新型总线技术研究 – Scale UP&OUT – UB

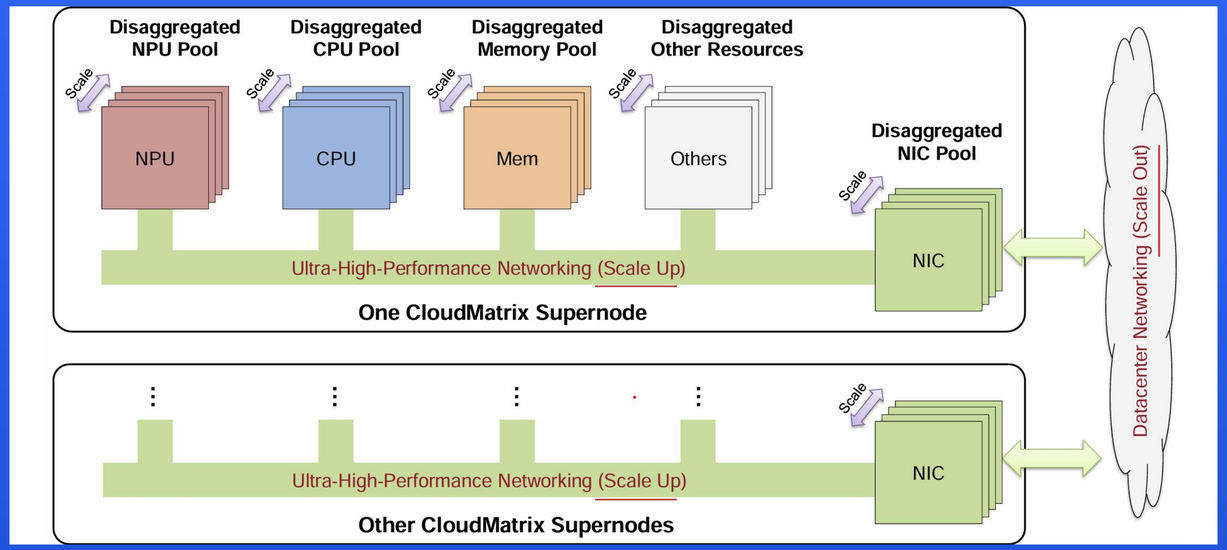
黎亮，openEuler Sig-Long committer，2025-08-22

