arXiv:2512.23858v1 [cs.LG] 29 Dec 2025

Yggdrasil：连接动态推测与静态运行时，实现延迟最优的基于树的 LLM 解码

Yue Guan[1, 2*]、**Changming Yu**[1, 2, *]、**Shihan Fang**[1]**,Weiming Hu**[1,2]**,Zaifeng Pan**[3]**,**
Zheng Wang[3]**, Zihan Liu**[1,2]**, Yangjie Zhou**[1,2]**,Yufei Ding**[3]**,Minyi Guo**[1,2]**,Jingwen Leng**[1,2]
[1]上海交通大学[2]上海期智研究院[3]加利福尼亚大学圣迭戈分校 {bonboru,fichmi,fang-account,weiminghu,
altair.liu,yj_zhou,guo-my,leng-jw}@sjtu.edu.cn,
{zapan, zhw100, yufeiding}@ucsd.edu

摘要

推测解码通过并行生成和验证多个词元来提升大语言模型推理性能，但由于动态推测与静态运行时假设之间存在不匹配，现有系统的性能并不理想。我们提出 Yggdrasil，这是一种协同设计的系统，通过上下文感知树草拟和编译器友好执行，实现延迟最优的推测解码。Yggdrasil 引入了适用于静态图兼容性的等增长树结构、面向延迟的草稿选择优化目标，以及用于降低开销的基于阶段的调度。Yggdrasil 支持未经修改的大语言模型，在多种硬件配置下，相较于当前最先进的基线，最高可实现 $3.98\times$ 加速比。

1 引言

生成式大语言模型（LLMs）[3] 在语言理解[10], 、推理[42], 和多模态任务 [11]等众多领域展现出卓越的性能。然而，大语言模型规模的不断扩大带来了巨大的计算与内存需求[52], , 导致延迟和部署成本高昂。为解决这些瓶颈，研究人员提出了众多优化策略[53, 28, 19]。其中，推测解码[22] 通过利用自回归生成过程中的并行性，实现无损加速，因而脱颖而出。

传统解码每次生成一个词元，每个新词元都依赖于前一个词元。这种严格的顺序处理方式使现代硬件的利用率严重不足。推测解码通过生成多个候选令牌，并使用原始模型并行验证它们，打破了这一瓶颈。如果草稿与预言机匹配，就可以在单个生成步骤中接受多个词元，从而提升整体吞吐量。这一概念与分支预测[38]和缓存预取[5]等经典架构优化技术相呼应。

然而，尽管前景广阔，现有的推测解码系统由于**动态草拟算法与现代基于编译器的运行时的静态假设之间存在根本性不匹配**，仍无法达到最优性能。如图 1 所示，推测解码会在每个解码步骤动态调整树结构和算子形状，以最大化词元接受率。这种行为与深度学习编译器所要求的静态数据流图存在内在冲突，而后者依赖固定的计算模式来进行图融合、内核调优和内存规划。

此外，当前方法通常在简单化的假设下优化平均接受长度（AAL），忽略了不均匀的验证成本。与此同时，运行时系统通常将推测

*两位作者对本工作的贡献相同。

神经信息处理系统第 39 届会议（NeurIPS 2025年）。

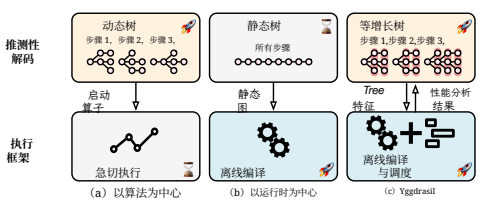

图 1：Yggdrasil 概览。

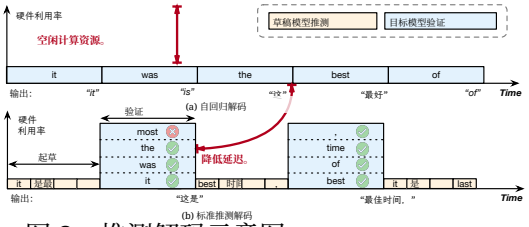

图 2：推测解码示意图。

解码视为黑盒，没有处理复杂的中央处理器逻辑开销，因此错失了利用推测解码专属优化的机会。这些相互交织的挑战限制了算法效率和系统级性能。

为克服这些局限，我们提出 Yggdrasil，这是一种延迟最优推测解码系统，协同设计算法与运行时执行。Yggdrasil 基于三个核心理念：（1）延迟感知目标：一种基于硬件性能分析的优化目标，通过选择最优树形结构（深度、宽度、验证范围），最大限度降低实际延迟，而非优化 AAL 等代理指标（§. 4.1）。（2）等增长树（EGT）：一种新颖的草稿树结构，在保持根据解码上下文灵活调整能力的同时，确保算子形状静态，以兼容编译时图优化（§. 4.2）。（3）基于阶段的调度：一种专为推测解码设计的搜索框架，通过消除条件分支并重叠执行依赖计算，最大限度降低 CPU-GPU 协调成本（§. 5）。

Yggdrasil 系统基于开源深度学习编译器 TorchInductor[2] 构建，无需修改模型架构，支持广泛的大语言模型。在不同硬件和模型配置下，与 SpecInfer、Sequoia 和 vLLM-Spec 等最先进系统相比，其加速比最高可达 $3.98\times$。本研究的主要贡献如下：

- 我们设计了一种等速增长树草拟算法，通过优化延迟感知目标，在保持图编译兼容性的同时实现上下文感知的推测解码（§. 4）。
- 我们提出了一种阶段调度运行时执行框架，通过重新编排和调度推测解码阶段来降低开销（§. 5）。
- 我们在真实世界的模型和数据集上实现并评估了 Yggdrasil，展示了相较于现有基线的一致提升（§. 6 & §. 7）。

2 背景

草拟与验证。 如图 2 所示，在推测解码过程中，草拟阶段会生成供后续步骤使用的候选词元序列。随后，这些序列会进入验证 阶段，由预言机模型评估其正确性。相应的模型分别称为草拟模型 和验证器／目标模型。通过实现词元并行验证，推测解码减少了解码迭代的次数。

平均接受长度。 推测解码的一项关键性能指标是平均接受长度 （AAL），它衡量每次解码迭代平均接受的词元数量。AAL 越高，表示效率越高，因为更多词元会被追加到输出中。

改进的来源。 推测解码之所以有效，其核心在于利用了未充分使用的计算资源。许多现代计算设备（例如 H100 GPU 卡 [33], ）在解码推理期间受内存限制，导致其计算能力未得到充分利用。这种利用不足为推测解码提供了提升资源效率的机会。此外，Transformer [43] 解码器架构是大多数大语言模型的基础，其本身支持并行性，因此非常适合推测解码。具体而言，Transformer 中的自注意力机制使用序列级掩码对输入序列 [43]中的因果依赖进行编码。如图 2 所示，这种设计能够通过一次并行推理执行来验证推测序列。推测草拟与高效的序列级并行验证相结合，提升了解码过程的效率。

