Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队

北京大学学生 Linux 俱乐部

Workshop 02.
AI Communication Stack

孔昊然

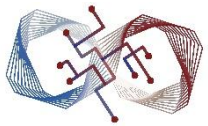
2026. 08. 23

Weiming Supercomputing Team
Linux Club of Peking University

Recall： Tensor Core session

回顾 Tensor Core 的发展，可以看到一个明显的趋势：随着矩阵计算吞吐不断提高，系统性能越来越不只取决于 Tensor Core 本身能够算多快，还取决于能否持续、及时地把操作数送到计算单元，并把结果送往下一阶段。即围绕它建立起来的 data movement + synchronization + pipeline。

Recall：Tensor Core session



数据搬运经历了明显的抽象升级：

Volta：thread-driven → 每个线程自己算地址， ld.global 读进寄存器、st.shared 写进 shmem。

Ampere：async copy → cp.async 把搬运从计算线程中解耦出来，可以形成 software pipeline。

Hopper：TMA → 一个线程发起一个描述符驱动的 tile 级搬运，进一步把地址生成、步长、边界处理等责任交给专门硬件单元在后台完成。（cp.async 时线程还得自己循环、自己生成地址。）

Blackwell：TMA + TMEM → 数据搬运和 tensor computation 的边界进一步被重新组织。

数据搬运正在从 programmer-managed 逐渐走向 hardware-managed。

Tensor Core 自身也经历了类似的同步到异步执行的变化：

Volta的 WMMA → warp 内线程同步执行矩阵乘，结果在寄存器 fragment 里。

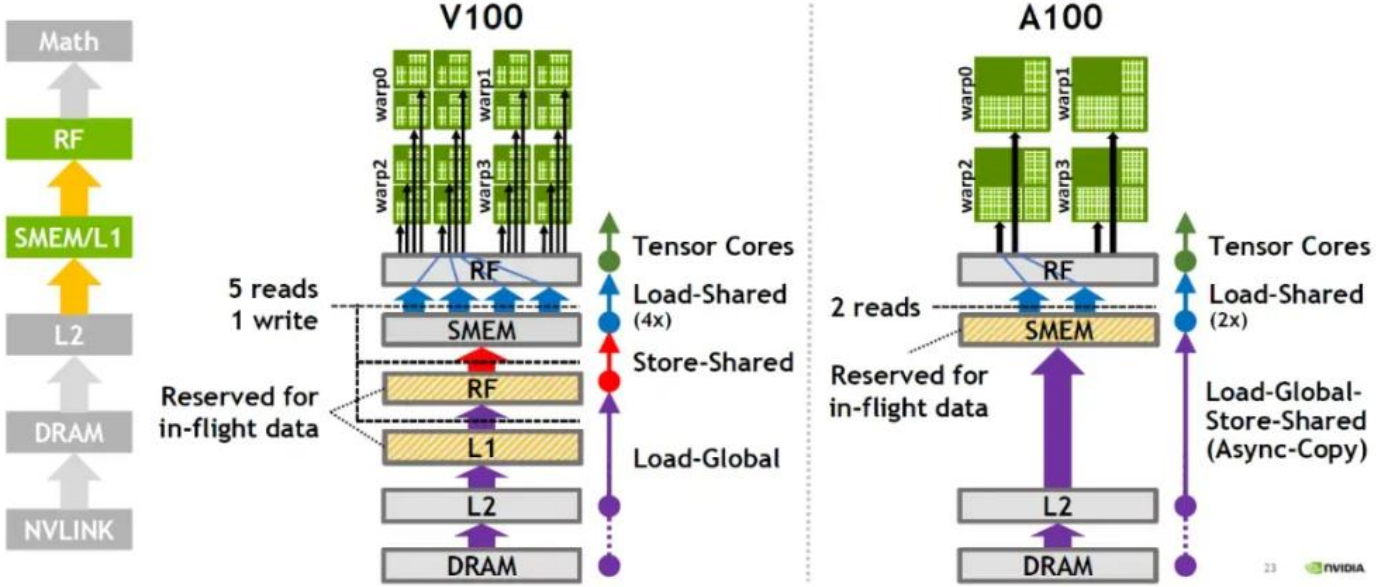
Ampere 的 MMA → 更细粒度的 shape / layout / data movement 控制。

Hopper 的WGMMA → tc从一个被线程调用的计算单元变成了一个异步的计算代理。它不需要等整个 warpgroup 都准备好，可以提前发射然后显式 commit_group / wait_group，能直接从shmem中读输入。

tcgen05 → accumulator 的驻留位置从寄存器扩展到 TMEM，是独立于寄存器的新存储层。

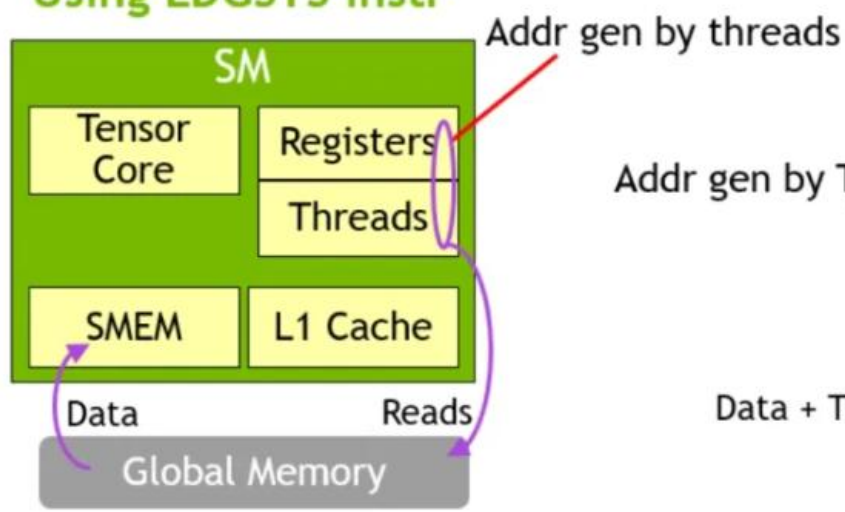
A100 SM DATA MOVEMENT EFFICIENCY

3x SMEM/L1 bandwidth, 2x in-flight capacity



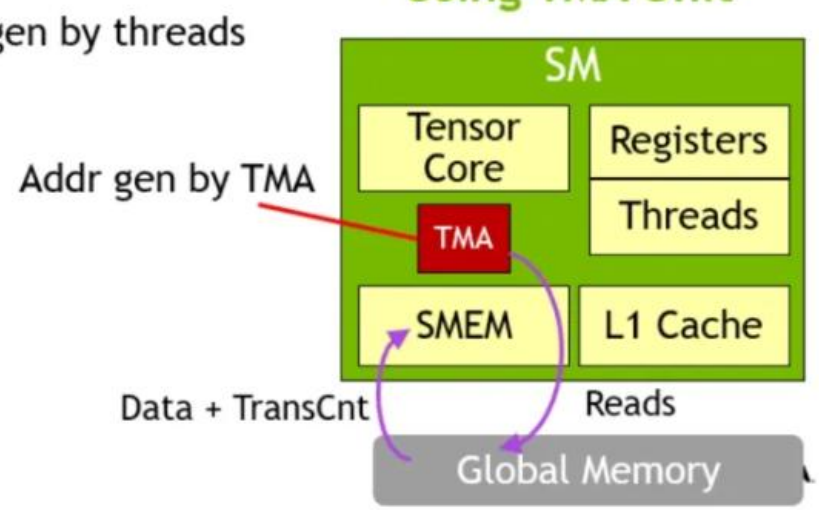
A100

Using LDGSTS instr

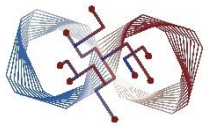


H100

Using TMA Unit



Recall：Tensor Core session



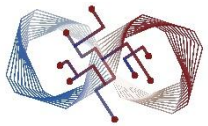
Data Movement 和 Computation 的逐渐异步化，使传统“发出一条指令 → 等待结果 → 执行下一条指令”的同步执行模型已经不足以描述新的执行方式。

例如，TMA 可以异步发起一个 Global Memory → Shared Memory 的后台 Tile 搬运；程序可以同时继续进行其他计算。这就产生了一个新的依赖问题：Consumer 什么时候可以安全地读取这块 Shared Memory？关键不再只是生产者线程是否执行到了某个位置，而是对应的异步操作是否已经完成，以及生产的数据是否已经满足 Consumer 的访问和可见性条件。

因此，同步语义也从 Thread / Control-flow Synchronization 转向 Data / Event Synchronization。传统的 __syncthreads() 主要协调线程之间的控制流并建立相应的内存同步语义； Hopper 引入的 mbarrier 等机制则可以进一步将线程到达与异步事务完成关联起来，使 Consumer 能够等待特定的数据生产事件。

TMA、WGMMA、tcgen05 等异步操作由 async proxy 执行，而普通 Load/Store 属于 generic proxy；不同 proxy 之间并不会自动建立我们所需要的完整 ordering，因此必须通过显式的同步和等待机制建立依赖。在此基础上，mbarrier 的 transaction count 以及 Blackwell 中的 tcgen05.wait 等机制可以进一步表达和等待特定异步 Memory / Tensor 操作达到所需的完成状态。

Recall：Tensor Core session



对矩阵密集型 workload，瓶颈正在从单纯的计算吞吐扩展为一整条数据供应链：数据是否已经准备好，搬运能否与计算重叠，producer 和 consumer 之间的依赖能否被细粒度表达，以及数据搬运、同步和计算是否会争用相同的硬件资源。

GPU 的同步机制同步的对象不再只是“线程是否到达”而越来越是“哪个异步操作完成了、哪些数据已经 ready、Consumer 什么时候可以安全地继续执行”。同步机制开始承担描述、传播和维护数据流依赖的职责。专用的数据搬运硬件、异步编程模型、细粒度 barrier 和 completion mechanism，本质上都在解决同一个问题：如何让正确的数据在正确的时间到达正确的计算单元，并让计算持续推进？

我们也已经讨论了 NVIDIA GPU 如何通过 Pipeline 组织数据搬运、矩阵计算和阶段依赖使 Tensor Core 尽可能保持 busy。不同的 Pipeline Ordering 不只是代码排列方式不同，它们会改变 producer 和 consumer 的先后关系、buffer 的生命周期、同步发生的位置，以及计算、搬运和同步资源之间的重叠方式。

Recall：Pipeline Ordering session

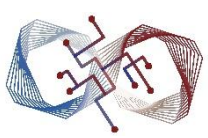
Where Does Uncertainty Live?

model	main orchestration	principal trade-off
GPU pipeline	dynamic readiness + explicit waits	buffering / occupancy
TPU systolic array	spatial wave through MAC cells	shape rigidity / fill-drain
Graphcore IPU	placement + BSP exchange phases	mapping / global imbalance
Groq TSP	functional slices + static stream schedule	compiler burden / less dynamism

因此Pipeline 背后真正的问题是：**谁来决定什么工作在什么时候、在哪个资源上执行？谁负责维护依赖、推进执行，并判断一个阶段何时真正完成?**

这并非简单的硬件调度 or 软件调度二选一，其中涉及到一组连续的**设计trade off**：硬件与软件的责任划分、编程复杂度与硬件复杂度、静态与动态调度、显式控制与自动管理、灵活性与可预测性，以及细粒度重叠带来的性能收益与状态和同步开销。

Communication 引入



在单 GPU 内部，这些责任可以由 Compiler、Warp Scheduler、Scoreboard、Pipeline、Barrier、Copy Engine 和 Memory Hierarchy 共同承担。我们关注数据如何在 Register、Shared Mem、片上专用存储和 HBM 之间流动并保证计算正确推进。

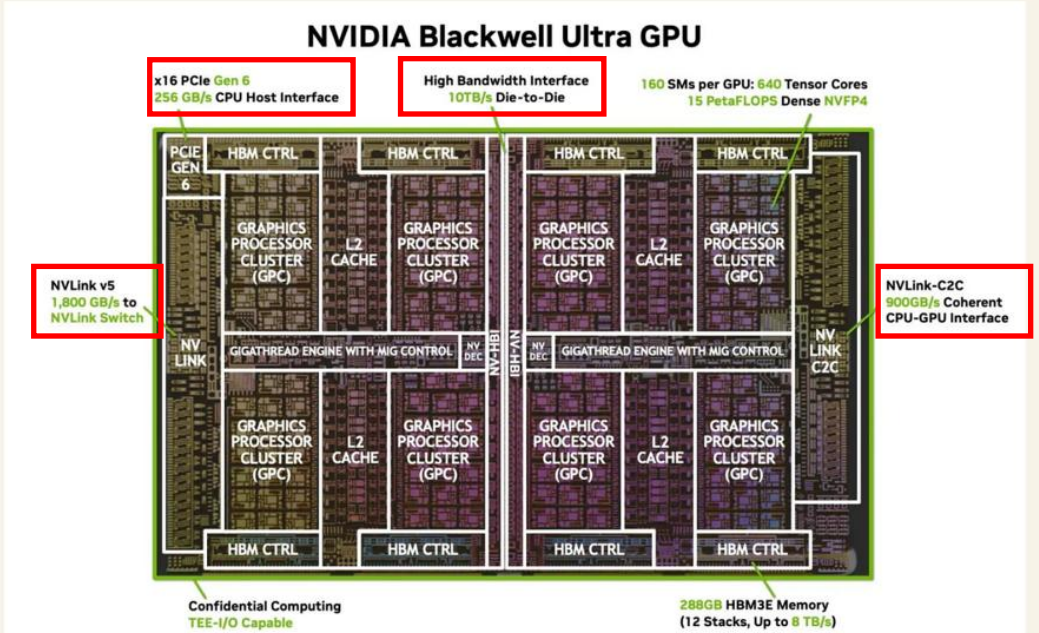
今天，我们把同一个问题扩展到 GPU-to-GPU 高速互联。 此时数据的 Producer 和 Consumer 不再位于同一块 GPU 上，数据可能需要跨越 GPU Fabric、NVLink / NVSwitch、PCIe、RNIC、RDMA Transport 和交换网络。原本局限在单 GPU 内部的 Readiness、Ordering、Completion 和 Visibility，现在必须跨越不同的设备、地址域、队列、协议乃至故障域继续成立。

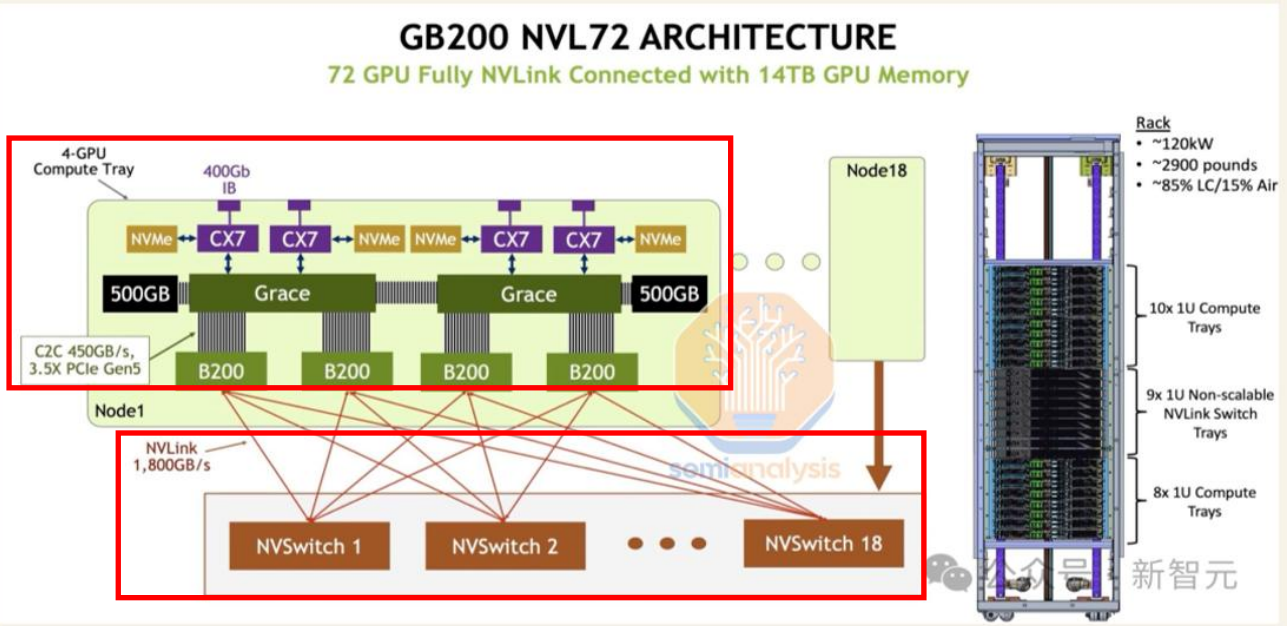
这时一个发送操作实际上包含两个问题：

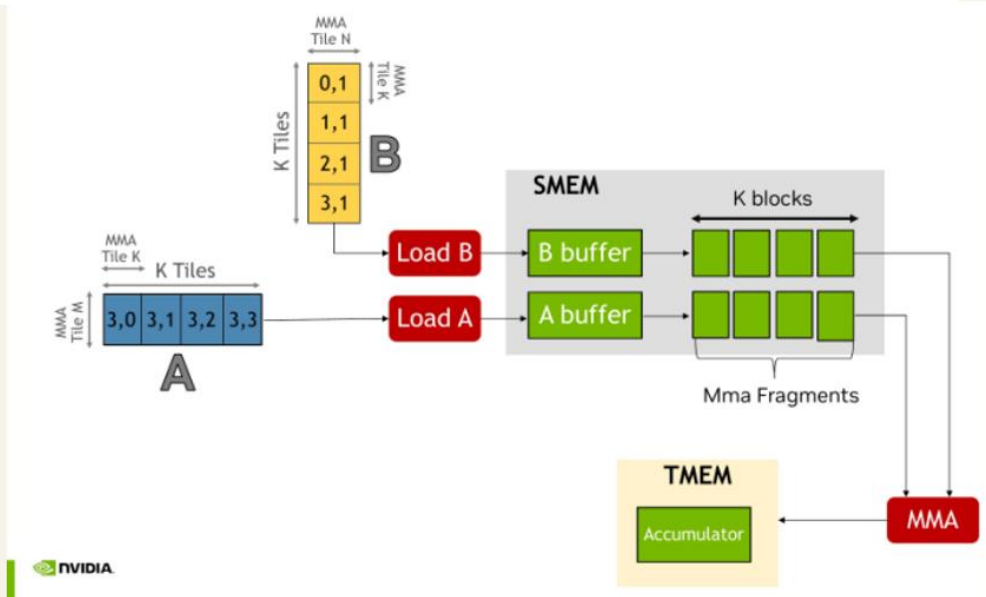
一块 Tensor Chunk 从 GPU A 发往 GPU B 时，实际经过了什么路径?

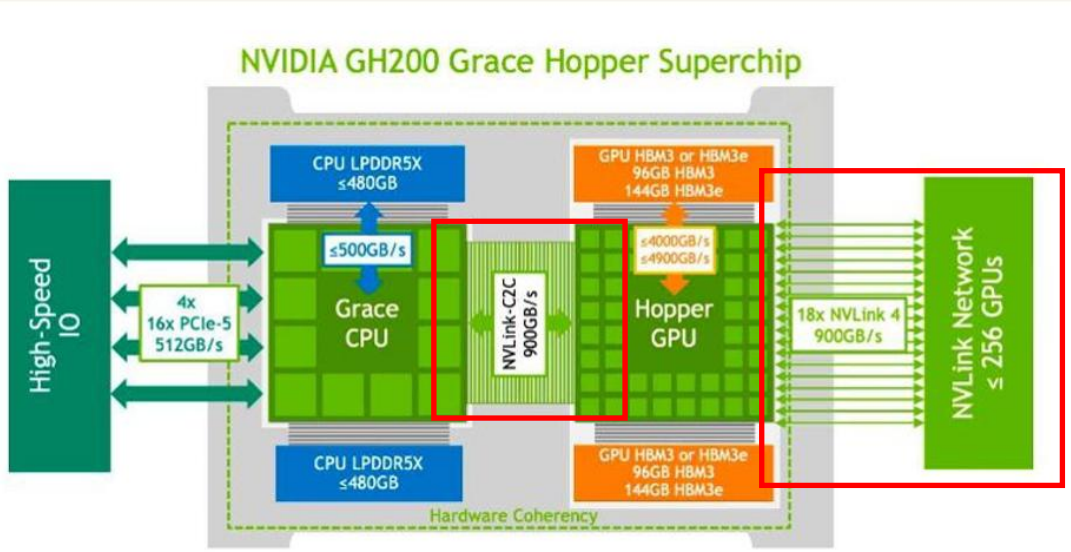
GPU B 如何判断这块数据现在已经可以被 consumer kernel 正确地使用?

