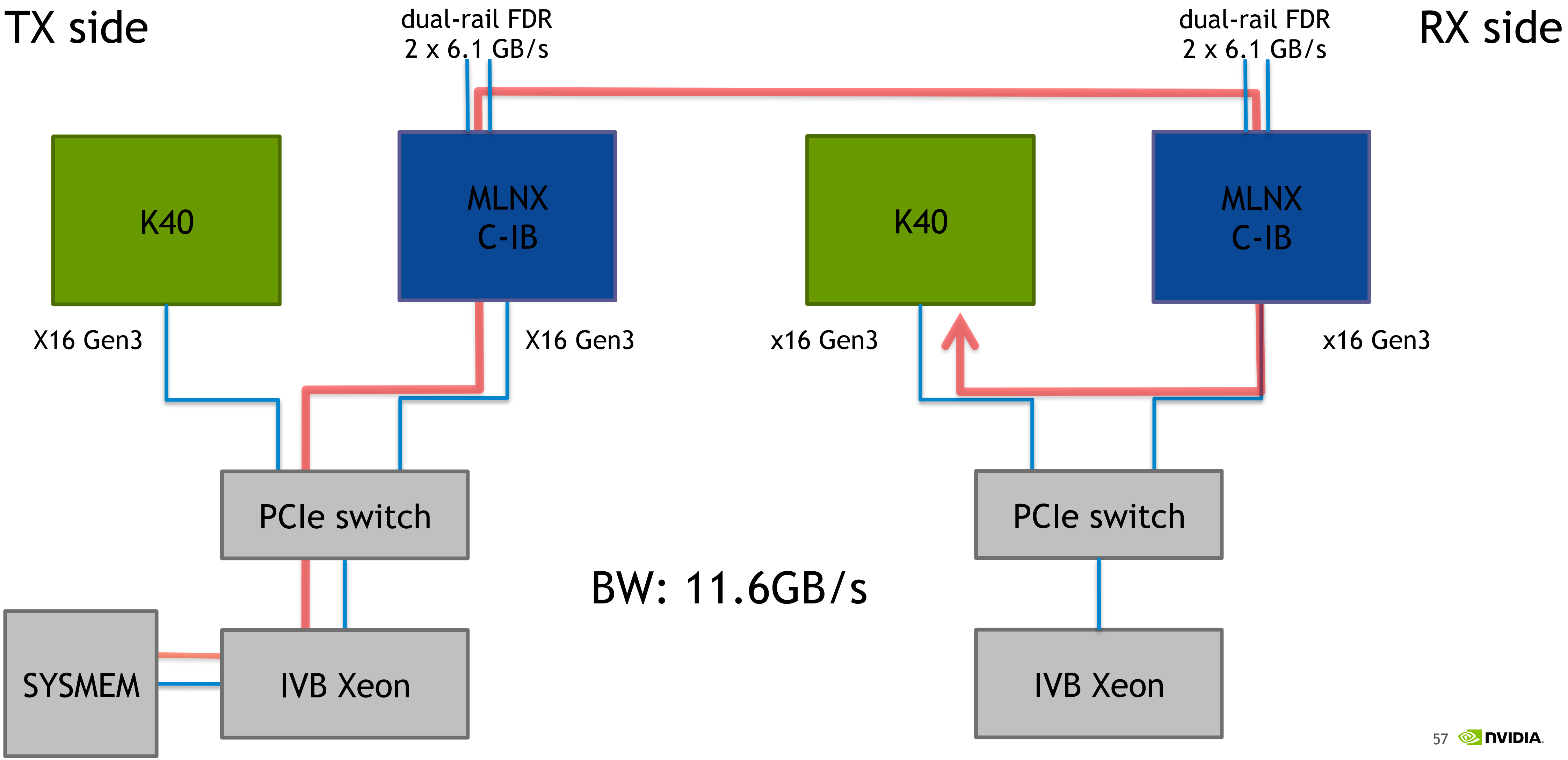
BW: 6.1GB/s
lat: 1.9us

GPU to GPU, TX/RX across PCIe switch (Ivy Bridge)



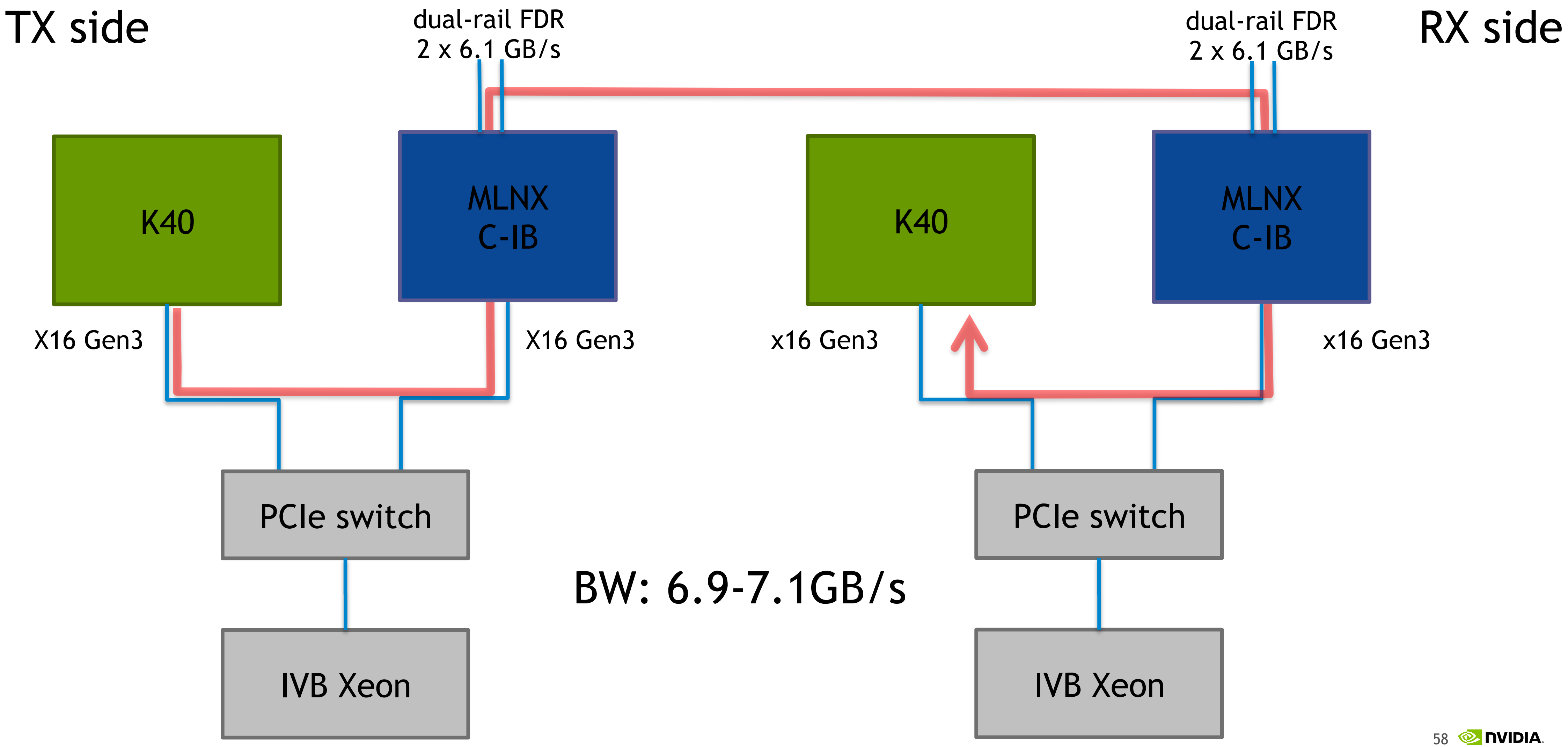
BW: 5.8GB/s
lat: 1.9us

Host to GPU, RX across PCIe switch, dual-rail (Ivy Bridge)



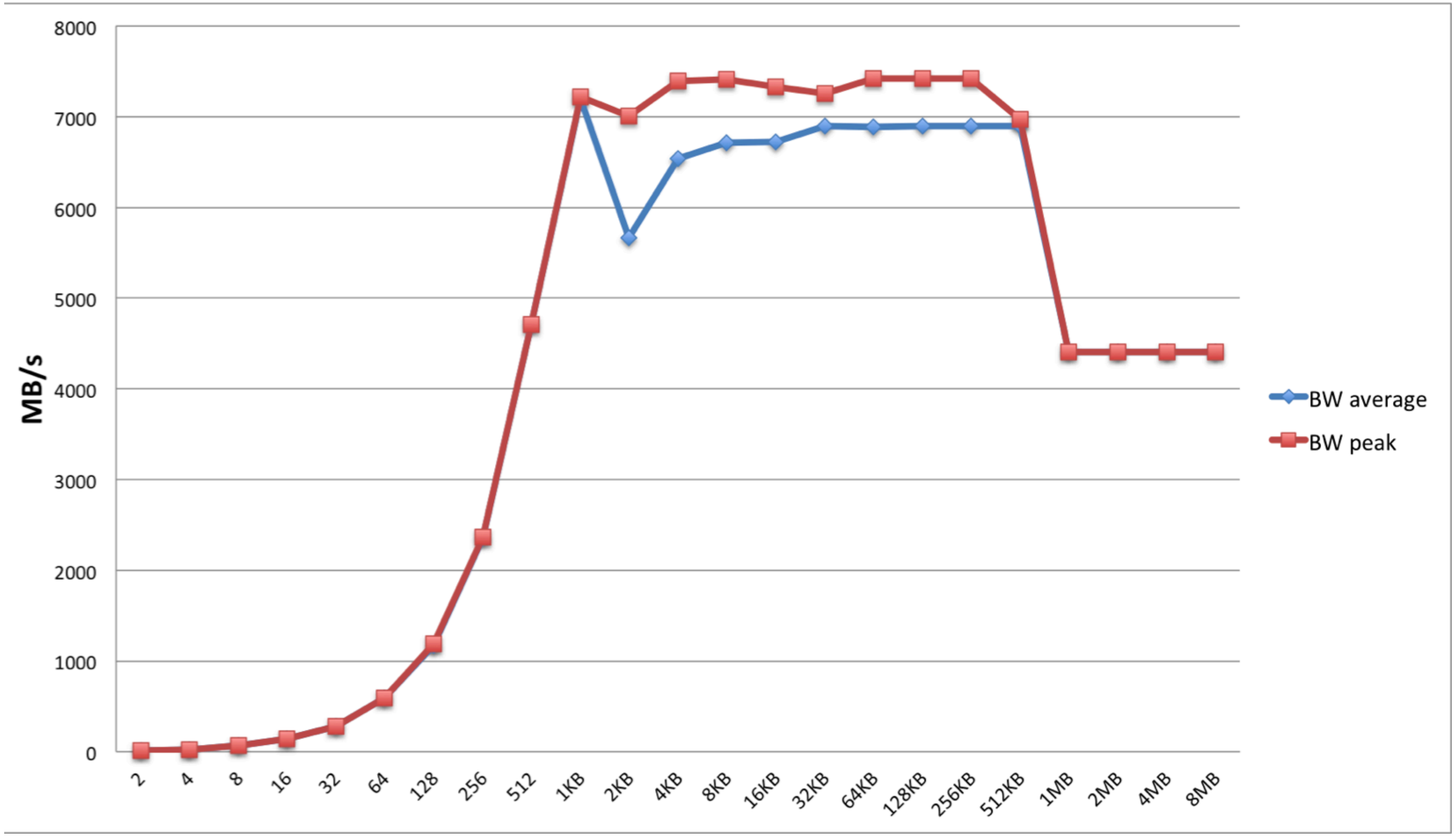
BW: 11.6GB/s

GPU to GPU, TX/RX across PCIe switch, dual-rail, (Ivy Bridge)



BW: 6.9-7.1GB/s

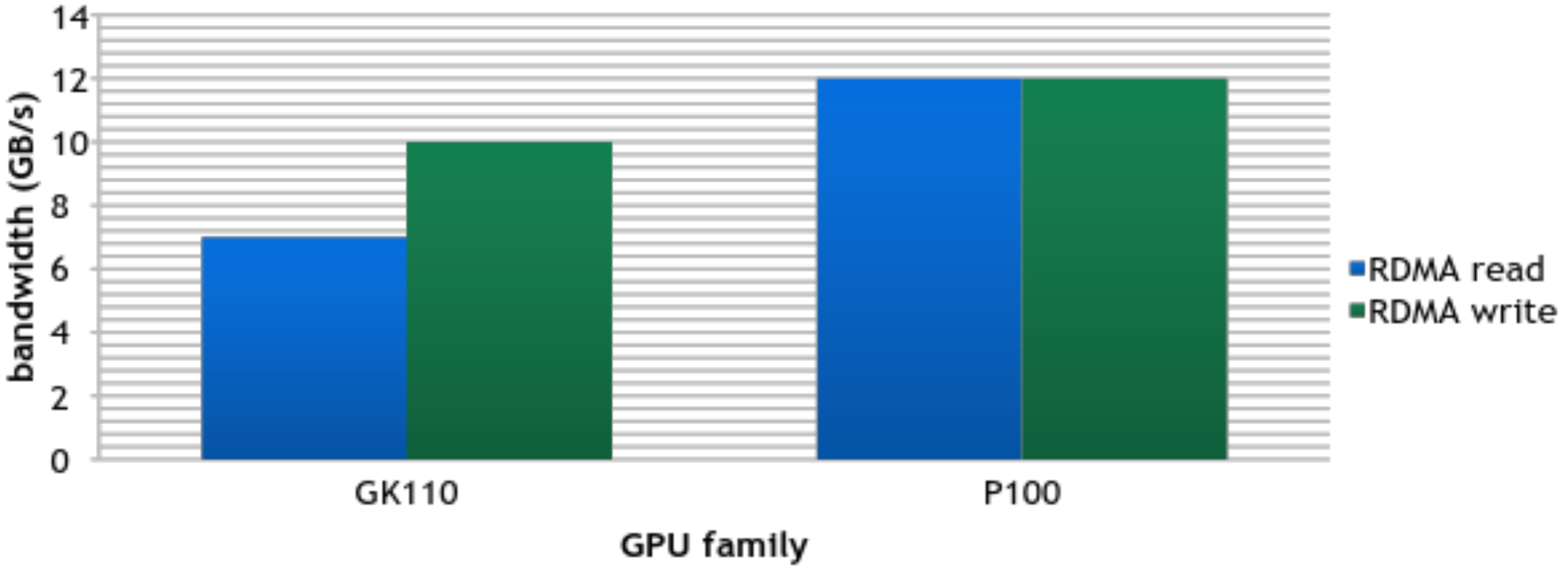
GPU to GPU, TX/RX across PCIe switch, dual-rail



- ib_write_bw bandwidth test
- K40m at boost clock
- dual-rail FDR HCA
- two nodes with PCIe switch
- 5000 iterations at varying message size

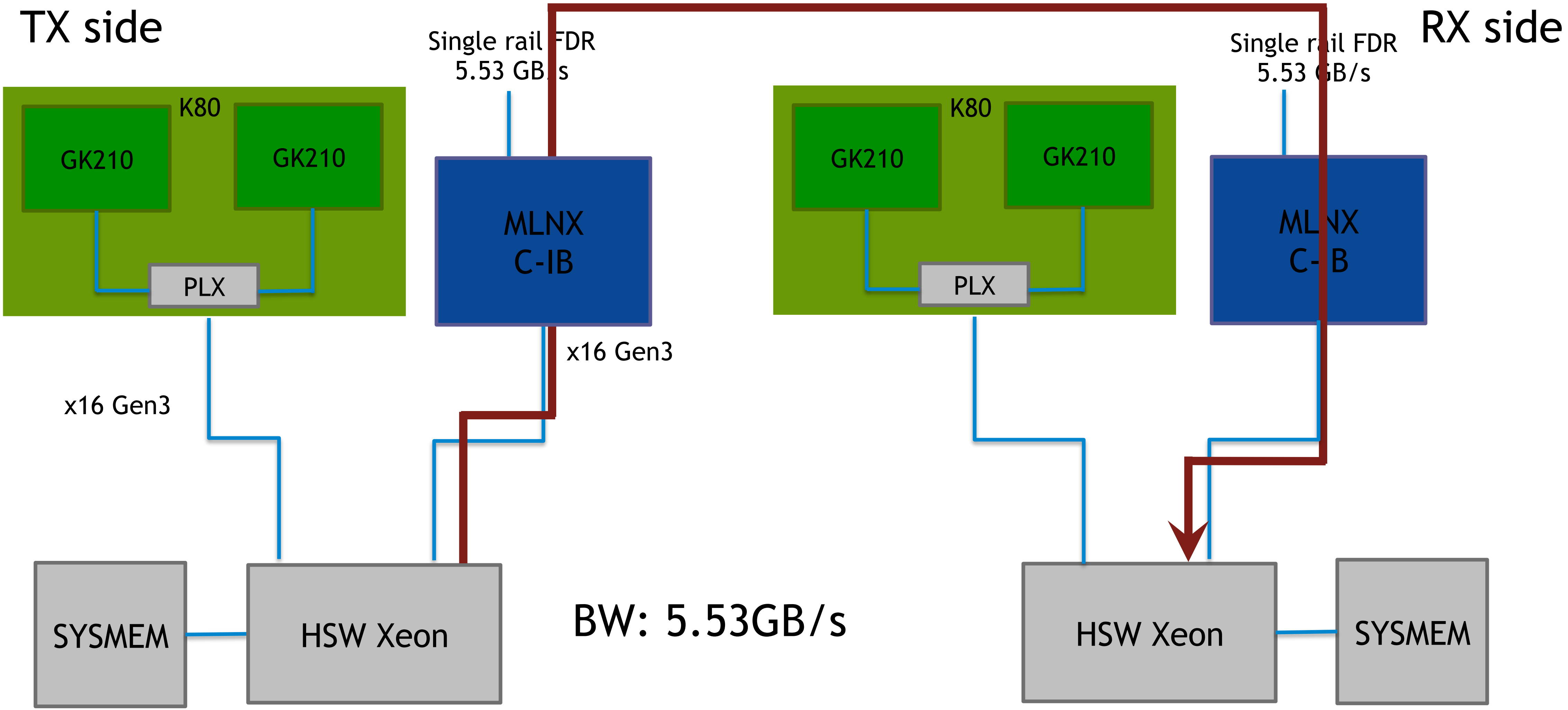
GPUDIRECT RDMA on Pascal

HW improved, still need proper topology



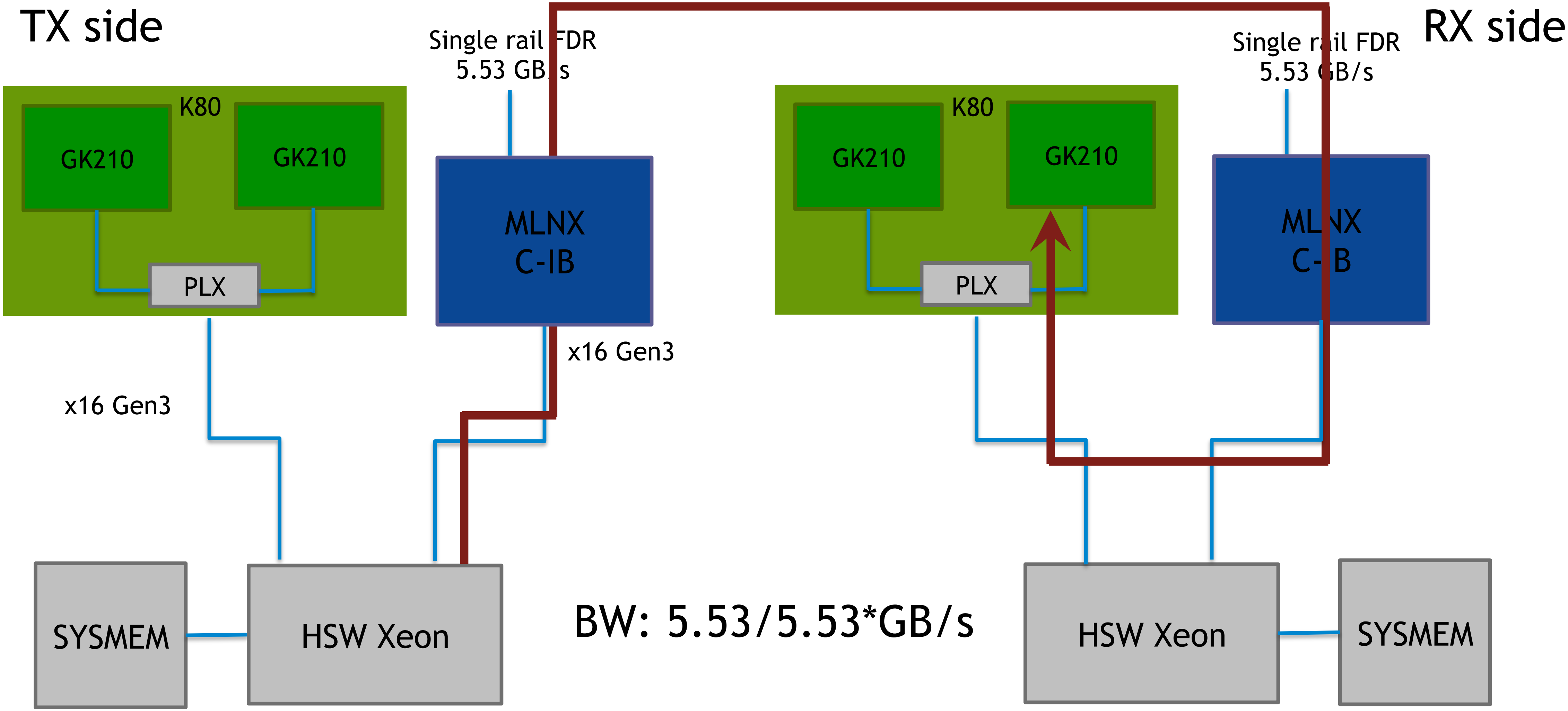
[*] peak bandwidth, optimal PCIe fabric

Host to Host (Haswell)



