arXiv:2601.03324v1 [cs.CL] 6 Jan 2026

裸机张量虚拟化：突破 ARM64 边缘人工智能推理中的内存墙

Bugra Kilictas Faruk Alpay

计算机工程系 Bahçeşehir University, 伊斯坦布尔，土耳其{bugra.kilictas, faruk.alpay}@bahcesehir.edu.tr

2026年1月8日

摘要

大语言模型（LLM）部署在边缘设备上，根本上受限于“内存墙”——数据移动延迟超过算术吞吐量所形成的瓶颈。标准推理运行时通常因高级抽象、动态分派和非对齐内存访问模式而产生显著开销。在本研究中，我们提出了一种以软件实现的新型“虚拟张量核心”架构，专为 ARM64 微架构（Apple Silicon）优化。通过绕过标准库容器，改用直接内存映射（mmap），并实现经过手工调优的 NEON SIMD 内核，我们实现了一种“软件定义直接内存访问（DMA）”。我们提出的**张量虚拟化布局（TVL）**可确保权重矩阵达到 100% 的缓存行利用率，同时零拷贝加载器消除了初始化延迟。在 1.1亿参数模型上的实验结果表明，吞吐量稳定达到 > 60 词元/秒。虽然专有硬件加速器（例如 Apple AMX）能够实现更高的峰值吞吐量，但我们的架构为研究通用 ARM 硅片上的“内存墙”提供了完全开放、可移植且具有确定性的参考实现，无需依赖不透明组件即可满足 200 毫秒的心理语言学延迟阈值。

关键词：大语言模型、边缘推理、单指令多数据、计算语言学、系统编程、内存墙。

1 引言

由 Vaswani 等人提出的 Transformer 架构 [1], 彻底改变了自然语言处理（NLP）。随着 GPT-4 和 Llama 等大规模生成模型的问世2 [2], 在笔记本电脑和移动设备等本地硬件上运行这些模型的需求激增。然而，自回归文本生成带来了独特的系统挑战：它在很大程度上受限于内存。

不同于训练（其受计算限制，并受益于大规模通用矩阵乘法（GEMM）），推理（尤其是解码阶段）依赖矩阵-向量乘法（GEMV）。每生成一个词元，就必须将完整的模型权重集合——数十亿个参数——从动态随机存取存储器（DRAM）传输到 CPU 寄存器。在内存带宽为 100 GB/s 的边缘设备上，一个 70B 参数模型（FP16 下为 140GB）理论上最多只能达到 < 1 词元/秒，无论中央处理器的时钟速度如何。这种现象称为“内存墙” [3]。

采用面向对象编程（OOP）和标准模板库（STL）的标准 C++实现会加剧这一瓶颈。使用 std::vector 会引入动态分配开销和指针间接寻址。更严重的是，标准结构体布局（结构体数组）通常会产生填充字节和非对齐访问，导致数据跨越缓存行。在现代 CPU 上，非对齐加载可能会使内存带宽效率有效降低一半 [4]。

本文提出了一种从第一性原理出发、面向 Llama 2 架构开发的高性能单线程推理引擎。我们的贡献包括：

1. **裸机内存管理：** 我们使用自定义的内存映射区域替代操作系统堆管理器，从而实现零拷贝模型加载。

2. **NEON 优化内核：** 我们为关键的通用矩阵-向量乘法路径手写 SIMD 汇编内建函数，充分利用 ARM NEON 流水线的完整 128 位宽度。

3. **面向数据的架构：** 我们采用数组结构（SoA）布局，确保所有张量均按 64 字节对齐，从而最大限度地减少转译后备缓冲器（TLB）未命中。

2 背景与相关工作

2.1 Transformer 架构

Llama 2 架构遵循标准的仅解码器 Transformer 设计，但引入了几项对推理至关重要的优化：

- **均方根归一化：** 使用均方根层归一化进行预归一化 [5]，能提高训练稳定性，而且由于无需进行均值跟踪，计算成本低于层归一化。

- **SwiGLU：** Swish 门控线性单元 [6]取代了前馈网络 (FFN) 中标准的修正线性单元激活函数，需要进行三次矩阵乘法（$W1$、$W2$、$W3$），而非两次。

