

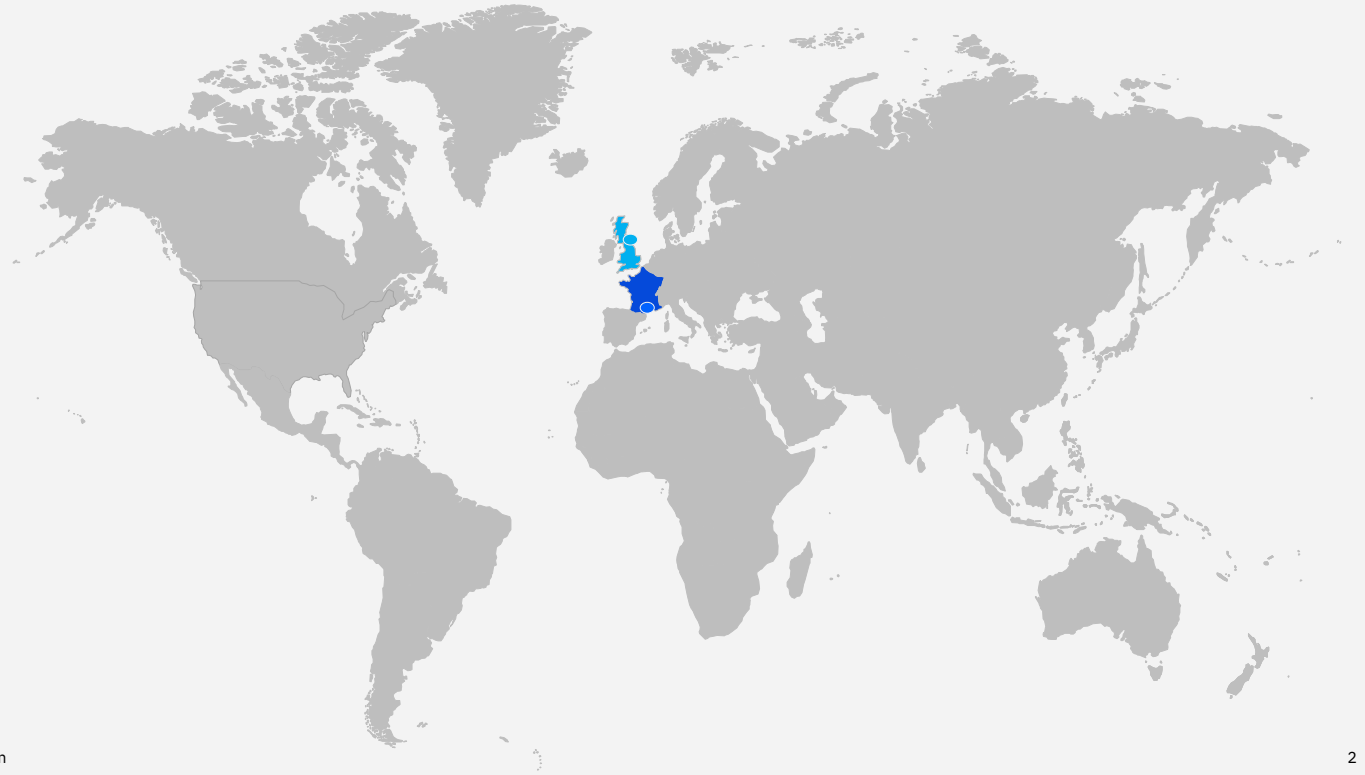
N8 CIR Bede
HPC Use

—

Ludovic ENAULT
Power HPC expert

IBM Garage for Systems Montpellier

IBM European HPC center



IBM Garage for Systems Montpellier

Some of our projects

11	Marconi-100 - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
18	PANGEA III - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France	291,024	17,860.0	25,025.8	1,367

N8 CIR Bede hardware



N8 CIR Bede Cluster

2x "login" nodes, each containing:

- 2x POWER9 CPUs @ 2.4GHz (40 cores total and 4 hardware threads per core), with NVLink 2.0
- 512GB DDR4 RAM
- 4x Tesla V100 32G NVLink 2.0
- 1x Mellanox EDR (100Gbit/s) InfiniBand port

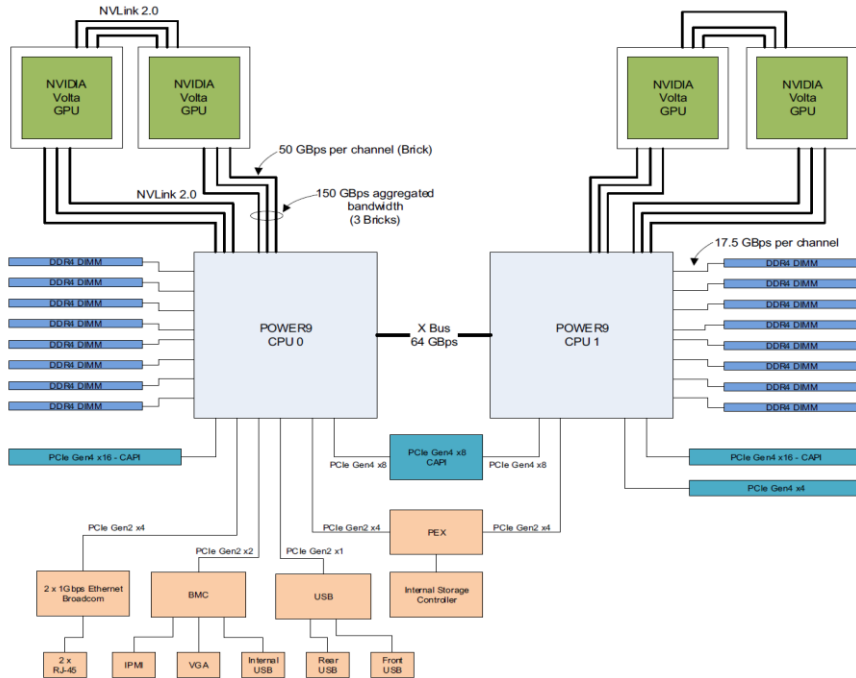
32x "gpu" nodes, each containing:

- 2x POWER9 CPUs @ 2.7GHz (32 cores total and 4 hardware threads per core), with NVLink 2.0
- 512GB DDR4 RAM
- 4x Tesla V100 32G NVLink 2.0
- 2x Mellanox EDR (100Gbit/s) InfiniBand ports

4x "infer" nodes, each containing:

- 2x POWER9 CPUs @ 2.9GHz (40 cores total and 4 hardware threads per core)
- 256GB DDR4 RAM
- 4x Tesla T4 16G PCIe
- 1x Mellanox EDR (100Gbit/s) InfiniBand port

GPU node - overview



2 NUMA Nodes, each with

- 1x POWER9 CPU
- 2x NVIDIA Telsa V100

Shared PCI-Express 16x 4.0

- 8x PCI-Express 4.0 to each CPU

POWER9 Direct Attach Memory Processor

14nm technology

18-24 POWER9 Cores w/ new, more efficient SMC uArch

- **4 Threads/Core**
- **2 cores per chiplet**
- Fused core to allow 2 cores to be viewed as single core partition
- Improved efficiency Perf/Watt and Perf/memory BW @ constant Tech.
- LINUX Radix Page Table support

Large, Low-latency Cache

- **512k private L2, 10MB 20W NUCA L3 w/ improved LRU per chiplet**

Direct Attach Memory support

- **8 DDR4 channels**
- ~130GB/s Stream benchmark

SMP 2 Socket support via 8B 16Gbs X-bus

NV link 2.0 Interface

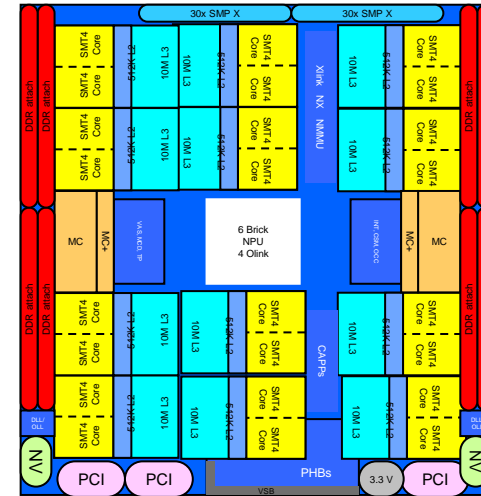
- 48 Lanes@ 25Gbs High speed optical or short reach electrical Interconnect
- GPU/CPU coherency
- Address translation Services

Accelerator Support

- 2 Coherent Accelerator Support (CAPI)
 - 2-16x PCIe Gen4 attach (16GB/s)
- GZIP, 8-4-2 compression & cryptograph

Network Interconnect

- Cluster shared Memory
- Address translation services



Gen4 PCIe 48 Lanes at 16gb/s

- 6 PHBs

Self-boot capability

Instant ON/OFF

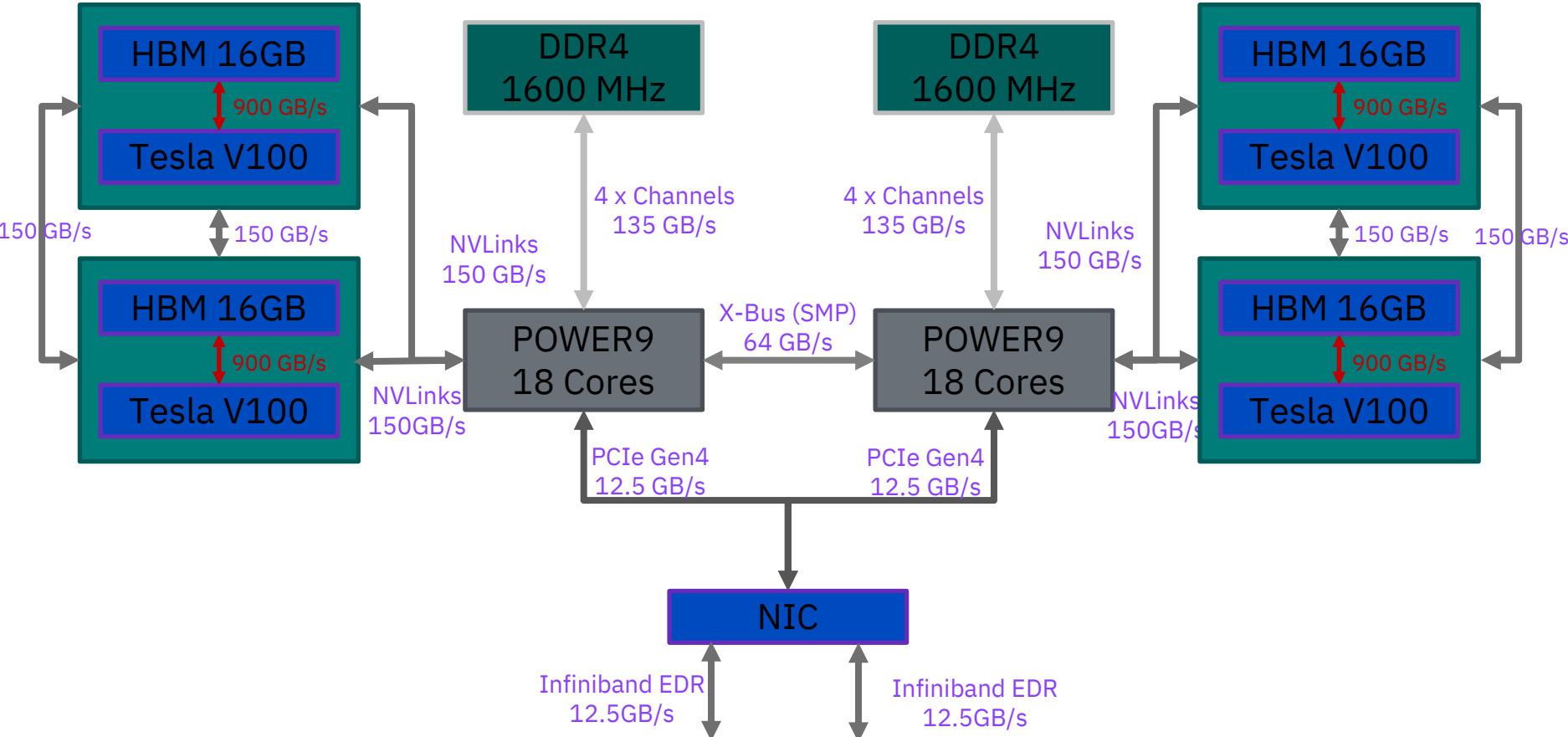
Cloud Management QoS

Partial Good Support

- Full Chiplet good: 2 cores 512k L2 10ML3
- ½ Chiplet: 1 core full 512k L2 5MML3
- Full Chiplet bad:
- Product offerings TBD



System Architecture w/Data Link



POWER9 – Premier Acceleration Platform

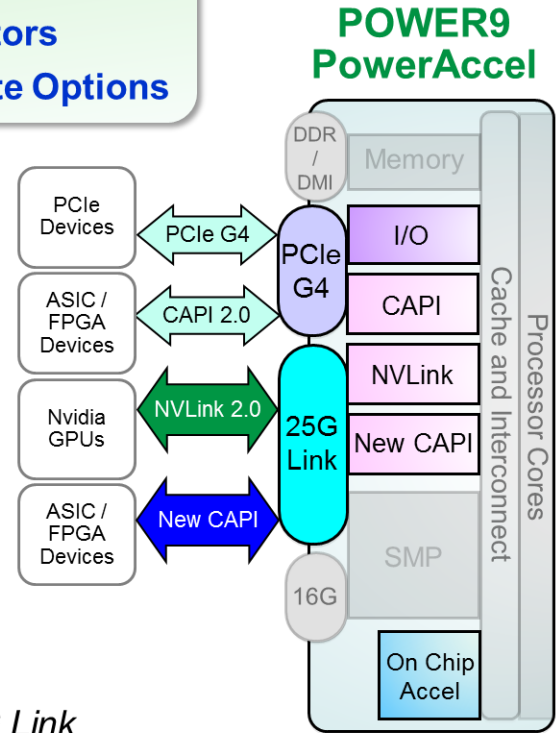
- Extreme Processor / Accelerator Bandwidth and Reduced Latency
- Coherent Memory and Virtual Addressing Capability for all Accelerators
- OpenPOWER Community Enablement – Robust Accelerated Compute Options

- State of the Art I/O and Acceleration Attachment Signaling

- PCIe Gen 4 x 48 lanes – 192 GB/s duplex bandwidth
- 25G Link x 48 lanes – 300 GB/s duplex bandwidth

- Robust Accelerated Compute Options with OPEN standards

- On-Chip Acceleration – Gzip x1, 842 Compression x2, AES/SHA x2
- CAPI 2.0 – 4x bandwidth of POWER8 using *PCIe Gen 4*
- NVLink 2.0 – Next generation of GPU/CPU bandwidth and integration
- New CAPI – High bandwidth, low latency and open interface using *25G Link*



Cache Hierarchy

L1

- 128 Byte Line
- 64 Byte Sectored D Cache
- Store Thru Data cache
- I fetch always 128B
- Adaptive Prefetch
- 8 Load Miss Queues

L2

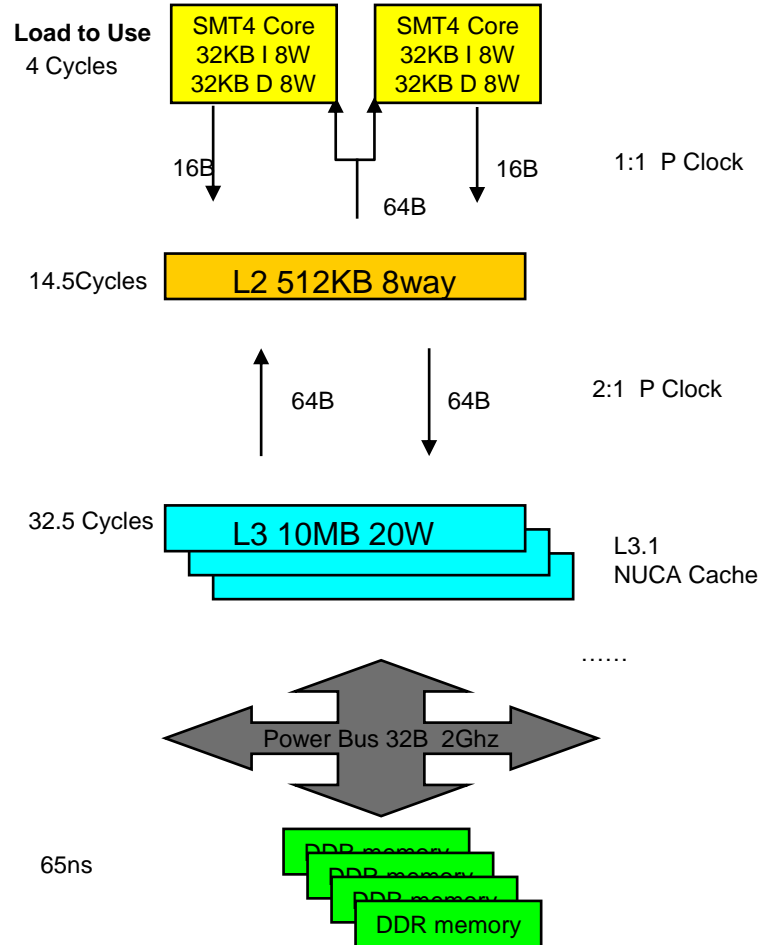
- 128B Line 64B Data Sector support
- Store in advanced 13 state protocol
- True LRU
- 16 RCQ, 48 STQ

L3

- 128B 64B Sectored
- Store in victim cache for L2
- Weighted history predictor WHP LRU
- Large asc. relieves Mem BW pressure
- 32 L3 Prefetch Queues
- Large amount of data queuing
 - 16 read 16 Castin
 - 16 Cast out 16 snoop
 - 16 write inject

Memory

- 64 entry command l queue per port
- 64 Bytes returned in congested and 64 OK sent



P9 SMC Core – Sliced Design

SMC Execution Slices

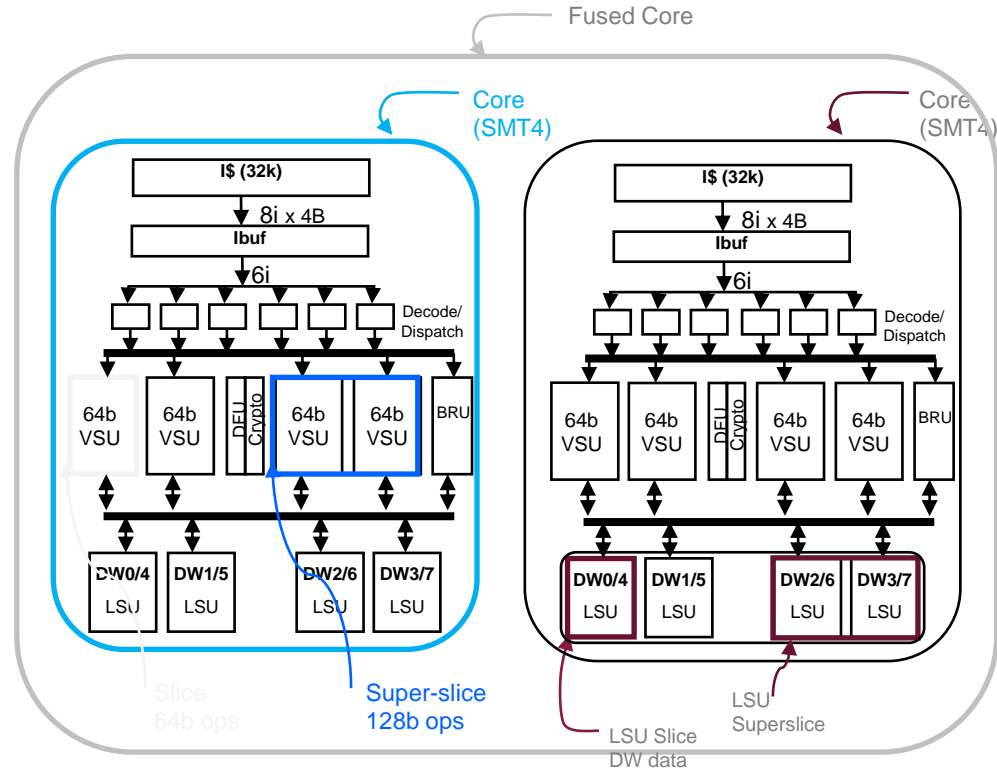
- Common 64b dataflow execution “slices”
 - Merged FX/VSU
- 128b ops are executed across two slices; a.k.a. “Superslice”

