FlashQLA：面向门控 DeltaNet 的高效上下文并行

Xi Lin[1,2,*] Chengruidong Zhang[2,*] Huiqiang Jiang[2] Xiao Li[2] Zekun Wang[2] Junda Feng[2] Yunfei Mao[2] Siqi Zhang[2] Siyu Wang[2] Man Yuan[2,†] Rui Men[2,†] Bo Zheng[2,†] Dayiheng Liu[2,†] Bohan Zhuang[1,†] Jianwei Zhang[2,†]

[1]浙江大学 [2]阿里巴巴集团 Token Hub

*贡献相同。†通讯作者。

arXiv:2611.32039v1 [cs.LG] 24 Nov 2026

摘要

门控 DeltaNet（GDN）通过循环状态更新实现线性计算复杂度，但块之间的顺序依赖限制了并行性，尤其是在小批量大小或每个 GPU 上注意力头级并行性有限的长上下文工作负载中。我们提出了 FlashQLA，这是一种面向 GDN 的上下文并行执行框架，能够释放额外的序列级并行性，同时降低边界状态预处理开销。FlashQLA 通过衰减感知准备降低边界状态预处理开销，并借助 Dual CP 在 GPU 之间扩展序列并行性；Dual CP 可在不同并行级别之间共享摘要计算。面向工作负载的 Auto CP 和上下文并行感知的内核协同设计进一步提升了执行效率。FlashQLA 实现了 1.7×–3.1× 倍前向加速，相较于单个 GPU 上的 Flash Linear Attention（FLA），反向加速最高可达 4.4× 倍。在 8 个 GPU 上、上下文长度最高达 1M 个词元时，Dual CP 的前向加速最高可达 4.8× 倍，而相较于 FLA KCP，前向与反向执行的综合加速比可达 4.4× 倍。代码见https://github.com/QwenLM/FlashQLA。

1 引言

线性注意力通过循环状态更新，以线性复杂度计算取代 Softmax 注意力中 token 之间的二次复杂度交互（Katharopoulos 等人，2020 年）。门控 DeltaNet（GDN）进一步结合 Delta 规则更新与数据依赖的状态衰减，使模型能够有选择地更新循环状态，同时控制历史信息的保留量（Yang 等人，2024年；2025 年）。GDN 已应用于大规模混合语言模型，从 Qwen3-Next 到 Qwen3.8 系列和 Qwen3.8-Flash-Next，凸显了它在高效长上下文建模中的重要性日益提升（QwenTeam，2025 年；2026c；b）。

高效的 GDN 预填充执行依赖于基于扩展 WY 表示的分块公式，该公式可利用适合 GPU 的矩阵乘法，在每个块内进行并行计算（Yang 等人，2025 年）。然而，循环状态仍必须按顺序在各块之间传递，

随着序列长度增加，形成了一条很长的依赖链。这种块间递归会成为长上下文执行的主要性能瓶颈。

当批量大小和注意力头数提供的并行性不足时，序列维度可以提供额外的并发性。上下文并行（CP）通过对序列进行分区来利用这一机会，从而实现并行执行。然而，每个分区都依赖于前序分区生成的输入状态。Kimi Context Parallel（KCP）（Kimi Team，2026 年）通过边界状态预处理解决这些依赖关系，该预处理包括状态摘要准备（用于计算局部摘要）和边界状态校正（用于重建分区输入状态）。然而，这种预处理本身可能成为主要瓶颈。例如，在具有 1M 个 token 的序列上使用 8 个 GPU 时，它占据了 Flash Linear Attention（FLA）（Yang and Zhang，2024年）中 KCP 前向执行时间的 50.8%，限制了通过上下文并行可实现的性能提升。

我们提出了*FlashQLA*，这是一种面向 GDN 的高效上下文并行执行框架，能够释放序列级并行性，同时降低其预处理开销。传统的状态摘要准备会处理分区内的每个块，而不考虑历史状态贡献受到门控衰减影响的程度。FlashQLA 通过衰减感知准备利用这种逐头衰减行为：对于衰减强烈的注意力头，截断其计算；对于具有长程依赖的注意力头，则保留全历史准备。这样既避免了不必要的计算，又显著降低了预处理开销，同时在实践中保持了足够的数值精度。

增加分区数量可以释放更多并行性，但也会增加预处理和调度开销。因此，FlashQLA 引入了*AutoCP*，利用轻量级延迟代理模型和感知波次搜索，确定卡内 CP 是否有益，并为每种工作负载选择高效的分区长度。

在更长的上下文长度下实现高效的 GDN 执行，还得益于将序列级并行扩展到多个 GPU。为此，FlashQLA 引入了 *Dual CP*，将卡内上下文并行与卡间上下文并行结合起来。简单地合并这两个层级会导致状态摘要准备工作重复。Dual CP 通过在两个层级之间共享摘要计算，并执行分层边界状态校正，避免了这种冗余，在增加每个 GPU 内序列级并行的同时避免冗余计算。

在卡内和卡间执行过程中，FlashQLA 进一步将其 GPU 内核与 CP 执行调度协同设计，融入代数化简、CP 感知融合和手动线程束专门化，以提升硬件利用率和执行效率。

在单个 GPU 上，FlashQLA 相较于 FLA 实现了 1.7×–3.1× 倍前向加速比，以及最高 4.4× 倍反向加速比。在扩展到 8 个 GPU、上下文长度达到 1M 词元时，Dual CP 相较于 FLA KCP 实现了最高 4.8× 倍前向加速比，以及 4.4× 倍的前向与反向联合执行加速比。

本报告作出以下贡献：

1. 我们提出了衰减感知准备，利用头级门控衰减大幅降低上下文并行的边界状态预处理开销。2. 我们开发了 *Auto CP*，这是一种工作负载自适应分区策略，综合考虑增加序列级并行性与更细粒度分区开销之间的权衡，从而在各种工作负载下实现有效的卡内上下文并行。3. 我们提出了 *Dual CP*，通过共享摘要计算和分层边界状态校正，高效整合卡内与卡间上下文并行，避免在不同并行级别之间重复计算。4. 我们通过代数化简、阶段感知融合和手动线程束专门化，结合 GPU 内核与 CP 执行调度进行协同设计。综合这些技术，FlashQLA 在单个 GPU 上相比 FLA 实现了 1.7×–3.1× 倍前向加速；在上下文长度最高达 1M 个词元时，使用八个 GPU 相比 FLA KCP 的加速比最高可达 4.8× 倍。

2 背景与问题表述

2.1 门控 DeltaNet

门控 DeltaNet（Yang 等人，2025 年）维护一个循环状态，该状态在每个词元处通过将数据依赖衰减与增量规则校正相结合来更新。令 $\mathbf{S}_t \in \mathbb{R}^{d_k \times d^v}$ 表示词元 t之后的循环状态，其中 $\mathbf{q}_t$、$\mathbf{k}_t \in \mathbb{R}^{1\times d_k}$、$\mathbf{v}_t \in \mathbb{R}^{1\times d_v}$分别表示查询、键和值，衰减门为 $\alpha_t \in (0,1)$，增量更新强度为 $\beta_t \in (0,1)$。门控增量规则为

$$\mathbf{S}_t = \alpha_t \left(\mathbf{I} - \beta_t \mathbf{k}_t^\top \mathbf{k}_t\right) \mathbf{S}_{t-1} + \beta_t \mathbf{k}_t^\top \mathbf{v}_t, \qquad \mathbf{o}_t = \mathbf{q}_t \mathbf{S}_t. \tag{1}$$

衰减门 α_t 控制前一状态的保留程度，而 β_t 决定沿当前键方向进行增量规则更新的强度。

逐词元递推在序列维度上本质上是顺序执行的。为展现并行性，GDN 将 C 个连续词元分组成一个块，并使用扩展 WY 表示来并行化每个块内的更新（Yang 等人，2025 年）。为简单起见，我们假设块大小均为 C的完整块；不完整的块可以通过填充或掩码处理。我们使用带方括号的下标 $[c]$ 来标记块，以区别于词元级下标。令 $\mathbf{Q}_{[c]}$、$\mathbf{K}_{[c]} \in \mathbb{R}^{C\times d_k}$ 和 $\mathbf{V}_{[c]} \in \mathbb{R}^{C\times d_v}$ 分别表示块 c的查询、键和值，并令 $\alpha_{[c],t}$ 和 $\beta_{[c],t}$ 分别表示局部词元位置 $t = 1$、……、C处的衰减门和更新强度。我们定义

$$\gamma_{[c],t} := \prod_{j=1}^{t} \alpha_{[c],j}, \qquad \mathbf{B}_{[c]} := \operatorname{diag}\left(\beta_{[c],1}, \ldots, \beta_{[c],C}\right), \qquad \mathbf{\Gamma}_{[c]} := \operatorname{diag}\left(\gamma_{[c],1}, \ldots, \gamma_{[c],C}\right). \tag{2}$$

令 $\mathbf{S}_{[c]}$ 表示处理前 c 个块后的循环状态，其中 $\mathbf{S}_{[0]} = \mathbf{S}_0$。分块式

GDN 表述为

$$\mathbf{A}_{[c]} = \left(\mathbf{I} + \mathrm{StrictLower}\left(\mathbf{B}_{[c]}\boldsymbol{\Gamma}_{[c]}\mathbf{K}_{[c]}\mathbf{K}_{[c]}^{\top}\boldsymbol{\Gamma}_{[c]}^{-1}\right)\right)^{-1}, \tag{3}$$

$$\mathbf{U}_{[c]} = \mathbf{甲}_{[c]}\mathbf{乙}_{[c]}\mathbf{丙}_{[c]}, \tag{4}$$

$$\mathbf{W}_{[c]} = \mathbf{A}_{[c]}\mathbf{B}_{[c]}\boldsymbol{\Gamma}_{[c]}\mathbf{K}_{[c]}, \tag{5}$$

$$\mathbf{S}_{[c]} = \gamma_{[c],C}\mathbf{S}_{[c-1]} + \gamma_{[c],C}\mathbf{K}_{[c]}^{\top}\boldsymbol{\Gamma}_{[c]}^{-1}\left(\mathbf{U}_{[c]} - \mathbf{W}_{[c]}\mathbf{S}_{[c-1]}\right), \tag{6}$$

$$\mathbf{O}_{[c]} = \boldsymbol{\Gamma}_{[c]}\mathbf{Q}_{[c]}\mathbf{S}_{[c-1]} + \boldsymbol{\Gamma}_{[c]}\,\mathrm{Lower}\left(\mathbf{Q}_{[c]}\mathbf{K}_{[c]}^{\top}\right)\boldsymbol{\Gamma}_{[c]}^{-1}\left(\mathbf{U}_{[c]} - \mathbf{W}_{[c]}\mathbf{S}_{[c-1]}\right). \tag{7}$$

其中，$\mathbf{A}_{[c]}$，$\mathbf{U}_{[c]}$，和 $\mathbf{W}_{[c]}$ 是块内辅助量，而 StrictLower 和 Lower 分别表示矩阵的严格下三角部分和下三角部分。

FLA（Yang and Zhang，2024年）通过多个 GPU 内核阶段实现分块式 GDN。在作为我们基线的 FLA 实现中，其前向执行通过部分融合的块内操作构造扩展 WY 表示，随后使用独立的内核进行循环状态传播和输出计算。这种多阶段设计要求在各阶段之间物化中间张量，从而引入额外的内核启动和全局内存访问。

尽管扩展 WY 表示实现了每个块内的计算并行化，但块边界状态仍然是顺序的：$\mathbf{S}_{[c+1]}$ 依赖于$\mathbf{S}_{[c]}$。因此，同一序列中的各个块无法在首先解决其边界状态依赖关系之前独立处理，这促使我们采用上下文并行执行。

2.2 GDN 上下文并行的边界状态预处理

要将上下文并行应用于 GDN，需要将块序列划分为多个子序列，但每个分区的输入状态取决于其前面的块。KCP（KimiTeam，2026 年）通过先构建每个分区的局部状态转移，再确定相应的边界状态，解决了这一依赖问题。边界状态预处理分为两个阶段：状态摘要准备会计算每个分区的零初始化终止状态和累积转移，而边界状态校正则会在并行执行前确定每个分区的输入状态。