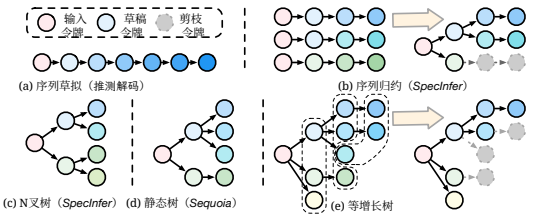

图3：草拟结构对比。

研究	草拟方法		编译	
	适应性	结构	草稿	验证
推测解码[22]	静态	序列	○	○
DISCO[29]	动态	序列	○	○
SpecInfer[31]	静态	Tree	○	○
vLLM-Spec[27]	静态	序列	●	●
Sequoia[8]	静态	Tree	●	○
Yggdrasil	动态	Tree	●	●

表1：已有工作的比较。

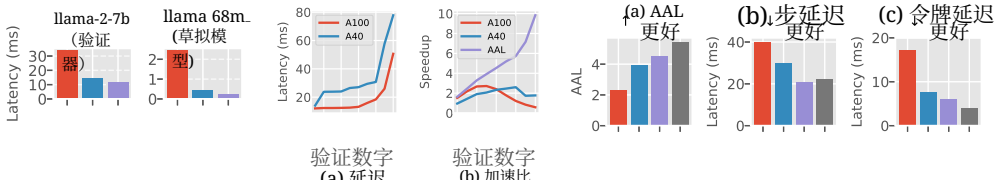

图4：不同运行时的逐步延迟基准测试。图5：延迟与加速比特征。图6：AAL、延迟和每令牌延迟的比较。

基于树的草拟。 尽管推测解码利用了验证过程中的冗余计算资源，但如图 2-(b) 所示，其生成式草拟阶段仍未得到充分利用。推测解码的一种先进方法是采用基于树的草拟，即以树结构的层级形式组织预测。这种结构支持并行验证，并扩展了候选令牌的探索空间。虽然基于树的草拟显著拓宽了推测探索空间，并提高了 AAL 指标，但其效果很大程度上取决于树结构及其构建策略，而这些因素会影响系统性能和计算开销。

SpecInfer [31]提出使用独立的草拟模型生成多个草拟序列，并将其归约为树结构。虽然这种方法扩展了探索空间，但也引入了重复推测，导致草拟开销增加。SpecInfer 的另一种方法是在每个草拟步骤中从预测分布中采样前 K 个词元，以构建 K叉树。该方法覆盖了广泛的推测可能性，并保证较高的 AAL。然而，由于指数复杂度，随着树的深度增加，这种方法会变得不切实际，从而限制了其草拟长度。

DISCO[41] 引入了一种完全动态的树结构，其树的深度和宽度根据序列上下文动态确定。这种动态方法在固定验证预算下实现了较高的 AAL，但由于依赖动态控制流和可变的操作形状而产生了显著的运行时开销，详见第 3 节。

Sequoia [8] 采用了不同的方法，提出一种数据集自适应树结构，在 AAL 与运行时开销之间取得平衡。Sequoia 通过针对给定的验证预算和数据集分布，对最优树结构进行性能分析，从而优化性能。然而，它对所有输入令牌都采用相同的树结构，没有考虑不同上下文中草拟探索空间的差异。

3 动机

尽管推测解码在加速大语言模型推理方面具有巨大潜力，但由于算法设计与运行时支持之间存在根本性差距，现有系统无法实现最佳性能。具体而言，我们将在下文重点介绍两个瓶颈。

动态草稿打破静态编译。 如第 2 节所述，上下文感知树草拟能够提升 AAL，同时不增加验证令牌的数量。然而，这种动态性与当今编译器优化中根深蒂固的静态图假设正面冲突，如图 4 所示。例如，使用 CUDA Graphs 捕获 Llama-2-7B 可带来 $2.32\times$ 加速比，但动态草拟引入的控制流分支无法固化为单一图，从而阻碍了这一收益。算子级自动调优[17]也面临同样的问题。针对某一固定形状编译的内核可额外带来 $1.23\times$ 加速比，但前提是形状保持不变。简言之，提升 AAL 的灵活性恰恰使我们无法利用那些能加速大语言模型推理的编译时优化。

高 AAL 并不等于高端到端加速比。 大多数先前研究都在一个不言自明的假设下最大化 AAL：接受更多词元会按比例转化为实际运行时间节省。只有在验证开销较低时，这种等价关系才成立，如图 5-(a) 所示。随着验证令牌数量增加，每一步的验证延迟也会随之上升。此时，我们仍可能观察到 AAL 加速比，但实际每令牌加速比曲线会趋于平缓，最终甚至反转（图 5-(b)）。孤立地优化 AAL 会带来递减收益，甚至产生负收益。任何切合实际的目标都必须权衡当前解码上下文中的接受率提升与验证开销。

总结。 如图 6 所示，这些瓶颈导致推测式系统设计并非最优。Sequoia 等基于树的方法能够提高 AAL，但仍保持较高的每步延迟。vLLM [27]和 TRT-LLM [34]等面向编译的系统通过静态内核降低步延迟，却无法利用动态推测，从而限制了 AAL。这些因素共同导致并非最优的每令牌延迟；现有框架尚无法同时实现高 AAL 和低运行时延迟，凸显出迫切需要能够协调动态草拟与编译效率的技术。

4 延迟最优的 TreeSpeculation

本节将介绍一种新颖的树形草拟方法，以应对动态推测解码所面临的挑战。我们首先提出一个面向推测解码的延迟感知优化目标，使系统实际运行时间加速比最大化。随后，我们介绍一种有利于编译时优化的上下文感知动态树形草拟算法。

4.1 延迟感知优化目标

如第 3 节所述，现有系统通常将 AAL 最大化，作为系统加速比的近似指标。

$$Speedup_{\text{naive}} = \frac{T_{\text{generative}}}{T_{\text{speculative}}} \simeq \frac{Num_{\text{iteration}} * T_{\text{target}}}{T_{\text{target}}} = AAL \tag{1}$$

其中， $Num_{iteration}$ 是朴素生成范式为达到与推测解码相同的输出长度而执行的前向推理步数，该长度等于 AAL。然而，这种近似忽略了：(1) 草稿迭代的开销，以及 (2) 模型的延迟特性。

因此，我们提出采用一种更准确的加速比指标，将实际执行延迟纳入考量。我们先从标准推测解码设置开始说明。

$$Speedup = \frac{T_{\text{generative}}}{T_{\text{speculative}}} = \frac{AAL * T_{\text{verifier}}(1)}{Num_{\text{draft}} * T_{\text{drafter}} + T_{\text{verifier}}(Num_{\text{draft}} + 1)} \tag{2}$$