SMC Load/Store Slices

- Common 64b dataflow; each handles data for $\frac{1}{4}$ of 32k data-cache
- Two quad-word aligned LSU slices form an “LSU Superslice”

Fused Core

- Two SMC cores are paired to form an 8 threaded “fused core”



P9 Competitive SIMD Position

	P8	Haswell	P9	Skylake
Vector Regs	64x 128	16x 128 16x 256	64x 128	16 x 128 16 x 256 32 x 512
SIMD Disp	2x VFMA + 2x LDV		1x VFMA + 2x LDV*	2x 512
64b issue	2x FMA	2x FMA	4x FMA	2x FMA
128b SIMD issue	2x VFMA	2x VFMA 3x 128b int	2x VFMA	2x VFMA 3x 128b int
256b SIMD issue	-	2x 256b	-	2x 256b
512b SIMD issue	-	-	-	2x 512
LD 128 issue	2x 16B	2x 32B	2x 16B	
LD+ST 128 issue	2x LDV + 1x STV		2x LDV + 2x STV	

Simultaneous Multi-Threading (SMT)

One Physical Processor Core = Multiple Hardware Threads

- Multiple instructions can run at the same time on the same physical processor

Performance Benefit

- Highly depends on the workload

Management

- Changing SMT mode does not require reboot

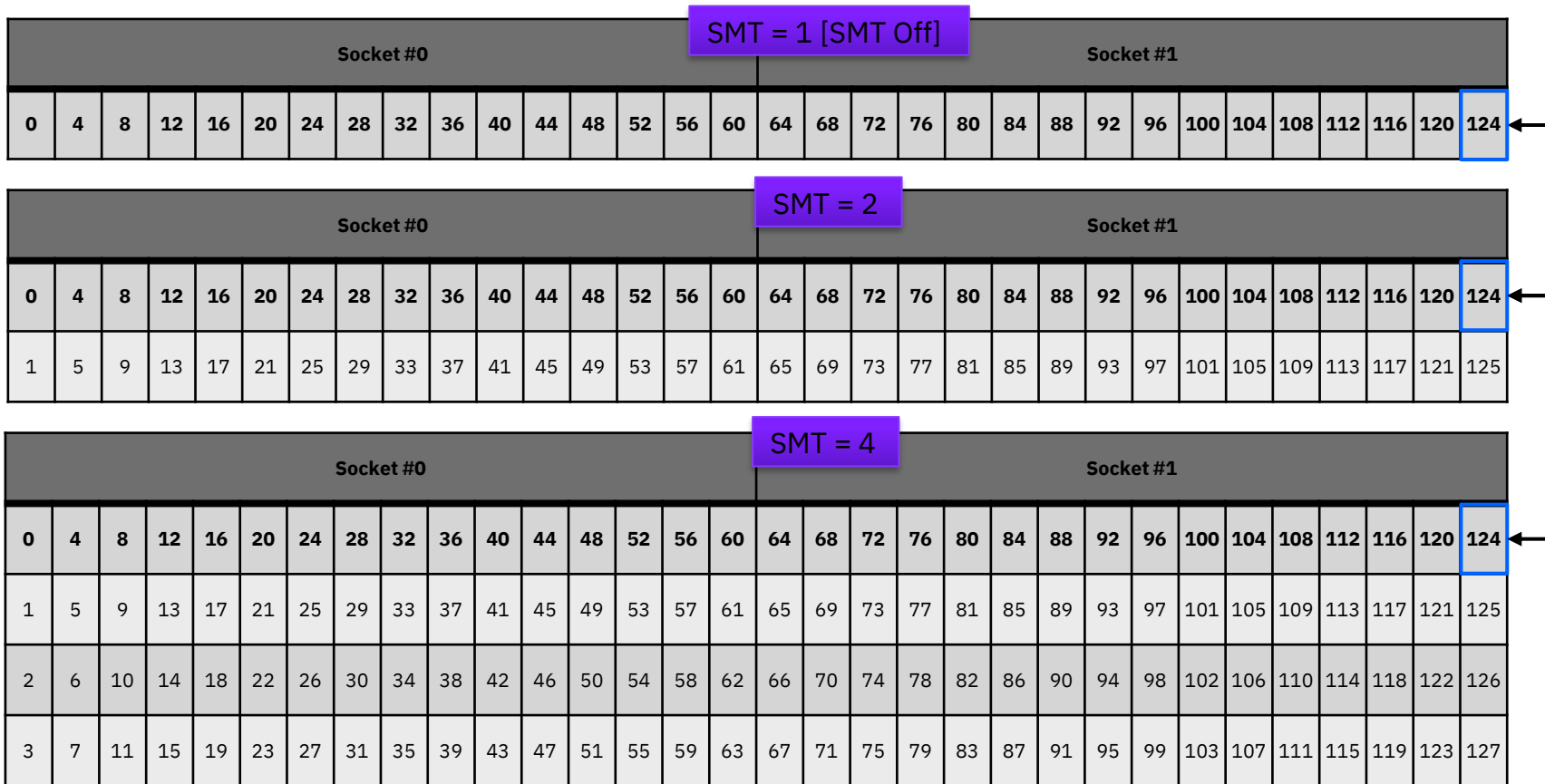
Mode	Physical Cores	Logical Cores
Single Thread (SMT Off)	32	32
SMT2	32	64
SMT4	32	128

HPC workload experience:

- run “single” process / thread context per physical core
- Extra thread will benefit to non-application threads (communication, ...)

System Topology: CPU

Numbering is “independent” of SMT mode



GPU node – NVIDIA Tesla V100 GPU

V100/POWER9 CAPABILITIES (2017)

2nd Generation NVLink

25Gbps signaling

6 NVLinks on GPU and on CPU

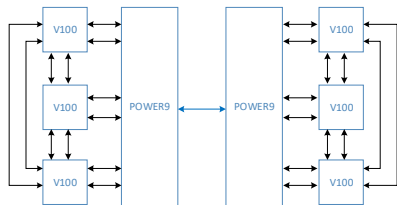
300GBps bidirectional BW

Shared address space

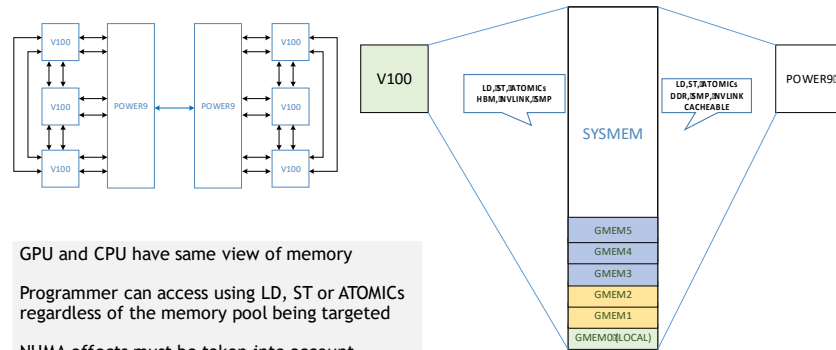
GPU, CPU can do load/store/atomic to any target

Hardware cache coherence

Address translation services



SHARED ADDRESS SPACE



GPU and CPU have same view of memory

Programmer can access using LD, ST or ATOMICS regardless of the memory pool being targeted

NUMA effects must be taken into account

ATOMICS

CPU and GPU support different atomics

Most efficient to complete at destination

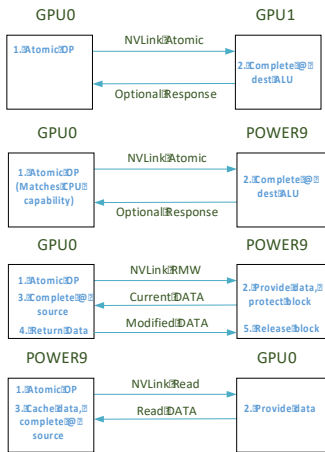
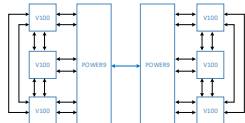
All GPU atomics exported to peer natively

GPU atomics that CPU can accept sent to CPU

GPU specific atomics to CPU use Read-Modify-Write

CPU atomics to GMEM completed on CPU

Remote GPUs look like SYSTEMEM



COHERENCE

CPU is a latency optimized core - LOC

Latency is critical

CPU can cache GMEM in its cache hierarchy

CPU read from cache - 10ns (😊)

CPU read from GMEM - 500ns (😞😞😞)

Every effort to keep data in cache

GPU has probe filter tracking block state

Simple 2-state checkout protocol

PF watermark to avoid demand probing

GPU is a throughput optimized core - TOC

Bandwidth is critical

GPU caches SYSTEMEM in its L1

Data-Race-Free Coherence

GPU write-through cache for immediate update back to SYSTEMEM

GPU can fetch/re-fetch data from SYSTEMEM as quickly as it can from GMEM

GPU node – POWER9+V100 integration

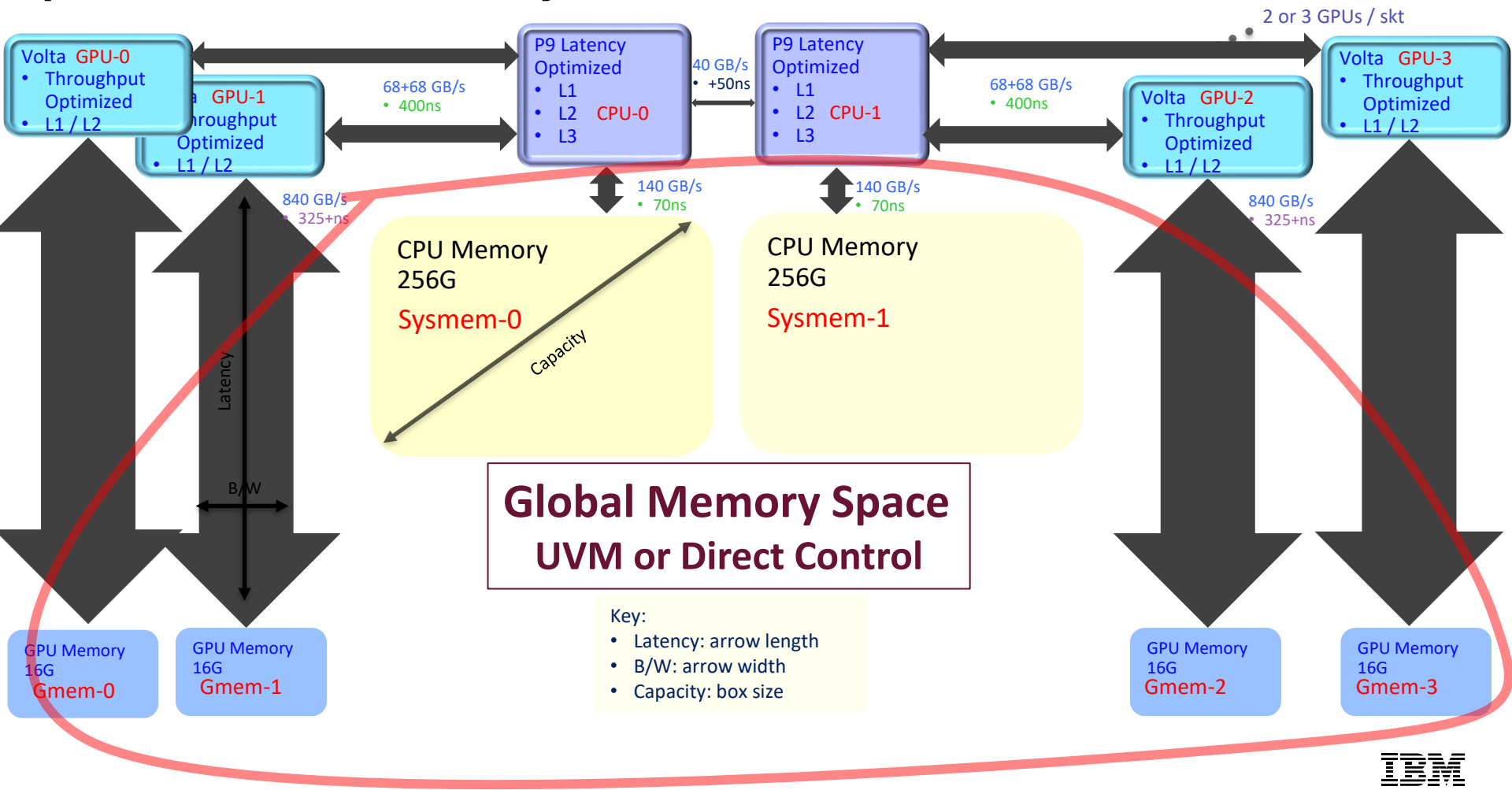
Key differentiating capabilities in POWER9 + NVLink

- Hardware-supported flat address space
- Hardware CPU-GPU cache coherence
- High bandwidth: 75 + 75 GB/s (4-GPU systems)
- Support for hardware atomic operations
- Fast fine-grained access (at hardware memory + interconnect latencies)
- Efficient, high-bandwidth bulk transfers

Benefits to application performance and programmability

- NVLink B/W + coherence enable larger-than-GPU memory problem sizes
- Faster and more flexible CPU + GPU porting of applications
 - Without requiring specialized memory allocation calls
- Hardware-supported flat address space does not require specialized memory allocation calls or page-based migration for coherence

Open Power Server Memory Model



System topology: hwloc-ls (lstopo-no-graphics)

Machine (633GB total)

Group0 L#0

 NUMANode L#0 (P#0 252GB)

 Package L#0

 L3 L#0 (10MB) + L2 L#0 (512KB)

 L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0

 PU L#0 (P#0)

 PU L#1 (P#1)

 PU L#2 (P#2)

 PU L#3 (P#3)

 L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1

 PU L#4 (P#4)

 PU L#5 (P#5)

 PU L#6 (P#6)

 PU L#7 (P#7)

 L3 L#1 (10MB) + L2 L#1 (512KB)

[...]

 NUMANode L#2 (P#252 32GB)

 NUMANode L#3 (P#253 32GB)

 NUMANode L#4 (P#254 32GB)

 NUMANode L#5 (P#255 32GB)

[...]

System Topology: nvidia-smi topo -m

```
# nvidia-smi topo -m
```

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	mlx5_0	mlx5_1	mlx5_2	mlx5_3	CPU Affinity	NUMA Affinity
GPU0	X	NV2	NV2	SYS	SYS	SYS	NODE	NODE	SYS	SYS	0-63	0
GPU1	NV2	X	NV2	SYS	SYS	SYS	NODE	NODE	SYS	SYS	0-63	0
GPU2	NV2	NV2	X	SYS	SYS	SYS	NODE	NODE	SYS	SYS	0-63	0
GPU3	SYS	SYS	SYS	X	NV2	NV2	SYS	SYS	NODE	NODE	64-127	8
GPU4	SYS	SYS	SYS	NV2	X	NV2	SYS	SYS	NODE	NODE	64-127	8
GPU5	SYS	SYS	SYS	NV2	NV2	X	SYS	SYS	NODE	NODE	64-127	8
mlx5_0	NODE	NODE	NODE	SYS	SYS	SYS	X	PIX	SYS	SYS		
mlx5_1	NODE	NODE	NODE	SYS	SYS	SYS	PIX	X	SYS	SYS		
mlx5_2	SYS	SYS	SYS	NODE	NODE	NODE	SYS	SYS	X	PIX		
mlx5_3	SYS	SYS	SYS	NODE	NODE	NODE	SYS	SYS	PIX	X		

Legend:

X = Self

SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)

NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node

PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)

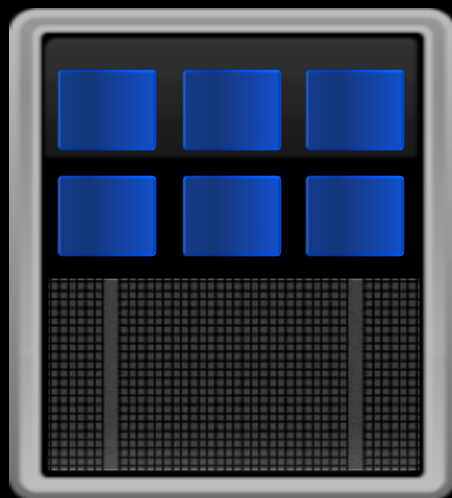
PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)

PIX = Connection traversing at most a single PCIe bridge

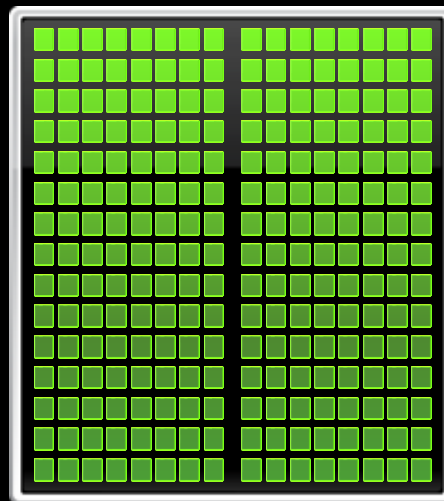
NV# = Connection traversing a bonded set of # NVLinks