开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

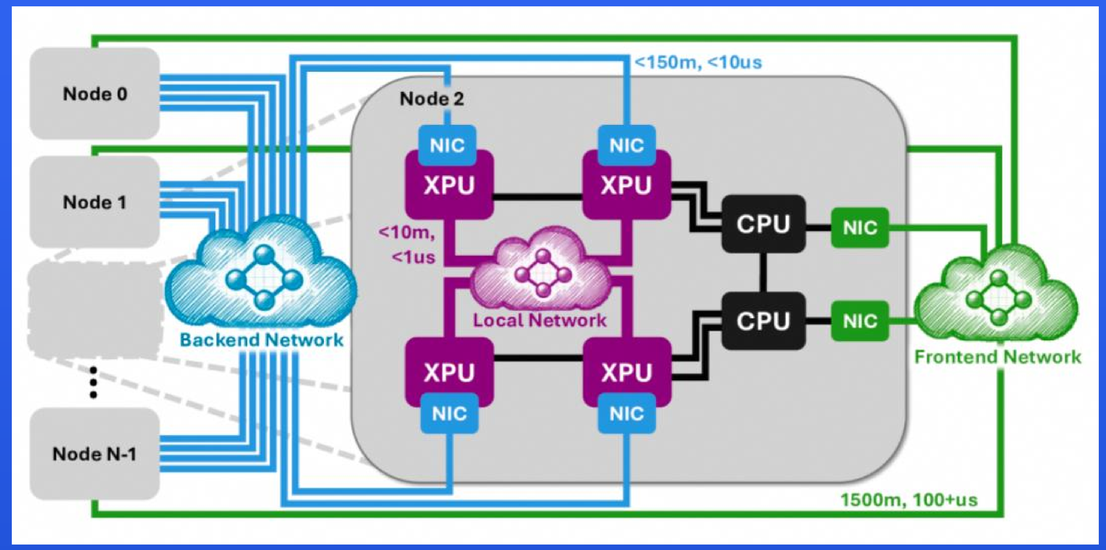
目录

大规模异构计算系统的互联架构

(大规模) 异构计算系统



https://arxiv.org/pdf/2503.20377



https://arxiv.org/pdf/2506.12708

