顶点课程个人报告

第四组

曹晨旭

日期：2026 年 1 月 11 日

摘　　要

本文为顶点课程结题报告，主要介绍个人在本次顶点课程项目实践过程中的探索与成果。

课程中，我们组的项目包括模型微调、推理优化、APP 开发、科研探索等几个方面，最终我们成功完成了这几部分。其中，我负责推理优化、APP 开发，在科研探索之后总结为新方法，并应用到手机端 APP 的推理框架中。

关键词: 大语言模型，端侧部署，推理优化

1　研究内容

本节详细介绍本人负责的推理优化与 Android 端应用开发的实现过程，包括整体设计思路、关键技术实现方式，以及在实际落地过程中遇到的问题与解决方法。手机端应用 **PeakChat** 基于 `llama.cpp` 官方示例 `examples/llama.android` 二次开发完成，通过修改推理逻辑与上下文管理方式，实现了面向移动端 CPU 的长上下文推理优化。

1.1　设计思路与总体结构

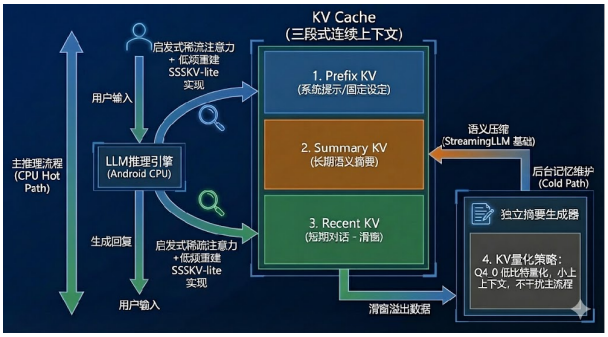

图 1: 推理架构

如图**??**本工作的核心设计目标是在不改变模型结构的前提下，通过对上下文组织方式和 KV cache 管理策略的优化，缓解移动端长对话场景中 KV cache 膨胀和推理性能退化、甚至注意力丧失的问题。整体上将推理流程划分为两条相互解耦的执行路径：

- **主推理链路（Hot Path）**：负责正常对话生成，强调推理稳定性与生成质量；
- **摘要维护链路（Cold Path）**：低频触发，用于生成和更新历史摘要，尽量减少对主推理流程的干扰。

在上下文管理方面，采用三段式连续上下文结构，包括 Prefix、Summary 与 Recent 三部分。其中 Prefix 用于存储系统提示与固定设定，Summary 用于保存历史语义摘要，Recent 则采用滑动窗口方式维护近期对话内容。该设计在控制 KV cache 规模的同时，保证了语义连续性。

1.2 Android 端改造与推理逻辑调整

本项目在 `llama.android` 示例的基础上，对推理逻辑进行了定制化改造，主要包括以下三个方面：

1. **推理输入组装方式调整**：由原先的全量历史拼接，改为按 `Prefix + Summary + Recent + NewQuery` 的形式构造输入提示；
2. **上下文状态管理扩展**：新增 Recent 滑动窗口管理、历史丢弃计数、摘要更新标志等状态变量；
3. **双 Context 推理结构**：引入独立的摘要生成 context，用于摘要推理，避免与主推理链路共享 KV cache 和采样状态。

```
/**
 * 三段式上下文管理：PREFIX + SUMMARY + RECENT
 */
struct ContextSegments {
    llama_tokens prefix_tokens;      // 系统提示词（固定）
    llama_tokens summary_tokens;     // 历史摘要（偶尔更新）
    llama_tokens recent_tokens;      // 最近对话（滑动窗口）

    // 配置参数
    int recent_max = 4096;           // 最近窗口最大 token 数
    int summary_max = 256;           // 摘要最大 token 数
    int trigger_discard = 2048;      // 触发摘要更新的丢弃量

    // 统计
    int dropped_since_last_summary = 0;

    int total_size() const {
        return (int) (prefix_tokens.size() + summary_tokens.size() + recent_tokens.size());
    }
```