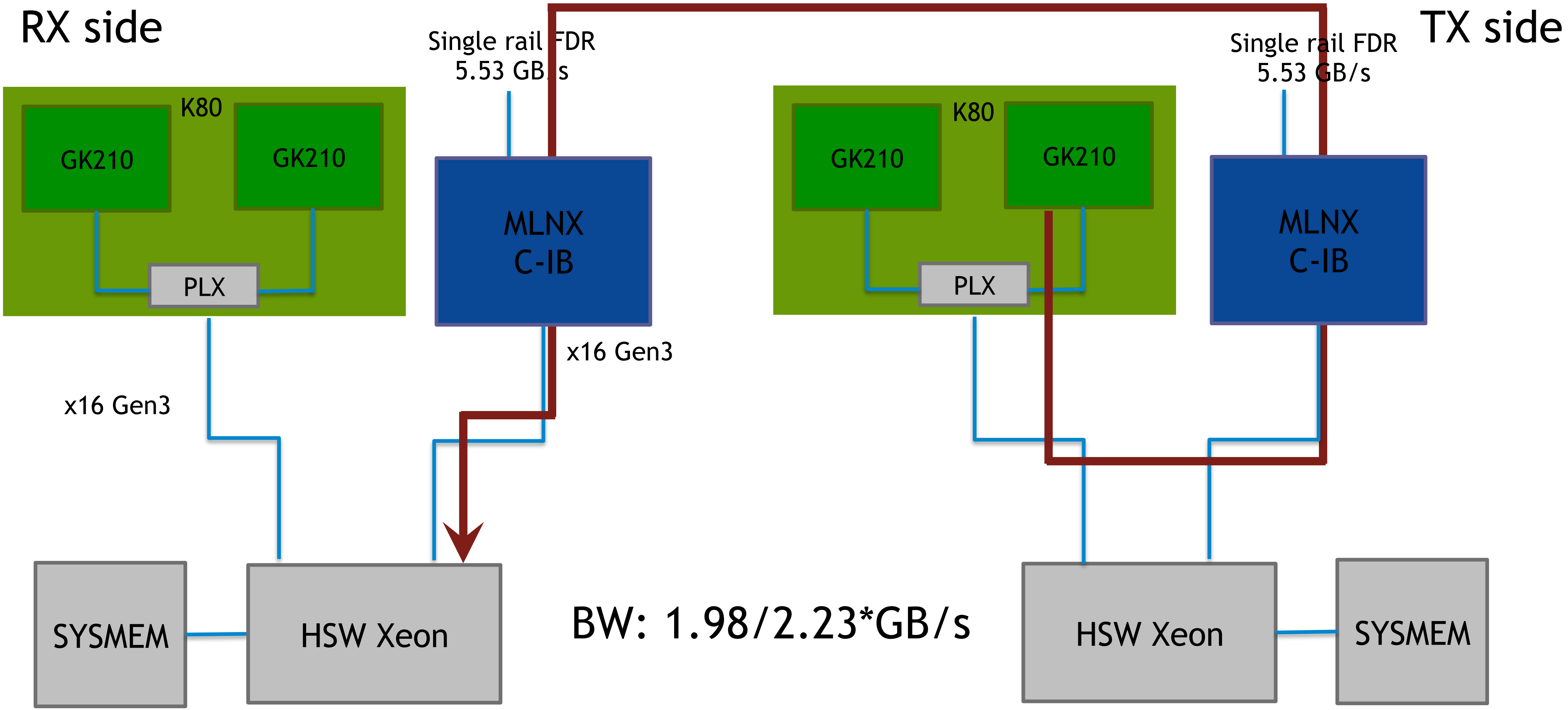
BW: 5.53GB/s

Host to GPU (Haswell)



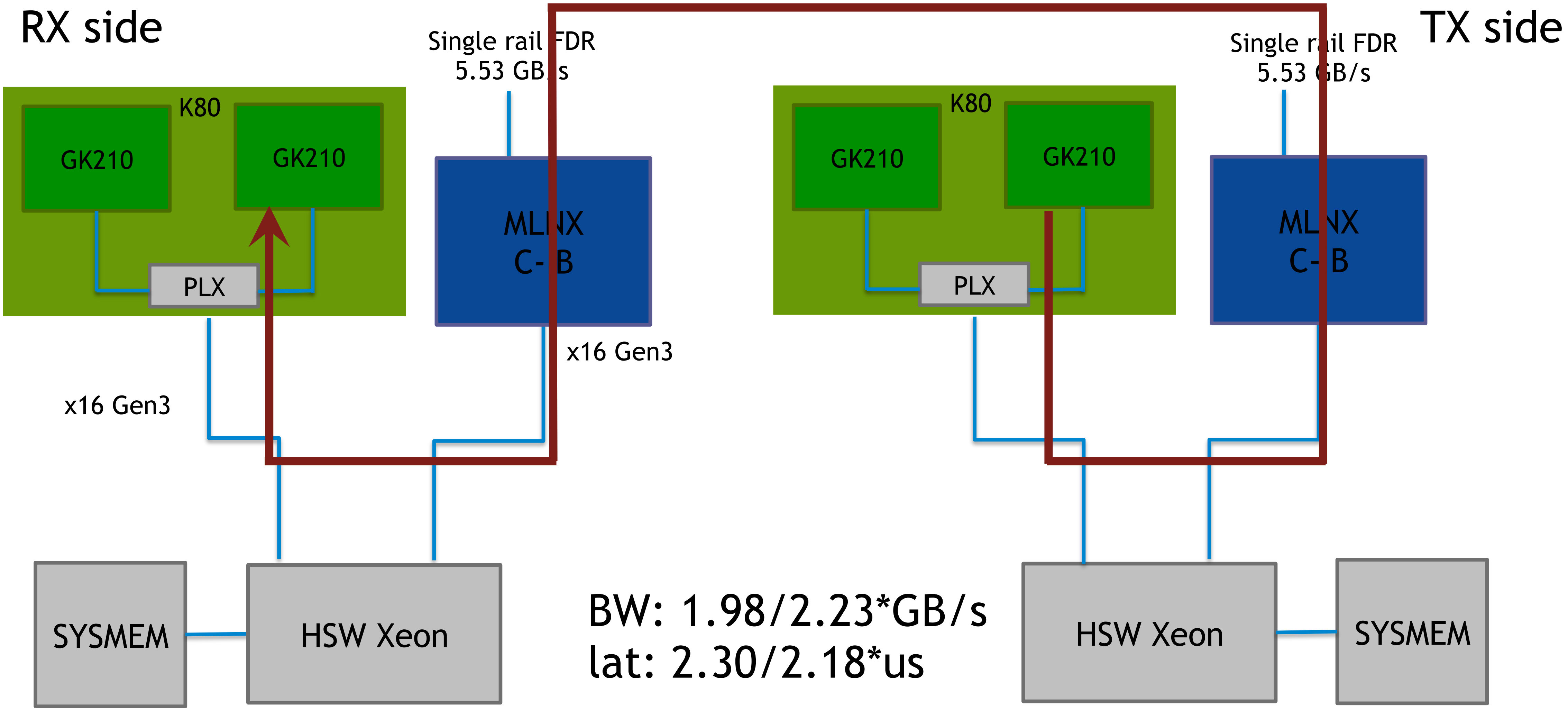
BW: 5.53/5.53*GB/s

GPU to Host (Haswell)



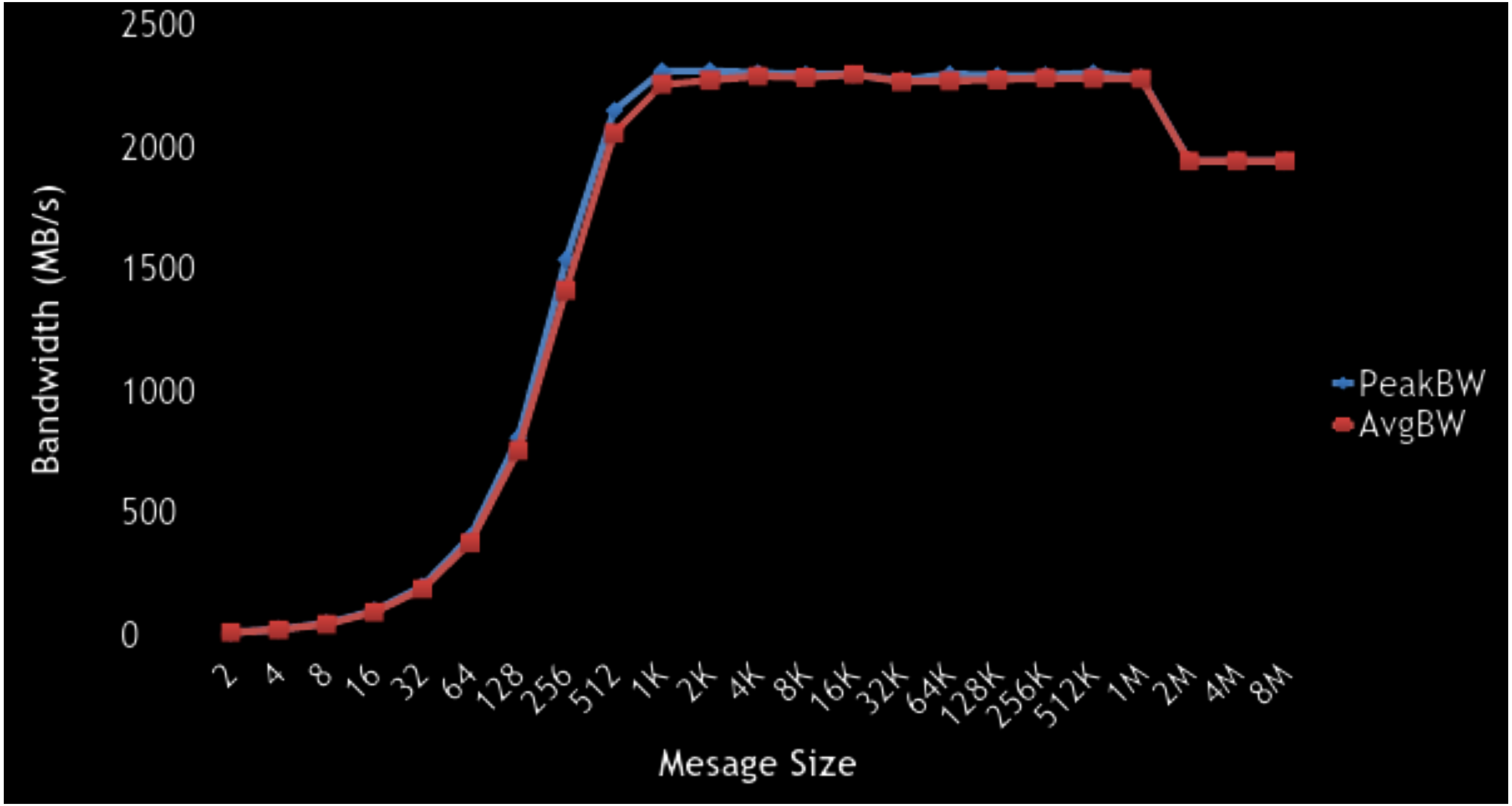
BW: 1.98/2.23*GB/s

GPU to GPU (Haswell)

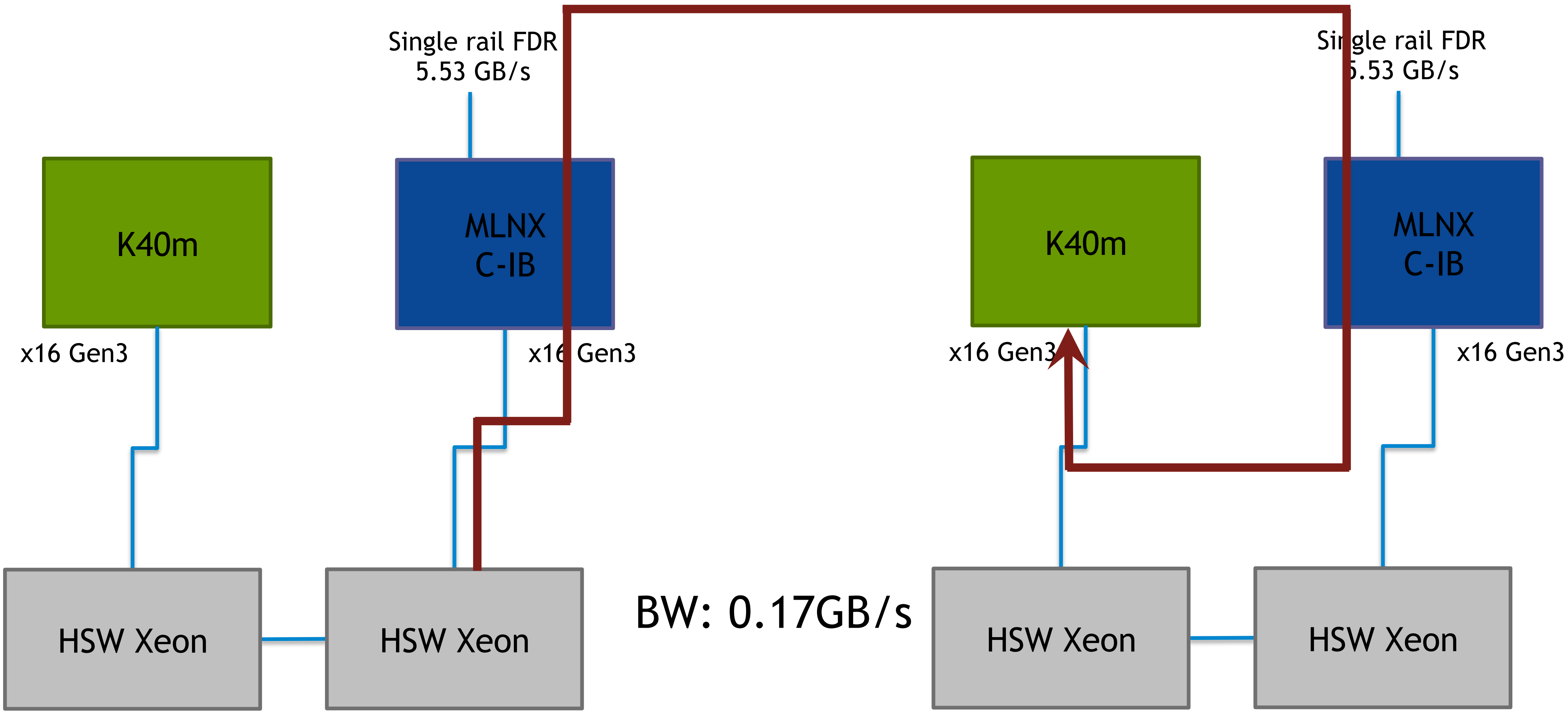


BW: 1.98/2.23*GB/s
lat: 2.30/2.18*us

GPU to GPU (Haswell)

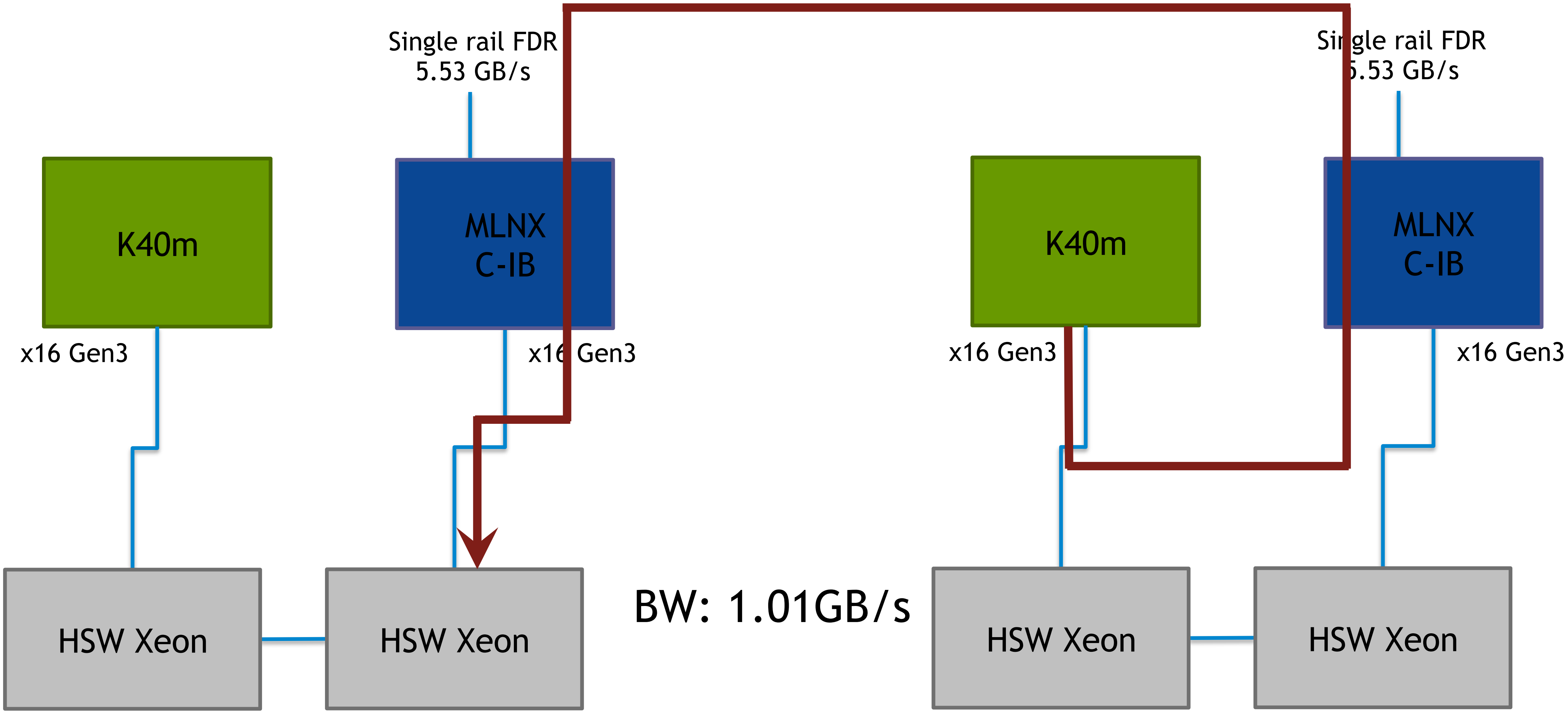


Host to GPU, across QPI (Haswell)



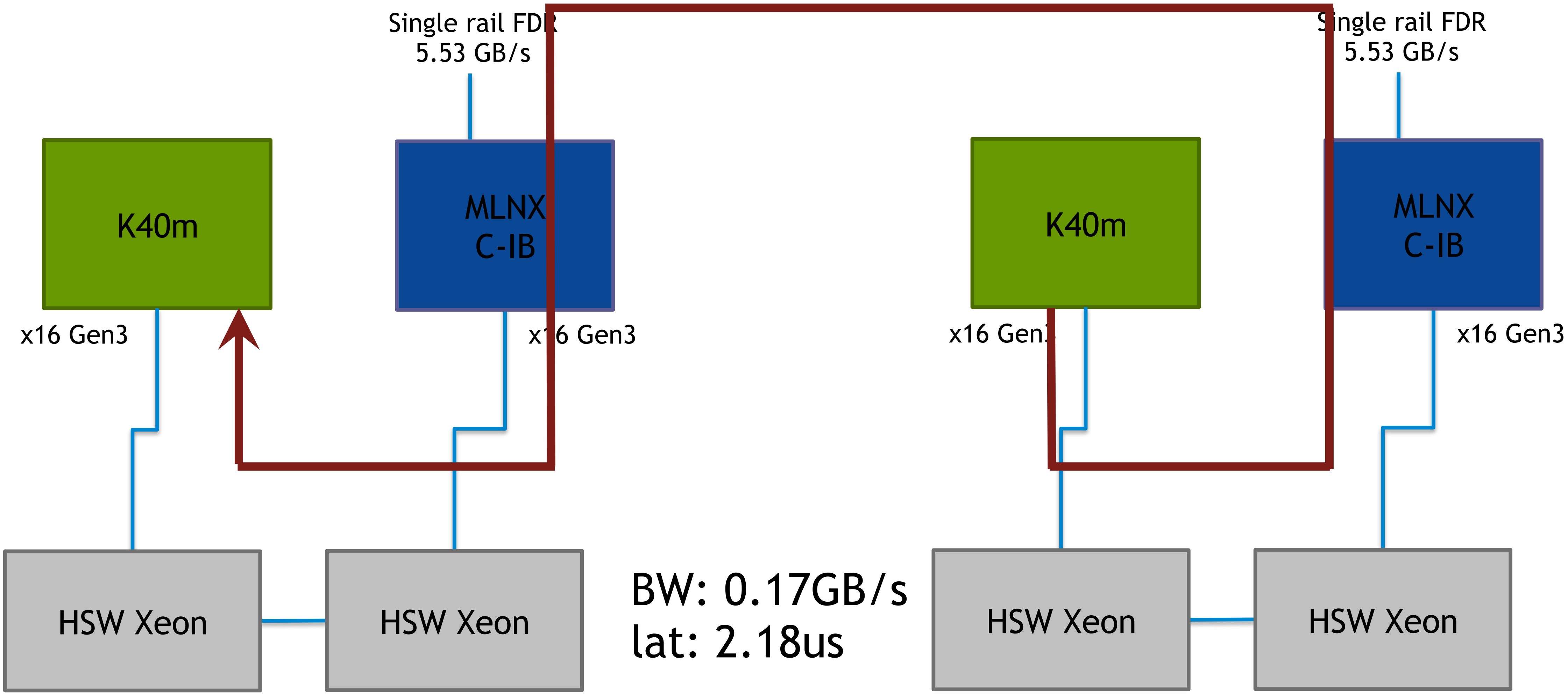
BW: 0.17GB/s

GPU to Host, across QPI (Haswell)



BW: 1.01GB/s

GPU to GPU, across QPI (Haswell)



BW: 0.17GB/s
lat: 2.18us

GDRCOPY

GDRCOPY

(ab)use CPU to copy data to/from GPU mem

Available on https://github.com/NVIDIA/gdrcopy

What: uncached user-space mappings of GPU memory

- copy library, super low-latency

How: three sets of primitives:

- Pin/Unpin, setup/tear down BAR1 mappings of GPU memory
- Map/Unmap, memory-map BAR1 on user-space CPU address range
  - CPU can use standard load/store instructions (MMIO) to access the GPU memory.
- Copy to/from PCIe BAR, highly tuned R/W functions

GDRCOPY

performance

.3us write latency vs ~1us for loopback vs 4-6us for cudaMemcpy

D-H: 20-30MB/s H-D: 6GB/s vs 9GB/s for cudaMemcpy

- sensitive to CPU MMIO performance
    - NUMA effects, eg on IVB H-D=3GB/s when on wrong socket
- AMO (atomics) not supported
- no automatic I/O coherency on GPU (SW enforced PoC)

RESTORING EAGER PROTOCOL FOR GPU MEMORY

RESTORING EAGER

recalling Eager protocol for point-to-point MPI privitives

Eager protocol on IB:

sender:

- copy on pre-registered TX buffers
- ibv_post_send, on UD,UC,RC,…

receiver:

- pre-post temp RX buffers, use credits for proper bookkeeping
- RX matching then copy to final destination (CPU or GPU memory)

RESTORING EAGER

Eager essential tool for low-latency / small msgs

get rid of rendezvous as:

bad interplay with apps

excessive sync among nodes

more round-trip’s

Problem:

staging: moving data from host temp buf to GPU final dest

- cudaMemcpy has >7us overhead (~40KB threshold at 6GB/s)

RESTORING EAGER

exercise your brain….

