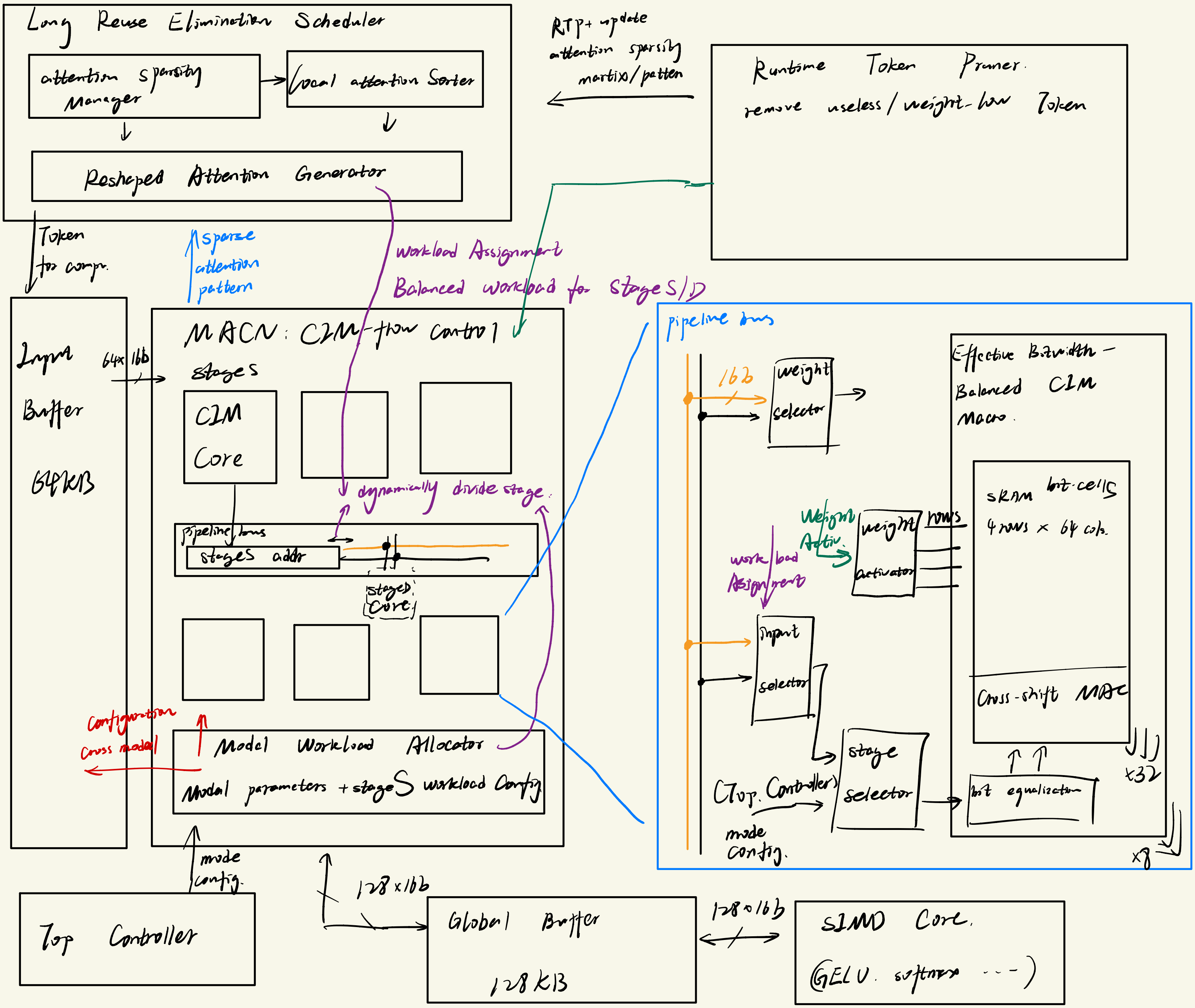
CIM-Core: stageS / stageD: Dynamicly Distro, generate QKV / $Q \cdot K^T$. AV



Long Reuse Elimination Scheduler: 管理 attention spersity
控制 attention 的 token 输入顺序和 sparsity matrix 调度.