Add GPUs: Accelerate Science Applications

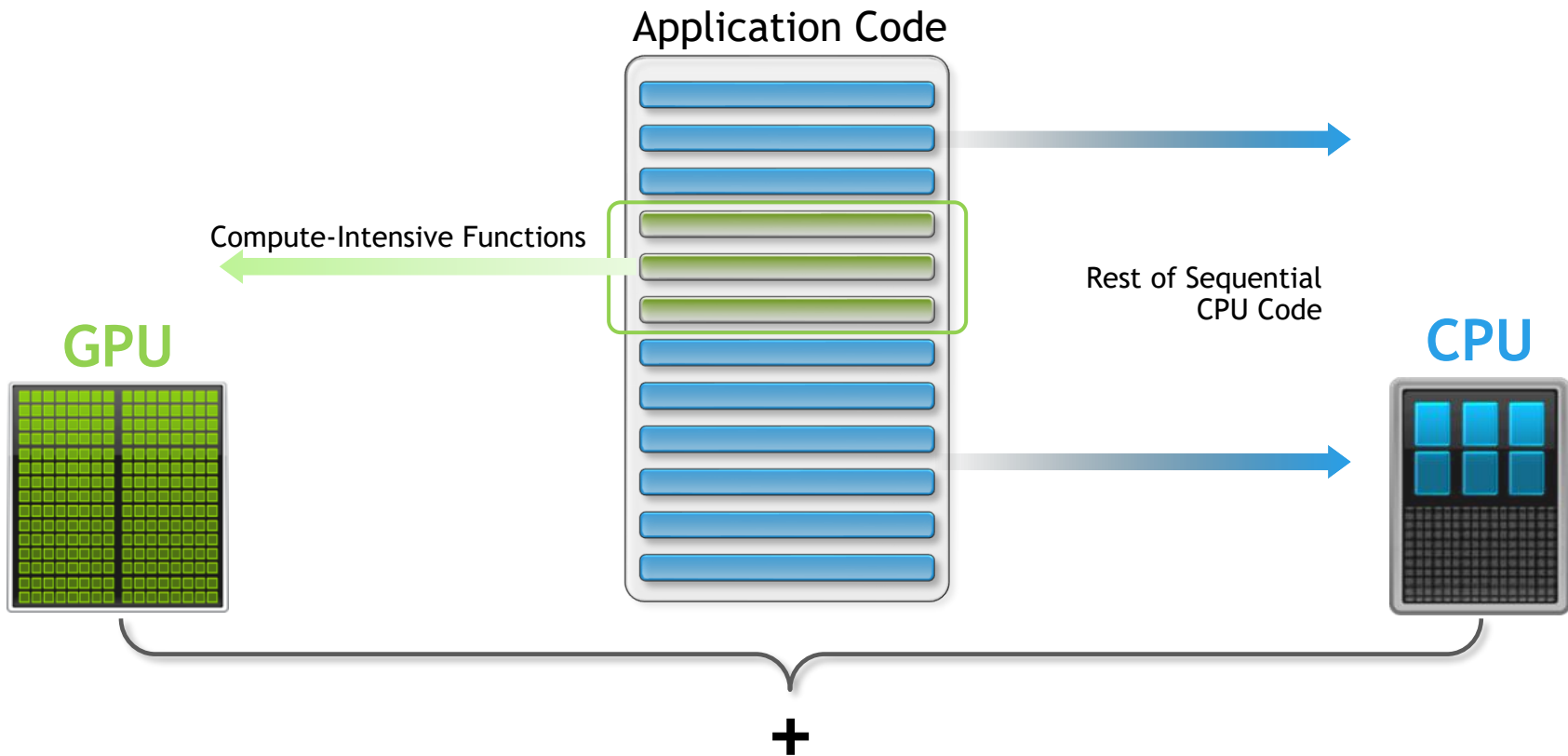
CPU



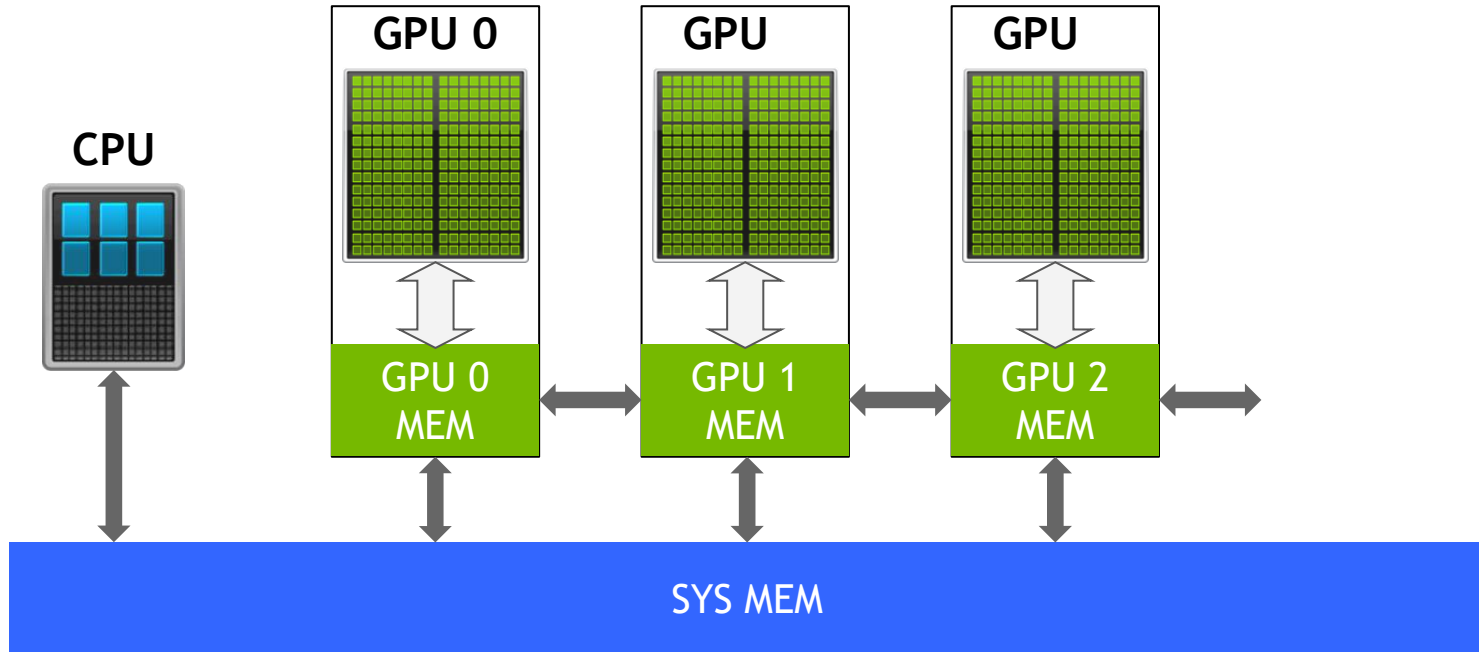
GPU



How GPU Acceleration Works

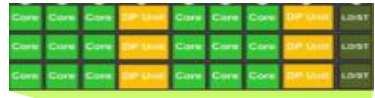
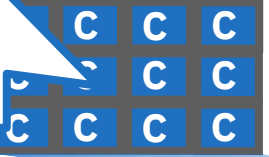


HETEROGENEOUS ARCHITECTURES

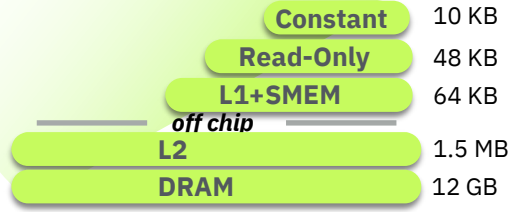
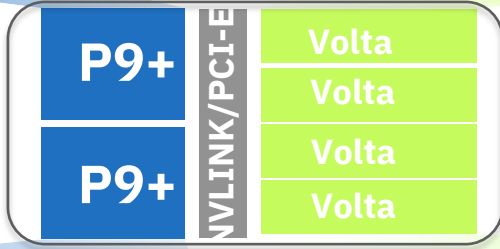


Compiler view of the Open POWER Architecture

Fat cores
 SMT 4 (4 threads/core)
 2x16x4
 threads/processor
 Complex branching
 prediction
 Out-of-order pipeline

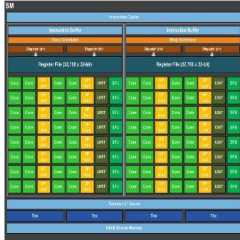


Highly parallel cores
 2048 threads/SMX
 $O(10^5)$ threads/device
 Thread divergence is
 serialized
 Limited synchronization
 (within CUDA block only)



GPU Architecture

SM-0



SM-1



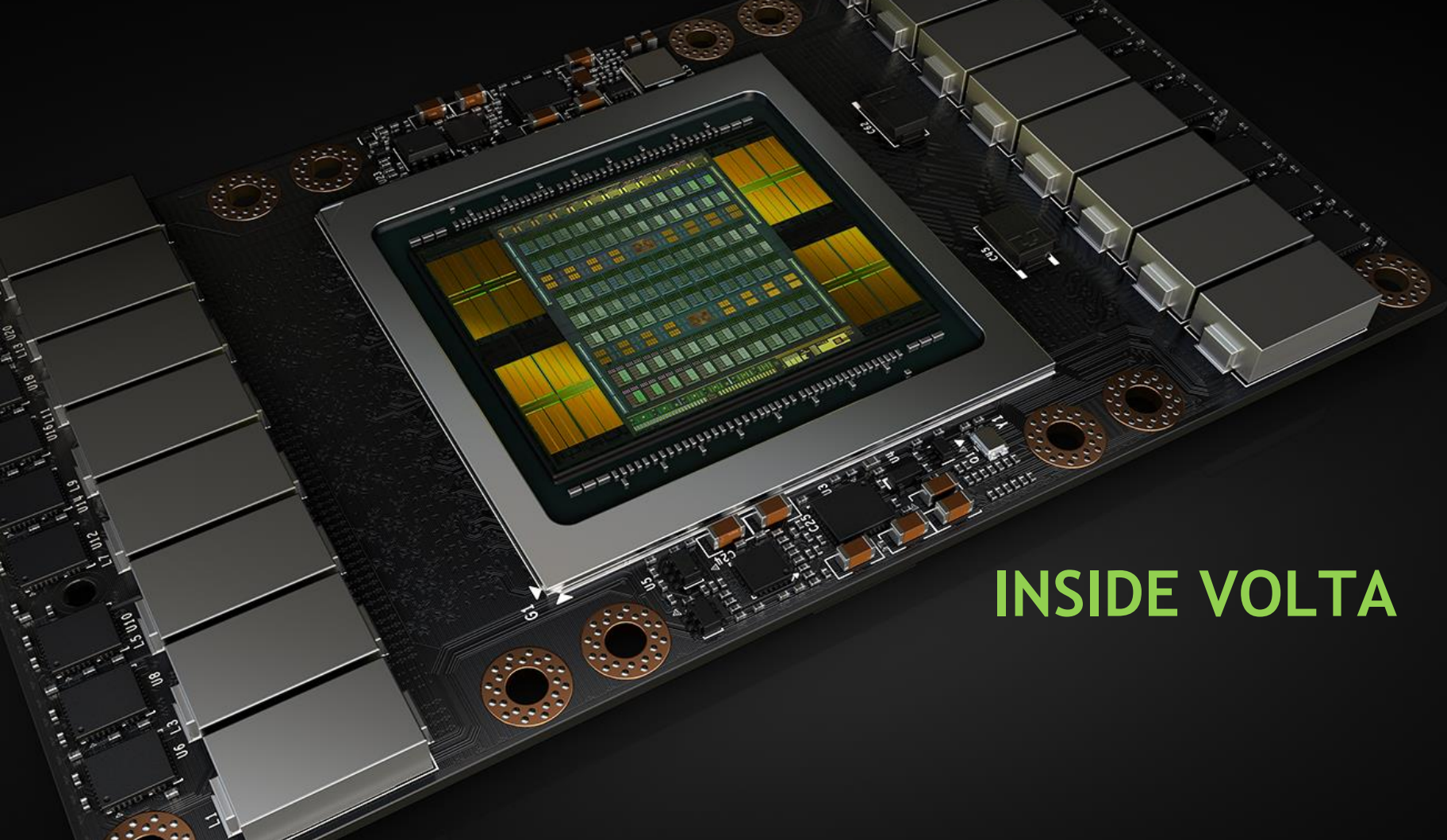
SM-N



GPU L2

GPU DRAM

PCI-E or NVLINK



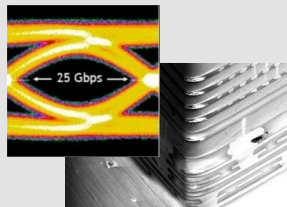
INSIDE VOLTA

Volta Architecture



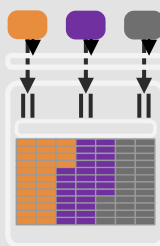
Most Productive GPU

Improved NVLink & HBM2



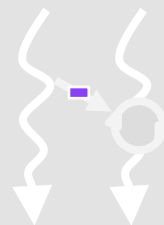
Efficient Bandwidth

Volta MPS



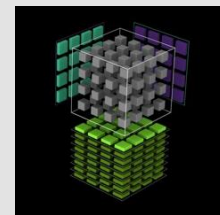
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core



120 Programmable
TFLOPS Deep Learning

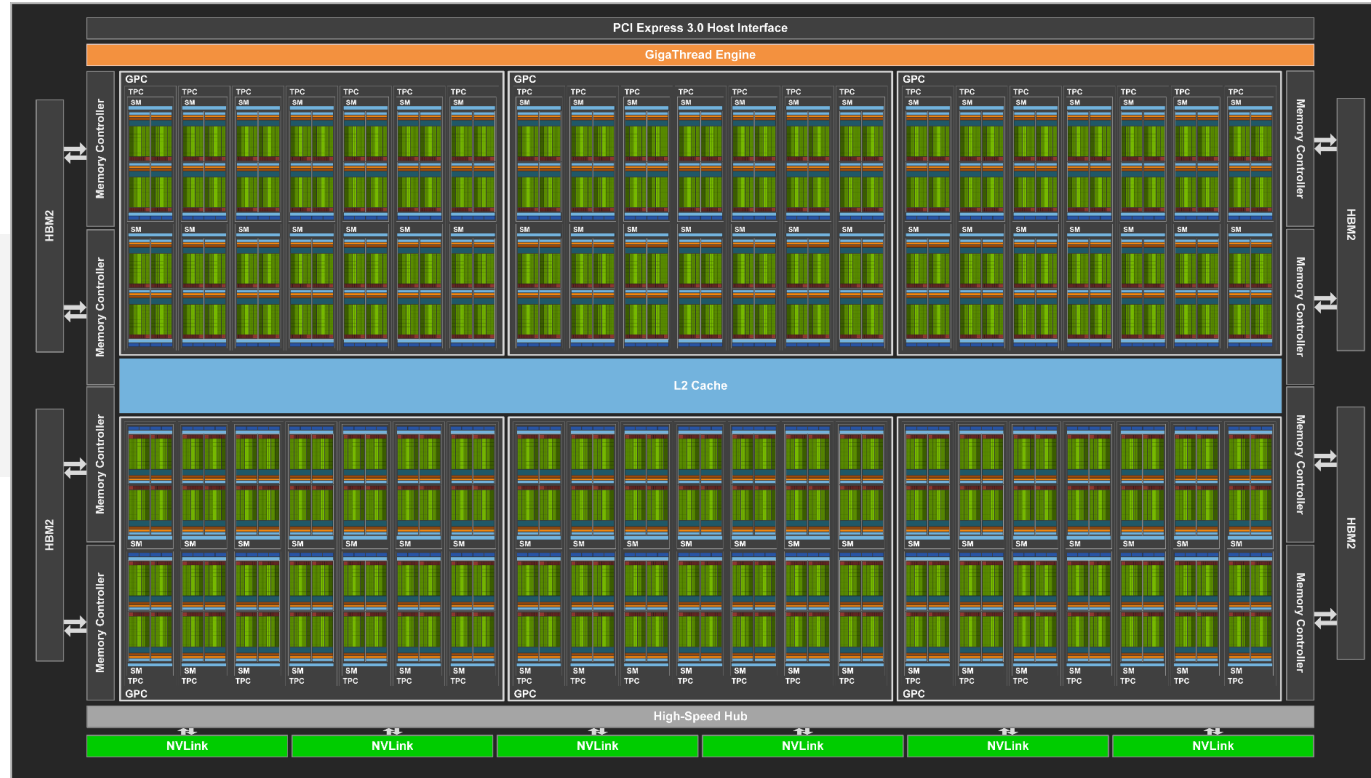
The Fastest and Most Productive GPU for Deep Learning and HPC

Tesla v100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

16 (32) GB HBM2
900 GB/s HBM2
300 GB/s NVLink



VOLTA GV100 SM

GV100	
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

Completely new ISA
 Twice the schedulers
 Simplified Issue Logic
 Large, fast L1 cache
 Improved SIMT model
 Tensor acceleration

=

The easiest SM to program yet

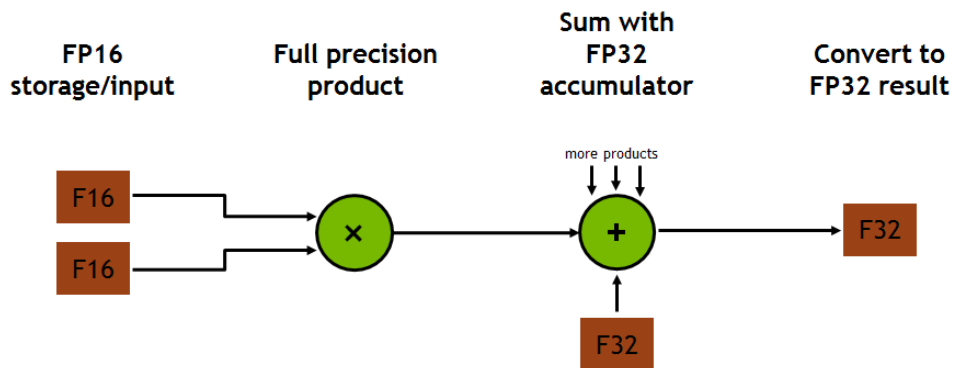


GPU SM Architecture

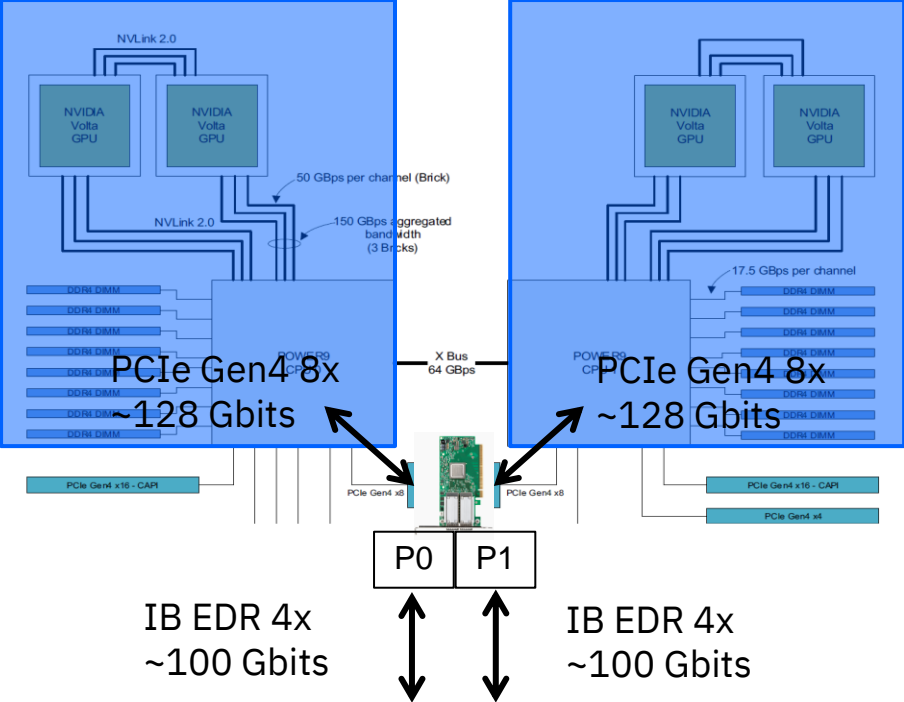
Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32



GPU node - Network



Mapping “interface” ↔ “path”

interface	Type	socket	Port
mlx5_0	physical	0	0
mlx5_1	physical	0	1
mlx5_2	virtual	1	0
mlx5_3	virtual	1	1

Mainly two strategies:

- Balanced: “dedicate” 1 IB link per socket
- E.g.: socket 0 => mlx5_0, socket 1 => mlx5_3
- Bandwidth: interleave path from both socket (to get rid of PCI-E bandwidth constraints)
- E.g.: mlx5_0 and mlx5_3

N8 CIR Bede software



Summit vs Bede stack comparison

- GPFS
 - Spectrum LSF
 - Spectrum MPI
 - ESSL / OpenBLAS
 - GCC / XL Compilers / NVHPC (PGI)
- Lustre
 - [SLURM](#)
 - [OpenMPI](#)
 - ESSL / OpenBLAS
 - GCC / XL Compilers / NVHPC (PGI)

Compiler Offerings supporting Acceleration Enabled Programming Models

OpenACC

Directives For Accelerators

- Designed to simplify Programming of heterogeneous CPU/GPU systems
- Directive based parallelization for accelerator device



Initial implementation in GCC 5
Improved implementation in GCC 8

Open Source

PGI

OpenACC directive-based GPU programming in Fortran, C/C++
CUDA Fortran extensions for native CUDA programming in Fortran
OpenMP 3.1 multicore parallelization for Linux/OpenPOWER
Support for POWER CPUs, including VSX SIMD vectorization
OpenACC and CUDA Fortran interoperability with CUDA Unified Memory



- Gives direct access to the GPU instruction set
- Supports C, C++ and Fortran
- Generally achieves best leverage of GPUs for best application performance