Recall： Networking System session

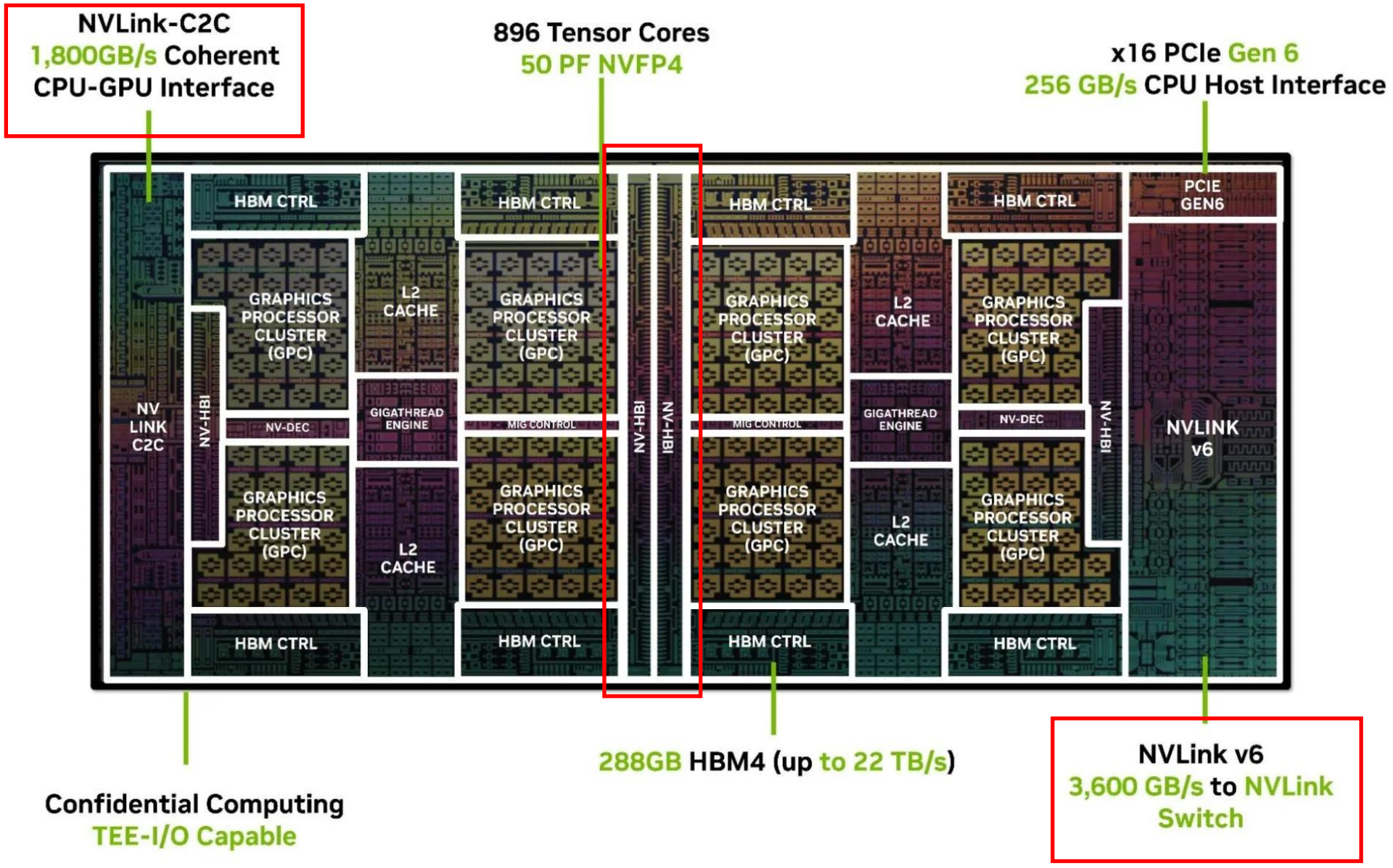








What's more in Rubin



Rubin GPUs contain up to 224 SMs and 288GB HBM4 Memory. Available SM count and HBM varies by SKU.

What's more in Rubin

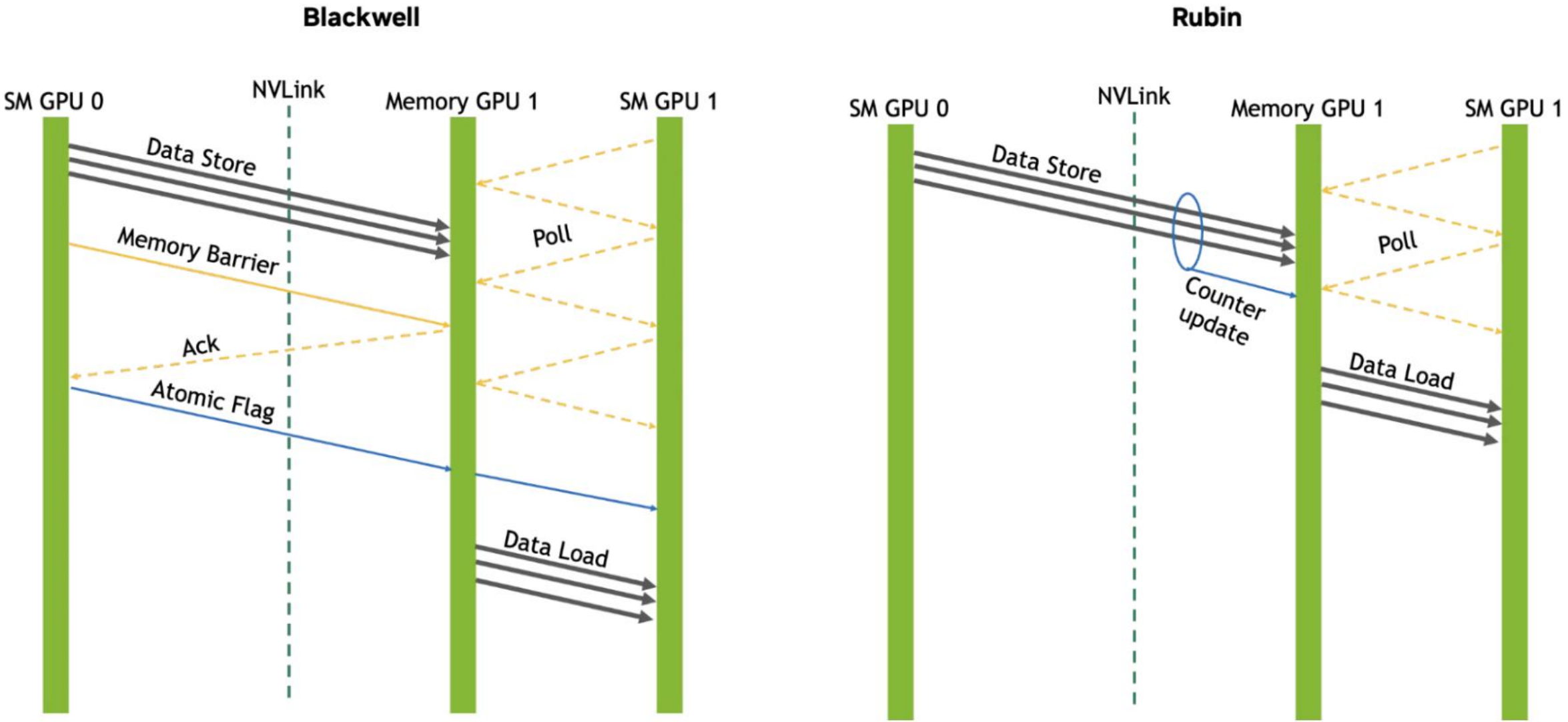
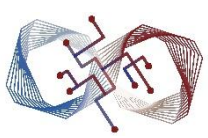


Figure 7. Rubin accelerates NVLink communications using counted writes

核心问题



数据到达某个端点并不等于它已经可以被消费。至少还需要区分：

Delivery：必要的 packet 或 fragment 是否已经到达?

Placement：数据是否被写入正确的目标 buffer 和 offset?

Completion：本层定义的传输或操作义务是否已经完成?

Visibility：目标GPU的 memory system 是否已经能够观察到这些写入?

Ordering：payload、completion signal、fence 和 consumer launch 之间是否满足必要的先后关系?

这些条件是我们判断GPU B 是否获得消费资格时必须逐项检查的语义条件。

需要注意： Sender-side completion 不等于 remote consumability。本地 CQE、DMA 完成或 packet 到达，都不意味着 GPU B 的 consumer kernel 已经可以观察并使用目标数据。

以 GPU A 向 GPU B 做 RDMA Write 为例：A 写完 source buffer、A 向 RNIC 提交 WQE、发送方 CQE、transport ACK、B 的 RNIC 把 bytes 写入目标 buffer、runtime 发布 ready signal / counter、B 的 consumer 执行 acquire/wait、B 安全读取并计算。

简介 OVERVIEW

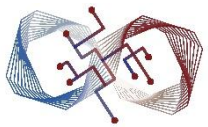
为了建立这些条件，系统需要维护 sequence、offset、buffer ownership、credit、ACK、reorder、completion、memory ordering、timeout 和 recovery 等状态。

通信栈会向上层隐藏其中的大量工作，但隐藏不等于消除：这些状态仍然需要占用 GPU、CPU、NIC、交换机、HBM、片上 SRAM、队列和执行时间。

因此今天要讨论的是：这些状态和控制责任应该放在哪里？

它们可以分布在 GPU kernel、通信库、host runtime、transport、NIC、SmartNIC/DPU、交换机和集群控制面中。不同系统决定：谁发现数据缺失、乱序或拥塞；谁持续推进通信；谁发布 completion；谁保证 consumer visibility；谁在局部故障后重传或绕路；谁在无法局部恢复时向上层报告错误并重建执行。

简介 OVERVIEW



责任放得越靠近硬件，通常越容易获得较低延迟、较快反应和较小的软件抖动，但可能增加芯片状态、协议耦合和演进成本。

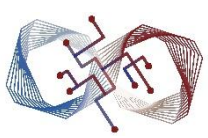
责任放在软件中，通常更灵活、更容易适配 workload，但会增加 CPU/GPU 开销、调度延迟和可见信号的限制。

本次 workshop 希望沿着同一个 Tensor Chunk，追踪上层依赖如何被逐层转换成可完成、可恢复、可观测的跨设备执行。

当一个 Tensor Chunk 从 GPU A 发往 GPU B 时，它实际经过哪些物理与协议边界？GPU B 在什么可验证的条件下，才真正获得消费它的资格？

为了让这个 Chunk 在依赖满足后，以正确的顺序写入正确的位置并对消费 kernel 可见，系统需要维护哪些状态、执行哪些控制协议？这些责任应当分配给 GPU、通信库、Runtime、Transport、NIC、交换机还是控制面？不同的责任划分又会带来怎样的性能、复杂度、灵活性、可预测性和可扩展性取舍？系统设计者必须在哪些边界上做出取舍？

总目录



我们接下来将分六个部分展开我对这些问题的理解与思考：

1. 什么才叫可消费？

   Delivery、Placement、Completion、Visibility、Ordering 分别证明什么？

2. 数据跨过哪些边界？

   HBM、GPU fabric、PCIe、NIC、RDMA 与交换机之间发生了什么？

3. 状态与控制由谁维护？

   可靠性、顺序、拥塞、progress 和故障状态在哪些层维护？

4. Workload 如何把流量变成等待？

   Ring、All-to-All、MoE、P2P/KV 与 distributed kernel 各自让谁等待谁？

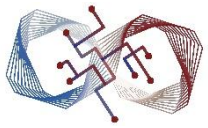
5. 哪些开销会进入 Critical Path？

   启动/重启、稳态执行、故障检测恢复为何在大规模下不可忽略？

6. 如果重新划分边界会怎样？

   改变 compute、memory 与 interconnect 的落点后，责任如何重新划分？

本次workshop介绍的通信层次划分



workload dependency

↓ framework graph / parallel strategy

collective, P2P, EP or object API

↓ algorithm / channel / chunk schedule

CPU or GPU execution and progress

↓ memory readiness / staging / registration / DMA

transport connection / congestion / reliability

↓ packetization / path selection / switch queues

link serialization / FEC / replay / credit

↓ destination placement / completion / memory ordering

consumer synchronization and recovery

话题 1、2

1、什么才叫可消费?

Delivery、Placement、Completion、Visibility、Ordering

数据到达端点、写入目标地址、本层操作完成、对 GPU consumer 可见，以及满足执行顺序，并不是同一个事件。这里定义了后续所有讨论的终点：可消费究竟需要哪些证据。

2、数据究竟跨过了哪些边界?

一条可能的路径：HBM_A → GPU memory system / local fabric → PCIe / C2C → RNIC DMA → RDMA transport over switch fabric → remote RNIC / fabric → remote GPU fabric / PCIe / C2C → HBM_B

检查每跨过一个边界后，地址、ownership、队列、完成语义、可见性和故障模型如何变化。需要注意的是：Scale-up 路径可能完全不经过 RNIC，使用 host staging 时路径中还会出现 DRAM，payload、control、completion、recovery 不一定走同一条路径，Delivery 不等于目标 GPU 已经完成写入，Placement 不等于 consumer 已经可见。数据路径上存在多个硬件和协议边界，通信栈对我们隐藏了什么?

话题 3、4

3. 状态与控制究竟由谁维护?

可靠性、顺序、拥塞控制、progress、completion 和故障恢复分别由哪些层协作完成。

我们会比较 RC、MRC、Falcon、UCCL-Tran、UET 等不同设计所选择的状态位置与控制闭环。

核心是：谁能观察到问题，谁能及时采取行动，谁承担状态和恢复成本?

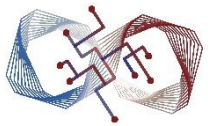
4. Workload 如何把流量变成等待?

不同 Workload 不只产生不同的网络流量，还会塑造不同的依赖关系和等待关系。

我们会分析并比较几类通信场景和执行组织方式：P2P、collective、MoE routing、KV/object movement，以及 distributed kernel。理解它们分别产生什么样的 traffic pattern、通信粒度和同步关系。然后进一步问：谁负责推进通信？Contention 出现在哪里？通信能否与计算重叠？一个局部 Stall 又如何传播到整个 Collective?

核心因果链：Workload dependency → traffic pattern → topology / queue / flow control → progress and synchronization → wait-for graph → exposed critical path

话题5



5. 哪些开销会在 Scaling 下进入通信生命周期的 Critical Path?

并非所有开销都会进入关键路径。真正决定端到端时间的是那些无法被 overlap 隐藏、无法被充分摊销并最终阻塞下一条必要依赖的开销。

我们从三个时间尺度来看：1、启动 / 重启：Bootstrap、Communicator、QP 创建与资源初始化。2、稳态执行：HBM footprint、RTT / BDP、Queue、Progress 与 Contention。3、故障检测与恢复：Retry、Reroute、Diagnosis、Communicator Rebuild 与 Restart

我们也会通过一些案例分析场景如何改变通信的 Critical Path 并对通信系统提出新要求。

OpenAI MRC体现大规模训练场景下对 data-plane / path resilience 的要求。Meta 的 100K+ GPU 系列工作进一步说明：随着scaling，原本隐藏在通信路径之外，可以忽略或摊销的成本（初始化、HBM / QP 状态管理、BDP 流控、故障诊断和可观测性等等）会成为通信生命周期中的重要成本，甚至成为系统 Critical Path 的一部分。（Collective Communication for 100k+ GPUs 与 SIGCOMM26: Connecting 100K+ GPUs）

话题6

6. 如果重新划分计算、内存和互联边界会怎样？（放飞自我环节）

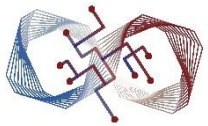
如果我们不再把边界视为固定条件，而是把边界本身作为设计变量，会发生什么？

通过 Cerebras、Groq、HBF、PNM、3D Memory-Logic Stacking 和 NetDAM 等 Case Study，我们进一步反过来审视前面的问题：原本由通信栈承担的工作，究竟被真正消除了，还是被转移到了 Compiler、片上存储、I/O Die、Memory Tier 或 Network Appliance?

Cerebras把部分 chip-to-chip movement 变成 wafer-internal movement，但容量和 off-wafer I/O 成为新约束；Groq：把部分 ordering/progress 决策前移到 compiler/static schedule，但不等于所有动态控制都消失；HBF：研究方向，带来页延迟、写寿命和写放大问题；PNM/3D stacking：缩短 memory-compute 距离，但引入 thermal 和 mapping/interface coupling；NetDAM：研究原型，仍需要 ownership、ordering、completion、权限和故障恢复。

对每一种架构，我们重点检查：哪些数据搬运和同步机制被消除了？哪些等待路径被缩短或消失？哪些状态和控制责任被转移到了新的边界？新的边界又引入了哪些容量、带宽、协议、编程和故障恢复约束？

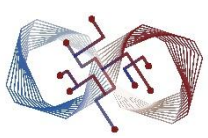
小结



通信栈的核心工作是：把 producer→consumer 依赖转化为可验证、可推进、可恢复的 readiness contract。数据位于正确的位置，满足必要的顺序和可见性条件且 consumer 能够安全地使用它。不同协议、Runtime 和硬件架构的根本差异，在于这份 contract 的状态与控制被放在哪个边界，以及系统为此承担了什么成本。

有了这些insight之后，我们开始Dive into 具体的细节。

一次调用下藏着的工作



0 workload dependency 为什么 B 需要这批数据

1 framework / strategy rank、group、并行维度

2 collective / P2P API 语义：reduce、send、get、put

3 algorithm / schedule ring、tree、chunk、channel

4 kernel / progress 谁 post、谁 poll、谁发 signal

5 source memory readiness、layout、staging

6 device API / DMA registration、WQE、doorbell、DMA

7 transport ordering、CC、ACK、retry

8 topology / path NIC、switch、ECMP、多平面

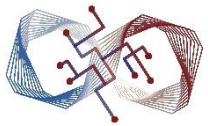
9 queue / serialization credit、排队、线速发送

10 link protection FEC、CRC、replay、flow control

11 destination placement address、offset、reorder、commit

12 consumer / recovery fence、timeout、重试、上层恢复

通信延迟



T_visible =

T_source_ready　　等生产者写完/布局准备

+ T_launch_progress　launch、post、poll、doorbell

+ T_stage_copy　　staging、pack、registration、DMA setup

+ T_queue　　endpoint / NIC / switch 排队

+ T_serialize　　bytes ÷ 有效链路速率

+ T_propagate　　线缆、交换跳数、PHY/FEC

+ T_place　　乱序重组、目标写入、cache/memory commit

+ T_complete　　ACK、counter、CQE、fence

+ T_consumer_wait　　消费者真正能观察到数据

+ T_recovery　　丢包、重放、超时、重路由

- T_hidden_overlap　　被计算或其他 chunk 隐藏的部分

通信延迟

这个通信延迟的公式不是精确的排队模型，主要是排查顺序。我们可以对一次通信逐段做时间戳：

producer event→WQE post 是 launch/progress；

post→first packet/DMA 是 source/NIC；

first→last byte 是 queue+wire；

last byte→ready signal 是 placement/completion；

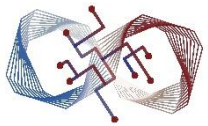
ready→consumer start 是 scheduling/ordering。

空载 ping 很短不能证明高负载时短，因为 T_queue 和 T_recovery 只在压力下出现。

另外 AllReduce、MoE dispatch 和 pipeline stage 等都含同步依赖。即使平均完成时间短，只要下一个阶段必须等 8 个 rank，step 仍由最慢 rank 决定。慢点可能来自一条 hash collision、一个热点 expert、一次重传、CPU progress gap 或目标 HBM争用，因此平均网络带宽可能看起来很好。

这里要区分平均/P99/P999、最大 rank 时间和 barrier wait。诊断顺序是先在 per-rank timeline 找第一个偏离者，再沿它的 producer、path、queue、retry、completion 查因果；同时看其他 rank 的 idle/barrier gap。优化平均值但不缩短最大依赖不会改善性能。

话题 1 什么叫可消费？



假设 GPU A 在 HBM 中产生了一个 32 KiB chunk，GPU B 的下一个 tile 依赖它。

必要的 bytes/fragment 是否到达规定的接收端（Delivery）；

是否写入正确的目标地址、offset 和 generation（Placement）；

哪一项传输或发布义务已经结束（Completion）；

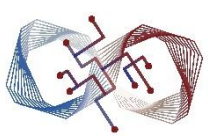
B 的 memory system 和 consumer 是否能在所需 scope 下观察到写入（Visibility）；

payload、signal、fence 和 consumer launch 之间需要的 happens-before 是否成立（Ordering）。

（它们是分析可消费资格的证据类别，不是所有协议都严格串行执行的五个wire stage）

这里要区分 sender-side completion 与 remote consumability：本地 CQE 可能只允许 A 复用 source buffer，不必然证明 B 的 kernel 已经 acquire 到目标数据。另外，Buffer ownership/lifetime 是横向的安全前提：即使五类证据已经满足，如果 A 已经过早复用 source buffer，或者 B 的 target slot 已经被下一轮覆盖，B 仍不能安全消费。