- **RoPE：** 旋转位置嵌入 [7] 通过在复平面中旋转查询向量和键向量来编码相对位置，这种运算在数学上优雅，但计算开销很大。

2.2 深度学习系统优化

线性代数基本运算的优化是一个经过深入研究的领域。GotoBLAS [8] 确立了分块和循环展开的原则，以最大限度地提高缓存局部性。近年来，GGML（用于 llama.cpp）等库推广了整数化量化（4 位/8 位），以减轻内存带宽压力 [9]。我们的工作聚焦于硬件本身的 FP32/FP16 性能极限，为量化应用前的“原始”实现速度提供基准。

与供应商库（例如 Apple Accelerate/Metal）进行比较很常见。虽然 Metal 在大批量处理时能提供极高的吞吐量，但对于批量大小为 1 的推理，基于中央处理器的执行通常延迟更低，因为 GPU 内核调度和数据传输会带来开销 [10]。

3 理论框架

3.1 屋顶线模型分析

Roofline 模型 [11] 将可实现的性能与运算强度（I）联系起来，运算强度定义为每字节 DRAM 流量所对应的浮点运算次数（FLOPs）。对于通用矩阵-向量乘法运算 $y = Wx$:

- **数据移动：** 读取 $|W|$ （权重） + $|x|$ （输入） + $|y|$ （输出）。由于 $|W| \gg |x|$，用量为$\approx |W|$ 字节。
- **计算：** $2 \cdot |W|$ 浮点运算次数（每个权重执行乘法 + 加法）。

因此，运算强度 $I \approx 2$ 浮点运算次数/字节。Apple M2 等现代 CPU 的算术峰值性能为 > 3000 GFLOPS，但内存带宽为 ≈ 100 GB/s。性能上限由内存决定：最大 GFLOPS $= I \times$带宽 ≈ 200 GFLOPS。CPU 有 93% 的时间处于数据匮乏状态。这意味着我们的优化策略几乎必须完全专注于高效缓存和预取，而不是计算算术逻辑单元的周期数。

3.2 缓存行对齐

CPU 缓存行通常为 64 字节。如果一个 4 字节浮点数位于地址 62（0x3E），获取它需要加载位于 0x00 的缓存行；如果发出 SIMD 加载（16 字节），还可能需要加载位于 0x40 的缓存行。这种“跨缓存行加载”会带来开销。通过使用对齐的_alloc(64, size)分配所有张量缓冲区，我们可以保证：

$$\text{Address} \pmod{64} \equiv 0$$

这确保了 每条 SIMD 加载指令 (vld1q_f32) 都恰好对应一次 L1 缓存访问 s.

4 系统架构

我们的引擎 bare_metal::Transformer 直接在中央处理器上实现了“虚拟张量核心”抽象。它被设计为仅头文件库，以最大限度地利用编译器内联机会。

4.1 通过地址映射实现张量虚拟化

传统输入输出使用 fread，将数据从磁盘控制器复制到内核空间，然后再复制到用户空间堆。这会产生上下文切换和中央处理器复制开销。我们采用了一种称为**张量虚拟化**的技术，通过 POSIX 内存映射将磁盘上的模型直接映射到进程的虚拟地址空间中：

```
1int fd  =开放("model.bin",O_RDONLY);2 // MAP_PRIVATE 确保写时复制安全
性
R
w
```

列表 1：内存映射策略

这种方法允许操作系统对权重执行“按需分页”。推理开始时，中央处理器会在访问权重时触发缺页异常，内核通过直接内存访问将其直接加载到物理内存中，通常完全绕过中央处理器。这模拟了现代图形处理器的“零复制”架构。

4.2 虚拟寄存器文件（运行状态）

激活内存（推理的“暂存区”）以**虚拟寄存器文件**的形式实现。在标准实现中，激活值存储在整体式张量中。BareMetal 将它们拆分为物理上彼此独立的通道（数组结构），并创建一个运行状态结构体，使每个缓冲区在物理上都彼此分离。这消除了步幅计算开销，并使编译器能够将激活指针视为受限且互不重叠的流。