图 2: 三段式上下文新增的变量管理

1.3 三段式连续上下文的实现

在实现过程中，重点保证主推理 context 中的 token 始终以连续块形式存储，从而使 CPU 上的注意力计算保持顺序内存访问，提高 cache 命中率。

- **Prefix**：在初始化阶段写入系统提示，后续推理过程中保持不变；
- **Summary**：由摘要生成链路维护，在更新后触发主 KV cache 重建；
- **Recent**：在每轮对话后追加新 token，超过窗口上限时从左侧丢弃。

当 Recent 超过设定的最大长度时，将被丢弃的 token 数量累计记录，为后续摘要触发提供依据。

```cpp
// Check context overflow
if (current_position >= DEFAULT_CONTEXT_SIZE - OVERFLOW_HEADROOM) {
    LOGw("%s: Context full! Triggering compression...", __func__);
    check_and_compress_context();
    if (g_needs_rebuild) {
        if (rebuild_kv_cache() != 0) {
            return nullptr;
        }
    }
}
```

图 3: 一次更新逻辑

1.4 长期记忆压缩与 KV cache 重建策略

为避免频繁生成摘要带来的额外计算开销，本工作采用基于丢弃累计量的触发机制。当 Recent 滑动窗口持续丢弃的 token 数量累计达到设定阈值后，才触发一次摘要更新流程。

摘要更新流程包括以下步骤：

1. 构造摘要输入（由旧 Summary、被丢弃的部分 Recent 组成）；
2. 在独立的摘要 context 中生成新的语义摘要；
3. 更新 Summary 内容，并标记主推理 context 需要重建；
4. 清空主 KV cache，并重新以 `Prefix + Summary + Recent` 进行 prefill。

其中，生成 summary 的方式为：先采用启发式策略进行稀疏注意力的稀疏采样，将 old summary+recent discard 采样得到 sparse summary；再调用生成摘要专用的独立 context 进行 decode，生成最终的 new summary。

该“低频重建”策略避免了 token 级 KV 管理所带来的复杂内存访问问题，更适合 Android CPU 的性能特点。

```cpp
// 策略：保留开头 + 结尾 + 均匀采样中间
const int head = g_segments.summary_max / 4;
const int tail = g_segments.summary_max / 4;
const int middle = g_segments.summary_max - head - tail;

// 开头
result.insert( position: result.end(), first: tokens.begin(), last: tokens.begin() + head);

// 均匀采样中间
const int middle_start = head;
const int middle_end = (int) tokens.size() - tail;
const int step = std::max(1, (middle_end - middle_start) / middle);
for (int i = middle_start; i < middle_end && (int) result.size() < head + middle; i += step) {
    result.push_back(tokens[i]);
}

// 结尾
result.insert( position: result.end(), first: tokens.end() - tail, last: tokens.end());

return result;
```

图 4: 稀疏策略

1.5 面向 CPU 的 SnapKV-lite 实现

考虑到移动端 CPU 上难以低成本获取精确的注意力权重，且 token 级稀疏 KV 访问容易引发 cache miss，本工作未采用严格的 token-level SnapKV，而是通过启发式稀疏注意力采样实现工程化近似。

具体策略是依据先验知识（如 StreamingLLM 中对于维持注意力的关键 attention sink）我们分别保留 kv 的前后 $\frac{1}{4}$，对于中间的进行均匀稀疏采样。仅将这些稀疏 token 作为摘要输入，从而在降低摘要生成成本的同时，保留“重要信息优先”的核心思想。

1.6 KV cache 量化策略

在 KV cache 优化方面，采用任务感知的量化策略以平衡性能与质量：

- 主对话推理 context 使用 Q8_0 KV 量化，以保证生成质量；
- 摘要生成 context 使用 Q4_0 KV 量化，优先降低内存占用与计算开销。