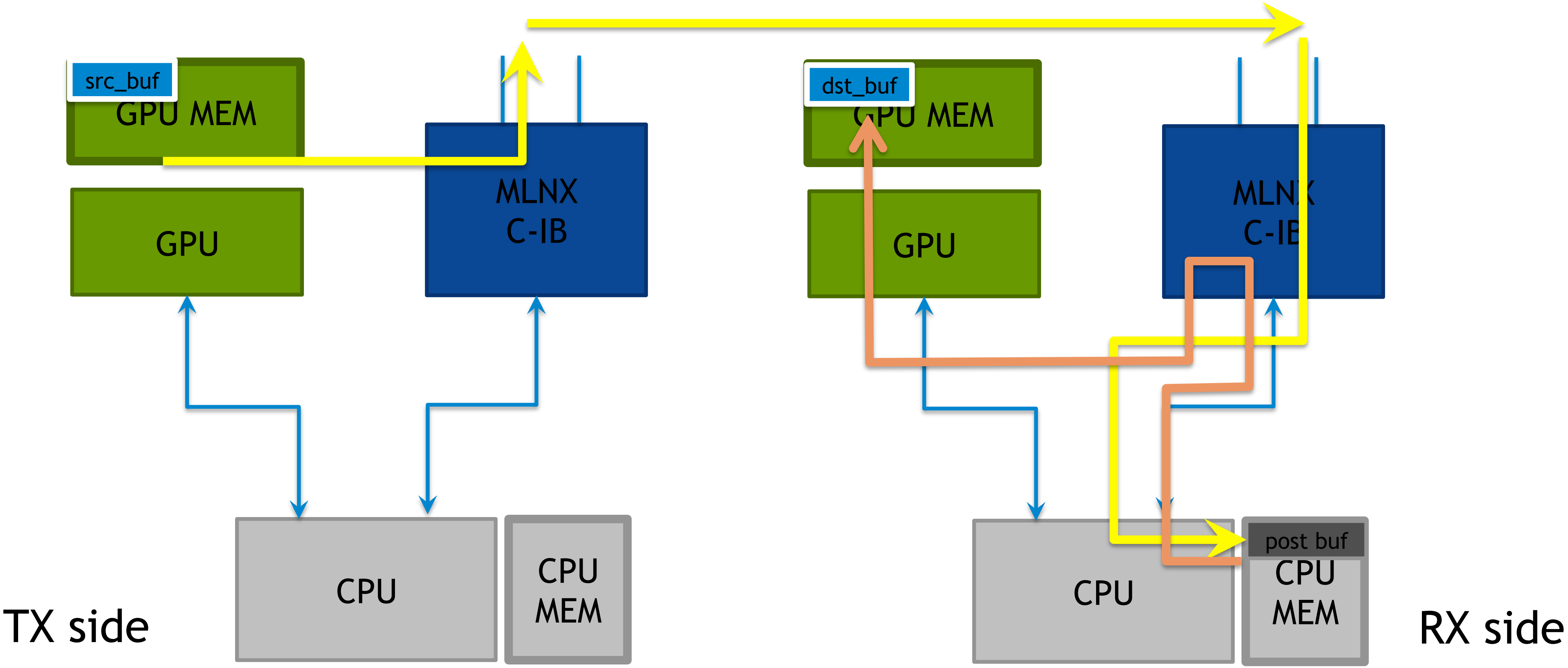
IB loopback

use NIC as low-latency copy engine

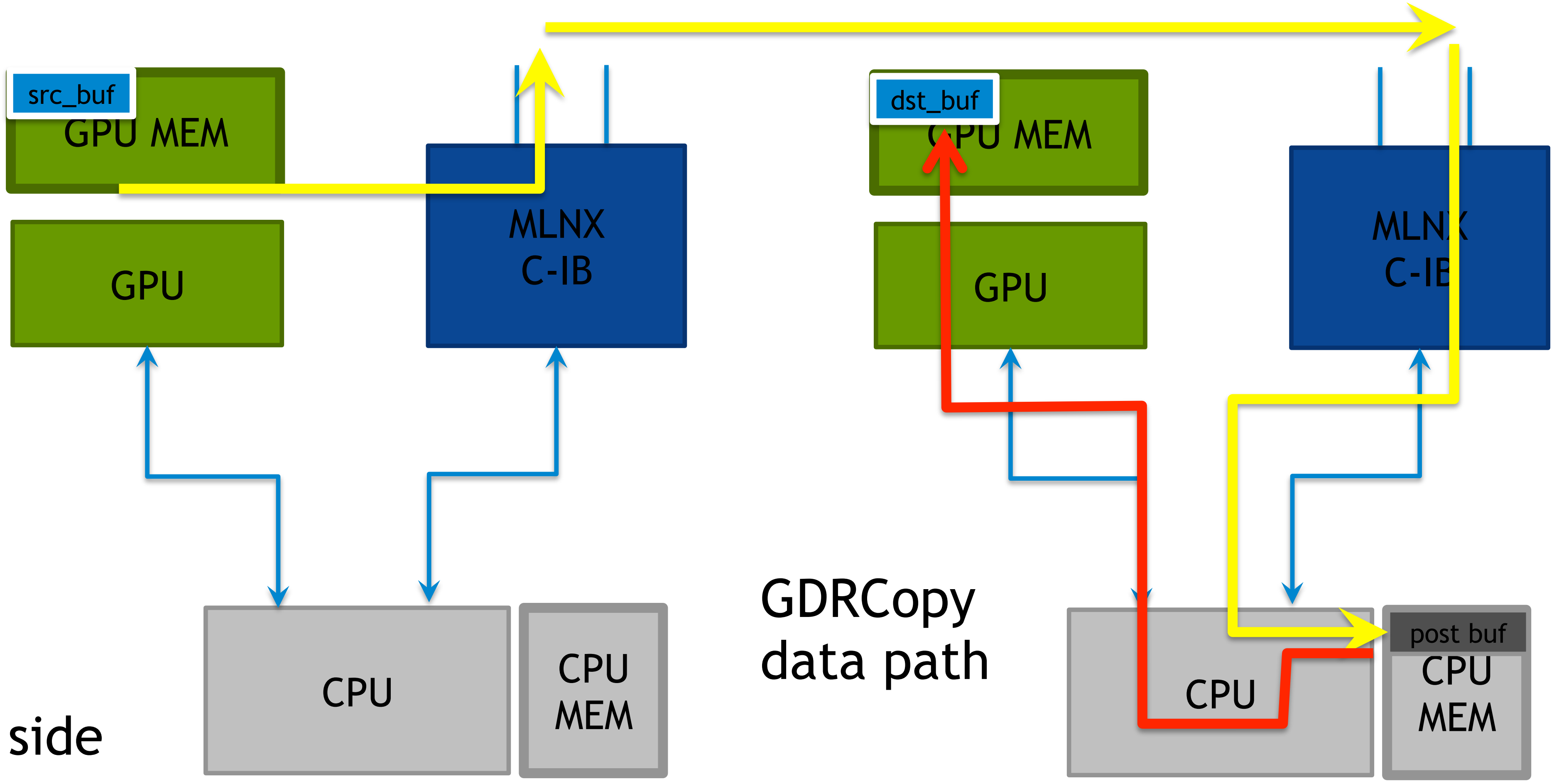
GDRCopy

CPU-driven zero-latency BAR1 copy

MPI EAGER + RX LOOPBACK FLOW



MPI EAGER + RX GDRCOPY FLOW



GDRCopy
data path

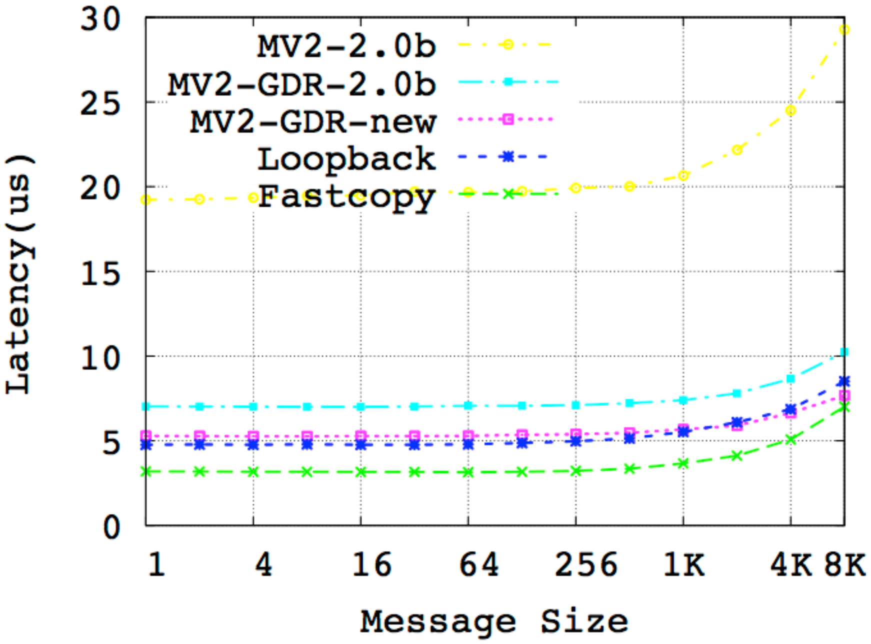
TX side

RX side

BENCHMARK RESULTS

IVB Xeon + K40m + CUDA 6.0 + MLNX C-IB FDR + OFED 2.2

```
#    MV-2.0b   MV2-GDR-2.0b MV2-GDR   loopback gdrcopy
0    1.27      1.31         1.16      1.22     1.21
1    19.23     7.03         5.29      4.77     3.19*
2    19.26     7.02         5.28      4.78     3.18
4    19.35     7.01         5.26      4.77     3.17
8    19.45     7.00         5.26      4.79     3.17
```



* MVAPICH2 GDR 2.0 pre-release, improved in final

A FEW MORE FACTS

low-latency tricks

CUDA 6.0:

- sync copy, much less overhead (~4us)

CUDA 6.5:

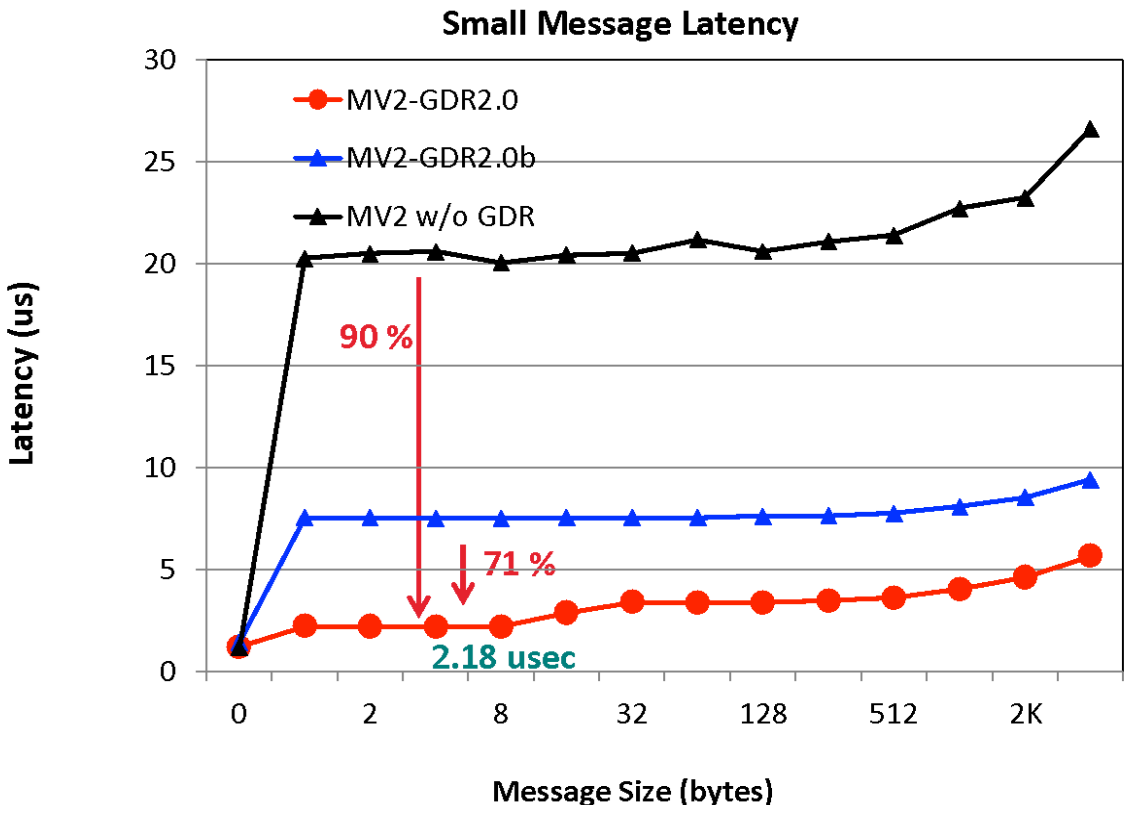
- introduce a memory-deallocation interception framework (samples/7_CUDALibraries/cuHook)
- enable registration cache, avoiding calls to cuPointerGetAttribute() [.5us]

K40m since 11/2013:

- 16GB BAR, perfect for GPUDirect RDMA

BENCHMARK RESULTS

GPU-to-GPU latency

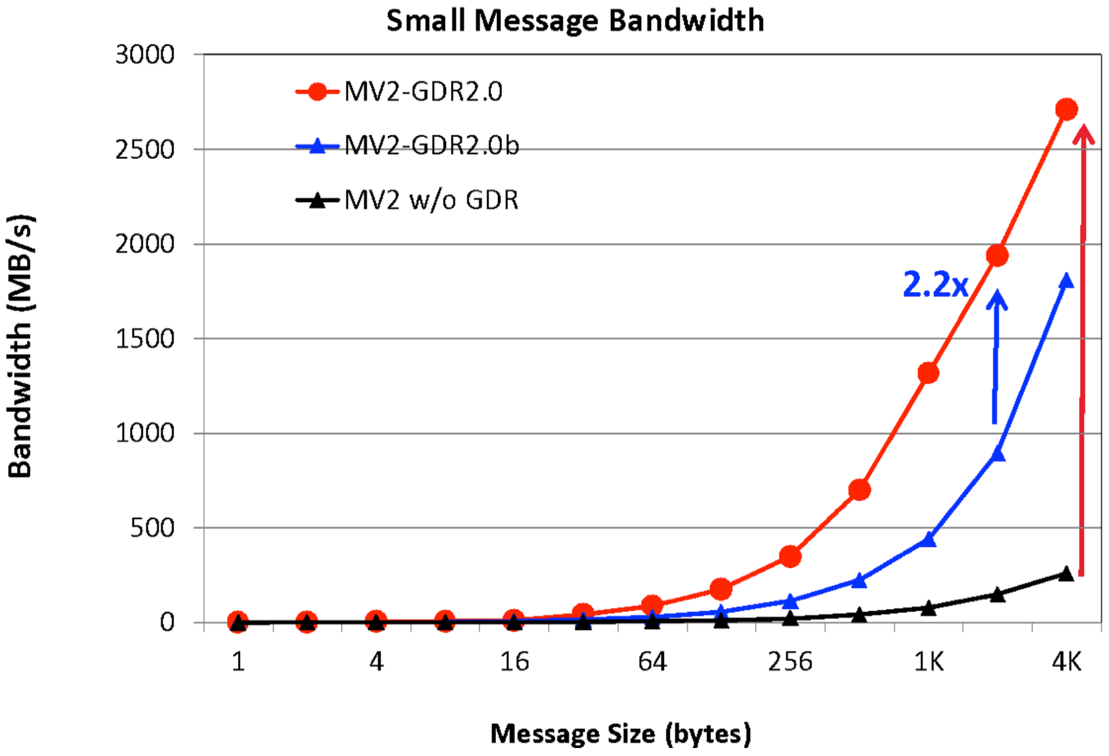


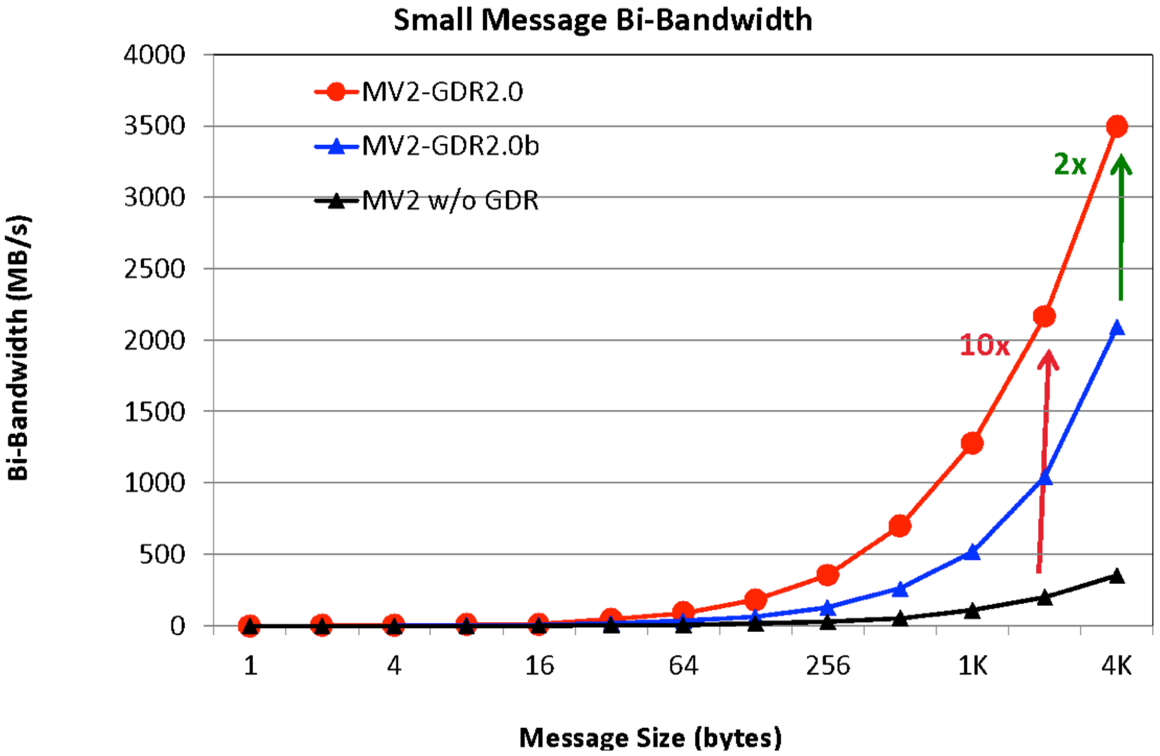
- IVB Xeon (E5-2680 v2)
- K40c + CUDA 6.5
- MLNX Connect-IB FDR
- MLNX OFED 2.1

