WebLLM：浏览器内高性能 LLM 推理引擎

杨其臻

Sep 02, 2026

1 导言

大模型能力爆炸式增长的同时，隐私、延迟和成本问题也日益凸显。传统云端推理需要将用户数据上传至远程服务器，这不仅带来网络往返延迟、按 Token 计费的直接成本，还可能触及敏感数据合规红线。能否在浏览器中「零部署」地完成大模型推理，成为前端与算法工程师共同关注的命题。WebLLM 给出的答案是：借助 WebGPU 与 WebAssembly，在浏览器沙箱内完成模型加载、KV Cache 管理与算子执行，从而把推理流程完整封闭在用户设备上。接下来的内容将依次梳理背景动机、核心架构、关键技术、性能评测、开发实践以及生态现状，帮助读者建立对浏览器端大模型推理的系统认知。

2 背景与动机

云端推理的瓶颈集中体现在三个方面。首先，跨地域的网络延迟在对话场景下尤为致命，一次往返动辄数百毫秒，严重影响交互体验；其次，按 Token 计费的商业模型在高并发或长文本时成本迅速攀升；最后，医疗、金融等行业对数据驻留有严格要求，任何出域传输都可能违反法规。端侧推理的演进路径经历了从移动端 NPU 到 PC 端独立显卡，再到浏览器 GPU 的三级跨越。WebGPU 1.0 的定稿、WebAssembly SIMD 指令集的普及，以及 WebTransport 带来的高效双向通信，让浏览器首次具备了可与原生环境比拟的并行计算能力。这些能力在离线写作辅助、敏感数据处理、CDN 边缘 Agent 等场景中迅速找到了用武之地。

3 WebLLM 是什么

WebLLM 定位为纯前端、零依赖后端的浏览器内推理引擎，模型文件经量化后总下载量可控制在 50 MB 以内。项目核心由三部分组成：Model Loader 负责分片读取并利用浏览器缓存；Runtime 基于 WebGPU Compute Shader 或 Vulkan 后端执行矩阵运算；Tokenizer 则用 WebAssembly 重写，以获得接近原生 C 的分词性能。当前支持 Llama-2/3、Phi-2、Mistral、Gemma 等主流模型，并提供 FP16 与 INT4 两种量化版本。与同类方案相比，Ollama 仍依赖本地服务进程，MLC-LLM 需要完整的 Python 环境，而 Transformers.js 侧重模型转换而非端到端推理速度。WebLLM 在模型体积与浏览器原生集成度上取得了独特平衡。

4 关键技术拆解

内存布局与 KV Cache 优化是 WebLLM 性能的基石。传统注意力机制需要为每个生成步缓存历史 Key 和 Value，显存占用随序列长度线性增长。WebLLM 借鉴 PagedAttention 思想，将 KV Cache 切分为固定大小的 Page，并在 WebGPU 存储缓冲区中做虚拟地址映射，从而把显存碎片率控制在 5% 以内。算子融合方面，项目采用 TVM Unity 与 MLC LLM 编译栈，把 LayerNorm、QK^T、Softmax、MatMul 四个子算子合并为单一 Shader 内核，减少了多次 CPU-GPU 同步开销。量化策略上，团队实现了 AWQ 与 GPTQ 的 WebAssembly 端到端流程，并引入 INT4 Group-wise 量化，把 7B 参数模型压缩至 3.9 GB，同时在 MMLU 基准上仅损失 0.7 个百分点。异步流水线通过 Web Worker 与 OffscreenCanvas 实现，主线程仅负责 DOM 更新，推理线程独占 GPU 队列，避免了 UI 卡顿。模型分片则利用 HTTP Range Request 按 64 MB 分块拉取，并以 IndexedDB 做二级缓存，实现「用时加载、后台预取」的懒加载策略。

5 性能基准与评测

评测环境覆盖 Apple M2 Max、RTX 4090 与 Intel Arc A770 三种典型 GPU。首 Token 延迟方面，M2 Max 在 Llama-2-7b-chat-q4f16_1 模型上平均为 420 ms，RTX 4090 则降至 180 ms。生成速度上，INT4 量化版本在 M2 Max 达到 38 Tokens/s，在 4090 上突破 110 Tokens/s，均满足实时对话需求。精度损失测试选取 MMLU 与 HumanEval 两个基准，FP16 版本与原始模型差距小于 0.3%，INT4 版本在 MMLU 上下降 0.7 个百分点，在 HumanEval 上下降 1.2 个百分点。移动端功耗测试显示，持续 30 分钟对话后，M2 MacBook Air 电池续航仅下降 6%，证明浏览器内推理的能效已达到工程可用水平。