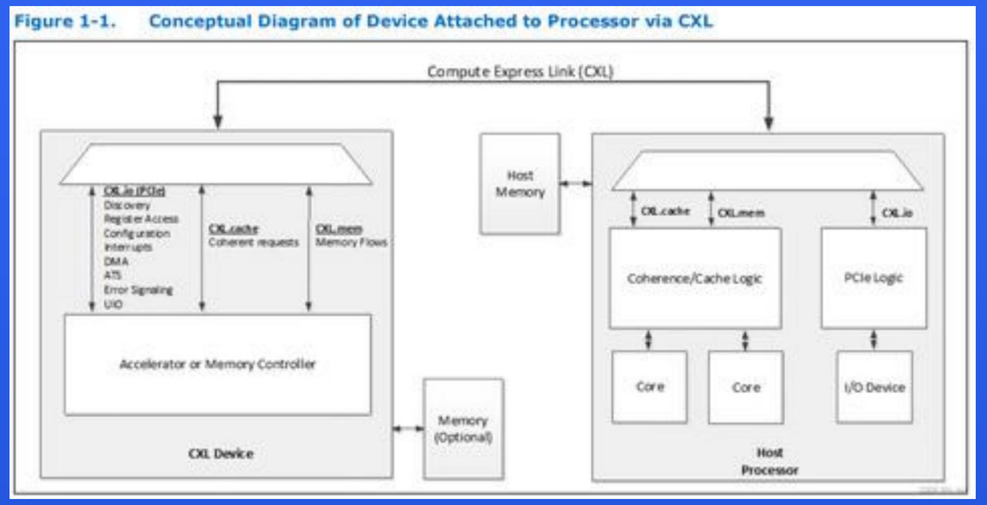
Scale UP / OUT 边界区分的维度:

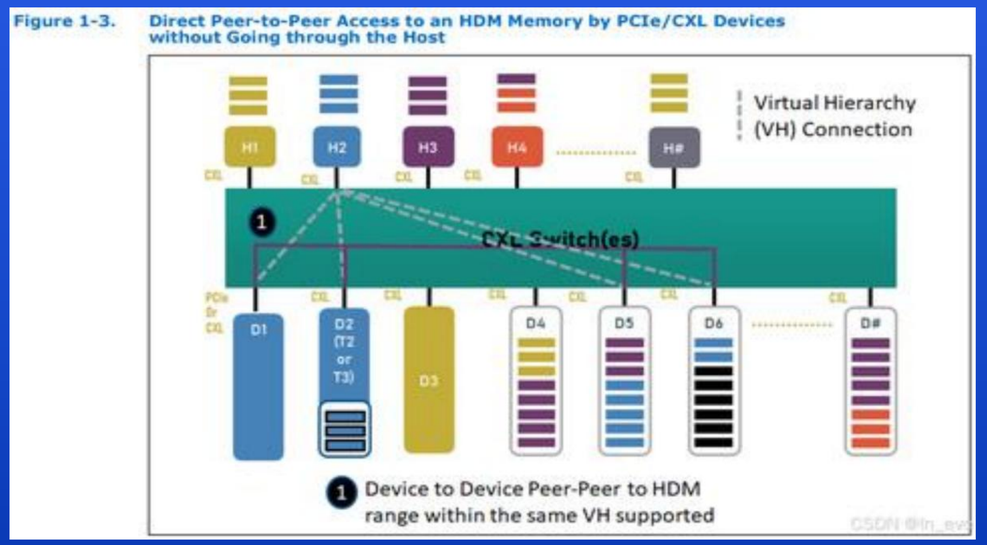
- 互联介质
- 物理距离
- 传输带宽
- 访问时延
- 编/寻址与数据移动方式
- 可靠性
- tbd

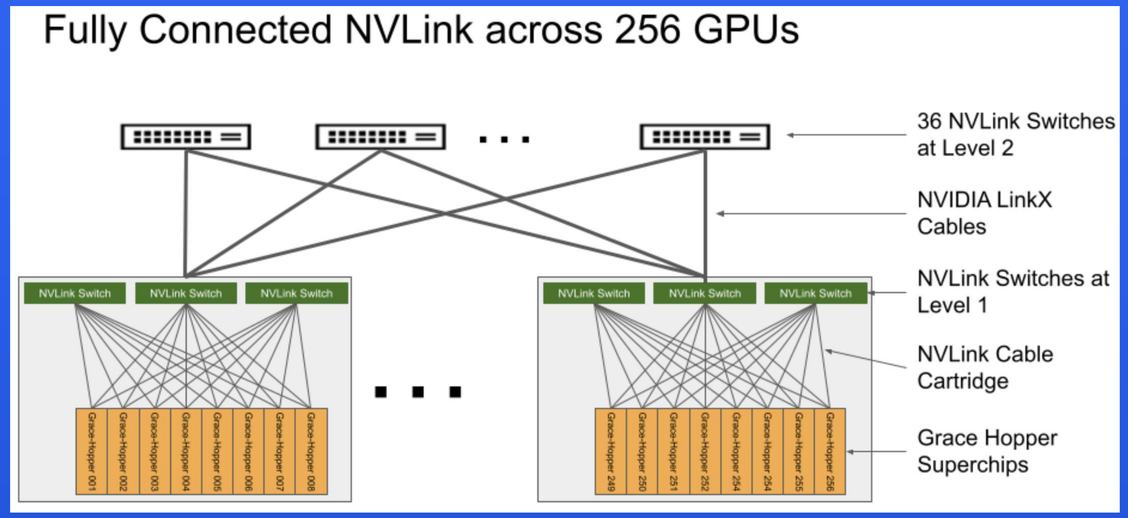
Topic: 当基于内存语义时，边界还存在吗?