$T_{\text{verifier}}(W)$ 表示验证模型并行验证 W 个词元所需的验证时间，如图 5-(a) 所示。Num_{draft} 则是草拟模型生成推测内容所需的推理迭代次数，该次数大于或等于 AAL-1。这里的 -1 来自验证模型产生的奖励令牌。不过，这一项难以准确估算，因为它取决于硬件和模型规模。以往的工作通过严格限制验证数量必须小于某个阈值，并将其视为常数来简化问题。这种简化忽略了利用额外验证时间生成超过阈值数量词元的机会；当当前推测质量较高时，这样做会带来益处。

树状推测使问题变得更加复杂，因为 AAL 和 $T_{\text{draft}}(W_{\text{draft}})$ 取决于树结构，具体如下所述。

$$Speedup = \frac{AAL(W_{\text{draft}}, D_{\text{draft}}, W_{\text{verify}}) * T_{\text{verifyer}}(1)}{\sum_{D_{\text{draft}}} T_{\text{drafer}}(W_{\text{draft}}) + T_{\text{verifier}}(W_{\text{verify}})} \tag{3}$$

其中 W_{draft}， D_{draft}， W_{verify} 是与树相关的设置，分别表示树的宽度和深度。具体来说，树的深度决定了要启动的草拟迭代次数，而树的宽度决定了要并行草拟的词元数量。基于这一目标，我们能更好地估算系统延迟，使其反映实际的动态树结构。

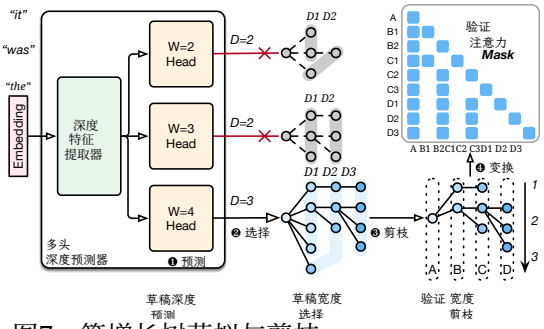

图7：等增长树草拟与剪枝

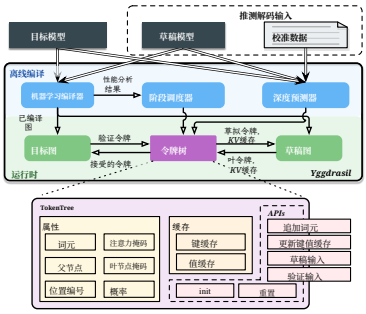

图8：系统实现。

ng.

4.2 等增长树

难点在于找到最优树结构，在适应上下文的同时兼容编译时优化，从而最大化加速比目标。在上一节中，公式 3 表明，加速比不仅取决于 AAL，还取决于紧密耦合的三元组 $\langle W_{\text{draft}}, D_{\text{draft}}, W_{\text{verify}} \rangle$。精确搜索这个高维空间难以实现，因此我们将问题分解为三个贪心但相互协调的子决策，并提出等增长树（EGT）算法将它们结合起来。

EGT 预测 D_{draft}，并启动所有草稿计算图，以消除逐词元控制流停顿。然后，我们选择在预测树深度下能最大化加速比目标的 W_{draft}。最后，我们剪枝草稿树，得到大小为 W_{verify}、能够最大化加速比目标的子树。综合这些步骤，我们得到一个上下文自适应且运行时友好的 树，在 AAL 和吞吐量方面均优于以往的动态序列方法[30, 27]。

草稿深度预测。 我们首先预测 D_{draft}，即草稿模型调用的次数。我们加入一个轻量级多头深度预测器，其中包含一个两层多层感知机编码器和多个深度预测头。该预测器以目标模型的末词元嵌入为输入，并输出预期的接受长度。我们针对每个数据集以及草拟模型/验证器组合离线训练该预测器，训练数据通过在领域内验证语料库上进行性能分析收集一次。与逐轮方法不同，我们会同时实例化所有 D_{draft} 草稿计算图，从而消除中央处理器分支误预测并最大限度地实现图形处理器重叠执行。

草稿宽度选择。 随后，我们选择 W_{draft}，在预测的树深度下最大化加速比目标。这是一种贪心选择，根据预测深度选择最优宽度，以平衡 AAL 与验证时间之间的权衡。虽然 EGT 每个草稿步骤恰好增长 W_{draft} 个叶节点，但我们可以将它们附加到部分树中的任意位置：我们选择路径级预期 AAL 增益最大的那些位置，并使用生成概率作为接受可能性的替代指标[44]。

验证宽度剪枝。 完成草稿构建后，我们求解一个最大值子树问题，以满足验证数量 W_{verify} 的限制。此时，公式 3 中的其他项均已确定，因此问题可简化为：在已构建的推测树中寻找一棵子树，使整体加速比目标最大化。我们通过动态规划算法求解该问题：以自底向上方式遍历树，计算每个节点及其子节点的加速比值。只有在在线等速增长采用贪心预测的情况下，才需要这一后处理步骤；如果增长过程中已知预言机最优解，则无需剪枝。

最后，我们根据树的因果依赖关系生成注意力掩码，为并行验证准备生成的树结构[36]。

5 基于阶段的调度运行时

借助 EGT，我们可以获得一种适合运行时的草拟算法，为图形处理器执行启动静态计算图。然而，故事并未就此结束。推测解码复杂的执行流程和动态控制逻辑会带来大量 CPU 开销，如图 9-(a) 所示。这意味着，某些图形处理器操作由中央处理器根据评估结果动态启动，从而导致图形处理器运行时间出现空档。例如，根据验证结果，我们会

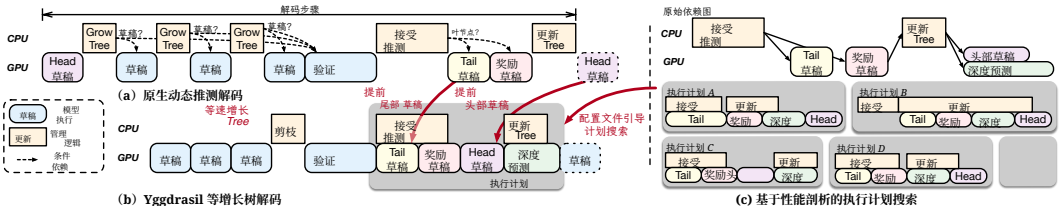

图 9：基于阶段的推测解码调度运行时。

在中央处理器上执行接受推测管理，并即时决定是否为最后一个尾部令牌进行草拟。在执行接受推测阶段时，这会让图形处理器处于空闲状态。为此，我们讨论了两种专门针对推测解码的运行时优化，它们源于推测解码动态且多阶段的特性。首先，我们通过推测打破某些阶段之间的依赖，使其执行提前进行，从而提高重叠效率。然后，我们根据当前 EGT 设置和性能剖析统计数据，搜索性能最佳的执行计划。