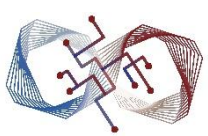
话题 1 什么叫可消费？



我们把 payload、控制信息和完成通知分开：payload 负责搬 bytes，控制路径负责序号、credit、ACK、counter、fence 或 error。任何一条路径停住，B 都可能表现为算力没有跑满。这里的三类角色不是三条必然独立的物理路径：控制信息可能与 payload 一起传输，completion 也可能 piggyback 在控制消息上或者由同一个 endpoint 直接产生。

需要分别追问的是：bytes 是否搬到位，控制/metadata 是否使它能被正确解释和推进，以及相应层次的 completion/publication 是否已经发布。对 B 的 readiness contract 来说，只要其中一类必要工作没有完成，consumer 就可能继续等待，表现为 kernel 未启动、polling 或 pipeline bubble；与 B 无关的 source-side ACK 延迟则不一定阻塞 B。

1.1 Delivery



Delivery 问的是，必要的 packet / fragment 是否到达协议定义的接收端点？需要识别：

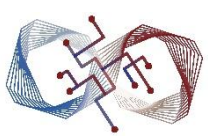
- message / chunk identity
- sequence / offset / length
- integrity 与 duplicate handling
- gap / loss 是否已被处理（若协议允许乱序或重传）

到达接收端 ≠ 已写入目标 buffer ≠ consumer 已经可以读。

到达的观察点可以是 remote RNIC 收到并验证了 fragment，也可以是 scale-up endpoint 接收了一个 transaction；不能在没有协议定义的情况下把它直接说成到达 GPU HBM。

Delivery 需要 message identity、offset、length、完整性和 gap/duplicate 状态，接收端可能允许乱序到达但必须能判断哪些片段仍缺失。可靠传输中的 ACK、receive completion 或后续 transport state 可以证明某一层的 delivery obligation；它们不自动证明 payload 已经完成目标写入 or B 的 kernel 已经 acquire 到数据。这个区分会贯穿后面 RC、MRC、Falcon、UCCL 和 UET 的比较。

1.2 Placement



Placement 问的是，bytes 是否已经写入正确的address/offset/generation？必须成立：

- address translation / protection
- range 与 length 检查
- fragment coverage / hole tracking（若发生分片或乱序）
- destination buffer ownership

在协议允许乱序时，direct placement 可以允许乱序写入，但不能允许错误位置或错误 generation。

Placement 目标可能是远端 HBM、host staging buffer、片上 scratchpad 或一个由 runtime 管理的 slot；必须明确最终 consumer 读取的地址域。在支持分片乱序放置的协议中，可以按 fragment offset 直接写入，不要求 payload 按网络到达顺序到齐；此时接收端还要维护 hole/bitmap、长度和 generation，避免迟到的上一轮数据污染当前 buffer。顺序传输则可能不需要这些额外的 reorder state。

DMA 已经写入一个目标地址，不等于整个 logical chunk 已经完整放置；一个 message 也可能经过 staging、reorder 或分片提交。证据应不只看 sender-side CQE，而包括目标写入范围、fragment completion、reorder state 和 buffer lifetime。

1.3 Completion

Completion 不是一个全局事件。可能分别表示：

sender：source buffer 可以复用

transport：delivery / retry 义务结束

endpoint：本层 DMA / write obligation 已结束

runtime：ready signal / counter 已发布

consumer：已满足 acquire 条件并开始使用（readiness endpoint）

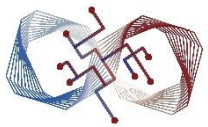
每个 CQE / ACK / counter / event都必须问：它覆盖哪一层?

发送方 CQE 可能只表示本地 WQE 已由 RNIC 处理，足以解除 source-data obligation；

transport ACK 可能只表示对端接受了协议层义务；

remote write completion、runtime signal 和 consumer acquire 是不同层次。具体 API 的语义必须以它的规范和 memory-ordering 定义为准。还要区分 definite success、definite rejection 和 unknown failure。超时只是本地观察到等待超限，不自动证明远端没有执行；如果 buffer 要复用或重试，必须知道当前 completion/ownership 到底覆盖了什么。

1.4 Visibility



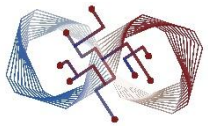
Visibility 问的是 consumer 的 memory access 能否在规定 scope 下观察到 payload?

DMA write ≠ kernel-visible write， signal observed ≠ payload visible （除非定义了所需的 memory ordering）

Visibility 是相对于 consumer 和 memory scope 定义的。不同平台可能提供硬件一致性、显式 fence、event/stream dependency、acquire/release 或其他 scope。关键是要有协议或 memory model 规定：目标写入何时进入 consumer 可观察的域。

常见错误是把 flag 当成 payload ready 的充分证据：producer 先写 payload，再更新 flag；只有当 flag 的发布带有适用的 release/ordering 语义，并且 consumer 以匹配的 acquire 或同步方式读取时，观察到 flag 才能作为读取 payload 的条件。否则，consumer 可能已经看到 flag，却仍不能据此证明 payload 已在所需 memory scope 内可见。同样，payload 已进入目标 memory system，通常只证明某一层的 placement 或 DMA/write completion；不自动证明 consumer 已观察到 payload 或者 已执行所需的 acquire/fence 并获得执行资格。验证时应分别寻找 payload visibility 的 memory-model 证据、consumer-side acquire、event/fence、完成记录以及可重复的 protocol trace。

1.5 Ordering



Ordering 是必要的先后关系必须成立：

payload writes → publish signal，phase n → phase n+1，metadata → payload interpretation

producer ready → transfer launch，completion → consumer launch

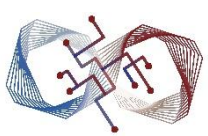
sequence / phase / fence / barrier / acquire 共同表达这些 happens-before 约束。

Ordering 必须说明“在哪个域、对谁成立”：

多路径或 direct placement 可以允许 packet 乱序到达；只要 message identity、offset、长度、sequence/bitmap 和 commit rule 能让接收端识别所需 fragment 是否齐全，接收端就可以把 bytes 放到正确位置，并按 logical order 发布相应的 completion/publication。这里解决的是 fragment 的识别、覆盖和提交顺序，不等于 consumer 已经获得读取资格。

即使 packet 按序到达且目标端已经报告 placement 或某一层的 completion，也不自动意味着 payload 已进入 consumer 所需的可见性范围，更不意味着 consumer 已执行所需的 acquire/fence。

1.5 Ordering



应分别检查：

arrival order　　packet/fragment 何时到达 endpoint

placement order　　bytes 何时写入 address / offset / generation

completion order　　哪一层的 transfer/publication 义务何时结束

visibility　　payload 何时对 consumer 在所需 memory scope 下可见

execution order　　consumer 何时完成 acquire/fence 并获得执行资格

一个协议可能只保证其中一部分，其余条件由 endpoint、runtime、memory model 或 consumer kernel 补齐。

注意防止 stale signal：buffer 复用后，上一轮迟到的 completion、flag 或 counter 不能被解释成当前 generation 的 ready；发布条件必须带有可验证的 generation/epoch/phase，或采用等价的复用保护机制。（我们在话题 3中展开对状态归属问题的比较）

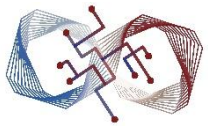
1.6 小结

Delivery	规定接收端看到了所需 fragment
Placement	bytes 位于正确 address / offset / generation
Completion	指定层的 transfer / publication 义务结束
Visibility	consumer 在所需 scope 下可观察 payload
Ordering	payload、signal、phase、launch 的先后约束成立

它们可以被同一个 counter、fence 或 completion record 一次性发布，也可以由多个层次分别维护。Buffer ownership/lifetime 是横向安全条件。

一个协议可能只保证其中一部分，其余条件由 endpoint、runtime、memory model 或 consumer kernel 补齐。还要防止 stale signal：buffer 复用后，上一轮迟到的 completion、flag 或 counter 不能被解释成当前 generation 的 ready；发布条件必须带有可验证的 generation/epoch/phase 或采用等价的复用保护机制。

2. 数据跨过了哪些边界



在讨论数据跨过哪些边界之前，我们首先需要认识承载这些通信的服务器拓扑。因为同一个 send、all_reduce 或 remote write，在不同系统中走过的物理路径可能完全不同。

在传统 PCIe GPU 服务器中，GPU 与 RNIC 可能连接在同一个 PCIe switch 下，也可能需要跨 PCIe switch、CPU Root Complex 甚至 NUMA domain；这些差异会直接影响 GPU Direct RDMA 的带宽、延迟和资源竞争。

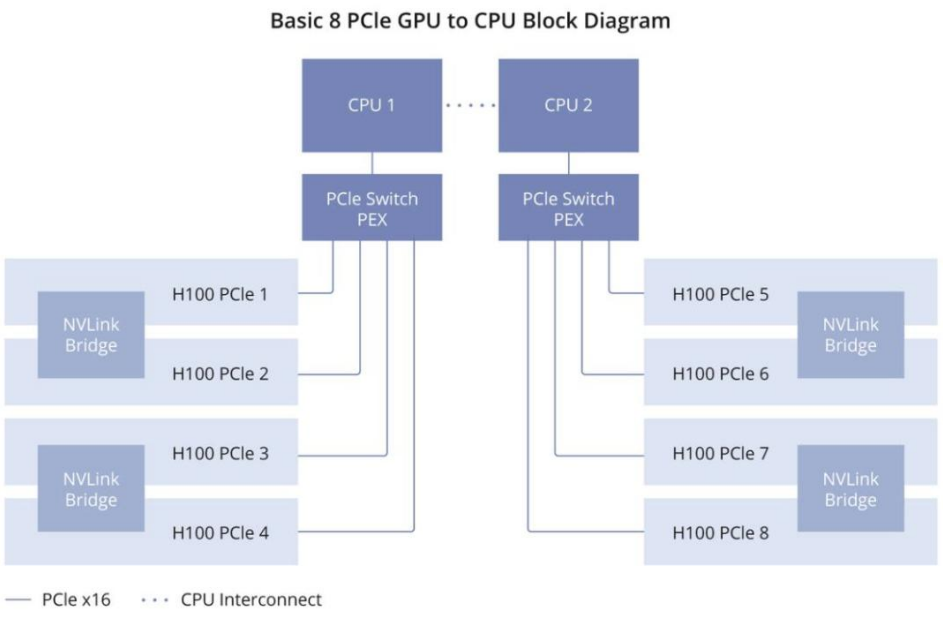
在 DGX/HGX 系统中，同一 NVLink domain 内的 GPU 可以通过 NVLink/NVSwitch 通信，而跨节点数据通常仍需进入 PCIe/RNIC 和 scale-out network。

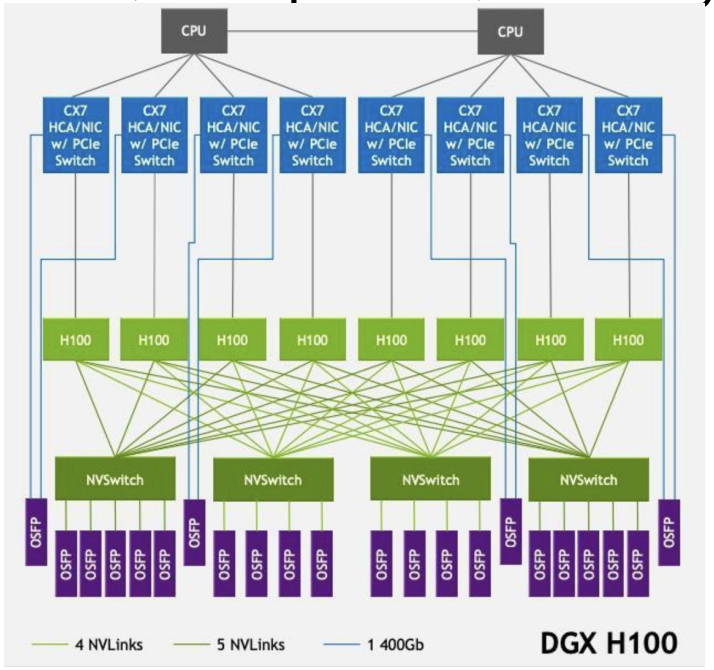
到了 GH200、GB200 等Grace Hopper/Grace Blackwell 系统，CPU 与 GPU 之间又增加了 NVLink-C2C，改变了传统 PCIe host-device boundary；在 NVL系统中，NVLink/NVSwitch 还会进一步扩展 rack-scale GPU fabric。

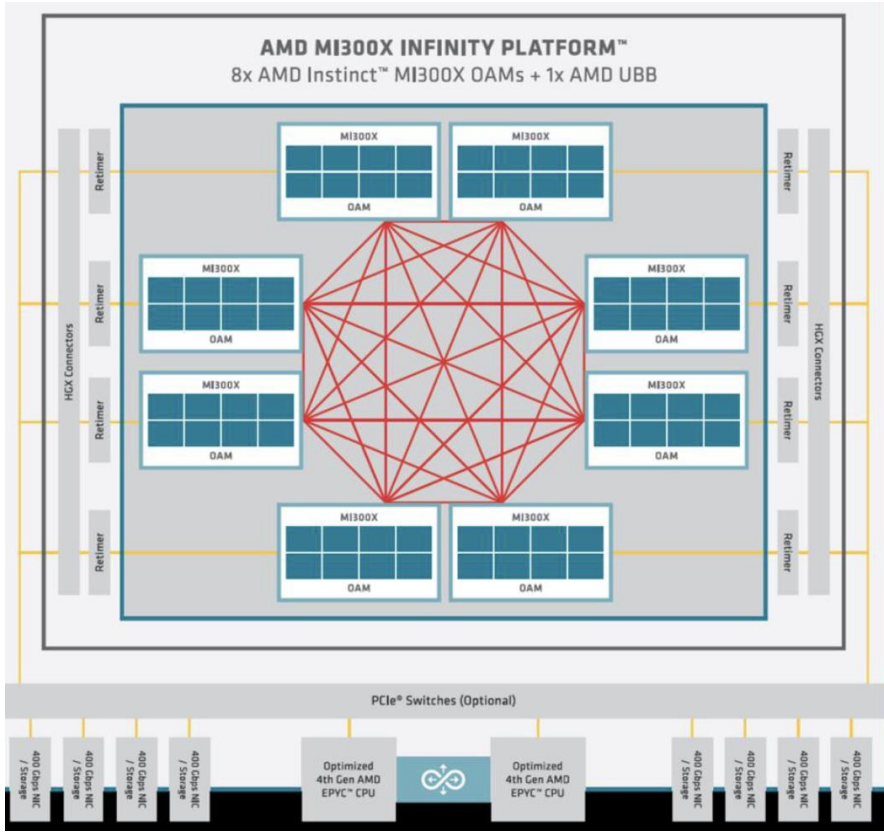
Vera Rubin 则继续沿着 C2C、scale-up fabric 与 scale-out network 分层演进，但具体数据路径仍取决于实际产品配置和拓扑。

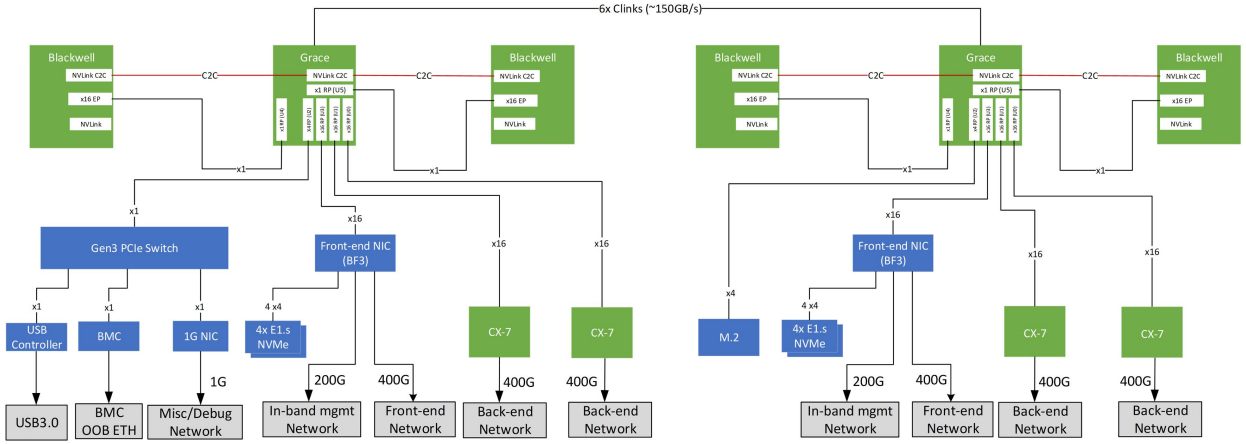
2. 机器和集群的topo举例

我们先看清这些服务器如何连接 GPU、CPU、PCIe switch、NVLink fabric 和 RNIC，再逐个追问：数据每跨过一个边界，地址、ownership、queue、completion、visibility 和 failure model 发生了什么变化。

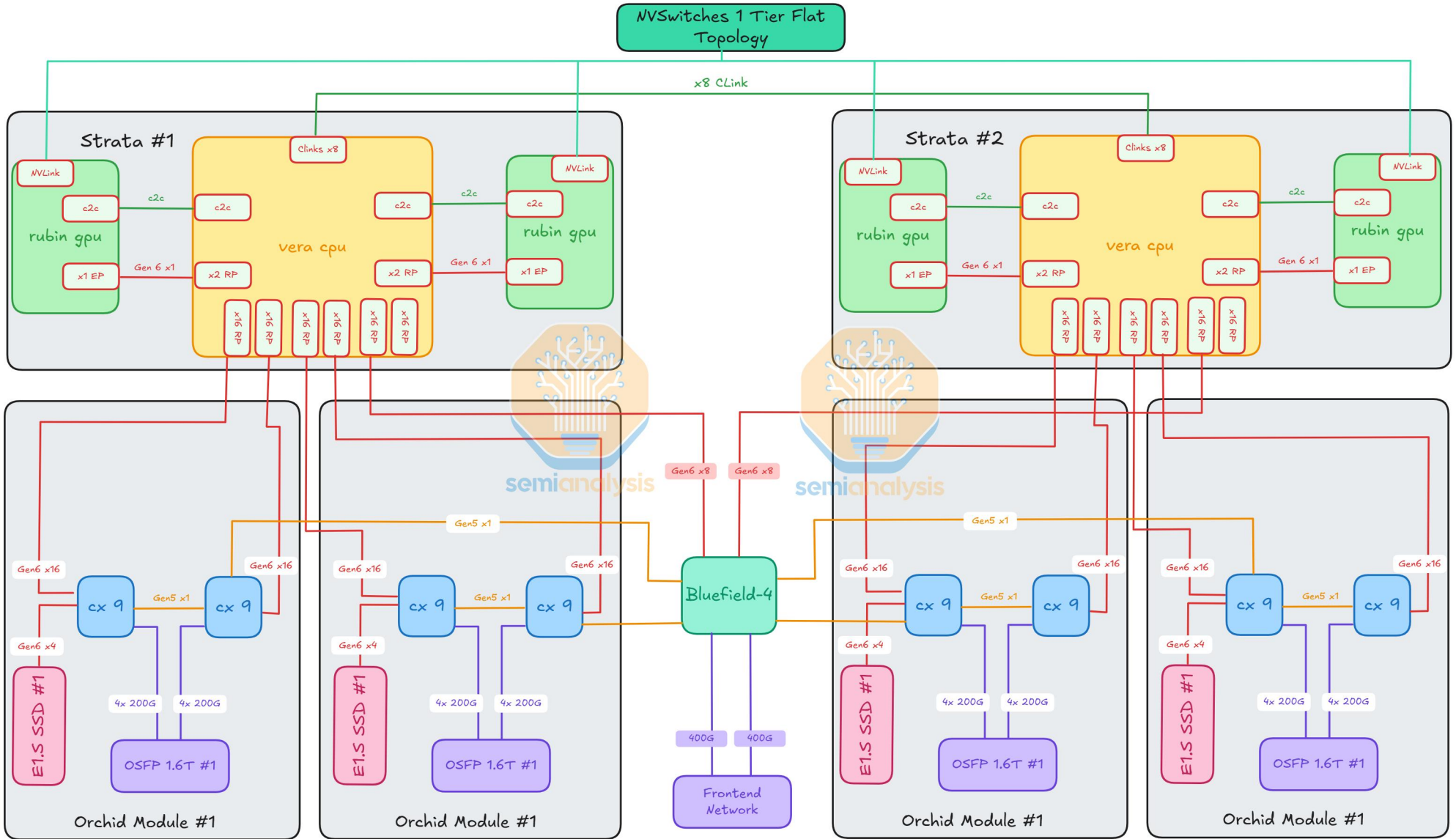




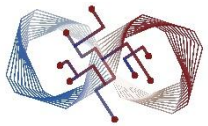




2. 机器和集群的topo举例



2.1 Inside AI Servers: Package, Node and NUMA



第一条路径：节点内专用 fabric

producer kernel writes x[i] into GPU A HBM

↓ source-ready signal / dependency

GPU A fabric endpoint reads or sources the bytes

↓ NVLink / xGMI / HCCS / accelerator fabric

switch / crossbar arbitrates and forwards

↓

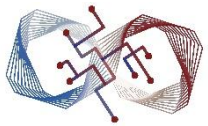
GPU B endpoint places x[i] into GPU B memory

