GMP

- Sync And Async
- AI加速芯片上的2D单元
- GMP
- 架构/微架构
- 架构_微架构
- 算力单元
- DynamicGraphMultiProcessor架构
- 微架构文档规范

Sync And Async

?????

1. 多种类型的硬件单元需要进行同步
2. 不确定的循环次数
3. N to N的同步需求
4. 无缝的同步，无缝的并行
5. 频繁的同步需求，频繁的状态pulling，低latency
6. 灵活的抽象适应所有的同步需求
    1. transformer的Flash-attention就需要在L1内做多次fusion，不是简单的DMA和算力的同步
7. 方便的软件使用

??Global??????????????

1. 软核实现
2. 类似一个全局锁，成为性能瓶颈点
3. 调度颗粒度大，难以实现精细化的控制
    1. DMA、Kernel之间的同步
4. 难以做到无缝的调度

??????????

1. 采用专有的，多层级的Sync硬件，自带计数和复位功能，配合mailbox实现N to N异步依赖
2. 可以实现复杂的producer consumer模型Pipeline
3. 软件编写困难
    1. 软件需要根据需求管理全局硬件资源得使用
    2. 不同得同步需求，使用不同得同步硬件，增加软件复杂性
4. 中心化的控制模式，灵活度不够，不能实现一些比较精细得控制
5. 性能不是最理想
    1. 虽然可以做到比较快的同步，但是每次发同步命令（write mbx，wait mbx）都要经过多次写总线并ack，latency和带宽占用都不理想

?????????? mbarrier

1. 去中心化，每个硬件单元都可以同步别人和被别人同步，每个单元都能实现对应得标准得M barrier的功能
    1. 提高灵活性
    2. 统一的同步方式，简化设计和使用
2. 传输数据的同时，传递同步信号
    1. 数据传递带上依赖信息，目的存储器可以识别依赖信息，并正确得处理，而不用返回数据发送方
    2. 避免没有必要的fence
    3. 改进前Core0 Store() Core0 Fence() memory Ack() Core0 Signal(Core1 mbx)

    4. 改进后Core0 Store_mbarrier() Core1 Wait(mbarrier)
3. 灵活性
    1. 每一轮的同步都可以根据实际运算数据的大小，动态设置需要的同步单元和数据搬运数量
4. 性能好
    1. 每个单元只用wait自己local的mbarrier单元的counter信号就可以保证同步，latency小
    2. 需要的信号transaction最小化

AI??????2D??

1. 卷积天然的数据复用度是Dot的9倍，对于芯片的压力更小
2. 算力缩放是一个非常重要的问题，涉及架构各代之间的稳定性，保护客户的价值
3. L1/L2/L0 怎么支持reshape或者swizzel
4. 不同的layout（NHWC等）以及BPI BPK FF
5. 不同数据精度的支持和混合计算
6. L0 L1 Fusion的支持
7. 存储的mapping 利于运算和fusion
8. feature*weight vs weightT*featueT

?2D Dot?????

1. 指令表达为小尺寸，对不同的计算需求（1D、2D）的fusion比较友好
    1. 但是小尺寸，需要每个单元的L0进行broadcast来复用给其他单元，节省L1的带宽需求，增加了2D的复杂度
    2. 小尺寸对load/store的要求也比较高
    3. 需要**2D**的计算表达有一个好的抽象，同时配合非常精密的同步，解决上述问题
    4. 为了在各代的代码不发生变化，需要硬件/编译器有自动合并/优化的能力
2. cuda因为需要兼容simt的编程，复用warp内thread的寄存器，同时为了拼接更多的硬件做一个更大的2D（为了PPA好）
    1. 整体上单个thread表达的数据计算尺寸比较小
    2. 硬件自动进行合并成大的2D指令
    3. 硬件自动处理同步
    4. 硬件自动处理数据搬运
    5. 因为需要协调多个sm subcore，甚至是block group，灵活性一样是个大问题
3. XXU4.0利用多个thread之间的手动给的footprint order信息来组织多个thread的数据共享
    1. 左数每次都从L1load重复load，右数缓存在L0，并且各个subcore复用
4. 2D-1D-2D的Fusion
    1. 涉及到多次计算后结果的数据精度（位宽）不匹配的问题，需要合并和拆封来保持操作数的宽度一致
        1. 8bit int8 需要64个组成512bit，而32bit只要16个
    2. 不同的操作的输入输出，可能需要有装置的操作
    3. layout的区别：NCHW NHWC
    4. 卷积的stride不等1
5. 预测：2D算力会无限的膨胀下去，1D和2D的比例会比较固定
6. 精细化控做到L1，则所有的抗latency的容量就会开在L1，消耗：L1_base x latency，如果是在L0，则L0_base比L1_base小很多。节省很多面积
7. **swizzle**和**renaming**都可以解决**L0**的**bank conflict**，**renaming**会消耗大量的面积

????

1. 算力和存储器（SRAM）都不能做到单体很大
    1. 很大的单体有利于软件使用方便
    2. 多个小的存储器实例（bank）组成一个大的抽象存储器
2. 物理单元之间很难方便做到cycle by cycle的同步，必须使用额外的同步逻辑
3. 硬件电路虽然可以并行，理论算力很高，但是latency是必然存在的

NV Blackwell

从 PTX、SASS可以看到AI面对的问题和很多解决方法都是共同的。

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#tensorcore-5th-generation-family-instructions https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#blackwell-instruction-set

1. L0的 memory变化
    1. 显式的手动管理tensor memory，是不是2D/1D更像是独立的一个NPU/DSA单元?
    2. Tensor计算直接使用专门的Tensor memory，做 memory to memory的 Tensor计算，且支持软件动态分配
    3. 1D 可以直接load/store这块 memory
    4. 这样让L0可以动态，目的估计是破除2D/1D 每写一种L0 fusion就要写一份code 示例，Fusion算子可以隔离开发
2. 更大的矩阵形状128/256，降低2D带宽需求
3. 支持TPC 内 2个 SM之间的2D 协同计算，看起来是2个 core支持的读数据复用
4. 支持OCP-MX的 micro-scaling
5. 标量thread issue Tensor计算指令，不再是SIMT style的多个thread
6. 直接支持卷积指令，支持weight-stationary GEMM，带上mask bit 表达哪些是padding 的 0

GMP

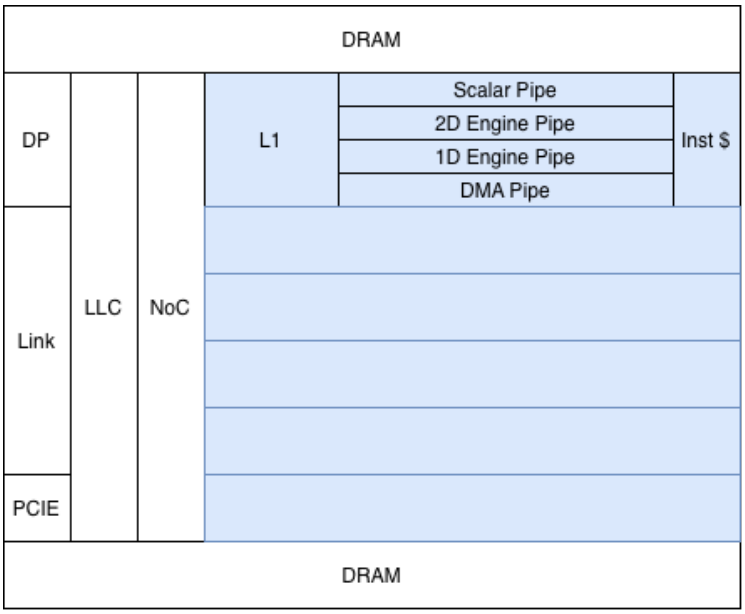
1. 背景
    1. 适应未来的AI计算需求
        1. 存储足够量的权重，但是明显的热点内容访问
        2. 强动态性，大范围、多次的随机动态访问
        3. 节能、低带宽需求，高效率数据流
        4. 低延迟
    2. 软件定义硬件，硬件尽量简单、透明、底层抽象、灵活
2. 目标
    1. 软硬件结合
        1. 大量依靠软件优化，发挥最大的物理效率，同工艺下架构效率达2倍
        2. 算法和硬件协同优化，同模型精度条件下效率达4倍
            1. DRAM或者多机的不确定数据延迟直接整合到算法处理，硬件不做竞争
            2. 硬件采用固定的LUT计算（可能不能等价到矩阵乘法，甚至是乘法本身）
            3. 全模型网络级别优化，利用编译器对整个模型进行搜索优化，生成静态计算图
        3. 指令控制流水线
            1. 指令明确指定指令的调度、**L0 Cache**的使用、依赖关系的建立和解除
    2. 原生支持动态算法：
        1. MoE
        2. 动态算法
    3. 统一的异步通讯管理方案
        1. 整个系统有大量的不同的通讯和同步机制
            1. 流水线内的credit，L1的数据缓冲
            2. 算力核之间的数据交互核同步，NoC的各种协议
            3. L2/L3的复用
            4. 分布式栈：网卡的片上调度，网络的延迟不确定性，通路的复用
        2. outstanding/各种缓存的管理和设计
        3. 异步的调度
        4. launch控制及指令加载加速
    4. 统一的数据流拆分模型
        1. 提供统一的编程模型对数据流进行描述
        2. 硬件加速的数据流动态计算，减少冗余且高成本的除法/模运算的地址计算
            1. 通过自动的预计算和特殊硬件加速
        3. 自动处理的原子操作，以消除写入全局内存时的warp级串行化
        4. 自动进行乒乓双缓冲机制
    5. 权重和临时数据（KV Cache ..）分离的架构
        1. 权重使用固定的通路
        2. 两种数据使用不同的数据流模版进行设计
    6. 算力
        1. **Binary Lut** 方案，减少算力能耗和面积，降低数据搬运量?
        2. **CIM** 专用权重通道，近存架构，存内架构，存内计算?
    7. 1D动态算力架构

