Dynamic Graph Multi Processor 架构

背景

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。

核心特点

- 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

动态/静态

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

特点

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

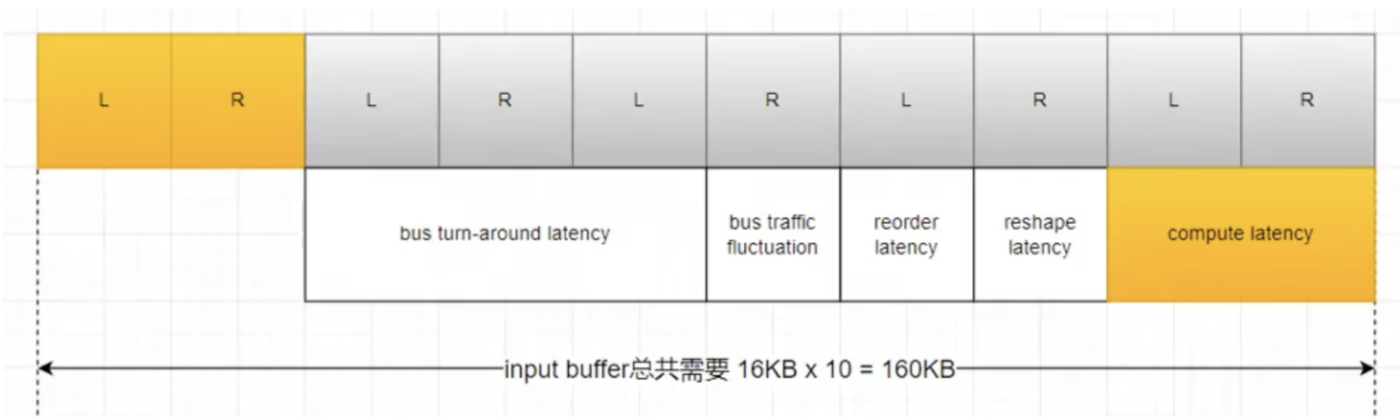
和主流ASIC区别

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

方案

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

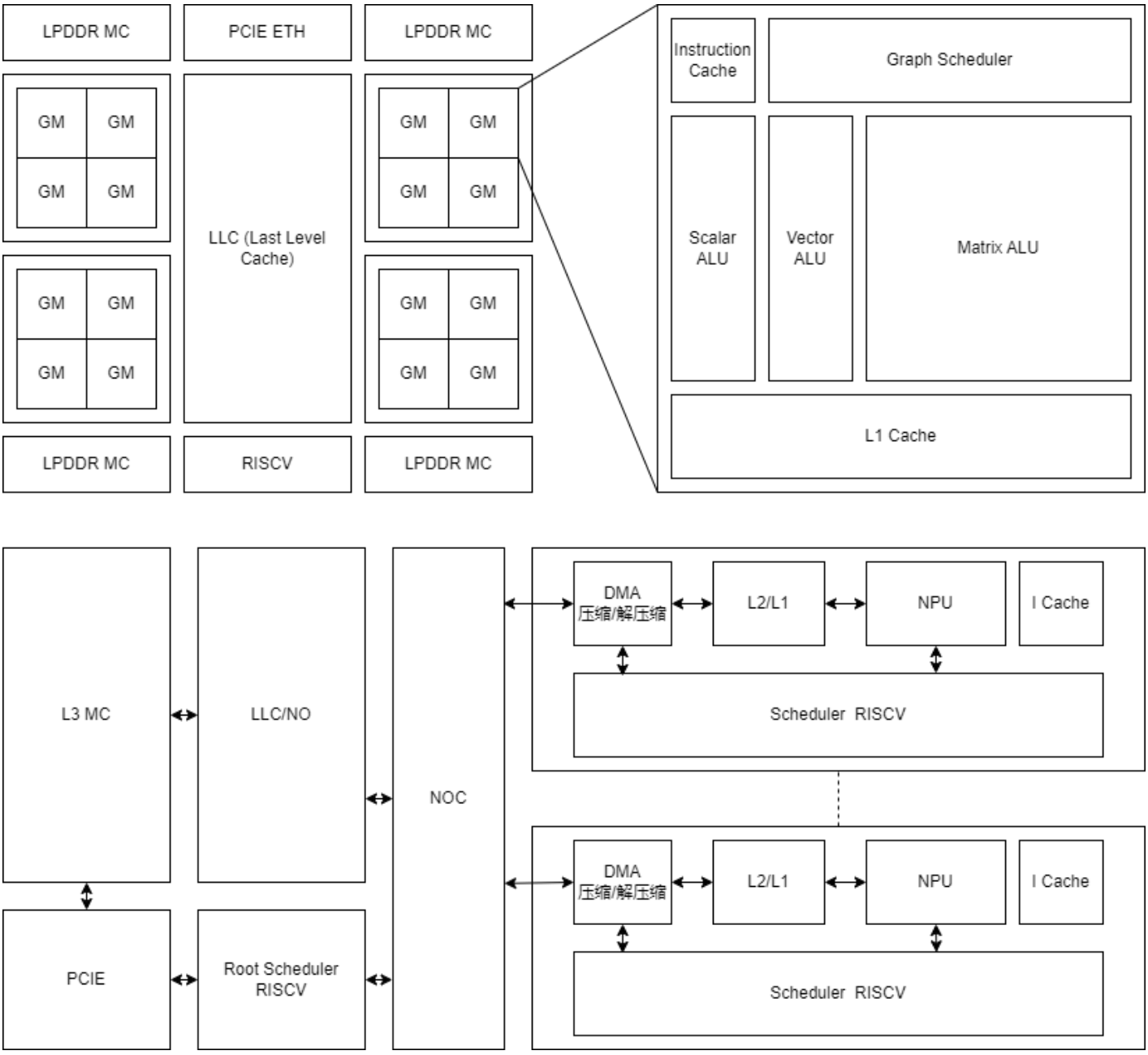
3. Launch/Sync/DataFlow
   1. 高效的实现 Launch和 Sync，抽象出 Fork/Join指令
      1. 主动 **Fork**和 **Join**的不一定是同一个对象，可以 **A fork B**，**C join B**
      2. fork等价与 launch，join等价与 wait
      3. 可以一次性 fork多个，也可以一次性 join多个
      4. 硬件单元存储一个 fork出来的任务的队列进行调度执行，不同单元的并行度不一样
   2. 支持片上动态的 Launch和 Sync
   3. 在 NoC里面同时实现 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。