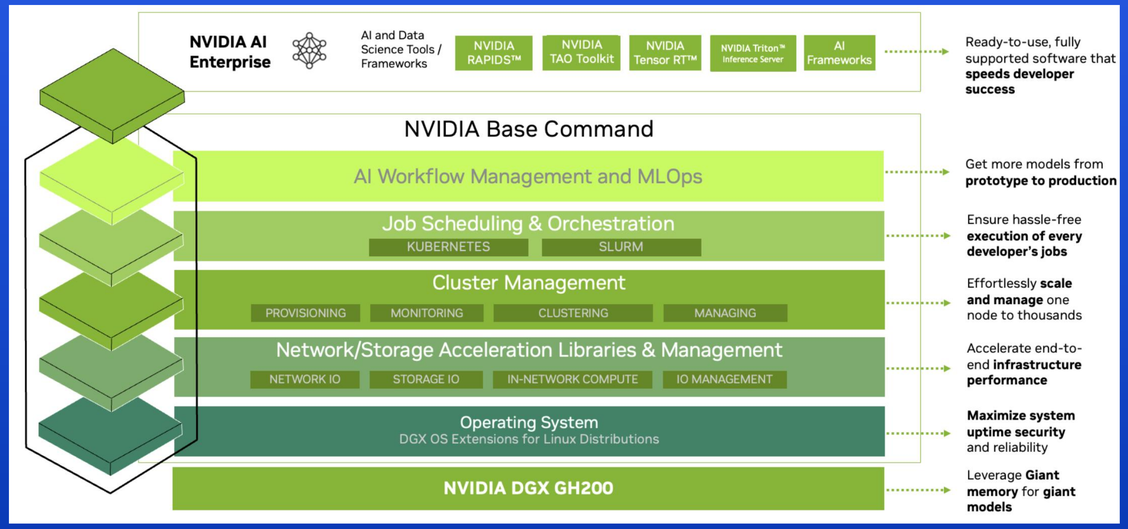
Scale UP 总线技术观察

CXL， NVLink， Scale-UP Ethernet， UB -- <1>



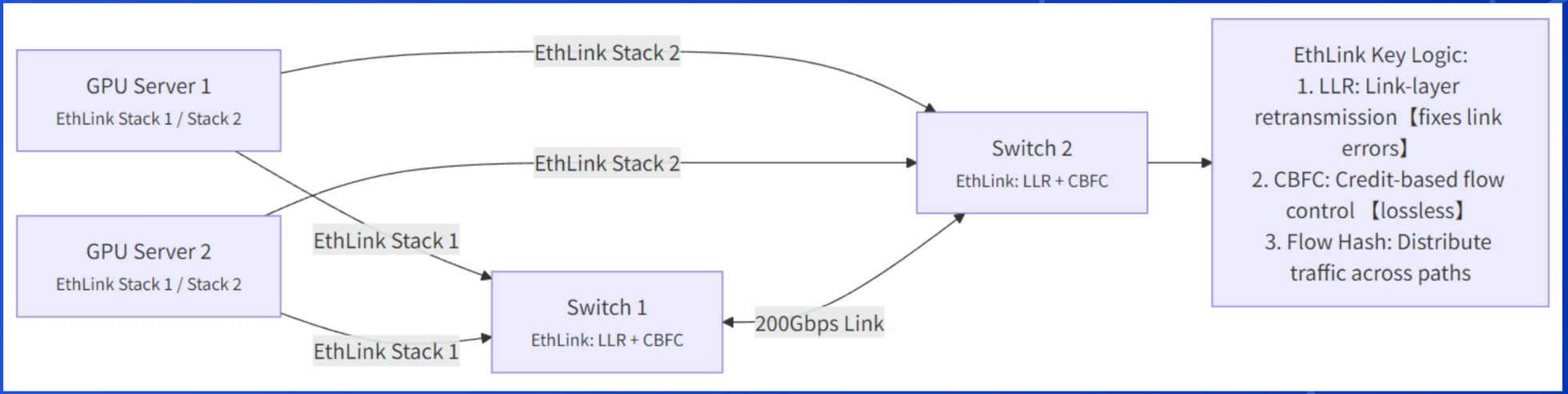






开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

CXL, NVLink, Scale-UP Ethernet, UB —<2>

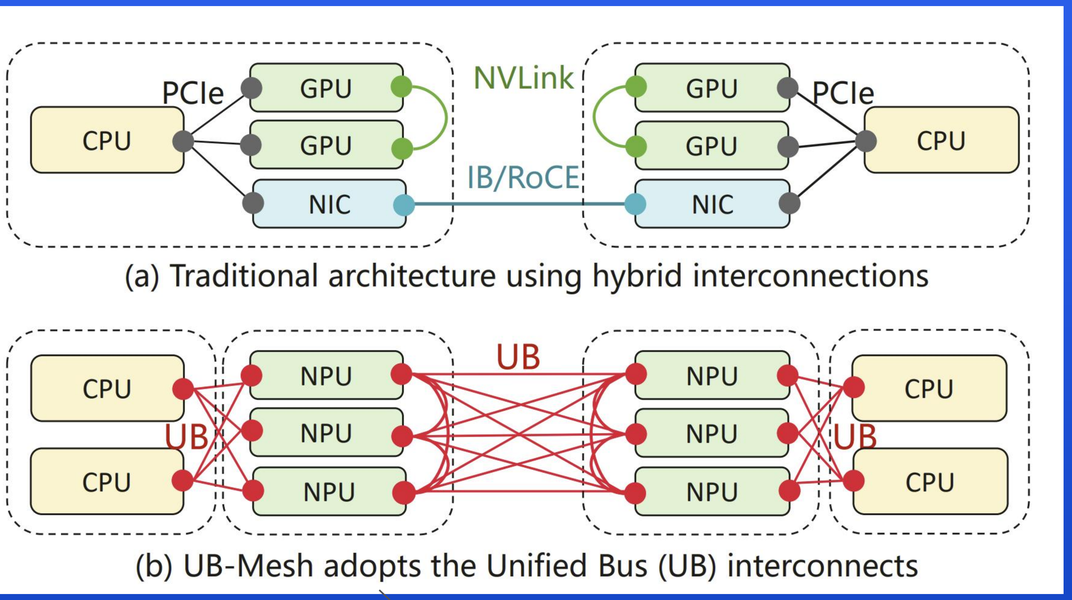


维度	EthLink	ETH-X	SUE
主导者	字节跳动（企业级）	ODCC（行业联盟）	博通
节点规模	单域 1024+（横向扩展）	256 卡全互联（中规模）	10-100（纵向扩展）
协议复杂度	双层优化（保留传统层）	贴近标准以太网（易兼容）	极简链路 + 事务层（无 IP/TCP）
软件依赖	需上层处理乱序	统一编址简化软件	完全硬件化（零软件）
硬件要求	标准交换机 + 多接口 NIC	支持 PAXI 的智能交换机	普通交换机 + 标准 PHY
典型延迟	亚微秒级（100-500ns）	1-5 微秒	1-10 微秒