Highly optimized for POWER processors
Scalar MASS library and vector/SIMD MASSV library tuned for POWER
CUDA C/C++: XL C/C++ works as host compiler for POWER
CUDA Fortran: XL Fortran supports CUDA Fortran extensions
OpenMP 4.0/4.5 support



OpenMP

- OpenMP 4.0 introduces offloading and support for heterogeneous CPU/GPU
- Leverage existing OpenMP high-level directives support



Full support for OpenMP 4.5 including GPU offloading

Open Source

N8 CIR Bede some example



Infiniband devices

```
# lstopo-no-graphics
Machine (633GB total)
  Group0 L#0
    NUMANode L#0 (P#0 252GB)
      Package L#0
    ...
  HostBridge L#3
    PCIBridge
      PCI 15b3:1019
        Net L#2 "ib0"
        OpenFabrics L#3 "mlx5_0"
      PCI 15b3:1019
        Net L#4 "ib1"
        OpenFabrics L#5 "mlx5_1"
```

```
NUMANode L#1 (P#8 256GB)
  Package L#1
  ...
  HostBridge L#13
    PCIBridge
      PCI 15b3:1019
        Net L#14 "ib2"
        OpenFabrics L#15 "mlx5_2"
      PCI 15b3:1019
        Net L#16 "ib3"
        OpenFabrics L#17 "mlx5_3"
```

Same physical port

Same physical port

OpenMPI 4.x

- Rely on UCX:
 - “low level” transport details handled by UCX
 - Need to compile a version with “--enable-logging” to get access to the details

```
$ module use /projects/bddir04/modulefiles/utils
$ module load ucx
$ ucx_info -b | grep CONFIGURE
#define UCX_CONFIGURE_FLAGS      "--
prefix=/projects/bddir04/ucx/1.9.0 --enable-
logging --enable-mpi"

$ grep -i ucx job.sh

module load ucx

export UCX_LOG_LEVEL=INFO
```

OpenMPI 4.x – Device usage

```
# Run on a single node with 1 process per GPU
```

```
+ bede-mpirun --bede-par 1ppg:none /projects/bddir04/utils/task_placement/src/task_placement
```

```
[1615264830.252844] [gpu030:32197:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615264830.258488] [gpu030:32197:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_1:1 cuda_copy/cuda);
[1615264830.266304] [gpu030:32197:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_1:1 cuda_ipc/cuda);
[1615264830.268818] [gpu030:32197:0] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_1:1 cuda_ipc/cuda);
[1615264830.271435] [gpu030:32197:1] ucp_worker.c:1627 UCX INFO ep_cfg[4]: tag(posix/memory cma/memory rc_mlx5/mlx5_1:1 cuda_ipc/cuda);

[1615264830.253237] [gpu030:32198:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615264830.259647] [gpu030:32198:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_1:1 cuda_copy/cuda);
[1615264830.266304] [gpu030:32198:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_1:1 cuda_ipc/cuda);
[1615264830.270420] [gpu030:32198:0] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_1:1 cuda_ipc/cuda);

[1615264830.224959] [gpu030:32199:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615264830.258555] [gpu030:32199:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_3:1 cuda_copy/cuda);
[1615264830.266244] [gpu030:32199:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
[1615264830.268812] [gpu030:32199:0] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
[1615264830.271436] [gpu030:32199:1] ucp_worker.c:1627 UCX INFO ep_cfg[4]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);

[1615264830.251655] [gpu030:32200:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615264830.258463] [gpu030:32200:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_3:1 cuda_copy/cuda);
[1615264830.266244] [gpu030:32200:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
[1615264830.271431] [gpu030:32200:0] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
```

Using same physical link !!!!

OpenMPI 4.x – device selection example

```
[lenault@login1 test]$ cat helper-ibv-devices.sh
#!/bin/bash
#
# This scripts "binds" to mlx5 adapters
# It is trying to simulate the behavior of Spectrum MPI:
#   https://www.ibm.com/support/knowledgecenter/SSZTET_10.2/releasenotes/smpi_releasenotes.html
# - application running on "socket 0" will use variable PAMI_IBV_DEVICE_NAME
# - application running on "socket 1" will use variable PAMI_IBV_DEVICE_NAME_1
#
# building the device array:
# defaulting to mlx5_0 on socket 0 and mlx5_3 on socket 1
IBV_DEVICE_NAME=(
  "${PAMI_IBV_DEVICE_NAME:-mlx5_0:1}"
  "${PAMI_IBV_DEVICE_NAME_1:-mlx5_3:1}"
)
# Assuming the first cpu allowed represents the socket on which I am running
MY_FIRST_CPU=$(taskset -pc $$ | awk '{ print $NF }' | tr '-' ' ' | cut -d, -f 1)
CPU_PER_SOCKET=$(lscpu | grep '^CPU(s):' | awk '{ print $NF/2 }')
((MY_SOCKET=MY_FIRST_CPU / CPU_PER_SOCKET))
# echo $MY_SOCKET
export UCX_NET_DEVICES="${IBV_DEVICE_NAME[${MY_SOCKET}]}"
```

OpenMPI 4.x – Device usage

```
# Run on a single node with 1 process per GPU
```

```
+ bede-mpirun --bede-par 1ppg:none ./helper-ibv-devices.sh /projects/bddir04/utills/task_placement/src/task_placement
[1615305149.452471] [gpu014:118872:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615305149.456705] [gpu014:118872:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_0:1 cuda_copy/cuda);
[1615305149.464168] [gpu014:118872:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_0:1 cuda_ipc/cuda);
[1615305149.469219] [gpu014:118872:1] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_0:1 cuda_ipc/cuda);

[1615305149.452471] [gpu014:118869:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615305149.456875] [gpu014:118869:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_0:1 cuda_copy/cuda);
[1615305149.464121] [gpu014:118869:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_0:1 cuda_ipc/cuda);

[1615305149.452452] [gpu014:118870:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615305149.456650] [gpu014:118870:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_3:1 cuda_copy/cuda);
[1615305149.464242] [gpu014:118870:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
[1615305149.469225] [gpu014:118870:1] ucp_worker.c:1627 UCX INFO ep_cfg[3]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);

[1615305149.452528] [gpu014:118871:0] ucp_worker.c:1627 UCX INFO ep_cfg[0]: tag(cuda_copy/cuda); rma(gdr_copy/cuda);
[1615305149.456612] [gpu014:118871:0] ucp_worker.c:1627 UCX INFO ep_cfg[1]: tag(self/memory cma/memory rc_mlx5/mlx5_3:1 cuda_copy/cuda);
[1615305149.464262] [gpu014:118871:0] ucp_worker.c:1627 UCX INFO ep_cfg[2]: tag(posix/memory cma/memory rc_mlx5/mlx5_3:1 cuda_ipc/cuda);
```

Different IB link

Multiple process per GPU

GPU compute mode:

DEFAULT compute mode: Multiple host threads can use the device at the same time.

EXCLUSIVE_PROCESS compute mode: Only one CUDA context may be created on the device across all processes in the system. The context may be current to as many threads as desired within the process that created that context.

PROHIBITED compute mode: No CUDA context can be created on the device.

Context switching very expensive: unnecessary data CPU <-> GPU data migration

Advised compute mode: EXCLUSIVE_PROCESS

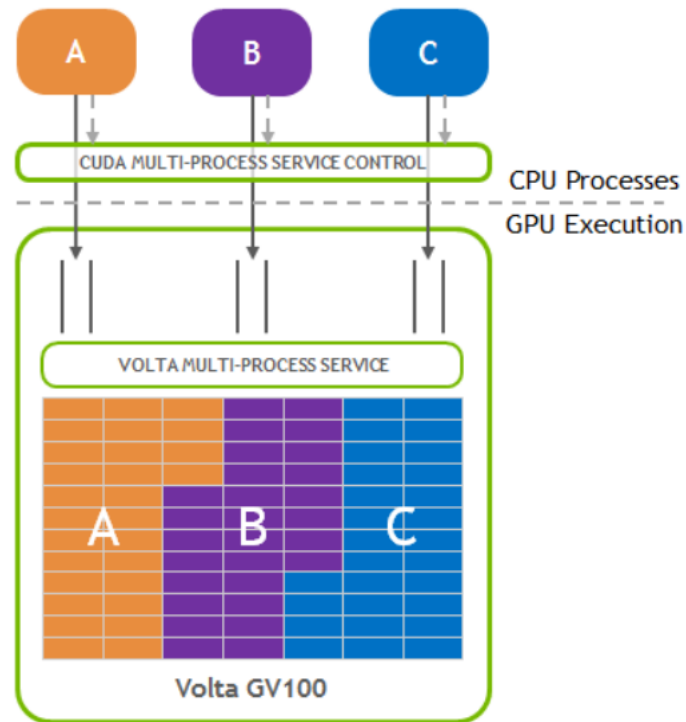
CUDA Multi Process Service

The Volta architecture introduced new MPS capabilities. Compared to MPS on pre-VoltaGPUs, Volta MPS provides a few key improvements:

- Volta MPS clients submit work directly to the GPU without passing through the MPS server.
- Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.
- Volta MPS supports limited execution resource provisioning for Quality of Service(QoS).

Source:

https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf



Helper script for MPS

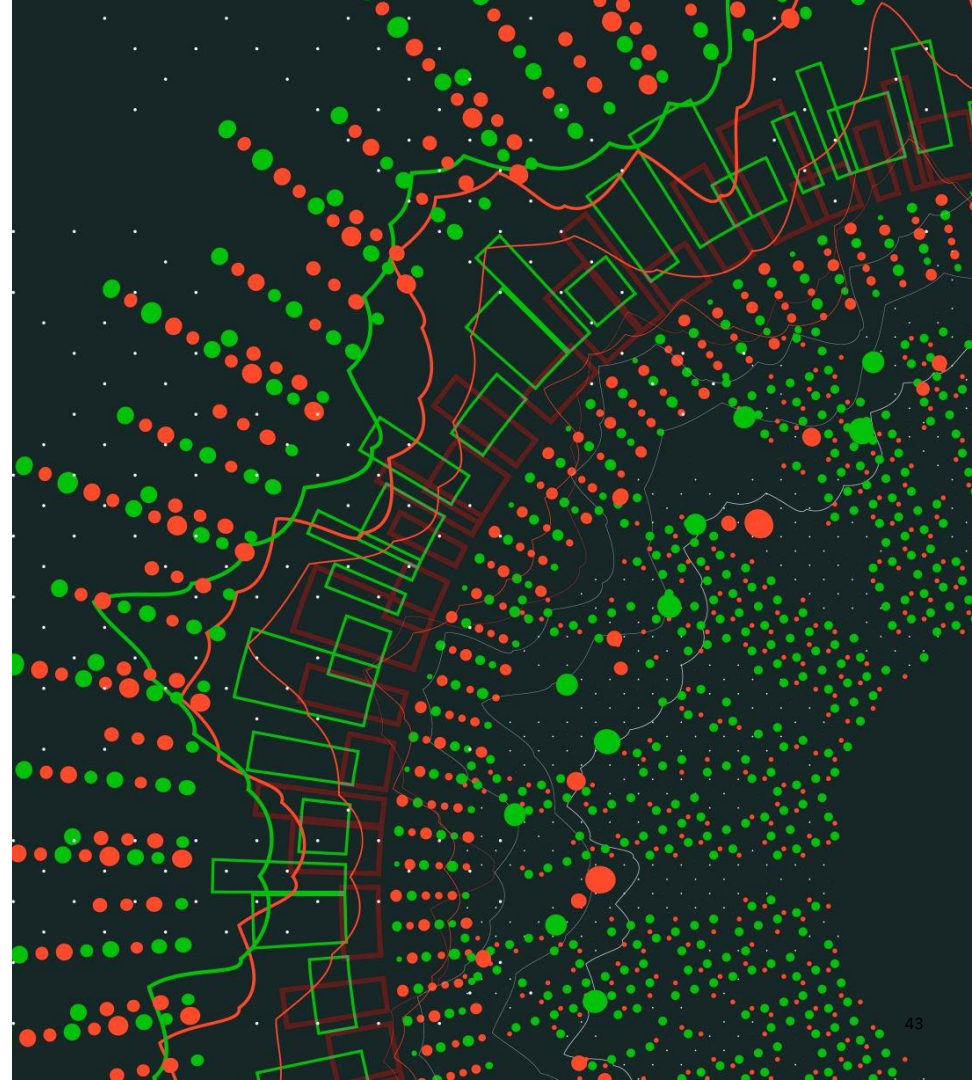
initiate with “mpirun ./helper.sh my_app ...”

```
# Configure MPS
# Process with LOCAL_RANK 0 (ie 1 per node) will start MPS
export CUDA_MPS_PIPE_DIRECTORY=/dev/shm/${USER}/mps
export CUDA_MPS_LOG_DIRECTORY=/dev/shm/${USER}/mps_log
unset CUDA_VISIBLE_DEVICES # use all GPU
if [ $MY_LOCAL_RANK = 0 ]; then
    if [ $MY_RANK = 0 ]; then
        echo starting mps ...
    fi
    rm -rf ${CUDA_MPS_PIPE_DIRECTORY} ${CUDA_MPS_LOG_DIRECTORY}
    mkdir -p ${CUDA_MPS_PIPE_DIRECTORY} ${CUDA_MPS_LOG_DIRECTORY}
    MPSCPUS=$(echo ${mpscpu[@]} | sed 's: :, :g')
    CMD="taskset -c $MPSCPUS /usr/bin/nvidia-cuda-mps-control -d"
    # echo $CMD
    eval $CMD
    sleep 1
else
    sleep 4 # don't start process before MPS is active
fi
```

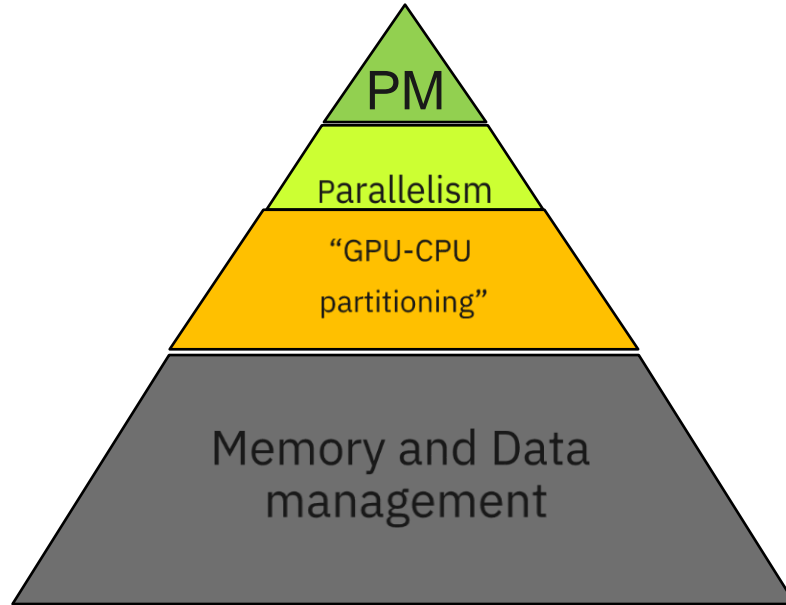
```
# Start application
$@

# Stop MPS
if [ $MY_LOCAL_RANK = 0 ]; then
    if [ $MY_RANK = 0 ]; then
        echo stopping mps ...
    fi
    sleep 1
    export CUDA_MPS_PIPE_DIRECTORY=/dev/shm/${USER}/mps
    export CUDA_MPS_LOG_DIRECTORY=/dev/shm/${USER}/mps_log
    echo "quit" | /usr/bin/nvidia-cuda-mps-control
    sleep 1
    rm -rf ${CUDA_MPS_PIPE_DIRECTORY} ${CUDA_MPS_LOG_DIRECTORY}
fi
```

N8 CIR Bede Introduction to GPU programming



PROGRAMMING PRINCIPLES



CUDA Execution Model

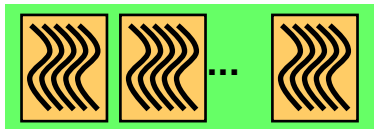
Software



Thread



Thread Block

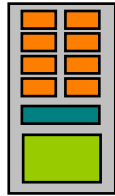


Grid

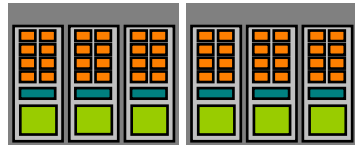
Hardware



Scalar Processor



Multiprocessor



Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

blocks and grids can be multi dimensional (x,y,z)

Parallelism on a GPU – CUDA Blocks



el”

the GPU is called a

Grid



A block can identify itself by reading **blockIdx.x**

CUDA Threads

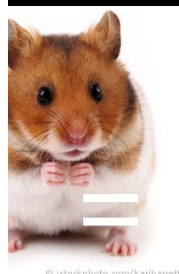
Block

s”

- Each



threadIdx



threadIdx.x = 2

= THREAD



threadIdx.x = M - 1



© istockphoto.com/karlbarrett

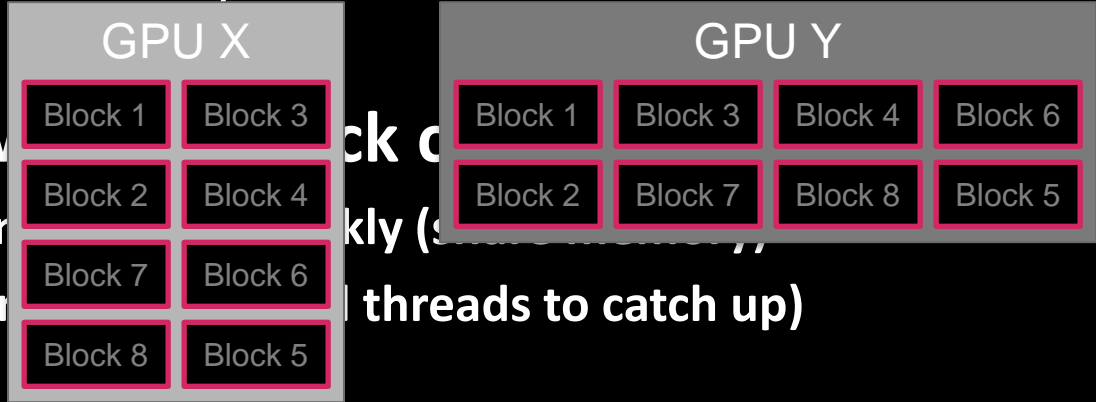
- A thr

threadIdx.x

- The t

can be read with blockDim.x

Why threads and blocks?



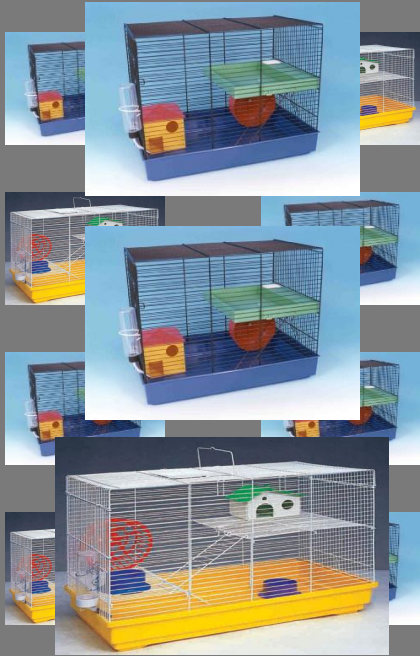
- **Threads w**
 - **Time**
 - **Commun**
 - **Synchron**
- Block c
kly (s
threads to catch up)

Why break up into blocks?

- A block cannot be broken up among multiple SMs (streaming multiprocessors), and you want to keep all SMs busy.
- Allows the HW to scale the number of blocks running in parallel based on GPU capability

Why threads and blocks?

Symmetric Multiprocessor



SM can hold up to 32 **blocks** (cages)

Blocks (cages) can be from different kernels

Depending on size and number of

threads (hamsters) in a **block** (cage),

= **Section of the store**
fewer **blocks** (cages) may fit into

SM

Why threads and blocks?