[*] D.K. Panda talk @ MVAPICH2 user group meeting

BENCHMARK RESULTS

uni- and bi-dir GPU-to-GPU Bandwidth

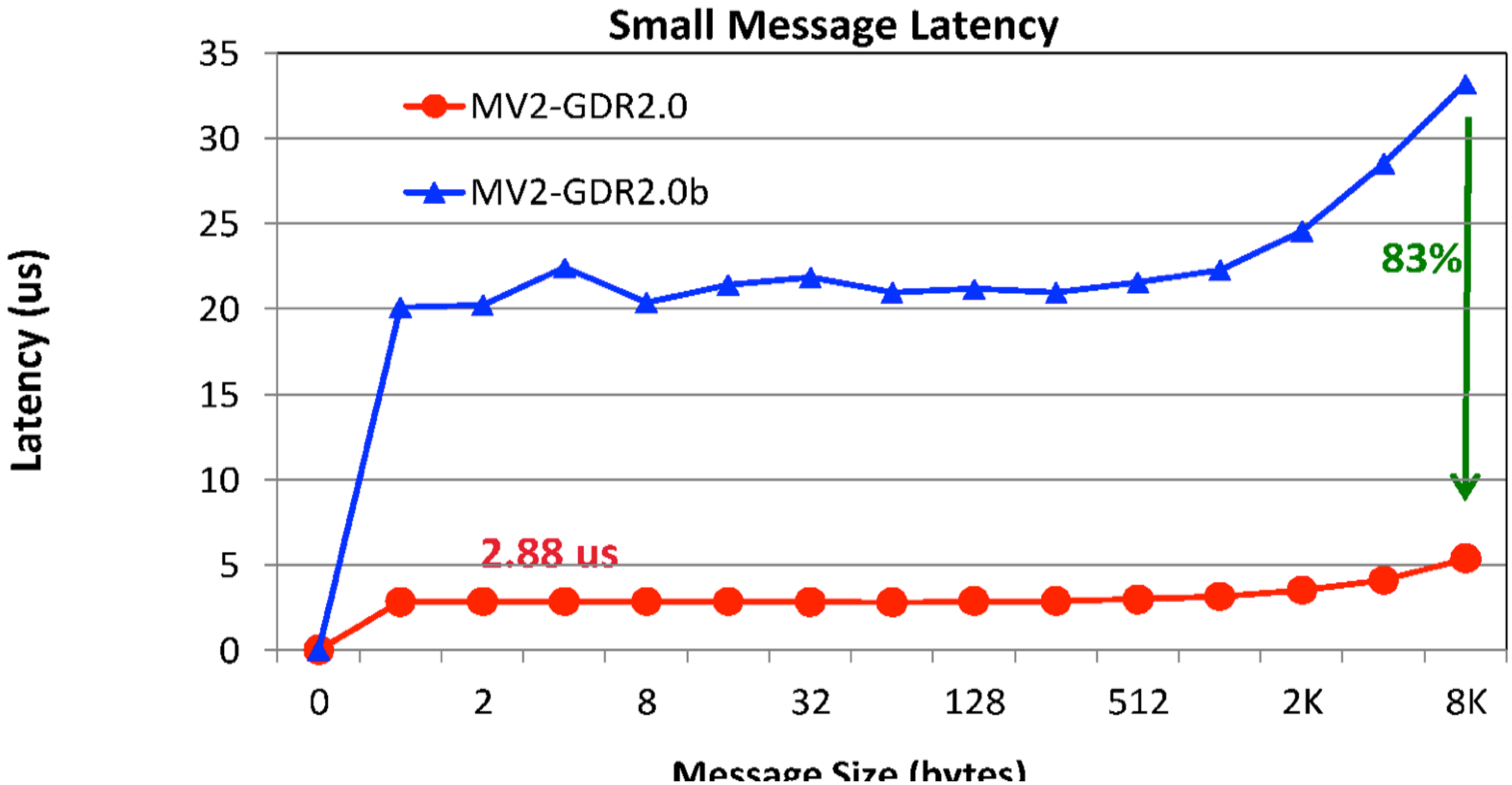




[*] D.K. Panda talk @ MVAPICH2 user group meeting

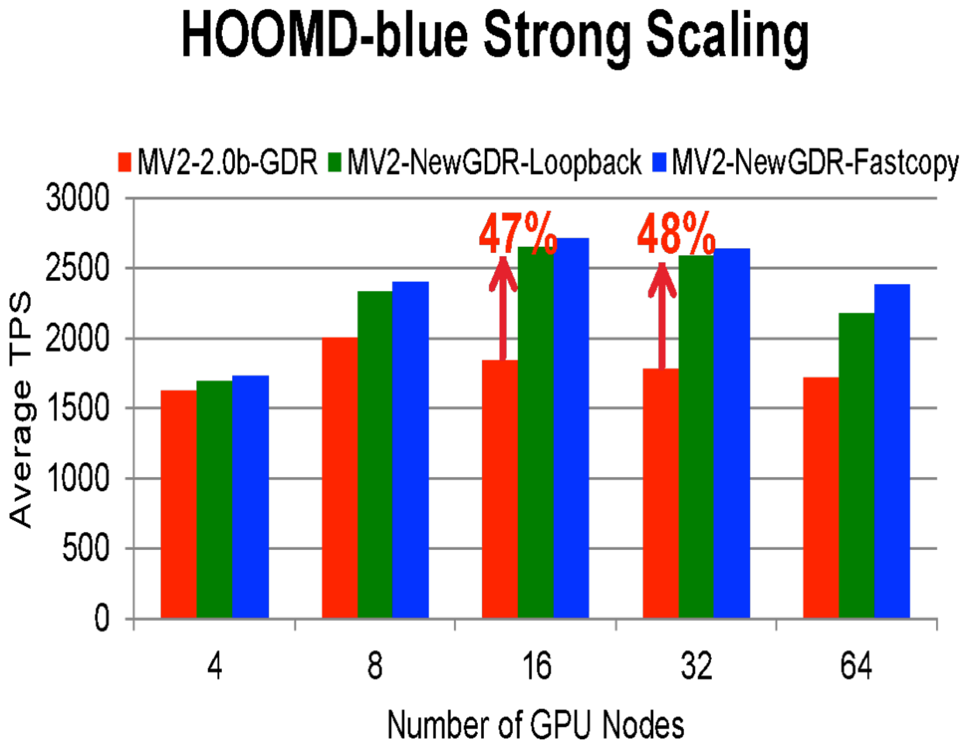
BENCHMARK RESULTS

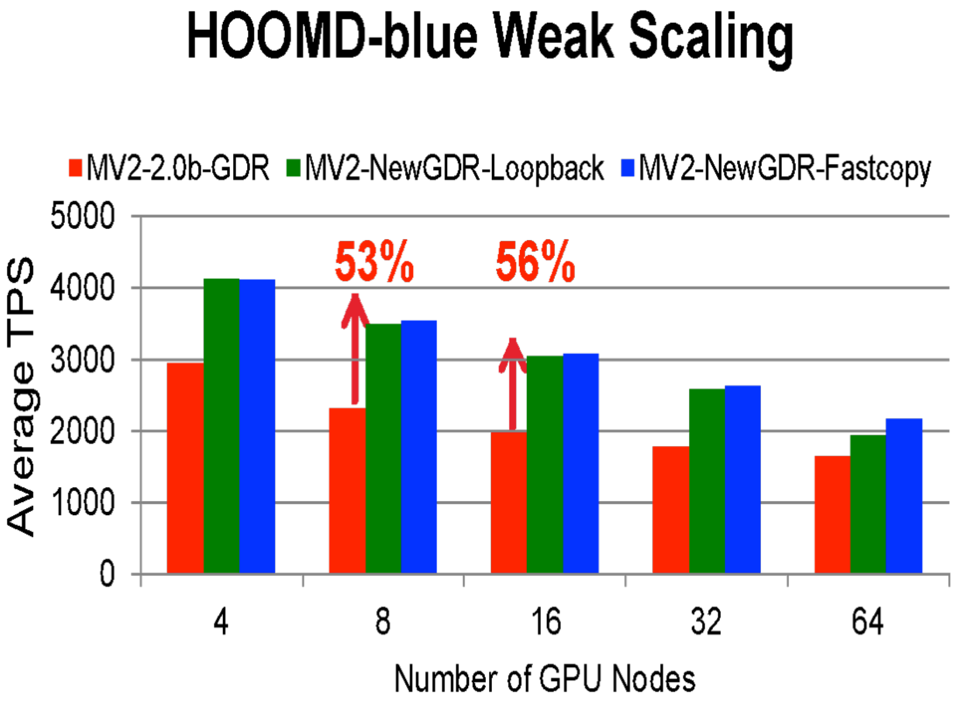
one-sided communication / RMA



BENCHMARK RESULTS

HOOMD-blue scaling

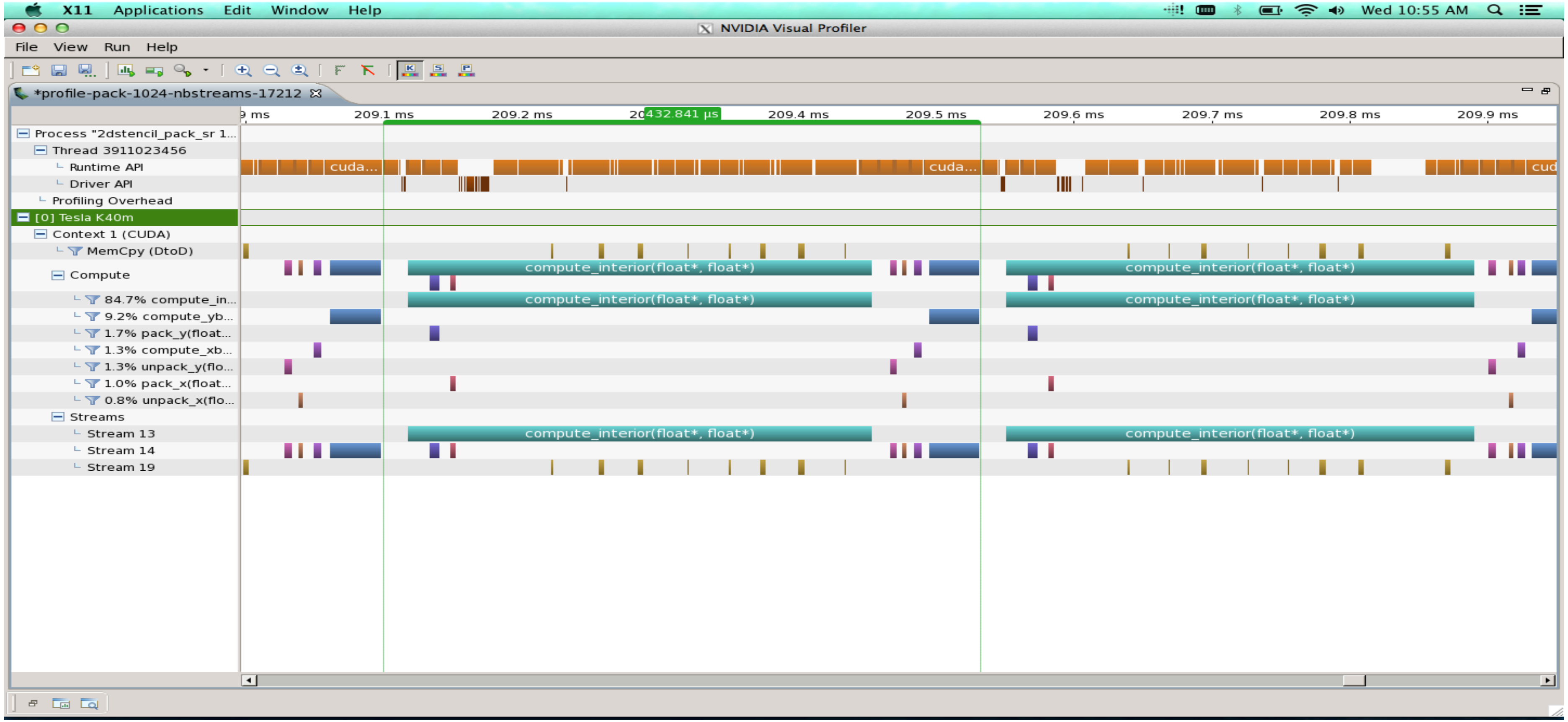




GPUDIRECT ASYNC

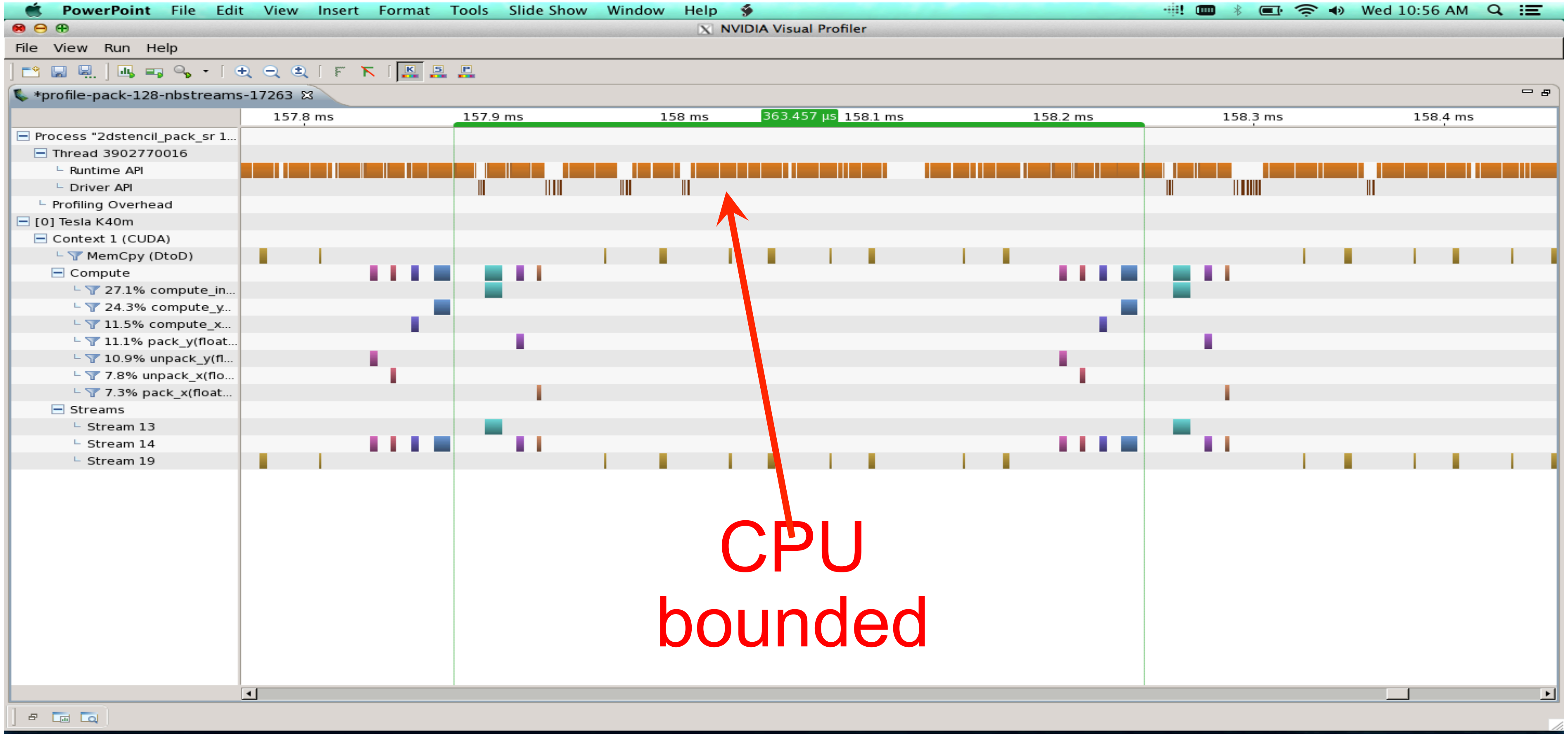
ASYNC: MOTIVATION

VISUAL PROFILE - TRADITIONAL



(Time marked for one step, Domain size/GPU – 1024, Boundary – 16, Ghost Width – 1)

VISUAL PROFILE - TRADITIONAL

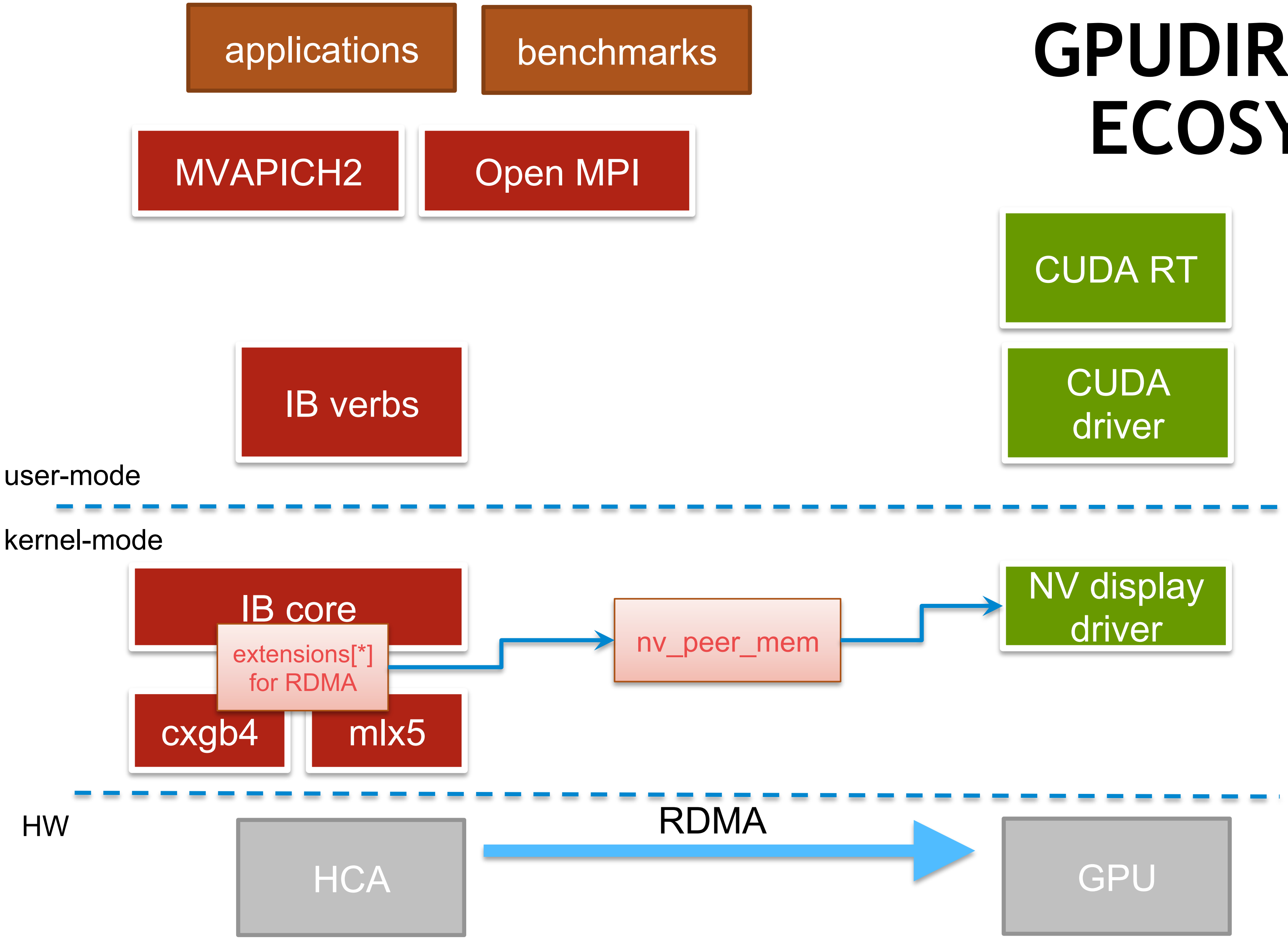


CPU bounded

(Time marked for one step, Domain size/GPU – 128, Boundary – 16, Ghost Width – 1)

SW ARCHITECTURE

GPUDIRECT SW ECOSYSTEM



applications

benchmarks

MVAPICH2

Open MPI

CUDA RT

IB verbs

CUDA driver

user-mode

kernel-mode

IB core

extensions[*] for RDMA

nv_peer_mem

NV display driver

cxgb4

mlx5

HW

HCA

RDMA

GPU

🟩 proprietary

🟥 open-source

🟫 mixed

[*] MLNX OFED, Chelsio www.openfabrics.org/~swise/ofed-3.12-1-peer-direct/OFED-3.12-1-peer-direct-20150330-1122.tgz