RTP 参数/info → LRES 调度 → MACN Input Buffer　　stageD (attention 计算) 前, 提供 Input Buffer 和 MACN 调度方式
stageS QKV / stageD $Q \cdot K^T$. A·V → SIMD 后处理

Runtime Token Pruner: 实时计算 Token 的 weight/activate. 参数. 减少无用输入

EBB-CIM: 调节不同的 位稀疏性, 提高 CIM 并行计算效率
动态调节 位宽, 适应不同模态的 Token ...

{ SRAM. bit cells; 存储 weight.
cross shift MAC: 动态控制 多模态输入通道.
bit equalization: 平衡不同数据的 bit 分布.

Modal-Workload-Allocator: 切换 Model. (Cross-modal attention CFC-Attention)

计算结果存入 Global buffer, SIMD 完成激活. (非线性)

LRES. stageD: $A \cdot V$. $A = Q \cdot K^T$, $Q = \vec{q}$ (input) $K = \vec{weight}$ (in CIM)

一些 K 长期闲置不运算. (K 只有当计算 q_{cls}, q_i 时才被计算) → 节省 CIM 资源和开销

LRES: Original attention —reshape→ global-like + local-like

- Global like: 提取对当前数据块的 q, k 保留更长时间, → long reuse? (short)
- Local like: q, k 相当少数 (相邻) 数据相关. → quick refresh 或立即用

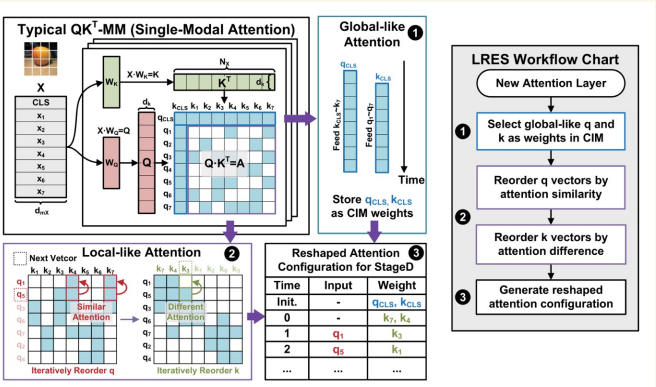


Fig. 6. Long reuse elimination workflow, with a QK^T-MM example.

Original unordered Attention matrix?

in transformer (google): $A = Q \cdot K^T$ → Attention matrix.

$A[i, j]$ → ith Token (Query i) 对 jth Token (key j) 的关注程度.

原则上: Token 可以 Dense. 但实际上稀疏明显

- unorder: key: far ← 0 0 0 0 ... 0 0 0 1
  - 无法批次载入, → 频繁换入换出
  - q_i 要对很多 k 计算 { 这 无效
- long reused: k_i 载入后等很久才会再次计算 / 无用
- Idle: k_i 载入 未使用 → 硬件闲置

unordered: k 无法批次载入 / reuse distance 太长

How: ① 从 attention matrix 中找出 最常说、与其他 Token 交互最多的量 → 驻在 CIM 中长期使用 (cache hit)

② 重排列序 q, k. 尽可能把 相邻块 → 使相似的 q 在一起, 与之相联 k 在一起

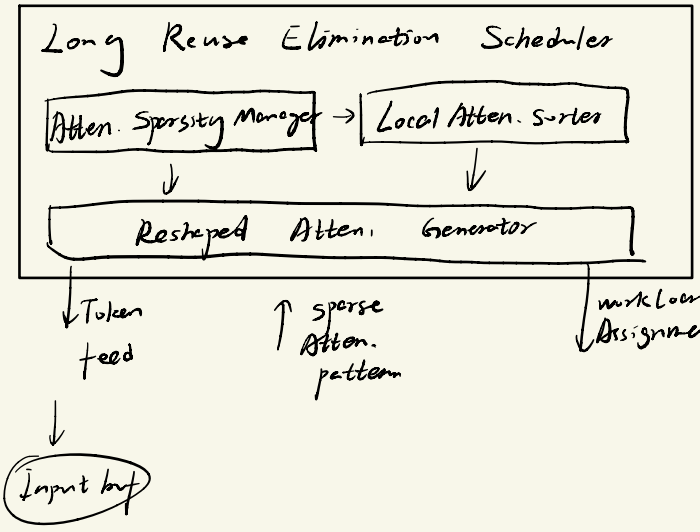
③ generate reshaped attention Configuration → q 在何时被输入 / CIM 中如何载入 k