摘要 context 在程序初始化阶段完成创建，并配置独立的 batch 与 sampler，从而实现摘要推理过程与主推理流程的资源隔离。

1.7 实现过程中的问题与解决方法

在实现过程中主要遇到以下问题：

1. **长对话导致推理速度下降**：通过引入滑动窗口与摘要机制限制 KV cache 规模；
2. **频繁摘要带来的性能抖动**：采用累计丢弃阈值触发，降低摘要生成频率；
3. **摘要生成干扰主推理状态**：通过独立 context 实现资源隔离；
4. **CPU 上稀疏 KV 管理效率低**：改用稀疏采样输入与连续块 KV 存储相结合的 SnapKV-lite 方案。

通过上述设计与优化，本项目在 Android CPU-only 环境下实现了稳定、高效的长上下文推理流程。

2 成果展示

本节重点展示本人所负责的推理优化技术在实际系统中的效果，并与未优化情况下的推理行为进行对比说明。需要强调的是，整体系统层面实现了端到端推理性能的显著提升，而本人主要贡献集中在 **上下文管理、KV cache 优化以及长期记忆保持机制**，其核心效果体现在长上下文场景下模型注意力行为的显著改善。

2.1 优化前：长上下文下的注意力丧失问题

在未引入上下文压缩与 KV cache 管理机制之前，手机端推理主要依赖全量历史拼接或简单滑动窗口策略。当对话长度持续增长时，系统面临以下问题：

- **KV cache 无界增长**：历史 token 持续累积，导致注意力计算复杂度和内存占用快速上升；
- **推理性能显著退化**：在 Android CPU-only 环境下，推理延迟随上下文长度线性甚至超线性增长；
- **注意力丧失现象明显**：当上下文超过模型可承受范围后，早期关键信息被滑动窗口直接裁剪，模型无法正确回忆用户约束、设定或早期结论。

