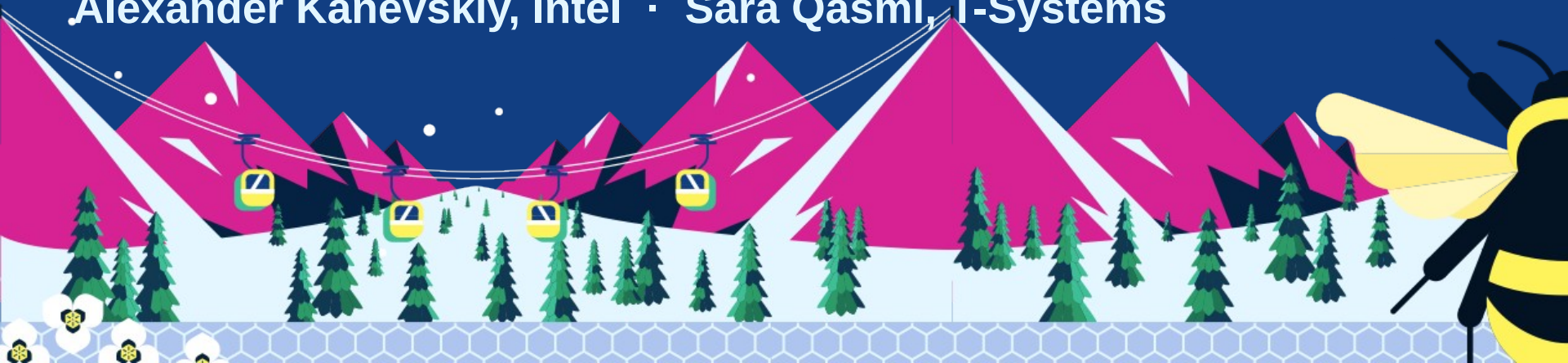
KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

Time to Learn New Tricks to Improve Performance on DRA-Managed Devices

Alexander Kanevskiy, Intel · Sara Qasmi, T-Systems



Who we are

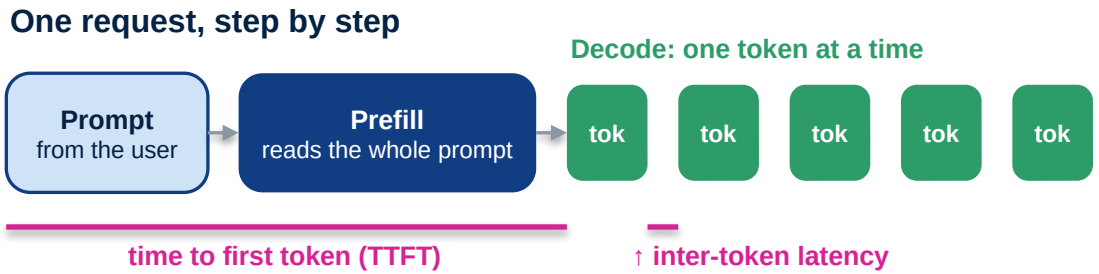


Alexander Kanevskiy
Principal Engineer, Cloud Software
Intel



Sara Qasmi
Platform Engineer
T-Systems

LLM serving: what we measure, and why



We measure	What it means for users
Time to first token (TTFT)	how long until the answer starts
Inter-token latency	how smoothly the answer streams
p99 of inter-token latency	the slowest 1% of gaps: the stutter people notice
Throughput: tokens/s per GPU	users per GPU, so cost

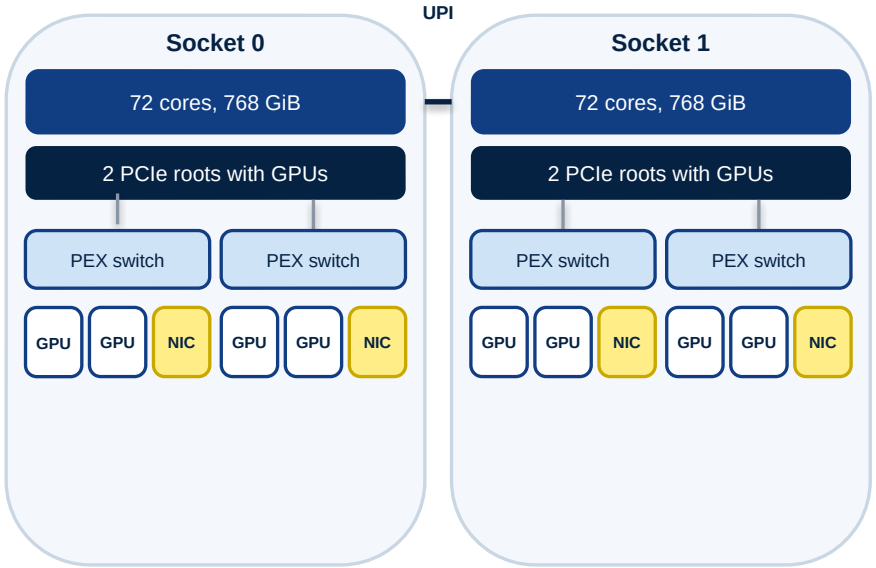
Every token crosses GPUs, CPUs, memory and PCIe links inside the box

Real platforms

KubeCon CloudNativeCon North America 2026

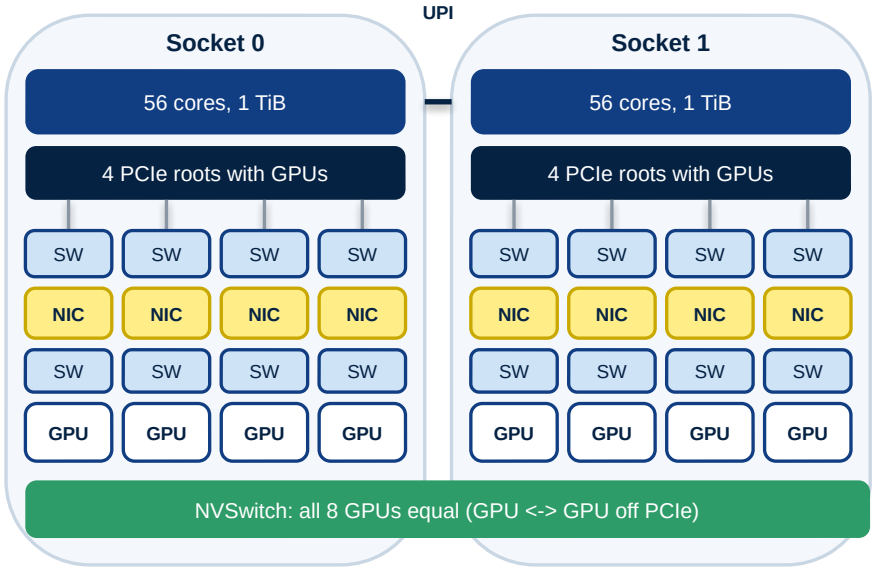
RTX PRO 6000 server

Intel Xeon 6 (P-core), 8 GPUs over PCIe, 4 x 400G RoCE NICs



DGX B200

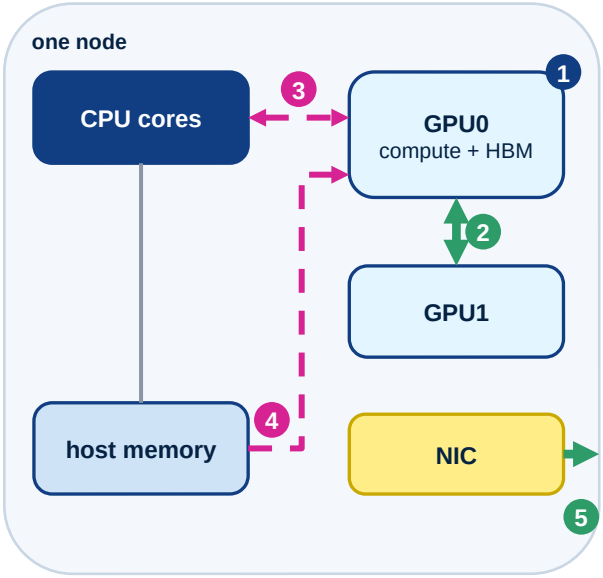
5th Gen Intel Xeon Scalable, 8 GPUs over NVSwitch, 8 x 400G IB NICs



PCIe root = CPU-side PCIe controller · **PEX / SW** = PCIe switch · **UPI** = link between sockets · **NVSwitch** = GPU-to-GPU fabric, off PCIe

Know your bottlenecks first

Observe → Identify → Know your hardware → Optimize what you ask for



#	Where it hurts	In LLM serving	What helps
1	**Inside the device** (compute, device memory bandwidth / capacity)	the GPU's own compute and memory	right device size or partition — topology will not help
2	**Device ↔ device** (P2P)	a replica's 2 GPUs exchange results, in prefill and decode	peers under one switch / one socket, P2P actually allowed
3	**Device ↔ CPU** (latency)	the scheduler hands every step to the GPU	driver / host threads on CPUs local to the device
4	**Device ↔ host memory** (bandwidth)	weights at start-up; KV cache that spills	buffers in memory local to the device's socket
5	**Node ↔ node** (fabric)	KV cache from the prefill to the decode node	each GPU next to its NIC