GPUDIRECT ASYNC

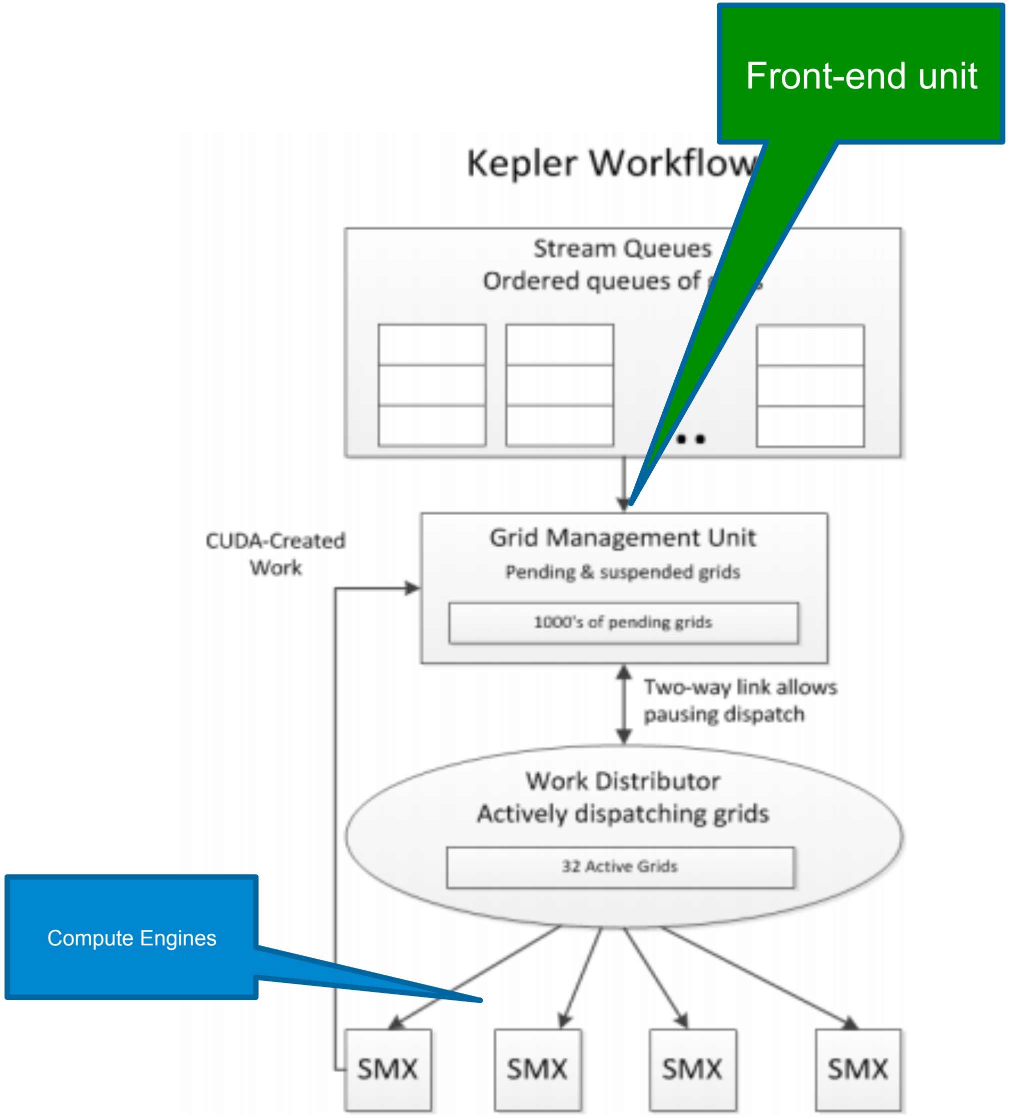
leverage GPU front-end unit

Communication plan prepared by CPU

- hardly parallelizable, branch intensive
- GPU orchestrates flow

Run by GPU front-end unit

- Same one scheduling GPU work
- Now also scheduling network communications

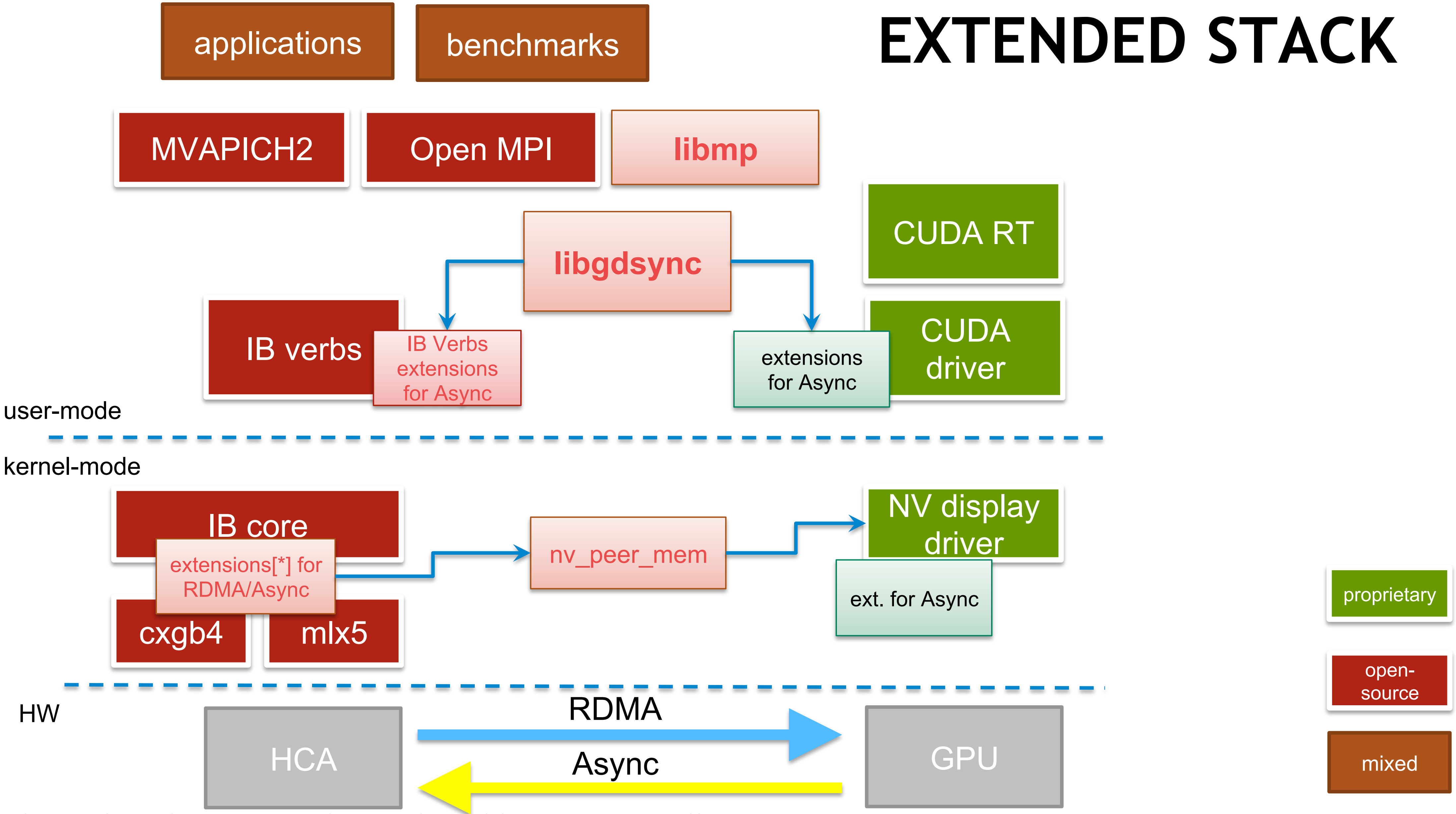


GPUDIRECT ASYNC

SW stack bits

- new APIs: CUDA Async, IB Verbs peer-direct async extensions
- nv_peer_mem: extended, enabling peer mappings (for Async)
- libgdsync: new, for Async, bridging CUDA & IB
- libmp: new, technology demonstration, for Async
    - MPI-like _on_stream() primitives
- MPI support

EXTENDED STACK



[*] MLNX OFED, Chelsio www.openfabrics.org/~swise/ofed-3.12-1-peer-direct/OFED-3.12-1-peer-direct-20150330-1122.tgz

GPUDIRECT ASYNC + INFINIBAND

preview release of components

- CUDA Async extensions, preview in CUDA 8.0 EA
- Peer-direct Async extensions, in MLNX OFED 3.x, soon
- libgdsync, on github.com/gpudirect, soon
- libmp, on github.com/gpudirect, soon

ASYNC: CUDA APIS

STREAM MEMORY OPERATIONS

```
 CU_STREAM_WAIT_VALUE_GEQ   = 0x0,
 CU_STREAM_WAIT_VALUE_EQ    = 0x1,
 CU_STREAM_WAIT_VALUE_AND   = 0x2,
 CU_STREAM_WAIT_VALUE_FLUSH = 1<<30
CUresult cuStreamWaitValue32(CUstream stream, CUdeviceptr addr,
cuuint32_t value, unsigned int flags);

 CU_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER = 0x1
CUresult cuStreamWriteValue32(CUstream stream, CUdeviceptr addr,
cuuint32_t value, unsigned int flags);

 CU_STREAM_MEM_OP_WAIT_VALUE_32  = 1,
 CU_STREAM_MEM_OP_WRITE_VALUE_32 = 2,
 CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES = 3
CUresult cuStreamBatchMemOp(CUstream stream, unsigned int count,
CUstreamBatchMemOpParams *paramArray, unsigned int flags);
```

guarantee memory consistency fpr RDMA

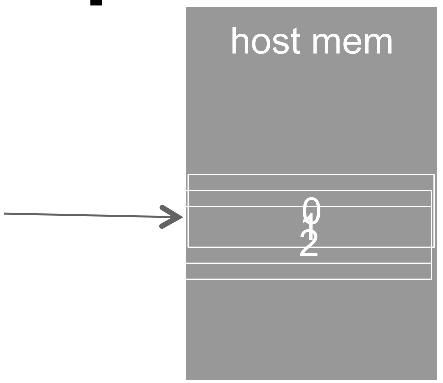
polling on 32-bit word

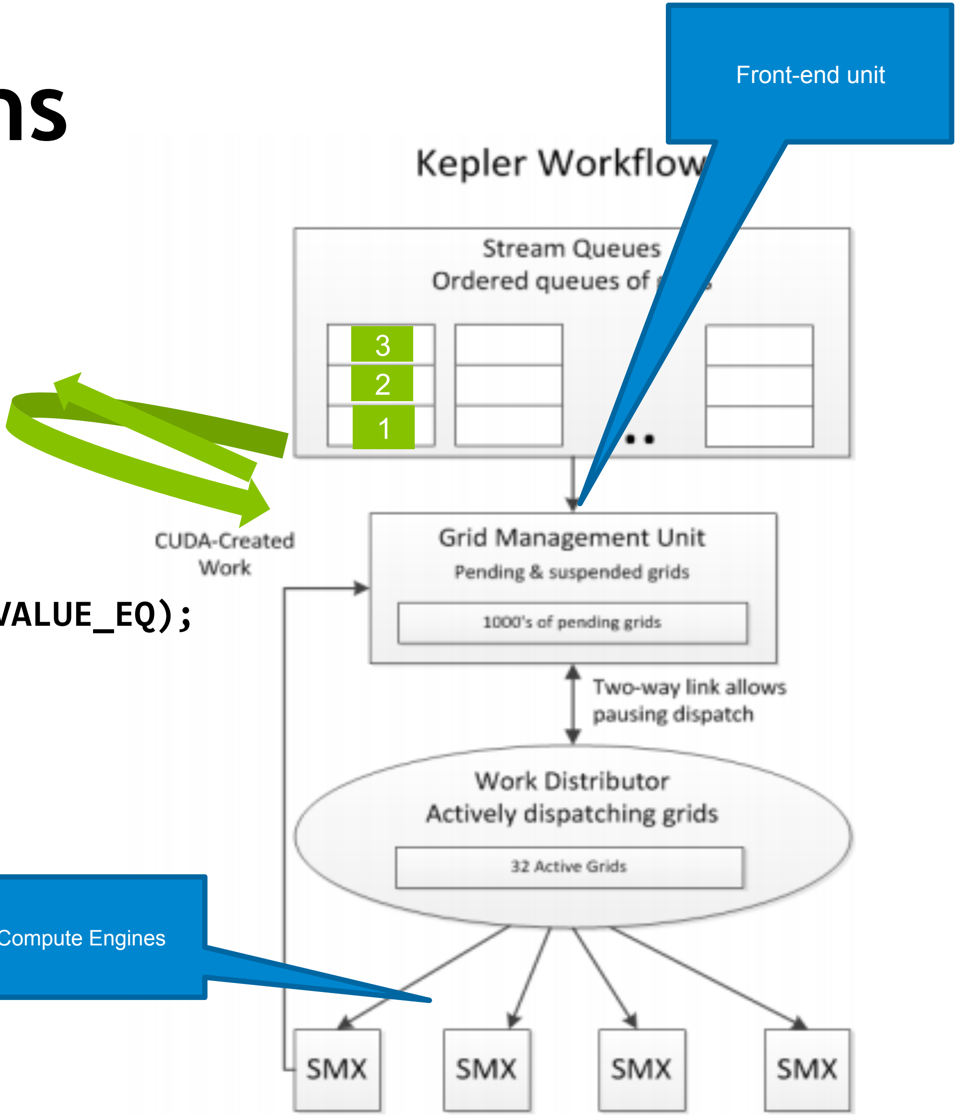
32-bit word write

low-overhead batched work submission

stream memory operations

GPU front-end unit

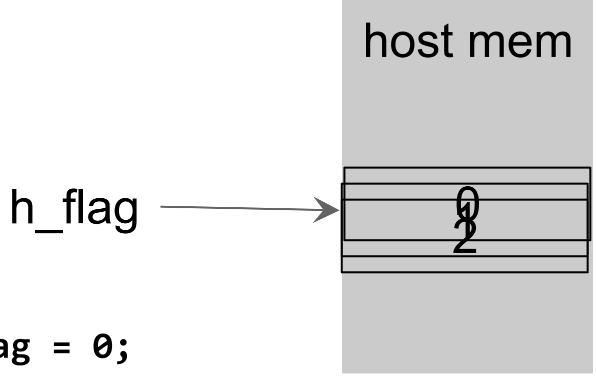




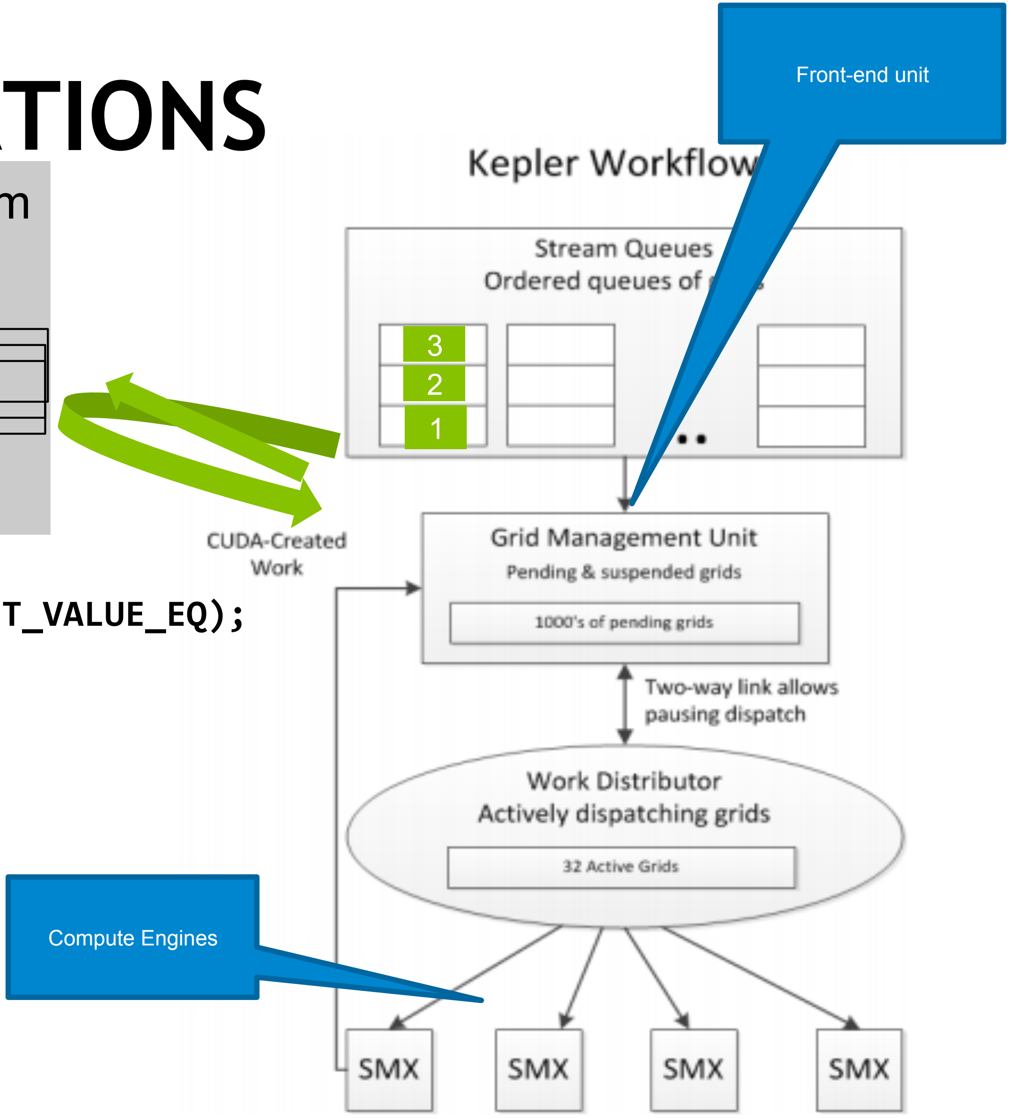
```
*(volatile uint32_t*)h_flag = 0;
…
cuStreamWaitValue32(stream, d_flag, 1, CU_STREAM_WAIT_VALUE_EQ);
calc_kernel<<<GSZ,BSZ,0,stream>>>();
cuStreamWriteValue32(stream, d_flag, 2, 0);
…
*(volatile uint32_t*)h_flag = 1;
…
cudaStreamSynchronize(stream);
assert(*(volatile uint32_t*)h_flag== 2);
```

STREAM MEMORY OPERATIONS

GPU front-end unit



```
*(volatile uint32_t*)h_flag = 0;
…
cuStreamWaitValue32(stream, d_flag, 1, CU_STREAM_WAIT_VALUE_EQ);
calc_kernel<<<GSZ,BSZ,0,stream>>>();
cuStreamWriteValue32(stream, d_flag, 2, 0);
…
*(volatile uint32_t*)h_flag = 1;
…
cudaStreamSynchronize(stream);
assert(*(volatile uint32_t*)h_flag== 2);
```



GPUDIRECT ASYNC

APIs features

- batching multiple consecutive mem ops save ~1us each op
  - use cuStreamBatchMemOp
- APIs accept device pointers
  - memory need registration (cuMemHostRegister)
  - device pointer retrieval (cuMemHostGetDevicePointer)
- 3rd party device PCIe resources (aka BARs)
  - assumed physically contiguous & uncached
  - special flag needed

GPU PEER MAPPING

accessing 3rd party device PCIe resource from GPU

```
struct device_bar {
        void *ptr;
        CUdeviceptr d_ptr;
        size_t len;
};

void map_device_bar(device_bar *db)
{
        device_driver_get_bar(&db->ptr,&db->len);
        CUCHECK(cuMemHostRegister(db->ptr, db->len,

CU_MEMHOSTREGISTER_IOMEMORY));
        CUCHECK(cuMemHostGetDevicePointer(&db->d_ptr, db->ptr,
0));
}


…
cuStreamWriteValue32(stream, db->d_ptr+off, 0xfaf0, 0);
```

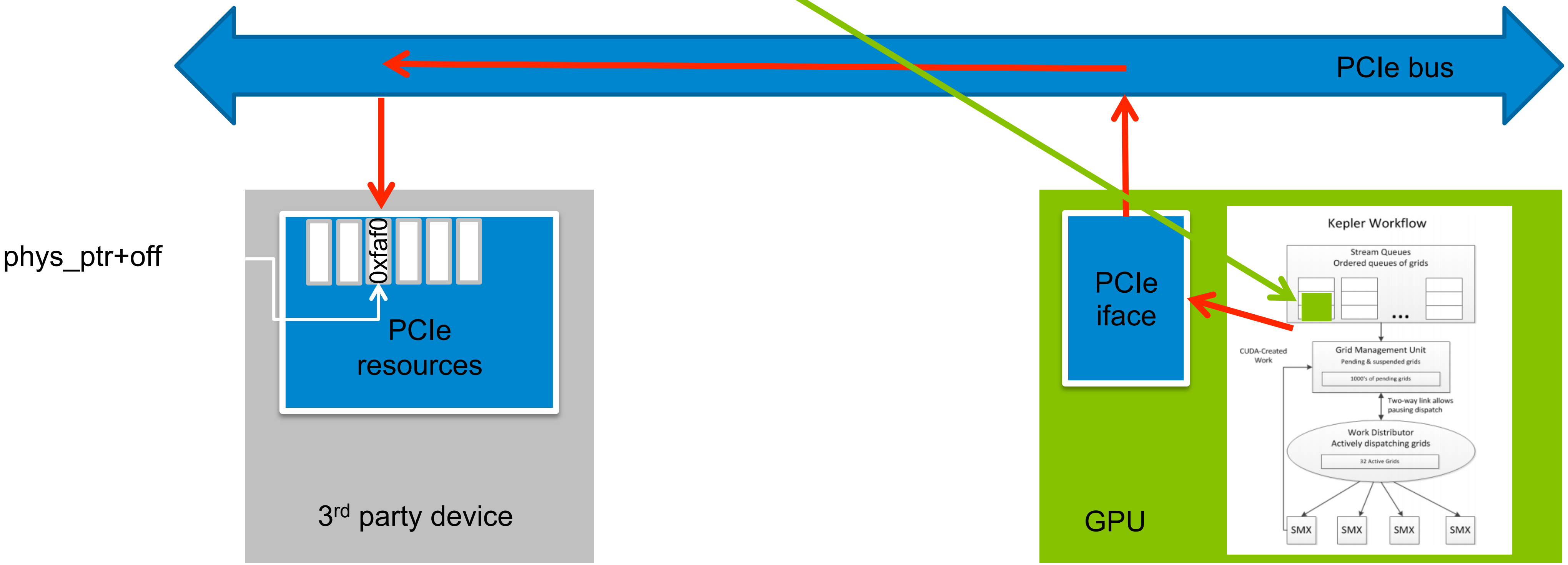
← registration is mandatory

← new flag

← GPU access to device thru device pointer

GPU peer mapping + Async

```
cuStreamWriteValue32(stream, db->d_ptr+off, 0xfaf0, 0);
```