long Idle: long reused: global-like,

Attention sparsity: local-like.

Comput./transport chaos: Configuration.

LRES architecture.



step ①: Attention sparsity Manager.

Select: global like q, k

step ②: Local Attention Sorter

re-order q, K (removed) —format→ Local-like.

Step ③: Reshaped Attention Generator

根据 global-local 排列 输出 runtime Configuration

ASM: ↑: 稀疏的 + RTP (remove-info).

①: row sum: → global q (找出与最的 Token 联的 query) 发出 attention

column sum: → global k 哪个 key 被更多的 query 关注

起步数 0

到阈值 → global → FIFO

LAS：只能让 attention 相似的被淘汰，让它们共享一组 K（Core-window）

生物学的可能有：slide-window 聚合，Attention Similarity-based Clustering

很多个 query 共享 1 个 key，CIM 独得好，数据搬运就小

RAG：生成它的 attention 稀疏方阵表

time：？ Input：？ weight：？ global/local？

---

attention similarity-based clustering

相似 attention pattern 的 token 归为一类

简化/共享 attention 计算 → effective + accelerate + compression

→ 很多 token 是比较类型是一样的/相似

事先分析 query 向量特征 + cluster + 每个 cluster 保留一个代表 query

compression：{ memory：少存 / bandwidth：无需对所有计算

cluster → global + local

{ 直接掉不重要的 attention block

global-like / less query？ / reduce useless computation

Reshaped Attention Generator

↓ Configuration

1. 控制 Q、K 生成顺序 stage S

2. 控制 CIM macro 读取 buffer 的 Q-K stage？

---

slide-window 聚合（CNN？）

让 query 只关注附近的 token

减少计算量和 memory reuse distance

初始的 Attention 是 dense 的

$A = Q \cdot K^T$ （Sl2 × Sl2）

如让 query 只关注 window=1 的，即向左右各关注 1 个

每次只计算 2w+1 ←w→ 0 ←w→

CNN fixed weight / Attention slide windows runtime compute weight

---

fixed pattern / pattern based on segement

2D pic. / language token.

？ sparsity optimization.

slide-window：{ reduce data transportation.

keep k used frequently

parallel + stream pace.

Local Attention Sorter

（mechenisim）

row：让下一个输入的 Q 与上一个有最大的 similarity

两个 Q attention 的 K 相似，无需替换 CIM 中的 weight

col：调整 K 载入的排列

出去不用回来

FIFO：① global - Q K

② local Q

③ local K

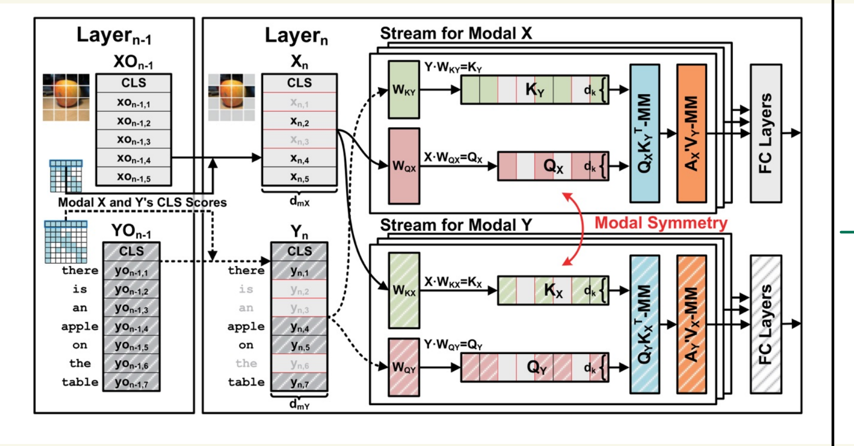
RTP:

① Token prune based on Class Score (CLS)

Transformer 中有专门汇总整体信息的 CLS Token

CLS Token 对 Token 的 attention 高，则 Token 重要.