Symmetric Multiprocessor

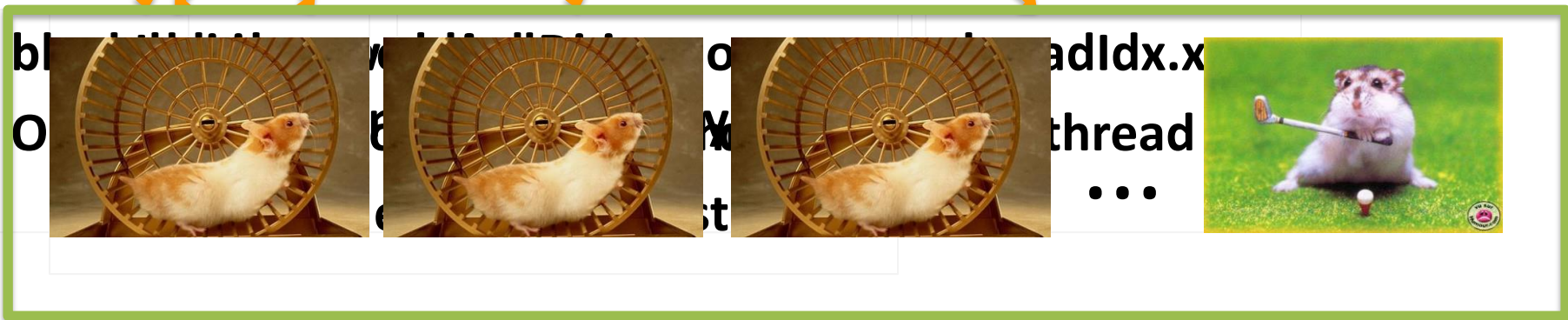


- In addition, each SM has a certain number of **units** (hamster wheels) for the **threads** (hamsters) to do work on
- Once all the **threads** (hamsters) in a **block** (cage) have finished their work, a new **block** (cage) is swapped in.
- Finally, a **GPU** (pet store) can be made up of different numbers of **SMs**

SAXPY kernel

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

Block



SAXPY kernel – with data

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

10 threads (hamsters)
each with a different i

- For $\text{blockIdx.x} = 0$

- **Let's work with 30 data elements**

- $i = 0 * 10 + \text{threadIdx.x} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- **Broken into 3 blocks, with 10 threads per block**

- So, $\text{blockDim.x} = 10$

- For $\text{blockIdx.x} = 1$

- $i = 1 * 10 + \text{threadIdx.x} = \{10, 11, 12, 13, 14, 15, 16, 17, 18, 19\}$

- For $\text{blockIdx.x} = 2$

- $i = 2 * 10 + \text{threadIdx.x} = \{20, 21, 22, 23, 24, 25, 26, 27, 28, 29\}$

Programming GPUs

Applications

Libraries

“Drop-in”
Acceleration

Compiler
Directives

Easy to use
Portable code

Programming
Languages

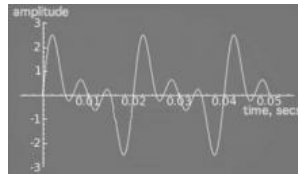
Maximum Performance
and Flexibility

NVIDIA Developer Libraries

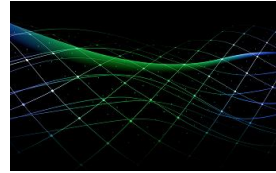
cuBLAS
cuBLAS-XT
NVBLAS



cuFFT
cuFFT-XT



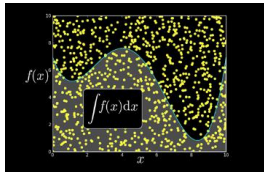
cuSPARSE
cuSOLVER
AMGX



cuDNN



cuRAND



THRUST



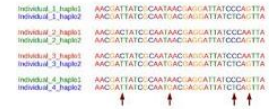
NPP



NVENC



NVBIO



OPENACC DIRECTIVES

Manage
Data
Movement

```
#pragma acc data copyin(a,b) copyout(c)  
{  
  ...  
  #pragma acc kernels  
  {  
    #pragma acc loop gang vector  
    for (i = 0; i < n; ++i) {  
      z[i] = x[i] + y[i];  
      ...  
    }  
  }  
  ...  
}
```

Initiate
Parallel
Execution

Optimize
Loop
Mappings

OpenACC
Directives for Accelerators

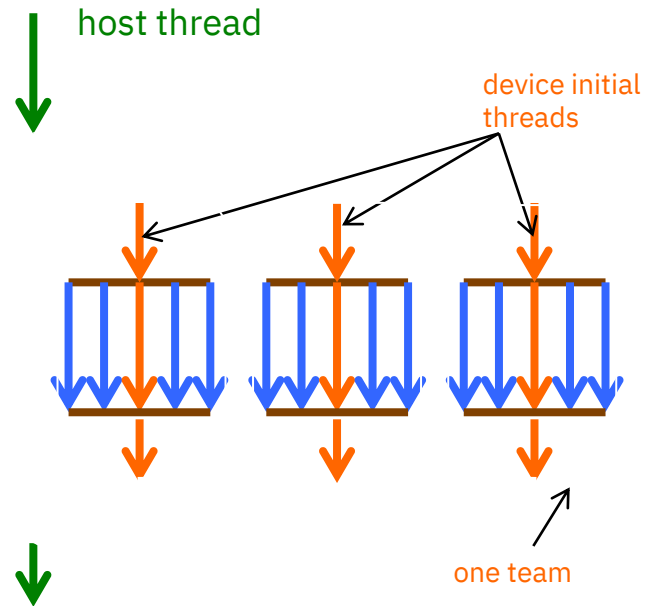
- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU

OpenMP directives

Combined target teams construct

```
#pragma omp target teams  
map(to: a, b) map(from: c)  
{  
  int n = 64;  
  #pragma omp distribute  
  for(int i=0; i<n; i++) {  
    #pragma omp parallel for  
    for(int j=0; j<n; j++)  
      for(k=0; k<n; k++)  
        c[i, j] = a[l, k] * b[k, j];  
  }  
}
```

- target transfer control of execution to one device thread per team
- every team initially execute the same code
- in a “#pragma omp distribute”, each team get it's subset of iteration space



CUDA

Maximum flexibility

CPU code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096 * 256, 2.0, x, y);
```

GPU code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
    int i = blockIdx.x * blockDim.x +
            threadIdx.x;
    if (i < n) y[i] = a * x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096, 256>>>(n, 2.0, x, y);
```

Memory management

EXPLICIT CONTROL

Custom Data Movement

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code

```
void *data, *d_data;  
data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(data, N);  
cudaMemcpy(d_data, data, N, ...)  
gpu_func2<<<...>>>(d_data, N);  
  
cudaMemcpy(data, d_data, N, ...)  
cpu_func3(data, N);  
  
free(data);  
cudaFree(d_data);
```

OpenACC Data directive + CUDA

Custom Data Movement

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code

```
void *data;  
data = malloc(N);  
#pragma acc enter data create(data)  
cpu_func1(data, N);  
#pragma acc update device(data)  
#pragma acc host_data use_device(data)  
gpu_func2<<<...>>>(data, N);  
  
#pragma acc update host(data)  
  
cpu_func3(data, N);  
  
#pragma acc exec data delete(data)  
free(data);
```

UNIFIED MEMORY

Single Pointer, Automatic Migration

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

UNIFIED MEMORY

Deep Copy Nightmare

Explicit Memory Management

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
    data[i] = (char*)malloc(N);

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
    cudaMalloc(&h_data2[i], N);
    cudaMemcpy(h_data2[i], h_data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data2, N*sizeof(char*), ...);

gpu_func<<<...>>(data, N);
```

Unified Memory

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
    data[i] = (char*)malloc(N);

gpu_func<<<...>>(data, N);
```

UNIFIED MEMORY FUNDAMENTALS

When Is This Helpful?

Quick and dirty algorithm prototyping, focus on the *compute part* first

Iterative process with lots of *data reuse*, migration cost can be amortized

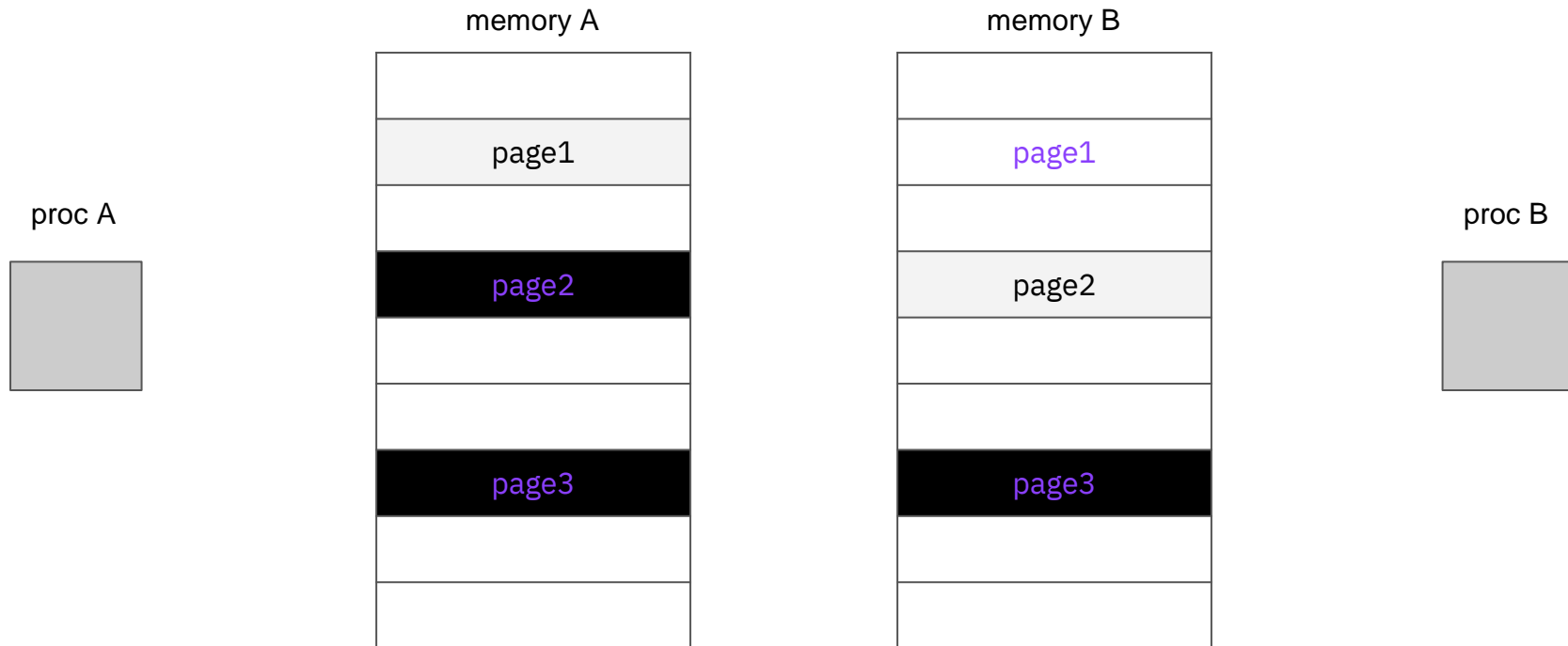
Simplify application debugging, increase your *productivity*

Irregular or *dynamic* data structures, unpredictable access

Large datasets on single GPU, data partitioning between multiple GPUs

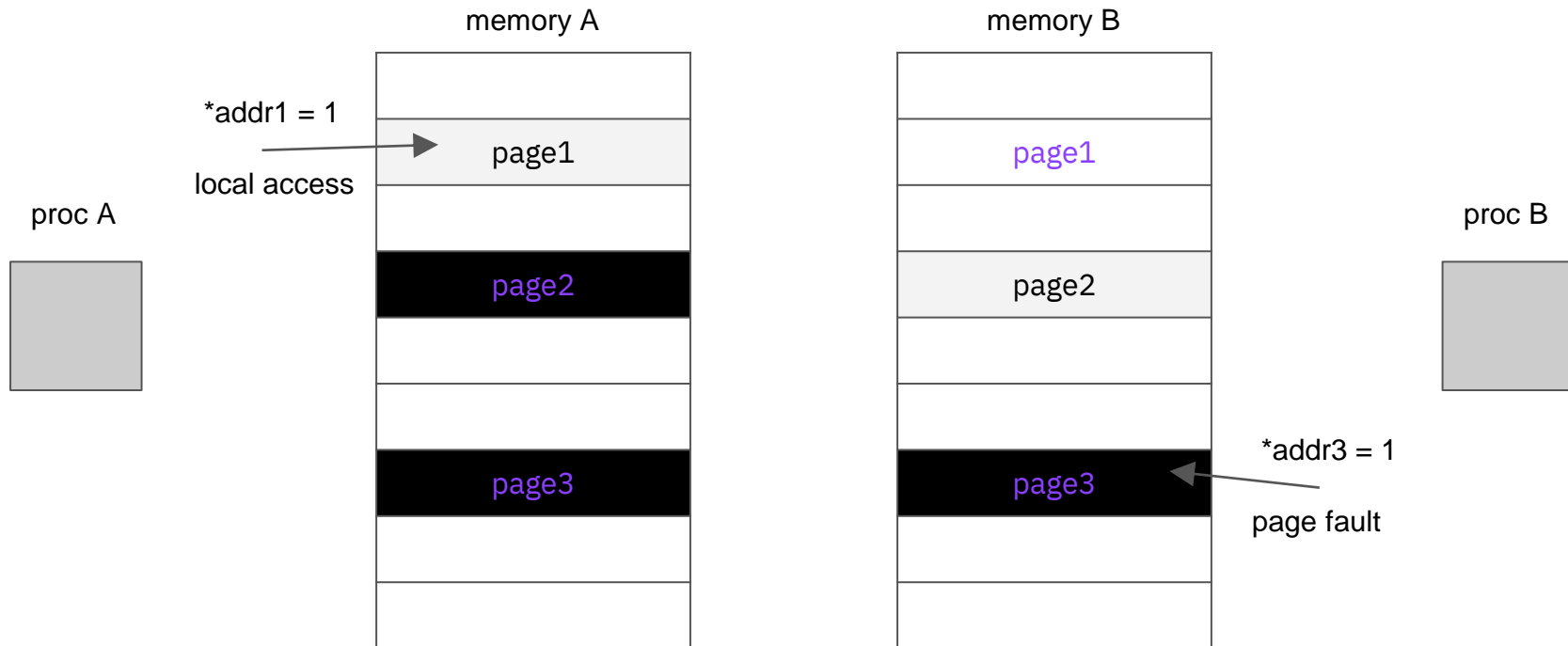
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



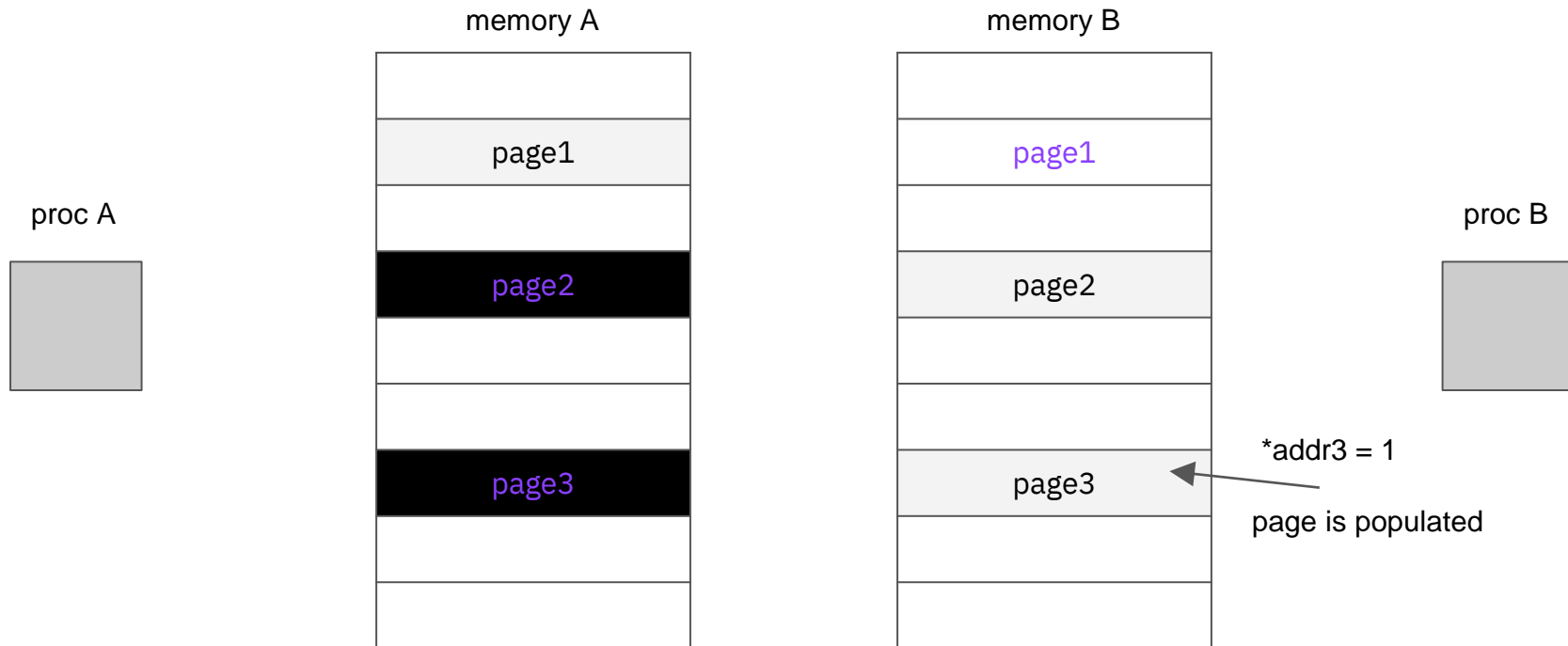
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



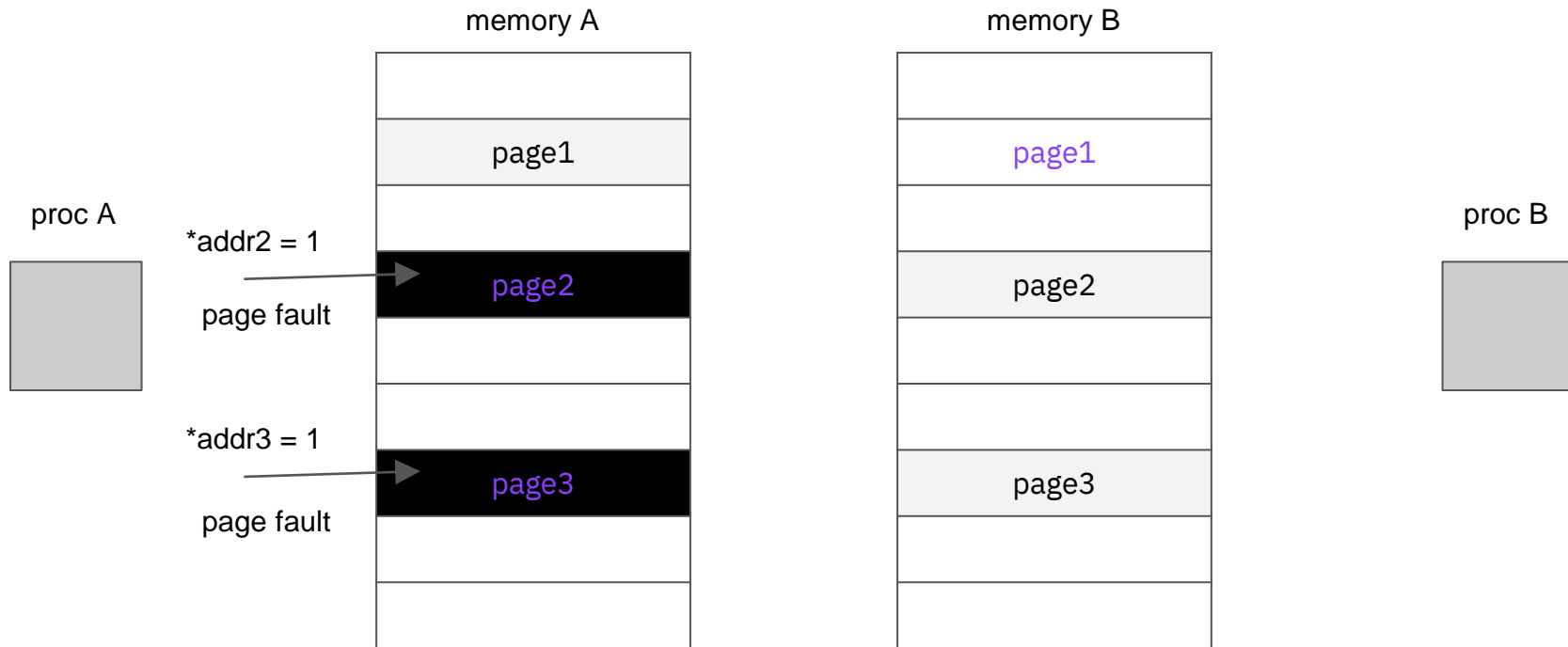
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