ASYNC: LIBMP

LIBMP

Prototype message passing library

- in-order matching between process pairs
- zero copy transfers, uses flow control of IB RC transport
- contiguous data

An example for runtime level designs in PAMI or MPI

A means to try initial ports of MPI applications

LIBMP API – CPU-SYNC COMMUNICATION

Send/Recv

```
int mp_irecv (void *buf, int size, int peer, mp_reg_t *mp_reg, mp_request_t *req);

int mp_isend (void *buf, int size, int peer, mp_reg_t *mp_reg, mp_request_t *req);
```

Put/Get

```
int mp_window_create(void *addr, size_t size, mp_window_t *window_t);

int mp_iput (void *src, int size, mp_reg_t *src_reg, int peer, size_t displ, mp_window_t *dst_window_t,
mp_request_t *req);

int mp_iget (void *dst, int size, mp_reg_t *dst_reg, int peer, size_t displ, mp_window_t *src_window_t,
mp_request_t *req);
```

Wait

```
int mp_wait (mp_request_t *req);

int mp_wait_all (uint32_t count, mp_request_t *req);
```

LIBMP API – STREAM-SYNC COMMUNICATION

Send

```
int mp_isend_on_stream  (void *buf, int size, int peer, mp_reg_t *mp_reg, mp_request_t *req, cudaStream_t
stream);
```

Put/Get

```
int mp_iput_on_stream (void *src, int size, mp_reg_t *src_reg, int peer, size_t displ, mp_window_t
*dst_window_t, mp_request_t *req, cudaStream_t stream);
```

```
int mp_iget_on_stream (void *dst, int size, mp_reg_t *dst_reg, int peer, size_t displ, mp_window_t
*src_window_t, mp_request_t *req, cudaStream_t stream);
```

Wait

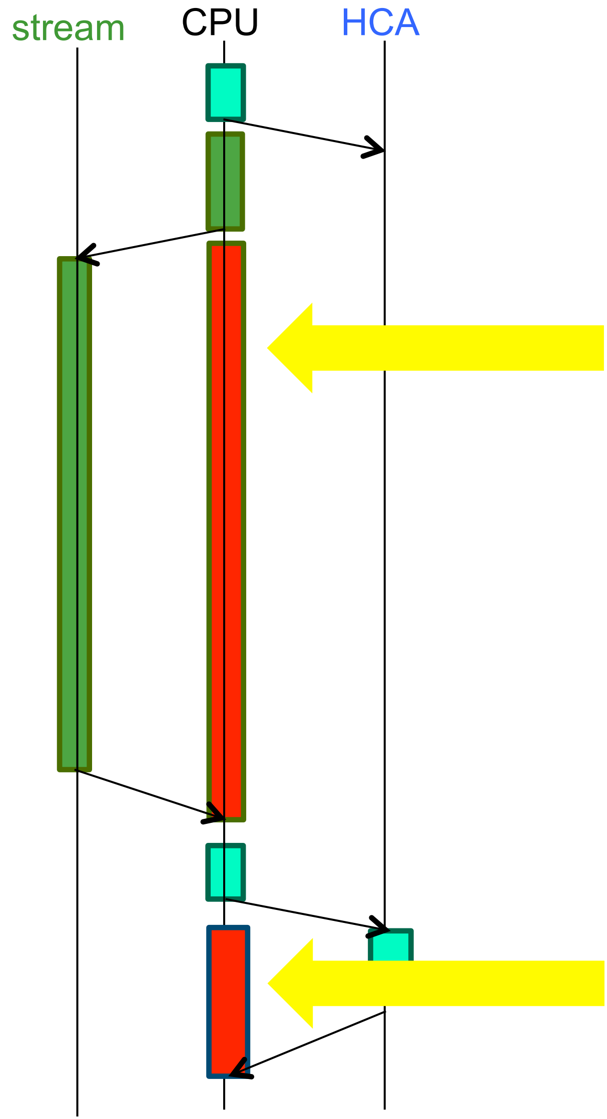
```
int mp_wait_on_stream (mp_request_t *req, cudaStream_t stream);
```

```
int mp_wait_all_on_stream (uint32_t count, mp_request_t *req, cudaStream_t stream);
```

Using Normal API

```
Loop {
        mp_irecv(…)
Loop {
        compute <<<…,stream>>> (buf)
    compute (on GPU)
         cudaStreamSynchronize (stream)
    near neighbor exchange
         mp_isend(…)
}
         mp_wait_all (…)
}
```

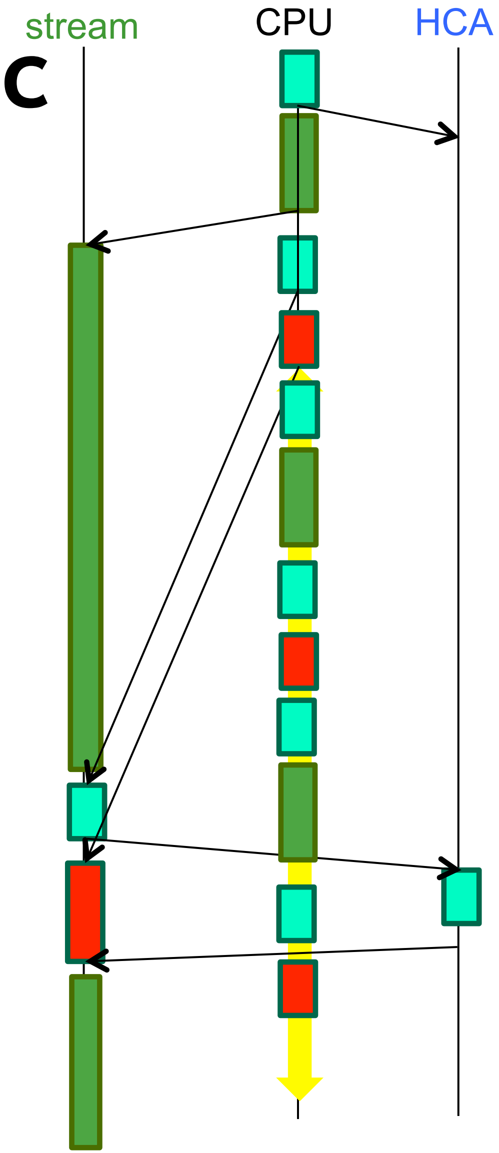
100% CPU utilization
Limited scaling!



Using Stream-ordered API + Async

```
Loop {
        mp_irecv(…)

        compute <<<…,stream>>> (buf)

        mp_isend_on_stream(…)

        mp_wait_all_on_stream (…)
}
```

CPU is free !



LIBMP API – AGGREGATION

Prepare

```
int mp_send_prepare (void *buf, int size, int peer, mp_reg_t *mp_reg, mp_request_t *req);
```

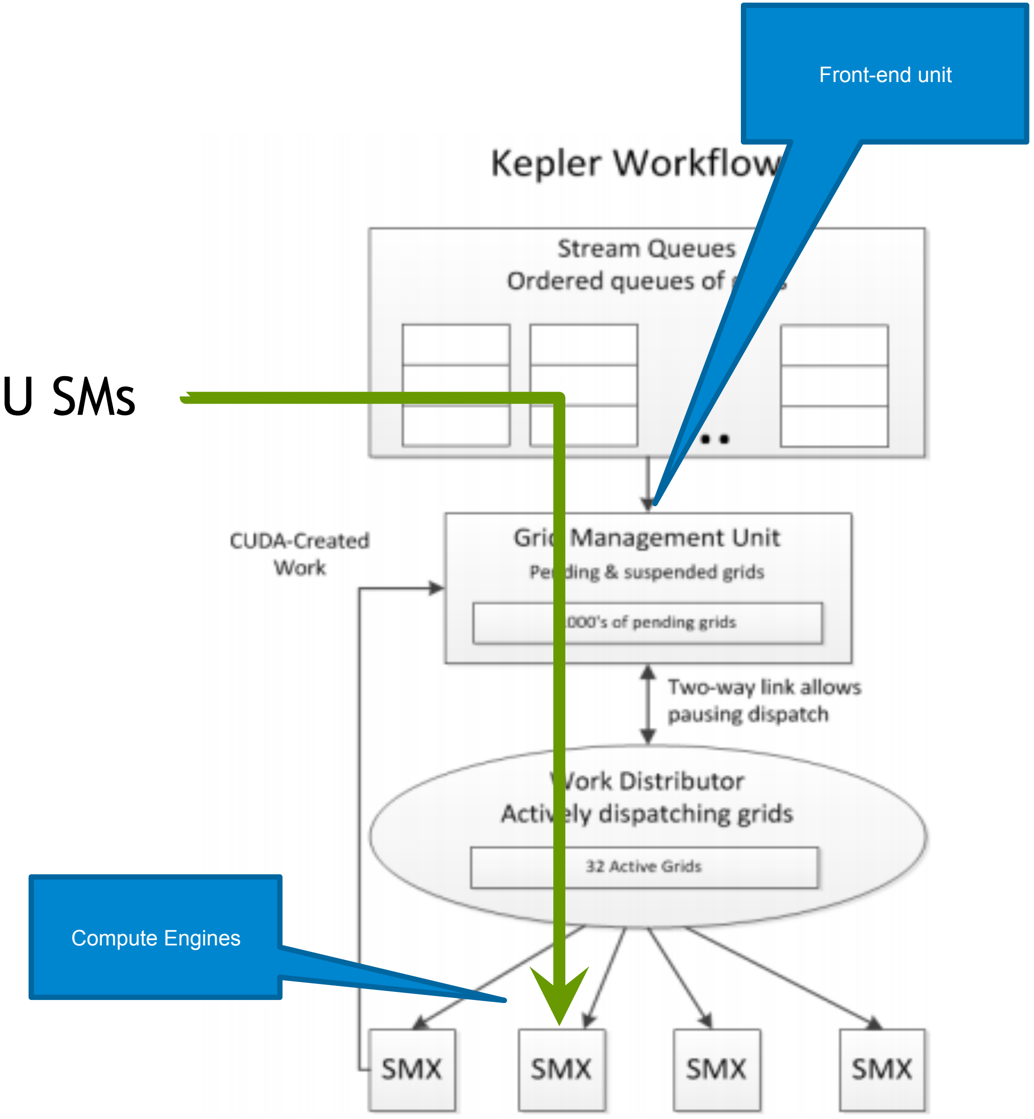
Post/Post all

```
int mp_isend_post_on_stream (mp_request_t *req, cudaStream_t stream);
```

```
int mp_isend_post_all_on_stream (uint32_t count, mp_request_t *req, cudaStream_t stream);
```

FUTURE WORK

- libmp: IB communications initiated by GPU SMs [prototyped]
- libmp: GPU & remote peer affinity
- libmp: striping over dual-lane HCAs



ASYNC: BENCHMARKS

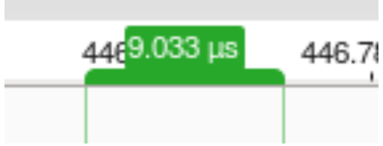
ping-kernel-pong benchmark

using LibMP APIs

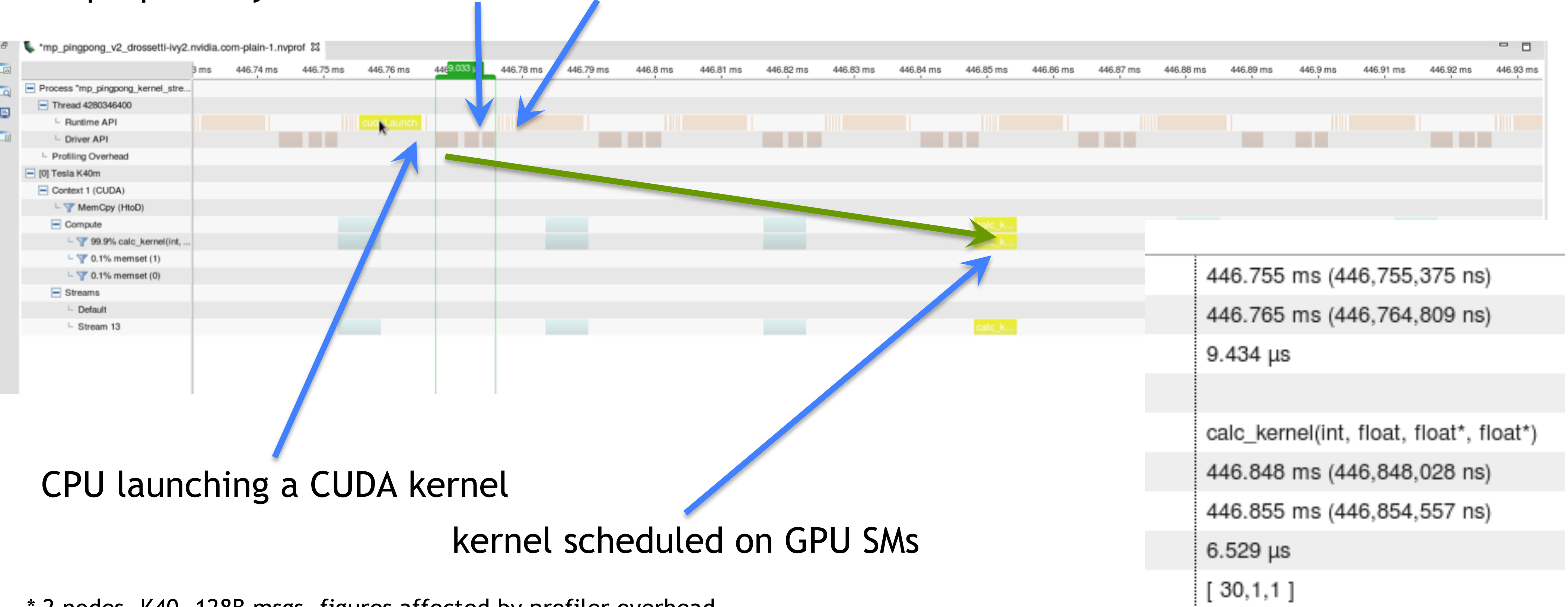
```
// pre-post mp_irecv()’s …
for (j=0; j<steps_per_batch; j++) {
  if (!my_rank) {
    MP_CHECK(mp_wait_on_stream(&rreq[j], stream));
    gpu_launch_calc_kernel(kernel_size, stream);
    MP_CHECK(mp_isend_on_stream(sbuf_d), size, peer, &sreq[j], stream));
  } else {
   MP_CHECK(mp_isend_on_stream(sbuf_d), size, peer, &sreq[j], stream));
   MP_CHECK(mp_wait_on_stream(&rreq[j], stream));
   gpu_launch_calc_kernel(kernel_size, stream);
}
…
cudaStreamSynchronize(stream);
```

PING-KERNEL-PONG BENCHMARK

20-30% improvement



CPU prepares sync network send + recv on GPU



446.755 ms (446,755,375 ns)
446.765 ms (446,764,809 ns)
9.434 µs

calc_kernel(int, float, float*, float*)
446.848 ms (446,848,028 ns)
446.855 ms (446,854,557 ns)
6.529 µs
[30,1,1]

CPU launching a CUDA kernel

kernel scheduled on GPU SMs

* 2 nodes, K40, 128B msgs, figures affected by profiler overhead

2DSTENCIL

Regular

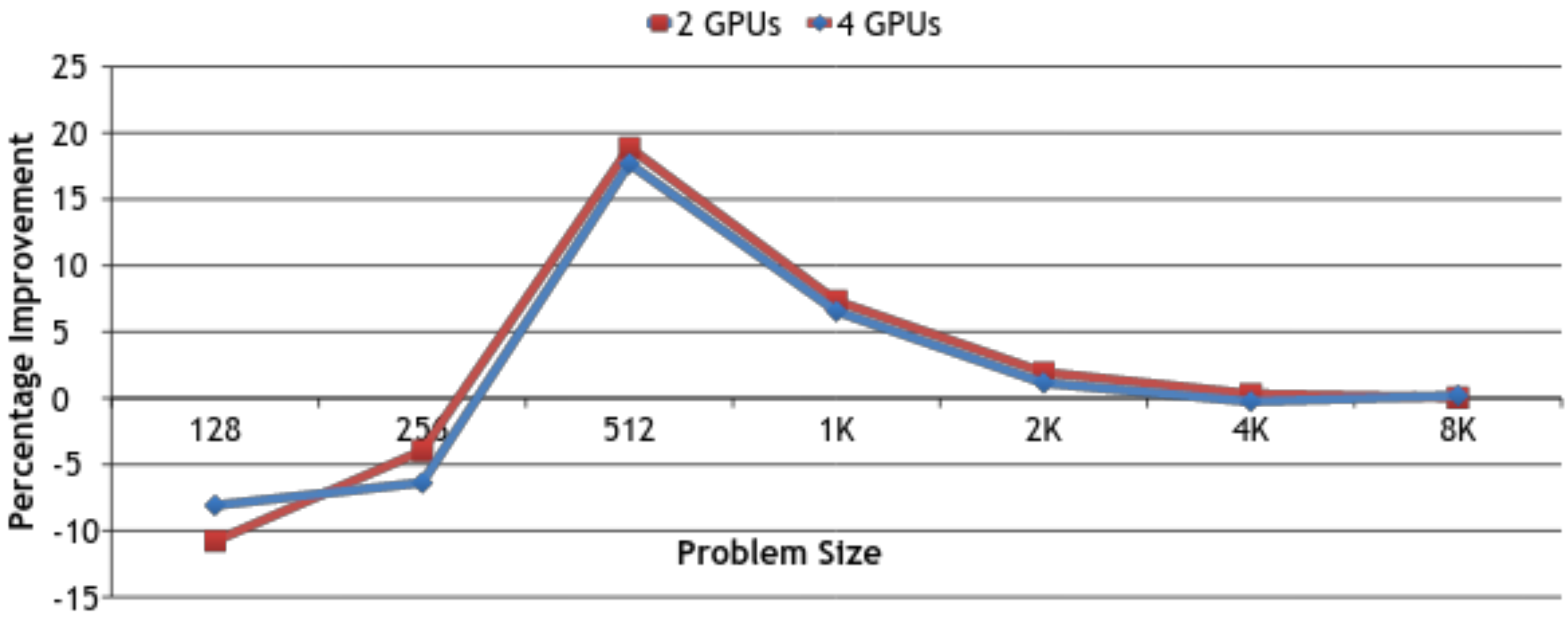
```
Loop {
    mp_irecv(…,rreq)

    pack_boundary <<<…,stream1>>> (buf)

    compute_interior <<<…,stream2>>> (buf)

    cudaStreamSynchronize (stream1)

    mp_isend(…,sreq) //boundary exchange

    mp_wait (rreq)

    unpack_boundary <<<…,stream1>>> (buf)

    compute_boundary <<<…,stream1>>> (buf)

    cudaDeviceSynchronize ()
}
mp_wait_all(sreq)
```

Async

```
Loop {
    mp_irecv (…, rreq)

    pack_boundary <<<…,stream1>>> (buf)

    compute_interior <<<…,stream2>>> (buf)

    mp_isend_on_stream(…, sreq, stream1)

    mp_wait_on_stream (rreq, stream1)

    unpack_boundary <<<…,stream1>>> (buf)

    compute_boundary <<<…,stream1>>> (buf)

    //synchronize between CUDA streams
}
mp_wait_all(sreq)

cudaDeviceSynchronize ()
```

2DSTENCIL PERFORMANCE

2 streams: interior, boundary



Each node is Ivybridge Xeon CPU + 1 MLNX Connect-IB FDR adapter + 1 NVIDIA K40m GPU