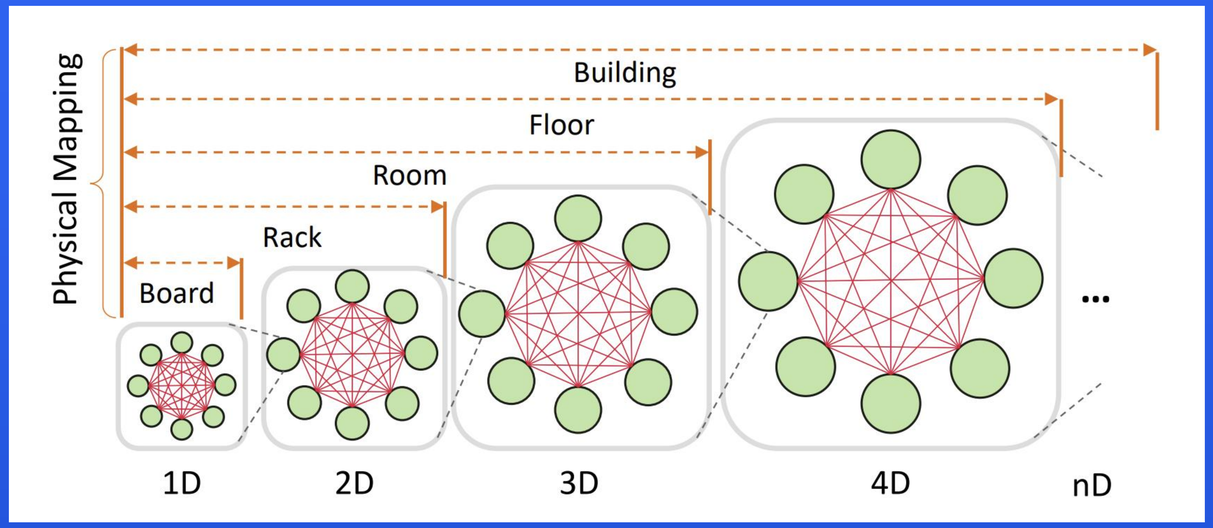
UB — to be announced

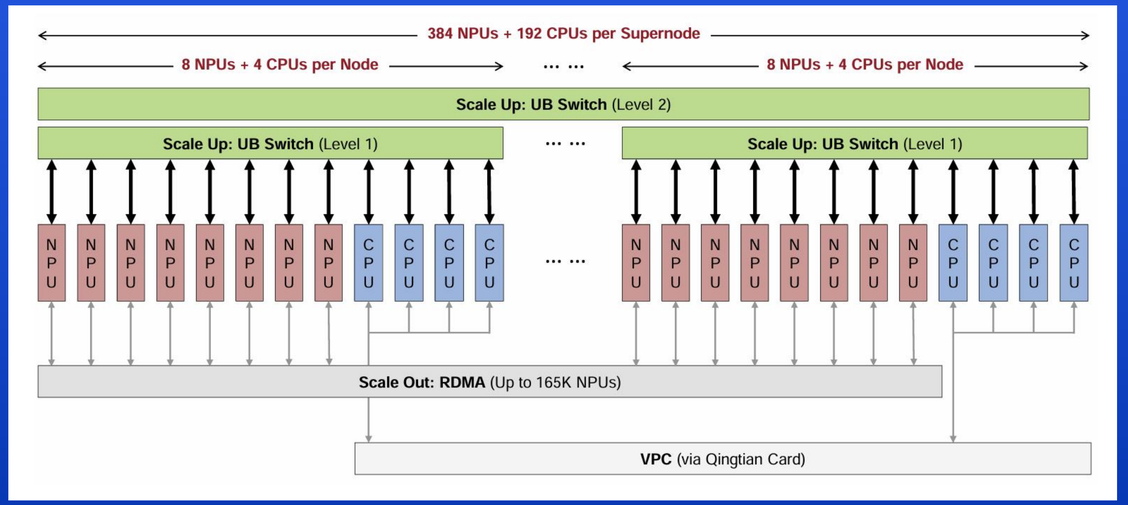
UB – Scale Up & Out 融合总线

UB – Scale Up & Out - Topo

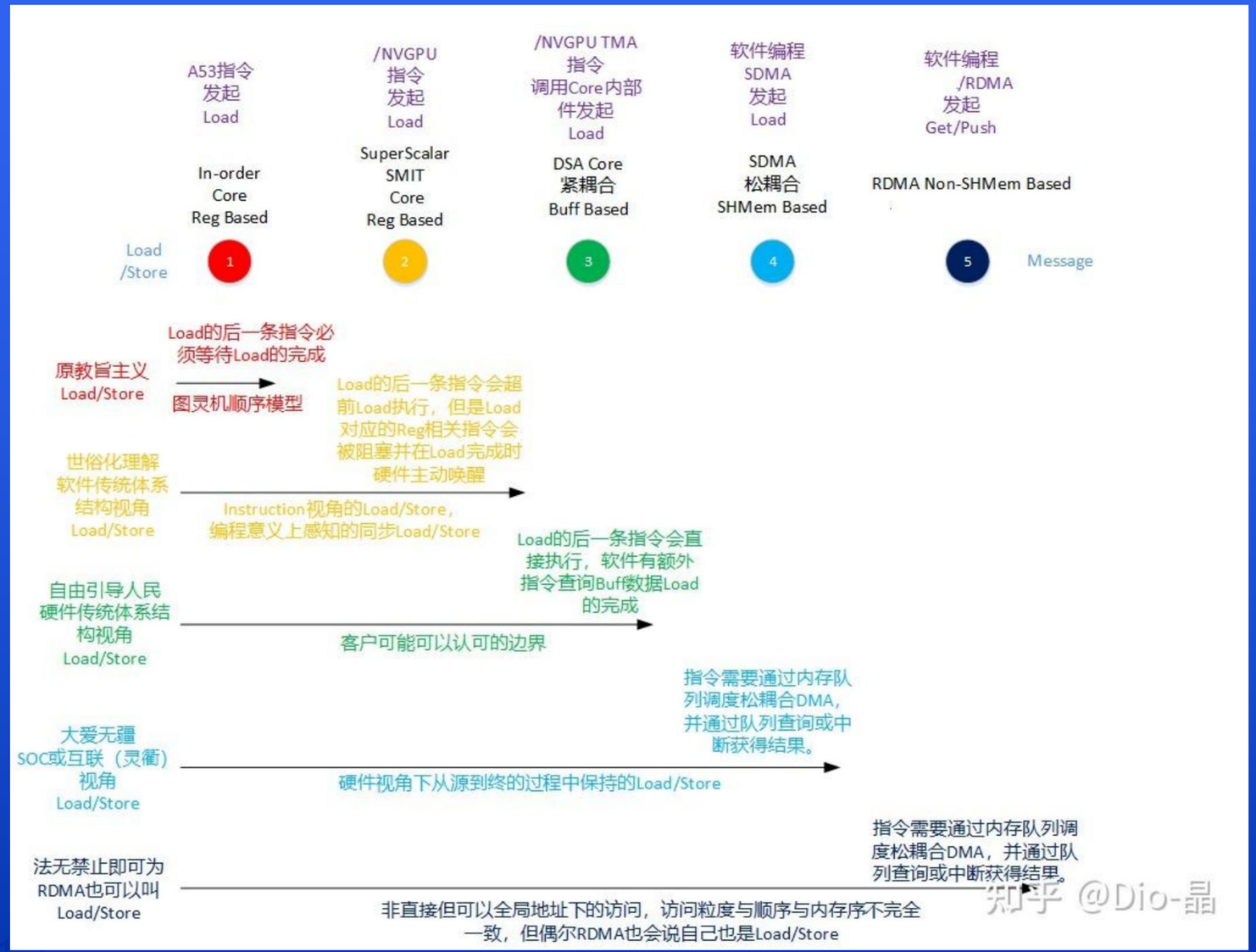


CXL4.0 memory extension ?





内存语义 – Scale Up/Out 界限在哪里?



> https://zhuanlan.zhihu.com/p/1888313027880808563

“

支持内存语义其实是指软件基于内存语义所表达出的丰富的软件行为，能够尽可能多且最低损耗地在Scale-Up系统上运行。

所以一个Scale-Up协议支持load/store这种原生内存访问行为是为了如上的“尽可能多“和“最低损耗”的达成，这是个度的概念，你可以无脑定个70%，或者换句话说，Load/Store更倾向于是Native内存语义的表达。

为了保证更丰富和机制的内存语义在Scale-Up系统上无损的传递，一个好的Scale-Up协议应当不仅仅是Load/Store，还需要定制若干Macro行为来让某些更复杂内存语义在跨机时的高效表达，这是下一个哑谜，也是Ualink/Nvlink还没有做到的事情。

”