What we measured, path by path

KubeCon CloudNativeCon North America 2026

1. Best placement
devices chosen by hand = 100%

→

2. Default pod
no constraint in the claim, no CPU settings

→

3. Aligned pod
constraint in the DRA claim, or NRI

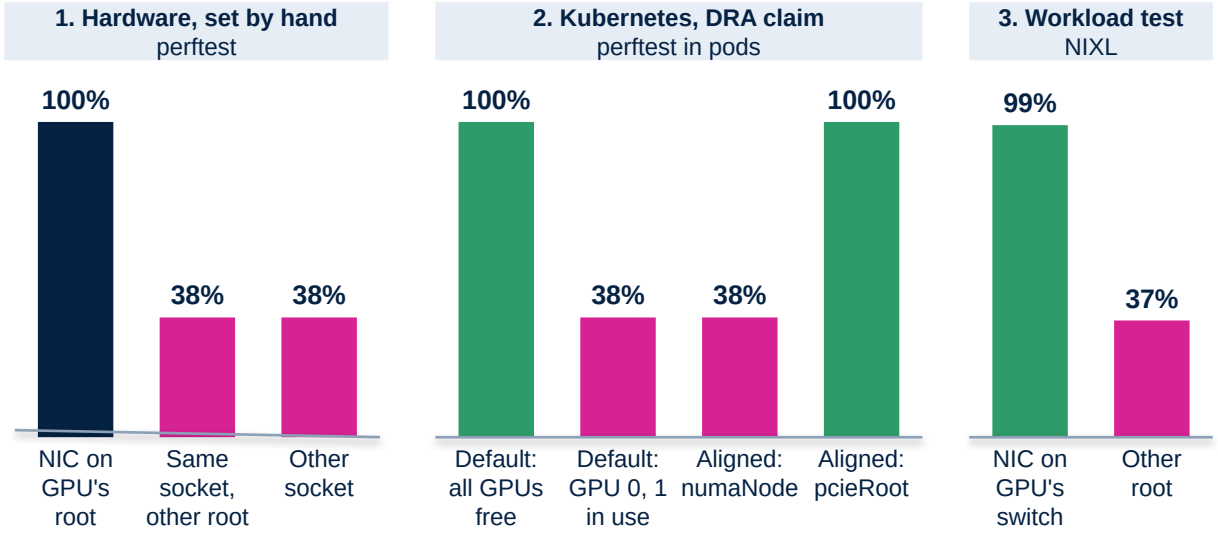
#	Path	Microbenchmark	Library / workload	Aligned with
5	Node ↔ node (GPU ↔ NIC, 2 nodes)	perftest GPUDirect RDMA	NIXL (KV-cache transfer)	DRA claim: GPU + NIC
2	Device ↔ device (GPU ↔ GPU)	nvbandwidth P2P	NCCL all_reduce (TP2/TP4/TP8); vLLM serving (TP2)	DRA claim: GPUs; NCCL_P2P_LEVEL
3	Device ↔ CPU (latency)	nvbandwidth host↔GPU latency	SGLang, vLLM: gpt-oss-120b	NRI topology-aware, CPU class
4	Device ↔ host memory	nvbandwidth, 8 GPUs at once	cold start (vLLM); HiCache memory audit	memory on the GPU's socket

Results are shown as % of the best placement on the same platform, unless stated. Default pods are tested with all GPUs free and with some GPUs already in use.

Path 5: GPU ↔ NIC

KubeCon CloudNativeCon North America 2026

B200: RDMA write between 2 nodes, % of best (higher is better)



Every pair in socket 0 (% of best)

GPU \ NIC	sw0	sw1	sw2	sw3
sw0	100	38	38	38
sw1	38	100	38	38
sw2	38	38	100	38
sw3	38	38	38	100

```
constraints:
- requests: [gpu, nic]
  matchAttribute:
    resource.kubernetes.io/pcieRoot
```

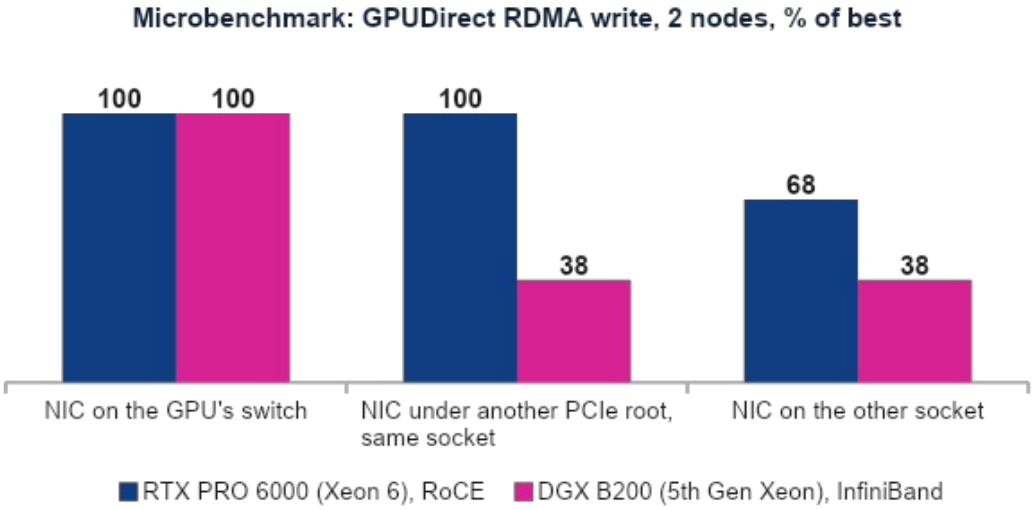
Method: 2 nodes, through the NIC. 100% = GPU memory on both ends, NIC on the GPU's switch, set by hand (layer 1).
1. perftest: host memory shows the network limit; then GPU memory, GPU fixed, NIC moved. 3 repeats, median.
2. Same perftest; a claim for 1 GPU + 1 NIC picks them. 1 pod pair per claim, 3 runs.
3. NIXL, the library that moves the KV cache: GPU memory, 4 GPU pairs, 64 MiB blocks, against 4 × best. 3 runs.

Microbenchmark · path 5: GPU ↔ NIC

In serving: impacts the KV cache transfer from prefill to decode nodes (PD)

Measured on one specific system. Your mileage may vary.

Path 5: GPU ↔ NIC, same claim, two answers



- **Xeon 6:** any NIC on the same socket gives full speed, so matching on socket is enough
- **5th Gen Xeon:** only the NIC on the GPU's own switch gives full speed, so pcieRoot is needed

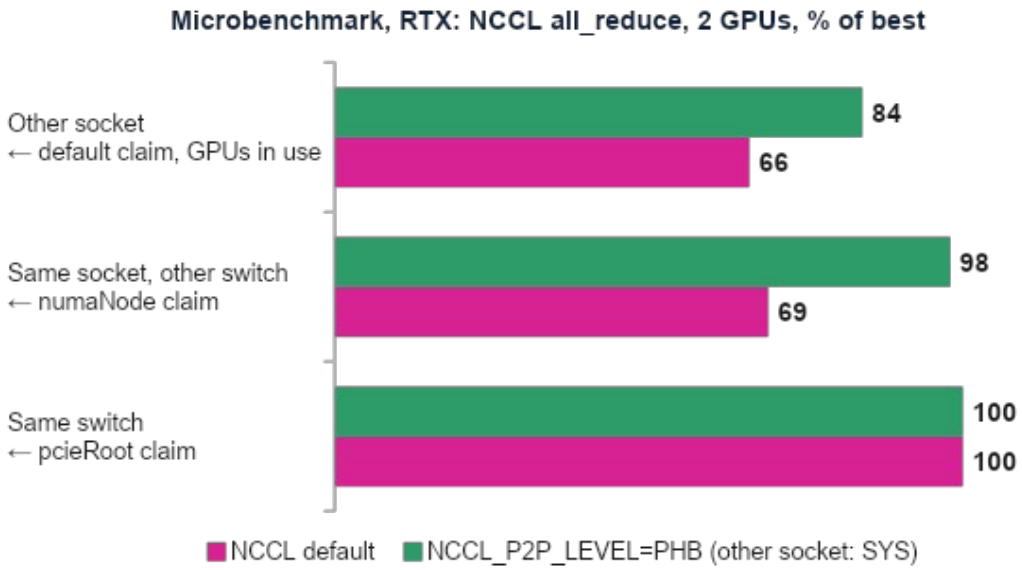
Method: same perftest on both platforms, 2 nodes, GPU fixed, only the NIC changes. Median of 3 runs (B200), 3 runs (RTX).

Microbenchmark · path 5: GPU ↔ NIC

In serving: impacts the KV cache transfer from prefill to decode nodes (PD)

Measured on one specific system. Your mileage may vary.

Path 2: GPU ↔ GPU: PCIe



Method: NCCL all_reduce, 1 GiB, 2 GPUs, median of 3 runs, idle node. Default claim's pick: 5 fresh pods with 3 GPUs already in use.

- **The claim picks the pair:** on a busy node, the default claim took the across-sockets pair in 5 of 5 pods
- **The library picks the path:** by default NCCL copies directly only under one switch, otherwise through host memory
- **It isn't the hardware:** raw copies between switches reach 100% on the same socket

```
constraints:
- requests: [gpu]
  matchAttribute:
    resource.kubernetes.io/numaNode
```

```
If the pair is on two switches, env:
NCCL_P2P_LEVEL=PHB
```

Microbenchmark · path 2: GPU ↔ GPU

In serving: prefill, decode. Two switches: TTFT p99 ITL ~+10%, throughput up to −6%

Measured on one specific system. Your mileage may vary.