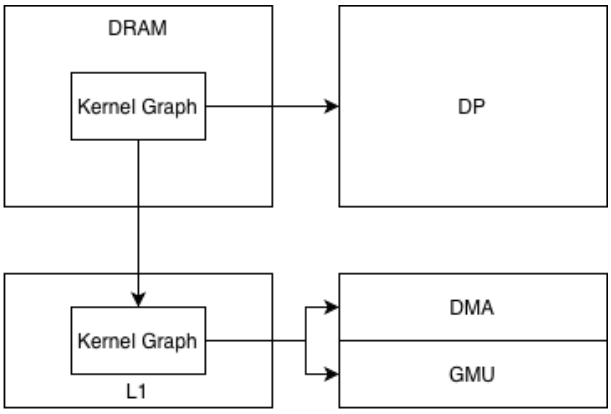
1. 一维计算阵列，动态性体现在一维的长度上，对应到 sequence的长度
2. 二维数据复用，数据通道提供精密的排布和数据复用的调度
3. 编译出向量指令流（LD MUL ST），支持批量动态配置，两块指令流之间流水可以重叠
   ---> DSA?
4. 例化固定数量的 L1读写计数器，用于同步 dma和 Mac，dma和 Mac，自动从 l1 加载指令
5. 问题:
   1. 1D指令表达大算力 2D
      1. VLD VST MLD MST VMUL VMUL_reduce MUL_join 等等 指令
   2. LD/ST的自动 mbarrier
      1. 软件管理 L1的所有 bank，每个 bank的每个地址都记录一个 count，启动初始化的时候初始化 count

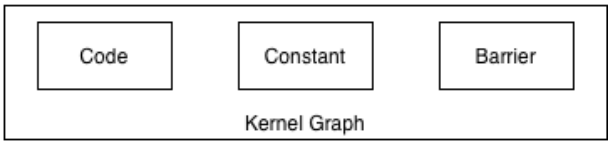
3. 架构考虑
   1. 动态性的表达
   2. 硬件竞争的管理
   3. 灵活性，扩展性，从 edga到集群
   4. 自举，所有单元支持自配置，自启动
   5. 线程内的依赖都是静态的软件调度，软件直接调度流水线，减少硬件的调度
4. 架构方案
   1. 图：编译整个动态计算图，支持 Fork（launch） join sync
   2. 平铺：按照可编程硬件单元进行编程，为每个单元生成一定数量的逻辑线程，支持 sync
   3. … …
5. 规格
6. 指令流
   1. load和 fetch
   2. 基于图的信息，和数据流一样得方式，需要发命令和同步
7. ISA
8. Launch
9. Sync

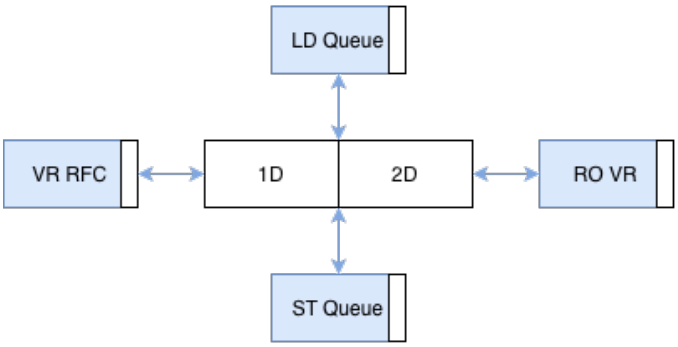
总结

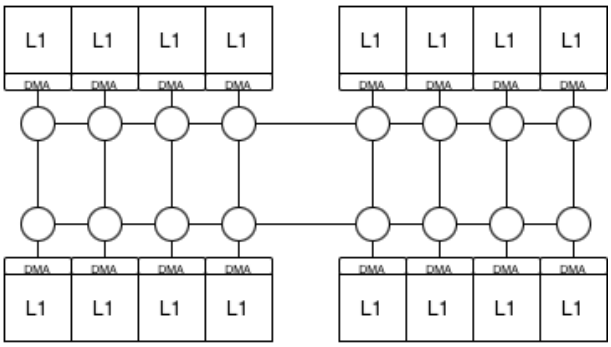
1. 灵活、低延迟的异步 /同步机制
2. 简单、透明的硬件拓扑，软件的深度控制

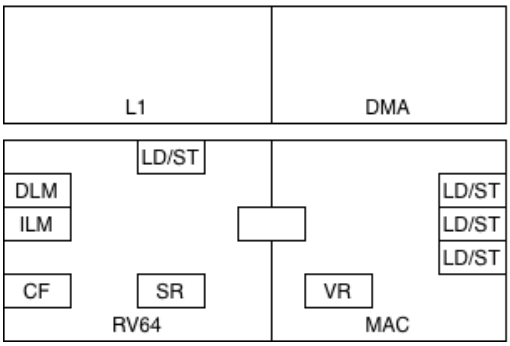

DRAM
DP
L1
Scalar Pipe
2D Engine Pipe
1D Engine Pipe
DMA Pipe
Inst $
Link
LLC
NoC
PCIE
DRAM


DRAM
Kernel Graph
DP
Kernel Graph
L1
DMA
GMU


Code
Constant
Barrier
Kernel Graph


LD Queue
VR RFC
1D
2D
RO VR
ST Queue


L1
DMA


L1
DMA
LD/ST
DLM
ILM
CF
SR
RV64
VR
MAC
LD/ST
LD/ST
LD/ST

??/???

??

1. 标量寄存器和向量寄存器统一，支持自动进行转换
2. 异步单元（SP-PU-L1-DMA）之间都采用异步机制，依赖转移到异步目标
    1. 统一的同步机制
    2. 静态分配同步资源
    3. 原生软硬件支持动态图的执行
    4. LD/ST 避免使用 fence功能
3. Launch：fork
    1. 资源初始化（同步资源，各种存储器，状态）
4. launch/signal/wait:join
    1. launch pu instrution：write_back_id local_id
        1. write_back atomic add/sub
    2. wait instruction: local_id
        1. local_id GE LE counter
    3. wait remote instruction: remote_id
        1. local_id GE LE counter
5. 灵活性&性能
    1. 支持灵活的数据尺寸
    2. 支持灵活的算子
        1. 混合精度的 scale
        2. 1D/2D并行
        3. 全局 Reduce算子，排序算子
            1. L3 Atomic
    3. 支持灵活的本地数据复用，尽量减少数据搬运数据
    4. 尽量避免算力浪费
        1. leading tailing latency
            1. LD/ST inflight delay，DMA Delay
                1. ping-pong
                2. pipeline
            2. kernel启动 latency
                1. kernel预加载
            3. kernel间的 gap
                1. kernel间的本地数据复用?
                2. 多个算子的 fusion
        2. 数据没有 ready、需要 fence引起的气泡
            1. 无缝的同步机制
                1. m-barrier
        3. 不对齐、尾数

MAC

1. Vector计算单元支持乱序和并行
2. 软件编译指定明确的VR寄存器依赖，RO WO属性
3. 自动拆解大指令成Vector指令，并行执行
4. 软件静态排布VR寄存器，生成依赖关系，申请和释放管理
5. Vector指令支持标量对其进行动态配置

Software

1. 使用独立的硬件仿真软件，不依赖硬件仿真运行

ISA

1. 标量
    1. RV64i
2. 向量
    1. VLD VST VMUL VADD REDUCE_ADD REDUCE_MAX REDUCE_MIN VMUL_REDUCE_ADD
    2. MLD MST
3. 张量
    1. GEMM
        1. 左/右数: 128个 =7位 + 3位扩展矩阵 = 10位
        2. 输出: 7位
        3. Opcode
4. fence
    1. L1 cache line 计数
5. VR
    1. 软件管理VR data hazard? VR之间的依赖?
        1. 增加指令的表达信息
    2. 软件分配VR 还是硬件rename ing，解决bank冲突?
    3. 利用VR count? 软件进行管理依赖关系?
        1. LD ST 计算的三类指令之间可以并行，通道内部没有必要并行
            1. 因为硬件资源没有特殊性，不会因为并行而减少气泡
        2. 默认，GEMM指令一定要在前面LD指令之后执行
        3. 默认，ST指令一定要在前面计算指令之后执行
6. L1
    1. 软件管理cache line
    2. **cache line**硬件计数，自动异步等待
    3. 针对**L1 CacheLine**的编程?
        1. 软件指定**Load**到**L1 cache line**的位置和有效长度
        2. 向量指令按照**cache line**的粒度和**mask**来执行指定的计算
        3. 这整个流程都是提前编译好，从**L1-L1**都是提前确定的
        4. 针对不同**MNK**大小需求，可以通过标量指令来快速配置，支持动态性
7. DMA

1. 附带一片ROM空间，可以在Launch Kernel的时候主动加载一片指令编码，和当前kernel绑定的
2. 支持一个cmd端口，接受其他核心的控制命令
3. DMA作为对外的接口，通过配置kernel的linear copy kernel作为launch的起点？
    1. 简化系统复杂度，通用的功能组件，避免开发专用的launch constructor
8. 增加到128个标量寄存器支持RV64扩展?
9. 增加指令流控制
10. 增加配合/加速向量单元的定制指令?通过兼容RiscV-V的指令来实现?？

??_???

??/???

??