内存语义与编程范式讨论

内存语义 – 编程示例

```c
#include <stdint.h>

// 功能：将 src 地址的 len 字节数据拷贝到 dst 地址
void ldst_memcpy(uint8_t *dst,
                 const uint8_t *src,
                 uint32_t len) {
    for (uint32_t i = 0; i < len; i++) {
        // Load: CPU 从 src[i] 读取数据到寄存器
        uint8_t data = src[i];
        // Store: CPU 从寄存器将数据写入 dst[i]
        dst[i] = data;
    }
}

// 调用示例
int main() {
    uint8_t src[1024] = "LD/ST Test Data";
    uint8_t dst[1024] = {0};

    ldst_memcpy(dst, src, sizeof(src));   // 执行拷贝
    return 0;
}
```

```c
#include <linux/dmaengine.h>
#include <linux/dma-mapping.h>

// 功能：通过 DMA 控制器拷贝“内核内存 → 外设缓冲区”（如网卡/SSD）
int dma_memcpy(struct device *dev, void *dst, const void *src, size_t len) {
    struct dma_chan *dma_chan;    // DMA 通道（需硬件分配）
    struct dma_slave_config cfg;  // DMA 配置参数
    dma_addr_t src_dma, dst_dma;  // DMA 可访问的物理地址（非虚拟地址）
    struct dma_async_tx_descriptor *desc;  // DMA 传输描述符

    // 1. 分配 DMA 通道（根据设备类型，如“内存→内存”或“内存→外设”）
    dma_chan = dma_request_chan(dev, "dma0");
    if (!dma_chan) return -ENODEV;

    // 2. 映射虚拟地址到 DMA 物理地址（让 DMA 硬件可见）
    src_dma = dma_map_single(dev, (void *)src, len, DMA_TO_DEVICE); // 源：内存→设备
    dst_dma = dma_map_single(dev, dst, len, DMA_FROM_DEVICE);       // 目标：设备→内存
    if (dma_mapping_error(dev, src_dma) || dma_mapping_error(dev, dst_dma))
        return -EIO;

    // 3. 配置 DMA 传输参数（方向、地址、长度）
    memset(&cfg, 0, sizeof(cfg));
    cfg.direction = DMA_MEMCPY;  // 传输方向
    cfg.src_addr = src_dma;      // 源 DMA 地址
    cfg.dst_addr = dst_dma;      // 目标 DMA 地址
    cfg.src_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES;  // 数据宽度（4字节对齐）
    dmaengine_slave_config(dma_chan, &cfg);

    // 4. 创建并提交 DMA 传输任务
    desc = dmaengine_prep_slave_single(
        dma_chan, dst_dma, len, DMA_MEMCPY, DMA_PREP_INTERRUPT  // 完成后触发中断
    );
    dmaengine_submit(desc);
    dma_async_issue_pending(dma_chan);  // 启动 DMA 传输

    // 5. 等待 DMA 完成（或通过中断回调处理）
    dmaengine_wait_for_completion(dma_chan);

    // 6. 释放资源
    dma_unmap_single(dev, src_dma, len, DMA_TO_DEVICE);
    dma_unmap_single(dev, dst_dma, len, DMA_FROM_DEVICE);
    dma_release_chan(dma_chan);
    return 0;
}
```

```c
#include <infiniband/verbs.h>
#include <stdio.h>
#include <string.h>

--- write to remote side ---
#define REMOTE_IP "192.168.1.100"  // 远程设备 IP
#define MEM_LEN 4096               // 传输数据长度（4KB）

int main() {
    struct ibv_context *ctx;       // RDMA 设备上下文
    struct ibv_pd *pd;             // 保护域（隔离内存区域）
    struct ibv_mr *local_mr;       // 本地内存区域（MR）
    char *local_buf;               // 本地内存缓冲区
    struct ibv_qp *qp;             // 队列对（用于 RDMA 通信）
    struct ibv_send_wr wr = {0};   // 发送工作请求
    struct ibv_sge sge = {0};      // 散射聚集元素（描述内存区域）

    // 1. 初始化 RDMA 设备（获取上下文、保护域）
    ctx = ibv_open_device_by_name(NULL, "mlx5_0");  // 假设使用 mlx5 网卡
    pd = ibv_alloc_pd(ctx);

    // 2. 分配本地内存并注册为 RDMA 内存区域（MR）
    local_buf = malloc(MEM_LEN);
    strcpy(local_buf, "RDMA Test Data");  // 填充待传输数据
    local_mr = ibv_reg_mr(
        pd, local_buf, MEM_LEN,
        IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE  // 权限：本地写、远程写
    );

    // 3. 与远程端交换 MR 信息（需通过 TCP 等协议，此处简化）
    // （实际需发送：local_mr->rkey（远程访问密钥）、local_mr->addr（物理地址））
    struct {
        uint64_t remote_addr;  // 远程内存地址（从远程端获取）
        uint32_t remote_rkey;  // 远程内存访问密钥（从远程端获取）
    } remote_info;
    tcp_exchange_mr_info(REMOTE_IP, &remote_info, local_mr);  // 自定义 TCP 交换函数

    // 4. 配置 RDMA Write 工作请求（写本地数据到远程内存）
    sge.addr = (uint64_t)local_buf;    // 本地内存地址
    sge.length = MEM_LEN;              // 传输长度
    sge.lkey = local_mr->lkey;         // 本地 MR 密钥

    wr.opcode = IBV_WR_RDMA_WRITE;     // 操作类型：RDMA 写
    wr.sg_list = &sge;                 // 关联内存区域
    wr.num_sge = 1;
    wr.wr.rdma.remote_addr = remote_info.remote_addr;  // 远程内存地址
    wr.wr.rdma.rkey = remote_info.remote_rkey;         // 远程 MR 密钥
    wr.send_flags = IBV_SEND_SIGNALED; // 传输完成后发信号

    // 5. 提交 RDMA Write 请求（硬件直接执行）
    ibv_post_send(qp, &wr, NULL);

    // 6. 等待传输完成（通过完成队列 CQ 检查）
    wait_for_rdma_completion(qp);  // 自定义等待函数

    // 7. 释放资源
    ibv_dereg_mr(local_mr);
    free(local_buf);
    ibv_dealloc_pd(pd);
    ibv_close_device(ctx);
    return 0;
}


---- recv side ---

// 核心步骤：分配远程内存 → 注册 MR → 通过 TCP 发送 MR 信息（addr + rkey）
char *remote_buf = malloc(MEM_LEN);
struct ibv_mr *remote_mr = ibv_reg_mr(
    pd, remote_buf, MEM_LEN,
    IBV_ACCESS_REMOTE_WRITE  // 允许远程端写此内存
);
tcp_send_mr_info(LOCAL_IP, remote_mr->addr, remote_mr->rkey);  // 发送给本地端
```