GPU groups: claim and NCCL level must fit together

KubeCon CloudNativeCon North America 2026

GPU group (RTX PRO 6000)	Claim	NCCL_P2P_LEVEL	Microbenchmark: NCCL all_reduce vs default
TP2, one switch	**pcieRoot**	default is enough	already direct: no gain from a level
TP4, one socket	**numaNode**	**PHB**	**+40%**
TP8, whole node	none	**SYS**, only with no memory-heavy neighbor	**+21%**

Workload (vLLM TP2): two switches lose up to 6% throughput as load grows, TTFT p99 is about 10% worse at every load, inter-token p99 11% worse at 32–64 requests; under one switch or with PHB, 100%.

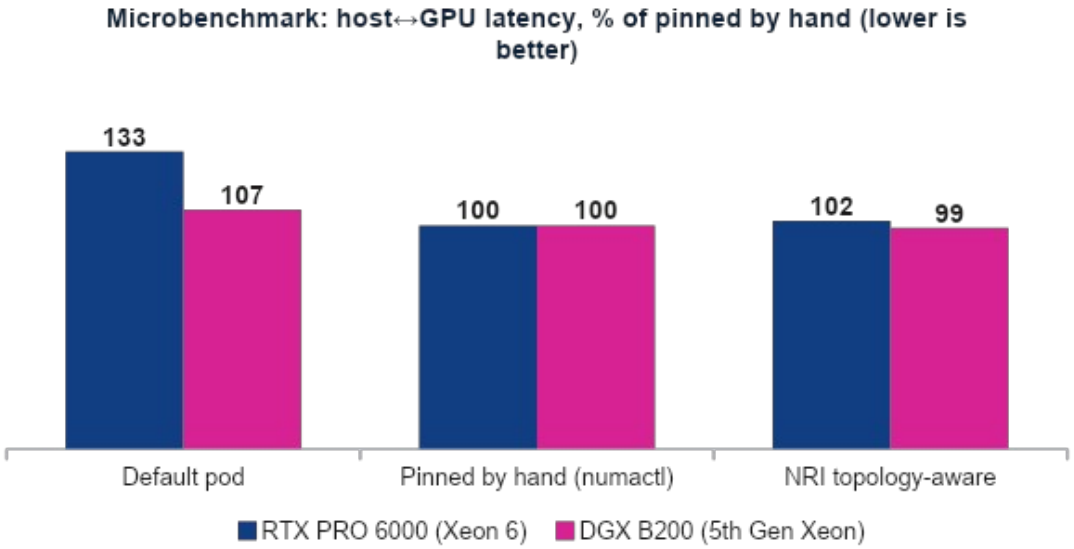
Method: NCCL all_reduce, 1 GiB messages, median of 3 runs, idle node. Workload: gpt-oss-120b, 8k tokens in, 1k out, 3 rounds. TP4 and TP8

Microbenchmark + workload · path 2

In serving: prefill, decode. Two switches: TTFT p99 ~+10%, throughput up to −6%

Measured on one specific system. Your mileage may vary.

Patch 3: CPU ↔ GPU



Method: nvbandwidth latency, 1 GPU, 10 fresh pods per case, median, idle node, GPU on socket

Default: cpuset 0-287
all CPU threads, both sockets

NRI: cpuset 73-143,216-287
socket 1 only, memory too

- No CPU settings in the pod
- NRI placed it on GPU's socket

Microbenchmark · path 3: CPU ↔ GPU

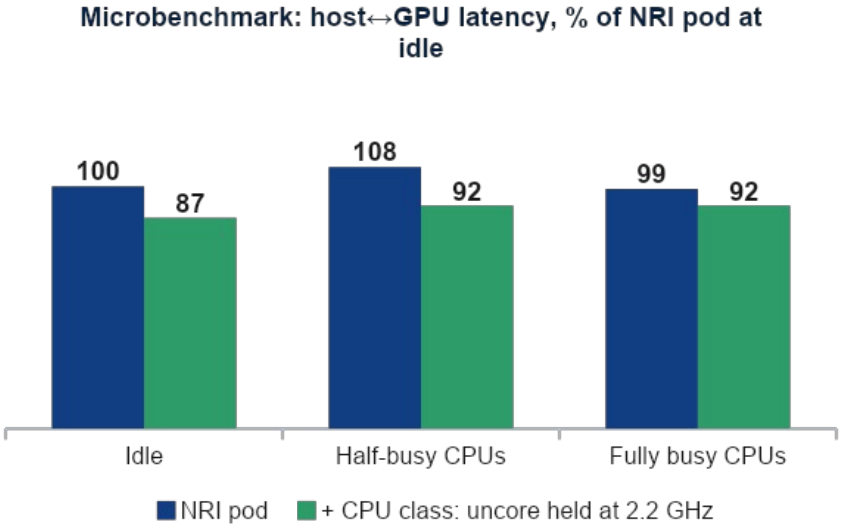
In serving: scheduler. Same throughput; earlier first token if busy; low-load p99 ≈ noise

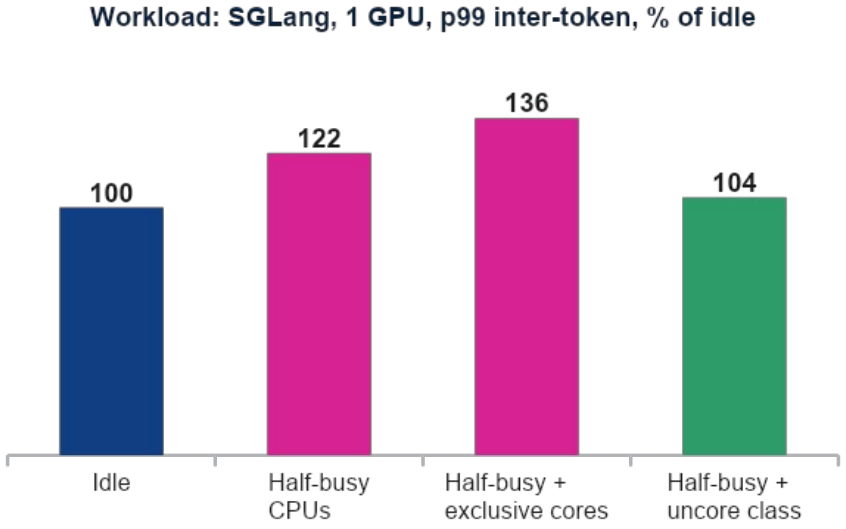
Measured on one specific system. Your mileage may vary.

Xeon 6: uncore clock cuts latency

KubeCon CloudNativeCon North America 2026

Both charts: RTX PRO 6000 server (Xeon 6), pod on the GPU's socket through NRI (lower is better)





Method: nvbandwidth: 10 fresh pods per case, median.
SGLang: 1 user, 1k tokens in, 256 out, 3 runs, mean. Half / fully busy: neighbor pod with 144 workers, 64-core limit / no limit.

Microbenchmark + workload · path 3

In serving: scheduler. Workload: ITL p99 −22% at low load with the CPU class

Measured on one specific system. Your mileage may vary.

KubeCon CloudNativeCon
North America 2026

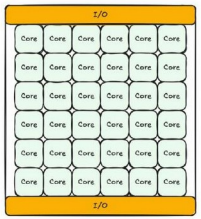
Why: inside the box, the kernel and DRA



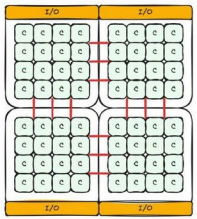
Where PCIe lives: I/O dies and many root complexes per socket

- PCIe/CXL lanes come out of I/O dies (Xeon 6) or the I/O die (EPYC)
- Each I/O stack is exposed to Linux as its **own PCI host bridge / root bus**: `pci0000:00`, `pci0000:15`, `pci0000:80`, ...
- A 2-socket box typically shows 6-16 root buses
- Example from a dual-socket 3rd Gen Xeon Scalable server (from KEP-5941):

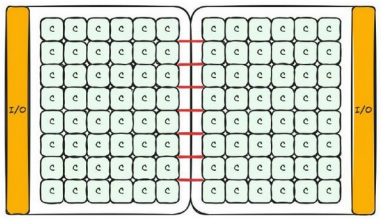
```
"root"="pci0000:00" "localCPUs"="0,2,4,...,102" "NUMANode"=0
"root"="pci0000:17" "localCPUs"="0,2,4,...,102" "NUMANode"=0
"root"="pci0000:3a" "localCPUs"="0,2,4,...,102" "NUMANode"=0
"root"="pci0000:85" "localCPUs"="1,3,5,...,103" "NUMANode"=1
"root"="pci0000:ae" "localCPUs"="1,3,5,...,103" "NUMANode"=1
"root"="pci0000:d7" "localCPUs"="1,3,5,...,103" "NUMANode"=1
```