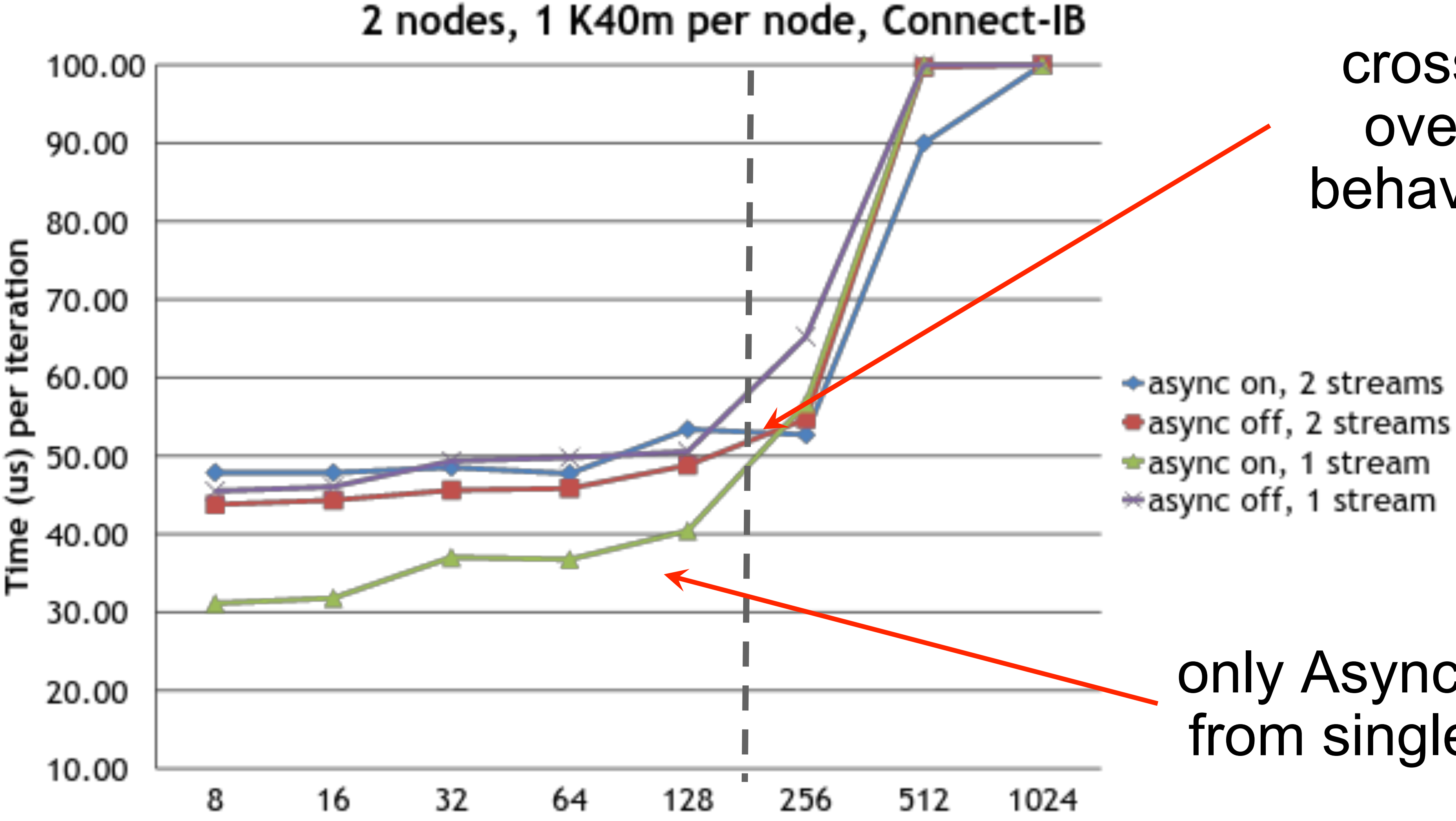
2DSTENCIL

Async, single stream

```
Loop {
    mp_irecv (…, rreq)

    pack_boundary <<<…,stream>>> (buf)

     mp_isend_on_stream(…, sreq, stream)

     compute_interior <<<…,stream>>> (buf)

     mp_wait_on_stream (rreq, stream)

     unpack_boundary <<<…,stream>>> (buf)

     compute_boundary <<<…,stream>>> (buf)

     ~~//cross-synchronize between streams~~

     mp_wait_on_stream(sreq)
}
cudaDeviceSynchronize ()
```

2DSTENCIL PERFORMANCE

one vs two streams

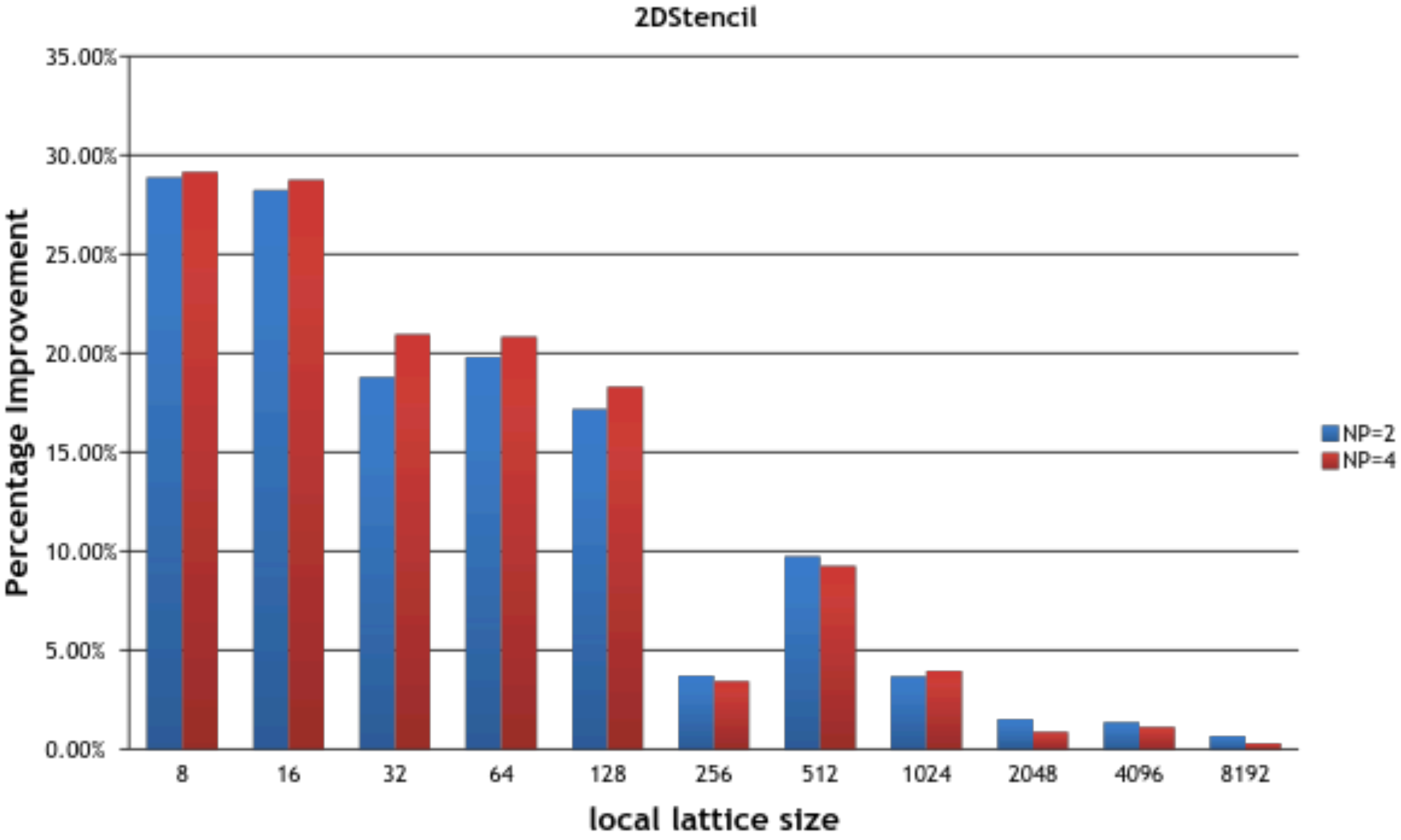


cross-over behavior

only Async benefits from single stream

2DSTENCIL PERFORMANCE

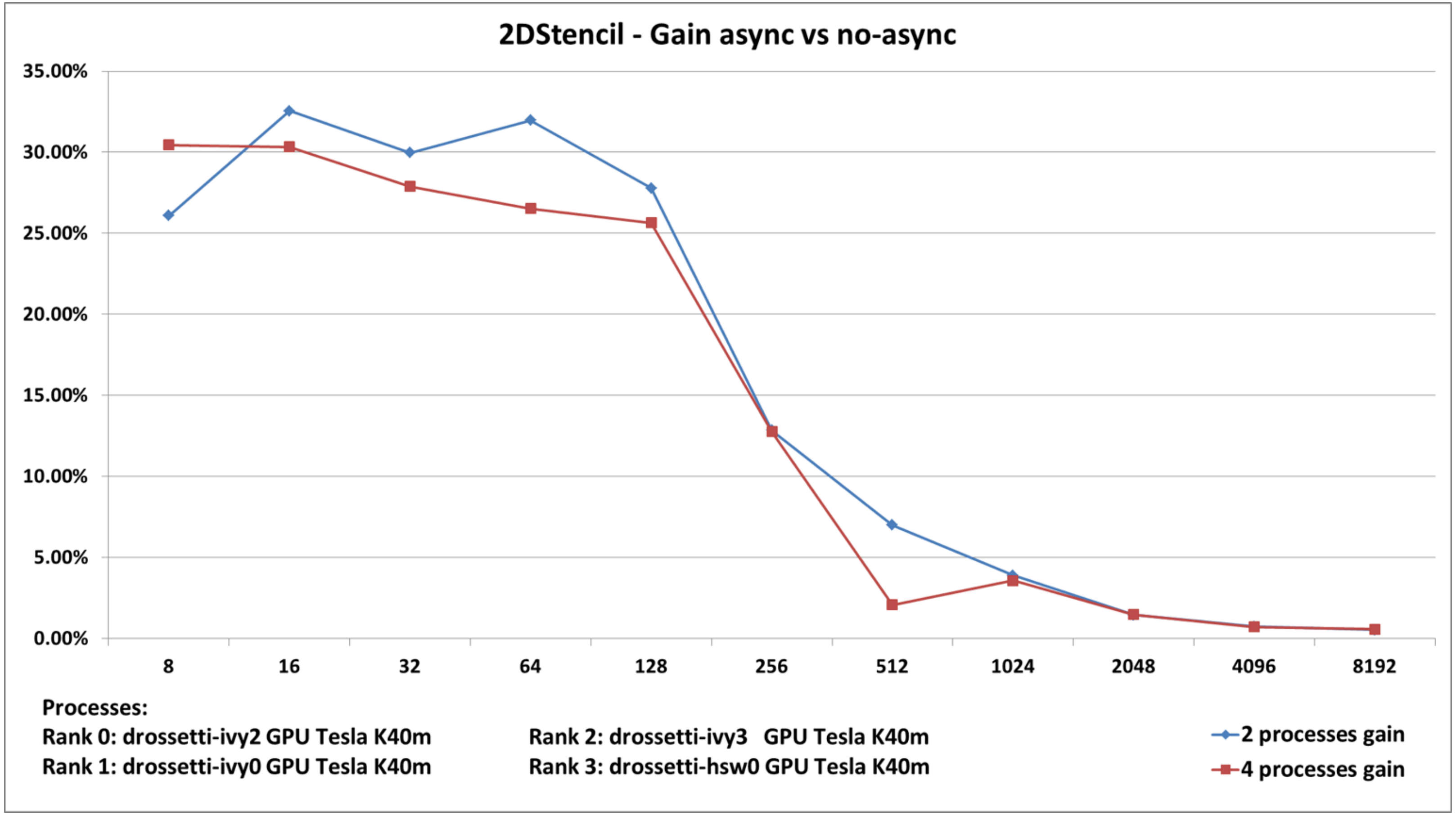
weak scaling, RDMA vs RDMA+Async



two/four nodes, IVB Xeon CPUs, K40m GPUs, Mellanox Connect-IB FDR, Mellanox FDR switch

ASYNC VS NO-ASYNC

2D Stencil example



Processes:

Rank 0: drossetti-ivy2 GPU Tesla K40m

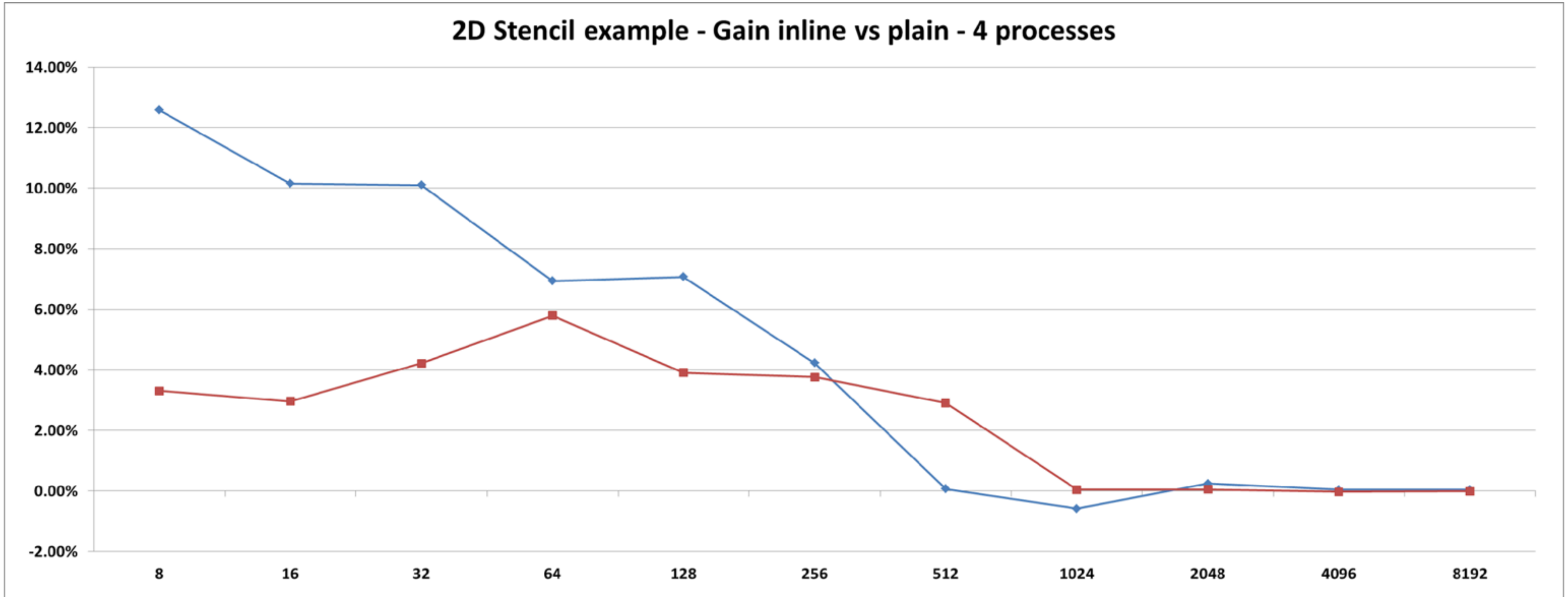
Rank 1: drossetti-ivy0 GPU Tesla K40m

Rank 2: drossetti-ivy3 GPU Tesla K40m

Rank 3: drossetti-hsw0 GPU Tesla K40m

INLINE VS PLAIN

2D Stencil example



Processes:

Rank 0: drossetti-ivy2 GPU Tesla K40m

Rank 1: drossetti-ivy0 GPU Tesla K40m

Rank 2: drossetti-ivy3 GPU Tesla K40m

Rank 3: drossetti-hsw0 GPU Tesla K40m

HPGMG-FV

High Performance Geometric Multigrid is a benchmark designed to proxy the finite volume based geometric multigrid linear solvers.

CUDA implementation by Nikolai Sakharnykh:
https://devblogs.nvidia.com/parallelforall/high-performance-geometric-multi-grid-gpu-acceleration

Use LibMP to implement HPGMG’s communications.

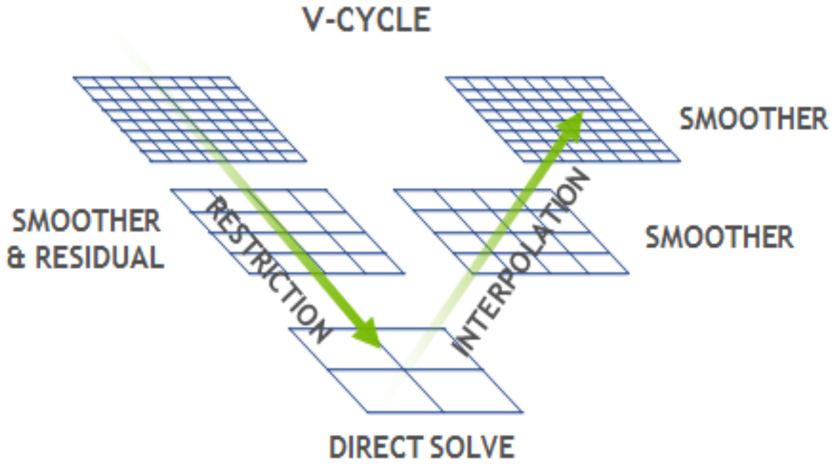
Compare performance: MPI, LibMP no-Async, LibMP Async, GPU-Initiated (experimental)

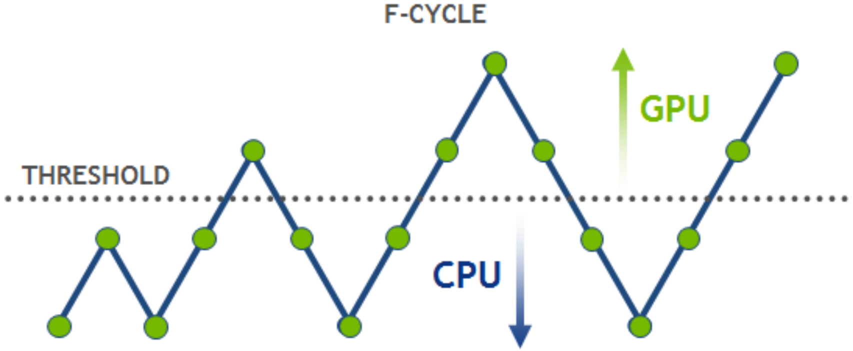
HPGMG-FV

Communications

Communication periods:

- Exchange boundaries (2D Stencil): most frequent operation
- Interpolation: (low level to high level)
- Restriction: (high level to low level)



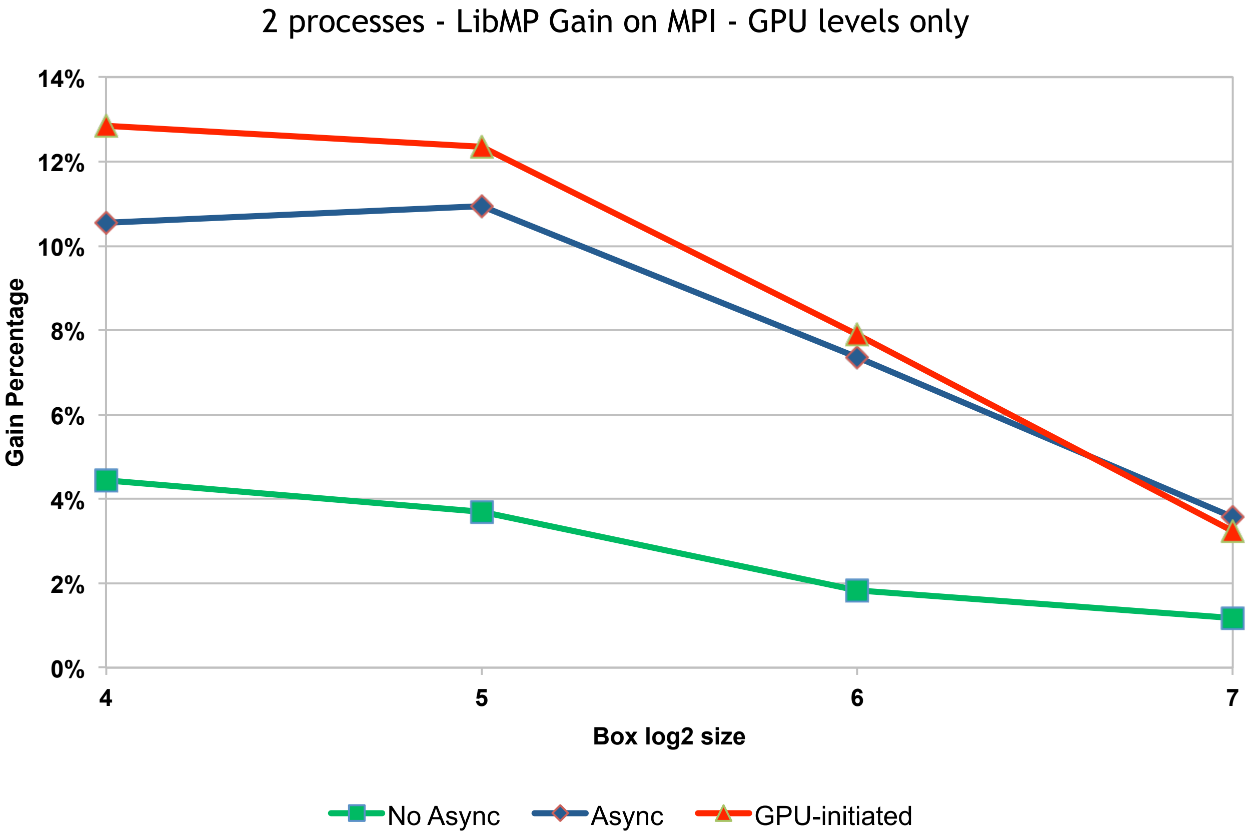


Level Size Threshold:

- Lower levels computed on CPU -> Synchronous Communications
- Higher levels computed on GPU -> Asynchronous Communications

HPGMG-FV

Async vs MPI



- 2 processes, IVB Xeons
- Each node has a K40m GPU
- Clock boost to 875 MHz
- OS: RHEL 6.6
- OMP_NUM_THREADS = 1