So 根据 CLS 进行裁剪.



flow: $layer_{n-1}$ 中有 CLS Token.

attention matrix 第一行为 CLS Token 对其他 Token 的 attention.

得分低的是被裁剪的 Token

(裁剪后可能导致 Model X 和 Model Y Token 长度差异更大)

Challege 2: (MACN 解决)

X 的 Token 比 Y 少，裁剪后更少)

在 cross-modal attention 中，Q_X 先于 K_Y 生成

Q_X 会空转等待 K_Y 计算

↓

pipeline stages 延时长，CIM 使用率↓

MACN arch.

Multiply-Accumulate CIM Network.

包括：① Modal Workload Allocator

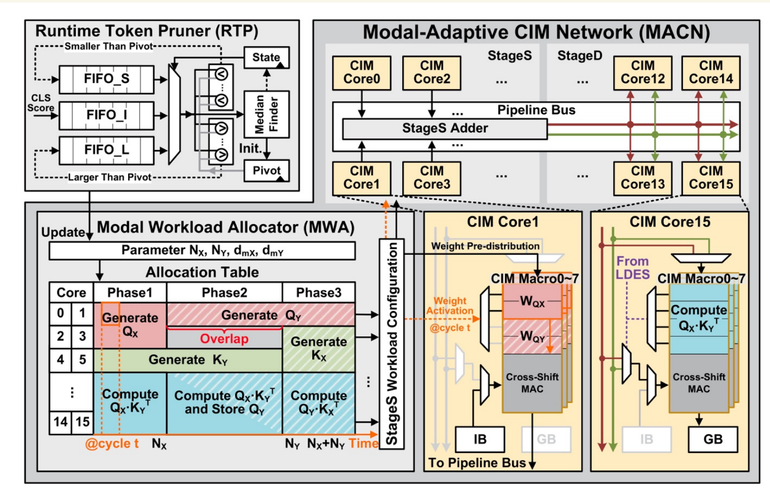


Fig. 11. RTP and MACN architecture.

CLS Token 是如何生成的?

CLas Sification.

一般是一个可学习的嵌入向量

Runtime Token Pruner

mechanism.

从上层得到 CLS

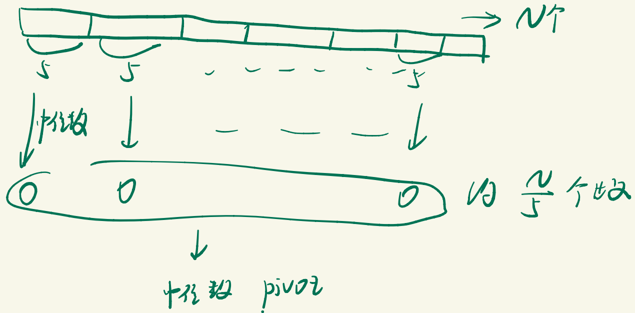
利用 median-of-median top-n 选择算法.

计算最重要的 n 个 token.

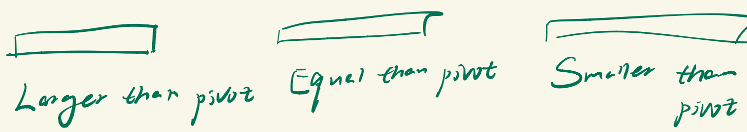
median-of-median top-n.

选出无序数列中第 n/前 n 大的算法.

如要找前 k 大的元素.