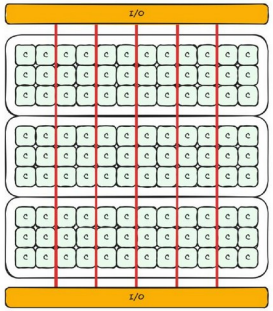
Traditional



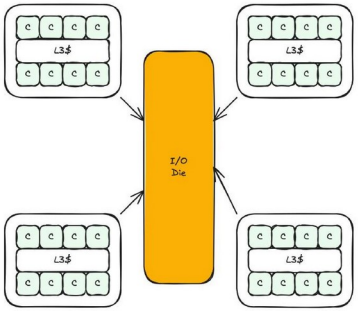
4th Gen Intel® Xeon®



5th Gen Intel® Xeon®



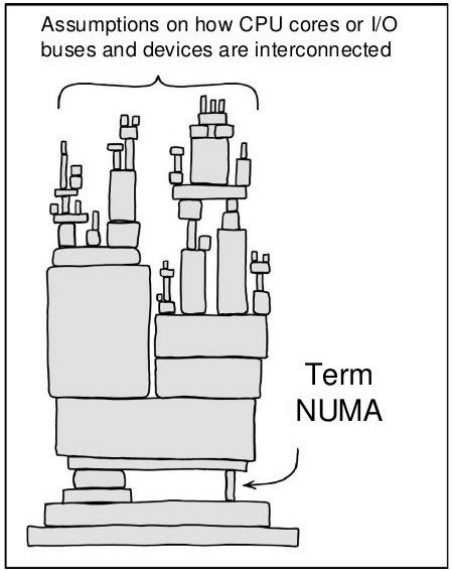
Intel® Xeon® 6



AMD Epyc

Why NUMA is the wrong topology selector for devices

1. **It is a memory partition, not an I/O map.** SNC/NPS change NUMA ids; the PCIe wiring does not change.
2. **Granularity is the host bridge.** Linux gives every device under one root bus the same `numa_node` (from ACPI `_PXM` on the root bridge). No switch, no root port, no die.
3. **It depends on firmware.** No `_PXM` -> `numa_node = -1`. On modern Intel and AMD Zen there is no hardware fallback.
4. **`-1` poisons the rest.** `local_cpulist` then reports *all* CPUs; "first CPU's package" silently says socket 0.
5. **NPS0 / single-node BIOS:** every device reports the same node, so "match NUMA" matches everything.
6. **Kernel P2P logic never looks at NUMA** (next part).



Linux has no interface that ties a PCI device to a die, chiplet, L3 domain or I/O-die quadrant: PCI locality stops at the NUMA node.

Kernel pseudo-code: "may A and B talk directly?"


```
/* pci_p2pdma_distance(provider, client) -> calc_map_type_and_dist()   p2pdma.c:705 */
map_type p2p_admission(pci_dev *A, pci_dev *B)
{
    /* 1. Walk up both trees looking for a common upstream bridge (a switch) */
    for (a = A; a; a = upstream_bridge(a)) {
        acs += has_acs_redirect(a);             /* ACS ctrl & (RR|CR|EC) */
        for (b = B; b; b = upstream_bridge(b))
            if (a == b) goto common_switch;
    }
    goto through_host_bridge;                   /* different root ports / RCs */
common_switch:
    for (b = B; b != a; b = upstream_bridge(b))
        acs += has_acs_redirect(b);
    if (acs == 0)
        return BUS_ADDR;                        /* switch-local P2P, no IOMMU */
    warn("add pci=disable_acs_redir=...");      /* ACS forces it up to the RC */
through_host_bridge:
    if (cpu0.vendor == AMD && cpu0.family >= 0x17)    /* Zen+: any topology */
        return THRU_HOST_BRIDGE;
    /* host bridge "identity" = vendor:device of the first device on its root bus
       (only if it is devfn 00.0 or a root port; otherwise: unknown -> refused) */
    ha = find_host_bridge(A);  hb = find_host_bridge(B);
    if (ha == hb ? whitelisted(ha, SAME_HB)
                 : whitelisted(ha, OTHER_HB) && whitelisted(hb, OTHER_HB))
        return THRU_HOST_BRIDGE;                /* IOVA mapping via IOMMU */
    return NOT_SUPPORTED;                       /* distance() == -1 */
}
```

Who decides? This is the in-tree answer (Intel `xe`, AMD `amdgpu`, NVMe, RDMA importers). NVIDIA GPUs decide in their own out-of-tree driver and closed CUDA components, with NCCL's `NCCL_P2P_LEVEL` on top — invisible to the scheduler and to this code.

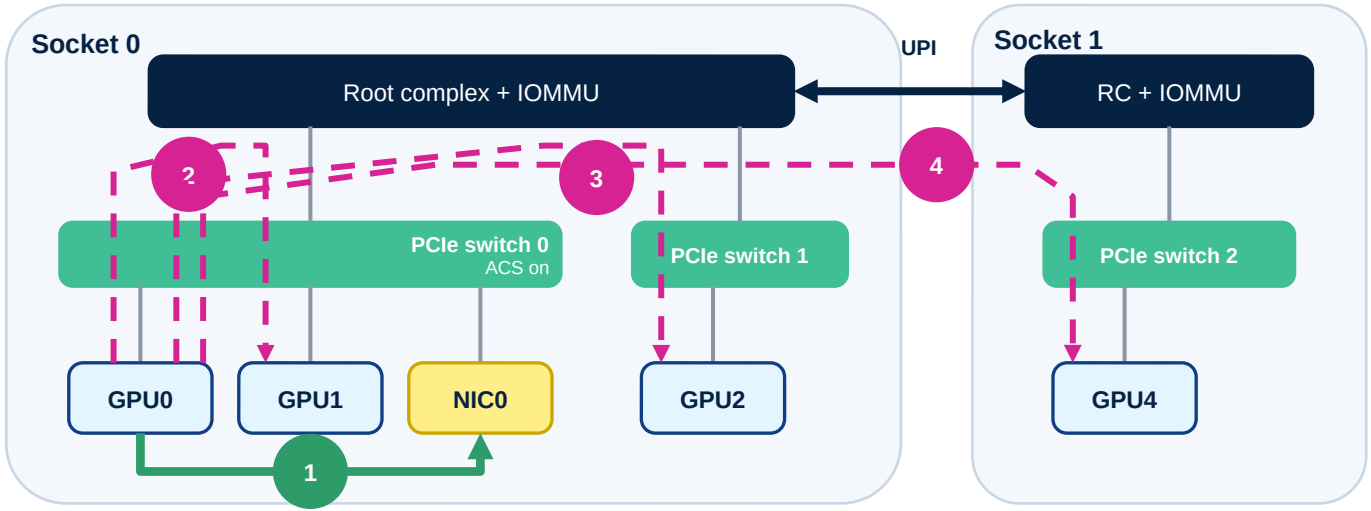
The whitelist: CPU generation decides, topology does not

`drivers/pci/p2pdma.c` `pci_p2pdma_whitelist[]`, matched on the **first device of each root bus**:

CPU / platform	Same host bridge	Different host bridges (other die or other socket)	Mechanism
Intel Xeon E5 v1, Xeon E5/E7 v3	allowed	**refused**	`REQ_SAME_HOST_BRIDGE` (`0x3c00/01`, `0x2f00/01`)
Intel Xeon E5 v2 / v4	**refused**	**refused**	not in table
Intel Xeon Scalable 1st-3rd Gen	allowed	allowed	IDs `0x2020`, `0x2030-0x2033`, `0x09a2`
Intel Xeon Scalable 4th/5th Gen, Xeon 6, next-gen Xeon	allowed	allowed	`0x09a2` or DSA/IAA/QAT as first device
AMD Zen (family >= 0x17)	allowed	allowed	CPUID check, no table, no topology
Hygon	refused	refused	different vendor id, not matched
Google SoCs / Nvidia Vera CPU	allowed	allowed	table entries
Anything else (incl. **QEMU q35 / i440fx / pxb**)	**refused**	**refused**	not in table

- "Other I/O die, same socket" and "other socket" are the **same branch** in the code
- The kernel says "yes" to cross-socket P2P on every modern CPU — physics may still say "slow"
- Upstream plan: replace the table with ACPI HMAT P2P characteristics (not implemented yet)
- **Measured (Part 1, NVIDIA GPUs):** GPU ↔ NIC under another root of the same socket — 100% on Xeon 6, about 38% on 5th Gen Xeon. Software allows the path on both; the speed differs by CPU generation, the exact limit in the I/O fabric is still under investigation → measure your platform

Traffic flows on bare metal (Variant B, modern Xeon, IOMMU on)



1. same switch, redirect cleared → BUS_ADDR, switch-local
2. same switch, ACS RR/CR on → up to RC + IOMMU and back
3. other switch → THRU_HOST_BRIDGE (whitelist)
4. other socket → allowed, crosses UPI

All four are "allowed" on modern Xeon / AMD — the danger on bare metal is not failure, it is that P2P works slowly. The kernel only decides if a path is allowed; which devices the Kubernetes scheduler allocates (DRA) decides which path your all-reduce takes.

The kernel enables ACS redirect when the IOMMU remaps DMA (intel_iommu=on); iommu=pt alone doesn't change it. Disable it only on bare-metal nodes, since VMs with VFIO need it.