CAVEATS

Good platform

- GPUDirect RDMA & Async
  - need correct/reliable forwarding of PCIe transactions
- GPUDirect Async
  - GPU peer mapping limited to privileged processes (CUDA 8.0 EA)
- Platform:
  - non-std HCA FW
  - best: PCIE switch
  - limited: CPU root-complex

ASYNC: MPI BINDINGS

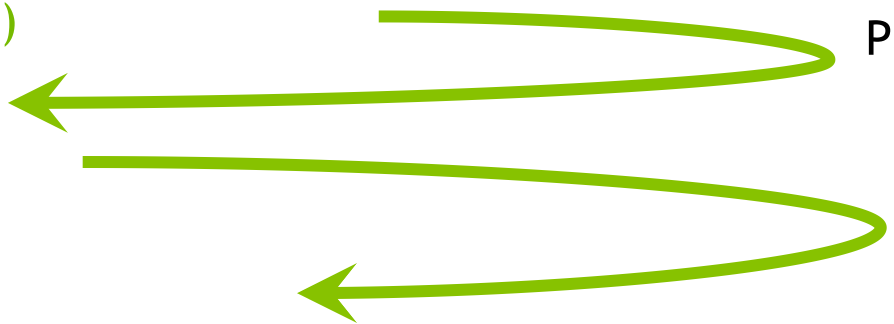
CODE SAMPLE

Current

```
Loop {

    kernel <<<nblocks, nthreads, stream>>> (buf1, buf2);

    cudaStreamSynchronize (stream);

    MPI_Irecv(buf2, count, MPI_INT, peer, 0, comm, &req[0]);
    MPI_Isend(buf1, count, MPI_INT, peer, 0, comm, &req[1]);

    MPI_Waitall (2, req, statuses);
}
```

WHAT DO WE NEED?

```
Loop n {
    kernel <<<…,stream>>> (…)

    MPI_Irecv(…,&req[i])
    MPI_Isend(…,&req[i+1])


    libasync_wait (stream, req[i])

    i = i+2
}

MPI_Waitall (2*n, req, …)
```



Pass stream info
into MPI

Get CQE info
from MPI

OPTION 1

Stream communicator

```
MPI_Comm_dup(MPI_COMM_WORLD, &stream_comm);
MPI_Info_set(info, “CUDA_STREAM”, stream);
MPI_Comm_set_info(stream_comm, info);

Loop {

    kernel <<<nblocks, nthreads, stream>>> (buf1, buf2);

    MPI_Irecv(buf2, count, MPI_BYTE, peer, 0, stream_comm, &req[i]);
    MPI_Isend(buf1, count, MPI_BYTE, peer, 0, stream_comm, &req[i+1]);

    MPI_Waitall(2, req+i, MPI_STATUSES_IGNORE); // nonblock, break semantics

    i = i+2
}
```

OPTIONS 2

MPI type attributes

```
Int stream_attr;
MPI_Datatype int_async;
MPI_Type_create_keyval(MPI_TYPE_NULL_COPY_FN, MPI_TYPE_NULL_DELETE_FN, &stream_attr, NULL);
MPI_Type_dup(MPI_INT, &int_async);
MPI_Type_set_attr(int_async, stream_attr, (void *)&stream);

Loop {
    kernel <<<nblocks, nthreads, stream>>> (buf1, buf2);

    MPI_Irecv(buf2, count, int_async, peer, 0, comm, &req[i]);
    MPI_Isend(buf1, count, int_async, peer, 0, comm, &req[i+1]);

    libasync_waitall (2, stream, req + i);

    i = i+2
}

MPI_Waitall(nloop*2, req, MPI_STATUSES_IGNORE);
```

OPTION 3

Custom API for Accelerators

```
cudaStream_t stream;
Loop {

    kernel <<<nblocks, nthreads, stream>>> (buf1, buf2);

    MPIX_Acc_Irecv(stream, buf2, count, peer, 0, comm, &req[i]);
    MPIX_Acc_Isend(stream, buf1, count, peer, 0, comm, &req[i+1]);

    MPIX_Acc_Waitall(stream, 2, req+i, MPI_STATUSES_IGNORE);

    i = i+2
}
```

GPU-aware MPI

GPU-AWARE MPI

detect ptr to GPU mem and demploy optimized protocols

- point-to-point primitives:
  - intra-node: CUDA IPC + GPUDirect P2P for BW, IB loopback for latency
  - inter-node: pipelined staging for BW, GPUDirect RDMA+GDRcopy+IB loopback for latency
- RMA: use GPUDirect RDMA and IB RDMA_READ/WRITE
- *Offloaded MPI*, or CUDA stream-synchronous communications
  - intra-node: cudaMemcpyAsync(stream,..)
  - inter-node: GPUDirect Async
- CUDA kernel initiated communications

NCCL

Overview of NCCL

Accelerating multi-GPU collective communications

GOAL:

- A library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

APPROACH:

- Pattern the library after MPI’s collectives
- Handle the intra-node communication in an optimal way
- Allow for both multi-threaded and multi-process approaches to parallelization

NCCL Features

(Green = Available in Github: https://github.com/NVIDIA/nccl)

Collectives

- Broadcast
- All-Gather
- Reduce
- All-Reduce
- Reduce-Scatter
- Scatter
- Gather
- All-To-All
- Neighborhood

Key Features

- Single-node, any number of GPUs
- Host-side API
- Asynchronous/non-blocking interface
- Multi-thread, multi-process support
- In-place and out-of-place operation
- PCIe/QPI support
- NVLink support

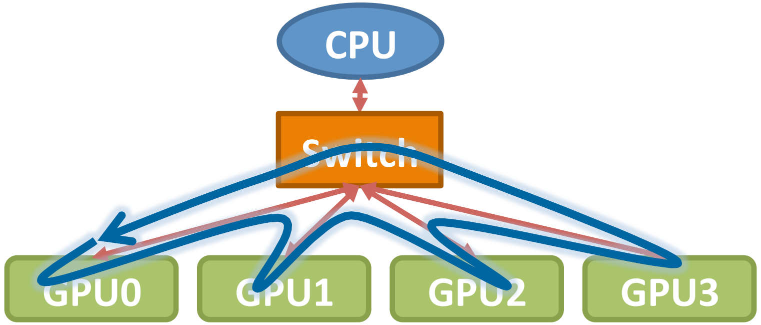
NCCL Implementation

Implemented as monolithic CUDA C++ kernels combining the following:

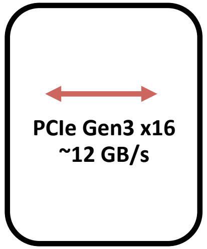
- GPUDirect P2P Direct Access
  - Implicit overlap between compute and data access – warp scheduling
  - Implicit coalescing of loads/stores
- Three primitive operations: Copy, Reduce, ReduceAndCopy
- Intra-kernel synchronization between GPUs
- One CUDA thread block per *ring-direction*

Ring-based Collectives

A primer



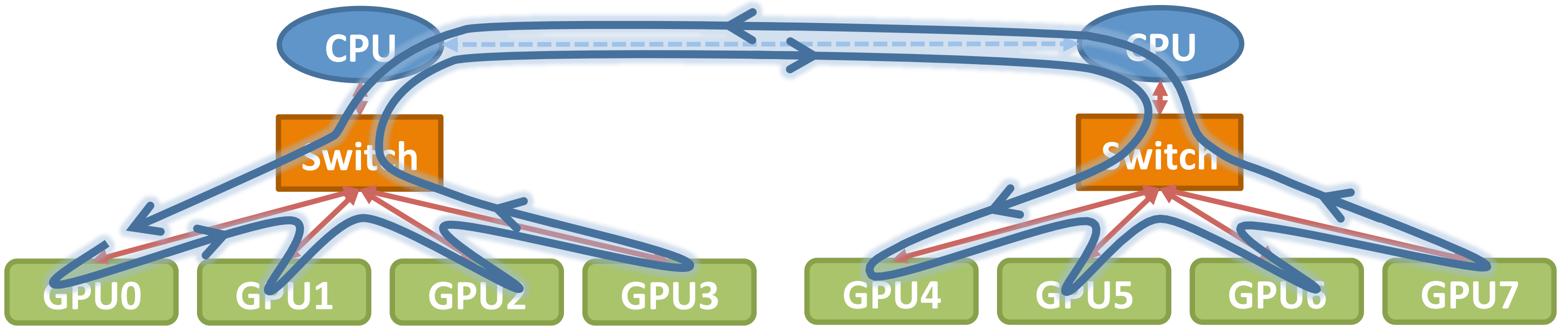
4-GPU-PCIe

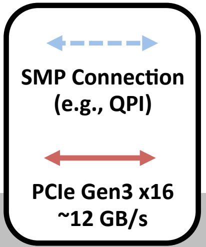


Most collectives amenable to bandwidth-optimal implementation on rings, and many topologies can be interpreted as one or more rings [P. Patarasuk and X. Yuan]

Ring-based Collectives

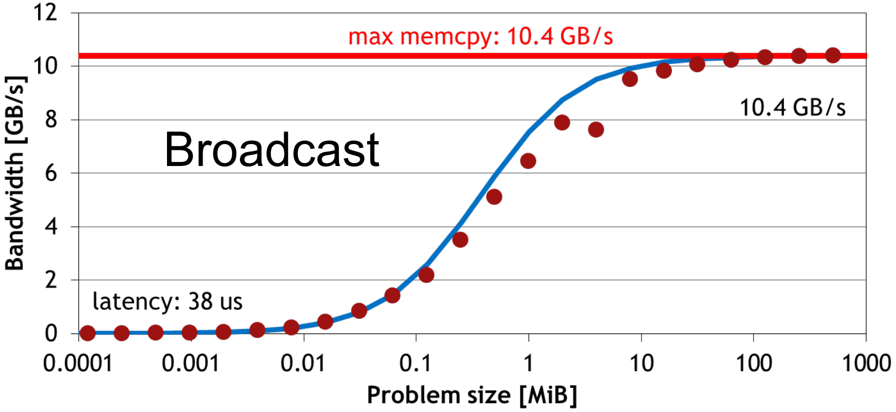
...apply to lots of possible topologies

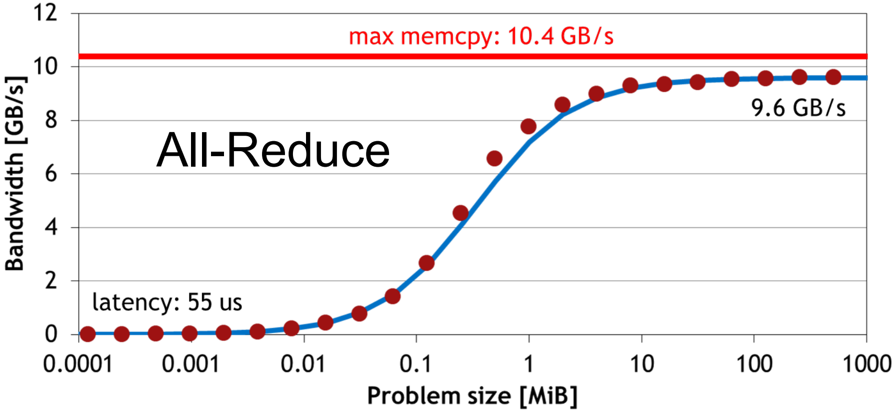


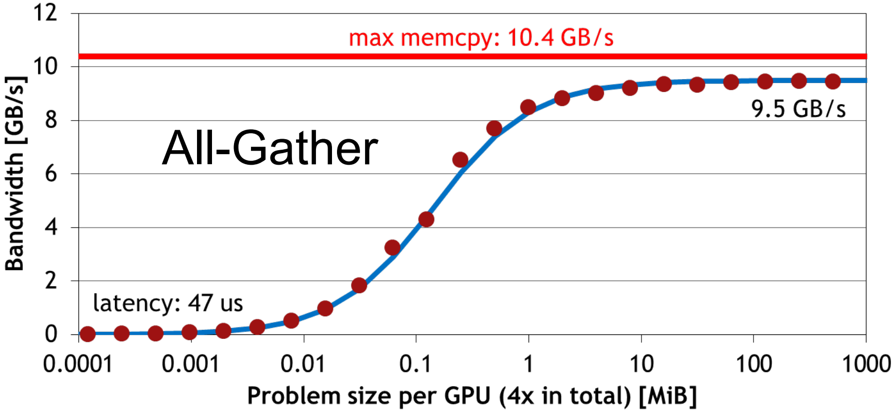


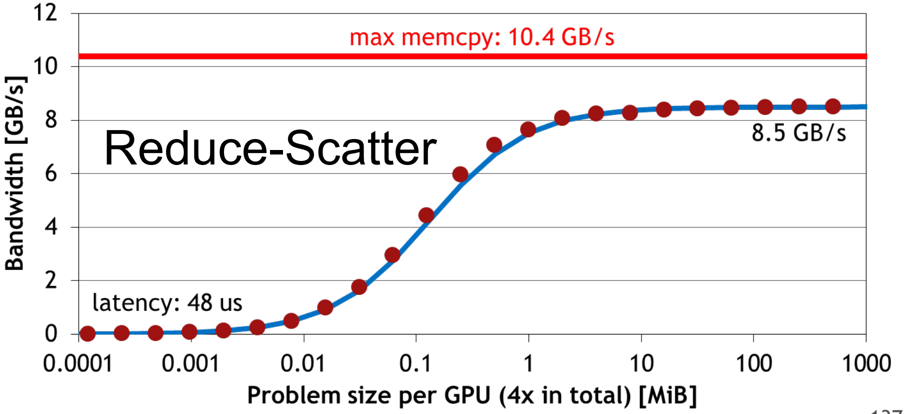
NCCL Performance

Bandwidth at different problem sizes (4 Maxwell GPUs)



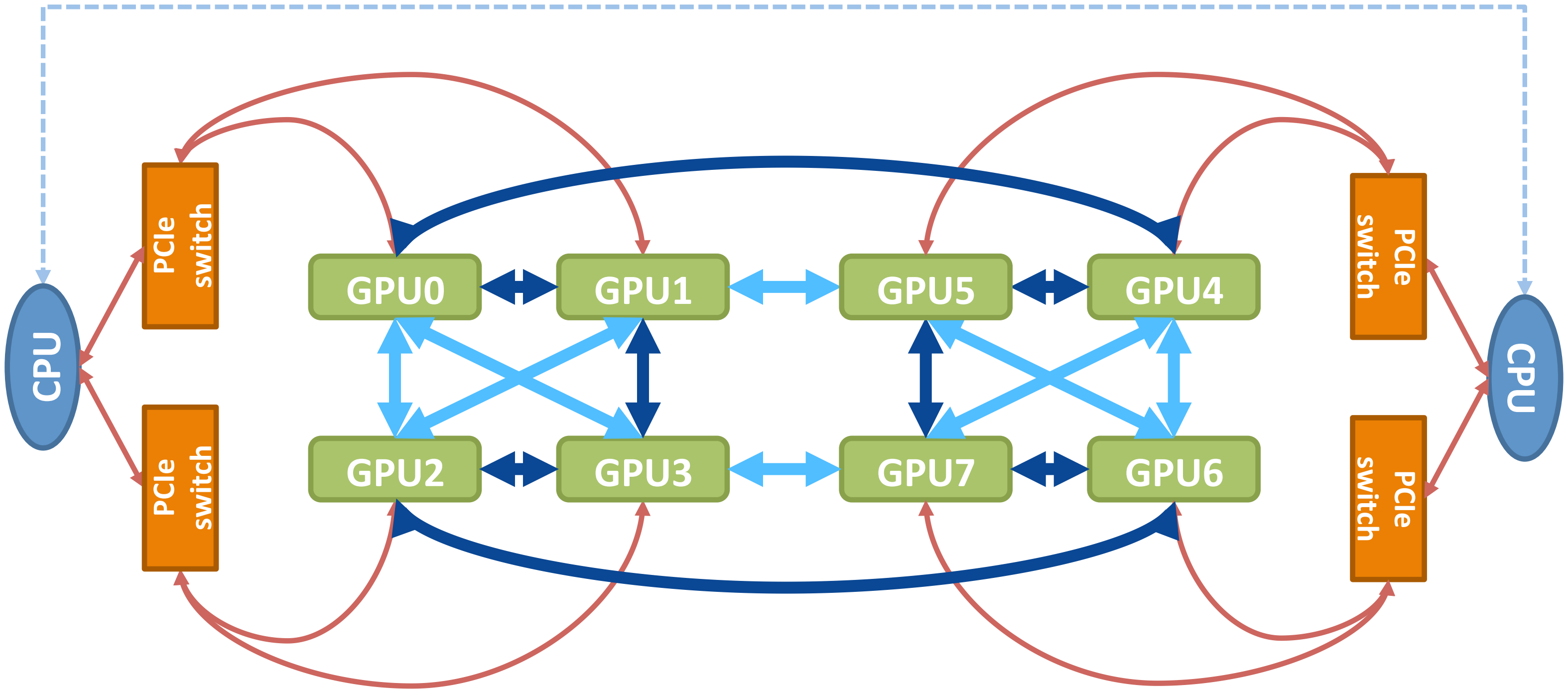






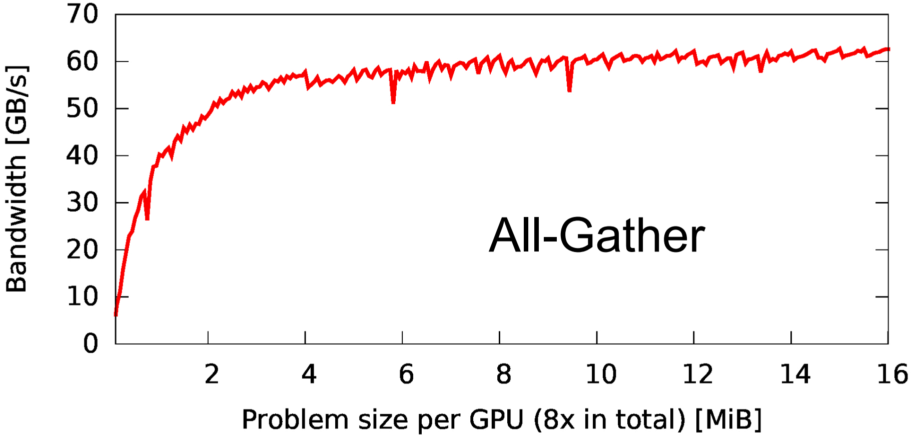
NVLink Support

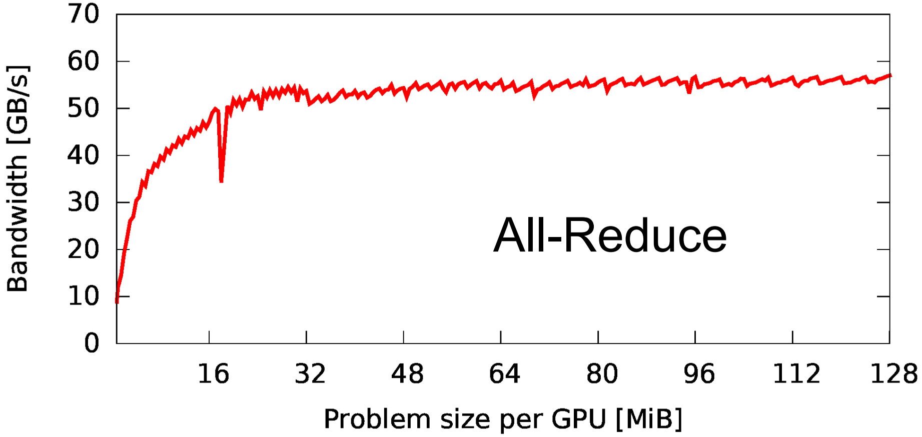
DGX-1 Server

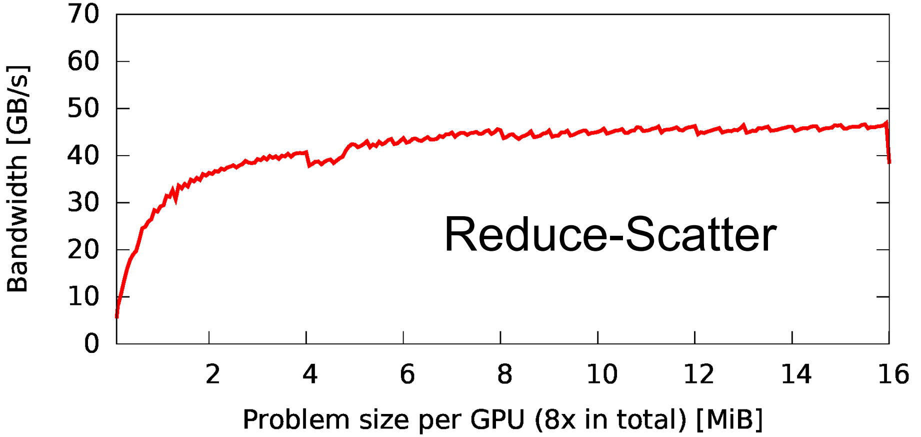


NCCL Performance

Bandwidth at different problem sizes (8 Pascal GPUs)







Summary

GPU Computing enabling rapid innovation in HPC and Deep Learning

Pascal and NVLink

Significant enhancements for programmability and performance

GPUDirect Technologies

P2P, RDMA - efficient ways to move data to/from GPU

Async – giving GPU control

CUDA-aware MPI

allows developers to benefit transparently or with minimal effort

GAME OVER

NVIDIA