开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

SMP 兼容的共享内存 – 必要性?

```cpp
// 定义共享内存的键值
#define SHM_KEY 1234

void *get_shm_ptr(void)
{
    int shmid = shmget(SHM_KEY, sizeof(std::atomic<int>), IPC_CREAT | 0666);
    if (shmid == -1) {
        std::cerr << "Failed to create shared memory segment." << std::endl;
        return 1;
    }

    void *shm_ptr = shmat(shmid, nullptr, 0);
    if (shm_ptr == (void *)-1) {
        std::cerr << "Failed to attach shared memory segment." << std::endl;
        return 1;
    }

    return shm_ptr;
}

int main(int argc, char *argv[])
{
    void *shm_ptr = get_shm_ptr();
#if 1
    std::atomic<int> *g_atomic_var = new (shm_ptr) std::atomic<int>(0);
#else
    std::atomic<int> *g_atomic_var = (std::atomic<int> *)shm_ptr;
#endif

    std::cout << "g_atomic_var value = "
              << g_atomic_var->load() << std::endl;
    g_atomic_var->fetch_add(5);
    std::cout << "g_atomic_var value = "
              << g_atomic_var->load() << std::endl;

    g_atomic_var->fetch_sub(2);
    std::cout << "g_atomic_var value = "
              << g_atomic_var->load() << std::endl;
    auto old_value = g_atomic_var->exchange(10);
    std::cout << "old_valud: " << old_value
              << ", g_atomic_var value = " << g_atomic_var->load()
              << std::endl;

    auto r2 = g_atomic_var->compare_exchange_strong(old_value, 11);
    std::cout << "r2: " << r2
              << ", g_atomic_var value = " << g_atomic_var->load()
              << std::endl;

    shmdt(shm_ptr);
    shmctl(shmid, IPC_RMID, nullptr);
    return 0;
}
```

可换为：get_global_shm_ptr

Transaction based

SMP style Lock based

```cpp
static auto v0 = 0, v1 = 0;
std::mutex g_lock;

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// 事务函数，用于交换两个整数的值
void swap_a(int& a, int& b) __transaction_atomic {
    int temp = a;
    a = b;
    b = temp;
}

static void thread_f(void)
{
    for (auto i = 0; i < 1024 * 1024; i++) {
#if defined(__TA)
        __transaction_atomic
#else
        std::lock_guard<std::mutex> lock(g_lock);
#endif
        {
            v0++;
            swap(v0, v1);
            v1++;
            swap(v1, v0);
        }
    }
}

int main() {
    int x = 10;
    int y = 20;

    std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;

    // 调用事务函数进行交换
    swap_a(x, y);

    std::cout << "After swap: x = " << x << ", y = " << y << std::endl;

    std::cout << "Before tp: v0 = " << v0 << ", v1 = " << v1 << std::endl;

    std::thread tp[16];
    for (auto &t : tp) {
        t = std::thread(thread_f);
    }

    for (auto &t : tp) {
        t.join();
    }

    std::cout << "After tp: v0 = " << v0 << ", v1 = " << v1 << std::endl;
    return 0;
}
```

亦可为：其他 Bus Native Semantics - TBD

开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

Scale-UP 计算系统中的内存模型 – 如何定义?

比如NVIDIA GPU 的Memory Consistency Model 定义了三种保序模式:

(1) Acquire： 标记为Acquire类型的Load Store，可以看作一个前fence操作，后面的relaxed 类型或者Acquire类型的指令不能被乱序到前面执行。

Acquire

```
cuda::atomic<int> a;

//Prior load
int before = array[0];

//Atomic load
val = a.load(std::memory_order_acquire);

// Later load
int after = array[0];
```

- Loads and stores **cannot** be moved **before** the acquire
- **Same-address ordering** is preserved within a single-thread

(2) Release: 标记为Release类型的Load Store，可以看作一个后fence操作，写在前面的relaxed 类型或者Release类型的指令不能被乱序到后面执行。

Release

```
cuda::atomic<int> a;

//Prior load
int before = array[0];

//Atomic load
val = a.load(std::memory_order_release);

// Later load
int after = array[0];
```

- Loads and stores **cannot** be moved **after** the release
- **Same-address ordering** is preserved within a single-thread

(3) Relaxed: 标记为Relaxed类型的Load/Store 之间可以乱序执行

Relaxed

```
cuda::atomic<int> a;

//Prior load
int before = array[0];

//Atomic load
val = a.load(std::memory_order_relaxed);

// Later load
int after = array[0];
```

- Loads and stores **can** be moved **before** and **after** relaxed
- **Same-address ordering** is preserved within a single-thread

Memory ordering in some architectures[8][9]

Type	Alpha	ARMv7	MIPS	RISC-V		PA-RISC	POWER	SPARC			x86 [a]	AMD64	IA-64	z/Architecture
				WMO	TSO			RMO	PSO	TSO				
Loads can be reordered after loads	Y	Y	depend on implementation	Y		Y	Y	Y					Y	
Loads can be reordered after stores	Y	Y		Y		Y	Y	Y					Y	
Stores can be reordered after stores	Y	Y		Y		Y	Y	Y	Y				Y	
Stores can be reordered after loads	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic can be reordered with loads	Y	Y		Y			Y	Y					Y	
Atomic can be reordered with stores	Y	Y		Y			Y	Y	Y				Y	
Dependent loads can be reordered	Y													
Incoherent instruction cache pipeline	Y	Y		Y			Y	Y	Y	Y	Y		Y	

知乎 @知返

\+ Bus Order = ?

> https://zhuanlan.zhihu.com/p/1922796547814458461

讨论 & 展望

THANKS