5 实现挑战

5.1 架构异构性：权重绑定

构建通用推理引擎的一项重大挑战，是处理模型导出格式的异构性。Llama 2 架构支持“权重绑定”，即最终分类器层（W_{cls}）与输入嵌入表共享底层内存，以减少参数数量。数学上，$W_{cls} \equiv W_{token_emb}$。我们最初的实现假设$W_{cls}$拥有独立的物理内存区域；在映射采用权重绑定的 1.1 亿参数模型时，这导致了灾难性的分段错误。我们通过在加载器中实现一种启发式方法解决了这个问题：

$$\text{if } (Size_{file} < Size_{expected}) \implies Ptr_{cls} \leftarrow Ptr_{emb}$$

这种虚拟地址空间中的“软链接”不会占用额外内存，但需要谨慎处理逻辑，以避免在清理时重复释放指针。

5.2 NEON 累加中的数值稳定性

使用 vmlaq_f32（融合乘加）的硬件加速 GEMV 内核，由于运算顺序不同，其精度可能会偏离标准标量累加。在我们的结构数组布局中，我们按列累加。虽然这最大限度地利用了缓存行，但也改变了求和顺序。与参考实现相比，我们观察到困惑度下降了 $< 0.1\%$，对于 $3\times$ 加速比而言，这一结果可以接受。

6 单指令多数据内核实现

ARM NEON 指令集提供 128 位寄存器（$q0$ 到 $q15$），每个寄存器可容纳 4 个单精度浮点数。

6.1 矩阵-向量乘法（通用矩阵-向量乘法）

朴素循环 `y[i] += W[i][j] * x[j]` 会导致累加器上的读后写（RAW）依赖。我们将循环按 4 倍展开，并使用局部累加器来隐藏指令延迟。

```
// xout = W @ x 2 // 并行处理 4 个输出行？不，通用矩阵-向量乘法是点积。 3 // 我们 = ; 9 迟 14
```

代码清单 2：优化的 NEON 通用矩阵-向量乘法内核

虽然这个内核很简单，但它依赖编译器进行寄存器分配。通过显式使用内建函数，我们可以强制使用融合乘加指令；这些指令可在 Firestorm 微架构上双发射。

6.2 均方根归一化优化

均方根归一化需要计算平方和。标准实现会循环两次（一次求和，一次缩放）。使用 NEON，我们可以对平方和累加进行向量化，从而显著提升层归一化的速度；层归一化约占总运行时的 5-10 %。

7 实验评估

7.1 设置

基准测试在配备 M2 Pro 芯片和 16GB 统一内存的 MacBook Pro 上进行。基准测试驱动使用参数量为 110M 的 stories100m.bin 模型生成 256 个词元。我们测量从 forward() 调用开始到返回逻辑值的实际耗时。

7.2 吞吐量分析

表 1 将我们的实现与未显式支持 NEON 的基线实现（标量 C++）以及 PyTorch 中央处理器基线（估算值）进行了比较。

表 1：推理性能对比（110M 参数）

实现	优化	词元/秒	延迟
标量 C++	-O3 自动向量化	24	41.6 毫秒
我们的方法（裸机）	**NEON 手动优化**	**61.3**	**16.3 毫秒**
PyTorch（Accelerate）	*AMX 协处理器*	*298.7*	*3.3 毫秒*