公式（6）中的分块状态更新关于其输入状态是仿射的：

$$\begin{aligned}
\mathbf{S}_{[c]} &= \mathbf{M}_{[c]}\mathbf{S}_{[c-1]} + \mathbf{E}_{[c]}, \\
\mathbf{M}_{[c]} &:= \gamma_{[c],C}\left(\mathbf{I} - \mathbf{K}_{[c]}^{\top}\boldsymbol{\Gamma}_{[c]}^{-1}\mathbf{W}_{[c]}\right), \\
\mathbf{E}_{[c]} &:= \gamma_{[c],C}\mathbf{K}_{[c]}^{\top}\boldsymbol{\Gamma}_{[c]}^{-1}\mathbf{U}_{[c]}.
\end{aligned} \tag{8}$$

其中，$\mathbf{M}_{[c]}$ 将输入状态通过块c进行传播，而$\mathbf{E}_{[c]}$ 表示与该状态无关的块局部贡献。

这种仿射递推的一个重要性质是，两条状态轨迹之间的差异会独立于加性贡献而演化~。令$\mathbf{S}_{[c]}$ 表示使用相同块输入的零初始化~轨迹所得到的状态，并定义 $\Delta\mathbf{S}_{[c]} := \mathbf{S}_{[c]} - \mathbf{S}_{[c]}$。处理块$a$ 至b 后的差异满足

$$\Delta\mathbf{S}_{[b]} = \mathbf{M}_{[b]}\cdots\mathbf{M}_{[a]}\Delta\mathbf{S}_{[a-1]}. \tag{9}$$

该性质表明，在任意初始条件下，终端状态都可以通过零状态展开及相应的累积转移重建。详细推导见附录 D。

将 L 个块划分为 N 个有序分区，其中第 i 个分区包含 l_i 个连续块，且 $\sum_{i=1}^{N} l_i = L$。我们使用上标 (i) 标识分区，并使用带方括号的下标 [j] 对每个分区内的块进行局部索引。令 $S(i)_{[0]}$ 表示分区 i 的输入状态，令 $S(i)_{[j]}$ 表示处理其前 j 个块后的状态。输入边界状态满足 $S(1)_{[0]} = S_0$，且 $S(i)_{[0]} = S(i-1)_{[l_{i-1}]}$，其中 $i > 1$。

在状态摘要准备期间，每个分区独立执行零状态展开，生成~$\mathbf{S}^{(i)}_{[l_i]}$，其中 ~$\mathbf{S}^{(i)}_{[0]} = \mathbf{0}$，以及累积转移

$$\mathbf{M}^{(i)}_{[l_i \leftarrow 1]} := \mathbf{M}^{(i)}_{[l_i]}\cdots\mathbf{M}^{(i)}_{[1]}. \tag{10}$$

应用式（9），分区 i 的最终状态为

$$\mathbf{S}^{(i)}_{[l_i]} = \widetilde{\mathbf{S}}^{(i)}_{[l_i]} + \mathbf{M}^{(i)}_{[l_i \leftarrow 1]} \mathbf{S}^{(i)}_{[0]}. \tag{11}$$

因此，状态摘要准备会在各分区间并行进行，每个分区计算两个摘要：$\widetilde{\mathbf{S}}^{(i)}_{[l_i]}$ 和 $\mathbf{M}^{(i)}_{[l_i \leftarrow 1]}$。从 $\mathbf{S}^{(1)}_{[0]} = \mathbf{S}_0$, 开始，边界状态校正会跨分区摘要依次应用式（11），以确定后续分区的输入状态。预处理完成后，所有分区的主要计算即可并行进行。

尽管上下文并行能够为主要计算提供额外的序列级并行性，但也会带来大量边界状态预处理开销。具体而言，状态摘要准备需要在每个分区内的块上额外执行顺序递推，其计算成本与主要前向传播相当。无论分区是在 GPU 内部还是跨 GPU 分布，这种开销都可能显著削弱并行执行带来的性能提升。在 8 个 GPU 上处理 1M 个 token 的序列时，边界状态预处理占 FLA KCP 前向执行时间的 50.8%，凸显了预处理开销是实现高效上下文并行执行的一大关键瓶颈。

3 衰减感知准备

如第 2.2 节所述，传统的边界状态预处理需要对每个分区内的所有块进行状态摘要预处理。FlashQLA 利用门控衰减的头级变化来降低这项开销：对衰减较强的注意力头采用截断准备，同时对具有长程依赖的注意力头保留全历史准备。

3.1 头级门控衰减特性

我们首先考察 GDN 各注意力头中的历史信息衰减速度。给定注意力头 h 的按块累积衰减因子 $\gamma_{[c],C,h}$，我们定义

$$D_h(c-k+1, c) := \prod_{r=c-k+1}^{c} \gamma_{[r],C,h}, \tag{12}$$

$$K_{c,h} := \min\left\{1 \le k \le c : D_h(c-k+1, c) < \mathrm{TOL}\right\}, \tag{13}$$

$$W_{c,h} := CK_{c,h}, \tag{14}$$

其中，TOL 是累积衰减阈值，$K_{c,h}$ 表示按块计算的衰减窗口，$W_{c,h}$ 表示相应的词元窗口。衰减窗口越小，历史信息衰减越快。如果在可用历史范围内没有窗口满足该阈值，我们就设置 $K_{c,h} = \infty$ 并采用全历史准备。

为表征短衰减窗口的普遍程度，我们在 TOL $= e^{-10}$ 的条件下，评估 Qwen3.8-27B 和 Qwen3.8-2.4T-A95B（Qwen Team，2026c），使用 32K词元序列，并测量每层中衰减窗口短于 1K、2K、8K 和 32K 词元的注意力头占比。如图 1 所示，两个模型及其各层中都普遍存在短衰减窗口。具体而言，Qwen3.8-27B 中有 66.9% 的注意力头、Qwen3.8-2.4T-A95B 中有 80.7% 的注意力头，其衰减窗口短于 8K 词元。这些结果表明，快速门控衰减在 GDN 中十分普遍，因此可通过衰减感知计算大幅降低边界状态准备开销。

我们进一步考察每个注意力头的衰减行为如何随输入而变化。对于每个模型，我们收集 24 个长度为 32K 词元的序列中的对数门控值，并根据其中位数估计每个注意力头的衰减窗口，假设在累积衰减达到 TOL $= e^{-10}$ 之前，衰减率保持恒定。
如图 2所示，注意力头在不同输入上呈现出具有特征性的衰减模式，但其确切的窗口大小取决于输入，这促使我们采用在线窗口估计。
这些观察结果促使我们根据各个注意力头依赖输入的衰减行为来调整状态摘要准备。接下来，我们将考察门控衰减如何削弱历史状态贡献，并实现截断准备。

3.2 门控衰减与有效历史

考虑 t　注意力头 h 在块 b之后的边界，其衰减窗口为 $K_{b,\ h}$ 定义见公式（13）。T　o
检查 t　截断历史计算的影响，我们仅保留最近的 K_{bh} 块　s,

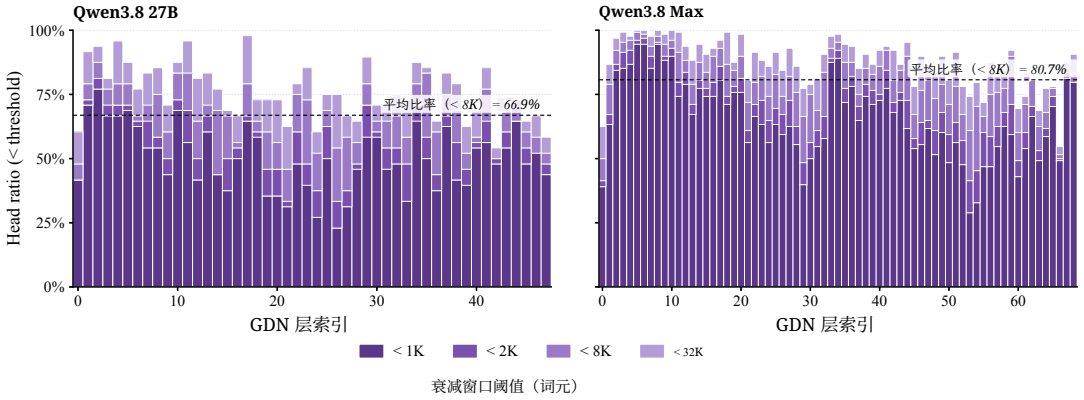


图1：**逐注意力头衰减窗口分布**。在 Qwen3.8-27B 和 Qwen3.8-2.4T-A95B 的各层中，衰减窗口短于 1K、2K、8K 和 32K 词元的注意力头比例；在 32K词元序列上使用 TOL $= e^{-10}$进行评估。总体而言，在这两个模型中，分别有 66.9% 和 80.7% 的注意力头的衰减窗口短于 8K 词元，凸显了短衰减窗口的普遍性。

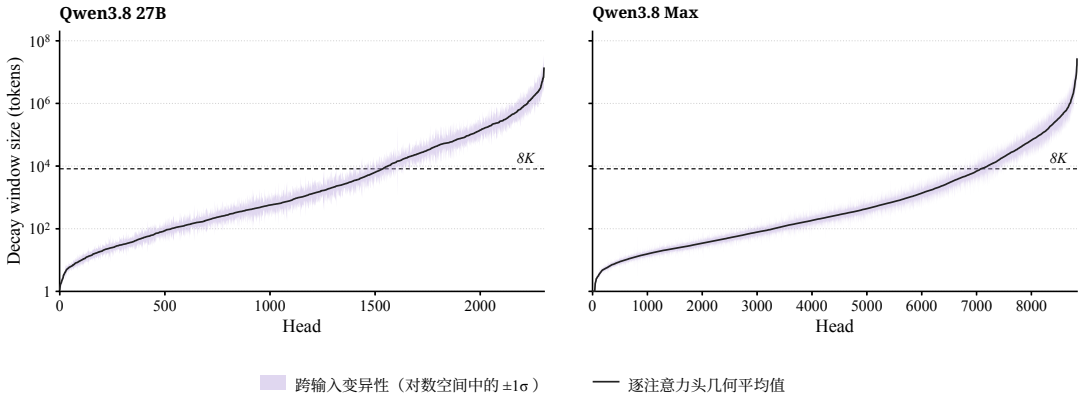


图 2：**逐注意力头衰减窗口的输入依赖性**。针对 Qwen3.8-27B 和 Qwen3.8-2.4T-A95B 的 24 个输入所估计的衰减窗口。曲线表示几何平均值，阴影区域表示对数空间中的±1 标准差。超过 32K 的窗口根据实测门控值进行外推。

从 $b - K_{b,h} + 1$ 到 b，并将该窗口起始处的状态初始化为零。截断轨迹和完整历史轨迹处理相同的块输入，但起始状态不同。我们将公式（8）中的块转移分解为累积门衰减因子和内容相关变换：

$$\mathbf{M}_{[c]} = \gamma_{[c],C}\,\overline{\mathbf{M}}_{[c]}, \qquad \overline{\mathbf{M}}_{[c]} := \mathbf{I} - \mathbf{K}_{[c]}^{\top}\mathbf{\Gamma}_{[c]}^{-1}\mathbf{W}_{[c]}. \tag{15}$$

令 $\Delta\mathbf{S}_{[b-K_{b,h}]}$ 表示保留窗口开始处完整历史状态与截断历史状态之间的差异。根据公式（9）中的状态差异传播性质，在块b 之后边界处的差异为

$$\Delta\mathbf{S}_{[b]} = \left(\prod_{c=b-K_{b,h}}^{b} \gamma_{[c],C}\right) \overline{\mathbf{M}}_{[b]} \cdots \overline{\mathbf{M}}_{[b-K_{b,h}]} \Delta\mathbf{S}_{[b-K_{b,h}]}, \tag{16}$$

这种分解表明，累积门衰减会减弱保留窗口之外历史状态的影响，因此需要进行截断状态摘要准备。FlashQLA 使用一个较小的 TOL 阈值 $= e^{-10}$ 来确定截断窗口。