1. 标量寄存器和向量寄存器统一，支持自动进行转换
2. 异步单元（SP-PU-L1-DMA）之间都采用异步机制，依赖转移到异步目标
    1. 统一的同步机制
    2. 静态分配同步资源
    3. 原生软硬件支持动态图的执行
    4. LD/ST避免使用fence功能
3. Launch：fork
    1. 资源初始化（同步资源，各种存储器，状态）
4. launch/signal/wait:join
    1. launch pu instrution: write_back_id local_id
        1. write_back atomic add/sub
    2. wait instruction: local_id
        1. local_id GE LE counter
    3. wait remote instruction: remote_id
        1. local_id GE LE counter
5. 灵活性&性能
    1. 支持灵活的数据尺寸
    2. 支持灵活的算子
        1. 混合精度的scale
        2. 1D/2D并行
        3. 全局Reduce算子，排序算子
            1. L3 Atomic
    3. 支持灵活的本地数据复用，尽量减少数据搬运数据
    4. 尽量避免算力浪费
        1. leading tailing latency
            1. LD/ST inflight delay，DMA Delay
                1. ping-pong
                2. pipeline
            2. kernel启动latency
                1. kernel预加载
            3. kernel间的gap
                1. kernel间的本地数据复用?
                2. 多个算子的fusion
        2. 数据没有ready、需要fence引起的气泡

        1. 无缝的同步机制
            1. m-barrier
    3. 不对齐、尾数

MAC

1. Vector计算单元支持乱序和并行
2. 软件编译指定明确的 VR寄存器依赖， RO WO属性
3. 自动拆解大指令成 Vector指令，并行执行
4. 软件静态排布 VR寄存器，生成依赖关系，申请和释放管理
5. Vector指令支持标量对其进行动态配置

Software

1. 使用独立的硬件仿真软件，不依赖硬件仿真运行

ISA

1. 标量
    1. RV64i
2. 向量
    1. VLD VST VMUL VADD REDUCE_ADD REDUCE_MAX REDUCE_MIN VMUL_REDUCE_ADD
    2. MLD MST
3. 张量
    1. GEMM
        1. 左 /右数: 128个 =7位 + 3位扩展矩阵 = 10位
        2. 输出: 7位
        3. Opcode
4. fence
    1. L1 cache line 计数
5. VR
    1. 软件管理 VR data hazard? VR之间的依赖?
        1. 增加指令的表达信息
    2. 软件分配 VR 还是 硬件 rename ing，解决 bank冲突?
    3. 利用 VR count? 软件进行管理依赖关系?
        1. LD ST 计算 的三类指令之间可以并行，通道内部没有必要并行
            1. 因为硬件资源没有特殊性，不会因为并行而减少气泡
        2. 默认， GEMM指令一定要在前面 LD指令之后执行
        3. 默认， ST指令一定要在前面计算指令之后执行
6. L1
    1. 软件管理 cache line
    2. **cache line**硬件计数，自动异步等待
    3. 针对 **L1 CacheLine**的编程?
        1. 软件指定 **Load**到 **L1 cache line**的位置和有效长度

   2. 向量指令按照**cache line**的粒度和**mask**来执行指定的计算
   3. 这整个流程都是提前编译好，从**L1-L1**都是提前确定的
   4. 针对不同MNK大小需求，可以通过标量指令来快速配置，支持动态性
7. DMA
   1. 附带一片ROM空间，可以在Launch Kernel的时候主动加载一片指令编码，和当前kernel绑定的
   2. 支持一个cmd端口，接受其他核心的控制命令
   3. DMA作为对外的接口，通过配置kernel的 linear copy kernel作为launch的起点 **? **
      1. 简化系统复杂度，通用的功能组件，避免开发专用的launch constructor
8. 增加到128个标量寄存器 支持RV64扩展?
9. 增加指令流控制
10. 增加配合/加速向量单元的定制指令? 通过兼容RiscV-V的指令来实现? ?

????

????????

把[[Pipe]]第6节§208的算力单元抽象展开到RTL起手前。每个计算实例 = 一条engine pipe实例。

0. ? DMA ?????

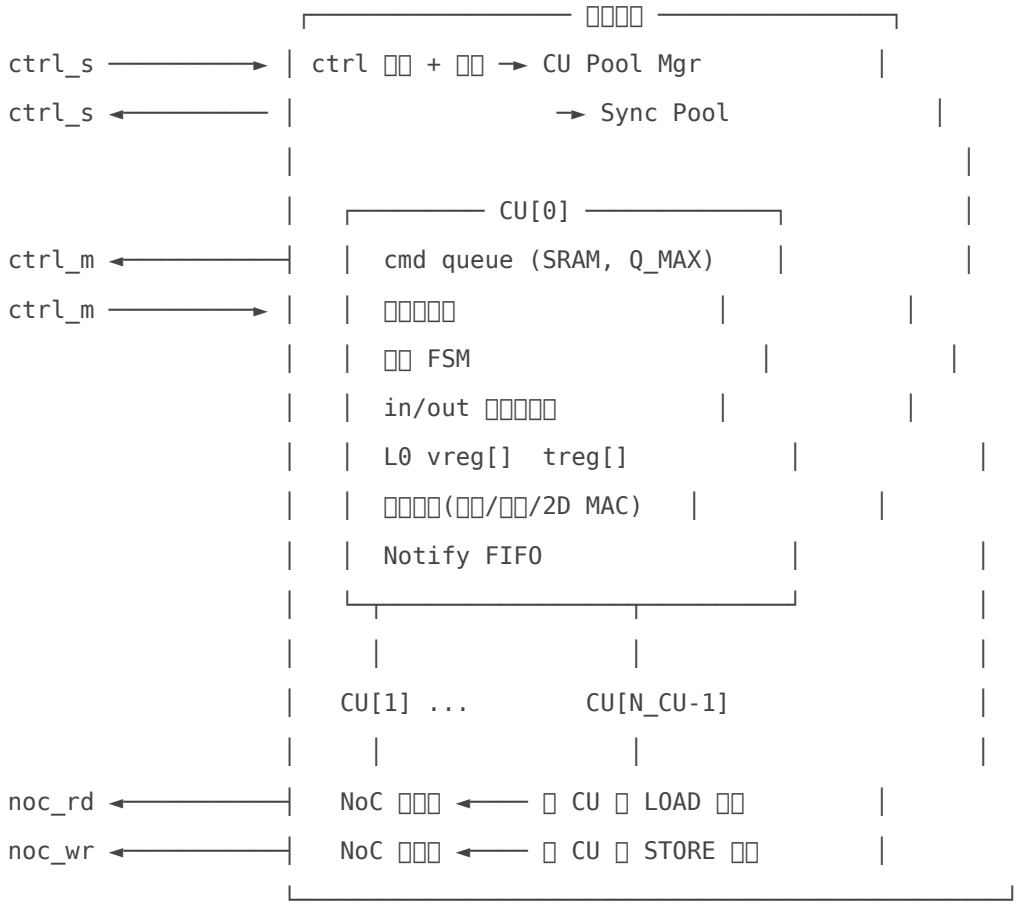
算力单元与[[DMA]]结构同构：都是"engine pipe集合 + sync pool + 控制接口 + 共享数据fabric master"。差别仅在执行通路：

	DMA	算力单元
执行通路	读 → transform → 写	读 → 计算阵列 → 写
数据端	被动单元（L1、DRAM）	被动单元（L1、DRAM）
内部状态	transform buffer	L0寄存器组
cmd_desc专属字段	xform_flags, xform_params	compute_op, loop, dtype, accum

下文凡是结构与DMA一致的，引用[[DMA]]不重复展开；只描述差异。

1. ????

参数	含义	默认	备注
`N_CU`	计算实例数（每个 = 一条engine pipe）	4	可异质（见§13）
`N_SYNC`	sync counter数	32	
`Q_MAX`	单实例命令队列硬件上限	16	2的幂
`K_IN / K_OUT / K_SIG`	cmd_desc列表上限	4/4/4	
`N_L0_VEC`	每实例L0向量寄存器数	8	
`N_L0_TEN`	每实例L0张量寄存器数	4	
`VEC_LANE`	向量阵列lane数	16	INT8 lane
`MAC_M / MAC_K`	2D MAC阵列尺寸	8 / 8	systolic
`MAC_ACC_W`	MAC累加器位宽	32	INT32
`OUT_TYPE_SET`	可选输出位宽	{8, 16, 32}	round/sat软件控
`LOOP_W`	inner-loop计数器位宽	16	



参数	含义	默认	备注
`STRIDE_DIM / STRIDE_W / SHAPE_W`	同 [[DMA]]	3 / 32 / 24	
其他 `COUNT_W / HANDLE_W / ADDR_W / NOC_W`	同 [[DMA]]		

2. ????

```
                          ┌──────────────────── 算力单元 ─────────────────┐
ctrl_s ────────────► │ ctrl 解码 + 路由 ─► CU Pool Mgr              │
ctrl_s ◄──────────── │                   ─► Sync Pool              │
                     │                                             │
                     │    ┌──────── CU[0] ────────┐                │
ctrl_m ◄─────────────┤    │ cmd queue (SRAM, Q_MAX) │                │
ctrl_m ────────────► │    │ 完成计数器              │                │
                     │    │ 执行 FSM                │                │
                     │    │ in/out 地址生成器       │                │
                     │    │ L0 vreg[]  treg[]       │                │
                     │    │ 计算阵列(标量/向量/2D MAC) │             │
                     │    │ Notify FIFO             │                │
                     │    └──┬──────────────────┬───┘                │
                     │       │                  │                    │
                     │   CU[1] ...         CU[N_CU-1]                │
                     │       │                  │                    │
noc_rd ◄─────────────┤   NoC 读仲裁 ◄──── 各 CU 的 LOAD 通路          │
noc_wr ◄─────────────┤   NoC 写仲裁 ◄──── 各 CU 的 STORE 通路         │
                     └───────────────────────────────────────────────┘
```

3. ????