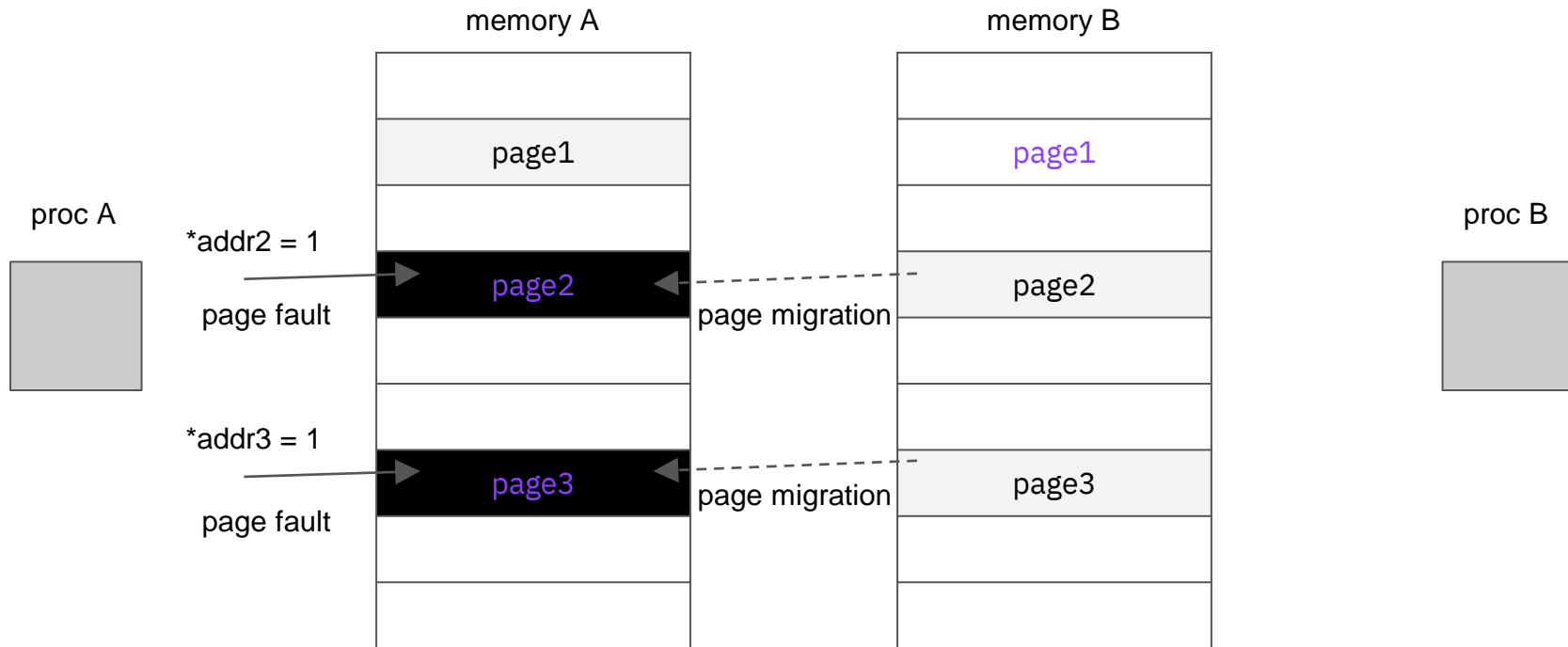
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



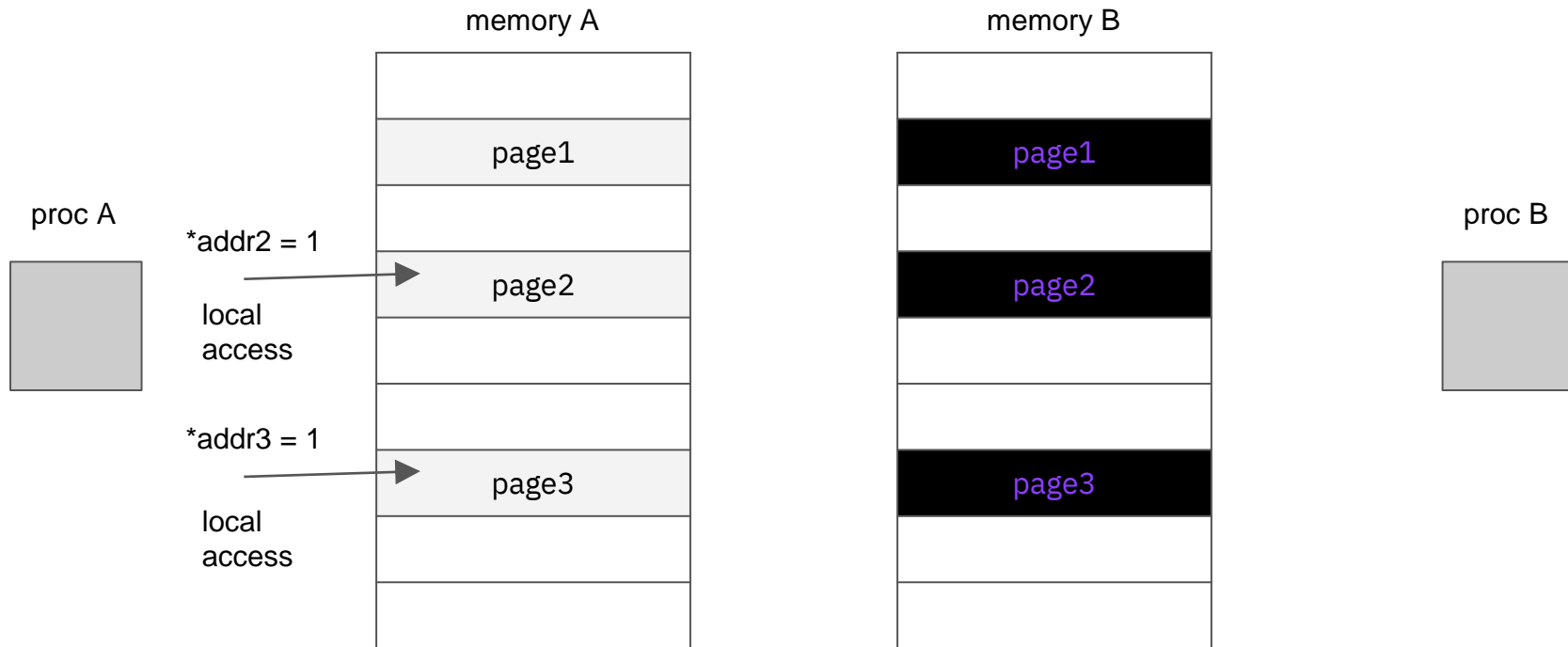
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



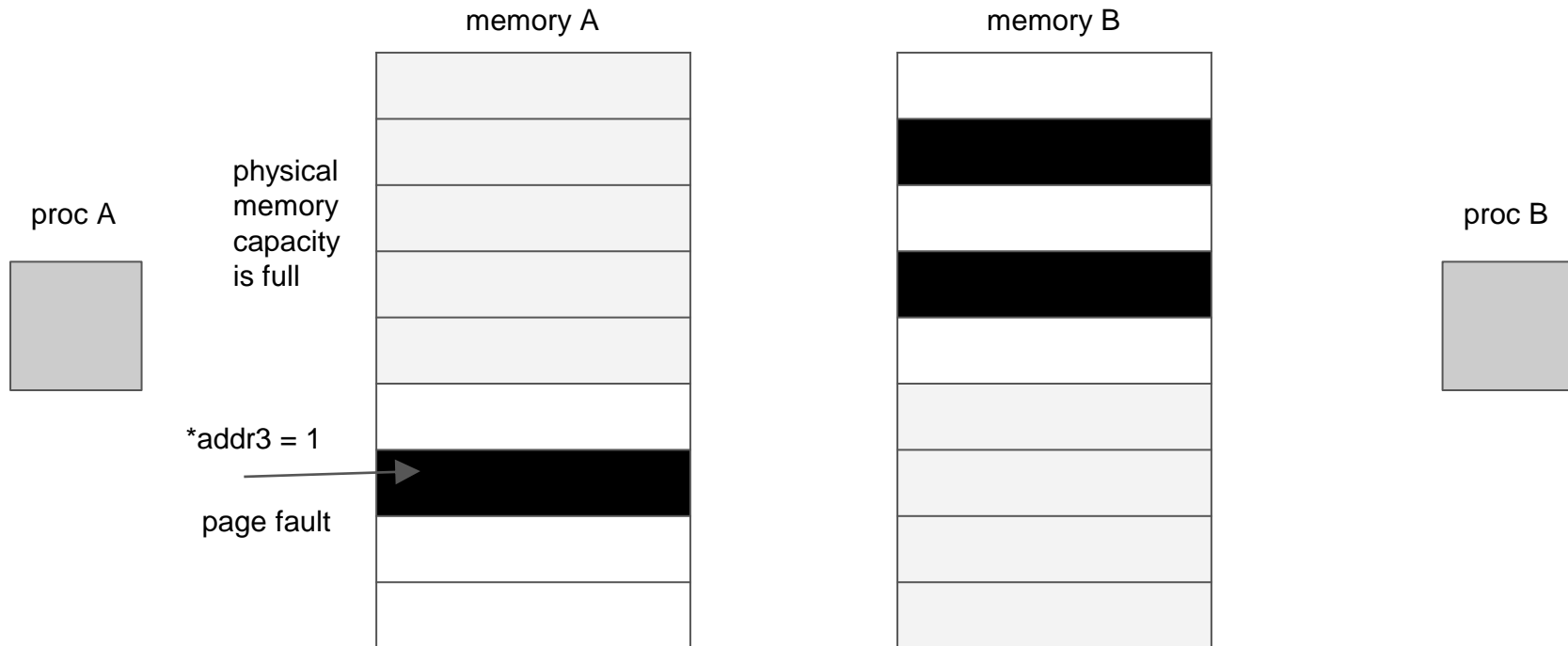
UNIFIED MEMORY FUNDAMENTALS

On-Demand Migration



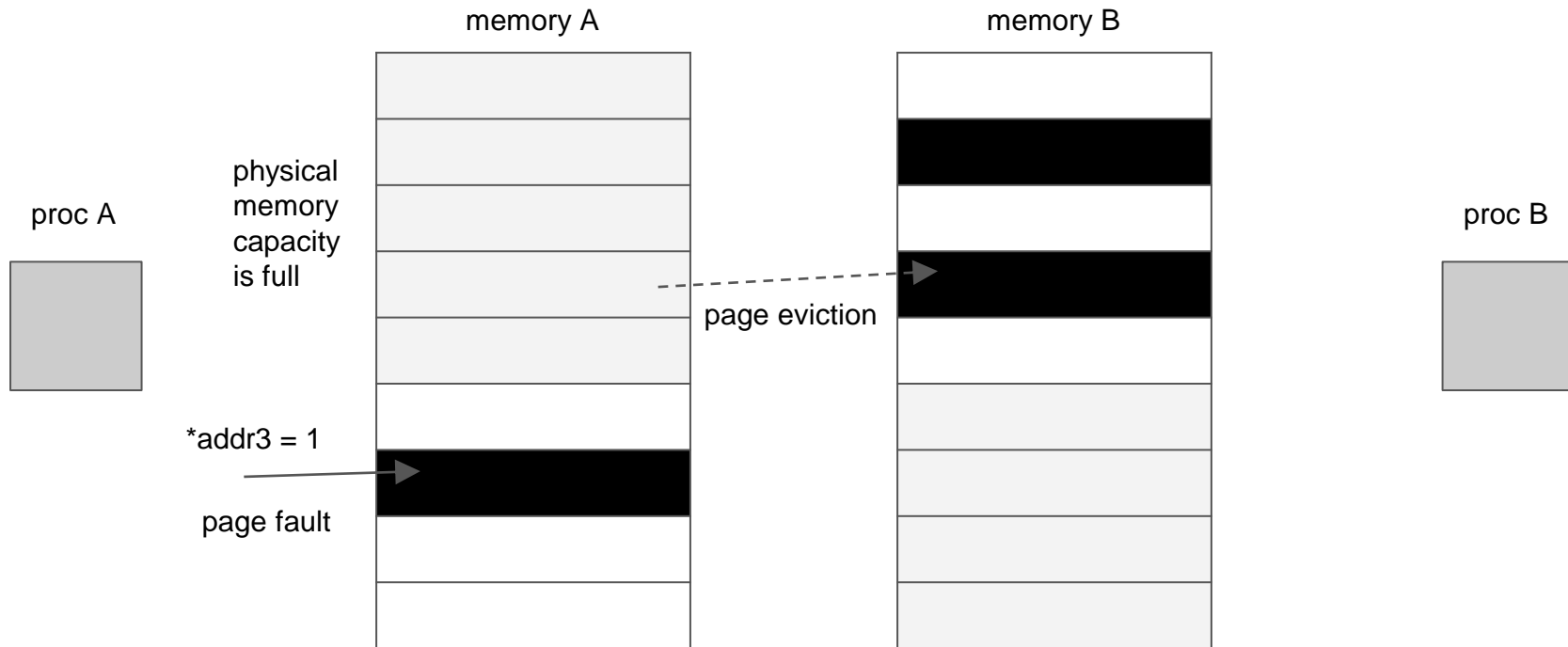
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



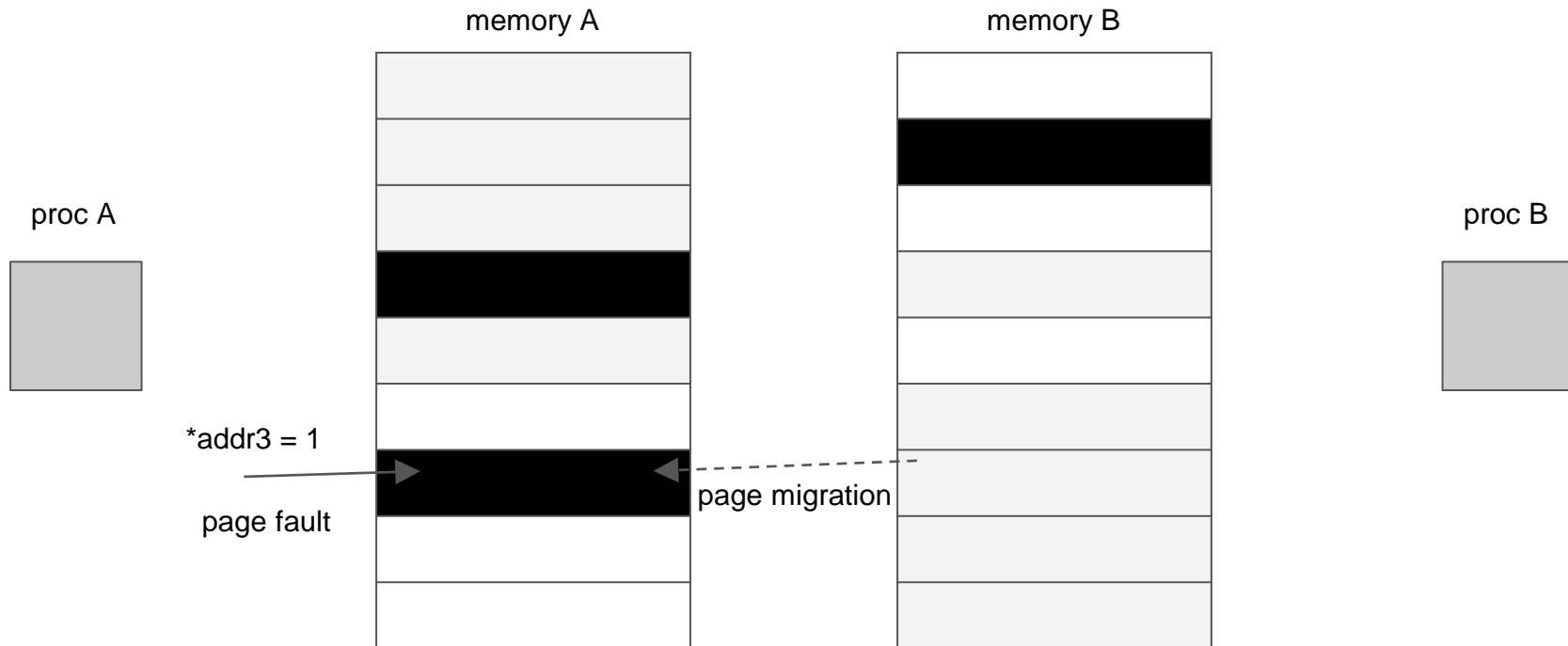
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



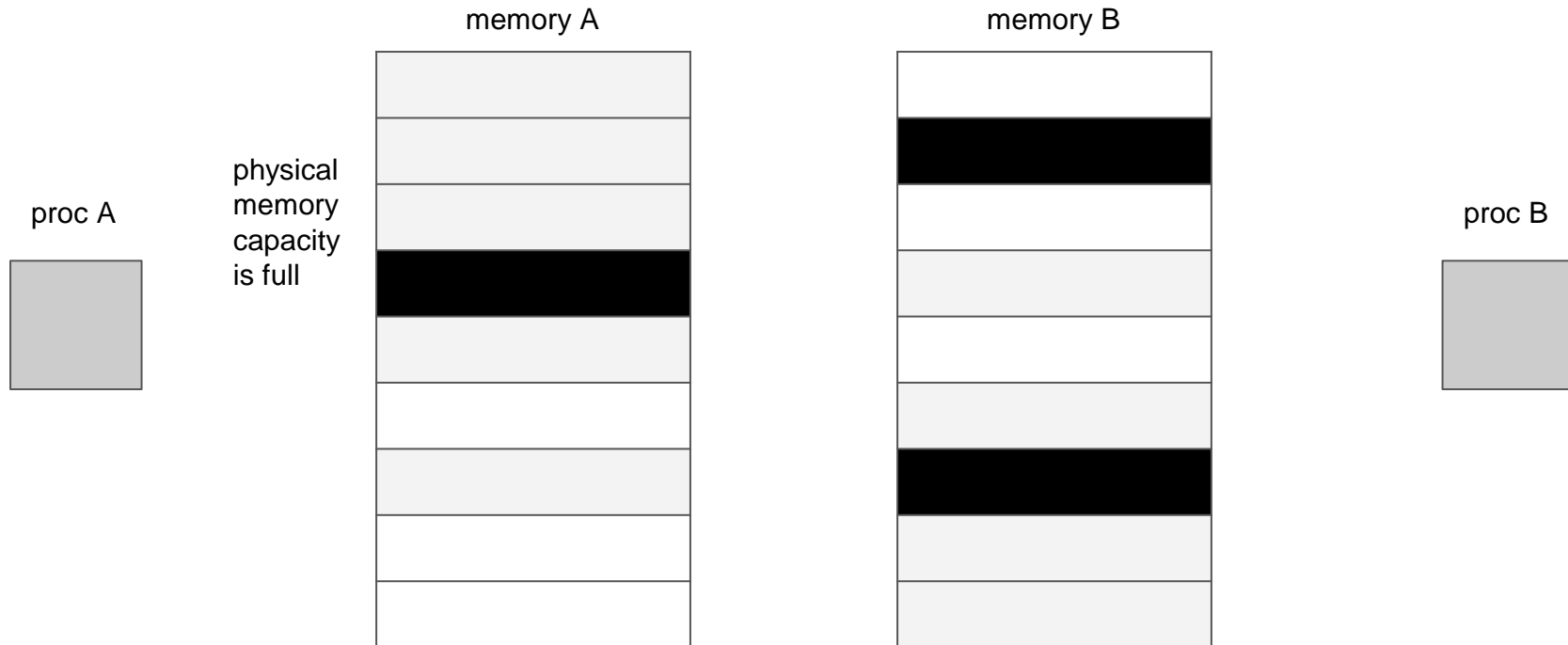
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