3.3 衰减感知的状态摘要准备

基于上述讨论的门衰减特性，FlashQLA 会为每个边界和注意力头自适应地选择准备窗口。由于门控值已经可用，因此可以通过对数衰减因子的累积和在线计算衰减窗口 $K_{c,h}$。我们将最大截断

窗口 $K_{\max}$ 设为前一个分区的长度，并将每个边界-注意力头对分类为：当 $K_{c,h} \le K_{\max}$ 时为强衰减，否则为弱衰减 。

为便于说明，省略注意力头索引，并令 $K_i := K_{b_i,h}$ 表示分区 i输入边界处的衰减窗口，其中 $b_i = \sum_{p=1}^{i-1} l_p$ 是前一分区最后一个块的全局索引。对于强衰减边界，FlashQLA 仅对前一分区的最后 K_i 个块执行零状态展开，从而得到截断状态 $\widetilde{\mathbf{S}}_{(i-1)\,[l_{i-1}],K_i}$。所得的近似边界状态为

$$\widehat{\mathbf{S}}_{[0]}^{(i)} = \begin{cases} \widetilde{\mathbf{S}}_{[l_{i-1}],K_i}^{(i-1)}, & K_i \le K_{\max}, \\ \widetilde{\mathbf{S}}_{[l_{i-1}]}^{(i-1)} + \mathbf{M}_{[l_{i-1}\leftarrow 1]}^{(i-1)} \widehat{\mathbf{S}}_{[0]}^{(i-1)}, & K_i > K_{\max}, \end{cases} \qquad i > 1, \tag{17}$$

其中 $\widehat{\mathbf{S}}_{(1)\,[0]} = \mathbf{S}_0$。对于强衰减边界，FlashQLA 通过截断零状态展开降低预处理成本；而对于弱衰减边界，则保留全历史准备，以保持长程依赖。所得的 $\widehat{\mathbf{S}}_{(i)\,[0]}$ 在实践中以有限的数值误差逼近真实边界状态 $\mathbf{S}_{(i)\,[0]}$，并作为后续分区并行计算的初始状态。

4 Auto CP

卡内上下文并行的收益很大程度上取决于工作负载。更细粒度的序列分区能够暴露更多并行性，但也会增加预处理和调度开销。因此，无论是在所有情况下都启用上下文并行，还是使用固定的分区大小，都无法在不同的输入配置下实现最优。为此，FlashQLA 引入了*Auto CP*，利用轻量级性能代理模型，自适应地确定是否以及如何对每个工作负载应用卡内上下文并行。

为适应不同长度的序列，我们通过最大分区长度$l_{\max}$（以块为单位）对上下文并行分区进行参数化。每个序列都会被划分为连续的分区，每个分区最多包含$l_{\max}$ 个块，最后一个分区的长度可能更短。一个批次中的所有序列共享相同的$l_{\max}$，而每个序列的分区数量取决于其长度。

4.1 内核延迟建模

我们将每个 GDN 内核的执行建模为一组协作线程数组（CTA），每个 CTA 执行数量取决于工作负载的连续计算步骤。令x 表示一种执行配置，指定输入形状、是否启用上下文并行，以及适用时的分区长度。对于配置为x的内核κ，令$k_{\kappa,r}(x)$ 表示 CTA r执行的步骤数，并令$R_\kappa(x)$ 表示 CTA 的总数。

上下文并行会以不同方式影响这些工作负载。对于循环内核和准备内核，使用更细的序列分区可以减少顺序步数，同时增加 CTA 数量。相比之下，随着分区数量增加，顺序校正阶段会变得更长。因此，选择合适的分区大小需要在这些相互对立的影响之间取得平衡。

Auto CP 使用关键调度深度、CTA 总数和固定开销项来近似内核延迟：

$$\widehat{t}_\kappa(x) = a_\kappa d_\kappa(x) + b_\kappa R_\kappa(x) + c_\kappa, \tag{18}$$

其中，$d_\kappa(x)$ 表示关键调度深度。系数 a_κ、b_κ 和 c_κ 针对每个内核族和硬件配置分别进行拟合。

在简化调度模型下，我们通过将 CTA 贪心地分配给累计步数最少的流式多处理器，并取各流式多处理器上的最大累计工作负载，来近似关键调度深度：

$$d_\kappa(x) := \max_{s=1,\dots,N_{\text{sm}}} \sum_{r \in \mathcal{A}_s(x)} k_{\kappa,r}(x), \tag{19}$$

w其中，N_{sm} 表示流式多处理器的数量，$\mathcal{A}_s(x)$ 表示分配给流式多处理器 s 的 CTA 集合。

每个执行配置都由一系列内核调用组成，各阶段之间存在依赖关系。令 $\mathcal{K}$（x）表示配置 x所需的内核调用序列。其总延迟估计为

$$\widehat{T}(x) = \sum_{\kappa \in \mathcal{K}(x)} \widehat{t}_\kappa(x). \tag{20}$$

模型系数通过对少量实测配置进行最小二乘法拟合离线获得。完成校准后，该模型可为候选配置提供低成本的在线延迟估计。

4.2 感知执行波的 PartitionSearch

对于给定的工作负载，Auto CP 会寻找使 $\widehat{T}(x(l_{\max}))$ 最小的子序列长度 $l_{\max}$，其中$x(l_{\max})$ 表示分区长度为 $l_{\max}$ 的 CP 配置。然而，评估 $\mathcal{L}$ 中所有支持的长度会增加在线搜索开销。由于 CTA 数量跨越执行波边界时，内核延迟可能会急剧变化，因此 Auto CP 会重点关注这些边界附近的候选长度。

对于一个包含 B 个序列的批次，这些序列的长度为 L_b，并且有 H 个独立注意力头，循环内核会为每个分区-头对分配一个 CTA。采用这种布局时，能够大致填满 w 个执行波的子序列长度为

$$l_{\max}^{(w)} \approx \frac{H \sum_{b=1}^{B} L_b}{w N_{\mathrm{sm}}}. \tag{21}$$

此估算会选择一个分区长度，使工作负载分布到约 wN_{sm} 个 CTA 上，从而填满 w 个执行波。

Auto CP 按升序枚举执行波数量 w，并将每个由此得到的 $l_{(w)}\max$ 映射到 $\mathcal{L}$ 中附近的支持长度，直到候选集达到预设大小上限。随后，延迟模型会从所得候选集 $\mathcal{L}_{\mathrm{wave}}$ 中选出最佳长度：

$$l_{\max}^{*} = \underset{l_{\max} \in \mathcal{L}_{\mathrm{wave}}}{\arg\min}\ \widehat{T}(x(l_{\max}))\,. \tag{22}$$

这种有界搜索限制了在线决策开销。第 7.5 节表明，Auto CP 选出的配置延迟接近穷举最优值，体现了搜索成本与执行性能之间实用的权衡。

4.3 CP 启用

当工作负载本身已提供足够的并行性时，CP 带来的收益有限，却会增加额外的预处理和调度开销。因此，Auto CP 会将预测性能最佳的 CP 配置与原始无 CP 执行进行比较，并且仅在预计能够降低延迟时启用 CP。

Auto CP 根据预测延迟，在最佳 CP 配置 $x(l^{*}\max)$ 和无 CP 配置 $x_{\varnothing}$ 之间进行选择：

$$x^{*} = \begin{cases} x(l_{\max}^{*}), & \widehat{T}(x(l_{\max}^{*})) < \widehat{T}(x_{\varnothing}), \\ x_{\varnothing}, & \text{otherwise.} \end{cases} \tag{23}$$

因此，Auto CP 会共同决定是否启用卡内上下文并行，并在有益时选择分区长度。

5 Dual CP

将 GDN 扩展到更长的上下文，需要将序列并行性扩展到单个 GPU 之外。现有的卡间方法（如 KCP）会为每个 GPU 分配一个序列分区，计算局部状态和转移摘要，并通过全收集交换这些信息，以进行边界状态校正。然而，当批量大小和头数较小时，仅依靠卡间 CP 可能无法在每个 GPU 内提供足够的并行性。因此，需要卡内 CP，以在每个 GPU 内暴露额外的序列级并行性。

一种直接的方法是先执行卡间 CP，以确定每个 GPU 的输入边界状态，然后对其本地序列应用卡内 CP。然而，这两个层级会在相同的块上分别执行~状态摘要准备，从而重复计算状态摘要 **S** 和转移乘积 **M**。

因此，我们提出*Dual CP*，它在卡内分区上执行共享摘要计算，并通过轻量级聚合生成卡间摘要。这些摘要支持在两个层级上进行分层边界状态校正，避免重复的预处理，同时在每个 GPU 内释放额外的序列级并行性。图3 展示了共享计算和分层校正。

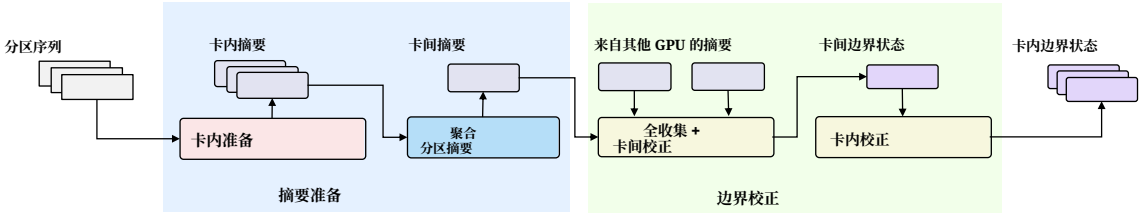


图 3：**Dual CP 概览**。卡内分区摘要仅计算一次，并聚合为卡间摘要，从而实现共享摘要计算和分层边界状态校正，避免重复预处理。

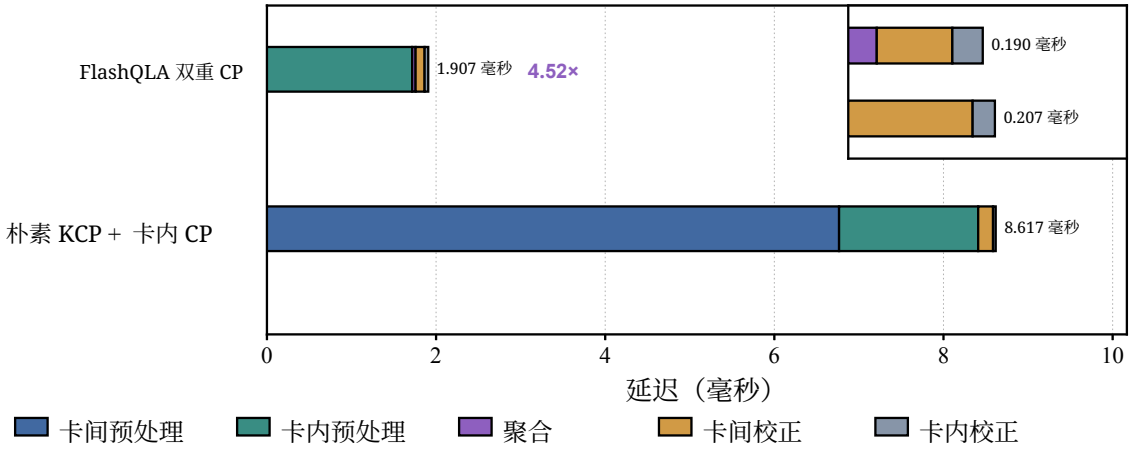


图 4：**Dual CP 的预处理延迟分解**。在 8 个 GPU 上，针对包含 1M 个令牌和 32 个注意力头的序列，将其与卡间 CP 和卡内 CP 的朴素组合进行比较。Dual CP 将总预处理延迟从 8.617 ms 降至 1.907 ms（4.52×）。右上方面板放大显示了较小的聚合和校正阶段。

图4 比较了在 8 个 GPU 上处理 1M 个 token 的序列和 32 个注意力头时，Dual CP 与卡间 CP 和卡内 CP 的朴素组合之间的预处理延迟。朴素方法由于重复的状态摘要准备而产生了 substantial 开销，而 Dual CP 通过共享摘要计算消除了这种冗余，仅产生少量聚合开销。因此，Dual CP 将总预处理延迟从 8.617 ms 降至 1.907 ms，实现了 4。52×倍加速比。