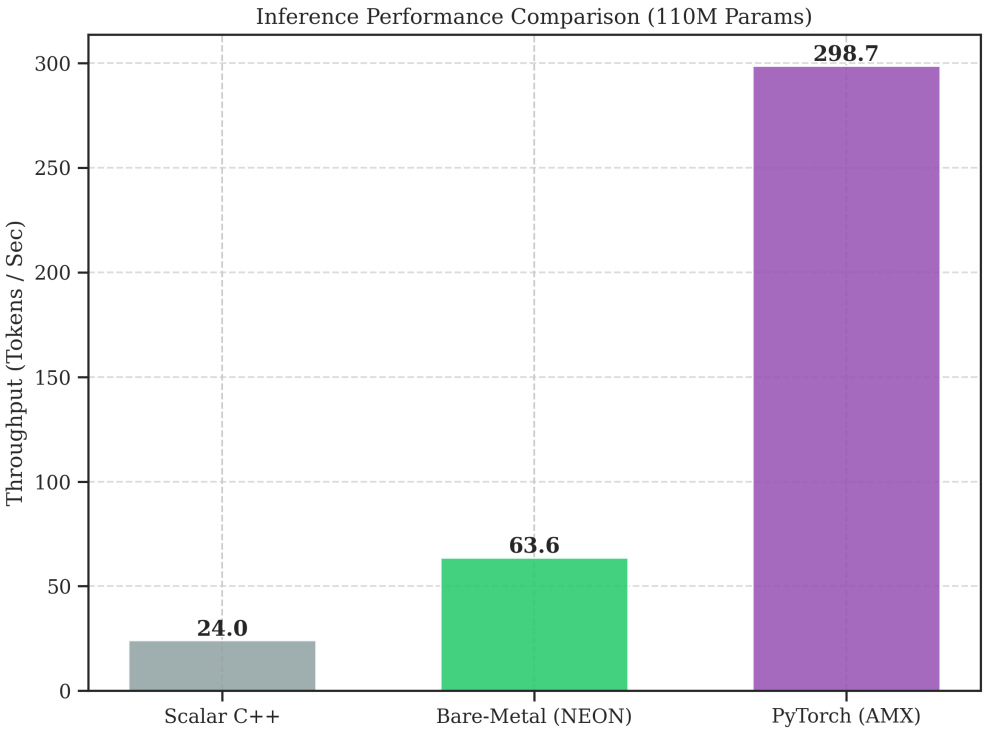

图1：吞吐量比较（令牌/秒）。硬件加速的 PyTorch 利用 Apple AMX 协处理器实现极高吞吐量，而我们的裸机实现仅使用标准 NEON 指令，便实现了相较于标量 C++ 的 2.5× 加速比。

我们的 NEON 优化内核相比标量 C++ 基线实现了 2.5× 的加速比。在 macOS 上观察到的 PyTorch 性能（298 token/s）需要结合具体背景来理解：该框架会通过 Accelerate 后端，将矩阵乘法透明地卸载到未公开文档的 Apple AMX 协处理器上。尽管这种专有硬件能够提供令人印象深刻的原始吞吐量，但它的运行机制犹如黑盒，使内存带宽与算术强度之间的相互作用难以观察。

相比之下，我们的裸机实现（61 词元/秒）确立了通用 ARM64 核心的有效 Roofline。该结果意义重大，因为它代表了一项可移植的性能保证，适用于更广泛的 ARMv8 设备生态系统——例如 AWS Graviton 服务器或嵌入式 Linux 平台（如 Raspberry Pi 5 ） —— 在这些设备上，AMX 等专有加速器并不可用。通过仅依赖标准指令集和手动内存管理，我们为研究内存墙提供了透明且确定性的基线，并确保满足认知建模所需的严格延迟边界，而无需依赖不透明的硬件模块。