完全复用 [[DMA]] §3 的端口。差别:

- `noc_rd_*` / `noc_wr_*` 可对接任意被动单元（L1、DRAM），目标由 cmd_desc 内的 in/out handle.unit_addr 决定
- `ctrl_s_*` / `ctrl_m_*` 协议完全一致

4. cmd_desc ????

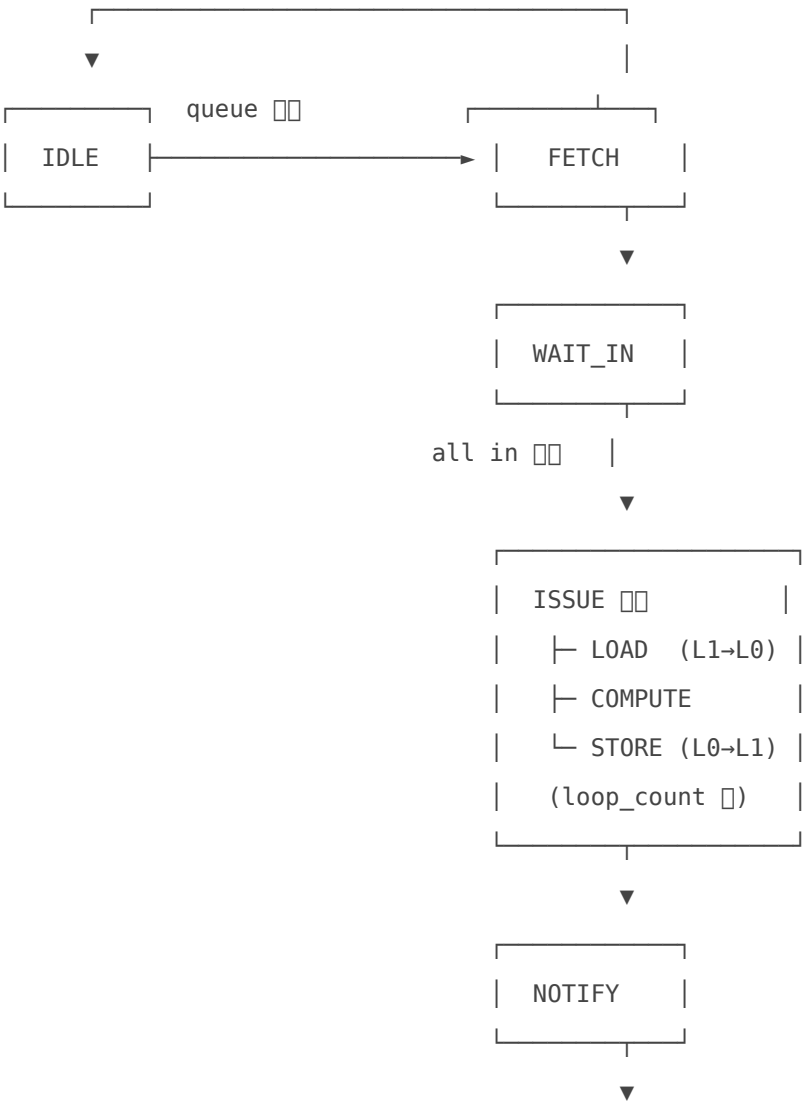
总宽 ≤ 1024 bit。与 [[DMA]] §4 同构，差别在 `op + 专属参数区`。

偏移	位宽	字段	说明

0	4	`op`	0=SCALAR_ALU, 1=VEC_ALU, 2=VEC_REDUCE, 3=MAC_2D, 4=ELEMENTWISE_ACT(ReLU/GELU), 5=SOFTMAX_PARTIAL, ...
4	4	`dtype_in`	0=I8, 1=I16, 2=I32, 3=FP8(预留), 4=FP16(预留)
8	4	`dtype_out`	同上
12	2	`accum_mode`	0=覆盖，1=累加（C += A*B），2=累加+饱和
14	2	`act_kind`	ELEMENTWISE 时的激活函数选
16	16	`loop_count`	inner-loop 次数；硬件按此自动 step 地址生成器
32	48	`src_a_base`	byte_addr（落在某条被动单元 pipe 内，见 [[L1]] §4.1 / [[DRAM]] 地址映射）
80	96	`src_a_stride[3]`	
176	72	`src_a_shape[3]`	
248	48	`src_b_base`	MAC_2D / VEC_ALU 用
296	96	`src_b_stride[3]`	
368	48	`dst_base`	
416	96	`dst_stride[3]`	
512	72	`dst_shape[3]`	
584	8	`l0_a_reg`	A 操作数落到哪个 L0 寄存器
592	8	`l0_b_reg`	
600	8	`l0_c_reg`	结果累加目标
608	4+4+4+4	`n_in / n_out / n_sig / _`	
624	96	`in_list[4]`	每项 {handle:16, delta:8}
720	96	`out_list[4]`	
816	96	`sig_list[4]`	
912	112	reserved/padding	

`src_b_*` 仅对二元算子（MAC / VEC_ALU）有效；一元算子（激活、reduce）写 0。

5. ??????



完全同 [[DMA]] §5。多 beat `PSEND` 同样 8 × 128 bit。opcode 表共享。

6. CU Pool ???

结构同 [[DMA]] §6。差别:

- `meta[N_CU]` 多一个 `cu_class` 字段（标量 / 向量 / MAC，若异质例化）
- PALLOC 时 `params.cu_class` 必须与本实例 class 匹配；不匹配视为软件错（行为未定义）

7. CU ?????×N_CU?

7.1 ????

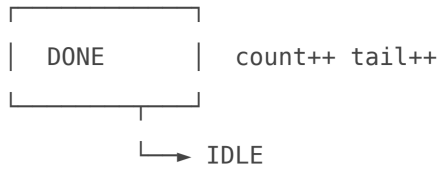
完全同 [[DMA]] §7.1。

7.2 ?????

完全同 [[DMA]] §7.2。

7.3 ?? FSM

```
              ┌──────────────────────────────────────┐
              ▼                                      │
        ┌──────────┐  queue 非空              ┌──────┴───────┐
        │   IDLE   ├─────────────────────────►│    FETCH     │
        └──────────┘                          └──────┬───────┘
                                                     ▼
                                              ┌──────────────┐
                                              │   WAIT_IN    │
                                              └──────┬───────┘
                                         all in 到位 │
                                                     ▼
                                              ┌──────────────────────┐
                                              │   ISSUE 流水          │
                                              │    ├─ LOAD  (L1→L0)  │
                                              │    ├─ COMPUTE        │
                                              │    └─ STORE (L0→L1)  │
                                              │   (loop_count 次)    │
                                              └──────┬───────────────┘
                                                     ▼
                                              ┌──────────────┐
                                              │   NOTIFY     │
                                              └──────┬───────┘
                                                     ▼
```



状态	时长	行为
IDLE	1 cycle	
FETCH	1 + (STRIDE_DIM-1) cycle	读 cmd_desc + 预算 stride wrap_delta（同 DMA §7.4）
WAIT_IN	不定	顺序 COUNT_QUERY in_list
ISSUE	流水，见 §7.6	
NOTIFY	(n_in + n_out + n_sig) cycle	入 Notify FIFO
DONE	1 cycle	

7.4 L0 ????

项	规格
`vreg[N_L0_VEC]`	每个 `VEC_LANE × dtype_in_bits` 宽；存 vector 操作数 / 结果
`treg[N_L0_TEN]`	每个 `MAC_M × MAC_K × dtype_in_bits` 宽；存 tile 操作数
端口	vreg: 2R/1W； treg: 2R/1W（FMA 类够用）
复位	不复位（undefined），软件必须先 LOAD 才用
共享	实例私有，不跨 CU 共享

实现： vreg 用 register file（small）， treg 用 SRAM macro（一个 treg ≈ 8×8×8 = 512 bit，N_L0_TEN = 4 时 2 KiB）。 treg SRAM 端口配 2R/1W 满足 MAC 2 输入 1 累加。

vreg 与 treg 物理分离 —— 二者访问宽度差很大。

7.5 ????

三类阵列共存（每个实例可裁掉某类，按 cu_class 决定）：

7.5.1 ??

- 1 lane，宽度 `max(dtype_in, dtype_out)` ≤ 32 bit
- op： ADD/SUB/MUL/SHIFT/CMP/SELECT/BIT
- latency 1 cycle， throughput 1/cycle

7.5.2 ??

- `VEC_LANE` 路并行 INT8 ALU + 16-bit accumulator option
- op： lane-wise ADD/SUB/MUL/MAC/CMP，跨 lane REDUCE_SUM/MAX/MIN
- latency 1-2 cycle， throughput 1/cycle
- elementwise activation： LUT 表（ 4 KiB ROM/SRAM）实现 ReLU / GELU / sigmoid 等

7.5.3 2D MAC

- `MAC_M × MAC_K` systolic 阵列
- 单元: INT8 × INT8 → INT16 partial → 累加进 MAC_ACC_W 寄存器
- 输入: A treg (M × K), B treg (K × N, 本设计 N = MAC_M); 权重和激活方向可在 cmd_desc 内对调
- 输出: C treg (M × N), 累加进已有 treg（accum_mode）
- pipeline depth = `MAC_K + 2` cycle
- 每 cycle 注入一行 A、一列 B; MAC_M = MAC_K = 8 时 8 cycle 填满, 第 9 cycle 起每 cycle 一个稳态结果

下溢 / 上溢 / round: cmd_desc.dtype_out 与 accum_mode 决定输出阶段的 saturate + truncate / round_to_nearest。专用硬件块 (输出后处理 stage) 实现。

7.6 LOAD/COMPUTE/STORE inner-loop ??

把一条 cmd 拆成 `loop_count` 次 micro-op, 三级流水:

```
cycle:  N    N+1   N+2   N+3   N+4   N+5
        |    |     |     |     |     |
LOAD    op0  op1   op2   op3   op4   op5
COMPUTE      op0   op1   op2   op3   op4
STORE              op0   op1   op2   op3
```

- L0 寄存器组作为级间 buffer: LOAD 写、COMPUTE 读 + 写、STORE 读
- 软件用 `l0_a_reg / l0_b_reg / l0_c_reg` 指定不同 op 用哪些 reg, 硬件按 `op_id % N_L0_REG` 做 cyclic 调度避免读写冲突
- leading 2 cycle 只 LOAD（COMPUTE/STORE 等数据）; tailing 2 cycle 只 STORE
- inner-loop 内不可中断 / 不可被外部 stall (除 NoC backpressure)

7.7 in / out ?????