5.1 共享摘要计算

令 N_{inter} 表示 GPU 数量。GPU i 拥有一个包含 l_i 个块的卡间分区，该分区被划分为 N^i_{intra}个有序卡内分区，其长度为 $l_{i,j}$，其中 $l_i = \sum^{N^i_{intra}}_{j=1} l_{i,j}$。我们分别使用上标 (i) 和 (i, j) 标识卡间分区和卡内分区，同时使用方括号下标标记每个分区内的块。按照第 2.2 节的内容，每个卡内分区独立计算零初始化终止状态 $\widetilde{S}(i,j)_{[l_{i,j}]}$ 以及累积转移 $M(i,j)_{[l_{i,j}\leftarrow 1]}$。这些摘要共同刻画了仿射状态转移：

$$\mathbf{S}^{(i,j)}_{[l_{i,j}]} = \widetilde{\mathbf{S}}^{(i,j)}_{[l_{i,j}]} + \mathbf{M}^{(i,j)}_{[l_{i,j}\leftarrow 1]} \mathbf{S}^{(i,j)}_{[0]}. \tag{24}$$

由于每个局部摘要都表示一次仿射状态转移，因此可以组合这些摘要来构建卡间摘要，而无需重新处理原始块。令 $L_{i,j} := \sum^{j}_{\ell=1} l_{i,\ell}$ 表示 GPU i 上前 j 个卡内分区的累积长度，其中 $L_{i,0} := 0$ 且 $L_{i,N^i_{intra}} = l_i$。将空前缀摘要初始化为 $\widetilde{\mathbf{S}}^{(i)}_{[0]} = \mathbf{0}$ 和 $\mathbf{M}^{(i)}_{[0\leftarrow 1]} = \mathbf{I}$ 后，我们按

分区顺序聚合：

$$\widetilde{\mathbf{S}}^{(i)}_{[L_{i,j}]} = \widetilde{\mathbf{S}}^{(i,j)}_{[l_{i,j}]} + \mathbf{M}^{(i,j)}_{[l_{i,j}\leftarrow 1]}\widetilde{\mathbf{S}}^{(i)}_{[L_{i,j-1}]}, \tag{25}$$

$$\mathbf{M}^{(i)}_{[L_{i,j}\leftarrow 1]} = \mathbf{M}^{(i,j)}_{[l_{i,j}\leftarrow 1]}\mathbf{M}^{(i)}_{[L_{i,j-1}\leftarrow 1]}. \tag{26}$$

在聚合所有 N^i_{intra} 分区后，我们得到 $\widetilde{\mathbf{S}}_{(i)[Li,N^i_{\text{intra}}]} = \mathbf{S}_{(i)[l_i]}$ 以及 $\mathbf{M}_{(i)[Li,N^i_{\text{intra}}\leftarrow 1]} = \mathbf{M}_{(i)[l_i\leftarrow 1]}$，它们构成了整个卡间分区的状态摘要。

因此，Dual CP 通过组合卡内仿射摘要来生成卡间摘要，无需再对原始块执行单独的准备遍历。由于聚合操作针对的是分区级摘要，而非单个块，其成本随卡内分区的数量扩展，并且仅引入有限的额外开销。

5.2 分层边界状态校正

聚合后，Dual CP 执行分层边界状态校正。通过全收集交换卡间摘要。全收集之后，每个 GPU 在本地计算有序的卡间仿射递推，以获得自身的初始状态。随后，这些状态将用于初始化卡内校正，从而计算本地分区的输入状态。

对于卡间校正，输入状态使用聚合后的摘要通过仿射递推计算得到，起始于 $\mathbf{S}_{(\mathbf{1})[0]} = \mathbf{S}_0$：

$$\mathbf{S}^{(i+1)}_{[0]} = \widetilde{\mathbf{S}}^{(i)}_{[l_i]} + \mathbf{M}^{(i)}_{[l_i\leftarrow 1]}\mathbf{S}^{(i)}_{[0]}, \qquad i = 1, \ldots, N_{\text{inter}} - 1. \tag{27}$$

对于卡内校正，每个 GPU 初始化 $\mathbf{S}_{(i,\mathbf{1})[0]} = \mathbf{S}_{(i)[0]}$，并将相同的关系应用于其局部摘要：

$$\mathbf{S}^{(i,j+1)}_{[0]} = \widetilde{\mathbf{S}}^{(i,j)}_{[l_{i,j}]} + \mathbf{M}^{(i,j)}_{[l_{i,j}\leftarrow 1]}\mathbf{S}^{(i,j)}_{[0]}, \qquad j = 1, \ldots, N^i_{\text{intra}} - 1. \tag{28}$$

各 GPU 独立执行局部校正。一旦初始状态计算完成，所有分区便可并行执行主要计算。

6 实现与内核设计

FlashQLA 通过结合 CP 的执行特性与内核级优化技术，为上下文并行 GDN 提供高效的 GPU 内核实现。该实现集成了代数化简、CP感知内核融合和手动线程束专门化。

6.1 代数化简

在分块式 GDN 表述中，累积门控因子既出现在公式（3）的三角逆矩阵中，也出现在周围的状态更新操作中。FlashQLA 将这些项从逆矩阵中提取出来，以显式展现代数消去并消除冗余计算。

我们定义与门无关的三角逆

$$\mathbf{A}'_{[c]} = \left(\mathbf{I} + \text{StrictLower}\left(\mathbf{B}_{[c]}\mathbf{K}_{[c]}\mathbf{K}^\top_{[c]}\right)\right)^{-1}. \tag{29}$$

原始的与门相关的逆$\mathbf{A}$ 随后可表示为

$$\mathbf{A} = \mathbf{\Gamma}_{[c]}\mathbf{A}'_{[c]}\mathbf{\Gamma}^{-1}_{[c]}. \tag{30}$$

将公式（30）代入分块式状态更新后，可以消除冗余的与门相关缩放，并简化块转移的计算。附录 B给出了推导过程以及所得的状态更新方程。

这种重新表述也提高了边界状态预处理中累积转换计算的数值稳定性。具体而言，块转移可以分解为

$$\mathbf{M}_{[c]} = \gamma_{[c],C}\underbrace{\left(\mathbf{I} - \mathbf{K}^\top_{[c]}\mathbf{A}'_{[c]}\mathbf{B}_{[c]}\mathbf{K}_{[c]}\right)}_{\overline{\mathbf{M}}_{[c]}}. \tag{31}$$

通过将门控因子与转移矩阵分离，FlashQLA 避免在低精度矩阵更新中反复纳入累积门衰减。

然而，通过低精度矩阵乘法直接累积与门无关的转移，会反复对累积结果进行舍入，从而在较长分区中引入数值误差。FlashQLA 则将转移累积重新表述为加性更新，使低精度矩阵乘法成为可能，同时将累积矩阵保留在 FP32 中：

$$\overline{\mathbf{M}}^{(i)}_{[j\leftarrow 1]} = \overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]} - \mathbf{K}^{(i)\top}_{[j]} \mathbf{A}'^{(i)}_{[j]} \mathbf{B}^{(i)}_{[j]} \mathbf{K}^{(i)}_{[j]} \overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]}. \tag{32}$$

这种表述方式使累积转移能够在多次更新过程中保留于 FP32 累加器中，避免反复进行低精度舍入，同时保留高效的矩阵乘法。附录 C给出了推导过程和数值稳定性分析。

6.2 CP 感知的内核融合

上下文并行GDN包含具有不同并行性和同步要求的计算阶段。因此，FlashQLA 按块、CP分区和序列粒度组织 GPU 内核，在融合兼容操作的同时，保留阶段之间的同步。

在前向传播中，块内操作（包括 $\mathbf{KK}^\top$ 计算和扩展的 WY 三角求解）使用块级网格融合到单个内核中。边界状态预处理和校正则作为独立内核实现，因为二者运行于不同的并行粒度：预处理使用 CP分区级网格，而校正使用序列级网格来解决跨分区的依赖关系。边界状态就绪后，循环状态传播和输出计算会在 CP 分区级别融合到单个内核中。

这种阶段感知融合减少了中间结果物化和内核启动开销，同时使每个内核的执行粒度与 CP 所暴露的并行性相匹配。批次维度和注意力头维度则为这些阶段提供了额外的并行性。

6.3 手动线程束专门化

FlashQLA 在 TileLang 中采用手动线程束专门化，在融合内核内重叠数据移动、矩阵乘法和逐元素计算。专用生产者线程束会为后续计算预加载数据，而消费者线程束组则处理可用数据。在依赖关系允许的情况下，消费者调度还会进一步重叠矩阵运算与独立的逐元素计算。

这种设计利用了内核内并发性，并补充了 CP 所暴露的序列级并行性。更多实现细节，包括流水线调度和线程束组分配，可在 FlashQLA技术博客中查看QwenTeam (2026a)。

7 评估

我们在一系列工作负载上评估 FlashQLA，这些工作负载具有不同的序列长度、批量大小和头数。我们首先将其单 GPU 和多 GPU 性能与现有实现进行比较，随后分解各项优化，以量化单项技术的贡献。然后，我们通过延迟模型准确性、配置选择质量以及与静态分区策略的性能比较来分析 Auto CP。

7.1 实验设置

所有实验均在一台配备八块 GPU 的服务器上进行。我们将使用 TileLang（v0.1.12）实现的 FlashQLA（v0.1.4）（Wang 等人，2026 年）与 Flash LinearAttention（FLA，v0.5.1）（Yangand Zhang，2024年）进行比较。除非另有说明，输入均使用 BFloat16，$d_k = d_v = 128$。各实验中将报告特定于工作负载的批量大小、头数和序列长度。

对于合成基准测试，我们生成在不同注意力头之间具有异质门控衰减行为的随机输入。根据真实模型中观察到的短衰减窗口的普遍性（第 3.2 节），我们采用一种具有代表性的配置：75%的注意力头表现出强衰减，其余25%的注意力头表现出较弱衰减。真实模型实验使用从Qwen3.8-2.4T-A95B执行过程中收集的激活值和门控值。

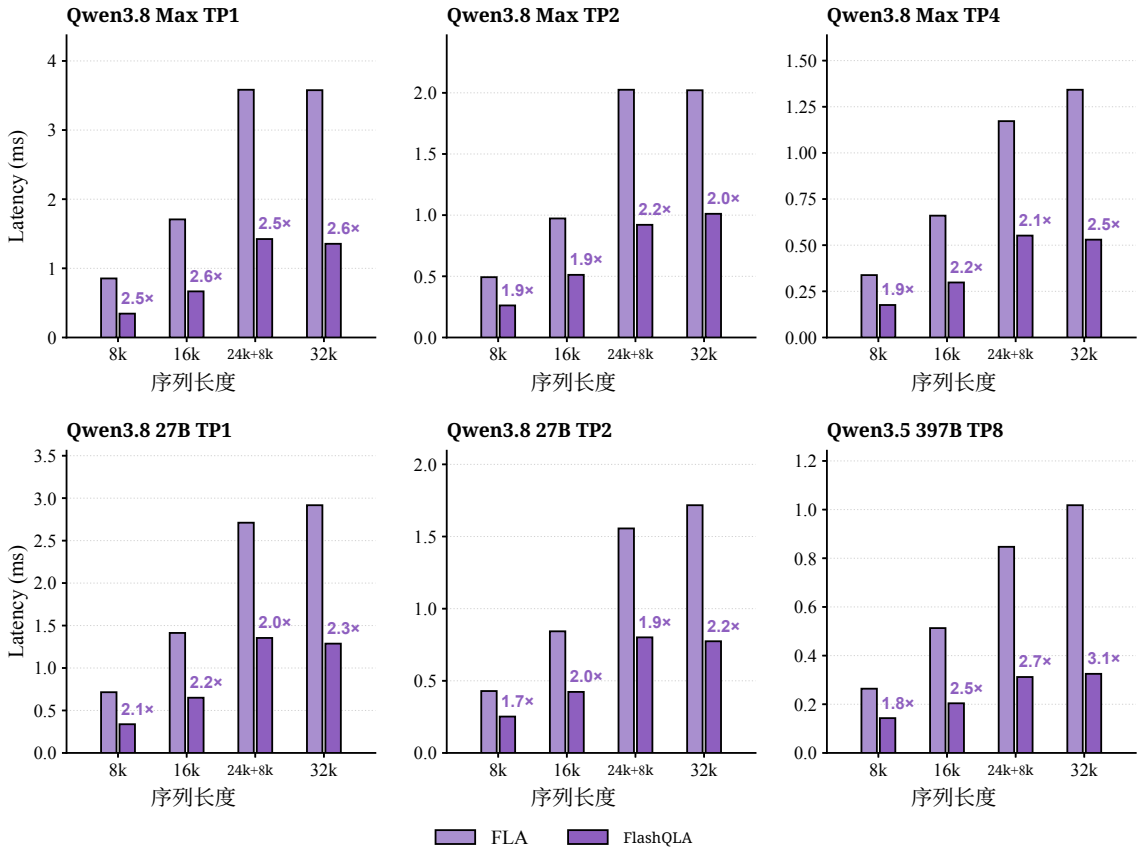


图 5：**单 GPU 前向性能**。比较 FlashQLA 和 FLA 在不同模型架构、张量并行配置以及 8K 至 32K 词元序列长度下的前向性能。FlashQLA 实现了 1.7×–3.1× 的加速比，且在更高 TP 度数和更长序列长度下收益更大。