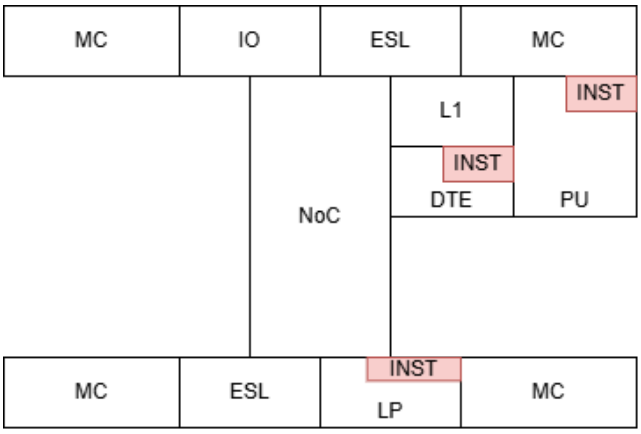
问题

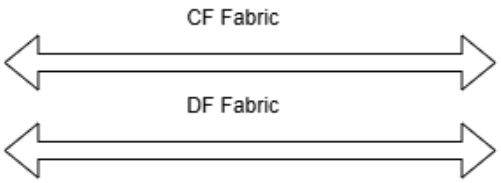
1. 支持多线程，为了充分利用硬件，不空泡
    1. 编程复杂
    2. 需要复杂的同步pipeline控制
    3. 需要复杂的编程L2L1复用
2. 不支持多线程
    1. 通过graph指令，配合编译器，单线程实现并行
    2. 专用的加速指令，加速DMA操作，地址计算，同步/异步等
    3. DMA和NPU等外设，可以直接读取Scheduler的Inst SRAM的方式执行，避免配置参数拷贝
    4. 相对于scheduler的外设，通过fork和join到对应的外设来支持异步编程，避免复杂的多线程语义
    5. 物理上可以有超线程并行的硬件，共用一个线程的寄存器和堆栈，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

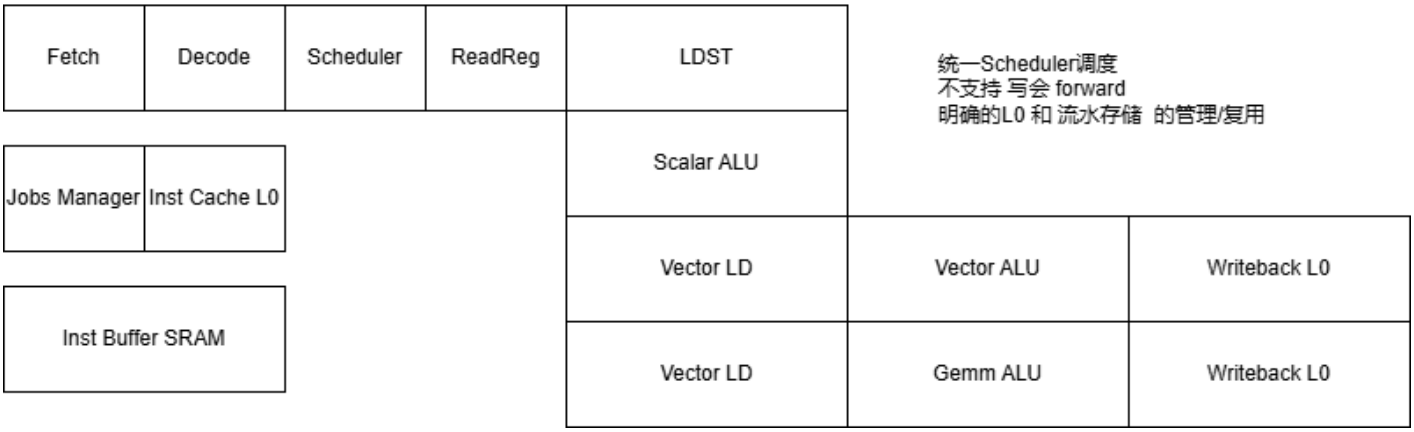
Graph Multiprocessor = GM



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

体系架构

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

架构

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

产品架构

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

Scalar

1. scalar 计算

Vector

Tensor

Revision #5
Created 11 January 2025 09:46:28 by Colin
Updated 14 March 2025 03:54:46 by Colin