每实例 3 个独立累加器（A、B、C）。复用 [[DMA]] §7.4 的累加器实现。差别:

- 目标地址落在被动单元（L1 / DRAM）的地址空间内
- inner-loop 由 `loop_count` 驱动, 与 `shape[3]` 一起决定终止
- 地址生成单次给一个 NoC 请求; burst 长度 = 一拍 L0 寄存器宽度 / NOC_W (典型 1 beat)

7.8 Notify FIFO

完全同 [[DMA]] §7.6。

8. Sync Pipe Pool

完全同 [[DMA]] §8。

9. ??????

9.1 NoC ? / ?

结构同[[DMA]] §9.1。NoC主端可对接L1 / DRAM; 对接L1时可按低延迟假设缩outstanding深度, 对接DRAM时按DRAM延迟特性配置。

9.2 ctrl_m ??

完全同[[DMA]] §9.2。

9.3 ctrl_s ????

完全同[[DMA]] §9.3。差别: PALLOC时`params.cu_class`决定走哪个cu_class的子池。

10. ????????

10.1 CU ? pending wait CAM

同[[DMA]] §10.1。

10.2 CU ? ctrl_m

同[[DMA]] §10.2。

10.3 CU ? NoC????????

- LOAD: CU发noc_rd_req给被动单元（L1 / DRAM, master）, 地址按§7.7生成; resp数据按req_id路由到CU的L0写口
- STORE: CU发noc_wr_req + wr_data给被动单元; 不等ack（fire-and-forget）, NOTIFY中的COUNT_DELTA由CU内部FSM在STORE流水尾部排出（保证落到目标单元后才发, 由NoC同id同地址保序保证）

10.4 ? cmd_desc ? in ? out ???? pipe

参见[[Pipe]] §289: 合法用法。硬件执行顺序:

1. WAIT_IN阶段查询该pipe的count ≥ in阈值
2. LOAD阶段读被动单元（数据已在）
3. COMPUTE
4. STORE阶段写被动单元
5. NOTIFY阶段: 对该pipe先发`COUNT_DELTA(-in_delta)`, 再发`COUNT_DELTA(+out_delta)`

被动单元端按到达顺序应用即可。In-place累加的语义保证依赖NoC同id同地址保序。

11. ?????

信号 / 状态	复位值
同[[DMA]] §11全部项	
L0 vreg / treg	不复位（undefined）

信号 / 状态	复位值
计算阵列pipeline寄存器	复位
MAC累加器	0

L0不复位的影响: 未LOAD就COMPUTE是软件bug, 硬件不阻止; 结果未定义。

12. ??????

同[[DMA]] §12, 加:

信号	宽度	用途
`dbg_cu_state[N_CU]`	N_CU × 3	每CU FSM状态
`dbg_mac_active[N_CU]`	N_CU	MAC阵列在跑
`dbg_inner_loop_idx[N_CU]`	N_CU × LOOP_W	当前micro-op序号
`dbg_l0_read_hazard[N_CU]`	N_CU	LOAD/COMPUTE同reg冲突计数 (应为0)
`dbg_pipeline_stall_cycles[N_CU]`	N_CU × 16	NoC backpressure累计stall

13. ??????

1. `N_CU`与异质组合 (如1个2D MAC实例+2个向量实例+1个标量实例vs同质)
2. `VEC_LANE / MAC_M / MAC_K`实际规模——直接决定算力
3. `dtype`支持集合 (INT4 / FP8 / FP16 / BF16)
4. MAC阵列拓扑 (systolic 1D vs 2D, weight-stationary vs output-stationary)
5. L0寄存器组的实际端口数 (FMA用3R+1W还是分段2R+1W)
6. inner-loop流水级数 (3级vs 4级, 多一级换stall余裕)
7. 输出round / saturate硬件块的位宽
8. 是否例化激活函数LUT共享还是per-CU独立

14. ??? Pipe ??

抽象层概念	本文档落点
engine pipe契约	§7.1队列+ §7.2 count + §7.3 FSM
sync pipe契约	§8
落地A	§6 + §8各自独立bitmap
落地B	§3 ctrl vs noc物理分开
落地D	§8 sync独立
特性3 (单pipe不分叉)	§7.3 FSM单channel严格串行
特性6 (硬件零检查)	§5 / §6 / §7.4 L0未初始化不报警
特性7 (参数创建即不可变)	§6 alloc后cu_class / queue_depth定死

抽象层概念	本文档落点
特性 8 (三类 pipe)	engine 在 §7，sync 在 §8
§289（in/out 同 pipe)	§10.4 NOTIFY 顺序

DynamicGraphMultiProcessor??

Dynamic Graph Multi Processor ??

??

1. Etched提出，GPU在过去四年间效率并没有变得更好，只是变得更大了：芯片每平方毫米的的TFLOPS几乎持平。
2. 「干净数据+大模型」和「脏数据+大模型」的效果，不会有太大差异。
3. Etched团队表示，H100有800亿个晶体管，却只有3.3%用于矩阵乘法，这种大模型推理时最常见的运算。只支持Tranformer的Sohu芯片FLOPS有效利用率超过90%（GPU大约是30%）
4. 在前Scaling Law时代，我们强调的是Scale Up，即在数据压缩后争取模型智能的上限；在后Scaling Law时代，需要比拼的则是Scale Down，即谁能训出更具「性价比」的模型。

????

- 针对AI应用，极高的规格参数
- 抛弃传统计算机的特性，只支持Int8，加减乘，固定的数据流水pattern，固定的计算管线
- 算子的输出输入精度更低，计算精度更高，自动混合精度
- 脉冲阵列，多周期指令（超级大的2D指令）=> 提高算力密度？？
- GPU的核心问题就是：管理DRAM的延迟和有限的带宽
- 所有的硬件单元（包括NoC，Cache等）都在执行一张计算图，通过统一的图关系指令（Fork/Join）来运行，软件可以精细得控制所有的硬件单元，甚至是NoC得flow control行为。
- 软件控制内存一致性问题，传递信息到NoC/Cache，硬件不做自动化处理
- 传统上架构，内存一致性（consistency corhenrency）需要通过fence+sync功能来保证，非常麻烦和易错。
  1. GMP通过显式的同步模型（fork/join）来管理同步关系
  2. 线程内的利用静态的依赖关系来尽量避免fence的使用
  3. 线程间的编译器自动掺入fence指令
- 全异步，各种单元之间可以主动同步，每种单元（IP）都可以执行一些指令，都是一个cpu核，执行自己的代码和调度，区别于传统设计，从IP一般都是通过主IP进行控制，比如cache单元被处理核通过cache_hint进行配置
- 边缘端的越级大模型，极度定制的芯片
  - 固定的计算单元的组合（DSA）
  - 整个SOC级别的动态调度
  - 以NOC为编程中心的调度和数据流
- 标量单元，向量单元，张量单元灵活组合，操作数（寄存器）可以灵活转换
  - 充分把合理的计算放到合理的单元，节省面积和功耗：一个标量x一个张量

    - 充分利用软件的编译
- DSA的效率，GPU的灵活性

??/??

1. 动态和静态的范围
    1. 全静态（graph）调度逻辑，包括DRAM都静态调
    2. 通过LLC隔离DRAM的动态性，LLC以下全静态 L3->LLC->DMA->L1->L0->MALU
2. 静态图
    1. 利用fork/join（硬件加速）描述所有的并行性，比如L1/L0的服用，DMA的操作后同步
    2. 有并行就有调度器，有调度就有缺陷
3. 动态
    1. Root 调度 & 核内调度 & 流水线调度
    2. 数据流、NoC、Cache
    3. Fork Join
    4. 充分利用CPU领域的灵活性，RISCV多核，加速的硬件调度指令（地址计算，IP配置等），同步指令，ForkJoin等
4. 静态
    1. NPU流水线，NPU Kernel，1D 2D指令及组合
    2. DSA Confige
    3. Atomic Reduce专用加速硬件，近存储计算，压缩解压缩
    4. LD/ST+Relu专用硬件

??