↓ completion + ordering

consumer kernel may read x[i]

如果 A、B 位于同一个带专用 scale-up fabric 的 domain，通信库可能选择 NVLink (xGMI、HCCS 等) peer path，payload 不必经过 host CPU/CPU DRAM、外部 RNIC 或 Ethernet/InfiniBand switch。（是否经过 PCIe 仍取决于具体 GPU、桥接器和拓扑）

2.1 Inside AI Servers: Package, Node and NUMA

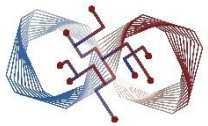


逐步看：A 的 producer 先把 32 KiB chunk 写完；某个 GPU warp、copy/collective engine 或 endpoint 获得 ready 条件；source endpoint 从 HBM/L2/SMEM 对应路径取数；fabric 仲裁；B endpoint 把 bytes 写入目标位置；completion/fence 最后释放 B 的 consumer。对调用 NCCL 或 device put的程序员来说，中间的 endpoint queue、fabric credit、arbitration、peer HBM write 与 signal 通常都是透明层。

瓶颈可能是 source HBM read、endpoint injection、共享 fabric edge、destination HBM write 或 completion wait。症状分别可能是 producer 后长 gap、fabric link 不满、某一 edge 饱和、B 侧 memory busy 或数据已到但 kernel 仍在 poll。证据应组合 NVLink（xGMI/HCCS） counters、per-link bytes/stall、HBM throughput 和 GPU timeline。

如果服务器没有可用的节点内专用 fabric，或者通信目标位于另一台服务器，数据就可能转入通用 I/O 路径：同一节点内可以是 PCIe P2P，跨节点则常见的是 GPUDirect RDMA。

2.1 Inside AI Servers: Package, Node and NUMA



第二条路径：节点内通用 I/O fabric

common host/runtime control:

setup: map/register peer memory (if required)

per operation: build DMA/copy descriptor → submit → poll event/fence

PCIe P2P (when the platform permits peer access):

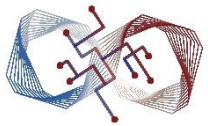
GPU A HBM → GPU A I/O endpoint → PCIe switch / root complex → GPU B I/O endpoint → GPU B HBM

completion/publication (semantics vary):

copy-engine event / fence / signal → required consumer-side ordering/acquire → consume

同一台服务器内、没有专用 GPU fabric 时一种 PCIe peer-to-peer 变体。若平台、IOMMU 和设备组合允许 peer access，GPU A 的 copy/DMA engine 可以通过本地 I/O endpoint、PCIe switch 或 root complex，把 payload 直接写入 GPU B 的显存。（不是 RDMA，不需要经过 RNIC 或外部交换网络）

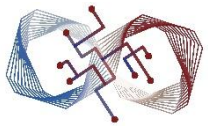
2.1 Inside AI Servers: Package, Node and NUMA



CPU/runtime 可能负责建立 peer mapping、提交 copy/DMA descriptor 和轮询 event/fence，但具体 progress 也可能由 GPU copy engine 或 runtime 异步完成。

需要分别测量 source HBM read、GPU I/O injection、PCIe switch/root-complex 争用、destination HBM write 和 copy completion。若 peer access、IOMMU 映射、PCIe bridge 或设备组合不满足，runtime 可能退回 host-memory staging。

注意两个 GPU 在同一台服务器不一定共享同一个 PCIe switch 或拥有最短路径，应使用 nvidia-smi topo -m、PCIe 拓扑、GPU peer-access 能力、NUMA 绑定、P2P DMA microbenchmark 和 GPU timeline 共同确认实际路径。



第三条路径：经过 CPU/host memory 的 fallback

GPU A HBM —DMA/copy→ source host buffer

source host buffer —RNIC/network→ destination host buffer

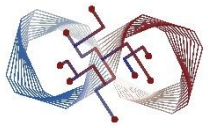
destination host buffer —DMA/copy→ GPU B HBM

同一批 payload 被 memory system 服务至少三次；每一段都有独立的 queue、completion 和 buffer lifetime。

当 peer access、IOMMU 映射、内存注册或设备组合不支持直达时，runtime 可能退回 staging。对同一个 32 KiB chunk，先从 A HBM 搬到 host buffer，等这一段 completion；RNIC 再读取 host buffer 并传到远端 host；最后再搬进 B HBM并发布 consumer-ready。

它会同时消耗 HBM、PCIe、CPU DRAM、RNIC DMA 和 launch/progress 资源；网络本身不变，端到端时间却可能明显增加。上层通常只看见 collective/P2P 变慢，并不知道 runtime 已 fallback。现象是 host memory traffic 与 GPU copy engine 活跃、NIC 仍接近原速、timeline 出现 HBM→host 和 host→HBM 两段。

2.1 Inside AI Servers: Package, Node and NUMA



1. 哪些 GPU、NIC、CPU socket 和 PCIe switch构成同一个高带宽、低延迟域？域内带宽和延迟是否均匀？
2. 对相关端点对来说，正向和反向路径是否对称？它们的 hop、带宽、延迟和拥塞行为是否一致？
3. 每个 NIC 连接到哪个 PCIe switch、root complex 和 NUMA domain？它与哪些 GPU 具有直接的 peer-DMA 路径或更近的 affinity？
4. 哪些 PCIe link、root complex、NIC uplink、fabric edge 或交换机存在 oversubscription？
5. 当数据离开这个高带宽域后， addressability、ownership、completion、ordering、 visibility 和 failure boundary 发生了什么变化？

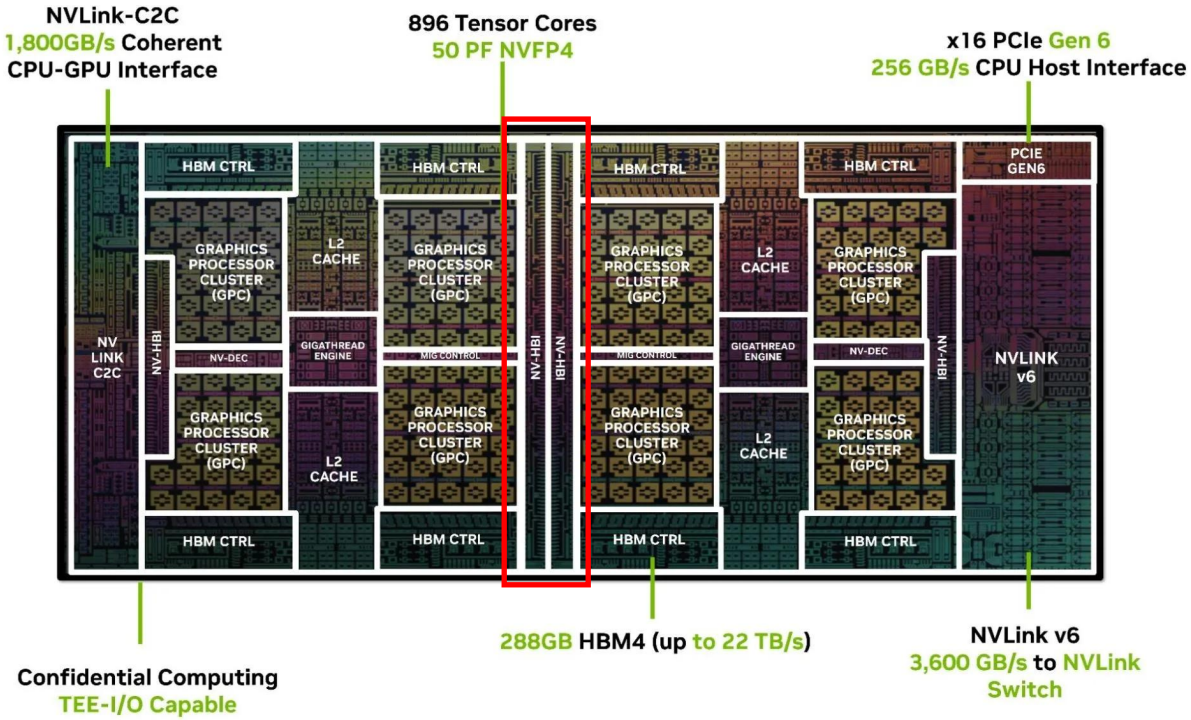
这五个问题会对通信方案构成重要的物理约束，影响 ring 顺序、hierarchical collective 分组、EP expert placement、prefill/decode 的 NIC 选择，是否需要跨 rail 转发或复制等等。

上层 API 通常只表达逻辑 rank，并不暴露完整的 GPU、NIC、PCIe 和 NUMA 映射。因此，通信库需要把 logical rank graph 映射到 physical topology 上。验证时，应把 topology、rank mapping、实际 route、GPU-NIC affinity、per-link bytes/queue 和 per-rank completion time 对齐；仅凭一张厂商拓扑图，无法预测真实通信时间。

2.1 Inside AI Servers: Package, Node and NUMA

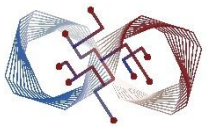
提醒：一个逻辑 device 仍可能有多种物理距离

NVIDIA 官方介绍 Blackwell：两个 reticle-limited GPU dies 由 10 TB/s chip-to-chip interconnect 连接成一个 “single, unified GPU”。这解释了为什么 CUDA 程序员看到一个 device，但不能推出每个地址、每个 HBM 访问和每个 cross-die transaction 都有相同的延迟、带宽或争用。因此逻辑地址空间统一，不等于物理路径同质；仍需把 local / cross-die / peer / remote 分开测量。



Rubin GPUs contain up to 224 SMs and 288GB HBM4 Memory. Available SM count and HBM varies by SKU.

2.2 From Node to Cluster: NIC, Switch and Topology



到这里我们只看了一台服务器：专用 GPU fabric、节点内 PCIe P2P，以及无法直达时的 host-memory staging。现在把边界扩展到另一台服务器。对于支持 GPUDirect RDMA 或类似 peer DMA 的系统，典型 payload 路径是：

GPU A HBM → 本地 GPU–RNIC I/O link → RNIC DMA/packet engine → RDMA 或其他 transport → Ethernet / InfiniBand switch fabric → 远端 RNIC → 远端 GPU–RNIC I/O link → GPU B HBM

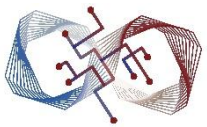
RNIC 接受 WQE 和地址，先验证权限并 DMA 取数，再分段、加 transport header、排队发包，最后产生本地或 transport completion；

交换机只看转发表、队列和 flow-control state；

DPU 可以额外运行 policy、storage/security/control；

scale-up endpoint 则更接近 load/store、tile 或 collective transaction。

2.2 From Node to Cluster: NIC, Switch and Topology

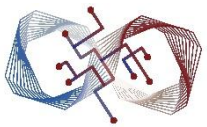


同一产品不必独占一种角色，关键是五件事分别由谁负责：initiator、progress engine、data mover、completion owner、recovery owner。

上层调用者通常看不见 switch queue 和 NIC SRAM，却会以 launch gap、低吞吐、P99 或 timeout 的形式承受它们。

验证时分别找 WQE/CQ、DMA、per-port queue/ECN、DPU core utilization 和 endpoint credit/replay 证据。

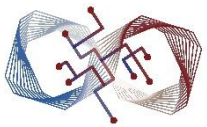
2.2 From Node to Cluster: NIC, Switch and Topology



一条以发送方为主视角的一条 RC RDMA WRITE 端到端事件链：

0. setup: register MR / establish QP（通常可摊销）
1. post: write doorbell → RNIC fetches WQE
2. validate: local lkey/source buffer，remote address/rkey/access permission
3. source DMA: GPU/host memory → RNIC
4. packetize and schedule: segment payload, add transport headers, path / CC / pacing / queue
5. fabric forwarding: RNIC → switch fabric → remote RNIC
6. remote placement: remote RNIC validates packet and rkey, reorder if needed, DMA-write → remote GPU/host memory
7. reliability: ACK/NAK/timeout, retransmit when required
8. retire: release outstanding/replay state, emit sender-side CQE
9. optional publication: WRITE_WITH_IMM / remote flag / counter → consumer-side ordering / acquire → consume

2.2 From Node to Cluster: NIC, Switch and Topology



一个 workload 也可能同时走两层互联（以AWS的CCL与DeepEP举例）：

global collective → intra-system phase: NeuronLink / accelerator fabric → boundary / aggregation buffer → inter-instance phase: EFA + SRD/libfabric path → remote intra-system distribution

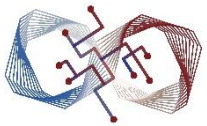
每次跨层都重新出现：chunking、buffer、completion、failure boundary

一个 global collective 可先在 Trainium system 内通过 NeuronLink/accelerator fabric reduce 或 gather；到系统边界后，把某些 chunk 交给 EFA + SRD/libfabric；远端收到后再在本地 fabric 分发。

上层只调用一次 collective，但下面至少有两套路径、两个 RTT 范围、两种故障半径，以及把二者接起来的 buffer/completion。若 intra-system phase 慢，EFA link 可能长期空闲；若跨实例 phase 慢，本地 accelerator fabric 会出现等待。应把 collective timeline 分成 local reduce、boundary ready、EFA transfer、remote distribute 并分别看本地 fabric 与 EFA counters。

DeepEP V1 是DS的一个开源EP通信库：normal/high-throughput kernels 将 NVLink 域内的数据整理后转发到 RDMA 域，目标是用较大的批量和流水化逼近带宽；针对 decode 等小消息场景，V1 另提供 pure-RDMA low-latency kernels

2.2 From Node to Cluster: NIC, Switch and Topology



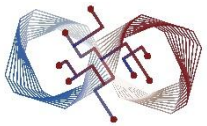
Clos/Fat Tree组网：

路径多，ECMP 通常按 flow 选一条路。但是多条等价路径 ≠ 一个大 flow 自动使用全部路径。path diversity 是拓扑上存在多少条等价路径；path selection 是交换机或 endpoint 如何把一个 flow 映射到其中一条路径；capacity balance 则要求给定 traffic matrix 后，每一个相关 cut、uplink 和交换机队列都有足够服务率。前者存在并不推出后两者自动成立。

在典型 Ethernet/RoCE Clos/Fat Tree 中，ECMP 多数依据五元组或实现相关的 flow/隧道字段做 per-flow hash；InfiniBand 等 fabric 的路由字段和选择机制可能不同。这样可以保持一个 flow 的包有序，但代价是一个长 flow 通常只使用一条 next-hop。

若两个长流的 hash 碰撞到 spine 0，它们会共享同一组 uplink 和队列；spine 1 即使有空闲容量，也不会替这两个 flow 自动分担。增加独立的 QP、subflow 或通信 channel，只有在实现确实将它们映射成独立网络 flow 时，才可能增加 hash 的样本数；即使如此也不能保证均匀：flow 数太少、hash seed 相同、多个 QP 共用相同字段，仍然可能发生 collision。具体实现中 NCCL 的 channel 可能映射到一个或多个 connection/QP/flow，不能把 channel 固定等同于某一个网络 flow。

2.2 From Node to Cluster: NIC, Switch and Topology



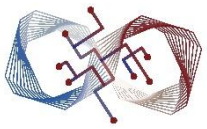
若采用 adaptive routing、flowlet switching 或 packet spraying 可以进一步提高 path entropy。相应的代价是路径选择状态、路径级拥塞控制和包乱序：接收端可能需要 sequence number、offset、gap bitmap 或 reorder buffer；transport 还要决定 per-path credit、ACK、timeout 和重传如何协调。

也就是说交换机侧减少的热点可能转化为 endpoint 的 reorder、recovery 和 SRAM 状态，不能只看 fabric 的平均利用率。

non-blocking 是相对于明确的端口比例和流量假设而言。即使端口比例不构成静态 oversubscription，特定的 incast、all-to-all traffic matrix、同步 burst 或故障后的 reroute 仍可能在某个 leaf、spine link 或 priority queue 形成瞬时 hotspot。

Collective 会把局部差异放大成全局 tail。Ring 中一个慢 edge 会拖住该 step 的下一个 send/recv；All-to-All 中一个热点 ingress 或 uplink 会让相关 rank 的 receive-ready 变晚；barrier 或 collective completion 最终通常由最慢 rank/channel 决定。

2.2 From Node to Cluster: NIC, Switch and Topology



Torus / Mesh互联: v4 TPUs have a direct connection to the nearest neighboring chips in 3 dimensions, resulting in a 3D mesh of networking connections. The connections can be configured as a 3D torus on slices where the topology, AxBxC, is either 2A=B=C or 2A=2B=C, where each dimension is a multiple of 4. For example, 4x4x8, 4x8x8, or 12x12x24.

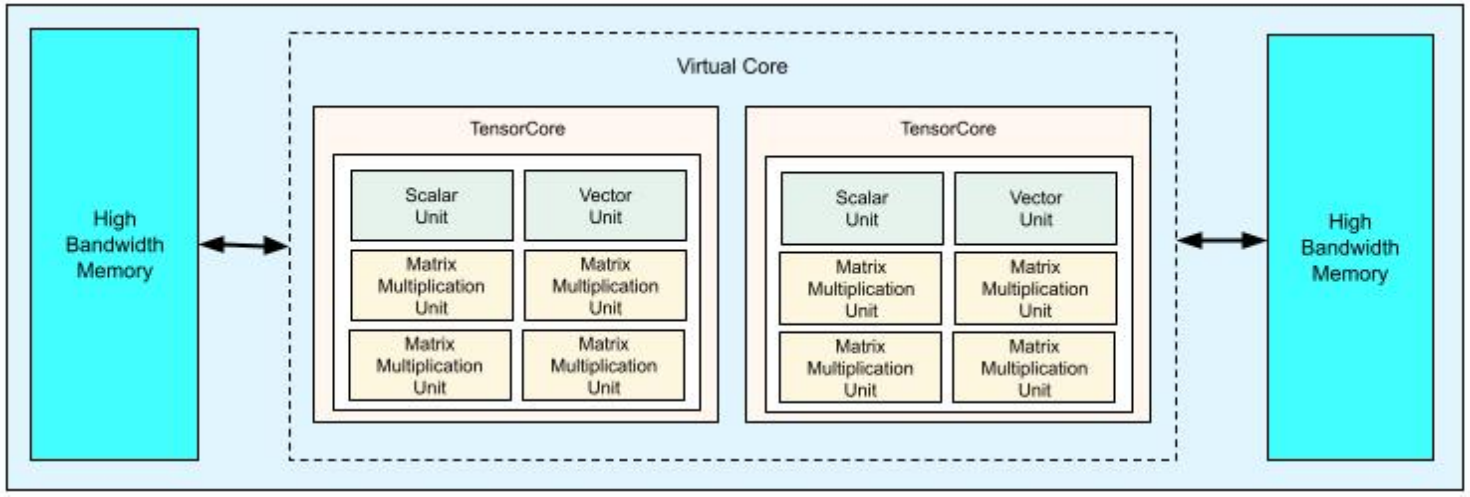
A TPU v4 Pod is composed of 4096 chips interconnected with reconfigurable high-speed links. TPU v4's flexible networking lets you connect the chips in a same-sized slice in multiple ways. When you create a TPU slice, you specify the TPU version and the number of TPU resources you require. You specify a TPU topology using a 3-tuple, AxBxC where A<=B<=C and A, B, C are either all <= 4 or are all integer multiples of 4. The values A, B, and C are the chip counts in each of the three dimensions.

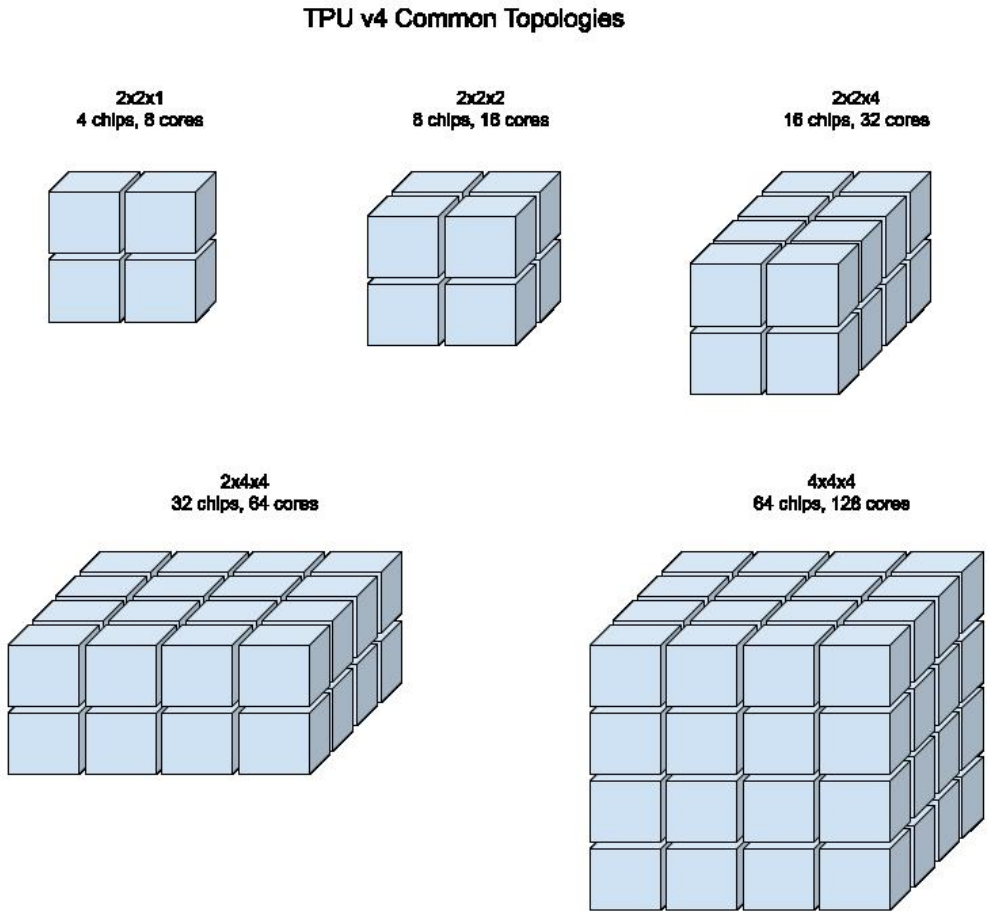
Topologies where 2A=B=C or 2A=2B=C also have topology variants optimized for all-to-all communication, for example, 4×4×8, 8×8×16, and 12×12×24. These are known as twisted torus topologies.