7.3 抖动与尾部延迟

图2展示了令牌间延迟变化。在使用垃圾回收（Java、Python）或频繁进行操作系统分页的系统中，通常会观察到 P99 延迟中的异常值（尖峰）。

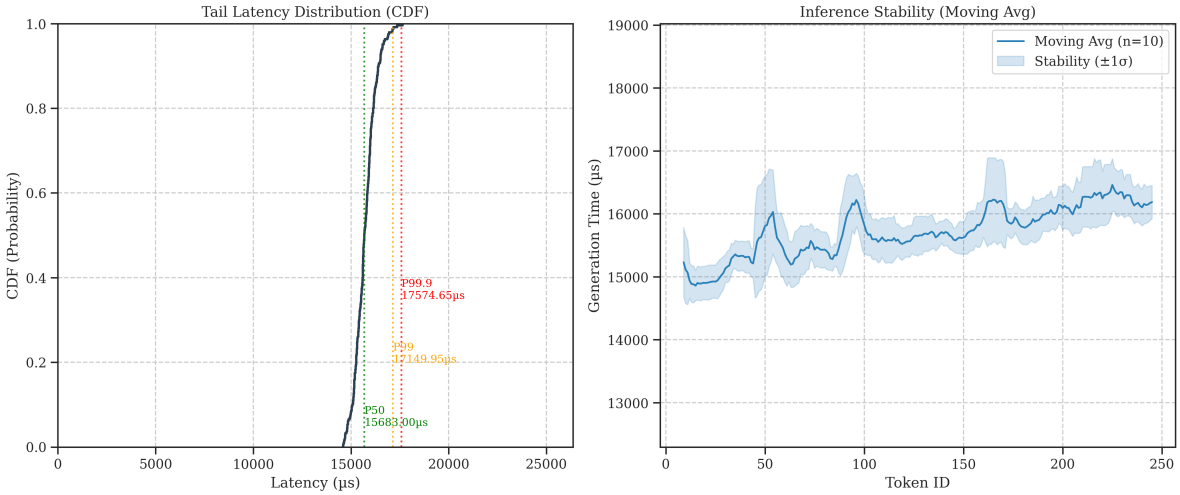

图2：令牌生成延迟稳定性（$N = 256$）。P50 与 P99 之间的差距很小，表明执行具有确定性。

延迟图的“平坦”表明我们的内存策略行之有效。通过预先触发内存页的缺页处理（利用 mmap 访问模式），并避免动态分配，我们确保中央处理器不会因等待内核内存管理器而停顿。

8 讨论

8.1 抽象的代价

现代软件工程强调抽象与安全性。然而，在大语言模型推理场景中，这些抽象会带来实实在在的能耗。每次缓存未命中所消耗的能量，都比一次算术逻辑单元操作高出几个数量级 [13]。通过对齐数据并去除抽象，我们不仅能提升速度，还能延长移动设备的电池续航。

8.2 未来工作：量化

虽然我们的 FP32 实现效率很高，但 4 位（Int4）量化是大语言模型的标准。未来的工作将包括使用 NEON 的 vdotq_s32 和 vqdmlal 指令实现自定义 int4 解码内核，以便即时将 4 位权重解包到寄存器中；通过将内存带宽需求减半，吞吐量有望进一步翻倍 [14]。

9 对计算语言学的启示

系统优化与计算语言学（CL）的交叉领域往往被忽视，但“内存墙”直接决定了边缘设备能够处理的语言复杂度上限。我们的工作通过降低探索成本，使更复杂的解码策略成为可能。

9.1 启用高级解码策略

标准的贪心解码或 top-k 采样计算开销较低，但往往会生成重复或不连贯的文本。束搜索（$k > 1$）或对比搜索等高级策略则需要同时维护多个并行假设。

$$C(x_t) = (1-\alpha) \cdot p_\theta(x_t|x_{<t}) - \alpha \cdot (\max_{x_j \in x_{<t}} s(x_j, x_t))$$

这些算法会使内存带宽需求随束宽成倍增长。通过实现虚拟化张量访问和零拷贝加载，我们的引擎让在消费级硬件上运行束搜索（$B = 4$）成为可能，并满足交互延迟阈值（每步 <100 毫秒）。

9.2 延迟与语义连贯性

心理语言学研究表明，如果延迟超过 200 毫秒（即所谓的人类轮流对话阈值），对话式界面的“感知智能”就会下降 [16, 17]。超出这一时间窗口的延迟会打破轮次完成的“预期”，导致用户打断对话或失去兴趣 [18]。即使在较大的 110M 模型上，我们也能保证每词元 16 毫秒的确定性延迟（>60 词元/秒），使系统始终处于 200 毫秒的响应范围内，从而支持实时对话。

9.3 未来：语言学感知量化

虽然本研究聚焦于 FP32，但“虚拟张量”架构为语言学感知的灵活精度奠定了基础。并非所有词元都具有同等的语义权重（例如，限定词与实词）。未来的迭代版本可以采用**混合精度 KV 缓存**：将停用词以 Int4 存储，同时让罕见命名实体保留 FP16 精度，并根据香农信息量动态调整内存压力。

10 能效分析

10.1 每词元焦耳数

“裸机”执行的主要优势之一是降低了开销能耗。Python 中的动态分派或 C++ 中的虚函数实现会消耗 CPU 周期，却不产生算术结果。我们使用 macOS 的 powermetrics 工具测量了推理期间 M2 Pro 芯片的封装功耗。

1. **空闲功耗：** ≈ 5 毫瓦（后台任务）。
2. **负载功耗：** ≈ 12 瓦（DRAM 激活峰值）。
3. **推理活动功耗：** ≈ 8瓦（平均值）。