1. NPU的流水线延迟固定，包括LD/ST（编译器已知），都是从L2/L1 <-> L2/L1
2. 都是MLI指令，可以打包成HLI指令，不支持跳转等
3. 不支持不对齐? 不对齐是不是延迟就未知了
4. 同时只能执行一个任务（一堆MLI指令）
5. 指令堆会有Leading/Tainling 时间，scheduler感知得到，可以进行调度，前后两条可以自动pipeline，NPU本身不考虑依赖
6. 强大的软件仿真和验证平台，减少硬件的debug需求，大大简化硬件的调试/检查/报错功能

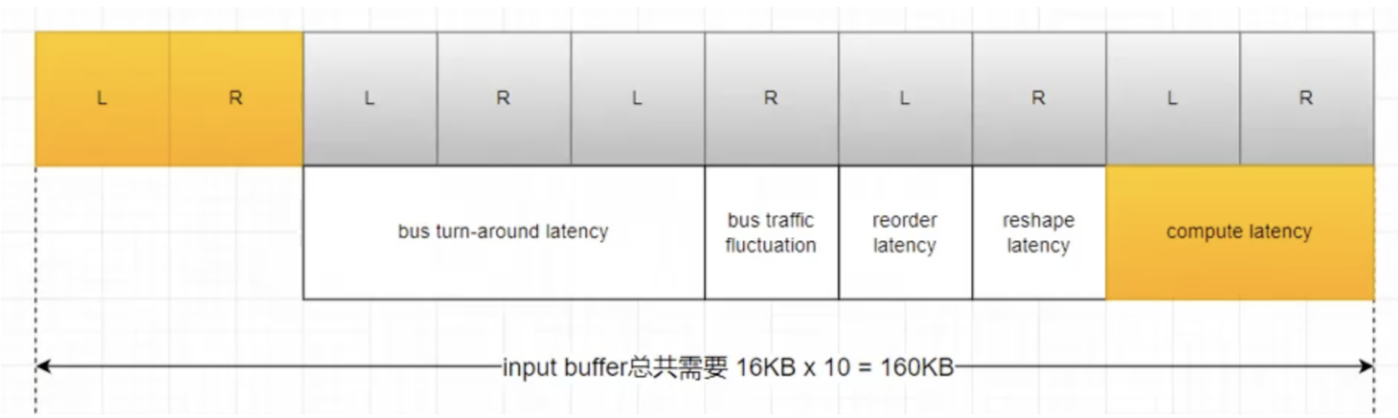
???ASIC??

1. MLI不是硬件拆分? 由编译器拆分，为了减少硬件的难度
2. scheduler控制NPU的指令搬运、
3. 明确latency
4. 支持打包成HLI

??

1. 多GM之间共享数据及同步
    1. scheduler可以做及其动态的交互和同步
2. 内存墙
    1. 自动精度缩放，权重存储的是<8bit，DMA搬运的时候变成8bit，每个参数都可以进行不同bit的缩放

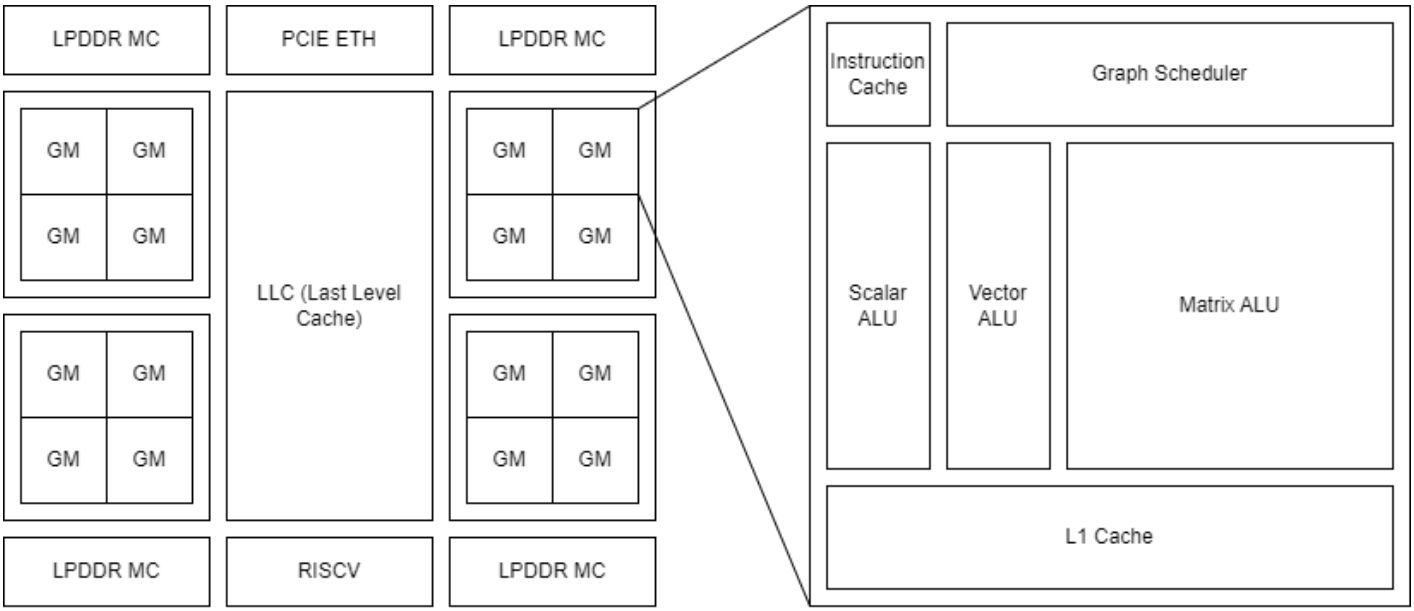
3. Launch/Sync/DataFlow
    1. 高效的实现Launch和Sync，抽象出Fork/Join指令
        1. 主动**Fork和Join**的不一定是同一个对象，可以**A fork B，C join B**
        2. fork等价与launch，join等价与wait
        3. 可以一次性fork多个，也可以一次性join多个
        4. 硬件单元存储一个fork出来的任务的队列进行调度执行，不同单元的并行度不一样
    2. 支持片上动态的Launch和Sync
    3. 在NoC里面同时实现Launch控制，同步，数据流
    4. 统一在一个地方调度，所有单元并行，一起执行一个大的Graph Kernel
    5. 所有单元的指令特点：**latency**短且固定，只存在一级调度，调度后不可阻塞
4. 精确的调用（launch）和同步带来的好处
    1. 突破内存墙：大算力->大延迟->大的in flight存储，大的pipeline存储，通过精密的同步和异步最小化存储器的使用
    2. 突破数据墙：越低的精度，需要的带宽比例越大，低精度算力面积呈指数下降，但是带宽需求是线性下降。只有精确的同步和调度才能最优化带宽资源的使用。
    3. 提升算力密度：算力单元的简化，静态化
    4. 大幅提升随机存取和计算的能力，是智能的重要指标
    5. 通过高效的控制和同步，最大化提高片上SRAM的复用率
        1. 总的片上的SRAM的容量很大，但是分散在很多小的L1，L1之间可以快速的同步数据，没有L2、LLC
        2. 分散的小的L1等价一个很大的cache，缓存下GEMM的整个右支
        3. 在有限的面积下，既满足了容量的需求，又满足了带宽的需求
        4. 支持非常细粒度的同步，比如L1的各个bank之间的读写的同步，计算核的各个thread内的指令间同步，计算核内的各种Engine（ALU，RegRead）的同步，MALU的ld Cal St的同步。
    6. 动态graph，动态体现在动态launch，子图的动态的调度，循环调度，动态高效的同步支持，子图是完全静态的。
    7. 不同的算子需求下，都能写出高性能的算子，DSA的架构可能高性能，但是通用性差。完备指令架构很难做到优秀的PPA。gpu有强大的gs来适应大部分的不同数据流需求，simt的灵活调度提供充足的指令流，simi的指令尽量避免（软件解决）各种data和structure的hazard，减少流水线的复杂度。
    8. 通过紧密的同步控制，结合编译器，可以在通用性的前提下做到DSA的性能。通过减少流水线的fence需求，和硬件资源空泡的概率，特别是TensorALU、LD、ST等长latency的单元。传统的解决办法是通过多线程来提供充足的standby的微指令给硬件单元自己调度，避免空泡。

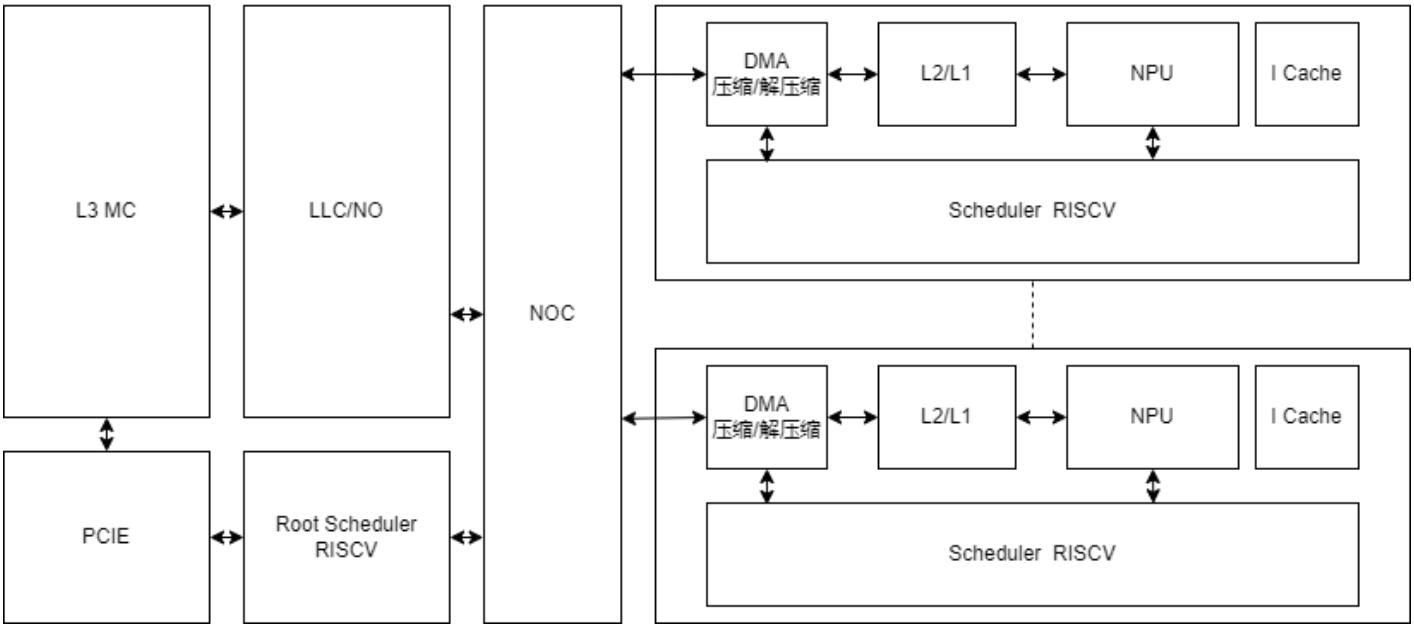


??