What drivers publish today (Oct 2026)

KubeCon CloudNativeCon North America 2026

Driver	`pcieRoot`	`pciBusID`	`numaNode`	Socket	PCIe switch	P2P / fabric	VFIO / KubeVirt
NVIDIA GPU	yes	yes	yes (list behind driver gate)	-	-	NVSwitch FM `partitionN`	yes
dranet (NICs)	yes	`dra.net/pciAddress`	yes (list behind gate)	-	-	cloud fabric attrs	**excludes vfio NICs**
Intel GPU	yes	yes	**no**	-	-	-	yes
Intel Gaudi	yes	-	-	-	-	-	-
Google TPU	-	-	-	-	-	node-level `topology` string	-
dra-driver-cpu	**list** (opt-in, amd64)	n/a	yes (scalar)	`dra.cpu/socketID`	n/a	n/a	n/a
dra-example-driver	mock, opt-in	-	-	-	-	-	-
nri-plugins topology-aware (v0.15)	**list** (socket's roots, if API accepts lists)	n/a	yes (scalar)	`dra.cpu/socketID`	n/a	n/a	n/a

- Nobody publishes a **PCIe switch**, a **P2P group**, or a **standard socket**
- NUMA (where published) comes from `/sys/bus/pci/devices/*/numa_node` — with all the problems from Part 2

Tricks 1+2 — standard `numaNode` and list attributes

- **One name for every driver:** `resource.kubernetes.io/numaNode` (KEP-6072) — **Status:** 1.37 stable · 1.38 stable
- **Lists** (KEP-5491): `ints`, `strings`, … ; `matchAttribute` = non-empty intersection across all devices — **Status:** `DRAListTypeAttributes` 1.36 alpha · 1.37 alpha · 1.38 beta (planned)
- `numaNode` as a list = own node + same-socket nodes at minimum SLIT distance: covers the socket on AMD NPS4, **not** on Xeon 6 with SNC-3
- Still NUMA: in Part 1, a same-NUMA claim got a NIC under another PCIe root (38%); `pcieRoot` fixed it

Device	Physical NUMA	`numaNode` (list)
CPU set	4	`4`
NVMe	0	`[0, 1, 2, 3]`
NIC VF	6	`[6, 4, 5, 7]`

```
constraints:
- matchAttribute:
resource.kubernetes.io/numaNode
  requests: [gpu, nic, cpu]
```

Trick 3 — derived attributes: one intent, a rule per platform

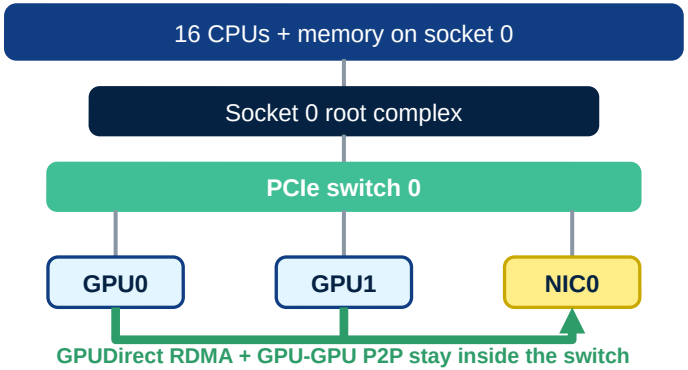
- The **workload author** computes a grouping key per candidate device with CEL, inside the claim; `matchAttribute` compares the results
- Different requests can derive the **same key from different driver attributes**
- **Status:** `DRADerivedAttributes` — 1.37 alpha · 1.38 beta (planned)
- Part 1 in one line: the **P2P domain** is the socket on Xeon 6, but the PCIe root (switch) on 5th Gen Xeon
- Names must be domain-qualified (`derived/…`); CEL sees one device only; a CEL error fails the pod — guard with selectors

```
# Xeon 6 node pool (SNC-3: 3 NUMA nodes per socket) — P2P domain = socket
- name: gpu
  exactly:
    deviceClassName: gpu.nvidia.com
    count: 2
    selectors:
    - cel: { expression: '"numaNode" in
device.attributes["resource.kubernetes.io"]' }
    derivedAttributes:
    - name: derived/p2pDomain
      expression: device.attributes["resource.kubernetes.io"].numaNode / 3
- name: nic
  exactly:
    deviceClassName: dranet
    selectors:
    - cel: { expression: '"numaNode" in device.attributes["dra.net"]' }
    derivedAttributes:
    - name: derived/p2pDomain
      expression: device.attributes["dra.net"].numaNode / 3
constraints:
- requests: [gpu, nic]
  matchAttribute: derived/p2pDomain
```

```
# 5th Gen Xeon node pool — P2P domain = PCIe root, no derivation needed
constraints:
- requests: [gpu, nic]
  matchAttribute: resource.kubernetes.io/pcieRoot
```