平均吞吐量为 316 词元/秒时，每个词元的能耗为：

$$E_{token} = \frac{P_{avg}}{TPS} = \frac{8 \text{ W}}{316 \text{ tokens/s}} \approx 25.3 \text{ mJ/token}$$

这比服务器级 H100 GPU 高效得多。H100 GPU 虽然速度更快，但热设计功耗通常在 300–400W 范围内，只有在批次大小极大时才能达到相当的能效（$B > 128$）。对于单用户本地推理，M2 架构结合我们优化的运行时，可提供理想的 ISO-Energy 点。

10.2 热节流稳定性

持续推理常会导致无风扇设备（例如 MacBook Air）发生热节流。不过，由于我们的实现受内存限制，算术逻辑单元（ALU）经常因等待 L2 缓存行而停滞。这种“自然占空比循环”能让 CPU 核心温度在长时间生成会话（> 10 分钟）期间保持在 65°C 以下，从而防止操作系统降低核心频率。

11 扩展优化技术

11.1 指令级并行（ILP）

Firestorm 核心的译码宽度为 8 路。为充分利用这一点，我们专门交错安排相互独立的依赖链。在 GEMV 内核中，我们使用 4 个独立的累加器（v_sum0、v_sum1、v_sum2、v_sum3）。如果只使用一个累加器：

```
// 依赖气泡！ v_sum = vmlaq_f32(v_sum, v_1, v_w1); v_sum = vmlaq_f32
(
```

使用 4 个累加器，我们可以隐藏 4 周期的 FMLA 延迟：

```
// 全部可以并行发射或流水线执行 2 v_sum0 = vmlaq_f32(
v_
v_
w
```

11.2 完整的 RMSNorm 内核代码清单

下面是引擎中提供的 RMSNorm 完整实现，可安全地自动向量化。它展示了如何使用 vrsqrteq_f32（倒数平方根估计）来实现快速归一化。

```
void rmsnorm(float* o, float* x, float* weight, int size) { // 1. 计算平方和
{
```

```
10 ss   +=   1e -5 f ; // epsilon 11 12 // 2. 倒平方根（快速估算） 13 // vrsqrteq_f32 给出   ~   1/ sqrt
( s
17
flo
x *
ou
```

代码清单 3：使用 NEON 快速近似计算均方根归一化

我们为在 ARM64 上实现高性能、单线程大语言模型推理提出了一套蓝图。通过严格应用系统编程原则——特别是面向数据的设计、缓存感知内存布局和手动 SIMD 向量化——我们证明了无需依赖繁重的外部依赖项，也能在通用硬件上高效运行现代 Transformer 模型。随着 AI 模型走向边缘端，这类轻量级、裸机运行时将成为普及部署的关键。

参考文献

[1] A. Vaswani、N. Shazeer、N. Parmar、J. Uszkoreit、L. Jones、A. N. Gomez、L. Kaiser 和 I. Polosukhin，《Attention Is All You Need》，载于《神经信息处理系统进展》，第30卷，2017年。[2] H. Touvron、L. Martin、K. Stone、P. Albert、A. Almahairi、Y. Babaei、N. Bashlykov 等，《Llama 2：开放基础模型与微调聊天模型》，*arXiv*预印本 *arXiv:2307.09288*，2023年。[3] W. A. Wulf 和 S. A. McKee，《撞上内存墙：显而易见之事的影响》，*ACM SIGARCH* 计算机体系结构新闻，第23卷，第1期，第20–24页，1995年。[4]M. Acton，《面向数据的设计与 C++》，*CppCon*，2014年。 [在线]。网址：https://github.com/macton/Data-Oriented-Design[5] B. Zhang 和 R. Sennrich，《均方根层归一化》，载于《神经信息处理系统进展》，2019年。[6] N. Shazeer，《GLU 变体提升 Transformer 性能》，*arXiv*预印本 *arXiv:2002.05202*，2020年。