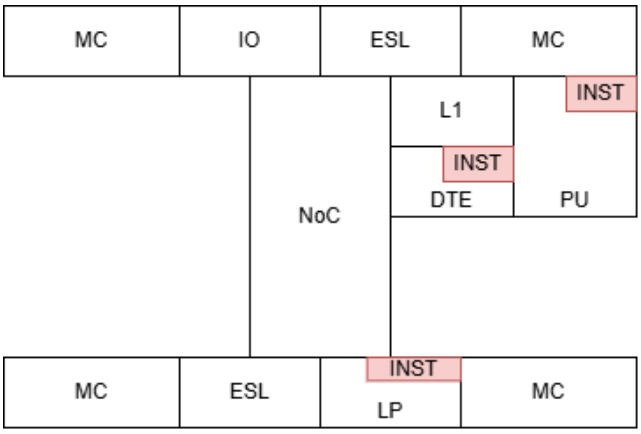
1. 支持多线程，为了充分利用硬件，不空泡
    1. 编程复杂
    2. 需要复杂的同步pipeline控制
    3. 需要复杂的编程L2L1复用
2. 不支持多线程
    1. 通过graph指令，配合编译器，单线程实现并行
    2. 专用的加速指令，加速DMA操作，地址计算，同步/异步等
    3. DMA和NPU等外设，可以直接读取Scheduler的Inst SRAM的方式执行，避免配置参数拷贝
    4. 相对于scheduler的外设，通过fork和join到对应的外设来支持异步编程，避免复杂的多线程语义
    5. 物理上可以有超线程并行的硬件，共用一个线程的寄存器和堆栈，通过fork，join软件控制表达并行和依赖
3. VMM怎么和VLD VST并行，pipeline
4. kernel种类太多，体积大怎么办？？
5. 各个IP同步的开销太大
    1. HWSync硬件指令，同步资源集成到RISCV CPU里面去，独立的中断信号线
    2. 各个IP之间通过mailbox相互trigger，mailbox可以动态配置，但是像pipeline那样太复杂了，可以支持forkjoin抽象
6. 数据的写透（fence），流水线都很长，导致leading tailing太大
    1. 每个NPU的任务都有一样的（明确的）leading和tailing时间，所以可以在scheduler直接流水发射

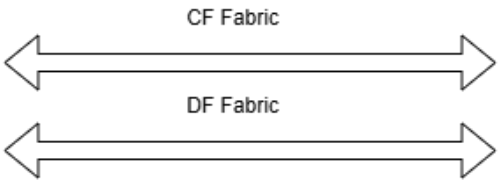
Graph Multiprocessor = GM

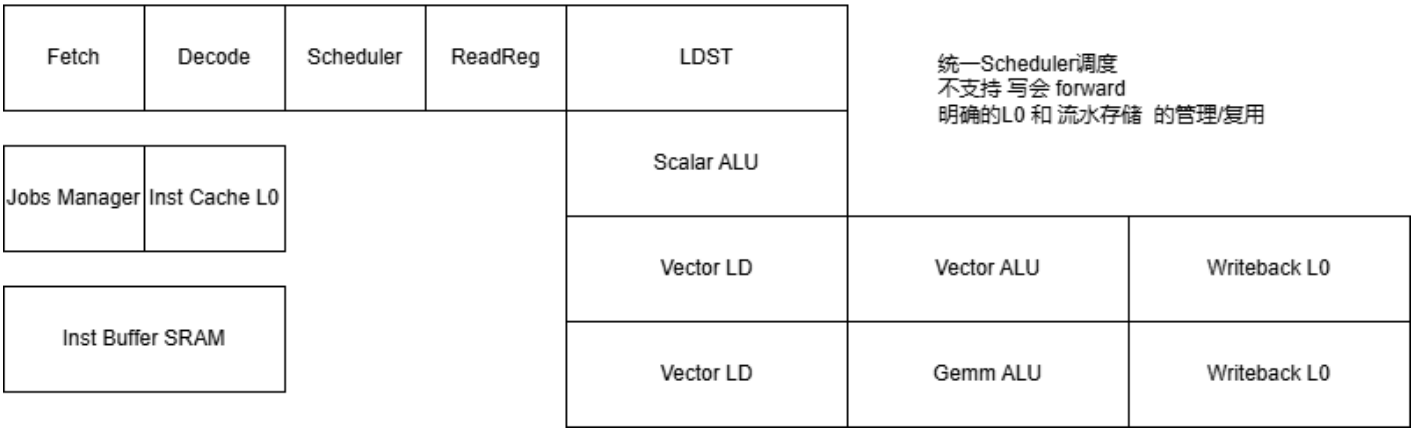




1. 同一片代码多个并行传递参数问题
    1. fork支持参数的计算，批量 fork自动累加传递的参数，类似于 for功能的批量 launch，
    2. 一个 kernel最小的 fork出来的执行单位表达类似于一个 gemm指令
2. 架构
    1. 32bit指令宽度
    2. 标量 +向量 +2D混合指令







统一Scheduler调度
不支持 写会 forward
明确的L0 和 流水存储 的管理/复用

????

1. 支持未来的超动态网络？加速动态网络？
    1. 通过静态图的自动动态调度执行，能高效进行基于数据的动态网络
    2. 动态的 launch/fork和 join
2. 编程的抽象
    1. SIMT的编程对象是一个 thread，并且要求所有的任务拆解成一堆一样的 thread
    2. **GMP**更像是 **MGMT**，**Multi Graph Multi Thread**，编程的基本对象是 **Graph**，**Graph** 可大可小，可以组合，可以动态执行（ **Launch**），可以被多个计算核并行加速
    3. 专用的 fork/join指令和硬件记录 graph的 launch和结束
    4. 单个处理单元内部的不是多个线程 free竞争 run，而是经过编译器严格规划的，graph是 processor运行的基本单元，graph内部的多个分支的并行需要精密的同步，
    5. 每个 fork出来的 subgraph对应的处理级别都是明确的，通过 PU执行的 graph，fork出来的 subgraph，就是 hw thread级别。
    6. 同步指令只能在同级别的 graph之间进行？
    7. fork和 join支持 “无主模式 ”，fork出来之后主 graph消失，每个子 graph记住自己和哪些 graph 是并行的，利用这个信息进行同步，直到执行 join之后，回到上一级 graph，此时的 join类似于 barrier。
    8. **counter**：wait 指令总是等待自己的一个 counter值，signal远程的一个 counter值 ，怎么实现灵活动态的同步需求？
        1. 预先的 pull一次，配置上动态 wait信息，然后再 wait signal

        2. 通过 launch来指定动态同步信息
        3. 动态性是通过 launch来体现的，动态信息是 launch的时候统一生成的
        4. 一次 launch出来的并行 sub graph可以有一个 graph group的信息资源，可以实现“signal下一个”功能，达到按序执行的调度目的
    9. 同步的指令都可以附带一个 delay信息，因为静态的流水线，可以提前预估未来发生的同步点
        1. 同步指令不需要有状态信息，不需要读通用寄存器，不需要和其他有 data hzd
    10. 大量的短的非 stall的流水线 替代 少量的长的流水线，更多的非 stall流水线有利于静态优化

??

1. Fork Join
    1. fork指令附带的信息
        1. 配置的代码段的参数
        2. 需要被 join的信息
        3. 代码段的 index
        4. 当前 launch的编号 ID：编号 ID能直接解析出硬件的 IP和 memory mapping
    2. fork：专用于指令 load的指令
    3. join指令的附带信息
        1. launch的编号 ID
2. 向量寄存器 VR L0.5，向量寄存器作为 L0.5存在，所有的软件线程公用
3. 向量寄存器 VR L0，软件线程独立的寄存器，每个线程只有 4个，直接设计在流水线上
4. 张量寄存器 TR L0.5，张量寄存器作为 L0.5存在，所有的软件线程公用
5. DTE支持 transpose pad slice deslice
6. PU单元内部的线程之间支持快速同步
7. 线程内支持微架构的流水线操作和控制
    1.

Width (bits)	4	6	3	3	1	
Meaning	Reuse flags	Wait barrier mask	Read barrier index	Write barrier index	Yield flag	St cycl

    2. Reuse flags 用于软件主动控制 4个 register的 cache
    3. 6个 barrier硬件资源用于软件主动创建 thread之间的异步依赖关系（cuda的关键调度流水线长度是 6），而不需要浪费硬件面积
8. async_group 对历史的异步指令进行分组，以便灵活的进行同步
    1. async copy bulk 一个指令拷贝多个数据
9. mbarrier：数据接受端支持主动维护数据传输状态，可以避免数据发起端频繁和接受端同步和 fence来保证信号的前后顺序
10. 所有的流水线的信息传递都因为有竞争，不能做无阻塞，常用的利用 credit和 valid/ready的阻塞方式，浪费太多的面积
    1. 本质上是硬件实现了某个程度的自动化的功能
    2. 如果能从软件角度就静态化，能节省大量的面积
    3. 从源头上（调度流水级）软件通过特殊指令控制发射的带宽来做到软流水无阻塞执行

????