所有报告的延迟均包含相应门控增量规则实现的完整执行过程，包括适用时的边界状态预处理和 GPU 间通信。我们在预热后使用 CUDA 事件测量执行延迟，并报告多次重复运行的中位数。

7.2 单卡性能

我们使用源自不同 Qwen 模型架构和张量并行（TP）配置的工作负载，对 FlashQLA 和 FLA 的前向执行与反向执行进行比较。对于前向执行，我们在不同每 GPU 头数下评估 8K 至 32K 词元的序列长度。对于反向执行，我们随机生成长度可变的序列批次，并将词元总数固定为 32K，以考察不同序列级并发程度下的性能。

图 5 和 6 表明，在所评估的工作负载中，FlashQLA 始终优于 FLA，前向加速比达到 1.7×–3.1×，并且在大多数配置下反向加速比超过 2×，最高达到 4.4×。在更高 TP 度数和更长序列长度下，前向收益更加显著。更高的 TP 度数会降低每 GPU 头数，使卡内 CP 更有优势；而更长的序列会产生更长的块间依赖链，这些依赖链可通过序列分区缩短。

对于反向执行，在总 token 数固定为 32K 的情况下，随着序列数量变化，FlashQLA 仍能保持相对稳定的延迟。相比之下，FLA 的延迟变化更大，并且序列数量越少，速度越慢，因为序列间并发性降低，使更多块间计算被串行化。通过暴露每个序列内部的额外并行性，FlashQLA 降低了对序列数量的敏感性，有助于缓解流水线并行训练中的流水线气泡。

为验证这些提升在真实模型工作负载中的效果，我们进一步使用从实际模型执行中收集的激活值和门控值，在 Qwen3.8-2.4T-A95B 上评估 FlashQLA。我们测量平均前向

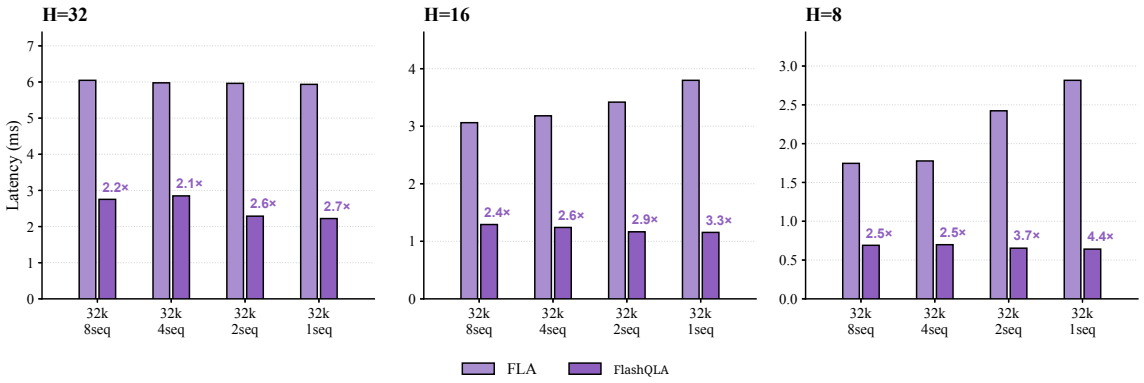


图6：**单 GPU 反向传播性能**。在模型架构和张量并行配置下，对 FlashQLA 与 FLA 的反向传播性能进行比较；总计固定为 32K 个词元，随机分布于 1、2、4 或 8 个序列中。FlashQLA 在不同序列数量下均保持稳定的延迟，而 FLA 在序列数量较少时会变慢。FlashQLA 在大多数配置下实现超过 2× 的加速比，最高可达 4.4×。

表1：**Qwen3.8-2.4T-A95B 上真实模型门控 delta 规则的性能**。在 TP8 下，使用真实模型激活值和门控值，统计 GDN 层中门控增量规则算子的平均前向延迟。

序列长度	FLA（毫秒）	FlashQLA（毫秒）	加速比
32K	2.309	0.711	3.25×
64K	4.630	1.299	3.56×

延迟：在 TP8 下，针对上下文长度为 32K 和 64K 词元的情况，测量所有 GDN 层中门控增量规则算子的延迟，并在输入和执行配置完全相同的条件下，将 FlashQLA 与 FLA 进行比较。
如表 1所示，FlashQLA 在 32K 和 64K 词元下分别实现了 3.25× 和 3.56× 的前向加速比。这些提升与我们的合成基准测试结果一致，并且随序列长度增加而增大，证明了 FlashQLA 在真实模型激活值和门控分布上的有效性。

在完整模型层面，在 TP8 下使用 32K 词元上下文时，FlashQLA 可使 Qwen3.8- 2.4T-A95B 的首词元生成时间（TTFT）缩短约 5%。端到端提升较小，是因为 TTFT 中用于门控增量规则计算的时间占比相对较小。

7.3 多卡性能

我们在八个 GPU 上评估多 GPU 执行，使用总序列长度 64K 至 1M 词元，以及注意力头数 $H \in \{16, 32, 64\}$。每个序列均匀分区到全部八个 GPU 上。
为区分 FlashQLA 的执行优化、卡内并行和 Dual CP 所带来的优势，我们比较了四种实现：

1. **FLA KCP**：采用卡间 CP（KCP）的 FLA 实现。2. **FlashQLA KCP**：采用卡间 CP、但不包含卡内 CP 的 FlashQLA 内核实现。3. **朴素 KCP + Intra CP**：卡间 CP 与卡内 CP 的独立组合，在每个层级分别进行预处理。4. **FlashQL**

nd 分层
Dual CP：卡间与卡内联合 CP，并采用共享摘要计算 a
校正。

如图 7所示，FlashQLA KCP 在 5×–2× 处相较于 FLA KCP 实现了约 $H = 64$的加速比。然而，在较低头数下，有限的卡内并行会限制内核优化带来的收益，因此需要采用卡内 CP 来释放额外的序列级并行性。

通过朴素 KCP 引入卡内 CP +Intra CP 提供了额外的序列并行性，但也会重复进行预处理。在 $H = 16$ 下，并行性的提升改善了性能，尽管由于独立的卡间预处理仍需按顺序处理每个 GPU 本地序列，这一收益受到了限制。在 $H = 64$下，额外并行性带来的较小收益不及预处理开销，导致朴素组合的性能低于 FlashQLA KCP。

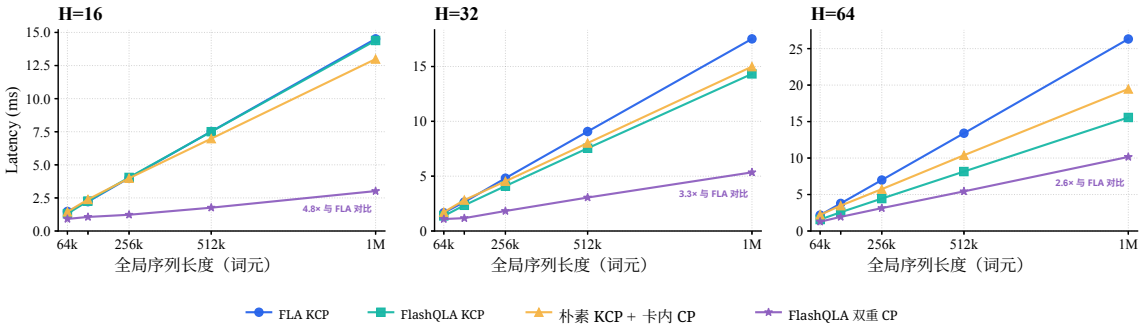


图7：**多 GPU 前向性能**。 在八张 GPU 上，对比 FLA KCP、FlashQLA KCP、朴素 KCP +卡内 CP 和 FlashQLA 双重 CP 在 64K 至 1M 词元的序列长度以及 16、32 和 64 的头数下的性能。双重 CP 避免了朴素组合中的重复预处理，并在低头数下实现了最大收益，此时额外的卡内并行最为有利。FlashQLA 双重 CP 相较于 FLA KCP 的加速比最高可达 4.8× 。

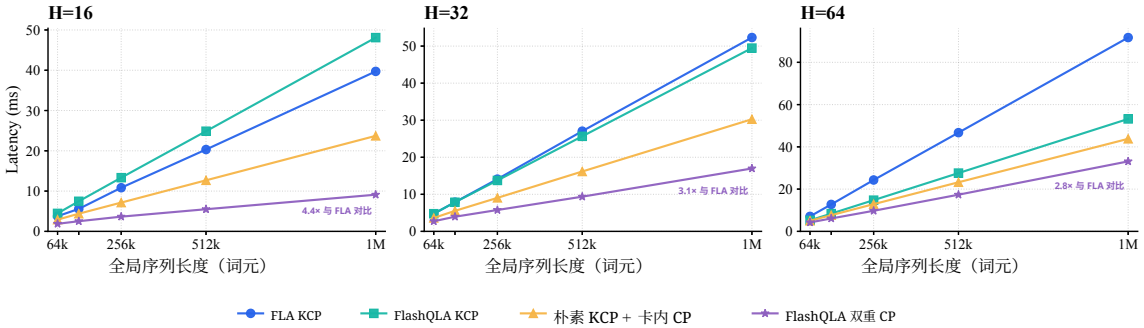


图 8：**多 GPU 前向和反向性能**。 在八张 GPU 上，四种上下文并行实现的前向与反向延迟。与仅前向执行相比，卡内 CP 可带来更大收益，而双重 CP 则通过共享摘要计算进一步降低预处理开销。FlashQLA 双重 CP 相较于 FLA KCP 的加速比最高可达 4.4× 。

Dual CP 通过在两个层级之间共享预处理来解决这一限制。从卡内摘要生成卡间摘要可以避免重复计算，并在预处理过程中释放序列并行性。与朴素 KCP +Intra CP 相比，Dual CP 在 $H = 16$下实现了最高 4.3× 的加速比，并在 $H = 64$下实现了 2× ，展现了在结合两个 CP 层级时协调预处理的重要性。

图 8 报告了前向与反向性能。反向传播更大的计算工作负载使卡内 CP 获得更大收益，因此即使是朴素 KCP + Intra CP 在仅前向执行中也能实现更显著的提升。Dual CP 通过共享摘要计算和分层校正进一步提升性能，在所评估的配置中，相较于朴素组合实现了约 1。2×–2。6×倍的加速比。

总体而言，FlashQLA 双重 CP 相较于 FLA KCP，在前向执行中最高可实现 4.8× 的前向加速，在前向与反向执行中最高可实现 4.4× 的加速比，证明了其在长上下文多 GPU 执行中的有效性。

随着 CP 大小从 1 增加到 8 个 GPU，我们进一步评估了 FLA KCP 和 FlashQLA 双重 CP 在前向与反向执行中的强扩展性能。每种实现均以其自身的单 GPU 延迟进行归一化。

如图 9所示，FlashQLA 双重 CP 在八个 45× GPU 上实现了 6.倍 $= 16$的加速比，相较于 FLA KCP 的 4.34× 倍。FlashQLA 双重 CP 展现出更强的扩展性，通过释放额外的卡内并行性并减少冗余预处理开销，更接近理想线性扩展。