2.2 From Node to Cluster: NIC, Switch and Topology

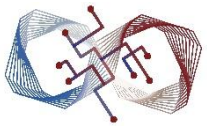
在 Google TPU 的 3D 拓扑中，Mesh 只连接相邻芯片、边界不回连；Torus 在 Mesh 上增加首尾相连的 wrap-around 链路，因此路径更短、更均衡，但需要更多链路与路由处理。

(虽然我们今天不会在这里展开，而且国内也较难获得 TPU v4 计算资源进行实机大规模测试，但 Google 这套设计在计算、互联和编程模型的协同上很有代表性，感兴趣的同学可以之后进一步了解。)





2.2 From Node to Cluster: NIC, Switch and Topology

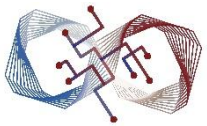


Rail-optimized 是在多 rail 物理拓扑之上，通过 endpoint placement、rank mapping 和 traffic policy 优化数据路径。

对规则 collective，尽量让同 local rank 使用匹配的 GPU–NIC–rail，降低 GPU→NIC first-hop 开销，并减少严重的 ECMP path collision 和最后一跳争用；对动态 P2P 或 EP token，则允许通过 PXN、本地转发、跨 rail 或备用路径进行更灵活的 traffic placement/routing。这里的 PXN、local forwarding、cross-rail traversal 是不同层次的机制。这样做的目标是同时提高整体利用率、路径可预测性和故障时的回退能力，但异常流也可能增加 local copy、hop、completion 和 path state。

rail-optimized 并不意味着所有通信都只能走自己的 rail，也不意味着任何 workload 都一定更快。评估时不应只看总端口带宽，还应观察 rail utilization、GPU–NIC traffic、cross-rail/PXN bytes、rank mapping 以及 failure fallback behavior。

2.2 From Node to Cluster: NIC, Switch and Topology

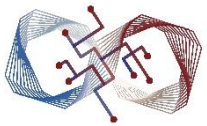


Clos/Fat Tree。它的核心是多个 endpoint 是否共享同一组 leaf、spine 和 uplink。ECMP 通常为每个 flow 选择一条 next hop；因此路径多只表示存在 path diversity，不保证 traffic 已经均匀分布。hash collision、leaf uplink oversubscription 或某个 priority queue 的 backpressure，都可能让一个 collective 的慢 rank 卡在同一条共享边上。

Torus/Mesh没有 Clos 的中心 spine，而是把 endpoint 放在规则网格中，通信主要沿邻居边和维度方向转发。它的优势是邻居关系和维度路由规则清晰，路径长度可以由坐标计算并能按预先设计的网格维度扩展。改变规模或 partition shape 后，rank placement、路由和 collective schedule 仍可能需要重新计算。

所以拓扑不是独立于软件的一张网络图，它规定“哪些通信边共享哪些物理资源”，而 mapping / policy 决定“哪些 rank 和 chunk 使用这些边”。同一个 ring 放在 Clos 上，慢点可能是 hash collision；放在 rail-optimized 的多 rail 系统上，慢点可能是 cross-rail rank edge；放在 Torus 上，慢点则可能来自 hop、partition shape、bisection 或 wrap-around link 的不均衡。Rail 结构可以叠加在 Clos、Torus 或其他 fabric 上；Torus 图是二维投影，真实 3D Torus 还会增加第三个维度和对应的路由选择。

2.2 From Node to Cluster: NIC, Switch and Topology



ScaleUp vs ScaleOut：

scale-up/out 是一次 transport contract 的改变。节点内专用 fabric 或 PCIe P2P 通常拥有较短 RTT、固定拓扑和较小故障域，credit、FEC、link replay 以及 endpoint completion 可以把许多瞬时错误限制在局部；一旦离开服务器，RNIC 成为网络入口，地址需要注册和保护，payload 被分段、排队、路由并在远端重新放置，状态也从单个 endpoint 扩展到 QP、transport、交换机队列和 runtime。RTT/BDP、路径数量、拥塞、丢包、端点重启和恢复状态都会进入设计。

这里的“接口倾向”不是硬分类：某些 scale-up fabric 也可以提供 message-like transaction，RDMA 也可以直接写入对端 GPU 地址空间。

真正要问的是：这个边界在哪里完成 ownership transfer，谁能证明 x[i] 已经可以被 B 消费？ 如果跨边界后只剩一个 opaque message，consumer 需要明确的 ACK/CQE、publication 和 acquire；如果操作进入对端地址空间，仍必须定义 placement、ordering、权限、重复/迟到数据和 terminal error。UALink、SUE、UET/RDMA 的差异，首先是这些状态放在哪一层、覆盖多大的故障域、由谁承担恢复成本。

2. 总结

节点内：HBM_A → local fabric / PCIe / staging → HBM_B

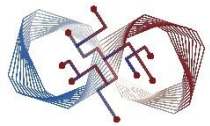
节点间：HBM_A → RNIC → transport + switch → RNIC → HBM_B

每跨一个边界都要重新确认：ownership → address/permission → queue/backpressure → delivery/placement → completion/publication → visibility/acquire → error/recovery scope

物理路径决定哪些资源会共享，拓扑与映射决定等待在哪里形成，协议边界决定哪些证据足以宣称已到达。上层 API 把这些差异压缩成一次 send 或 collective，但系统设计者不能把它们压缩成一个”完成”事件。

下面我们比较两类代表性边界契约：Memory/Transaction 与 RDMA Message/Verbs；它们分别把地址、进度、完成和恢复状态放在哪里。比较一下不同的设计中可靠性、顺序、拥塞、Progress 与故障状态由谁维护？

3. Memory Semantics vs. RDMA Message Semantics



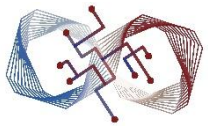
Memory/Transaction 与 RDMA Message/Verbs 的共同目标都是把 x[i] 从 producer 安全地交给 consumer；两者都要处理 source readiness、权限、数据搬运、backpressure、placement、completion、visibility 和 ordering。

区别在于表达方式和状态归属：

Memory / Transaction 通常让调用者围绕地址或 tile 发起 load/store /atomic/fence 或 copy，把地址翻译、请求排队、copy progress 和 fence 状态隐藏在 XPU、memory system 或 fabric endpoint 中；RDMA Message/Verbs 则要求调用者提交 SEND/RECV/WRITE/READ/atomic 等 verb，并提供已注册内存、lkey/rkey、QP/RQ 和 remote address，RNIC 据此维护 WQE、DMA、分段、ACK/retry、replay/reorder 和 CQ。

前者更容易嵌入 kernel dataflow，但可能占用 request queue、translation、copy engine 和同步状态；后者适合 bulk transfer、direct placement 和 routed fabric，但要付出 MR/QP/CQ 以及 transport 控制状态。真正应比较的是从 producer ready 到 consumer start 的完整路径，以及每一层 completion / publication / acquire 所提供的证据。

3. Memory Semantics vs. RDMA Message Semantics



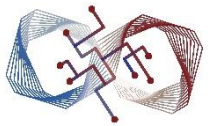
Memory/Transaction 路径的远端访问可能占用 load/store queue、MSHR，以及实现相关的 copy-engine、TMA 或 transaction-buffer 状态；

RDMA/verbs 路径则引入 WQE/doorbell posting、QP context、CQ progress、重传/乱序状态和内存注册成本，其中部分成本可以在初始化阶段摊销。

因而应比较从 producer ready 到 consumer visible 的完整时间，而不能只比较 API 或链路带宽。

一个系统可以用 memory-like API 驱动 message transport，RDMA WRITE 也可以在 GPUDirect RDMA 等条件下直接写入 GPU memory；因此要沿 producer ready → issue/post → data movement → placement → completion/publication → consumer visible → consumer start 观察端到端证据。

3. 状态与控制由谁维护



Delivery: link / transport 观察 gap、duplicate，并触发 retry

Placement: endpoint / DMA 维护 address、offset 与 buffer ownership

Completion: device / NIC / runtime 发布 CQE、counter 或 signal

Visibility: memory system / consumer 建立 scope 与 acquire 语义

Ordering: protocol / runtime / kernel 维护 sequence、phase 与 dependency

常见的责任分布是：link 或 transport 负责发现和恢复各自范围内的缺失与重复；endpoint/DMA 负责 address、offset 和 buffer ownership；device、NIC 或 runtime 发布对应层次的 completion；memory system 提供可见性和顺序的实现机制，consumer/runtime 通过 acquire、fence 和 memory scope 使用这些语义；protocol、runtime 或 kernel 维护 sequence、phase 和 dependency。

实际系统通常跨层协作，不存在固定的单层 owner。Scale-up 设计必须把 completion 与 acquire / release、memory scope、必要时的 atomicity，以及错误和未知结果语义对齐。

3.1 RC

RC：把可靠性、顺序和 progress 绑定在 QP/RNIC

QP / RNIC：address + protection， PSN + ordering， ACK/retry + replay， SQ/RQ/CQ progress

典型 RC 把寻址/保护、序列与顺序、可靠性和队列进度放进 QP/RNIC 的状态机。RNIC 观察 ACK、超时和协议错误，保存重放状态，推进 WQE，并通过 CQE 向 runtime 报告结果；拥塞控制可能由 RNIC、交换机反馈和配置共同完成，不能把它简化成单一模块。优点是 verbs 语义和现有生态成熟，代价是状态与 QP 紧耦合，路径切换、选择性恢复和大规模状态扩展不容易独立演进。

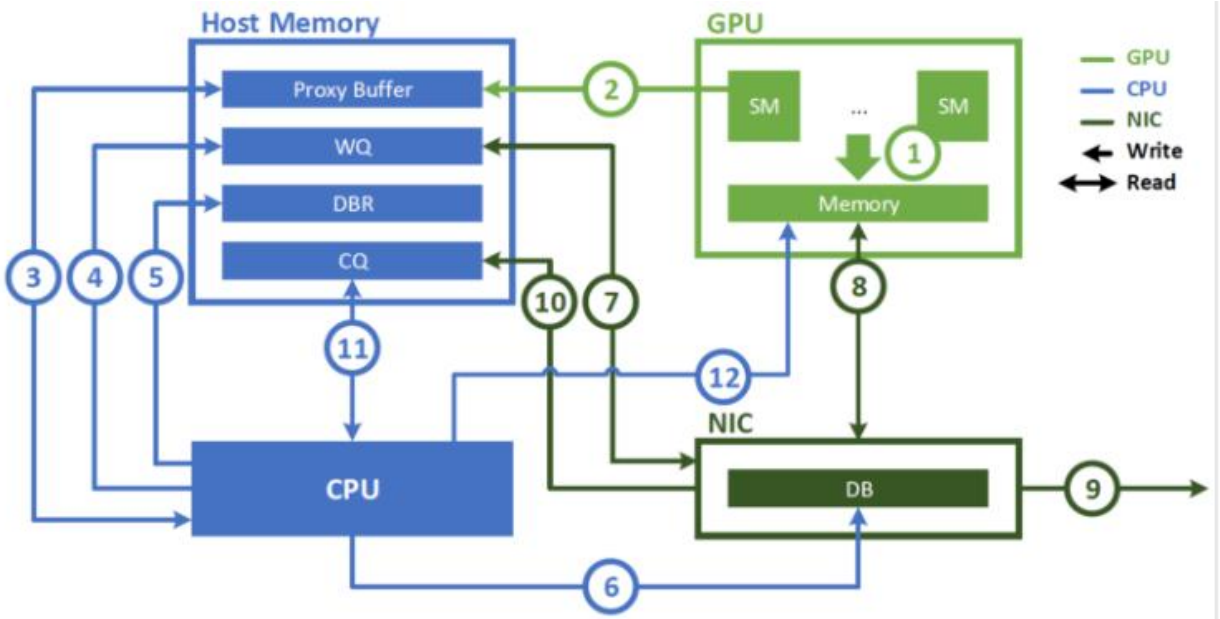


Figure 1. GPU communications using the CPU proxy to initiate NIC communications causes communication bottlenecks

1. The application launches a CUDA kernel that produces data in GPU memory.
2. The application calls an NVSHMEM operation (such as `nvshmem_put`) to communicate with another processing element (PE). This operation can be called from within a CUDA kernel when performing fine-grain or overlapped communication. The NVSHMEM operation writes a work descriptor to the proxy buffer, which is in the host memory.
3. The NVSHMEM proxy thread detects the work descriptor and initiates the corresponding network operation.

The following steps describe the sequence of operations performed by the proxy thread when interacting with an NVIDIA InfiniBand host channel adapter (HCA), such as the ConnectX-6 HCA:

1. The CPU creates a work descriptor and enqueues it on the work queue (WQ) buffer, which resides in the host memory.
2. This descriptor indicates the requested operation such as an RDMA write, and contains the source address, destination address, size, and other necessary network information.
3. The CPU updates the doorbell record (DBR) buffer in the host memory. This buffer is used in the recovery path in case the NIC drops the write to its doorbell (DB).
4. The CPU notifies the NIC by writing to its DB, which is a register in the NIC hardware.
5. The NIC reads the work descriptor from the WQ buffer.
6. The NIC directly copies the data from the GPU memory using GPUDirect RDMA.
7. The NIC transfers the data to the remote node.
8. The NIC indicates that the network operation is completed by writing an event to the completion queue (CQ) buffer on the host memory.
9. The CPU polls on the CQ buffer to detect completion of the network operation.
10. The CPU notifies the GPU that the operation has completed. If GDRCopy is present, it writes a notification flag to the GPU memory directly. Otherwise, it writes that flag to the proxy buffer. The GPU polls on the corresponding memory for the status of the work request.

3.2 MRC

MRC：把 path failure 的恢复下沉到 transport/endpoint

endpoint / path layer：packet multipath + direct placement，SACK / selective retry，ECN + path health，completion 仍需与 runtime 对接

MRC 的核心是让 endpoint/path 层拥有 packet spraying、路径健康、缺口记录和选择性恢复的闭环，从而把部分 link、path 和 switch failure 留在 transport 层。

它把更细的 replay/reorder/health state 换成故障韧性和更高的 path 利用率；但公开论文的 transport 子集主要是 WRITE/WRITE-with-IMMEDIATE，NIC 或 endpoint 本身失效仍可能使 QP/job 失败。MRC 的 completion 还必须接回 runtime，不能把 transport recovery 自动解释成 consumer visibility。

3.3 Falcon

Falcon：用硬件辅助缩短 reaction loop

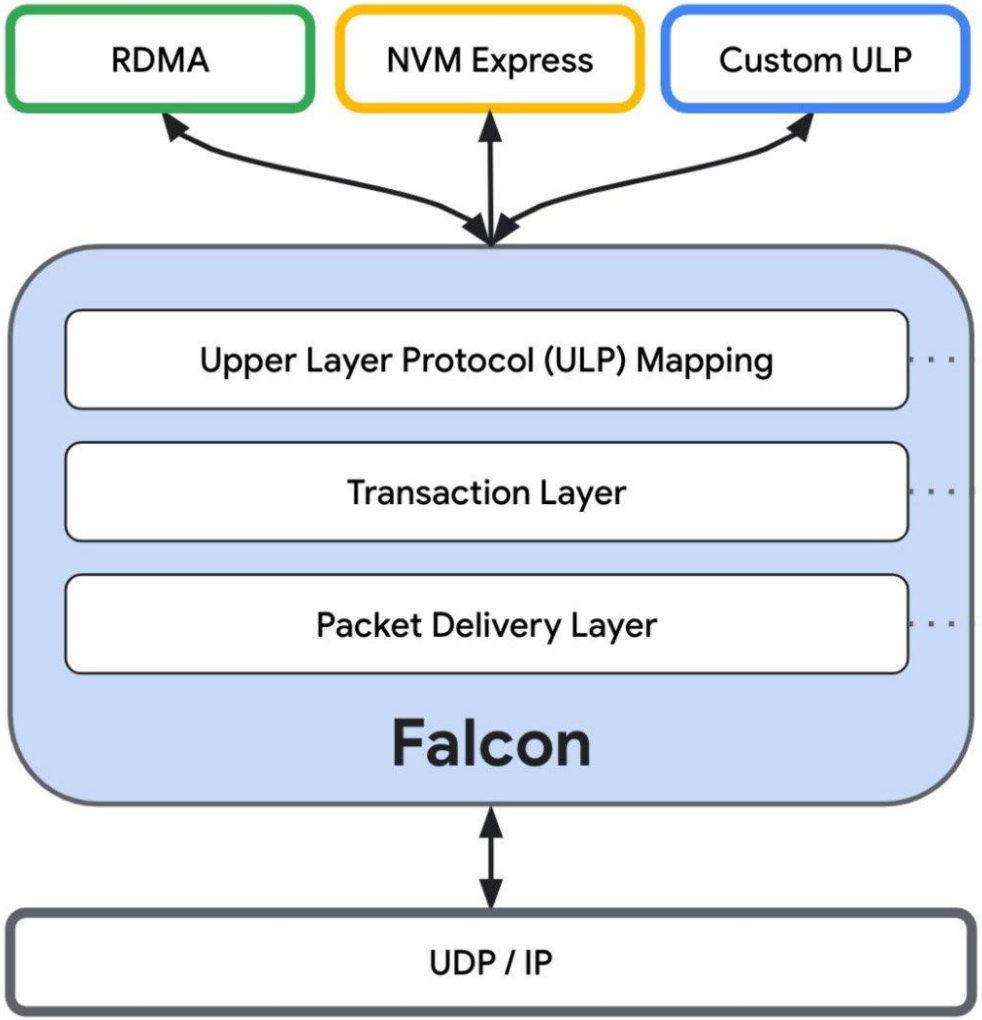
hardware-assisted transport：fine-grained RTT + delay-based CC，per-flow hardware traffic shaping，fast retransmission + error handling，multipath load balancing，programmable engine + ULP mapping

Falcon 是由硬件辅助的 transport、可编程 engine 和软件共同完成控制闭环。

设计点包括：面向有丢包且不依赖特殊交换机支持的通用 Ethernet，细粒度 RTT 测量、基于 delay 的拥塞控制、按 flow 的硬件 traffic shaping、硬件重传与错误处理，以及 multipath load balancing；ULP mapping 层还提供 RDMA/NVMe 兼容、灵活 ordering 和 graceful error handling。

这样的划分意图是缩短拥塞/丢包反应路径并降低 host software 对高消息率的负担，但具体 replay、timer、connection、ULP 和 completion 状态如何切分仍由实现决定。host/runtime 仍需负责 buffer lifetime、应用级 publication/visibility、端点重置和 communicator recovery。

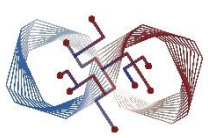
3.3 Falcon



Falcon Layer Function

- Upper Layer Protocol (ULP) Mapping — Messages, Flow Control, ULP Mapping to Falcon
- Transaction Layer — Resource Management, Ordering, Scheduling
- Packet Delivery Layer — Congestion Control, Reliability, Multipathing, Shaping

3.4 UCCL

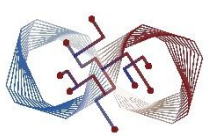


UCCL software-only transport layer：CPU engines: TX/RX · pacing · CC · path/LB， UC / raw UD: software reliability + recovery，RC / SRD: RNIC reliability; software path/chunk control，UC/RC: RNIC segmentation/reassembly + GPUDirect DMA，UD/SRD: scatter-gather + CPU/GPU reassembly path

UCCL-Tran （UCCL的 transport 实现）是位于 collective library 与现有 RDMA/AF_XDP primitives 之间的 software-only extensible transport layer，核心是解耦 data path 与 control path。