1. 单个芯片 die规模
   1. 4T int8算力
   2. 集成 256MB DDR3 DRAM 1866 16bit 约 3.7GB/s带宽
   3. 小规模的芯片，便于开发，降低风险和成本，便于仿真
2. 支持无缝的拼接，多个芯片能容易组合成一个大的芯片，板级集成
   1. 降低流片成本，单个 die非常小
   2. 降低封装成本，无需高级的芯片整合封装方案
   3. die to die 通过低成本的低速链接，不需要高速的 serdes方案

ISA

ctrl

1.
   1. Fork(code_index)
   2. launch
   3. Join()

Scalar

1. scalar 计算

Vector

Tensor

???????

适用于 `design/` 下所有微架构 `.md` 文档。参考实现: [[LLC]] + `LLC.pipeline.html` 。

0. ??

- 微架构文档是人面向 **RTL / [[logix]]** 模型实现的合同：写完 `.md` 就能据此写 RTL，不再回头补
- 每个单元一份主 `.md`（如 `L1.md`、`DMA.md`、`LLC.md`），描述功能与流水线
- 拍级流水线强制作为独立章节，不和功能描述混写
- 可选的交互式可视化 HTML 文件挂在同目录（`<unit>.pipeline.html`），不写入 git LFS、不依赖外部资源

1. ?????

主 `.md` 按以下顺序组织章节（编号用 `#### N. 标题`）：

编号	章节	内容
§1	顶层参数	容量 / 宽度 / 深度等常量的表
§2	顶层框图	ASCII 框图或 mermaid，标出对外端口与内部主要模块
§3	顶层端口	控制 / 数据 / 出对方向各一节，沿用本仓库已有端口约定
§4	地址映射	若是存储类单元；否则按需
§5	控制总线协议子集	opcode 表、异步操作约定、不做合法性检查的边界
§6 - §M	各功能模块详述	每个功能模块独立小节（阵列、命中 / 缺失、出口、维护等）
§M+1	流水线	拍级时序契约，见 **§2**
§M+2 -	Pin / Lock / Bypass 等可选语义、与其他单元关系、复位、调试、待规格化参数、DGMP 关联	

数字章节用 H4 `#### N.`、子节用 H5 `##### N.M`、子子节用 H6 `###### N.M.K`。三级到顶。

2. ?????

每个微架构文档必须有"流水线"章节。结构如下，编号 N 取主文档下一个未用数字（LLC.md 取 §11）：

2.1 ??

- 一段话说明：本章把哪些功能章节（§a-§b）在全局时钟上展开到拍级
- 工艺前提（如 SRAM 端口形态 1RW vs 1R1W）必须显式
- 配套交互式视图：`<unit>.pipeline.html` 的位置和用途
- 渲染说明：Mermaid / WaveDrom / YAML 在哪些环境能渲染

2.2 §N.1 ????

按拍管理的资源表。一个三列表足够：

资源	端口形态	来源
`<name>`	`<arb / sram_port / ff_rw / cam / table / comb>`	§x.y

资源是硬件实体——一个仲裁器、一组 SRAM 端口、一个 FF 寄存器组、一个表。不要把功能动作写进资源清单。

2.3 §N.2 ????

mermaid `flowchart LR` 显示所有路径的分叉、合流、异步链。每条路径用一个 stage 块（不展开内部）。

2.4 §N.3 ~ §N.X ????

每条路径一个子节，内部固定四件套（用 H6 ###### 编号 §N.X.Y）：

1. 拓扑：mermaid `flowchart LR`，stage 节点链 + 分支条件
2. 拍级展开：WaveDrom JSON 代码块，横轴 cycle，纵轴 stage / latch / 资源占用，跨信号依赖用 `edge` 字段
3. **latch** 字段表：每个 stage→stage latch 携带的字段（含字段名 + 简短解释）
4. 资源占用表：拍 × 资源 的矩阵

跨路径关系（与主路径的冲突、回填唤醒等）单独成一个子子节。

2.5 §N.{?? 2} ???????????

- 资源 × 占用方表：每个资源列出主要占用方 / 次要占用方 / 仲裁规则
- **stall** 触发表：什么条件下哪条路径会 stall、解除条件
- 全 **bank** 吞吐分析（如果适用）

2.6 §N.{?? 1} ??? schema

YAML 代码块作为机读真相源，给工具消费（未来生成 logix / RTL 骨架、跑静态检查）。schema 由四部分构成：

```
unit: <name>
clock: <main_clk>

resources:
```

```
  <name>: { kind: ..., ports/slots/width/banks: ... }

paths:
  <path_name>:
    stages:
      - { id: <S0/S1/...>, in: [...], uses: [...], out: [...] }

arbitration:
  - { resource: <res>, contenders: [...], rule: "<text>" }

stall_conditions:
  - { src: <res.cond>, effect: "<text>" }
```

写出哪几条路径作为样本即可，完整版可旁挂 `<unit>.pipeline.yaml` 文件。

3. ??????

模式	拍号	例
主路径	S0 / S1 / S2 / S3 / S4	读命中
主路径分支	S3' / S4'（带撇号）	读缺失从 S2 分叉到 mshr
异步流水线	F0 / F1 / F2	回填（fill）
独立 FSM	W0 / W1 / W2 / W3	维护扫描 walker
特殊短路径	B0 / B1	旁路 bypass
重发	S2'（带撇号，复用主路径名 + 撇号）	回填后 master 重发到 S2

不同路径的拍号空间不冲突，方便资源占用全景表交叉引用。

4. ????????

- 同文档内：`§N`、`§N.M`、`§N.M.K`
- 跨文档：`[[文件名]] §N`（Obsidian wiki link 风格），文件名不带 `.md` 后缀
- 大段引用别人的协议时，直接说 "完全沿用 [[L1]] §3.2 的 NoC slave 信号集"，不要重复贴信号表

5. ???????

载体	表达	谁读
Mermaid `flowchart`	拓扑结构（节点 + 箭头）	人（GitHub / Obsidian / VS Code 原生渲染）
WaveDrom JSON	拍级时序（波形 + 跨拍依赖 + 资源冲突）	人（wavedrom.com/editor.html、Obsidian + WaveDrom 插件）

载体	表达	谁读
嵌入式YAML	机读schema（resources / paths / stages / arbitration）	工具

三者信息互补，不互相替代。同一份事实在多处出现时，YAML是真相源——Mermaid / WaveDrom是渲染产物（手写阶段允许双源，工具成熟后单源生成）。

不要使用：drawio（不好diff、信息无法被工具消费）、PNG/SVG截图（同前+改一处要重导）、Word/PPT（同前）。

6. ??? HTML???????

文件名约定：`<unit>.pipeline.html`，与主`.md`平级。

单文件、零外部依赖（CSS / JS内嵌），打开即用。数据来源理想是同目录的`<unit>.pipeline.yaml`；起步阶段允许把数据写在`<script>`的JS对象里，等schema稳定再切换。

四块信息（沿用`LLC.pipeline.html`模板）：

1. 流水线甘特图：横轴cycle，纵轴路径，stage色块按cycle落格
2. 资源占用网格：横轴cycle，纵轴资源，单占用/多路径冲突分色
3. 按硬件级展开：横轴cycle，每拍列出所有承载的并行子功能
4. 按路径展开：每条路径横排stage卡片，卡片内显示功能/资源/输入/输出latch

第3/第4互为转置视图。点击任一处的stage/子功能高亮另外两处对应项。

7. ????

1. 文档先行：先写`.md`，再写RTL。`.md`锁定的拍数是RTL必须满足的契约。
2. 单源真相：拍级时序信息只在"流水线"章节一处描述。功能章节描述"做什么"，不描述"在哪一拍做"。
3. 工艺前提显式：1RW vs 1R1W、SRAM端口数等工艺选择必须在引言或资源清单中写明。
4. 不重复：流水线章节不重述功能章节的逻辑；功能章节不写拍级时序。
5. 资源**vs**功能：资源清单只列硬件实体，不列功能动作。功能动作是stage里的`comb`字段。
6. 沿用**LLC**用过的词：cacheline / set / way / pinned way / bypass / fill / writeback / mshr / walker。术语在不同文档间保持一致。

8. ?????????

放在主文档末尾。两类内容：

1. 硬件参数：容量、深度、端口数、是否例化ECC / prefetcher等产品级决策
2. 流水线相关：仲裁规则细节、retire拍号、FSM状态转移等取决于floorplan /实现的部分

不要把"功能未定义"的内容塞进待规格化——功能必须在功能章节里明确。

9. ?????

角色	文件
主文档样板	`design/LLC.md`
流水线章节样板	`design/LLC.md` §11
交互式 HTML 样板	`design/LLC.pipeline.html`
上层抽象（不写流水线）	`design/Pipe.md`、`design/架构_微架构.md`

新增模块文档时：拷贝 `LLC.md` 结构、改章节内容、删掉用不到的小节。流水线章节直接套用 §11 的子节结构。

10. ????

提交前过一遍：

- □ §1 - §M 功能章节完整，每个硬件模块都有独立小节
- □ 流水线章节存在，包含资源清单 / 路径一览 / 每条路径四件套 / 资源占用全景 / 嵌入式 YAML
- □ 工艺前提显式写出
- □ 拍号命名遵循 §3
- □ 跨文档引用用 `[[...]] §N` 形式
- □ 资源清单只列硬件实体
- □ 待规格化清单只放参数和实现细节，不放未定义功能
- □ 配套交互式 HTML（如有）数据与文档一致
- □ 沿用本仓库已有的术语（参考 LLC.md / Pipe.md / GMP.md）