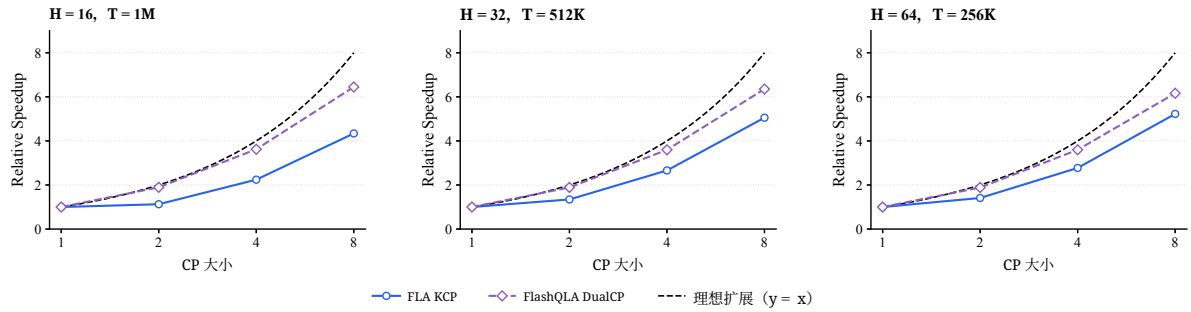


图 9：**多 GPU 前向与反向扩展。** FLA KCP 和 FlashQLA 双重 CP 在 CP 大小为 1、2、4 和 8 个 GPU 时的相对加速比，以各自实现的单 GPU 延迟为基准进行归一化。评估的配置为 $H = 16$，包含 1M 个词元；$H = 32$，包含 512K 个词元；以及$H = 64$，包含 256K 个词元。虚线（$y = x$）表示理想线性扩展。

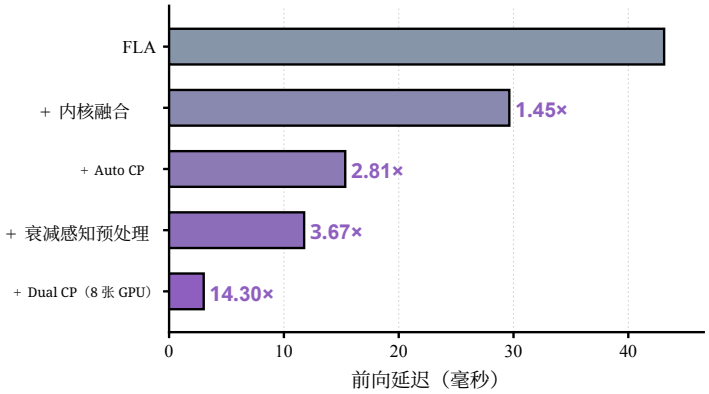


图 10：**FlashQLA 的优化分解。** 在包含 16 个注意力头、序列长度为 1M 个词元的 GPU 上，相对于单 GPU FLA 的累计前向加速比。各阶段逐步引入优化内核、结合 Auto CP 的卡内 CP、衰减感知准备，以及八 GPU 双重 CP。

7.4 优化分解

我们在一个具有 $H = 16$ 和 1M 词元序列长度的代表性工作负载上，考察了各项优化的独立贡献。从不使用 CP 的优化版 FlashQLA 内核开始，我们逐步启用结合 Auto CP 和衰减感知准备的卡内 CP，并扩展到使用 8 个 GPU 的 Dual CP。

如图 10所示，仅通过内核优化即可实现 1。45× 倍于 FLA 的加速比。启用结合 Auto CP 的卡内 CP 后，可进一步发挥序列并行性，使加速比提升至 2。81×倍，但实际收益仍受边界状态预处理开销限制。衰减感知准备显著降低了这一开销，使加速比进一步提升至 3。67×倍。最后，借助双重 CP 扩展至八个 GPU，可实现 14。30× 倍加速比，同时结合了多 GPU 并行、共享摘要计算和分层边界状态校正。

7.5 Auto CP 分析

Auto CP 依靠延迟预测，在一组有界的分区配置中进行选择。我们同时评估模型捕捉依赖分区的性能趋势的能力，以及最终选择方案的实测性能。

延迟模型准确性。 我们使用一个序列长度为 $[512, 2048, 32768]$ 和$H = 32$ 的变长工作负载，评估第 4.1 节中介绍的内核级延迟模型的预测准确性。我们考虑四个核心内核： h 准备内核、h 校正内核、融合前向内核和融合反向内核。在此次评估中，每个内核模型均使用四个实测配置进行校准，其余候选分区长度则用于评估预测准确性。

如图所示 如图 11所示，在所有四种 r

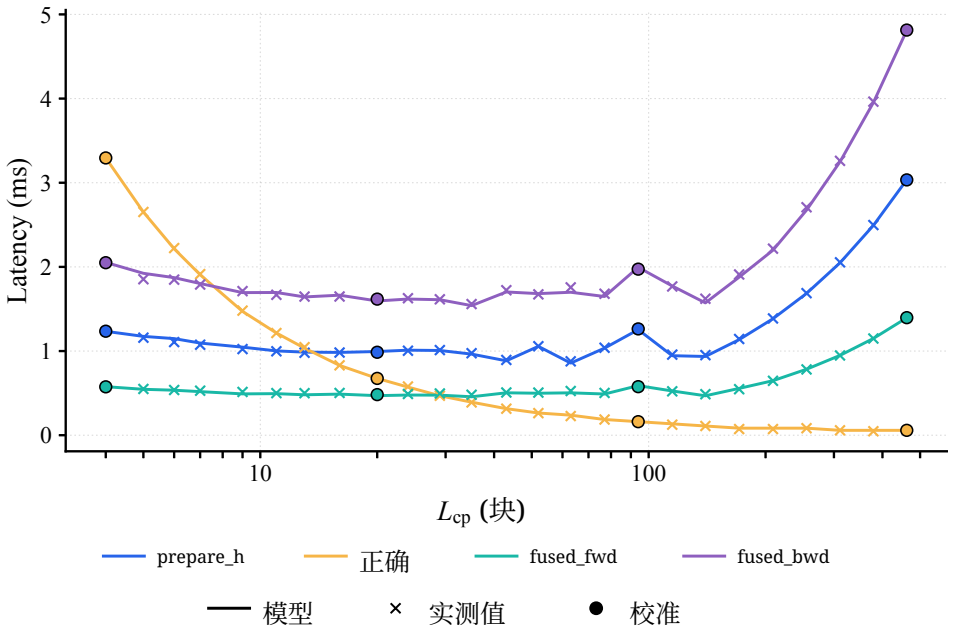


图 11：**内核延迟模型准确性**。对于一个序列长度为 $[512, 2048, 32768]$、头数为 32 的变长工作负载，展示四种内核在不同候选分区长度下的模型预测延迟曲线和实测延迟。

内核和广泛的分区长度范围内，实测延迟都与模型预测曲线高度一致。对于 h 准备内核、融合前向内核和融合反向内核，模型捕捉到了执行波次转换引起的延迟波动，以及更细分区带来的额外开销。对于 h 校正内核，模型捕捉到了随着分区长度减小、分区数量增加而不断增大的校正开销。这些结果表明，延迟模型能够捕捉不同分区配置的关键性能特征，为 Auto CP 选择提供可靠依据。

配置选择质量。接下来，我们评估这些预测能否带来高效的执行配置。我们评估了 90 种工作负载，涵盖不同的序列长度、批量大小、变长分布和头数，并分别考察前向以及前向与反向合并执行。

对于每种工作负载，我们穷举测量所有受支持的 CP 配置和无 CP 配置，以获得最优配置 x^*oracle。令 x^* 表示 Auto CP 选择的配置，$T(x)$ 表示配置 x下的实测执行延迟。我们通过相对延迟遗憾来量化选择质量：

$$\text{Regret} = \frac{T(x^*) - T(x^*_{\text{oracle}})}{T(x^*_{\text{oracle}})}. \tag{33}$$

该指标衡量所选配置的执行成本，而不要求其与最优配置完全一致。

如图 12所示，Auto CP 为 94.4% 的推理工作负载和 90.0% 的训练工作负载选择了最优配置，平均延迟遗憾分别仅为 0.15% 和 0.37%。这些结果表明，延迟建模结合有界波感知搜索，只需较小的候选集，平均即可实现接近最优的性能。

Auto CP 的性能收益。 我们进一步将 Auto CP 的性能收益与两种静态分区策略进行了比较：固定最大分区长度为 $l_{\max} = 32$，以及一种硬件感知启发式方法，将 $l_{\max} = \lceil N_{\text{sm}}/(BH) \rceil$ 设为，其中 N_{sm} 表示流式多处理器的数量，B 表示批量大小，H 表示注意力头的数量。我们在不同序列长度（从 1k 到 32k）和头数下，分别针对仅前向（推理）以及前向与反向结合（训练）两种设置，测量完整的线性注意力执行过程。对于每个头数，我们将各策略的延迟归一化为相对于 Auto CP 的值，并计算跨序列长度的几何平均值。

图13 示　显示 Auto CP 在所有评估策略中始终实现最低延迟　es
在所有　头数和执行设置。与固定分区相比，Auto CP 通过根据工作负载特征调整分区长度，提供　es

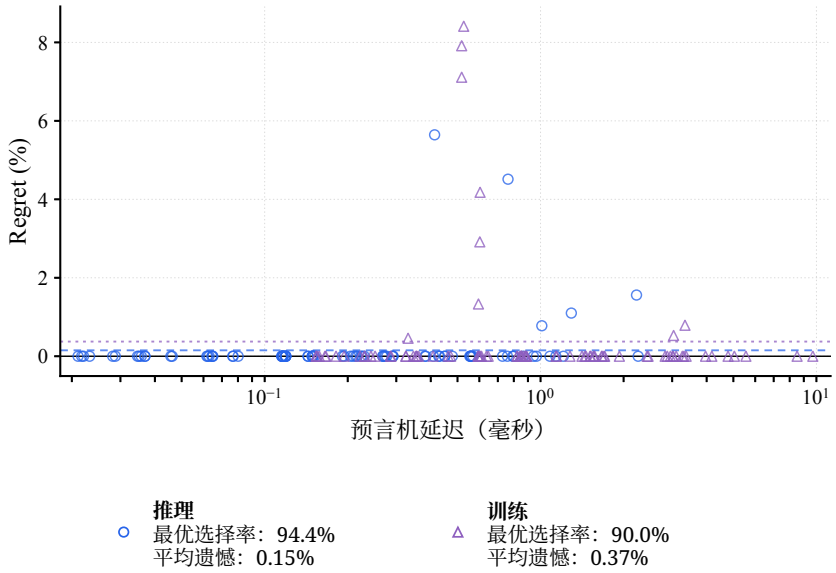


图12：**Auto CP 配置选择质量**。在推理（仅前向）和训练（前向与反向）执行下，90 个工作负载中，Auto CP 所选配置相对于穷举搜索基准的相对延迟遗憾，并以工作负载延迟为横轴绘制。

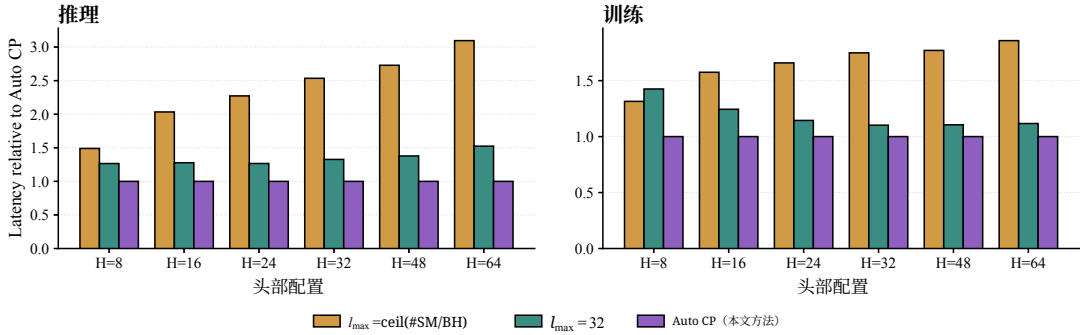


图13：**Auto CP 相对于静态分区的性能优势**。对于每种头数，展示不同序列长度下固定分区和硬件感知启发式方法的几何平均相对延迟，并以 Auto CP 进行归一化。左图和右图分别展示仅前向执行，以及前向与反向组合执行。Auto CP 在所有评估设置下均实现最低延迟，且在推理场景下性能提升更为显著。