CPU engine 负责 TX/RX、pacing、CC、path selection/LB、timeout、软件重传、跨 QP 的 chunk ordering 和连接状态。UC/RC 路径仍由 RNIC 做分段/重组和 GPUDirect payload DMA：UC 绕过固化的 packet reliability、CC 和乱序处理，由软件补上 ACK/timeout/recovery；RC 可在实现支持时关闭硬件 CC，但 packet reliability/ordering 仍在 RNIC。裸 UD 不提供分段/重组和可靠性，UCCL 用 send/recv scatter-gather 让 CPU control header 与 GPU packet payload fate-share，由 CPU 切分/跟踪、GPU reduction kernel 协助 scattered-copy/reorder。

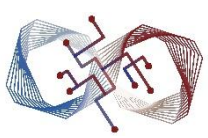
3.4 UCCL



AWS EFA 的 SRD 则是带硬件 multipath、reliability 和 CC 的 reliable datagram primitive；UCCL 可以复用其 datagram delivery，同时把 connection/LB 和 GPU reassembly 放在软件路径，不能将 SRD 简化成“裸 UD 软件可靠性”。非 RDMA 的 AF_XDP 还需 CPU 重组后 cudaMemcpy，也不能画成 GPUDirect。代价包括 CPU/NUMA/polling、QP context、GPU reorder/memory bandwidth、PCIe traffic、可见信号和软件尾延迟。

（这个工作的一个重要卖点就是对AMD GPU以及AWS的EFA网卡友好）

3.5 UET（Ultra Ethernet）



Ultra Ethernet：规定 wire contract，具体 owner 留给实现

UET wire specification： endpoint multipath， ACK / SACK / retry， congestion-control behavior，progress / timer / reorder state → implementation choice

UET 是 wire-level specification，不是一个固定 NIC 或 runtime。它规定端点之间如何表达可靠性、多路径、ACK/SACK 和拥塞控制以便不同实现互操作；但 replay buffer、timer、reorder、progress engine 和 recovery state 究竟放在 NIC、DPU、host 或 accelerator endpoint，规范本身不会替实现决定。评价 UET 时要区分“wire 行为标准化”和“产品内部状态 owner”。

3. 状态与控制由谁维护总结

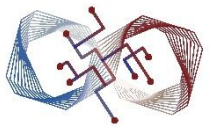
	reliability /	order /	congestion /	progress /	recovery
RC	RNIC/QP	RNIC/QP	RNIC+fabric	RNIC/CQ	QP→runtime
MRC	endpoint	endpoint	endpoint	endpoint	runtime+endpoint
Falcon	HW transport	HW	HW+telemetry	HW	HW→runtime
UCCL	software*	software	software	CPU	CPU/runtime
UET	wire contract implementation-defined				

* UCCL 的 UC/UD 与 RC 路径责任不同；RC 仍保留 RNIC reliability。

状态下沉到 RNIC/硬件，通常换来更短的 reaction loop 和较低 CPU 成本，但增加硬件面积、固件复杂度和部署绑定；状态上移到 host/runtime，通常换来可编程性和更快的策略迭代，但承担 CPU、NUMA、polling 和 tail budget；UET 只规定 wire contract，不能替某个实现填写内部 owner。

五列也不是五个独立模块：同一 endpoint 可能同时负责可靠性、顺序和 progress，runtime 仍要把 completion/publication 接到 consumer acquire。

3. 统一故障演算



packet loss / path degradation→ 谁观察 gap / RTT / ECN? → 谁保存 replay / reorder state? → 谁限制新流量或 reroute? → 谁发布 completion 或 terminal error?

RC RNIC/QP retry → QP error → runtime

MRC path health + SACK → selective recovery → endpoint/runtime

Falcon hardware fast loop → retransmit/reroute → runtime

UCCL CPU observes chunks → software recovery → runtime

UET wire behavior fixed → owner depends on implementation

用同一个丢包或路径退化事件，就能看出五种设计的根本差异：

RC 让 RNIC/QP 闭环，MRC 把部分路径故障留在 endpoint/path 层，Falcon 用专用硬件缩短反应回路，UCCL 让 CPU 以 chunk 粒度控制，UET 只约束互操作行为。若恢复无法继续，系统必须发布带范围和终态的错误，让 runtime 选择重建、切换或重启。

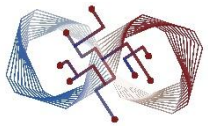
3. 小结

state down → faster reaction · lower CPU · more hardware/state coupling

state up → more programmability · easier iteration · CPU/tail cost

RC、MRC、Falcon、UCCL 和 UET 没有脱离 workload、拓扑和故障模型的绝对赢家。真正要比较的是：谁能观察问题，谁保存状态，谁推进数据，谁控制拥塞，谁发布终态，谁承担面积、CPU、HBM、复杂度和恢复成本。

4. Workload



workload 是 traffic shape 与 wait-for graph 的生成器。它决定参与者集合、对象粒度、fan-in/fan-out、依赖频率、可否分块或预取、何种 completion 才能解除下一条依赖以及局部排队会不会传播成全局 stall。即使移动相同数量的 bytes，AllReduce 可能形成所有 rank 必须等待的 reduction edge，pipeline activation 是一条强时序 P2P edge，KV miss 是带 identity 与 lifetime 的跨 tier object fetch，MoE dispatch/combine 则同时引入动态 routing、metadata、incast 和计算偏斜。

因此分析顺序应是 dependency → object/operation → traffic and state → progress/completion → wait-for graph → exposed critical path。

4. Workload

本章按四步推进：

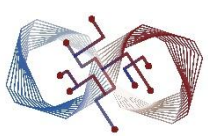
4.1 比较 collective reduction 留在 GPU，或下沉到 NIC、DPU、switch 与 shared-memory fabric 时，bytes、SM/HBM traffic、中间状态和恢复责任如何重新分配；

4.2 连续讨论 P2P、activation、KV cache 与 object/state movement，重点是 identity、placement、ownership、lifetime、tiering 和 publication；

4.3 用 EP/MoE 展示动态 routing、expert skew、incast 与 combine dependency 如何放大尾延迟；

4.4 最后用 topology/NUMA-aware placement 缩短路径，并用 chunking、async data movement 与 compute-communication overlap 隐藏仍不可避免的等待。

4.1 Collective Communication



Where Should Reduction Execute?

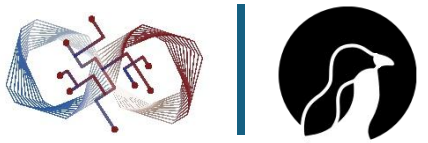
Ring AllReduce per-rank traffic ≈ 2 × (p-1)/p × message_size

先定义 reduction：多个 rank 各有同形状 partial tensor，要按元素做 sum/max 等结合操作，再把结果送回需要的 rank。普通 ring 把 chunk 沿端点逐跳传递，每一跳由 GPU 读入、reduce、再发出。

固定、可结合、shape 规则的 AllReduce 更适合下沉。SHARP 与 NVSwitch shared-memory collective 论文是两类公开案例。前序的 Aggregation Engine 是其架构中占地址空间的端点；这里的 SHARP 是 switch/fabric reduction placement，不是同一个对象。on-package collective engine 也是设计点，参考 Ascend 950 CCU。

观测时同时看 network bytes、GPU HBM traffic、communication-kernel SM time、tree setup、reduce-engine occupancy 和 fallback/error count。若 wire bytes 下降但 GPU 等待未降，瓶颈可能在 tree setup、completion 或结果分发，而不是 reduction 算力。

4.2 P2P and Object/State Movement



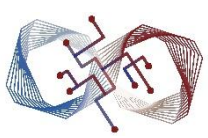
P2P 搬的是有生命周期的对象

object = payload + shape/layout + object/request id+ source/destination placement + ownership/lifetime + completion/error state

send(ptr, bytes) 只描述 payload 范围，没有回答接收端把它放到哪个请求、哪个 layer、哪个 token range，也没有回答 sender 何时可复用 buffer、receiver 超时后是否可重试。PP activation、PD KV、RL 权重和 remote cache 都使用 P2P，但生命周期不同：activation 通常在一个 microbatch 后失效，KV 可能跨多个 decode step 复用，权重更新还要求版本和 ownership 一致。对象语义必须先于传输参数确定，否则“传输完成”无法转化为正确的 ready state。

以 PD KV 为例，control plane 先决定 decode worker 和目标 pool，交换地址/handle 与 request metadata；data plane 搬 KV blocks；completion 更新 block table 或 ready state；失败路径处理重复传输、部分到达和 worker 退出。真正的性能指标因此不只是 GB/s，还包括 setup latency、bytes/object、P99 completion、buffer occupancy、cache hit 和 orphan object 数量。

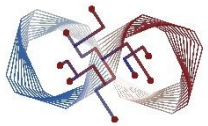
4.2 P2P and Object/State Movement



P2P中不同对象产生不同等待关系。PP中 的 stage B 通常必须很快消费 stage A 的 activation；发送慢会形成 pipeline bubble，接收 buffer 不足会反压前一 stage。PD 中的 KV 是请求级状态，可以预取、在 host/remote tier 暂存并被多个 token 复用；但 decode admission 必须确认所需 blocks 已发布且版本正确。PD 不是把 PP send/recv 原样复用，而是增加了 placement、cache miss、request lifecycle 和 failure recovery。

两者的观测也不同。PP 看 microbatch timeline、send/recv gap、stage bubble 和 queue depth；PD 看每请求 KV bytes、transfer P99、block-ready 时间、decode admission wait、cache hit/miss 和 host/GPU pool occupancy。

4.2 Mooncake

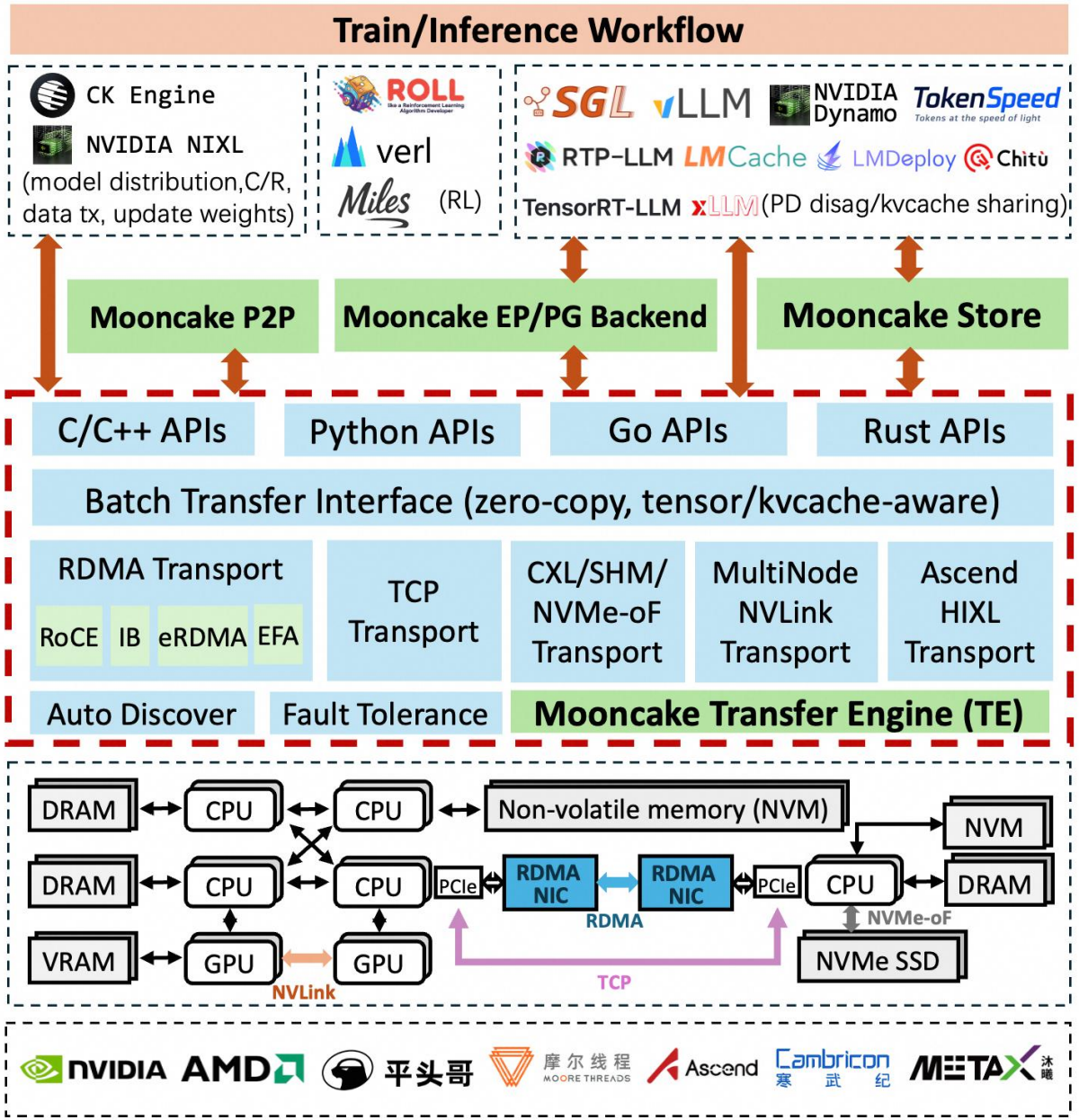


把“怎么搬”和“对象放哪儿”作为两个层次来看。Mooncake Transfer Engine（TE）是数据移动层：在调用方或 Store 提供的、已注册或映射的源/目标内存之间，根据拓扑和可用后端选择 RDMA、TCP、NVLink 等传输通道，提交批量或分片传输，并报告传输任务完成。这里的“零拷贝”是有条件的，具体取决于 transport、内存类型以及注册/映射是否可用。

Mooncake Store 是建立在 TE 之上的分布式 KV/对象缓存与生命周期管理层。它维护 key 到对象、分片和副本位置的映射，负责容量与 placement、复制、pin、淘汰以及对象可见性；实际的跨节点 bytes movement 通常由 TE 执行。TE 的传输完成并不必然等于Store 中对象已经完成提交或对外可读。

因此，零拷贝只消除了 payload 搬运中的额外拷贝，并不会自动消除 metadata lookup、placement 或 segment allocation、排队等待、容量/淘汰决策、副本协调和完成状态发布等控制路径成本。排查时应分别测量 TE 的注册/路径选择、排队、DMA/网络传输、重试和 completion 时间，以及 Store 的 lookup、分配/placement、队列、复制协调和淘汰时间，避免把 Store 的控制面延迟误归因于 RDMA 带宽。

4.2 Mooncake


Train/Inference Workflow
CK Engine
NVIDIA NIXL
(model distribution,C/R, data tx, update weights)
ROLL
verl
Miles
(RL)
SGL
vLLM
NVIDIA Dynamo
TokenSpeed
RTP-LLM
LMCache
LMDeploy
Chitu
TensorRT-LLM
xLLM
(PD disag/kvcache sharing)
Mooncake P2P
Mooncake EP/PG Backend
Mooncake Store
C/C++ APIs
Python APIs
Go APIs
Rust APIs
Batch Transfer Interface (zero-copy, tensor/kvcache-aware)
RDMA Transport
RoCE
IB
eRDMA
EFA
TCP Transport
CXL/SHM/ NVMe-oF Transport
MultiNode NVLink Transport
Ascend HIXL Transport
Auto Discover
Fault Tolerance
Mooncake Transfer Engine (TE)
DRAM
CPU
CPU
Non-volatile memory (NVM)
NVM
DRAM
CPU
CPU
PCIe
RDMA NIC
RDMA NIC
PCIe
CPU
DRAM
RDMA
NVMe-oF
VRAM
GPU
GPU
NVLink
TCP
NVMe SSD
NVIDIA
AMD
平头哥
摩尔线程
MOORE THREADS
Ascend
Cambricon
寒武纪
METAX
沐曦

4.2 Hicache

Mooncake/HiCache：object metadata 决定 bytes 能否被正确复用

key / request / layer / token range / dtype / layout / version

↓ lookup + placement

replica locations → transfer engine → destination pool

↓ publish / lease / eviction / failure cleanup

Mooncake Store 为 KV/object 管 placement、replica、eviction 和多介质 backend；SGLang HiCache 可接 Mooncake backend。当前 SGLang integration 还描述 HiSparse host pool / DSV4 C4 side pool 的 layer-first multi-buffer zero-copy 接口。

同样的一段 bytes，若缺少 layer/token range、layout、dtype 和 version，接收端无法安全复用；若 object 已在另一 replica，系统应选择更近路径，而不是重复从 storage 读取；若 transfer 部分失败，metadata 不能提前发布 ready。通信和缓存的边界是 payload completion 如何原子地更新 object state。

观测应分为 metadata hit/lookup latency、placement decision、transfer latency、publish wait、replica/eviction bytes 和 stale/orphan cleanup。

4.2 DualPath

适用 workload：multi-turn agent · long context · short append · high KV reuse

系统前提：PD disaggregation · layer-wise prefill · external KV storage

Frontend / Storage Network (SNIC)：storage access

Backend / Scale-out / Compute Network (CNIC)：EP/TP/PD relay

Scale-up Network：GPU-local collective / tensor path

PE-read: storage —SNIC→ PE host —CNIC/HBM→ prefill —CNIC→ DE

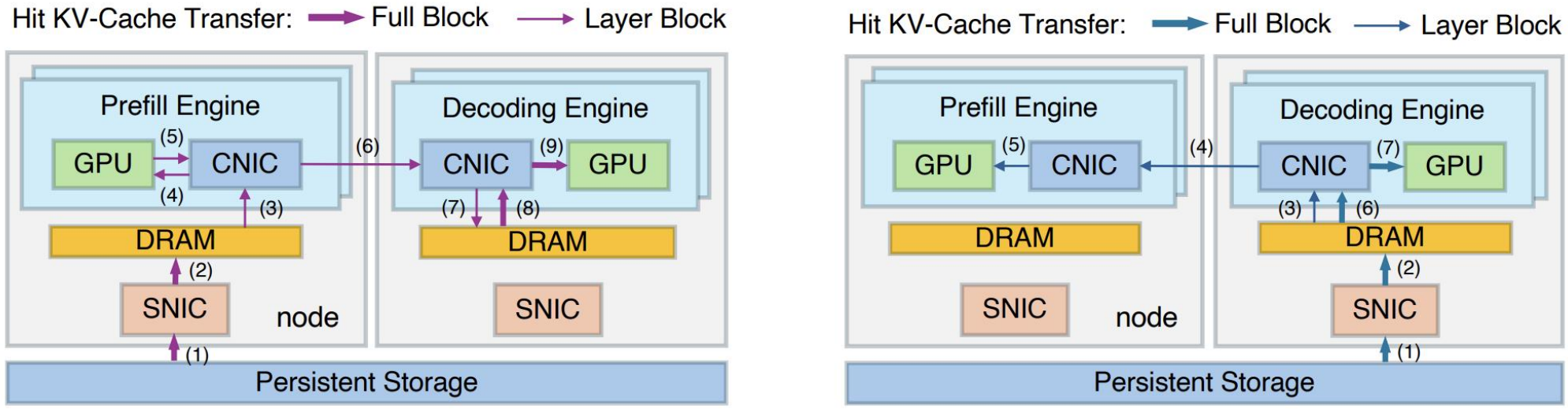
DE-read: storage —SNIC→ DE host —CNIC/RDMA→ PE prefill

↑ scheduler chooses path by queue/load

传统实现通常由 Prefill Engine（PE）的 storage NIC（SNIC）从集中式 KV store 加载命中 KV，DE 主要接收 PE 形成的完整 prompt KV。这是一个结构性入口不平衡：PE-SNIC 可能持续接近饱和，DE-SNIC 却有未利用带宽。DualPath 增加 DE-read：DE 通过自己的 SNIC 读取 KV，先落到 DE host buffer，再经 Backend/Compute NIC（CNIC）和 RDMA 按 layer relay 到 PE；PE-read 仍保留，由 scheduler 在线选择更合适的入口。

4.2 DualPath

这里的三网是部署语境中的角色划分。Frontend/Storage Network 通常承载存储、checkpoint 和 KV-store 访问，SNIC 在自建集群中可以是裸 NIC，在云平台中也可能由 DPU 承担虚拟化与存储服务；Backend/Scale-out/Compute Network 承载 EP/TP 等并行通信、PD relay，以及部分以内存构建的分布式 KV store；Scale-up Network 则是 NVLink 类的 GPU-local 域。三者常见于物理分离的网络，但是否跨集群、是否共享 PCIe 或由 DPU 终止，取决于平台部署。SNIC/CNIC 是职责标签。



(a) PE Read Path

(b) DE Read Path

Figure 4: Dual-path loading illustration. The scheduler dynamically distributes data traffic between the two paths.