5.1 提前阶段执行

如图 9-(a) 所示，在朴素的推测解码流水线中，控制流依赖会导致明显的空闲时段。我们的 EGT 方法消除了草拟阶段的大部分气泡，但验证后的校验步骤仍面临巨大的 CPU 开销。因此，我们将 CPU 管理与 GPU 推理重叠，以缩短实际耗时。挑战在于，推测解码包含数据依赖，看似阻碍了这种重叠——例如，我们必须先知道接受的词元集合，才能确定要草拟的下一个头部词元，从而将执行划分为离散阶段（图 9-(c)）。我们的关键观察是，可以通过使用超出实际所需范围的词元集合来执行下游阶段，从而打破其中若干依赖关系。具体而言，我们将两个阶段提前，使其能够预先执行。

提前尾部起草。 我们不再仅有条件地起草最后一个尾部词元，而是推测性地起草整个候选序列。一旦接受任一叶节点，我们只需复用相应结果，即可消除条件分支。由于接受操作和尾部起草都很轻量，因此可以并发执行。这样便将尾部草稿的 CPU 逻辑从关键路径中移除。

提前头部起草。 头部起草通常使用已确认的根节点解码单个词元，这会在执行关键路径上引入一个小型 GPU 内核。我们转而提出针对整个后续序列进行起草，并在上一轮迭代的额外起草之后立即发起头部起草。由此释放的依赖关系使头部起草能够与接受阶段重叠，进一步压缩执行时间线。

5.2 基于性能剖析的执行计划搜索

虽然提前执行能够实现重叠，但处理额外词元也会增加模型执行开销。这种权衡取决于 EGT 参数、各阶段的相对成本以及目标硬件，因此形成了一个庞大且高度依赖环境的设计空间。为此，我们采用一种离线、基于性能剖析的搜索框架，在图 9-(b) 的块区域内探索阶段重叠策略。根据图 9-(c) 所示的原始依赖图，我们可以看到，提前执行打破了依赖关系，并产生了可自由调度的并行阶段。因此，我们可以利用各阶段的性能分析结果估算每种候选调度方案的端到端延迟，从而搜索最佳执行计划。对于提前执行的阶段，将某个阶段提前会增加起草计算量，但也会带来更多重叠机会，因此最佳权衡点并不显而易见。我们保留原始执行阶段和提前执行阶段的性能剖析指标，并通过简单的网格搜索找到最佳执行计划。得益于定义明确的依赖图，搜索空间较小，可以在编译时离线完成。图 9-(c) 展示了几个有代表性的计划。

6 实现

图 8 展示了 Yggdrasil 的架构，该架构经过精心设计，旨在为现代推理系统优化推测解码。系统分为两个紧密集成的

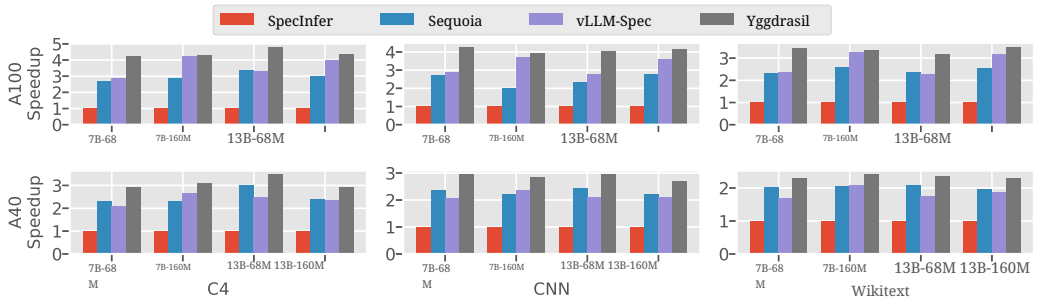

图10：相较于 SpecInfer[31] 的端到端每个词元生成延迟加速结果。

子系统：(1) 编译时优化模块和 (2) 运行时执行模块。在编译时，用户提供一个草稿模型、一个验证器模型和一个小型校准数据集。从此以后，工作流程完全以模型无关的方式运行，无需对原始网络进行任何源代码级修改。Yggdrasil 利用最先进的机器学习编译器对两个模型进行降级，并采用基于性能剖析的成本模型，为目标硬件生成经过优化的执行计划。此外，它还训练一个轻量级深度预测器来引导推测解码，从而确保高效且准确的执行。

在运行时，核心抽象是 TokenTree 类，它封装了 EGT 的语义。TokenTree 可与草稿模型和验证器模型无缝交互，以生成推测令牌并验证其正确性。它提供了完善的接口，用于与草稿模型和验证器模型交换令牌序列及其对应的 KV 缓存张量。KV 缓存存储以及树结构数组、注意力掩码和叶节点位置等其他关键元数据，都作为 TokenTree 的属性得到高效管理。

Yggdrasil 系统基于 PyTorch[2] 框架实现，并利用 TorchInductor 编译器后端实现高性能。值得注意的是，所提出的动态推测树生成算法和流水线运行时旨在适用于广泛的推理系统[34, 20], ，使 Yggdrasil 成为适用于现代 AI 工作负载的通用且高效的解决方案。

7 评估

7.1 实验设置

测试平台。 Yggdrasil 的评估涵盖了配备 NVIDIA A100（80G）和 A40 图形处理器卡，以及 Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz 处理器的服务器。我们的实验依赖以下关键组件：CUDA-11.7 和 TorchInductor。

基准测试与数据集。 所提出的 Yggdrasil 系统旨在与各种目标模型和数据集无缝协作，从而确保广泛的适用性。尽管模型对齐和任务复杂度等因素会影响 AAL 和加速比，但它们与系统设计正交。Yggdrasil 在各种设置下的表现始终优于基线。为展示其有效性，我们在 C4 [39], 维基百科 [45], 和 CNNDaily [18]等语言建模数据集上进行评估。我们使用 Llama-2-7B 和 Llama-2-13B 作为目标模型，并搭配 Llama-68M 和 Llama-160M 草稿模型 [31], ，以每令牌延迟（TPOT）为主要指标。

基线。 我们通过与广泛的相关系统进行比较来评估所提出的 Yggdrasil 系统。在端到端系统性能方面，我们将其与最先进的推测解码系统和大语言模型服务运行时进行比较，包括 vLLM [20] 和 FlexFlow [31]。为基准测试所提出树结构的有效性，我们参考算法领域的树结构开展实验，包括 Sequoia [8] 和 SpecInfer [31]。

7.2 大语言模型解码延迟基准测试

我们评估了 Yggdrasil 在大语言模型解码中的端到端延迟表现，重点关注每个令牌的延迟，以衡量解码效率。图10所示的结果表明，Yggdrasil 始终优于所有基线，平均解码延迟分别提升了 3.98×