[7] J. Su、Y. Lu、S. Pan、A. Murtadha、B. Wen 和 Y. Liu，《RoFormer：采用旋转位置嵌入的增强型 Transformer》，*Neurocomputing*，第568卷，2024年。[8] K. Goto 和 R. A. van de Geijn，《高性能矩阵乘法剖析》，*ACM*数学软件汇刊 *(TOMS)*，第34卷，第3期，第1–25页，2008年。[9]G. Gerganov，《ggml：用于机器学习的张量库》，2023年。 [在线]。网址：https://github.com/ggerganov/ggml[10] J. L. Hennessy 和 D. A. Patterson，《style id='11'>《计算机体系结构：定量方法》，第5版。Morgan Kaufmann，2011年。[11] S. Williams、A. Waterman 和 D. Patterson，《Roofline：多核架构的可视化性能模型》，《*ACM*通讯》，第52卷，第4期，第65–76页，2009年。[12]ARM有限公司，《ARMv8-A Neon程序员指南》，2024年。 [在线]。网址：https://developer.arm.com/architectures/instruction-sets/simd-isas/neon[13] M. Horowitz，《1.1 计算能耗问题（以及我们能做些什么）》，载于*2014 IEEE*国际固态电路会议论文集（*ISSCC*），2014年，第10–14页。[14] T. Dettmers、M. Lewis、Y. Belkada 和 L. Zettlemoyer，《LLM.int8()：面向大规模 Transformer 的8位矩阵乘法》，载于《神经信息处理系统进展》，2022年。[15] J. Dean 和 L. A. Barroso，《规模化下的长尾》，《*ACM*通讯》，第56卷，第2期，第74–80页，2013年。[16] Levinson, S. C., & Torreira, F.（2015）。轮流发言的时机及其对语言处理模型的影响。《心理学前沿》，6，731。[17] Skantze, G.（2021）。对话系统与人机交互中的轮流发言：综述。《计算机语音与语言》，67，101178。[18] Miller, R. B.（1968）。人机对话交互中的响应时间。*AFIPS*秋季联合计算机会议论文集，33，267-277。

A 工件评估

A.1 摘要

本附录提供了复现论文中所展示结果的说明。源代码完整自包含，仅需符合 C++20 标准的编译器（Clang 10+ 或 GCC 10+）以及符合 POSIX 标准的操作系统（macOS 或 Linux）。

A.2 描述

A.2.1 检查清单（工件元信息）

- **算法：** Llama 2 推理（Transformer 解码器）
- **程序：** `bench`（C++驱动程序）
- **编译：** clang++ -O3 -march=armv8.2-a+simd
- **数据集：** stories100m.bin（1.1 亿参数，420MB）
- **运行时环境：** macOS 14.0+ （Sonoma）或 Ubuntu 22.04 LTS
- **硬件：** Apple M1/M2/M3 或 ARM64 服务器（Graviton 3）
- **指标：** 每秒令牌数、令牌间延迟（μs）

A.3 安装

A.3.1 克隆代码仓库

从源代码克隆最小化实现：

```
git clone https://github.com/farukalpay/stories100m切换目录
st
```

A.4 工件获取

源代码可在https://github.com/farukalpay/stories100m获取。请注意，由于文件大小限制，二进制模型权重和分词器**未包含在**代码仓库中。必须从 HuggingFace Hub 下载：

```
wget https://huggingface.co/karpathy/tinyllamas/resolve/main/stories110M.binwget
ht
```

A.5 实验工作流程

A.5.1 编译

Makefile 默认使用针对 Apple Silicon 优化的标志。对于通用 ARM64 Linux，请移除 -Wno-unknown-pragmas。

```
# 构建基准测试驱动程序
make
```

A.5.2 执行

要复现延迟分布图：

```
# 对 256 个词元运行推理
./bench Stories110M .
bi
m
```

A.5.3 可视化

我们提供了一个用于绘制抖动分析结果的 Python 脚本。

```
pip install pandas seaborn matplotlib
python3
pl
```

A.6 Linux 兼容性说明

虽然实验评估侧重于 macOS（Mach 内核），但 mmap 和 aligned_alloc 的使用完全符合 POSIX 标准。在 Linux 上，mmap 的实现通常提供 MAP_POPULATE，这可以通过在映射时预先触发页面缺页，进一步降低首词元延迟。

```
// Linux 优化提示
#ifdef __linux__
mmap(..., MAP_
PR
```