4.2 DualPath

关键优势是把 PE SNIC + DE SNIC 聚合成可调度的存储带宽池，并把单一 PE ingress queue 拆成两条路径。layer-wise transfer 与 prefill compute overlap，miss token 新生成的 KV 再回到 DE，与已加载的 hit KV 合并。代价是 DE host buffer、额外 CNIC/PCIe/DRAM traffic、block layout 转换、路径选择和 completion/publication state；论文的 bottleneck-free 分析还假设合理 P/D ratio、良好 PCIe locality、均衡调度和 compute fabric 不拥塞。

测试集群的 compute network 与 storage network 物理隔离，但 DE→PE relay 会进入 compute network，因此不能说 DualPath 让 KV traffic 全程与模型通信物理隔离。论文实际使用 CNIC 的 VL/traffic-class QoS：模型 collective 走高优先级队列，KV transfer 走低优先级队列并使用剩余带宽，从而保护 latency-sensitive EP/TP/CP communication，同时给 KV 流量保留防饿死份额。

DualPath 的主要收益来源有三项。第一，dual-path loading 聚合分散在 PE/DE 节点上的 storage bandwidth。第二，scheduler 不只做 round robin，而是结合 storage read queue、unfinished token、GPU/HBM capacity 和计算负载选择 PE/DE 与 read path。第三，CNIC-centric traffic manager 把网络和本地搬运放进同一个可调度视图，避免 KV 只在 SNIC 上排队。

4.2 HiSparse

把完整 KV 留在 host，只把本轮需要的部分送回 GPU。

Decode GPU: fixed-size hot KV buffer

Decode CPU pinned memory: complete KV

sparse-attention top-k: on-demand host → device swap-in

HiSparse 的基本等待关系是：decode GPU 只保留固定大小 hot buffer，CPU pinned memory 保存更完整 KV；sparse-attention 选择 top-k blocks 后，runtime 才把需要的 blocks swap-in。它减少 HBM 容量需求和无用 KV movement，但增加 top-k decision、host lookup、PCIe/C2C transfer、LRU/slot management 和 graph-compatible staging。

数据路径必须按一次 miss 画：attention/selector 产生 block ids → coordinator 查 host pool → 分配 GPU slot → JIT swap-in kernel/DMA → completion 更新 device table → attention 消费。

若 top-k 变化快或预取不准，PCIe 和 launch tail 会暴露；若 hot buffer 太小，thrashing 会使相同 blocks 反复搬运。观测时看 hit rate、swap-in bytes/token、host→device P50/P99、buffer occupancy 和 decode stall。

4.2 HiSparse

PD direct-to-host：先把 KV 放到最终 host tier，避免一次无效中转

Prefill GPU ──RDMA──> Decode CPU pinned host pool

↓ on demand

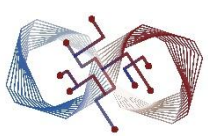
Decode GPU hot buffer

传统实现可能先把 P 的 KV 写到 D GPU staging，再搬到 D host full pool；如果 decode 最终只把少量 hot blocks 拉回 GPU，这次 staging 消耗 D GPU 容量和额外 PCIe/HBM traffic。

direct-to-host 让 Prefill GPU 通过 RDMA 直接写 Decode pinned host pool，完成后由 decode scheduler 发布 blocks，按需 swap-in GPU hot buffer。直接仍不等于没有控制路径：P/D 必须协商目标 host addresses/handles、request/layer offsets、buffer lifetime 和 completion；D 不能在部分写入时发布 block。Prefill 侧不感知 HiSparse，HiSparse 在 decode side 管 host/device tier。

验证收益时同时统计省掉的 D GPU staging bytes、D HBM/PCIe traffic、host write bandwidth、RDMA P99 和 decode admission；若 host memory 或 NUMA 成为瓶颈，direct-to-host 可能只是把队列从 GPU 移到 DRAM。

4.3 EP and MoE Communication



一个 MoE token 要经历七个步骤，all-to-all 只是中间一段

1 router 选择 top-k expert

2 count / quota 决定每个 destination 的 token 数

3 prefix / layout 计算目标 offset

4 pack payload + token/expert/source metadata

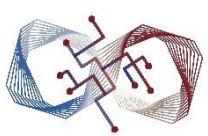
5 dispatch 到 expert rank 的 receive buffer

6 grouped GEMM 读取 expert-major layout

7 combine 回源并按 top-k 权重归并

假设一个 token 的 hidden state 发送给两个 expert。payload 只是 hidden vector；还要携带 token id、source rank、top-k slot/weight、目标 expert 和目标 offset，接收端才能计算并在 combine 时写回正确位置。若 destination count 未知，需要先 count/prefix 或使用预留 buffer；若热点 expert 超出 quota，会产生 drop、padding、reroute 或等待。

4.3 EP and MoE Communication



因此只测一个均匀 all-to-all 会漏掉 router/count、pack/layout、remote completion、GEMM 等待和 combine reduction。应记录每 rank/expert token count、max-to-mean、pack/unpack kernel、network bytes、receive-ready 时间、expert GEMM start 和 combine completion。

模型代码通常只看见 MoE layer；runtime 隐藏了 router output 到 destination count、offset 和 buffer allocation；EP 库隐藏了 pack、transport、notification 与 combine；RNIC/fabric 继续隐藏 packet path 和 recovery。任何一层变慢，用户看到的都可能只是 MoE layer latency 上升。诊断必须从 token histogram 开始，再沿 metadata/layout、payload、completion 到 GEMM。

DeepEP 隐藏的是一套 dispatch/combine 协议：

router output + token buffer → count/prefix + buffer layout → scale-up / scale-out transfer → receive counters / notification → expert-ready buffer → combine transfer + source-side reduction

4.3 EP and MoE Communication

DeepEP 是针对 EP 提供 high-throughput 与 low-latency 的专用 GPU communication kernels。它必须管理 token layout、source/destination metadata、buffer capacity、跨 NVLink/RDMA 的 hybrid path、completion/notification 和 combine。

V2 公开方向转向 NCCL GIN backend、统一 ElasticBuffer，并强调减少 SM 占用；这再次说明 payload path、control state 和 compute resources 要一起设计。

而NCCL EP 用两条执行路径服务不同消息形态：

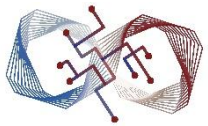
LL: direct P2P all-to-all for latency-sensitive inference

HT: NVLink intra-node aggregation + RDMA inter-node hierarchy

LSA + GIN: GPU-initiated data path

LL path 面向 latency-sensitive inference，倾向 direct P2P、少聚合和短 startup；HT path 面向更大 batch，先在 NVLink 域内聚合再用 RDMA 跨节点。LSA/GIN 提供 GPU-initiated data path，使 producer/consumer kernel 可以更紧密地协调。两条路径消息大小、并发、拓扑和资源预算不同。验证时按 token count/message size 分桶，分别看 startup、SM 占用、NVLink/RDMA bytes 和 P99。

4.3 EP and MoE Communication



UltraEP基于当前 post-gating load 实时复制热点 expert，并在 scale-up domain 内 reroute。

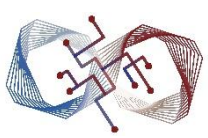
MoonEP通过dynamic redundant experts + symmetric-memory weight layout，目标是固定 rank compute load

当一个 expert 接收的 token 是平均值的数倍时，问题同时发生在三层：多个 sender 在最后一跳形成 incast；目标 rank 的 receive/GEMM queue 变长；其他 rank 已完成却在 combine/barrier 等待。只做 transport CC 能保护队列，却不能消除 expert 计算偏斜。

传统 auxiliary loss/EPLB 偏慢时间尺度；UltraEP/MoonEP 把 replica placement、weight sync 和 reroute 推进到 layer/microbatch critical path。它们通过多副本分摊热点，但要付出权重拷贝/同步、额外 slot、gradient reduction、route plan 和 cache 容量。

要同时看网络与计算证据：per-expert token histogram、receiver ingress queue、expert GEMM duration、replica sync bytes 和 combine wait。只看到 P99 network latency 上升时，不一定是 transport 算法问题。

4.3 EP and MoE Communication



Expert replica 应放在哪取决于“省下的等待”能否覆盖同步成本。

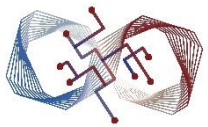
replicate hot experts inside fast domain

route bulk tokens across slower domain only when necessary

在快速 scale-up domain 内复制热点 expert，token reroute 可以少跨慢网络；但每个 replica 需要权重容量、版本同步和训练时 gradient reduction。跨 rack 复制可能增加 fault blast radius 和同步流量。一个简单判断是比较被消除的 dispatch/combine tail + GEMM queue wait，是否大于 replica 创建/同步/调度成本。

UltraEP 的公开实现把复制与 reroute 约束在其支持的 NVLink scale-up domain（说明 EP placement 不能脱离硬件域）。

4.4 Locality and Overlap



Overlap 只能隐藏“依赖允许且资源可并行”的那一段

串行： produce all — communicate all — consume all

流水： produce chunk i+1

├─ communicate chunk i

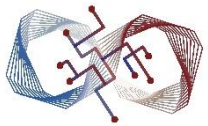
└─ consume chunk i-1

exposed communication = communication - safely hidden portion

先把 tensor 切成可独立发布的 chunk。producer 必须在 chunk i ready 后才能发；consumer 必须等 chunk i 完成并满足 ordering 后才能算。overlap 的收益来自把“等待完整 tensor”改成“等待当前需要的 chunk”。如果 consumer 仍要等所有 chunk，或者算法无法在 partial result 上继续，切得再细也不能缩短 critical path。

第二个前提是资源可并行。通信可能占用 SM 做 pack/reduce/poll，也会读写 HBM/L2；compute 也使用同一资源。如果两者都饱和 HBM，timeline 上重叠只是带宽互相稀释。衡量时计算 exposed_comm = step_with_comm - ideal_compute，并看 producer-ready→send、receive→consumer-start 的 gap。

4.4 Locality and Overlap



Multi-stream 只表达依赖，真正并发还需要资源预算

stream + event express dependency

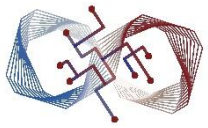
chunking creates independent work

resource partitioning makes concurrency real

stream 决定同一 stream 内顺序，event 建立跨 stream 的 happens-before；它们不会自动切 chunk、不会为 communication 预留 SM，也不会提高 HBM/PCIe/NVLink 带宽。一个正确的流水通常需要：producer 在 chunk-ready event 后发布通信，communication 在 completion signal 后唤醒 consumer，同时限制 in-flight chunk，防止 buffer 被覆盖。

常见失败有四种：event 放在整个 GEMM 末尾，导致无法 tile-level overlap；通信 kernel 占满 SM，让 compute 无法调度；两者争用 HBM，单项时间都变长；chunk 太小，launch/signal/协议开销超过隐藏收益。观测时看 stream dependency、SM active/occupancy、HBM throughput、copy/NVLink/NIC utilization、kernel duration inflation 和 buffer queue depth。

4.4 Overlap 的调度单位从 kernel 缩小到 tile



inter-kernel 最容易实现和调试，但只能在 chunk/kernel 边界交接；

fused producer-consumer 可在 GEMM epilogue 或 tile ready 时发送，减少中间 HBM round trip；

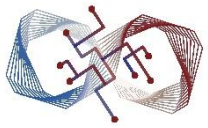
persistent distributed kernel 把 remote load/store、signal/wait 与 compute 放进同一 progress loop，启动间隙最小，但需要显式管理 credits、arrival counters、phase、buffer lifetime 和错误退出。

调度粒度越细，潜在 overlap 越大，固定开销和状态也越多。选择粒度时比较 chunk serialization time、producer/consumer tile 时间、launch/signal 成本、BDP 和可用 buffer。不是越小越好：过小会降低 wire efficiency、增加 metadata/atomic/counter 压力；过大又暴露尾部等待。

Existing grid-level PDL (sm_90+):

- prerequisite grid collectively triggers → dependent grid may start
- consumer waits before reading prerequisite output

4.4 Overlap 的调度单位从 kernel 缩小到 tile



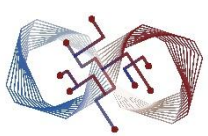
Rubin public claim: required tile becomes ready → corresponding consumer work can start earlier

普通 sequential launch 往往要求 producer kernel 整体完成，consumer kernel 才开始；但 consumer 可能只依赖 producer 已完成的一部分 tile。Programmatic Dependent Launch 允许 prerequisite grid 触发 dependent grid 更早开始，consumer 在真正读取依赖数据前再 wait，从而把 launch/scheduling 与剩余 producer 工作重叠。

现有 griddepcontrol.launch_dependents/wait 从 PTX 7.8、sm_90+ 已存在，但触发仍要求 prerequisite grid 的所有 CTA 已执行 trigger 或结束。Rubin 官方博客展示更细粒度 tile/thread-block producer–consumer coordination。

它缩短的是 kernel boundary 和调度等待，不保证 consumer 能立即读到任意 tile，也不消除资源竞争、layout transform 或中间 HBM/SMEM placement。验证时看 producer tile-ready、dependent launch、consumer wait release 和 SM resource overlap，而不是只看两个 grid start time。

4.4 Counted writes



把数据写入和远端完成计数绑定在一个操作里

sender kernel

└─ fabric put / reduction ──→ remote data

└→ remote byte counter += accessed bytes

receiver observes expected counter value → applies required ordering → consumes data

sender-side mbarrier → tracks local completion / error report

普通 remote write 常要另发 flag/atomic/ack：先写 payload，再保证顺序，再更新“已到”状态；这增加一条控制消息或一次 remote atomic，也容易出现 flag 先被观察而 payload 尚未按所需 scope 可见的错误。counted write 把目标数据访问与目标端 byte counter 更新绑定，receiver 等 counter 达到 expected bytes，再执行所需 ordering 后消费。

它减少的是独立 completion 消息和协调 round，不是让 receiver 无需同步。sender-side mbarrier 跟踪本地 operation completion/error report，remote counter 跟踪目标 bytes；两边语义不同。

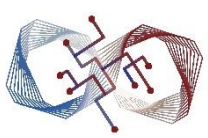
counter 复用还需要 generation/phase，防止上一轮迟到 write 污染下一轮。

4.4 Distributed kernel

distributed kernel 是让多个 GPU 或节点围绕同一条任务与数据流协同执行。典型实现由 host 启动一个或少数几个 GPU-resident persistent kernel；kernel 启动后，常驻 warp/CTA 从 device-visible task queue 获取 chunk，根据数据 placement 决定本地计算或远端访问，并执行 produce → remote put/get → publish ready，接收方则执行 wait/acquire → compute → release buffer。其中，distributed 描述执行参与者和地址空间跨设备，persistent 描述 kernel 长时间驻留，device-driven 描述任务调度、通信提交和完成处理主要由设备侧推进。remote operation 的实际 DMA/progress 仍可能由 NIC 自主完成，或由 host progress thread 代理，取决于具体通信后端。

与传统 host-driven path 在每个 kernel 或 chunk 边界把下一步工作交回 CPU/runtime 相比，这种模型可以绕开重复的 per-chunk host launch/hand-off gap，把流水细化到 chunk 或 tile 粒度：当 chunk i 正在传输时，producer 可以准备 i+1，consumer 则计算已经完成传输的 i-1。但这只有在 chunk 可独立消费、completion 能及时发布、依赖顺序明确，并且有足够的 in-flight buffer和 credit 时才会缩短 consumer-visible critical path。

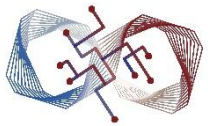
4.4 Distributed kernel



协议必须分别定义 payload、completion 和 lifetime：包括使用对称地址或已注册 memory handle、谁发布 ready counter、receiver 使用何种 acquire/fence、buffer 何时进入下一 phase、credit 耗尽后谁等待，以及 timeout、网络分区、授权变化、节点重启和 unknown result 的恢复方式。这些语义不会由普通跨设备 load/ store 或地址映射自动提供。

代价是常驻通信 warp/CTA 会占用 SM、寄存器、shared memory 和调度资源，并可能与主计算争用 HBM、L2、PCIe、NVLink 或 NIC 带宽。因此评估时不能只看 kernel 时间线是否重叠，还要测量 ready → submit、submit → complete、complete → consume 的阶段延迟、remote outstanding、credit starvation、phase transition，以及主 GEMM 的 occupancy 和 duration inflation。若通信控制路径导致计算明显变慢，减少 host hand-off 的收益可能会被资源争用抵消。

4.4 Case Study: DeepGEMM MegaMoe



真正融合的是端到端数据流：EP dispatch / remote pull → Linear1 gate/up GEMM (FP8 × FP4) → SwiGLU + amax + FP8 requantization → Linear2 GEMM → remote BF16 combine-buffer writes → final top-k combine reduction

实现细节：

persistent kernel + symmetric workspace

receive counters / arrival masks / token metadata

wave-and-phase scheduler

TMA + TMEM + two-CTA tcgen05 MMA

价值：把 dispatch、两个 GEMM、激活/重量化、远端 write-back 和最终 combine 放进同一个进度系统。

4.4 Case Study: DeepGEMM MegaMoe

Communication Latency Can Be Hidden.

The key insight of our EP scheme is that the communication latency can be effectively hidden beneath computation in MoE layers. As shown in Figure 5, in DeepSeek-V4 series, each MoE layer can be decomposed mainly into four stages: two communication-bound stages, *Dispatch* and *Combine*, and two computation-bound stages, *Linear-1* and *Linear-2*. Our profiling reveals that within a single MoE layer, the total time of communication is less than that of the computation. Therefore, after fusing communication and computation into a unified pipeline, computation remains the dominant bottleneck, implying that the system can tolerate lower interconnect bandwidth without degrading end-to-end performance.

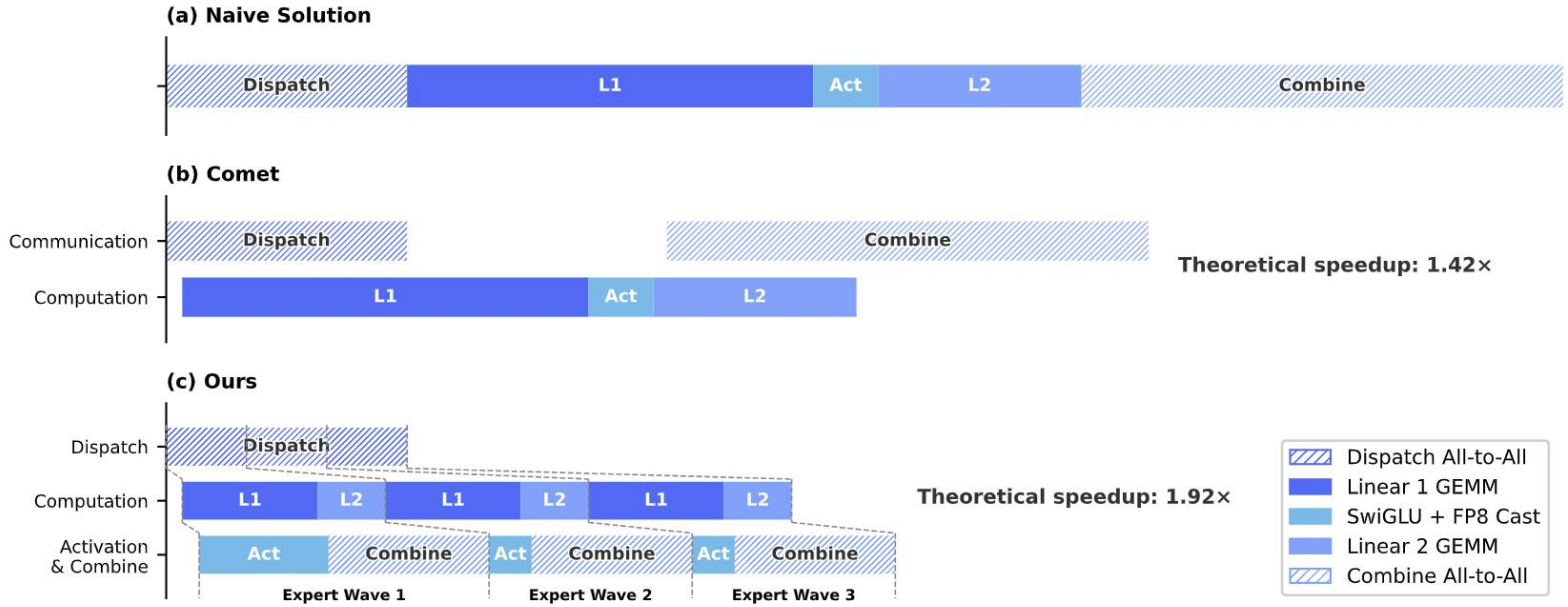
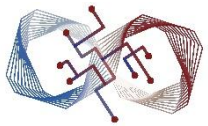


Figure 5: Illustration of our EP scheme with related works. Comet (98) overlaps Dispatch with Linear-1, and Linear-2 with Combine, separately. Our EP scheme achieves a finer-grained overlapping by splitting and scheduling experts into waves. The theoretical speedup is evaluated in the configuration of the DeepSeek-V4-Flash architecture.