Putting it together: one claim, the right box-local path

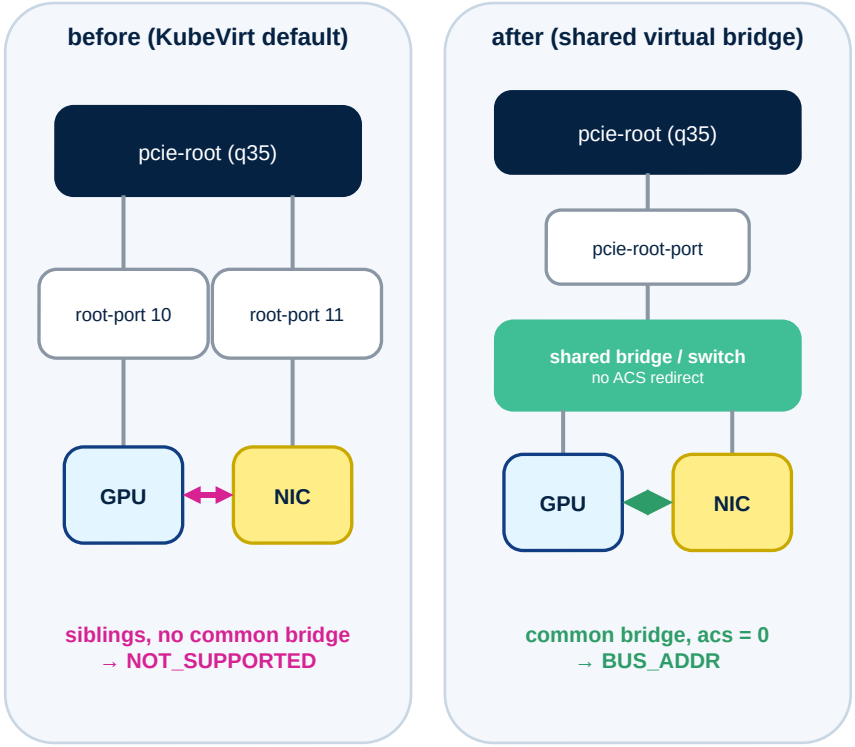

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplate
metadata:
  name: gpudirect-pair
spec:
  spec:
    devices:
      requests:
      - name: gpu       # 2 GPUs ...
      - name: nic       # ... 1 RDMA NIC ...
      # ... and 16 CPUs, all from one socket
      - name: cpu
      constraints:
      - requests: [gpu, nic]
        # P2P stays in its domain
        matchAttribute: derived/p2pDomain
      - requests: [gpu, nic, cpu]
        # host-side work stays on the socket
        matchAttribute: derived/socket
```



- Constraint 1: the P2P path stays in its P2P domain
- Constraint 2: driver threads, NCCL proxies and pinned buffers stay on the same socket
- KEP-5517 makes the CPU and memory part **real**: they are accounted in node allocatable and enforced in cgroups

VMs: the guest kernel decides on a virtual tree

- KubeVirt passthrough GPU (`GPUsWithDRA`, beta in v1.9) + RDMA NIC VF: the **guest kernel** runs the same P2P admission check
- Default guest layout: **one root port per device** on the q35 host bridge → no common bridge, q35 not whitelisted → **P2P refused** on Intel guests
- NUMA-aware placement (VEP-115) fixes guest NUMA locality, **not** P2P — P2P needs a shared bridge, not a shared NUMA node
- **Idea:** put the devices that must talk behind **one virtual bridge or switch** → guest returns `BUS_ADDR`; needs a domain-XML rewrite today (prototyped in our lab)
- KEP-5304 device metadata already gives KubeVirt each device's `pciBusID`; it could also carry `pcieRoot` / `numaNode` to drive guest placement



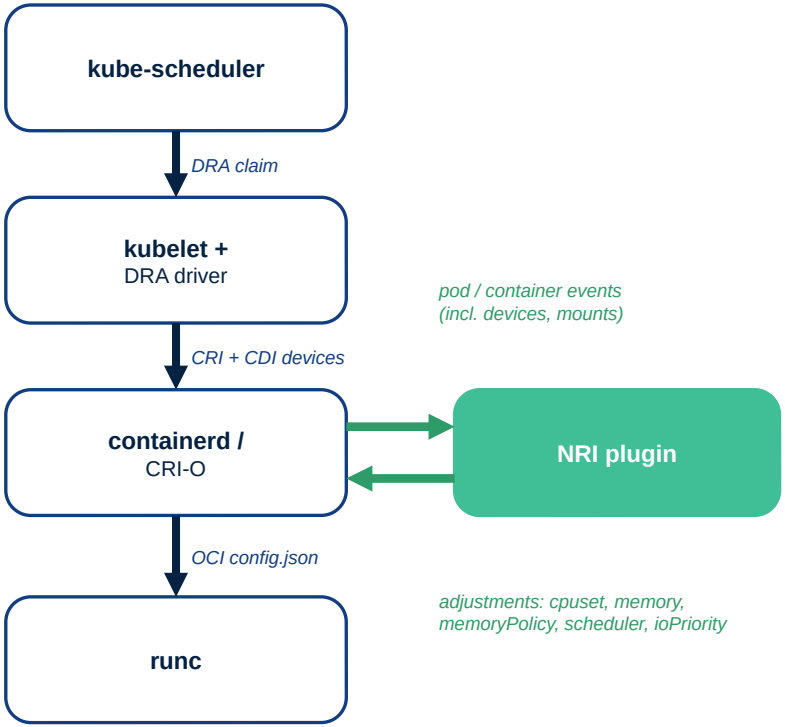
Trick 4 — node allocatable resources through DRA (KEP-5517)

- Problem: CPU/memory from a DRA driver (dra-driver-cpu) was **published twice** (node allocatable + ResourceSlice) and **consumed twice** (pod spec + claim) → overcommit, wrong cgroups
- A device declares how it maps to node allocatable `cpu`, `memory`, `hugepages-*`:
- Scheduler accounts it against node allocatable; kubelet puts it into cgroups and OOM score
- **Status:** `DRANodeAllocatableResources` — 1.36 alpha · 1.37 alpha · 1.38 beta (planned; off by default unless the minimum compatibility version is 1.38)
- Used today by dra-driver-cpu (opt-in), dra-example-driver (cpu profile), and as an option by the nri-plugins topology-aware DRA driver (v0.15)

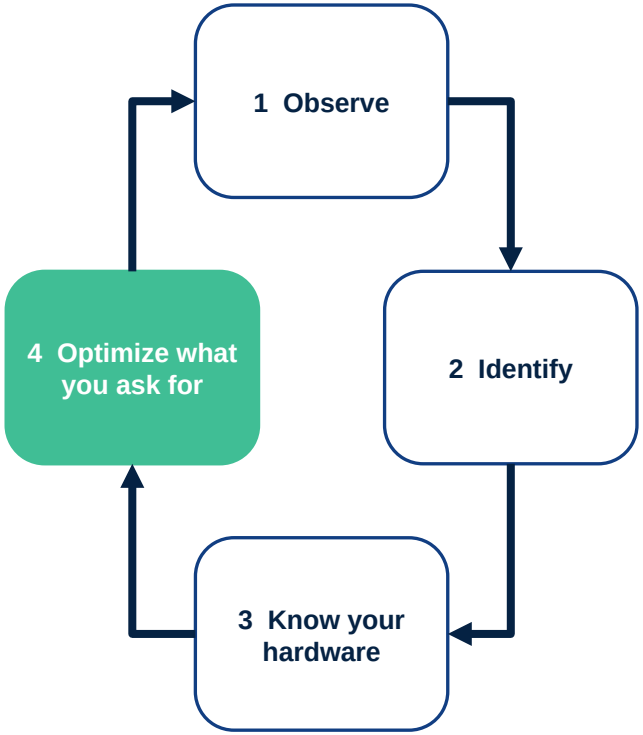
```
devices:
- name: cpudevnuma000
  allowMultipleAllocations: true
  capacity:
    dra.cpu/cpu: { value: "64" }
  nodeAllocatableResources:
    cpu:
      mapping:
        capacityKey: dra.cpu/cpu
        capacityMultiplier: "1"
# ...and an accelerator that needs host memory per pod
- name: xpu-model-x-001
  nodeAllocatableResources:
    memory:
      overhead:
        perPod: "1Gi"
        perContainer: "500Mi"
```

NRI: exact per-container placement and a few faster cores

- NRI plugins sit **beside containerd / CRI-O** and see the final container: devices from DRA/CDI, mounts, pod resources
- nri-plugins topology-aware policies turn those devices into CPU and memory placement — Part 1: a default pod costs up to +33% host↔GPU latency and NRI fixes it, but serving gained **no throughput** (only an earlier first token with busy CPUs)
- **v0.15:** topology-aware is also a **DRA driver** (`topology-aware.nri.io`): NUMA nodes as devices with `numaNode`, `dra.cpu/socketID`, `pcieRoot`; CPUs as consumable `dra.cpu/cpu`; per-claim memory policy; device IRQs routed to the container's CPUs
- Uncore minimum frequency per CPU class removed the p99 stalls on a half-busy Xeon 6 socket at low load
- **Priority Core Turbo** (Xeon 6): a few high-priority cores at maximum turbo, enforced by hardware (SST); balloons / topology-aware CPU classes `pctPriority: high|low` — for workloads whose **host side is on the critical path**
- **Status:** nri-plugins v0.15 (PCT since v0.13); **work in progress:** expose PCT cores through DRA together with KEP-5517 (1.38 beta, planned)



Observe, identify, know your hardware, optimize what you ask for



1. **Observe** — device utilization and telemetry, PCIe / inter-socket / memory bandwidth counters (uncore PMUs via `perf`), kernel P2P messages in `dmesg`
2. **Identify** — which path is the bottleneck: inside the device, device ↔ device, device ↔ CPU, device ↔ memory, node ↔ node
3. **Know your hardware** — switches, root complexes, sockets, BIOS NUMA mode (SNC/NPS), ACS / IOMMU settings, what the kernel and the device drivers actually allow
4. **Optimize what you ask for** — request only the constraints that address *that* bottleneck: `matchAttribute` on a derived P2P domain, socket or switch; CPUs and memory via DRA (KEP-5517) or NRI policies

- Every extra constraint costs schedulability — ask for what matters, not for everything

Key takeaways

- **Measure first:** default claims picked the slow path by chance; `pcieRoot` (plus the right `NCCL_P2P_LEVEL`) fixed GPU ↔ NIC and GPU ↔ GPU; aligning CPU/memory gave no throughput gain in serving
- DRA puts **device selection in the user's hands at scheduling time** — use it
- Vendor driver attributes are a starting point, **not the whole truth**:
  - some facts are missing (switch, P2P group, socket)
  - some are misleading (`numaNode` under SNC/NPS, `-1` without `_PXM`)
  - some must be **derived or reinterpreted** for your fleet (`derived/p2pDomain`, `derived/socket`)
- New tools: standard `numaNode` (6072), list attributes (5491), **derived attributes** (6080), node allocatable resources (5517), device metadata (5304)
- Driver authors: publish `pcieRoot`, `pciBusID`, `numaNode` (list form) — and please a switch / P2P-group attribute
- Platform teams: check ACS/IOMMU (bare metal: ACS redirect off on GPU/NIC switch ports; VM nodes: keep it), BIOS NUMA mode and library defaults (e.g. `NCCL_P2P_LEVEL`); they can undo a perfect schedule

KubeCon CloudNativeCon
North America 2026

Thank you!

Questions?

Time to Learn New Tricks to Improve Performance on DRA-Managed Devices



KubeCon CloudNativeCon
North America 2026

Appendix



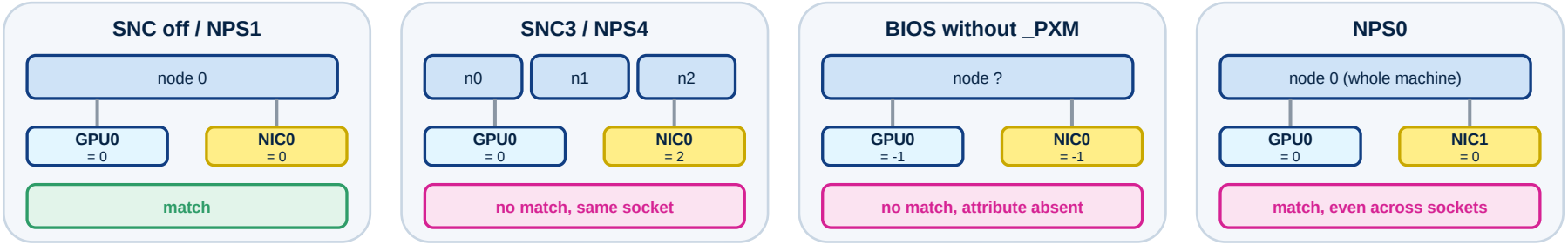
The cost ladder of a device-to-device transfer

Level	Path	Who routes the TLPs	Typical outcome
PIX	Same PCIe switch	Switch	Full link speed, lowest latency — *if* ACS lets it
PXB	Several switches, no host bridge	Switches	Close to PIX
PHB	Through the root complex of one host bridge	CPU root complex (+ IOMMU)	Depends on CPU; RC P2P throughput is often lower
NODE	Between host bridges inside one NUMA node / socket	CPU mesh / I/O die fabric	Lower, varies by CPU generation
SYS	Across sockets	UPI / xGMI	Often much lower; shares the link with memory traffic
NET	Across nodes	NIC + fabric (RDMA)	Good only if GPU <-> NIC is itself PIX/PXB
bounce	Device -> host DRAM -> device	CPU memory	Two DMA copies, CPU memory bandwidth consumed

- Every row down the table adds latency, removes bandwidth, or both
- The scheduler today sees none of these rows

One box, three BIOS settings, three different answers

Same dual-socket server, GPU0 and NIC0 both wired to socket 0:



BIOS setting	Socket 0 NUMA nodes	GPU0 `numa_node`	NIC0 `numa_node`	`matchAttribute: numaNode` (scalar)
SNC off / NPS1	0	0	0	match
SNC3 / NPS4	0, 1, 2 (or 0-3)	0	2	**no match** — same socket
BIOS without _PXM	—	-1 (not published)	-1 (not published)	**no match** — attribute absent
NPS0	0 (whole machine)	0	0	match — even across sockets

Even the "smart" NUMA list is not "same socket": real Xeon 6 SNC-3 distances

Dual-socket Intel Xeon 6, SNC-3 — SLIT table quoted in the kernel source:

```
node     0    1    2    3    4    5
    0:   10   15   17   21   28   26
    1:   15   10   15   23   26   23
    2:   17   15   10   26   23   21
    3:   21   28   26   10   15   17
    4:   23   26   23   15   10   15
    5:   26   23   21   17   15   10
```

KEP-6072 list form = own node + same-socket nodes at the **minimum** distance:

Device on node	Published `numaNode` list
0	`[0, 1]`
1	`[1, 0, 2]`
2	`[2, 1]`

- GPU on node 0 `[0,1]` ∩ NIC on node 2 `[2,1]` = `{1}` -> match (by luck of the middle cluster)
- ...plus a CPU set on node 0 (scalar `0`) -> `{0,1}` ∩ `{2,1}` ∩ `{0}` = ∅ -> **no match**, same socket

ACS: the switch is there, but the traffic still goes up

- **ACS** (Access Control Services) on switch downstream ports and root ports
  - RR / CR: redirect peer requests / completions **upstream** to the root complex
  - EC: egress control; UF: upstream forwarding; DT: direct translated P2P (ATS)
- Who turns it on: **IOMMU drivers** (VT-d, AMD-Vi, ARM IORT/SMMU, virtio-iommu/VIOT, OF, Xen PV) call `pci_request_acs()`
- Result with IOMMU on: P2P under a switch becomes `THRU_HOST_BRIDGE`

```
/* boot: IOMMU driver -> pci_request_acs() */
pci_enable_acs(port)                                    /* pci.c:1028 */
{
    ctrl = fw_ctrl = read(ACS_CTRL);
    if (iommu_requested_acs)
        /* redirect peer TLPs up */
        ctrl |= cap & (SV | RR | CR | UF);
    if (match("pci=disable_acs_redir=", port)) ctrl = fw_ctrl & ~(RR|CR|EC);
    if (match("pci=config_acs=<bits>@",  port)) ctrl = apply(fw_ctrl, bits);
    write(ACS_CTRL, ctrl);
}

/* iommu.c:1623 */
pci_device_group(dev)
{
    for (br = up(dev); !root_bus(br); br = up(br))
        if (acs_path_enabled(br, SV|RR|CR|UF)) break;    /* isolated */
        /* no ACS -> shared group */
        else join_group(br);
}
/* ACS on  => own IOMMU group (VFIO-assignable), P2P goes via RC (whitelist!)
 * ACS off => BUS_ADDR possible, but devices merge into one IOMMU group       */
```

Trick 2 — list attributes (KEP-5491)

- An attribute value can be a list: `ints`, `strings`, `bools`, `versions`
- `matchAttribute` = **non-empty intersection across all devices**; scalars count as one-element lists
- `distinctAttribute` = pairwise **disjoint**
- CEL: `device.attributes["d"].x.includes(v)` works on scalars and lists
- **Status:** `DRAListTypeAttributes` — 1.36 alpha · 1.37 alpha (off by default) · 1.38 beta (planned)

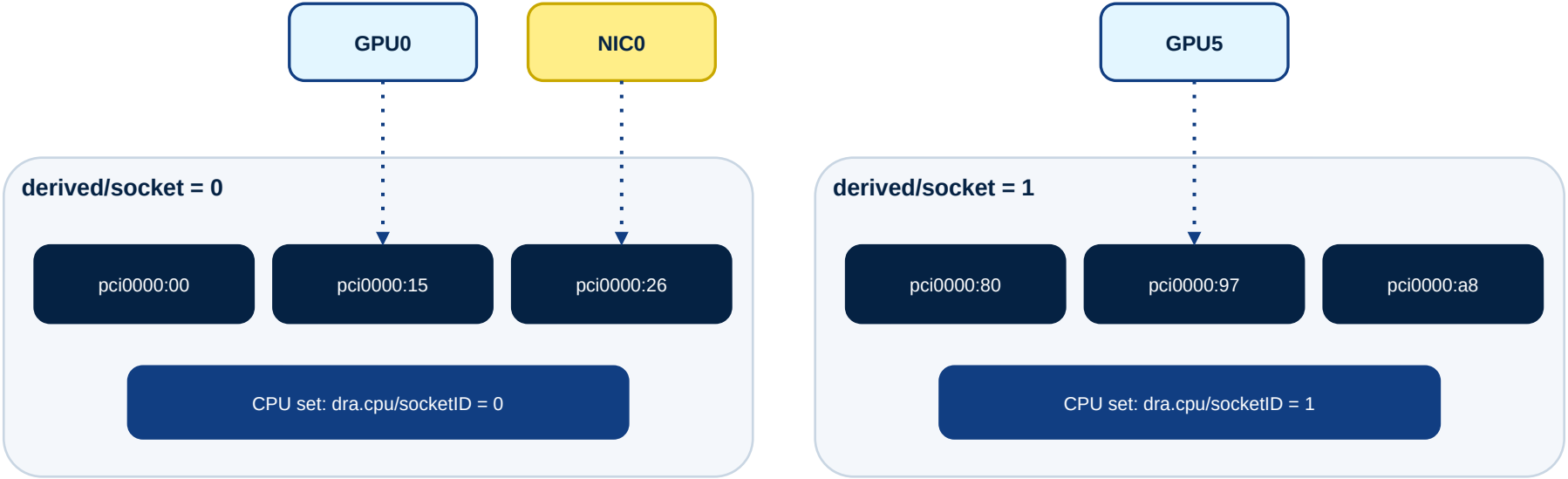
```
# CPU set from dra-driver-cpu (--expose-pcie-roots)
resource.kubernetes.io/pcieRoot:
  strings: [pci0000:00, pci0000:10]
```

Gotcha — intersection is **global**, not pairwise:

```
CPU  {pci0000:00, pci0000:10}
GPU  {pci0000:00}
NIC  {pci0000:10}
CPU ∩ GPU ∩ NIC = ∅   -> claim stays Pending, although both devices are "next to" the CPU set
```

Derived "socket": make every root on a socket look the same (1/3)

Goal: GPU, NIC and CPU set **on the same socket**, independent of SNC/NPS and of how many roots the socket has.



rule on this platform: pcieRoot ≥ "pci0000:80" → socket 1, else 0. GPU0 + NIC0 + CPU set match on derived/socket = 0 although their pcieRoot values differ

Derived "socket": make every root on a socket look the same (2/3)


```
- name: gpu
  exactly:
    deviceClassName: gpu.nvidia.com
    count: 2
    selectors:
    - cel: { expression: '"pcieRoot" in device.attributes["resource.kubernetes.io"]' }
    derivedAttributes:
    - name: derived/socket
      # This platform: BIOS gives socket 1 the upper half of the bus space
      expression: >-
        device.attributes["resource.kubernetes.io"].pcieRoot >= "pci0000:80" ? 1 : 0
- name: nic
  exactly:
    deviceClassName: dranet
    selectors:
    - cel: { expression: '"rdma" in device.attributes["dra.net"] && device.attributes["dra.net"].rdma' }
    - cel: { expression: '"pcieRoot" in device.attributes["resource.kubernetes.io"]' }
    derivedAttributes:
    - name: derived/socket
      expression: >-
        device.attributes["resource.kubernetes.io"].pcieRoot >= "pci0000:80" ? 1 : 0
```

Derived "socket": make every root on a socket look the same (3/3)


```
- name: cpu
  exactly:
    deviceClassName: dra.cpu
    capacity: { requests: { dra.cpu/cpu: "16" } }
    selectors:
    - cel: { expression: '"socketID" in device.attributes["dra.cpu"]' }   # grouped by socket/NUMA
    derivedAttributes:
    - name: derived/socket
      expression: device.attributes["dra.cpu"].socketID       # rename, no math
constraints:
- requests: [gpu, nic, cpu]
  matchAttribute: derived/socket
```

Derived "P2P group": what the drivers don't publish (1/2)

On a Variant-B box, every switch owns a contiguous **bus range**. Derive a switch id from the device's own BDF:

- Unknown devices fall back to their own BDF → they only "match" themselves, never a wrong peer
- GPU and NIC use **different attribute names** for the same fact — derived attributes hide that
- This is fleet knowledge in the claim. The right long-term fix: drivers publish a switch / P2P-group attribute

where `cel_bus(x)` stands for the inline expression (CEL has no user functions, but `cel.bind` is available):

```
cel.bind(b, x.substring(5, 7),
  b >= "18" && b <= "1f" ? "sw0" :
  b >= "28" && b <= "2f" ? "sw1" :
  b >= "98" && b <= "9f" ? "sw2" :
  b >= "a8" && b <= "af" ? "sw3" : x)
```

Derived "P2P group": what the drivers don't publish (2/2)