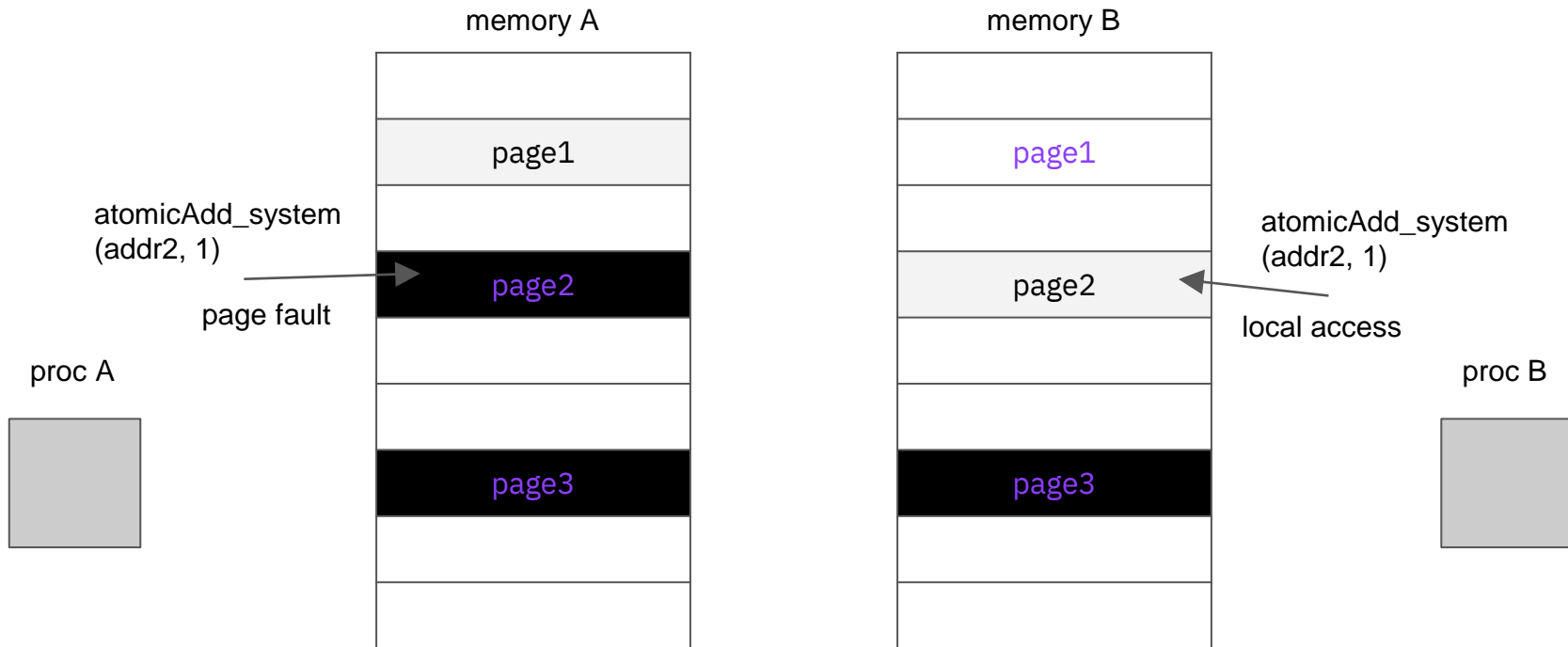
UNIFIED MEMORY FUNDAMENTALS

Memory Oversubscription



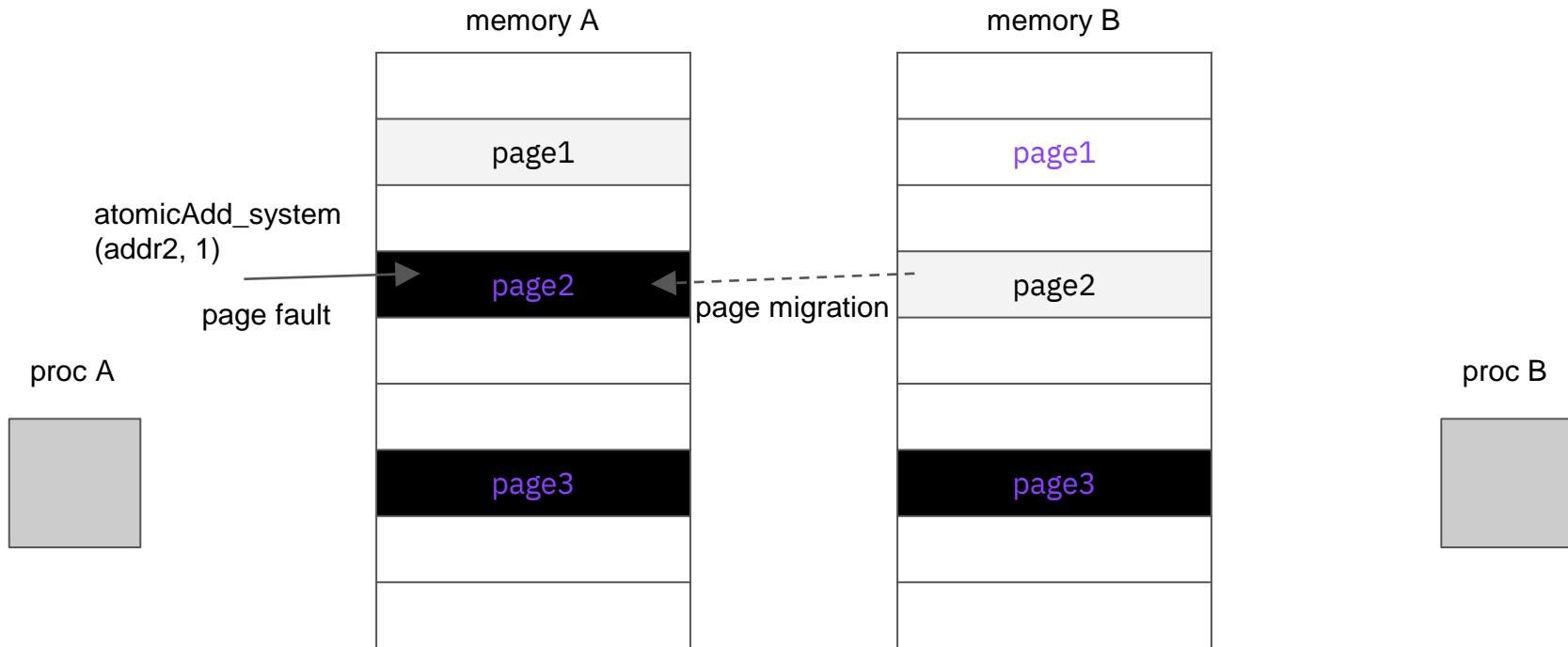
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



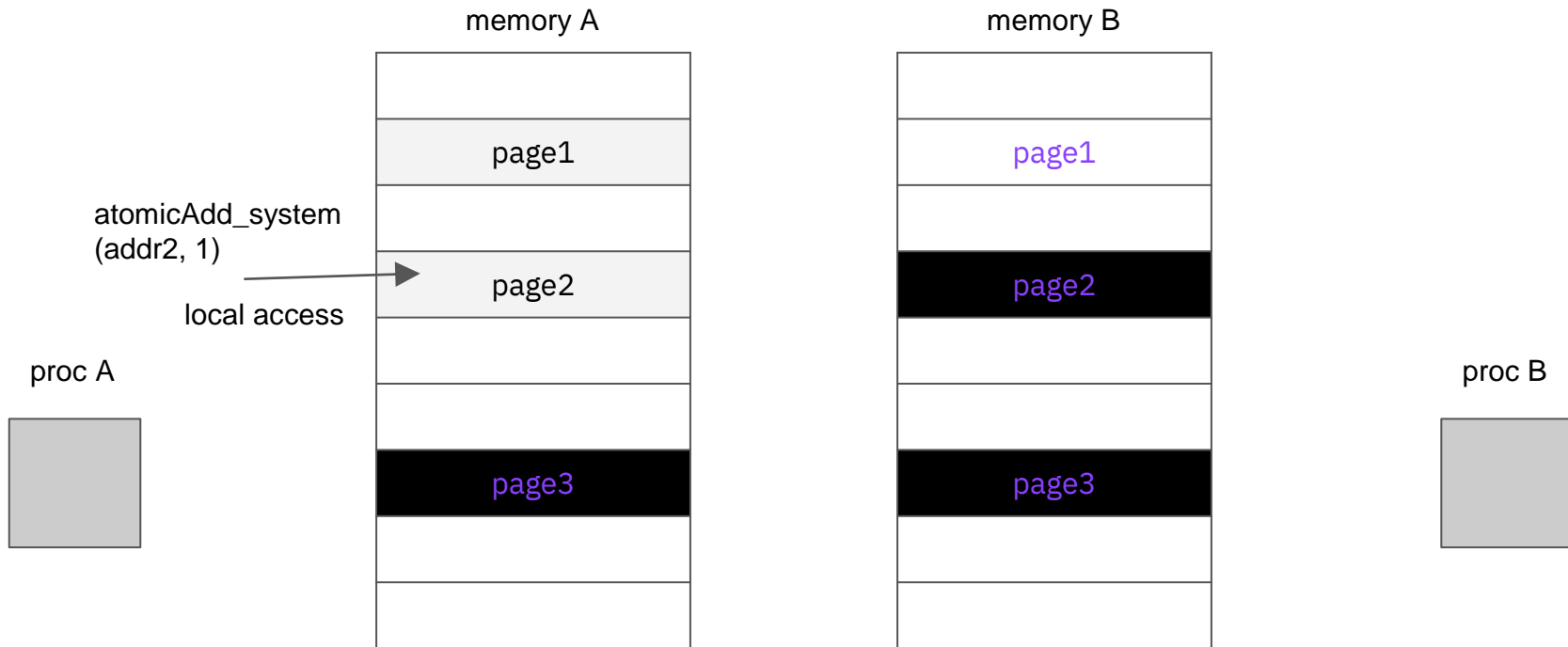
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



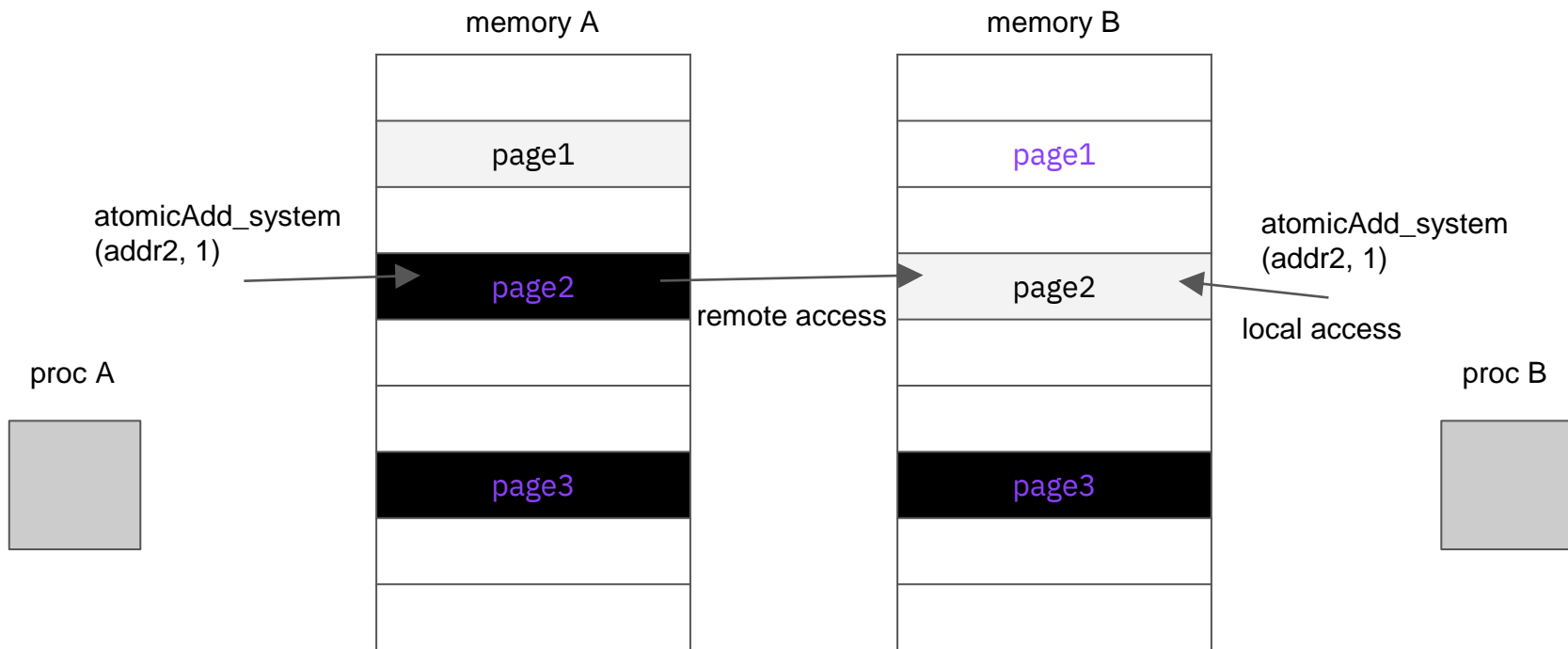
UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics with Exclusive Access



UNIFIED MEMORY FUNDAMENTALS

System-Wide Atomics over NVLINK*



*both processors need to support atomic operations

UNIFIED MEMORY ALLOCATOR

Available Options

CUDA C: **cudaMallocManaged** is your most reliable way to opt in today

CUDA Fortran: **managed** attribute (per allocation)

OpenACC: **-ta=managed** compiler option (all dynamic allocations)

Coming with OpenMP 5.0

malloc support is coming from Pascal+ architectures (Linux only)

Note: you can write your own malloc hook to use **cudaMallocManaged**

PERFORMANCE

Page Granularity Overhead

`cudaMallocManaged` *alignment*: 512B on Pascal/Volta, 4KB on Kepler/Maxwell

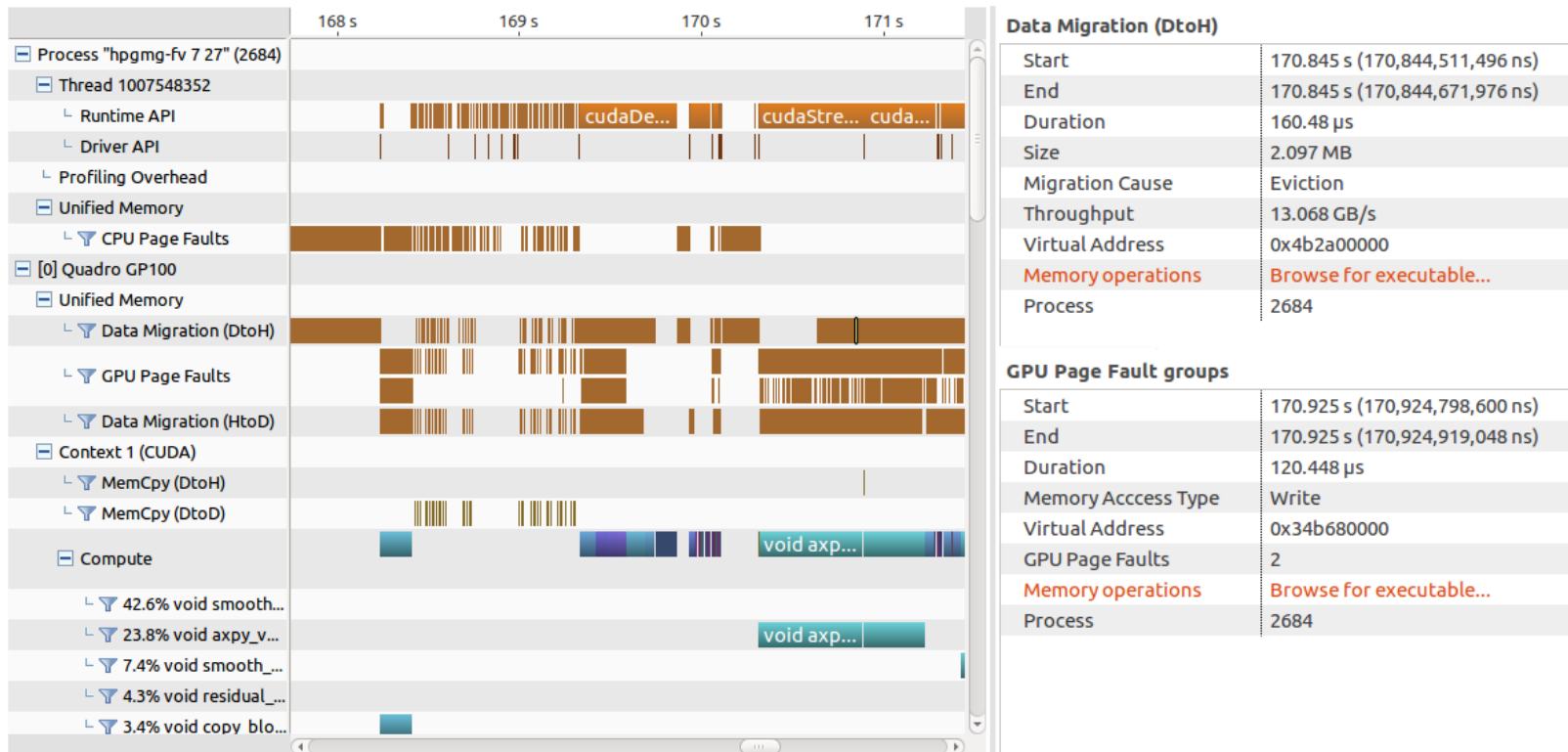
Too many small allocations will use up many pages

`cudaMallocManaged` memory is moved at *system page* granularity

For small allocations more data could be moved than necessary

Solution: use cached allocator or memory pools

PROFILER: INSPECT



PROFILER: FILTER

The screenshot displays a profiler interface with a timeline view at the top and a configuration panel below. The timeline shows various events for a process and GPU, including CPU Page Faults, Data Migration (DtoH), GPU Page Faults, and Data Migration (HtoD). The configuration panel, titled "Results", includes a "Filter Timelines" section with the following settings:

- Filter Timelines
- Start Address: End Address:
- Virtual address range size:
- CPU Page Faults
 - Access Type: Read Write
- GPU Page Faults
 - Access Type: Read Write Atomic Prefetch
- HtoD Migrations
 - Reason: User Coherence Prefetch
- DtoH Migrations
 - Reason: User Coherence Prefetch Eviction

On the left side of the configuration panel, there is an "Application" section with several analysis stages, each with a status icon (a green checkmark or a red X):

- Data Move...Concurrency
- Compute Utilization
- Kernel Performance
- Dependency Analysis
- NVLink
- Unified Memory

PROFILER: CORRELATE

The screenshot displays a performance profiler interface with three main panels:

- Process Hierarchy:** Shows the process "uvmmigration" (8021) and its threads. The thread "Thread 2816767808" is expanded to show "Data Migration (HtoD)" occurring between 0.463s and 0.465s.
- Properties Panel:** Details the "Data Migration (HtoD)" event:

Start	465.073 ms (465,072,848 ns)
End	465.15 ms (465,149,520 ns)
Duration	76.672 μs
Size	933.888 kB
Migration Cause	Prefetch
Throughput	12.18 GB/s
Virtual Address	0x50071c000
Memory operations	
Allocation	_Z13test_ondemandm@uvmmigration.cu:24
Free	_Z13test_ondemandm@uvmmigration.cu:40
Process	8021
- Code Editor:** Shows the source code for "uvmmigration.cu". The function "test_ondemand" is visible, with a highlighted section: `write<<<grid, block>>>(a, n); // on-demand migration to the gpu`.