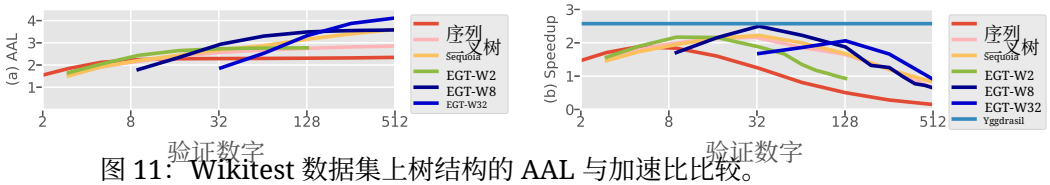

图 11：Wikitest 数据集上树结构的 AAL 与加速比比较。

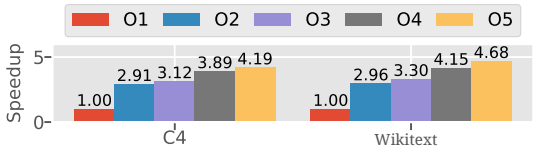

图12：以 Llama-68m 为草拟模型时，Llama-2-7B 模型的优化分解。

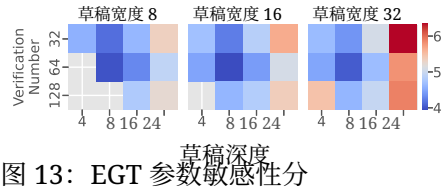

图 13：EGT 参数敏感性分析，使用 Llama-2-7B 和 Llama-68m。

在 A100 GPU 上提升了 $2.76\times$，在 A40 GPU 上也有提升。这证明了其协同设计方法的有效性，超越了 Sequoia 这样的以算法为中心的设计，以及以系统为中心的 SpecInfer。

在各基线中，SpecInfer 的表现最差，这是因为图形处理器执行和服务运行时带来了显著开销。Sequoia 和 vLLM-Spec 利用 TorchInductor [2] 优化模型执行，相比 SpecInfer 分别实现了 $2.45\times$ 和 $2.66\times$ 的平均加速比。不过，Yggdrasil 通过引入上下文感知推测和 CUDA 图优化，进一步将加速比提升至 $3.37\times$。值得注意的是，在 A100 GPU 上性能提升更明显，因为这类 GPU 的硬件利用不足问题更为严重，因此 Yggdrasil 能够充分发挥其优势。

7.3 树结构比较

随后，我们评估本文提出的优化技术的有效性，并共同实现端到端性能提升。我们首先对所提出的 EGT 树结构和硬件感知优化目标的有效性进行基准测试。

树结构。我们在图 11-(a) 中评估了各种树结构的 AAL。随着验证数量增加，序列式推测解码范式的 AAL 有限且很快达到饱和，这限制了其相较于基于树的结构的加速潜力。作为最先进的基线，Sequoia 根据数据集和验证数量静态调整其树结构，从而取得较高的 AAL。然而，它对所有输入都采用相同的树结构，忽略了上下文。相比之下，EGT 会动态调整宽度，在最优验证设置下始终优于 Sequoia 的静态树，凸显了上下文感知树结构选择的重要性。

上下文感知动态树。我们使用公式 3 中的理论加速比来近似树结构有效性，以评估 Yggdrasil 的动态树结构选择过程所产生的影响。图 11-(b) 的结果表明，序列式推测解码实现的加速比最低，凸显了采用树结构的必要性。由于资源饱和导致验证延迟增加（§. 6），N-way 和 Sequoia 等静态树的性能会随着验证数量增加而下降。相比之下，宽度各异的 EGT 在不同验证设置下均能实现最优加速比。Yggdrasil 会为每个请求动态选择最佳宽度和验证数量，始终优于所有基线，并利用上下文和硬件特征实现更优性能。

7.4 优化分解

接下来，我们分析 Yggdrasil 系统中所提出优化技术的有效性。如图 12 所示，我们将整体性能提升分解为五项优化。下文将讨论它们的性能表现及协同效应。

O1 仅应用了 §. 4 中介绍的**延迟最优树投机**。由于其改进效果在很大程度上与运行时支持相关，我们已在上一小节中使用其他树结构，分别对它们的有效性进行了基准测试。在此，O1 可视为通用基线，其改进效果由 AAL 指标体现。

O2 采用**图编译**，并通过 TorchInductor 编译器编译草拟模型和验证器。我们发现，O2 带来了平均 $2.775\times$ 的加速比，是四项优化中效果最显著的一项

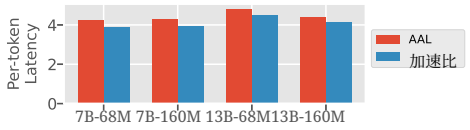

图 14：以加速比和 AAL 为优化目标的比较。

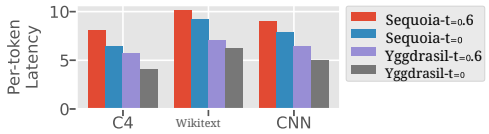

图 15：使用 Llama-7B 和 Llama-68M 时，不同采样温度下的延迟。

优化。这凸显了推测解码系统运行时支持的必要性。请注意，只有采用 EGT 设计，才能实现图，同时保持较高的 AAL。与端到端结果相比，我们发现，使用简单但对运行时友好的序列结构来充分利用编译优化，其效果优于某些复杂的推测算法。Yggdrasil 将这些优化结合起来，兼收两者之长。

O3 采用以加速比为目标的**验证宽度剪枝** 来增强上下文灵活性。通过动态调整验证数量，这项优化相比 O2 平均实现了 $1.07\times$ 的加速比。性能提升主要源于自适应验证策略，该策略通过使验证数量与推理墙钟时间曲线的饱和区间相匹配来降低开销（图 5）。这凸显了自适应验证的有效性；这一技术适用于所有推测解码系统。

O4 纳入了 §. 5 中讨论的**基于图的调度** 优化。与此前的优化相比，其平均性能提升为 $1.21\times$ 。请注意，该指标是根据一次解码迭代所产生的每令牌等效延迟计算的。由于接受的令牌数量会随该迭代中所有阶段的墙钟时间而变化，这项优化对系统性能意义重大。这表明，在现有执行运行时中利用推测解码专属优化十分重要。

O5 是**草稿深度预测器**，它根据上下文嵌入预测最佳起草步骤。为了进行比较，在不使用 O5 时，我们采用深度为 16、宽度为 8 的固定树结构作为基线。我们将在下文进一步分析树设置的影响。我们发现，预测器通过显式考虑上下文的难度，使系统获得了 $1.10\times$加速比。

7.5 敏感性分析

EGT 结构。 我们对 EGT 参数进行了敏感性分析，如图 13 所示。在 §. 4.2 中，我们介绍了在运行时确定最佳参数的方法。这里，我们直观地探索参数搜索空间，并排除无效配置。结果表明，草稿宽度、草稿深度和验证数量之间的相互作用会显著影响每令牌延迟。例如，在本次静态分析中，组合 $D_{draft} = 8$、 $W_{draft} = 8$ 和 $W_{verify} = 64$ 的性能最佳。