```
- name: gpu
  exactly:
    deviceClassName: gpu.nvidia.com
    count: 2
    selectors:
    - cel: { expression: '"pciBusID" in device.attributes["resource.kubernetes.io"]' }
    derivedAttributes:
    - name: derived/pcieSwitch
      # pciBusID = "0000:1b:00.0" -> bus "1b"
      expression: >-
        cel_bus(device.attributes["resource.kubernetes.io"].pciBusID)
- name: nic
  exactly:
    deviceClassName: dranet
    selectors:
    - cel: { expression: '"pciAddress" in device.attributes["dra.net"]' }
    derivedAttributes:
    - name: derived/pcieSwitch
      expression: >-
        cel_bus(device.attributes["dra.net"].pciAddress)      # different source attribute
constraints:
- requests: [gpu, nic]
  matchAttribute: derived/pcieSwitch
```

What KubeVirt builds by default

- Machine type `q35`; KubeVirt never sets a guest PCI address on passthrough hostdevs
- libvirt puts every address-less PCIe endpoint behind **its own `pcie-root-port` on `pcie.0`**; KubeVirt then adds empty root ports for hotplug

```
host bridge 00:00.0 [8086:29c0]  (q35 MCH)
  ├─ 00:02.0 pcie-root-port [1b36:000c] ── 01:00.0  GPU    (vfio)
  ├─ 00:02.1 pcie-root-port [1b36:000c] ── 02:00.0  NIC VF (vfio)
  └─ 00:02.x ...                          empty hotplug root ports
```

KubeVirt option	What it does	Guest P2P verdict (Intel host)
default	one root port per device on `pcie.0`	**refused** — siblings, q35 not whitelisted
`PCINUMAAwareTopology` gate (alpha)	one `pxb-pcie` per host NUMA node, one root port per device	**refused** — still siblings; each pxb is its own non-whitelisted host bridge
annotation `kubevirt.io/placePCIDevicesOnRootComplex`	devices as integrated endpoints on bus 0	**refused** — no upstream bridge at all
`interfaces[].pciAddress` (SR-IOV only)	fixed guest address for the NIC	no bridge to share; GPUs have no such field

The guest kernel's decision table

Guest layout	Common upstream bridge?	Guest CPUID	Verdict
separate root ports on `pcie.0`	no	Intel	`NOT_SUPPORTED`
separate root ports on `pcie.0`	no	AMD family >= 0x17 (e.g. host-passthrough on Zen)	`THRU_HOST_BRIDGE` (allowed)
separate `pxb-pcie` buses	no	Intel	`NOT_SUPPORTED`
behind one virtual bridge / switch, no ACS redirect	yes	any	`BUS_ADDR` (allowed)
behind one virtual switch, guest vIOMMU sets ACS RR/CR	yes, redirected	Intel	`NOT_SUPPORTED` (unless guest `pci=disable_acs_redir=`)

- The guest's "yes" is about the **virtual** tree. Real TLPs still follow the **host** path:
  - host IOMMU on + host ACS on (needed for separate VFIO groups) → peer traffic goes up to the host root complex
  - devices on different host sockets → guest says "same bridge", host sends it over UPI
- So the **host-side DRA selection** still has to pick a good pair (derived socket / P2P group claims)

Status board and caveats

KubeCon CloudNativeCon North America 2026

KEP	Feature	Gate	1.36	1.37	1.38 (planned)
6072	standard `numaNode`	none	—	stable	stable
5491	list attributes	`DRAListTypeAttributes`	alpha	alpha	beta
6080	derived attributes	`DRADerivedAttributes`	—	alpha	beta
5517	node allocatable resources	`DRANodeAllocatableResources`	alpha	alpha	beta (off by default)
5941	shared consumable capacity	`DRASharedConsumableCapacity`	—	—	alpha
5304	device metadata to workloads	none (driver opt-in)	alpha	beta	beta

Caveats when you try it:

- Turning on list or derived attributes in kube-scheduler moves **all** DRA allocation to the experimental allocator
- Enable list attributes in drivers only when the control plane has the gate too (dranet docs: slice writes fail otherwise)
- Derived-attribute CEL errors fail the pod — guard with `in` / selectors
- KEP text ≠ implementation in places: before writing claims or drivers, check the API types (`k8s.io/api/resource/v1`), the kubernetes.io DRA docs, the integration tests (`test/integration/dra/`) and the drivers' demo specs

Low-level controls that follow the topology

KubeCon CloudNativeCon North America 2026

OCI field	Since	What it sets	NRI adjustment	Status
`linux.memoryPolicy` {`mode`, `nodes`, `flags`}	1.3.0	`set_mempolicy(2)` for the container process	not yet (nri-plugins injects an `mpolset` wrapper)	runc 1.5 implements
`process.scheduler` {`policy`, `nice`, `priority`, `flags`, `runtime`, `deadline`, `period`}	1.1.0	`sched_setattr(2)`	`SetLinuxScheduler` (NRI >= 0.11)	runc implements
`process.ioPriority` {`class`, `priority`}	1.1.0	`ioprio_set(2)` for the process group	`SetLinuxIOPriority` (NRI >= 0.10)	runc implements
`linux.netDevices`	1.3.0	move host netdevs into the container netns	—	runc implements
`linux.intelRdt.schemata`	1.3.0	L2/L3/MB allocation	—	runc implements

```
"linux": {
  "memoryPolicy": {
    "mode": "MPOL_INTERLEAVE",
    "nodes": "0-2",
    "flags": ["MPOL_F_STATIC_NODES"]
  }
}
```

A few faster cores: Priority Core Turbo on Xeon 6

- **Priority Core Turbo (PCT)** on Xeon 6 P-core CPUs (e.g. 6700P / 6900P): a small number of **high-priority cores** per power domain run at maximum turbo, the remaining low-priority cores are capped near base frequency
- The **hardware** (Intel Speed Select: SST-CP + SST-TF) enforces the frequency budget — no OS frequency juggling
- Helps accelerator workloads whose **host side is on the critical path**: input pipeline, tokenization, kernel launch, communication proxy threads (bottleneck 3: device ↔ CPU)
- nri-plugins balloons / topology-aware policies (v0.13+): CPU class with `pctPriority: high|low`; optionally publish free HP cores as a node extended resource so the scheduler places pods where HP cores are available
- **Status:** available in nri-plugins today (extended-resource publishing is experimental); **work in progress:** expose PCT cores through DRA, together with KEP-5517 node allocatable resources (1.38 beta, planned), so HP cores become a schedulable, accounted DRA resource. Needs PCT-capable CPUs and a privileged plugin (SST interface)

```
# balloons policy (excerpt)
cpuClasses:
# idle + normal CPUs: low priority, capped
- name: default
  pctPriority: low
  pctMinFreq: min
  pctMaxFreq: base
- name: hp-pct              # few CPUs at top turbo
  pctPriority: high
  pctMinFreq: base
  pctMaxFreq: turbo
  # cpuclass.balloons.nri.io/hp-pct
  publishExtendedResource: true
balloonTypes:
- name: hp-bln
  cpuClass: hp-pct
  maxCPUs: 8
  preferNewBalloons: true
# pod: annotation + HP CPUs == cpu request
#   balloon.balloons.resource-policy.nri.io: hp-bln
#   limits: { cpu: "2", cpuclass.balloons.nri.io/hp-pct:
"2" }
```

How to look at your own box


```
# PCIe tree with switches
lspci -tv
# ACS state on a switch downstream port / root port
lspci -vvv -s 0000:15:01.0 | grep -A2 'Access Control Services'
# Which host bridge / root bus a device sits on (== resource.kubernetes.io/pcieRoot)
readlink -f /sys/bus/pci/devices/0000:1b:00.0      # /sys/devices/pci0000:15/...
# What firmware claims about locality
cat /sys/bus/pci/devices/0000:1b:00.0/numa_node     # -1 => unknown
cat /sys/bus/pci/devices/0000:1b:00.0/local_cpulist # all CPUs if -1!
cat /sys/devices/system/node/node*/distance
# Kernel P2P verdicts (dmesg)
dmesg | grep -i 'peer-to-peer\|p2pdma\|disable_acs_redir'
```

Where numa_node of a PCI device comes from (x86)


```
ACPI PCI root (PNP0A08) ──> pci_acpi_root_get_node()          arch/x86/pci/acpi.c:451
      _PXM on root bridge (walk ACPI parents) ──> pxm_to_node()  drivers/acpi/numa/srat.c:676-700
      no _PXM ──> x86_pci_root_bus_node()  (only AMD K8..Fam15h registers)   bus_numa.c, amd_bus.c
      node offline ──> NUMA_NO_NODE
root bus sysdata.node ──> every device below it: set_dev_node()  drivers/pci/probe.c:2735
sysfs: numa_node (writable FW workaround, taints), local_cpulist (= all CPUs if -1)
```

- SRAT Generic Initiator entries can name a PCI BDF, but do **not** set the device's numa_node
- SNC/NPS change the result only through firmware SRAT/_PXM; the kernel has no SNC logic in the PCI path

Glossary

Term	Meaning
RC	Root complex: CPU-side PCIe controller (per I/O stack / host bridge)
Host bridge / root bus	Linux `pci_host_bridge`; `pciDDDD:BB` in sysfs; = `resource.kubernetes.io/pcieRoot`
Root port	Bridge on the root bus that leads to one slot or switch
ACS	Access Control Services: redirect/block peer traffic at ports
ATS / DT	Address Translation Services / Direct Translated P2P
BAR	Base Address Register: a device's MMIO window, the P2P target
SNC / NPS	Intel Sub-NUMA Clustering / AMD NUMA-nodes-per-socket
SLIT / SRAT / HMAT	ACPI tables: NUMA distances / affinity / memory performance
`_PXM`	ACPI proximity method → NUMA node of a device or root bridge
pxb-pcie	QEMU PCIe expander bridge: an extra virtual host bridge, usually per guest NUMA node