USER HINTS

Why, When, and How to Use Them

If you know your application well you can optimize with hints

These are also useful to override some of the driver heuristics

`cudaMemPrefetchAsync(ptr, size, processor, stream)`

Similar to `move_pages()` in Linux

`cudaMemAdvise(ptr, size, advice, processor)`

Similar to `madvise()` in Linux

UNIFIED MEMORY OUTLOOK

Consider using Unified Memory for any new application development

Get your code *running* on the GPU much sooner!

Enjoy clean code and **virtually** no memory limits

Increase productivity, explore and prototype new algorithms

Use the explicit data management only *where you need it*

Resources

Learn more about GPUs

CUDA resource center: <http://docs.nvidia.com/cuda>

GTC on-demand: <http://on-demand-gtc.gputechconf.com>

Parallel Forall blog: <http://devblogs.nvidia.com/parallelforall>

Self-paced labs: <http://nvidia.qwiklab.com>

N8 CIR Bede C/C++/Fortran Application porting and tuning overview to IBM AC922 Power/GPU

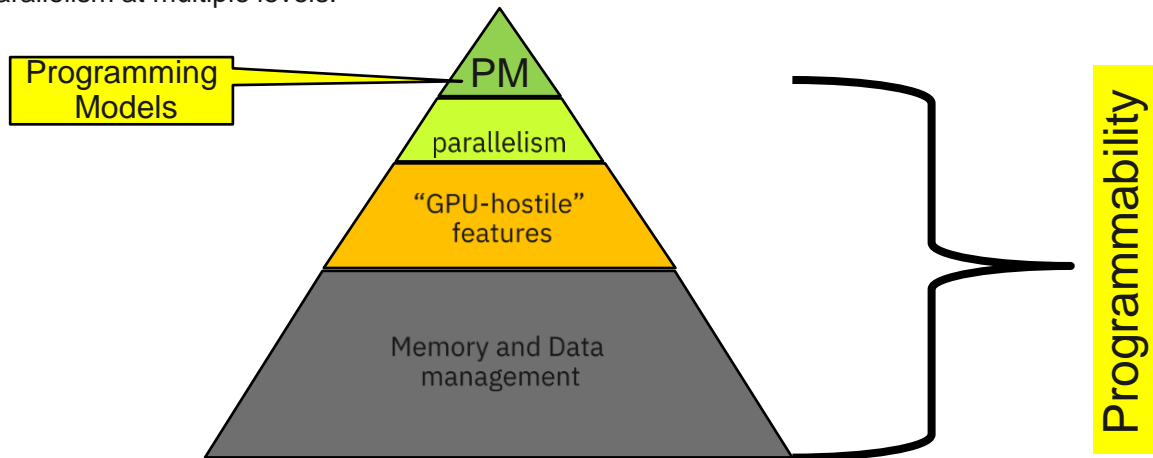


Technical Challenges: Programming Hybrid System

Observation 1: over about multiple years of readying applications, the level “comfort” of developers is improving.

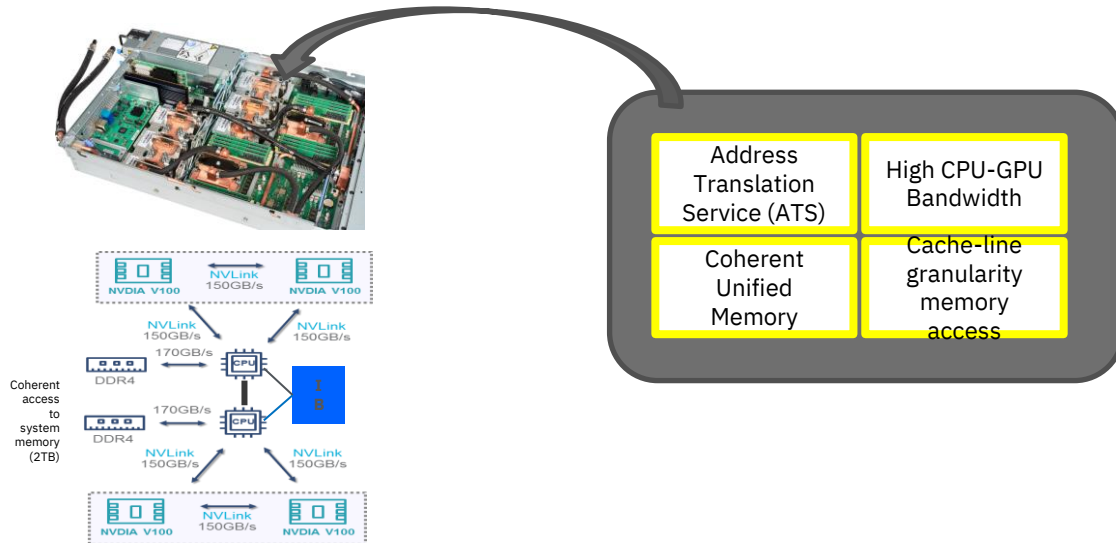
Observation 2: some times developers make different choices facing the same dilemmas....

Observation 3: with a non-unified memory in majority of cases developers spend more time in learning how to manage memory/data, what changes are needed and how to express parallelism at multiple levels.



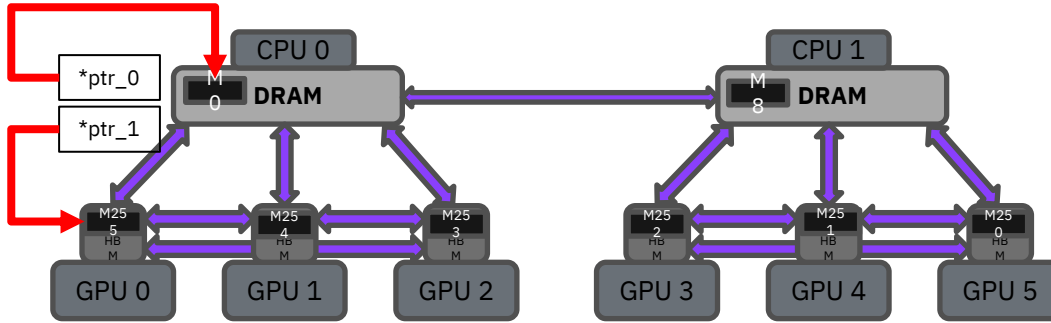
NVLink connecting POWER and NVIDIA GPUs is one of the major differentiators of the AC922 server with respect to the competition

NVLink is more than just high Bandwidth (BW), it is ahead of the competition as far as **memory/data management** is concerned; while many of the unique features work well, **most of the application developers/customers are not aware of** their **potential** use
it comes as a package:



AC922: Unified Virtual Memory Addressing and ATS.
Memory and data management: opportunities for application developers

*Unified Virtual Address Space and ATS enable applications developers to **reduce the complexity** of the code and the complexity of the memory/data management and to **improve performance** by **reducing** the overall **latency**.*



*Unified Virtual Address Space and ATS enable threads running on CPU cores and on GPU SMs to **access ANY** of the **AC922 memory pools** (NUMA domains)*

Main technical challenges and tuning direction

Maximize Data, GPU/CPU occupancy and locality

- Memory can become the sizing parameter for parallel executions
- **New “Data” parallelism and communication schemes** (including CPU/GPU topology and GPU direct features...)
- **New innovative memory and Data management** (memory allocators and transfers, GPU/CPU compression, Power nx-gzip...)

Code Optimizations

- **Parallelism extraction**
 - Maximize number of GPU threads
 - Hybrid implementation: MPI, OpenMP (CPU), threads (CPU & GPU), OpenACC/CudaC/CudaFortran (GPU)
- **Concurrent and Asynchronous GPU/CPU/Com/IO tasks => overlapping** – P9 NVlink advantage and MPI implementation
 - Computation/Communications overlapping
 - Computation/Computation GPU and CPU functions overlapping
 - GPU kernels overlapping (**cuda streams**), GPU kernels parallel execution
- **Communications optimization** (overlapping, asynchronism, GPU Direct)
- **GPU kernels optimization** (OpenACC, Cuda C/Fortran, shared memory, number of GPU registers...)
- **CPU function optimization**

Application optimization strategy

Objectives:

- **Production & TCO focus**
- **Performance**
- **Other?**

Programmability and portability

- **Only one source to maintain?**
- **No POWER/GPU dependency – standard programming model based on OpenACC & OpenMP**
- Specific tuning to highlight Power/GPU specific architecture
 - IO Interface ~7TB/s per chip
NVLINK, PCIe Gen4, 25GB/s technology, CAPI2.0/OpenCAPI, NX Gzip accelerator
 - Memory bandwidth: >270 Go/s peak
 - SMT 4 with >144 logical core per node

Specific technical focus

- Parallelism extraction – data parallelism
- Data and memory management
- Concurrency and Asynchronism for resource optimization
- Communication schemes
- GPU and CPU porting & tuning
- Runtime optimization

Memory management

CPU-GPU data allocation and transfers

- zero copy, duplication, memory allocators, async transfers, GPU Direct, topology, pinned memory...

Memory Allocators and management

- OpenACC/OpenMP Data Directives (GPU data allocation, update and transfer)
- CUDA allocator
- CUDA Fortran Managed or/and CUDA Managed Memory
- AC922 UVMA (map, malloc, posix-memalign, shem) including share segment

Double buffering
Shared memory
GPU direct disk access
GPU compression
Power nx-gzip

Memory locking and process binding for multi-instances executions

GPU specific tuning

New functions on GPU using compiler directives

Development of CUDA versions for intensive kernels

- Small perf difference for most of the kernels using Compiler directives vs CUDA on NVidia V100.

Asynchronism and concurrent operations

- GPU vs GPU kernels with CUDA Streams
- GPU kernels vs data transfers
- GPU kernels vs MPI communications
- GPU kernels vs CPU kernels
- CPU kernels vs CPU kernels

Kernel overlapping with CUDA Streams and priorities

Example to create CUDA Streams with CUDA and OpenACC

1. Declare stream variables

```
integer, parameter :: STREAM_SUPERINNER = 10
integer, parameter :: STREAM_BORDER = 20
integer, parameter :: STREAM_DAMPING = 20
integer, parameter :: STREAM_COMM_FILLRCV = 20
integer, parameter :: STREAM_DAMPING1 = 31
integer, parameter :: STREAM_DAMPING2 = 32
integer, parameter :: STREAM_DAMPING3 = 33
integer, parameter :: STREAM_DAMPING4 = 34
integer, parameter :: STREAM_DAMPING5 = 35
integer, parameter :: STREAM_DAMPING6 = 36
integer, parameter :: STREAM_BORDER_N = 31
integer, parameter :: STREAM_BORDER_S = 32
integer, parameter :: STREAM_BORDER_E = 33
integer, parameter :: STREAM_BORDER_W = 34
integer, parameter :: STREAM_BORDER_F = 35
integer, parameter :: STREAM_BORDER_B = 36
integer(kind=cuda_stream_kind) :: STREAM_SUPERINNER_CUDA, STREAM_BORDER_CUDA, &
    STREAM_DAMPING_CUDA, STREAM_COMM_FILLRCV_CUDA
integer(kind=cuda_stream_kind) :: STREAM_BORDER_N_CUDA, STREAM_BORDER_S_CUDA, STREAM_BORDER_E_CUDA, &
    STREAM_BORDER_W_CUDA, STREAM_BORDER_F_CUDA, STREAM_BORDER_B_CUDA
integer(kind=cuda_stream_kind) :: STREAM_DAMPING1_CUDA, STREAM_DAMPING2_CUDA, STREAM_DAMPING3_CUDA, &
    STREAM_DAMPING4_CUDA, STREAM_DAMPING5_CUDA, STREAM_DAMPING6_CUDA
call cudacheck(cudaStreamCreateWithPriority(STREAM_COMM_FILLRCV_CUDA,0,HIGH_PRIORITY_STREAM))
endif
```

3. name streams

```
call nameCudaStream(STREAM_BORDER_CUDA,"STREAM_BORDER")
call nameCudaStream(STREAM_BORDER_N_CUDA,"STREAM_BORDER_N")
call nameCudaStream(STREAM_BORDER_S_CUDA,"STREAM_BORDER_S")
call nameCudaStream(STREAM_BORDER_E_CUDA,"STREAM_BORDER_E")
call nameCudaStream(STREAM_BORDER_W_CUDA,"STREAM_BORDER_W")
call nameCudaStream(STREAM_BORDER_F_CUDA,"STREAM_BORDER_F")
call nameCudaStream(STREAM_BORDER_B_CUDA,"STREAM_BORDER_B")
call nameCudaStream(STREAM_SUPERINNER_CUDA,"STREAM_SUPERINNER")
call nameCudaStream(STREAM_COMM_FILLRCV_CUDA,"STREAM_COMM_FILLRCV")
! Set Openacc streams
call acc_set_cuda_stream(STREAM_SUPERINNER, STREAM_SUPERINNER_CUDA)
call acc_set_cuda_stream(STREAM_COMM_FILLRCV, STREAM_COMM_FILLRCV_CUDA)
call acc_set_cuda_stream(STREAM_BORDER, STREAM_BORDER_CUDA)
call acc_set_cuda_stream(STREAM_DAMPING, STREAM_DAMPING_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_N, STREAM_BORDER_N_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_S, STREAM_BORDER_S_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_E, STREAM_BORDER_E_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_W, STREAM_BORDER_W_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_F, STREAM_BORDER_F_CUDA)
call acc_set_cuda_stream(STREAM_BORDER_B, STREAM_BORDER_B_CUDA)
```

2. Create CUDA stream

```
mPriorityRange(LOW_PRIORITY_STREAM,HIGH_PRIORITY_STREAM))
thPriority(STREAM_SUPERINNER_CUDA,0,LOW_PRIORITY_STREAM))
thPriority(STREAM_BORDER_N_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_S_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_E_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_W_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_F_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_B_CUDA,0,HIGH_PRIORITY_STREAM))
thPriority(STREAM_BORDER_CUDA,0,HIGH_PRIORITY_STREAM))
```

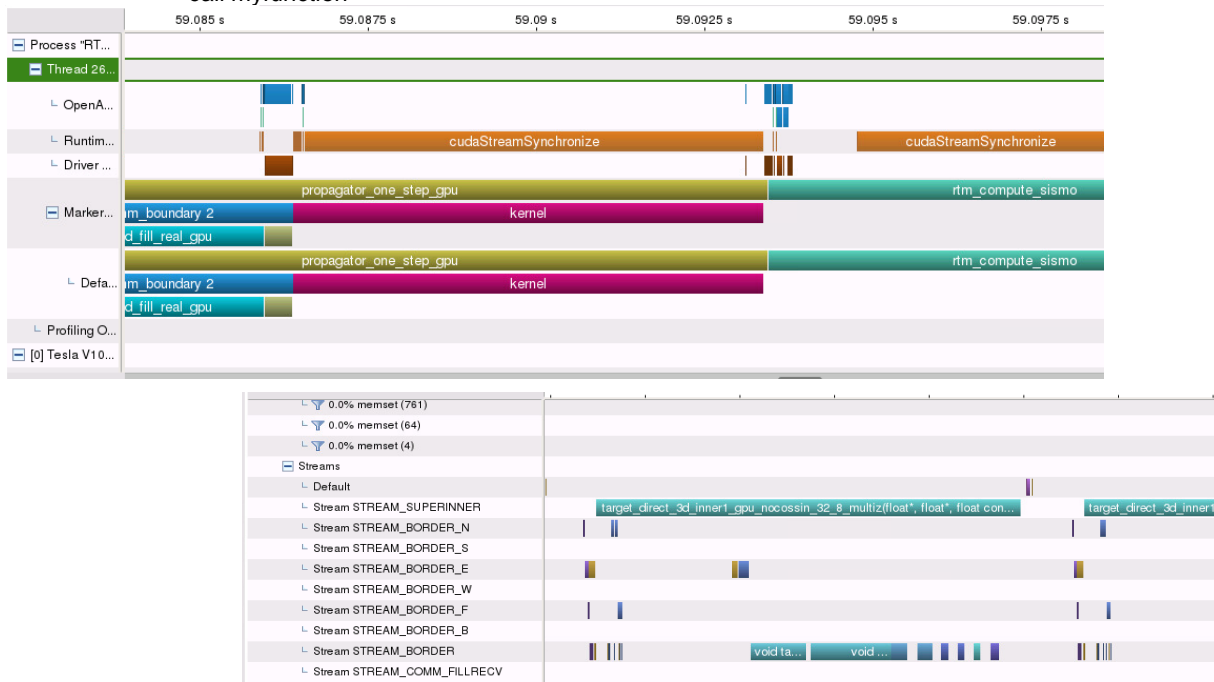
```
DER ) then
ORDER_CUDA
WithPriority(STREAM_DAMPING_CUDA,0,HIGH_PRIORITY_STREAM))
```

```
_FILLRCV_CUDA ) then
REAM_BORDER_CUDA
```

4. Associate Cuda and OpenACC streams

Code profiling - Instrument code with NVIDIA nvtx

- All NVTX API functions start with a nvtx name prefix and may end with one out of the three postfixes A, W, or Ex
- The core NVTX API is defined in file nvToolsExt.h
- Easy to use, no overhead (can be permanent), capability to set up colors:
Call **push_cudatag**("myfunction_name")
call myfunction

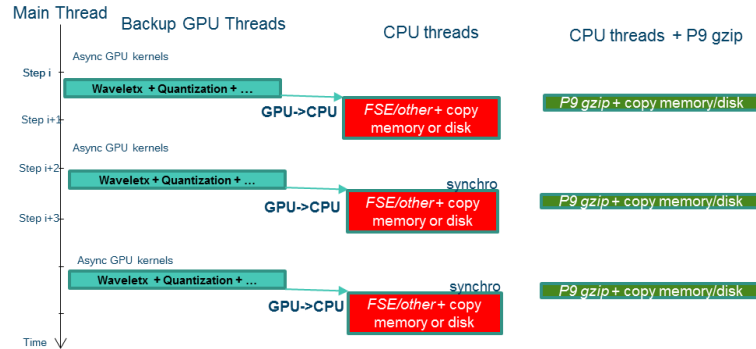


Example: both sync GPU and CPU Compression — for seismic imaging

Data GPU and CPU compression with async read/write with overlapping

P9 NX on-chip gzip accelerator (2 per system)

- Highest throughput on-chip gzip/gunzip engine in the industry
 - 7.7GB/s compress 15.6GB/s decompress peak (perf. number is for a single Deflate stream; not multiple engine aggregates)
 - 80 to 125x speedup over a CPU single thread
- Standard user-mode interfaces
- Option for higher compression ratio with dynamic Deflate block type support



Thank you

Ludovic ENAULT
Power HPC Specialist
IBM Garage for Systems Montpellier

—
ludovic.enault@fr.ibm.com

© Copyright IBM Corporation 2019. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. Any statement of direction represents IBM's current intent, is subject to change or withdrawal, and represent only goals and objectives. IBM, the IBM logo, and ibm.com are trademarks of IBM Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available at [Copyright and trademark information](#).