了持续的改进。与硬件感知启发式方法相比，其优势更加明显，在推理场景下最高可达 3×，因为该启发式方法考虑了硬件并行性，但未考虑内核效率和边界状态预处理开销。在训练中，优势不太明显，因为反向传播的计算成本更高，导致预处理所占的执行时间比例降低。总体而言，这些结果证明了延迟感知分区选择在各种工作负载中的有效性。

8 相关工作

循环线性注意力。线性注意力以循环式固定大小状态取代二次方级别的逐词元交互，从而实现线性时间序列处理（Katharopoulos 等人，2020 年）。DeltaNet 在这一形式中加入增量规则更新，有选择地修改循环状态，并引入扩展 WY 表示，以便在块内并行计算（Yang 等人，2024年）。门控 DeltaNet（GDN）进一步引入数据依赖的状态衰减，并开发出高效的分块式 GPU 实现（Yang 等人，2025 年）。Kimi Delta Attention（KDA）在 GDN 基础上扩展，引入细粒度的逐通道衰减门（Kimi Team，2025 年）。Flash QLA 聚焦于 GDN，而其基于仿射摘要的上下文并行框架，则为将序列并行扩展到其他循环线性注意力架构奠定了基础。

高效线性注意力执行与上下文并行。 Flash Linear Attention（FLA）为多种线性注意力架构提供了硬件高效的 Triton 实现，并作为我们的单 GPU 基线（Yang and Zhang，2024年）。在分布式执行方面，Kimi K3 采用 KDA 上下文并行（KCP），将序列划分到不同设备上，计算局部状态和转移摘要，并通过边界状态校正确定各分区的初始状态（Kimi Team，2026 年）。FlashQLA 结合卡内与卡间序列并行、共享摘要计算和门控驱动的预处理，为这些工作提供了补充。

FlashQLA 开源发布后，FlashInfer（Ye 等人，2025 年）和 NVIDIA cuDNN 中的优化 GDN 实现也采用了类似的代数重构和上下文并行策略。这些进展进一步证明，将代数化简与序列级并行性相结合，对于高效执行 GDN 具有实际意义。

9 结论

我们提出了 FlashQLA，这是一种用于门控 DeltaNet 的上下文并行执行框架，可在 GPU 内部及 GPU 之间挖掘序列级并行性，同时降低边界状态预处理开销。FlashQLA 利用头级门控衰减进行衰减感知准备，并通过共享摘要计算的 Dual CP 整合卡内与卡间并行性。工作负载自适应 Auto CP 和上下文并行感知的内核协同设计进一步提升了执行效率。

在单个 GPU 上，FlashQLA 相较 FLA 的前向加速比最高可达 3.1×，反向加速比最高可达 4.4×。扩展至八个 GPU、上下文长度最高达 1M 词元时，Dual CP 相较 FLA KCP 的前向加速比最高可达 4.8×，前向和反向执行合计的加速比最高可达 4.4×。这些结果表明，要实现有效的上下文并行，不仅需要挖掘额外的序列并行性，还需要降低为此所需的预处理开销。

致谢

我们谨向 Hongjie Xie 及其他社区开发者致以诚挚谢意，感谢他们为面向先进计算架构的 FlashQLA 内核开发与维护所作的贡献。他们的努力帮助 FlashQLA 扩展了对不同 GPU 架构的兼容性与支持。我们也感谢 Guangyun Han，以及 Flash-Linear Attention 和 TileLang 社区；与他们的技术交流和讨论为我们带来了许多启发。最后，我们感谢所有为 FlashQLA 开源项目作出贡献的人，感谢他们在代码维护、工程改进和项目持续发展方面所作的宝贵贡献。

参考文献

Angelos Katharopoulos、Apoorv Vyas、Nikolaos Pappas 和 Francois Fleuret。《Transformers are RNNs：借助线性注意力实现快速自回归 Transformer》。载于 第 *37* 届国际机器学习会议论文集，第 5156–5165 页，2020 年。网址：https://proceedings.mlr.press/v119/katharopoulos20a.html。Kimi Team。《Kimi linear：一种富有表现力且高效的注意力架构》。 *arXiv* 预印本 *arXiv:2510.26692*，2025 年。网址：https://arxiv.org/abs/2510.26692。Kimi Team。《Kimi K3：开放前沿智能》。 *arXiv* 预印本 *arXiv:2607.24653*，2026 年。网址：https://arxiv.org/abs/2607.24653。Qwen Team。Qwen3-Next-80B-A3B，2025 年 9 月。网址：https://qwen.ai/blog?id=qwen3-next。Qwen Team。《Flashqla：面向 CP/反向传播的 GDN 融合线性注意力内核》。 https://qwen.ai/blog?id=flashqla，2026 年 4 月 a。技术博客。Qwen Team。《Qwen3.8-Next 架构设计：评估、效率与训练稳定性》。技术报告，阿里巴巴集团，2026 年 8 月 b。Qwen Team。《Qwen3.8-Max：编程与协作的新标杆》，2026 年 8 月 c。网址：https://qwen.ai/blog?id=qwen3.8。Lei Wang、Yu Cheng、Yining Shi、Zhiwen Mo、Zhengju Tang、Wenhao Xie、Tong Wu、Lingx- iao Ma、Yuqing Xia、Jilong Xue、Fan Yang 和 Zhi Yang。《TileLang：连接现代神经网络内核的可编程性与性能》。载于 国际学习表征会议，2026 年。网址：https://proceedings.iclr.cc/paper_files/paper/2026/hash/76fb92288bf90360c527efb0d1c2aba6-Abstract-Conference.html。

Songlin Yang 和 Yu Zhang。《Fla：一个基于 Triton、用于实现硬件高效线性注意力机制的库》，2024 年 1 月。网址：https://github.com/fla-org/flash-linear-attention。

Songlin Yang、Bailin Wang、Yu Zhang、Yikang Shen 和 Yoon Kim。基于序列长度的 delta 规则并行化线性变换器。载于神经信息处理系统进展，第 37 卷，2024年。网址：https://proceedings.neurips.cc/paper_files/paper/2024/hash/d13a3eae72366e61dfdc7eea82eeb685-Abstract-Conference.html。

Songlin Yang、Jan Kautz 和 Ali Hatamizadeh。门控 delta 网络：通过 delta 规则改进 Mamba2。载于国际学习表征会议，2025 年。网址：https://proceedings.iclr.cc/paper_files/paper/2025/hash/4904fad153f6434a7bcf04465d4be2cc-Abstract-Conference.html。

Zihao Ye、Lequn Chen、Ruihang Lai、Wuwei Lin、Yining Zhang、Stephanie Wang、Tianqi Chen、Baris Kasikci、Vinod Grover、Arvind Krishnamurthy 和 Luis Ceze。FlashInfer：用于大语言模型推理服务的高效且可定制注意力引擎。载于机器学习与系统论文集，第 7 卷，2025 年。网址：https://proceedings.mlsys.org/paper_files/paper/2025/hash/dbf02b21d77409a2db30e56866a8ab3a-Abstract-Conference.html。

符号说明

本附录总结了全文使用的符号。普通下标表示令牌索引，方括号下标表示块索引或边界。上标表示分区：(i) 表示单级分区或卡间分区，(i,j) 表示 GPU i上的卡内分区。

表 2：门控 DeltaNet 上下文并行与边界状态预处理的符号说明。

符号	说明	符号	描述
一般记法			
C	每个块的标记数	L	每个序列的块数
B	批量大小	L_b	序列中的块 b
H	头数	$d_k,\ d_v$	键/值维度
t	标记索引	c	全局块索引
h	注意力头索引	$\mathbf{S}_0$	初始循环状态
门控 *DeltaNet*			
$\mathbf{S}_t$	词元 t 后的状态	$\mathbf{o}_t$	词元 t 的输出
$\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t$	词元级 Q/K/V	$\alpha_t,\ \beta_t$	衰减门 / 更新强度
$\mathbf{Q}_{[c]}, \mathbf{K}_{[c]}, \mathbf{V}_{[c]}$	块级 Q/K/V	$\mathbf{O}_{[c]}$	块输出
$\alpha_{[c],t}, \beta_{[c],t}$	局部词元门	$\gamma_{[c],t}$	累积词元衰减
$\mathbf{B}_{[c]}$	对角更新强度	$\mathbf{\Gamma}_{[c]}$	对角累积衰减
$\mathbf{A}_{[c]}$	扩展 WY 逆	$\mathbf{U}_{[c]}, \mathbf{W}_{[c]}$	扩展 WY 辅助量
$\mathbf{S}_{[c]}$	进入块的状态	$\mathbf{M}_{[c]}$	块转移矩阵
$\mathbf{E}_{[c]}$	块内贡献	$\gamma_{[块],C}$	块总衰减
边界状态预处理			
N	分区数量	l_i	分区中的块 i
$\mathbf{S}^{(i)}_{[j]}$	局部边界处的状态 j	$\mathbf{S}^{(i)}_{[0]}$	分区输入状态
$\mathbf{M}^{(i)}_{[j]}$	局部块转移	$\mathbf{E}^{(i)}_{[j]}$	局部块贡献
$\widetilde{\mathbf{S}}^{(i)}_{[j]}$	零状态展开	$\Delta\mathbf{S}^{(i)}_{[j]}$	由初始状态导致的状态差异
$\mathbf{M}^{(i)}_{[j\leftarrow 1]}$	累积转移	$\mathbf{S}^{(i)}_{[l_i]}$	分区终止状态

B 门控因子化 GDN 重构的推导

我们推导第 6.1 节提出的门控因子化 GDN 重构，并采用第 2.1 节定义的分块式公式。推导展示了如何将依赖门控的缩放从三角逆矩阵中提出，从而简化分块状态转移。

表 3：衰减感知准备和 Auto CP 的符号说明。

符号	描述	符号	描述
衰减感知准备			
$\gamma_{[c],C,\ h}$	注意力头 h的块衰减	$D_h(a,b)$	[aa、b]的累积衰减
TOL	衰减阈值	$K_{c,h}$	按块计算的衰减窗口
$W_{c,h}$	按词元计算的衰减窗口	$K_{\max}$	最大截断窗口
K_i	边界衰减窗口	$\overline{\mathbf{M}}_{[c]}$	衰减归一化转换
$\Delta\mathbf{S}_{[c]}$	由截断历史导致的状态差异	$\widetilde{\mathbf{S}}^{(i-1)}_{[l_{i-1}],\ K_i}$	截断零状态展开
Auto CP：延迟代理模型			
x	执行配置	κ	内核标识符
r	CTA 索引	$R_\kappa(x)$	CTA 数量
$k_{\kappa,\ r}(x)$	CTA 计算步数	N_{sm}	流式多处理器数量
$\mathcal{A}_s(x)$	分配给 SM s 的 CTA	$d_\kappa(x)$	关键调度深度
a_κ, b_κ, c_κ	拟合延迟系数	$\widehat{t_\kappa}(x)$	预测内核延迟
$\mathcal{K}(x)$	内核调用序列	$\widehat{T}(x)$	预测总延迟
Auto CP：分区搜索和启用			
$l_{\max}$	最大分区长度	$\mathcal{L}$	支持的分区长度
w	执行波次数	$l^{(w)}_{\max}$	波边界长度估计
$\mathcal{L}_{\text{wave}}$	波感知候选项	$l^{*\text{最大}}_{\text{值}}$	所选分区长度
$x(l_{\max})$	CP 配置	$x_\varnothing$	无 CP 配置
x^*	所选配置		