加速比目标。 同样，我们在图 14 所示的 C4 数据集上进行消融实验，以展示优化加速比目标而非 AAL 的优势。采用公式 3 后，与直接优化 AAL 相比，在所有草稿器-验证器设置下，性能均额外提升了 8%。这是因为，与 AAL 相比，将加速比作为优化目标能够更好地捕捉运行时特性，包括动态草稿和验证器执行时间。

温度的影响。 我们研究了调整采样温度对 Sequoia 和 Yggdrasil 的影响。如图 15 所示，当温度为 0 时，Sequoia 和 Yggdrasil 的性能都更好。这是因为温度较低时，草稿模型能够更好地与目标模型对齐，从而获得更高的 AAL。此外，在不同温度下，Yggdrasil 始终优于 Sequoia，平均加速比达到 $1.49\times$。

8 相关工作

推测解码变体与服务系统。 除了§. 3中讨论的工作外，还有大量研究提出了推测解码的变体。首先，模型透明研究在不修改目标模型结构的情况下，提高草稿准确率或效率。前瞻解码[14], ANPD[35], 和 NEST[23]采用更高效的检索或统计方法

来生成草拟令牌，而非使用草稿模型。多项研究[31, 41] 设计了更复杂的采样流程，以提高草稿准确率。这些工作与 Yggdrasil 正交。相比之下，模型侵入式方法会修改目标模型，以使目标模型与草稿模型对齐 [51] ，或利用目标模型的部分组件 [26, 12] 或额外的子模块 [4, 25] 生成推测内容，从而降低草稿开销。这些工作需要明确修改原始模型，超出了本文的支持范围。此外，除了§. 7中基准测试的服务系统 [31,8, 20] 外，SmartSpec [27] 和 MagicDec [6] 还研究了推测解码与连续批处理的结合，并探讨如何在在线服务场景中平衡服务等级目标（SLO）与吞吐量提升。然而，它们并未利用推测解码本身的执行效率，而这正是本文的主要关注点。

机器学习系统中的动态支持。 许多研究致力于处理机器学习系统中的动态工作负载。TensorFlow Eager[1] 和 PyTorch Eager[37] 等框架支持在运行时进行动态图构建[16] ，但会导致硬件利用率不佳。包括 PyTorch 2.0[2], DietCode[49], Nimble[21], 和 BladeDISC[50] 在内的机器学习编译器，通过自适应图变换和代码生成，对具有不同输入形状的模型进行优化。对于执行路径依赖于数据的模型, Cocktailer[47], Grape[48], Cortex[13], BrainStorm[24] 和 DyCL[7] 会优化动态控制流，以降低运行时开销。此外，在动态服务场景中，DyNet[32], BatchMaker[15], DVABatch[9], Brainstorm[24], ORCA[46], 和 Llumnix[40] 通过动态调整批次大小或优化调度来提升图形处理器利用率和吞吐量。尽管取得了上述进展，但在处理推测解码所带来的额外开销时，这些工作仍面临局限，因为不可预测的批次大小和控制流增加了显著的复杂性。

9 局限性

Yggdrasil 的核心结果是在延迟最优的设置下得出的：单个交互式请求独占所有可用的 GPU 内存和计算资源。这种配置反映了边缘部署或本地部署的情况，即专用加速器为单个用户或对延迟敏感的流水线提供服务。虽然这一假设便于清晰地分析内存带宽权衡，并为 EGT 起草提供了依据，但它并不适用于批量、面向吞吐量的服务。生产环境中的推理技术栈通常会对数十到数百个提示进行批处理，以最大限度地提高加速器利用率。在这种情况下，将推测解码与批次调度器协同起来，对于整体效率至关重要。设计一种统一策略，同时决定何时进行推测以及如何打包请求，是至关重要的。简言之，尽管 Yggdrasil 推进了单请求场景下的延迟前沿，但如何将这一框架扩展到延迟与吞吐量的联合优化空间——在这一空间中，推测解码与批次调度需要协同求解——仍是一个有待研究的问题。

10 结论

本文提出了 Yggdrasil，一种新颖的延迟最优推测解码框架，在保留编译时优化的同时支持上下文感知推测。此外，该框架基于性能分析驱动的运行时洞察，自适应地选择最优草稿树，从而动态地将执行延迟降至最低。为进一步提升效率，Yggdrasil 采用基于阶段的执行调度策略，简化了推测解码工作流。实验评估表明，Yggdrasil 取得了显著的性能提升，相较于当前最先进的系统实现了更优的加速比。

致谢

本工作得到国家自然科学基金委员会（NSFC）项目 62222210 和上海期智研究院创新计划 SQZ202316 的资助。本文中的任何观点、研究结果和结论仅代表作者个人意见，不一定反映资助方的观点。Jingwen Leng 为通讯作者。作者感谢匿名审稿人提出的富有洞察力的意见，这些意见为本工作的完成作出了重要贡献。

参考文献

[1] Martín Abadi、Paul Barham、Jianmin Chen、Zhifeng Chen、Andy Davis、Jeffrey Dean、Matthieu Devin、Sanjay Ghemawat、Geoffrey Irving、Michael Isard 等。{TensorFlow}：一个用于{大规模} 机器学习的系统。发表于第 *12* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 16*），第 265–283 页，2016 年。

[2] Jason Ansel、Edward Yang、Horace He、Natalia Gimelshein、Animesh Jain、Michael Voznesensky、Bin Bao、Peter Bell、David Berard、Evgeni Burovski、Geeta Chauhan、Anjali Chourdia、Will Constable、Alban Desmaison、Zachary DeVito、Elias Ellison、Will Feng、Jiong Gong、Michael Gschwind、Brian Hirsh、Sherlock Huang、Kshiteej Kalambarkar、Laurent Kirsch、Michael Lazos、Mario Lezcano、Yanbo Liang、Jason Liang、Yinghai Lu、C. K. Luk、Bert Maher、Yunjie Pan、Christian Puhrsch、Matthias Reso、Mark Saroufim、Marcos Yukio Siraichi、Helen Suk、Shunting Zhang、Michael Suo、Phil Tillet、Xu Zhao、Eikan Wang、Keren Zhou、Richard Zou、Xiaodong Wang、Ajit Mathews、William Wen、Gregory Chanan、Peng Wu 和 Soumith Chintala。PyTorch 2：通过动态 Python 字节码转换和图编译加速机器学习。发表于第 *29* 届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第 *2* 卷，ASPLOS '24，第 929–947 页，美国纽约州纽约，2024年。计算机协会。