6 开发上手：10 分钟跑通 Hello World

环境准备要求 Chrome 版本不低于 113，并在 chrome://flags 中启用 WebGPU 标志。安装过程只需一行 npm 命令即可完成依赖拉取。代码样例如下：

```
import { ChatModule } from "@mlc-ai/web-llm";
const chat = new ChatModule();
await chat.reload("Llama-2-7b-chat-q4f16_1");
const res = await chat.generate("Hello");
```

第一行引入官方封装的 ChatModule 类，它对 WebGPU 上下文、Shader 编译与 Tokenizer 初始化做了全生命周期管理。reload 方法内部会触发 Model Loader 的分片下载与 IndexedDB 缓存检查，若本地已存在对应分片则直接反序列化到 GPU 存储缓冲区。generate 方法默认以流式方式返回结果，开发者可通过 chat.interrupt() 随时中止生成。调试时可在 DevTools 的 Performance 面板中观察 GPU 任务队列长度与 Shader 编译耗时，快速定位瓶颈。

7 工程实践

生产级封装可借助 React Hook 进一步抽象。useChat Hook 内部维护消息列表与 AbortController 实例，向外暴露 send、stop 两个方法，同时把流式 Token 拼接到 DOM 的逻辑封装为自定义 Hook，组件层面只需

声明状态即可。安全层面，WebLLM 严格遵守同源策略，所有模型文件必须部署在与页面相同的源；通过 CSP 的 worker-src 与 script-src 指令，可限制 Web Worker 与 Wasm 模块的加载范围。模型版权合规要求开发者在应用中嵌入原始 License 文本，并在 UI 显著位置声明模型来源。离线 PWA 打包可使用 Workbox 预缓存模型分片与 Wasm 文件，再通过 GitHub Pages 完成一键部署，实现「安装即用、无网运行」。

8 生态与社区

生态地图已延伸至 Vercel Edge Functions、Obsidian 插件与 VS Code Tabby 本地 Agent 多个方向。Vercel 通过 Edge Function 做 Prompt 预处理，再把上下文加密后传回浏览器完成最终推理，兼顾了个性化与隐私。Obsidian 插件把 WebLLM 包装为本地 Copilot，在笔记编辑器内实时生成大纲与摘要。贡献路径包括 Shader 优化、模型量化脚本与多语言文档翻译，项目仓库已开放完善的 CONTRIBUTING 指南。典型案例来自某医疗 SaaS：患者病历摘要全程在浏览器内完成，数据既不落盘也不出域，满足 HIPAA 合规要求。

9 挑战与限制

浏览器兼容性仍是最大变量。Safari/WebKit 对 WebGPU 的实现进度落后 Chrome 约两个大版本，移动端 GPU 驱动也存在稳定性问题。模型尺寸与首屏加载的矛盾在低带宽环境下尤为突出，7B INT4 模型首次加载仍需 12 – 15 秒。多轮对话时，KV Cache 随上下文增长而膨胀，显存抖动可能导致浏览器标签页崩溃。最后，浏览器安全沙箱禁止直接访问本地文件系统，模型更新必须依赖网络或用户手动拖拽，限制了部分高级用法。

10 未来路线图

WebGPU 扩展方向包括 FP8 数值格式与 Cooperative Matrix 指令集，前者可把显存占用再降 30%，后者能把矩阵乘法性能提升 2 倍。多模态方面，LLaVA 等视觉-语言模型已在内部测试，预计年内并入主线。联邦学习与个性化 LoRA 微调将借助 WebLLM 的 Wasm 运行时，在用户设备上完成梯度计算，仅上传加密后的参数更新。项目团队正与 WASI-NN 工作组及 Chrome「Built-in AI」提案保持同步，争取把 WebLLM 的核心能力逐步内置到浏览器引擎中。

WebLLM 重新定义了「端-云」边界，让大模型推理从云端服务降级为浏览器原生能力。开发者可从引入 ChatModule 开始，逐步替换云端接口；产品经理则可在隐私合规模块中引入浏览器内推理，降低合规成本。期待读者在评论区分享实验结果，共同推动浏览器端大模型生态成熟。

11 附录

推荐阅读包括 WebGPU 规范、MLC-LLM 编译器论文以及 PagedAttention 原始文献。性能数据表格见项目仓库 benchmark 目录。参考文献与项目链接可在 GitHub MLC-AI/web-llm 获得。