5. 哪些开销会在 Scaling 下进入 Critical Path？



setup / restart → bootstrap · communicator · QP / resource state

steady state → bytes · RTT/BDP · queue · progress · contention

failure / operations → detect · recover · diagnose · restart

↓

consumer-visible time / goodput

exposed cost = total cost - safely hidden work - amortized work

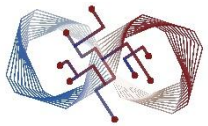
MRC → path / link failure

NCCLX → collective API / algorithm / resource scale

Meta 100K+ GPU production stack → library / transport / operations lifecycle

critical path 是从 producer dependency 到 consumer start 的最长必要依赖链。三篇论文分别切入这条链的不同位置：MRC 问 link/path failure 能否留在 transport/endpoint；Meta 的 CC for 100k+ GPUs 以 NCCLX/CTran 为核心，问 collective API、算法和资源状态能否扩展；SIGCOMM 26 的 Connecting 100K+ GPUs 把 library、transport 与 operations 放进完整生命周期。

5. 哪些开销会在 Scaling 下进入 Critical Path？



当同步 step 的最慢 rank 决定全局进度时，初始化、资源分配、路径恢复、诊断和重启只要不能被 overlap 或 amortize，就会从“系统外成本”变成 consumer-visible time。高 BDP、深队列或更多状态本身不一定是坏事；只有 in-flight state 不足导致链路空闲、过量 burst 造成 queueing，或恢复/诊断阻塞下一次必要依赖时，才进入关键路径。

跨层优化必须逐项计算：真正删除了多少 copy/message/round，工作转移到了 CPU、GPU、NIC、DPU 还是 switch，新增了多少 buffer、QP、reorder、metadata、staleness 和 recovery state。zero-copy 只表示少一次显式 copy，不保证没有 DMA、registration、flow control 或 consumer wait；100% link utilization 也可能意味着更深的队列和更差的 P99。

5.1 OpenAI MRC

同步训练：slowest transfer 决定 step + path collision / incast / link failure → multi-plane Clos + MRC + static SRv6 → job ride out many network failures

MRC 论文的摘要包含三个共同设计点。第一，MRC 在一个 QP 内把 packet spray 到多条路径并主动做负载均衡，避免传统 per-flow ECMP 的 flow collision；第二，用 multi-plane Clos 以较少交换层构建超过 100K GPU 的网络，并用多 plane 提供冗余；第三，用静态 SRv6 source routing 让 MRC 能直接绕开坏路径，而不是等待动态路由收敛。论文的目标是让 link、path、部分交换机数据面异常尽量不升级为 QP 或 job failure。

packet: remote VA + rkey + entropy value (EV)

EV set (约 128–256 / QP) → spray across planes / paths

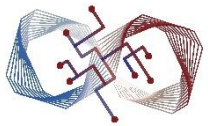
OoO direct placement → SACK / selective retry

ECN path health + trim/NACK + probes → retire / resurrect EV

SRv6 EV → static physical path

production evidence → job survives many path events ≠ recovery is free or instantaneous

5.1 OpenAI MRC



MRC 只扩展 RoCE RC 的一个明确子集，论文中重点是 WRITE 和 WRITE_WITH_IMMEDIATE。每个 packet 携带 RDMA virtual address、remote key 和 EV，接收 NIC 可按 offset 直接放置乱序到达的数据；发送端用 SACK 知道哪些 packet 已到，只重传缺失部分。拥塞时，packet trimming 保留控制头并优先送达，让接收端用 NACK 触发快速重传；ECN 反馈用于暂时避开拥塞 EV，背景 probe 决定坏路径何时恢复。SRv6 把 EV 映射到静态 source route，交换机不再同时运行另一套动态自适应路由。

代价：best-effort/lossy Ethernet 意味着 MRC 必须承担更快的 loss detection、SACK、reorder 和 replay state；每个 QP 要维护 EV/path-health 集合，NIC 需要 direct-placement 和 path mapping；静态 SRv6 简化了运行时路由收敛和故障可观测性，却要求拓扑模板、uSID/路径配置和健康探测正确。

生产案例说明恢复仍会暴露成本：论文报告的 75K GPU 启动实验中，MRC 通过丢包、重传和 EV 替换逐步淘汰坏路径；一个 50K GPU 案例中，T0 transceiver 连续 flap 造成约一分钟、约 25% 的吞吐下降，但 job 未 crash、QP 未失败。NIC–T0 端口故障后，job 可以继续，但 EV remap 需要时间，且系统随后少一个 plane 的容量；长期失效仍需隔离和维修。

5.2 Meta 通信栈NCCLX

NCCLX把 collective library 变成可编程、可扩展的通信层.

NCCLX APIs：host-initiated · host + GPU-resident metadata · device-initiated

CTran: host-driven transport, NVLink · InfiniBand/RoCE · socket,

collectives · P2P · RMA · custom algorithms

Meta 的 Collective Communication for 100k+ GPUs 摘要主张 NCCLX 面向超过 100k GPU，覆盖同步训练和低延迟推理。关键设计是把 API、算法、transport backend 和资源管理放到同一个可扩展框架：支持 host-initiated、带 GPU-resident metadata 的 host API，以及正在扩展的 device-initiated API；

底层 CTran 是 host-driven，提供 NVLink、IB/RoCE 和 socket backend，并支持 collective、P2P、RMA、zero-copy/SM-free 与 fault-tolerant collective。

5.2 Meta 通信栈NCCLX

这条路线的权衡是把一部分 progress、算法选择和控制状态放到 CPU/library：更容易快速定制和复用现有 verbs，也能避免 copy-based NCCL 消耗 SM/HBM；但 CPU control path、registration、QP/CQ 和 host scheduling 仍要付成本。

copy-based NCCL:user buffer → FIFO → RDMA → FIFO → user buffer

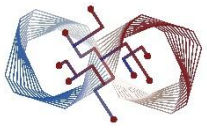
↓ extra SM/HBM copies + chunk/pipeline sync

CTran: user buffer ── direct RDMA ──> user buffer

↓ zero-copy / SM-free, but needs registration + flow control

CTran zero-copy 直接在 user buffer 之间做 RDMA，减少两侧 FIFO copy、SM/HBM 消耗和长 RTT 下的 pipeline/control wait；RMA Put 用于 TP 的细粒度 overlap；HSDP + Fault Tolerant AllReduce 通过 shrink/grow 让部分故障组退出并重新加入；MoE inference 的 GPU-resident collectives 和 low-latency path 则减少 padding、CPU 小消息开销。

5.2 Meta 通信栈 Connecting 100K+ GPUs



100K+ GPU RoCE fabric

multi-building RTT：最高约 30× intra-rack

MoE：bursty all-to-all → transient hotspots

hardware failures：从 anomaly 变成 frequent event

initialization / resource management / testing：进入生命周期

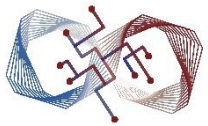
SIGCOMM 2026 的 Connecting 100K+ GPUs 摘要先给出判断：当物理拓扑跨越多个 building，远端链路的 bandwidth-delay product 和延迟差异会放大；MoE 的 bursty all-to-all 会造成瞬时热点；硬件故障的频率使原本轻量的初始化和资源管理不再能被忽略。因此论文的主题是把 library、CTran transport、HBM/QP 资源、流控、诊断、profiling 和 scale testing 作为一个 production communication stack 共同设计。

CCLX initialization / resource lifecycle → lazy connect · lazy channels · metadata slab

DQPLB → segment + QP/outstanding limits by topology / BDP

Fault Analyzer · PerfProfiler · CPU emulation → root cause · observability · control-plane scale

5.2 Meta 通信栈 Connecting 100K+ GPUs

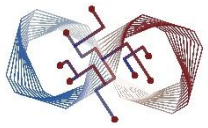


初始化方面CCLX 复用 global communicator、减少重复 bootstrap/AllGather，并把部分算法从 $O(N^2)$ 优化到 O(N)；资源方面，lazy algorithm/channel allocation 和 metadata slab 减少每个 communicator 的 HBM、QP 和 channel 常驻 footprint。

DQPLB 则在 zero-copy 下重新引入显式 segmentation 和 per-connection in-flight limit：根据 rack、zone、cross-zone、cross-DC 的 BDP 调整 data QP 数、segment size 和 outstanding 数；通过 sequence/immediate 与 receiver reorder map 保持多 QP 的逻辑顺序。它换来更少的 switch buffer buildup 和更好的跨距离利用率，但新增了 QP、sequence、reorder、CQ polling 和 topology-specific tuning 状态；上限太低会饿死链路，上限太高会制造 incast。

可观测性和测试也属于 progress/recovery 闭环：Fault Analyzer 用 CollTrace 与 inter-collective dependency graph 区分 root stalled collective 和 cascading timeout；PerfProfiler 记录 registration、control sync、WQE post/completion、QP idle/post frequency，帮助定位 slow rank、慢 I/O 或 QP/path；CPU emulation 用 mock CUDA/verbs 在大规模 CPU rank 上验证 initialization、handle 和 failure path，论文还用它发现 listen-queue overflow 等 control-plane 问题。

5. 横向比较：三篇工作把状态放在哪里？



MRC: endpoint/path。 fast failure bypass; packet/SACK/EV/reorder state

NCCLX：collective library/host transport。API and algorithm freedom; CPU/QP/HBM/control state

Connecting 100K+ ： full lifecycle / operations。 init + resource + flow + diagnosis + test

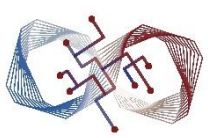
MRC 选择把路径健康、负载均衡和选择性恢复下沉到 endpoint/path，换取几十微秒级的坏路径规避和较低的人工运维压力，但代价是 NIC 协议状态、lossy fabric、OoO placement、静态 source-route 配置和有限 verbs 语义。

NCCLX 的选择是把 API、算法、progress 和资源管理上移到可编程 library/host transport，换取跨训练/推理 workload 的定制能力和更少的 GPU copy 竞争，但代价是 CPU control path、registration、QP/CQ 和软件调度。

SIGCOMM 2026 更进一步：它把初始化、HBM/QP footprint、BDP flow control、故障诊断、profiling 和 emulation 当作同一条生产闭环，代价是更多 telemetry、工具和跨层协作。

因此三篇论文在不同边界上选择观察能力、反应速度、状态成本和可编程性。

5. Takeaway



rare failure × 100K endpoints → routine recovery

per-rank state × communicators → HBM / QP footprint

RTT heterogeneity × bursty traffic → BDP / queue / tail

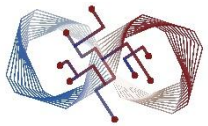
cascading timeout × synchronization → diagnosis / observability

startup / restart → training goodput

MRC 让一部分 link/path failure 留在 transport，但不能消除 NIC/port、GPU 或 job-level fault；NCCLX 让 collective API、zero-copy、初始化和资源管理随 workload 一起扩展，但不能消除 registration、CPU control 和 QP state；SIGCOMM 2026 说明没有 initialization、flow-control、diagnosis、profiling 和 emulation 的通信库，无法被可靠地运维到 100K+。

最终仍回到 critical path：如果恢复、预取、诊断或初始化能在独立工作期间完成，它们可以隐藏；如果它们阻塞下一次必要 collective、consumer 或 restart，它们就是端到端时间。评估应报告 consumer-visible latency、P99/tail、step goodput、restart budget、HBM/QP footprint 和 time-to-diagnosis。

6. 重新划分Compute、Memory与 Interconnect边界



Decode 为什么迫使我们重画边界?

Prefill：通常更接近 compute-bound

Decode：通常更接近 memory/latency-bound (需要区分Speculative Decoding/low latency场景)

active KV → warm KV → historical context

G1 HBM → G2 DRAM → G3 local SSD → G3.5 pod tier → G4 shared storage

TTFT · TPOT · tokens/s/user · KV capacity · fabric tail · power

这是 workload tendency，并不绝对；batch、序列长度、量化、并行策略和 kernel 实现会改变 roofline。Decode 每轮只生成少量 token，却反复读取权重、访问增长中的 KV cache，并可能等待跨芯片的小消息，因此瓶颈常从 Tensor Core 峰值转向内存层次、封装、互联和排队尾延迟。

KV 的确切字节数取决于层数、head 组织、dtype、量化和 attention 变体，不能用一个固定的“每 token 大小”代表所有模型；但容量会随上下文长度和并发增长，访问位置也会从 active request 延伸到 warm/history。评价硬件应同时看 TTFT、TPOT、tokens/s/user、KV capacity、fabric tail 和能耗。

6. CPU、GPU、DPU责任如何划分

CPU：orchestration · control · system memory

GPU：tensor compute · HBM · kernel dataflow

DPU/NIC：I/O · isolation · DMA/transport offload

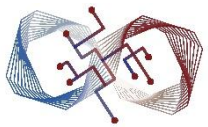
capacity boundary　　HBF / context tier

compute–memory boundary PNM / 3D stacking

compute–fabric boundary low-latency / in-network

这里的 CPU、GPU、DPU 是三组责任。CPU 通常承载调度、控制、协议编排和系统内存；GPU 承载张量计算、HBM 访问和 kernel 内 dataflow；DPU/NIC 可能承担 DMA、网络/存储终止、租户隔离和部分 progress/offload。DPU 的关键问题是明确它观察哪些状态、推进哪些队列、拥有哪种 completion 和 recovery 权限。四条研究路线对应不同边界移动：HBF 把容量边界外移，PNM 把部分 movement 靠近内存，3D stacking 提高带宽密度，低延迟互联缩短小消息同步；每次移动都要重新定义 placement、completion、visibility、QoS 和 recovery。

6.1 Cerebras 与 Groq



Cerebras: wafer-scale fabric + local SRAM → fewer fine-grained chip boundaries → off-wafer I/O and capacity become harder limits

Groq-style: compiler/static schedule + spatial dataflow → ordering/progress become more predictable → dynamic requests and external network remain

Cerebras 把大量细粒度芯片间 movement 收进 wafer-scale fabric，减少片间协议和同步边界；但 SRAM 容量、长上下文和 off-wafer I/O 仍是硬约束。SemiAnalysis 报告说明：WSE-3 约 44 GB SRAM/wafer、约 43 PB/s 聚合片上带宽和约 2.4 Tb/s off-wafer I/O。

Groq 支持 compiler-managed/static scheduling、spatial dataflow 和 SRAM-centric local memory。它把部分 ordering、资源分配和 progress 决策前移到编译器，减少动态仲裁和 timing variance；这不等于所有执行都 cycle-level deterministic，也不等于功耗和散热问题自动消失。容量、动态 batching、长上下文、MoE 路由和外部网络仍需运行时处理。

6.2 ICMS

NVIDIA context-memory 方向：把 KV / context 分到多个 tier

G1: GPU HBM active decode KV

G2: system DRAM staging / warm KV

G3: local SSD node-local warm KV

G3.5: CMX pod-level shared KV / context

G4: shared object / file durable history / cold artifacts

CMX = Ethernet-attached flash/context tier，Dynamo + NIXL → KV block pre-stage → G2 / G1，BlueField-4 + Spectrum-X → KV I/O and fabric

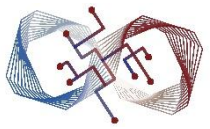
NVIDIA 在 2026 年 3 月的材料中将 CMX 定位为 Vera Rubin AI factory 的 pod-level context-memory storage platform：它是在 G1–G3 与 G4 之间补充一个面向 KV/context 的 G3.5 层。CMX 使用 BlueField-4 承担 KV I/O 和部分 control-plane/data-path offload，使用 Spectrum-X Ethernet 提供面向 RDMA 的 pod 内连接；Dynamo 的 KV block manager、NVIDIA Inference Transfer Library（NIXL）以及相关编排层负责决定对象放置、共享和预取。

6.2 ICMS

这条路径的关键是能否把 G3.5 的访问放在 decode 依赖之前：预取命中时，CMX 的延迟可以由 request-level lead time 隐藏；同步 miss 时，lookup、queue、Ethernet/RDMA transfer、flash read 和 promotion 会直接进入 TTFT/TPOT critical path。无论使用哪一层，都必须维护 object identity、版本/generation、ownership、completion、publication、eviction 和 failure policy。

（但是这个设计的好坏目前还没有得到验证，BlueField DPU的设计也一直被很多人所攻击。这里仍然介绍CMX/ICMS 仅作为一种设计供大家参考和批判性学习。）

6.2 Discussion: 集中式池化还是分布式缓存?

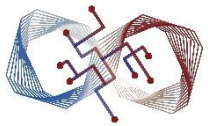


对warm KV cache来说，集中式池把 metadata、placement 和带宽仲裁放在共享 context pool，便于弹性扩缩容和跨节点复用，但中心目录、入口带宽和故障切换会成为系统级约束；分布式缓存把 ownership/目录分片到节点或 shard，命中时路径更短，也能利用 host DRAM 与 local SSD，但要处理 peer transfer、热点偏斜、失效副本、版本/generation 和重平衡。对可重算的 KV，通常可以用 lease、版本和明确的 stale/evict policy。

存储接入 Scale-Up 可减少 host staging 并获得更短 RTT，但必须定义地址/权限、completion、visibility 和 QoS；接入 Scale-Out 可获得更大的池化容量和弹性，但把 RNIC、交换机队列、多路径、拥塞隔离和 partial failure 带入路径。

比较两者时固定 workload、KV hit ratio、对象大小、prefetch slack 和故障模型，测端到端 consumer-visible latency。

6.3 HBF + PNM + low-latency interconnect



HBF cold weights / cold context capacity

PNM filter · transpose · reduce near memory

3D memory–logic dense bandwidth and energy efficiency

low-latency fabric small-message dispatch / completion

HBF 把容量边界外移，但页粒度、读延迟、写寿命和写放大限制其对象类型；对热 KV 不能只看容量，还要看持续写入和回收。

PNM 把部分 movement 靠近内存，减少 GPU/CPU 与 memory tier 之间的搬运，但增加分片、编程和一致性/完成语义；

3D memory–logic stacking 通过垂直互联提高带宽密度并降低每 bit 能耗，同时把热流、良率、封装、接口标准和 software mapping 绑在一起。

低延迟互联可缩短 MoE dispatch/combine 等同步边，却不能替代容量层次。

四者可以组合，但每个边界都必须重新定义 placement、ordering、completion、visibility、QoS 和 recovery。

6.4 NetDAM

把 PNM 和部分 collective/data movement 放到网络边缘。

GPU/XPU ── scale-up / Ethernet ── [NetDAM]

├─ SRAM/HBM

├─ vector/reduce engine

└─ programmable memory operations

NetDAM 是 2021 年 FPGA/100GE 研究原型：memory 直接挂在网络侧，并用可编程 in-memory ISA 做 vector/reduce 等操作。它适合拿来思考 MoE dispatch/combine、metadata transform 和部分 collective 如何靠近数据。NetDAM 真正新增的是一份协议责任账本：operation identity、capability、ordering、completion、backpressure、重复执行、unknown result、partial failure 和 recovery 都必须定义；MoE 还要保存 token、expert、sequence 和归约状态。因此它不是透明远端 RAM，而是“网络边缘的可编程数据平面”。

6. Takeaway

Cerebras 减少部分细粒度芯片间边界，但 SRAM capacity 与 off-wafer I/O 成为更硬的约束；

Groq 把更多 ordering/progress 决策前移到 compiler/static schedule，但动态请求、容量和外部故障仍需运行时处理；

CMX/warm-KV tier 减少本地容量压力和历史 KV 重算，却新增 pool metadata、prefetch、QoS、eviction 和 failure policy；

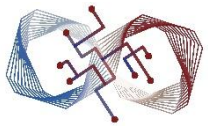
HBF 把冷对象移出热 HBM，却引入页延迟、寿命和写放大；

PNM/3D stacking 缩短 memory movement，却引入散热、封装和 software mapping；

NetDAM 把部分 collective/data movement 放到 network/memory 附近，却必须新增 operation identity、权限、ordering、completion 与 partial-failure recovery。

因此对每个新边界都要重新画 payload、control/metadata、completion/publication 三类逻辑角色，并标出 recovery 闭环，再计算 compute、metadata、buffer、staleness、failure 和 thermal cost。责任可能被真正删除，也可能只是被转移、复制或变成更紧的跨层耦合。

附录



时间原因，这里省去了话题 5、6 的很多细节，感兴趣的同学可以参考原文进一步学习。

关于 OpenAI MRC 的更多细节，可以参考 OpenAI 的官方介绍：https://openai.com/index/mrc-supercomputer-networking

关于 Meta 通信栈在100K+ GPU下设计的更多细节，可以参考：https://arxiv.org/pdf/2510.20171 与 https://dl.acm.org/doi/epdf/10.1145/3789240.3829152

另外，作为我们这里 baseline 的 NCCL，在过去两年里也发生了很大的变化和进步，感兴趣的同学可以直接阅读其最新代码并翻阅其Release：https://github.com/NVIDIA/nccl

NVIDIA 还提供了 DOCA Platform，是 BlueField/DPU 的基础设施与加速 SDK。笔者目前还没有实际尝试过，但这同样是一个值得持续关注的生态：https://github.com/NVIDIA/doca-platform

最后，如果大家还想进一步了解 LLM 推理所面临的硬件挑战以及未来的研究方向，可以参考 David Patterson 2026 年初的这篇工作：https://www.arxiv.org/abs/2601.05047

特别推荐：微信公众号**zartbot**，欢迎大家关注学习。