[3] Tom B Brown. 语言模型是少样本学习器。*arXiv* 预印本 *arXiv:2005.14165*，2020年。[4] Tianle Cai、Yuhong Li、Zhengyang Geng、Hongwu Peng、Jason D Lee、Deming Chen 和 Tri Dao。Medusa：一种采用多解码头的简易大语言模型推理加速框架。*arXiv* 预印本 *arXiv:2401.10774*，2024年。[5] Pei Cao、Edward W Felten、Anna R Karlin 和 Kai Li。集成预取与缓存策略研究。*ACM SIGMETRICS Performance Evaluation Review*，23(1):188–197，1995年。[6] Jian Chen、Vashisth Tiwari、Ranajoy Sadhukhan、Zhuoming Chen、Jinyuan Shi、Ian En-Hsu Yen 和 Beidi Chen。MagicDec：通过推测解码打破长上下文生成中的延迟-吞吐量权衡。*arXiv* 预印本 *arXiv:2408.11049*，2024年。[7] Simin Chen、Shiyi Wei、Cong Liu 和 Wei Yang。DyCL：通过程序重写和图优化实现动态神经网络编译。载于第 *32* 届 *ACM SIGSOFT* 软件测试与分析国际研讨会论文集，第 614–626 页，2023年。[8] Zhuoming Chen、Avner May、Ruslan Svirschevski、Yuhsun Huang、Max Ryabinin、Zhihao Jia 和 Beidi Chen。Sequoia：可扩展、稳健且硬件感知的推测解码。*arXiv* 预印本 *arXiv:2402.12374*，2024年。[9] Weihao Cui、Han Zhao、Quan Chen、Hao Wei、Zirui Li、Deze Zeng、Chao Li 和 Minyi Guo。{DVABatch}：面向多样性的 {多入口}{多出口} 批处理，用于在{DNN} 服务上高效处理 {图形处理器}。载于*2022*年 *USENIX* 年度技术会议（*USENIX ATC 22*），第 183–198 页，2022年。[10] Jacob Devlin。BERT：用于语言理解的深度双向 Transformer 预训练。*arXiv* 预印本 *arXiv:1810.04805*，2018年。[11] Alexey Dosovitskiy。一张图胜过 16×16 个词：用于大规模图像识别的 Transformer。*arXiv* 预印本 *arXiv:2010.11929*，2020年。

[12] Mostafa Elhoushi、Akshat Shrivastava、Diana Liskovich、Basil Hosmer、Bram Wasti、Liangzhen Lai、Anas Mahmoud、Bilge Acun、Saurabh Agarwal、Ahmed Roman 等。层跳过：实现提前退出推理和自推测解码。*arXiv*预印本 *arXiv:2404.16710*，2024年。

[13] David Eliahu、Vishal Bollu、Robert Lucian Chiriac 和 Omer Spillinger。Cortex：面向大规模机器学习的生产基础设施。 [在线], 2022。https://github.com/cortexlabs/cortex。[14] Yichao Fu、Peter Bailis、Ion Stoica 和 Hao Zhang。利用前瞻解码打破大语言模型推理中的顺序依赖。*arXiv*预印本 *arXiv:2402.02057*，2024年。[15] Pin Gao、Lingfan Yu、Yongwei Wu 和 Jinyang Li。利用细胞式批处理实现低延迟循环神经网络推理。载于第十三届 *EuroSys* 会议论文集，第 1–15 页，2018年。[16] Yue Guan、Yuxian Qiu、Jingwen Leng、Fan Yang、Shuo Yu、Yunxin Liu、Yu Feng、Yuhao Zhu、Lidong Zhou、Yun Liang、Chen Zhang、Chao Li 和 Minyi Guo。Amanda：深度神经网络统一检测框架。载于第*29*届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第*1*卷，ASPLOS '24，第 1–18 页，美国纽约州纽约，2024年。计算机协会。[17] Yue Guan、Changming Yu、Yangjie Zhou、Jingwen Leng、Chao Li 和 Minyi Guo。Fractal：联合调优多级稀疏模式，以提升深度神经网络剪枝的准确率和性能。载于第*29*届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第*3*卷，ASPLOS '24，第 416–430 页，美国纽约州纽约，2024年。计算机协会。[18] Karl Moritz Hermann、Tomáš Koˇciský、Edward Grefenstette、Lasse Espeholt、Will Kay、Mustafa Suleyman 和 Phil Blunsom。教机器阅读和理解。载于第*28*届神经信息处理系统进展国际会议论文集——第*1*卷，NIPS'15，第 1693–1701 页，美国马萨诸塞州剑桥，2015年。麻省理工学院出版社。[19] Weiming Hu、Haoyan Zhang、Cong Guo、Yu Feng、Renyang Guan、Zhendong Hua、Zihan Liu、Yue Guan、Minyi Guo 和 Jingwen Leng。M-ant：通过数学自适应数值类型实现面向大语言模型的高效低比特分组量化。载于*2025*年*IEEE*高性能计算机体系结构国际研讨会（*HPCA*），第 1112–1126 页，2025年。[20] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph Gonzalez、Hao Zhang 和 Ion Stoica。利用分页注意力实现大语言模型服务的高效内存管理。载于第*29*届操作系统原理研讨会论文集，第 611–626 页，2023年。[21] Woosuk Kwon、Gyeong-In Yu、Eunji Jeong 和 Byung-Gon Chun。Nimble：面向深度学习的轻量级并行图形处理器任务调度。神经信息处理系统进展，33：8343–8354，2020年。[22] Yaniv Leviathan、Matan Kalman 和 Yossi Matias。通过推测解码实现 Transformer 的快速推理。载于国际机器学习会议，第 19274–19286 页。PMLR，2023年。[23] Minghan Li、Xilun Chen、Ari Holtzman、Beidi Chen、Jimmy Lin、Wen-tau Yih 和 Xi Victoria Lin。用于大语言模型生成与归因的最近邻推测解码。*arXiv*预印本 *arXiv:2405.19325*，2024年。[24] Xin-Ye Li、Jiang-Tian Xue、Zheng Xie 和 Ming Li。跳出代码思维：头脑风暴助力大语言模型进行代码生成。*arXiv*预印本 *arXiv:2305.10679*，2023年。[25] Yuhui Li、Fangyun Wei、Chao Zhang 和 Hongyang Zhang。Eagle：推测采样需要重新审视特征不确定性。*arXiv*预印本 *arXiv:2401.15077*，2024年。[26] Fangcheng Liu、Yehui Tang、Zhenhua Liu、Yunsheng Ni、Kai Han 和 Yunhe Wang。Kangaroo：通过双重提前退出实现无损自推测解码。*arXiv*预印本 *arXiv:2404.18911*，2024年。[27] Xiaoxuan Liu、Cade Daniel、Langxiang Hu、Woosuk Kwon、Zhuohan Li、Xiangxi Mo、Alvin Cheung、Zhijie Deng、Ion Stoica 和 Hao Zhang。利用有效吞吐量优化大语言模型服务中的推测解码。*arXiv*预印本 *arXiv:2406.14066*，2024年。