表 4：Dual CP 的符号说明。

符号	描述	符号	描述
分区层级			
N_{inter}	GPU 数量	N^i	GPU i上的本地分区
l_i	分配给 GPU i的块	$l^{\text{内部}}_{i,j}$	本地分区 $(i,\ j)$中的块
$\mathbf{S}^{(i)}_{[c]}$	GPU本地状态	$\mathbf{S}^{(i,\ j)}_{[c]}$	卡内分区状态
共享摘要计算			
$\widetilde{\mathbf{S}}^{(i,j)}_{[l_{i,j}]}$	本地零状态终端状态	$\mathbf{M}^{(i,j)}_{[l_{i,\ j}\leftarrow 1]}$	局部累积转移
$L_{i,j}$	累积局部块计数	$\widetilde{\mathbf{S}}^{(i)}_{[L_{i,\ j}]}$	聚合零状态摘要
$\mathbf{M}^{(i)}_{[L_{i,j}\leftarrow 1]}$	聚合转移	$\widetilde{\mathbf{S}}^{(i)}_{[l_i]}$	GPU级零状态摘要
$\mathbf{M}^{(i)}_{[l_i\leftarrow 1]}$	GPU级转移		
分层边界状态校正			
$\mathbf{S}^{(i)}_{[0]}$	GPU输入边界状态	$\mathbf{S}^{(i,\ j)}_{[0]}$	局部分区输入状态

对于块 c，回顾公式（3），可知

$$\mathbf{A}^{-1}_{[c]} = \mathbf{I} + \text{StrictLower}\left(\mathbf{B}_{[c]}\mathbf{\Gamma}_{[c]}\mathbf{K}_{[c]}\mathbf{K}^{\top}_{[c]}\mathbf{\Gamma}^{-1}_{[c]}\right). \tag{34}$$

由于 $\mathbf{B}_{[c]}$ 和 $\mathbf{\Gamma}_{[c]}$ 均为对角矩阵，且对角缩放保持严格下三角结构，因此有

$$\begin{aligned}\mathbf{A}^{-1}_{[c]} &= \mathbf{\Gamma}_{[c]}\left(\mathbf{I} + \text{StrictLower}\left(\mathbf{B}_{[c]}\mathbf{K}_{[c]}\mathbf{K}^{\top}_{[c]}\right)\right)\mathbf{\Gamma}^{-1}_{[c]} \\ &= \mathbf{\Gamma}_{[c]}\mathbf{A}'^{-1}_{[c]}\mathbf{\Gamma}^{-1}_{[c]}.\end{aligned} \tag{35}$$

这里，$\mathbf{A}'_{[c]}$ 是公式（29）中定义的与门无关的三角逆矩阵。对两边求逆可得

$$\mathbf{A}_{[c]} = \mathbf{\Gamma}_{[c]}\mathbf{A}'_{[c]}\mathbf{\Gamma}^{-1}_{[c]}, \tag{36}$$

即公式（30）。

将这一因式分解代入公式（4）和（5）中的辅助量，得到

$$\mathbf{\Gamma}_{[c]}^{-1}\mathbf{U}_{[c]} = \mathbf{A}'_{[c]}\mathbf{B}_{[c]}\mathbf{\Gamma}_{[c]}^{-1}\mathbf{V}_{[c]}, \tag{37}$$

$$\mathbf{\Gamma}_{[c]}^{-1}\mathbf{W}_{[c]} = \mathbf{A}'_{[c]}\mathbf{B}_{[c]}\mathbf{K}_{[c]}. \tag{38}$$

第二个等式利用了$\mathbf{B}_{[c]}$ 与 $\mathbf{\Gamma}_{[c]}$的交换性。这些表达式消除了三角逆矩阵周围依赖门控的缩放。

将公式（37）和（38）代入公式（8）中的块级转移，可得

$$\mathbf{M}_{[c]} = \gamma_{[c],C}\left(\mathbf{I} - \mathbf{K}_{[c]}^{\top}\mathbf{A}'_{[c]}\mathbf{B}_{[c]}\mathbf{K}_{[c]}\right), \tag{39}$$

$$\mathbf{E}_{[c]} = \gamma_{[c],C}\mathbf{K}_{[c]}^{\top}\mathbf{A}'_{[c]}\mathbf{B}_{[c]}\mathbf{\Gamma}_{[c]}^{-1}\mathbf{V}_{[c]}. \tag{40}$$

因此，分块状态更新仍保持其仿射形式

$$\mathbf{S}_{[c+1]} = \mathbf{M}_{[c]}\mathbf{S}_{[c]} + \mathbf{E}_{[c]}, \tag{41}$$

但其转移矩阵现在可以使用与门无关的三角逆矩阵$\mathbf{A}'_{[c]}$进行计算。

最后，$\mathbf{E}_{[c]}$ 中的累积门控因子可以表示为相对衰减因子。对于每个局部词元位置t，

$$\begin{aligned}\frac{\gamma_{[c],C}}{\gamma_{[c],t}} &= \exp\left(\log\gamma_{[c],C} - \log\gamma_{[c],t}\right)\\ &= \exp\left(\sum_{j=t+1}^{C}\log\alpha_{[c],j}\right),\end{aligned} \tag{42}$$

其中，空和等于零。通过在指数运算之前计算累积对数门控值之差，我们避免了显式构造可能发生下溢的累积乘积，也避免了执行数值不稳定的除法。

因此，这种重新表述在保留原始分块式状态转移的同时，移除了三角逆矩阵中的依赖门控的缩放。

C 转移累积的数值稳定性

在边界状态预处理中，FlashQLA 会计算每个分区上的累积转移矩阵。由于这涉及跨块进行重复的矩阵乘法，因此数值误差可能会不断累积，尤其是在使用低精度矩阵乘法时。我们通过门控分离和 FP32累积转移更新来提高数值稳定性。

C.1 门控分离

回顾一下，块转移可以分解为

$$\mathbf{M}_{[j]}^{(i)} = \gamma_{[j],C}^{(i)}\overline{\mathbf{M}}_{[j]}^{(i)}. \tag{43}$$

因此，分区 i 的前 j 个块上的累积转移为

$$\mathbf{M}_{[j\leftarrow 1]}^{(i)} = \exp\left(\sum_{r=1}^{j}\log\gamma_{[r],C}^{(i)}\right)\overline{\mathbf{M}}_{[j\leftarrow 1]}^{(i)}, \tag{44}$$

其中

$$\overline{\mathbf{M}}_{[j\leftarrow 1]}^{(i)} = \overline{\mathbf{M}}_{[j]}^{(i)}\cdots\overline{\mathbf{M}}_{[1]}^{(i)}. \tag{45}$$

在直接低精度乘法中，门控缩放会被反复纳入累积转移，导致每次迭代都引入额外的舍入。通过将门控因子与转移矩阵分离，FlashQLA 可以累积与门控无关的转移，而无需反复应用门控缩放。累积门控因子则在对数域中单独跟踪，仅在需要完整转移时应用。

C.2 FP32累积转移更新

累积转移通过跨块的连续矩阵乘法计算得到。在低精度下直接累积转移矩阵会引入反复的舍入误差，这些误差可能在较长的分区上不断累积，从而降低数值精度。

为避免这一问题，我们通过展开块转移矩阵进一步简化累积转移更新。使用公式（31）中推导出的与门无关的转移，可得

$$\begin{aligned}\overline{\mathbf{M}}^{(i)}_{[j\leftarrow 1]} &= \overline{\mathbf{M}}^{(i)}_{[j]}\overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]} \\ &= \left(\mathbf{I} - \mathbf{K}^{(i)\top}_{[j]}\mathbf{A}'^{(i)}_{[j]}\mathbf{B}^{(i)}_{[j]}\mathbf{K}^{(i)}_{[j]}\right)\overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]} \\ &= \overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]} - \mathbf{K}^{(i)\top}_{[j]}\mathbf{A}'^{(i)}_{[j]}\mathbf{B}^{(i)}_{[j]}\mathbf{K}^{(i)}_{[j]}\overline{\mathbf{M}}^{(i)}_{[j-1\leftarrow 1]}.\end{aligned} \tag{46}$$

这种重新表述将恒等贡献与矩阵乘法分离开来。FlashQLA 在 FP32累加器中维护累积转移，并使用低精度矩阵乘法计算更新项。尽管更新项仍会受到低精度算术的影响，但累积转移在各次迭代中始终以 FP32 保持，从而避免了对整个矩阵反复进行低精度舍入。

D 边界状态预处理的推导

我们将推导第 2.2 节中使用的状态差传播性质和边界状态校正。

状态差传播。 考虑两条状态轨迹 $\mathbf{S}_{[c]}$ 和 $\widetilde{\mathbf{S}}_{[c]}$，它们处理相同的块输入，但从不同的初始状态开始。根据公式（8）中的仿射块递推，两条轨迹均满足

$$\begin{aligned}\mathbf{S}_{[c+1]} &= \mathbf{M}_{[c]}\mathbf{S}_{[c]} + \mathbf{E}_{[c]}, \\ \widetilde{\mathbf{S}}_{[c+1]} &= \mathbf{M}_{[c]}\widetilde{\mathbf{S}}_{[c]} + \mathbf{E}_{[c]}.\end{aligned} \tag{47}$$

定义 $\Delta\mathbf{S}_{[c]} = \mathbf{S}_{[c]} - \widetilde{\mathbf{S}}_{[c]}$，后，将两个递推式相减，即可消去相同的加性项：

$$\Delta\mathbf{S}_{[c+1]} = \mathbf{M}_{[c]}\Delta\mathbf{S}_{[c]}. \tag{48}$$

在块 a 到b 上反复应用此关系，得到

$$\Delta\mathbf{S}_{[b+1]} = \mathbf{M}_{[b]}\mathbf{M}_{[b-1]}\cdots\mathbf{M}_{[a]}\Delta\mathbf{S}_{[a]}, \tag{49}$$

由此可得式（9）。

边界状态校正。 对于分区i，令$\mathbf{S}_{(i)\,[0]}$ 表示其输入边界状态。状态摘要准备以 $\widetilde{\mathbf{S}}_{(i)\,[0]} = \mathbf{0}$为初始状态进行零状态展开。真实状态轨迹和零状态轨迹遵循相同的块转移，因此它们的初始差值为

$$\Delta\mathbf{S}^{(i)}_{[0]} = \mathbf{S}^{(i)}_{[0]}. \tag{50}$$

对所有l_i 个块应用状态差异传播性质，得到

$$\mathbf{S}^{(i)}_{[l_i]} - \widetilde{\mathbf{S}}^{(i)}_{[l_i]} = \mathbf{M}^{(i)}_{[l_i\leftarrow 1]}\mathbf{S}^{(i)}_{[0]}. \tag{51}$$

重新排列后得到

$$\mathbf{S}^{(i)}_{[l_i]} = \widetilde{\mathbf{S}}^{(i)}_{[l_i]} + \mathbf{M}^{(i)}_{[l_i\leftarrow 1]}\mathbf{S}^{(i)}_{[0]}, \tag{52}$$

恢复公式（11）。

E 支持锯齿上下文并行

现代大语言模型训练框架通常采用之字形上下文并行，以平衡因果全注意力层中各 GPU 的计算负载。这种布局会将不连续的序列片段分布到各个 GPU 上，因此要求 GDN 层支持相同的数据布局，以便与混合注意力架构兼容。

FlashQLA 通过其分层上下文并行设计支持锯齿布局。在卡间层面，每个 GPU 为其本地序列片段计算状态摘要和状态转移。随后，这些摘要会按照原始序列顺序排列，而卡间边界状态校正则重建跨片段的依赖关系，无论这些片段在物理上分布于哪个 GPU。校正后的边界状态随后用于初始化相应的本地计算，从而保留原始的 GDN 递归，而无需进行词元级重新分配。

重要的是，卡内 CP 和 Auto CP 仍独立于全局锯齿布局。一旦确定本地片段的边界状态，卡内 CP 就会将每个片段作为普通的连续序列进行处理，而 Auto CP 则根据本地工作负载选择其分区配置。两种机制都无需考虑片段如何分布在各个 GPU 上。这种分离将锯齿布局特有的处理限制在卡间边界状态校正中，并使 FlashQLA 能够复用现有的卡内执行和优化机制。

由于锯齿适配只改变片段级摘要的组织和校正方式，因此不会修改核心计算内核，同时仅引入极少的额外开销。在我们的评估中，支持锯齿布局对整体训练性能没有明显影响。