对所有数分为3部分



如: $k \le |L|$，就等于在 "L" 中找前k大的值

$|L|+1 < k \le |L|+|E|$，结果为 pivot

$k > |L|+|E|$，在 "S" 中找前 "$k-|L|-|E|$" 大的数.

② 16个 CIM cores ③ Pipeline bus.

MACN 有2个模式: FC: CIM core 并行运行 (parallel) (pipeline) attention: MWA 根据 RTP 分配 stage S/D

challenge 2 解法：（X、Y token 不同长度，负载不均等级 Q_X 空转）.

重叠计算两模态的 Q/K，以对称交错！

How: phase 1: Modal X stream 开始，

core 0–3: Gene. Q_X

core 4–5: Gene. K_Y

core 6–15: Compute. $Q_X \cdot K_Y^T$

Q_X 先完成计算

phase 2: Symmetric Modal Overlap.

Q_X 完成，core 0–1：去完成 Gene. Q_Y

在 Gene. K_Y 的同时 生成 Q_Y

phase 3:

core 4–5: 生成 K_X

core 6–15: Compute $Q_Y \cdot K_X^T$.

Q_X, K_X / Q_Y, K_Y 生成时间大致相同.

bit serial 计算 (传统方法)

在 INT8 精度下，每个数据为 8bit，要 8 个 cycle 完成计算

EB Balanced:

Idea: 如何： I_0 I_1 I_2 I_3

$EB\_I_0 = 4$ $EB\_I_1 = 1$ ....

解法：将 I_3 的高位 放到 I_1 上，一起计算

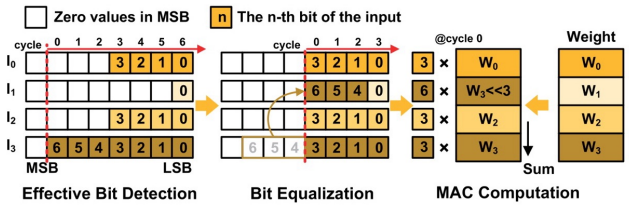


Fig. 13. EB-balanced multiplication.

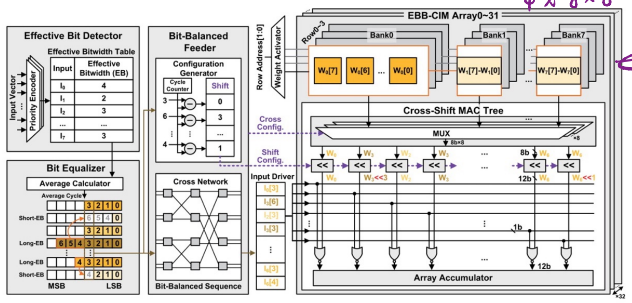


Fig. 14. EBB-CIM macro architecture.

如何计算结果： $I_0[3]\cdot W_0 + I_3[6]\cdot(W_3<<3) + I_2[3]\cdot W_2 + I_3[3]\cdot W_3$

让每个 cycle 最大化利用可用的 bit.

EB Balanced mechanism:

Components: ① 32个 EBB-CIM array: 多路 输入信号

② Effective Bit Detector: 检测有效位

③ Bit Equalizer: 比对齐的权重 优化精度并进行 balance.

④ Bit-Balanced-Feeder: 发送 (MSB-LSB) 控制 MAC tree.

EBB array:

每个 array 有: ① 4rows. 64 cols. 分成 8 个 bank.

② Cross-shift MAC tree 用于权重切换 和 移位比较.

③ 4行为1组 CIM 运算.

4 rows, 8 bits, bank 4 × 8 × 8

Effective Bit Detector:

输入 8 个元素 (一个 vector)

用 priority Encoder 判断最高位 → EB.

Bit Equalizer:

比较元素以找到最低位元

Bit Balance Feeder

从 MSB 到 LSB 的发送.

signal
- Cross-configuration: 权重index在 MAC tree 交换
- Shift-configuration: 权重左移位数.

用于

MAC: 读取 8 个 weight 根据 cross / shift configuration 乘加

每个 bank 算出 8 个