[28] Zihan Liu、Xinhao Luo、Junxian Guo、Wentao Ni、Yangjie Zhou、Yue Guan、Cong Guo、Weihao Cui、Yu Feng、Minyi Guo、Yuhao Zhu、Minjia Zhang、Chen Jin 和 Jingwen Leng。Vq-llm：面向向量量化增强型大语言模型推理的高性能代码生成。载于*2025*年*IEEE*高性能计算机体系结构国际研讨会（*HPCA*），第1496–1509页，2025年。[29] Jonathan Mamou、Oren Pereg、Daniel Korat、Moshe Berchansky、Nadav Timor、Moshe Wasserblat 和 Roy Schwartz。使用动态推测长度加速推测解码，2024年5月。arXiv:2405.04304 [cs].[30] Jonathan Mamou、Oren Pereg、Daniel Korat、Moshe Berchansky、Nadav Timor、Moshe Wasserblat 和 Roy Schwartz。使用动态推测长度加速推测解码。*arXiv*预印本 *arXiv:2405.04304*，2024年。[31] Xupeng Miao、Gabriele Oliaro、Zhihao Zhang、Xinhao Cheng、Zeyu Wang、Zhengxin Zhang、Rae Ying Yee Wong、Alan Zhu、Lijie Yang、Xiaoxiang Shi 等。Specinfer：通过基于树的推测推理与验证加速大语言模型服务。载于第*29*届*ACM*编程语言与操作系统架构支持国际会议论文集，第*3*卷，第932–949页，2024年。[32] Graham Neubig、Chris Dyer、Yoav Goldberg、Austin Matthews、Waleed Ammar、Antonios Anastasopoulos、Miguel Ballesteros、David Chiang、Daniel Clothiaux、Trevor Cohn 等。DyNet：动态神经网络工具包。*arXiv*预印本 *arXiv:1701.03980*，2017年。[33] Nvidia。Nvidia H100张量核心GPU。 [在线], 2023。https://www.nvidia.com/en-us/ data-center/h100/。[34] Nvidia。TensorRT-LLM。 [在线], 2023。https://github.com/NVIDIA/TensorRT-LLM。[35] Jie Ou、Yueming Chen 和 Wenhong Tian。通过自适应n元语法并行解码无损加速大语言模型。*arXiv*预印本 *arXiv:2404.08698*，2024年。[36] Zaifeng Pan、Yitong Ding、Yue Guan、Zheng Wang、Zhongkai Yu、Xulong Tang、Yida Wang 和 Yufei Ding。Fasttree：优化树结构大语言模型推理的注意力核与运行时。载于第八届机器学习与系统会议，2025年。[37] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga 等。PyTorch：一种命令式风格的高性能深度学习库。神经信息处理系统进展，第32卷，2019年。[38] James E Smith。分支预测策略研究。载于计算机体系结构国际研讨会*25*周年（精选论文），第202–215页，1998年。[39] David So、Wojciech Ma´nke、Hanxiao Liu、Zihang Dai、Noam Shazeer 和 Quoc V Le。探索适用于语言建模的高效 Transformer。神经信息处理系统进展，第34卷：6010–6022，2021年。[40] Biao Sun、Ziming Huang、Hanyu Zhao、Wencong Xiao、Xinyi Zhang、Yong Li 和 Wei Lin。Llumnix：面向大语言模型服务的动态调度。*arXiv*预印本 *arXiv:2406.03243*，2024年。[41] Ziteng Sun、Ananda Theertha Suresh、Jae Hun Ro、Ahmad Beirami、Himanshu Jain 和 Felix Yu。Spectr：通过最优传输实现快速推测解码。神经信息处理系统进展，第36卷，2024年。[42] Alon Talmor、Jonathan Herzig、Nicholas Lourie 和 Jonathan Berant。CommonsenseQA：一项面向常识知识的问答挑战。*arXiv*预印本 *arXiv:1811.00937*，2018年。[43] A Vaswani。Attention Is All You Need。神经信息处理系统进展，2017年。

[44] Jikai Wang、Yi Su、Juntao Li、Qingrong Xia、Zi Ye、Xinyu Duan、Zhefeng Wang 和 Min Zhang。Opt-tree：采用自适应草稿树结构的推测解码。*arXiv*预印本 *arXiv:2406.17276*，2024年。[45] 维基百科贡献者。抄袭——维基百科，自由的百科全书，2004年。 [在线；访问日期：2004]年7月22日。[46] Gyeong-In Yu、Joo Seong Jeong、Geon-Woo Kim、Soojeong Kim 和 Byung-Gon Chun。Orca：一种面向 {基于Transformer的} 生成模型的分布式服务系统。载于第*16*届*USENIX*操作系统设计与实现研讨会（*OSDI 22*），第521–538页，2022年。[47] Chen Zhang、Lingxiao Ma、Jilong Xue、Yining Shi、Ziming Miao、Fan Yang、Jidong Zhai、Zhi Yang 和 Mao Yang。Cocktailer：分析并优化深度学习中的动态控制流。载于第*17*届*USENIX*操作系统设计与实现研讨会（*OSDI 23*），第681–699页，2023年。[48] Zijian Zhang、Kaiyuan Zheng、Zhaorun Chen、Joel Jang、Yi Li、Chaoqi Wang、Mingyu Ding、Dieter Fox 和 Huaxiu Yao。Grape：通过偏好对齐实现机器人策略泛化。*arXiv*预印本 *arXiv:2411.19309*，2024年。[49] Bojian Zheng、Ziheng Jiang、Cody Hao Yu、Haichen Shen、Joshua Fromm、Yizhi Liu、Yida Wang、Luis Ceze、Tianqi Chen 和 Gennady Pekhimenko。DietCode：动态张量程序的自动优化。《机器学习与系统会议论文集》，第4卷：848–863，2022年。[50] Zhen Zheng、Zaifeng Pan、Dalin Wang、Kai Zhu、Wenyi Zhao、Tianyou Guo、Xiafei Qiu、Minmin Sun、Junjie Bai、Feng Zhang 等。BladeDISC：通过编译器方法优化动态形状机器学习工作负载。《*ACM* 数据管理汇刊》，1(3)：1–29，2023年。[51] Yongchao Zhou、Kaifeng Lyu、Ankit Singh Rawat、Aditya Krishna Menon、Afshin Rostamizadeh、Sanjiv Kumar、Jean-François Kagy 和 Rishabh Agarwal。Distillspec：通过知识蒸馏改进推测解码。*arXiv*预印本 *arXiv:2310.08461*，2023年。[52] Zixuan Zhou、Xuefei Ning、Ke Hong、Tianyu Fu、Jiaming Xu、Shiyao Li、Yuming Lou、Luning Wang、Zhihang Yuan、Xiuhong Li 等。大语言模型高效推理综述。*arXiv*预印本 *arXiv:2404.14294*，2024年。[53] Xunyu Zhu、Jian Li、Yong Liu、Can Ma 和 Weiping Wang。大语言模型模型压缩综述，2023年9月。arXiv:2308.07633 [cs]。