在实际对话中，这种注意力丧失表现为模型“遗忘”早期给出的回答、忽略用户先前明确提出的条件，甚至在长对话后出现语义漂移，严重影响交互体验。

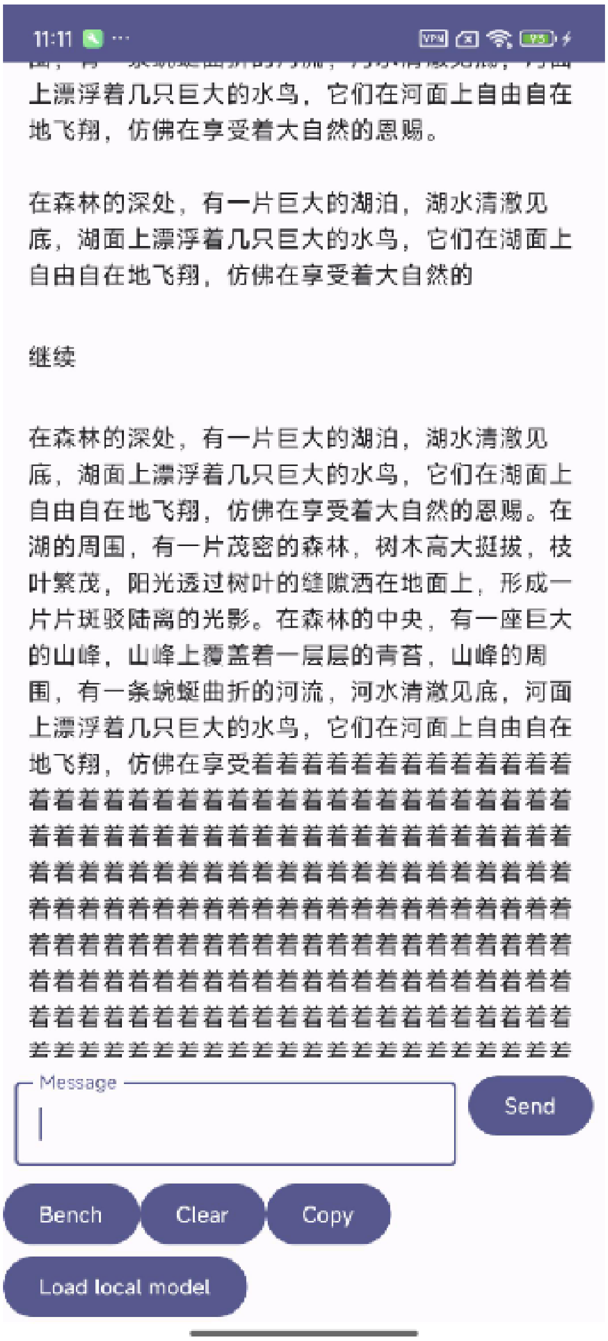

图 5: 优化之前，简单的 KV 裁剪无法维持注意力

2.2 优化后：近似无限长上下文的对话能力

针对上述问题，我们提出并实现了一套基于 StreamingLLM 的上下文优化方案，通过三段式连续上下文结构（Prefix / Summary / Recent）与 KV cache 压缩机制，使系统在有限 KV 资源下具备长期记忆能力。

优化后的系统具有以下显著特征：

- **长期语义信息得以保留**：被裁剪的历史内容不再直接丢弃，而是通过语义摘要形式持续注入上下文；
- **注意力行为更加稳定**：模型能够持续关注早期对话中的关键设定、约束与结论；
- **上下文长度不再成为功能瓶颈**：在理论上，对话轮次可以无限增长，系统仅以固定规模的 KV cache 运行。

尽管底层模型的最大上下文长度有限，但通过稀疏摘要驱动的记忆压缩机制，模型在语义层面实现了“近似无限长上下文”的对话能力。

2.3 对比分析：从注意力丧失到长期记忆保持

将优化前后的系统进行对比，可以总结如下变化：

- 从“历史 token 被直接裁剪”转变为“历史语义被持续压缩与保留”；
- 从“对话轮次增加导致模型遗忘”转变为“模型可稳定遵循长期设定”；
- 从“上下文长度受限于 KV cache 容量”转变为“上下文长度与 KV 规模解耦”。

上述改进并非来源于模型参数规模的扩大，而是通过对推理框架与上下文管理方式的优化实现，体现了我们在 **推理框架层面**的技术贡献。

此外，我们创新性地将 SnapKV 的提取关键注意力的思想，通过启发式的稀疏策略应用到了 summary 生成中，不仅能够加快生成速度，还能够保留关键的历史信息，在端侧移动平台做到了效率与质量的双重适配。

3 个人总结

本人的主要成果体现在以下方面：

- 设计并实现了面向 Android CPU 的三段式上下文管理机制；
- 通过语义摘要替代历史 token，实现长期记忆的压缩存储；
- 将 SnapKV 的稀疏注意力思想应用到了摘要生成策略中，既能加快生成速度又能够保证历史信息；
- 在 llama.cpp 提供的示例模板基础上完成了 APP 的开发；
- 解决了长上下文场景下模型注意力丧失与推理性能退化的问题。

该成果使端侧大语言模型在资源受限环境下具备了可持续的长对话能力，为后续进一步扩展模型规模或应用场景奠定了基础。

参考文献

[1] Dongwon Jo et al. “FastKV: Decoupling of Context Reduction and KV Cache Compression for Prefill-Decoding Acceleration”. In: *arXiv preprint arXiv:2502.01068* (2025).

[2] Yuhong Li et al. “SnapKV: LLM Knows What You are Looking for Before Generation”. In: *arXiv preprint arXiv:2404.14469* (2024).

[3] Zhenmei Shi et al. “Discovering the gems in early layers: Accelerating long-context llms with 1000x input token reduction”. In: *arXiv preprint arXiv:2409.17422* (2024).

[4] Guangxuan Xiao et al. “Efficient Streaming Language Models with Attention Sinks”. In: *arXiv* (2023).

[5] Dongjie Yang et al. “PyramidInfer: Pyramid KV Cache Compression for High-throughput LLM Inference”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 3258–3270. DOI: 10.18653/v1/2024.findings-acl.195. URL: https://aclanthology.org/2024.findings-acl.195/.

[6] Zhenyu Zhang et al. “H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 34661–34710. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf.