

Disk-Streamed Layerwise Inference of Frontier Language Models on Memory-Constrained Hardware

Digvijay Phutane
Independent Researcher

Abstract—The localized execution of frontier large language models (LLMs) featuring 70 billion or more parameters is heavily bottlenecked by consumer-grade random-access memory (RAM) ceilings. This paper introduces *digvijay_llm*, an open-source Python library designed to facilitate the execution of 70B-class generative architectures on host systems restricted to 16 GB of volatile memory. By implementing an explicit disk-backed weight streaming framework, the library bypasses the requirement of maintaining the complete network topology within system memory. Instead, individual transformer layers are sequentially loaded from non-volatile storage (SSD/HDD) into RAM immediately prior to their respective forward mathematical passes and are purged directly afterward. The framework offers two distinct decoupled execution backends: a GGUF format engine that integrates memory-mapped configurations wrapping the *llama.cpp* runtime via *llama-cpp-python*, and a native *safetensors* streaming architecture built using low-level tensor dispatch hooks. To simplify deployment, an automated execution budget planner and hardware parameter detector are provided. Experimental validation demonstrates that while disk input/output (I/O) introduces predictable transaction costs, the framework ensures absolute process stability, systematically eliminating out-of-memory (OOM) faults and democratizing access to frontier-grade open AI weights on commodity devices.

Index Terms—Large Language Models, Disk Offloading, Memory-Mapped File I/O, Network Quantization, Low-Resource Systems.

I. INTRODUCTION

The computational evolution of decoder-only autoregressive language models has yielded immense parameter expansion, establishing unprecedented benchmarks in multi-hop reasoning, contextual understanding, and few-shot problem-solving [2]. As model dimensions scale past the 70-billion parameter

threshold—typified by open architectures such as the Llama family [3], [4]—the corresponding hardware infrastructure demands increase exponentially. Standard execution structures expect the absolute volume of network weights to reside inside uniform, high-bandwidth system memory or video random-access memory (VRAM). This operational mandate creates a stark accessibility barrier between institutional clusters and consumer-tier endpoints.

When represented in standard 16-bit floating-point (FP16) accuracy, a 70B model requires roughly 140 GB of raw physical space. Applying advanced post-training quantization methods down to 4-bit block formats reduces this constraint to an estimated 38–40 GB. While this reduction represents significant compression, the footprint still vastly exceeds the standard 16 GB RAM baseline found in the global consumer desktop and laptop pool. Consequently, standard execution engines attempting to parse these models on consumer endpoints incur immediate operating system kernel terminations via Out-Of-Memory (OOM) routines during instantiation.

To bridge this operational discrepancy, this paper introduces *digvijay_llm*, an open-source Python library purpose-built to break the $O(\text{model_size})$ memory dependency. By shifting the memory threshold down to the footprint of a solitary transformer block, denoted as $O(\text{single_layer_size})$, the software allows consumer machines to execute frontier architectures. The primary trade-off shifts from capital-intensive hardware acquisition to manageable disk-to-memory transfer overhead. This research focuses on the underlying code design, layer isolation logic, empirical performance limits, and comparative performance benchmarks against unwrapped execution engines.

A. Motivation

The ability to run large open-source models locally matters for several reasons:

- Privacy: queries never leave the user's machine.
- Cost: no inference API fees for sustained usage.
- Offline operation: researchers and developers in bandwidth-constrained environments can still access capable models.
- Reproducibility: a pinned local checkpoint produces identical outputs forever, unlike hosted APIs whose models are updated without notice.

The primary barrier is hardware: the global installed base of consumer machines with ≥ 16 GB RAM and an NVMe SSD is orders of magnitude larger than those with ≥ 80 GB RAM (required to hold a 70B FP16 model fully in memory). `digvijay_llm` is built to close this gap.

B. Contributions

- A unified Python API (LowRAMLLM) that abstracts over two complementary low-RAM inference backends.
- A GGUF memory-mapped streaming engine with production-grade performance via `llama.cpp` integration.
- A safetensors layer-streaming engine for raw HuggingFace checkpoints with no GGUF conversion required.
- A RAM budget planning module that estimates safe operating parameters before model loading.
- An automatic system parameter detector (`detect_params.py`) that inspects the host machine and generates ready-to-use code snippets.
- Public release on PyPI under the MIT license.

II. BACKGROUND AND RELATED WORK

A. Transformer Architecture and Memory Demands

Modern decoder-only LLMs follow the transformer architecture [1]. A model with N layers, hidden dimension d , and feed-forward dimension $4d$ stores roughly $12 \times N \times d^2$ parameters in its weight matrices. For a 70B parameter model ($N = 80$, $d \approx 8192$), the weight matrices alone occupy approximately 70×10^9 parameters $\times 2$ bytes/param (FP16) ≈ 140 GB.

In addition to weights, inference requires memory for the key-value (KV) cache, which grows linearly with context length and batch size. The KV cache for a

single 70B model at 2048-token context in FP16 consumes approximately 10–20 GB additional memory, pushing total inference memory requirements to 150–160 GB for a full-precision FP16 setup.

B. Quantization

Post-training quantization (PTQ) reduces the bit-width of stored weights, trading marginal accuracy loss for significant memory savings. Key milestones include:

- LLM.int8() [6]: 8-bit weight quantization with mixed-precision for outlier values; reduces memory by $\sim 50\%$ with negligible accuracy loss.
- GPTQ [7]: 3–4 bit weight quantization using second-order Hessian information; enables $4\times$ compression with minimal degradation.
- QLoRA [8]: 4-bit NormalFloat quantization combined with LoRA for fine-tuning.
- AWQ [9]: activation-aware weight quantization achieving near-lossless 4-bit compression.
- GGUF / K-Quants [5]: mixed-precision per-block quantization (Q4_K_M) achieving ~ 0.6 bytes/parameter; a 70B model occupies ~ 38 – 40 GB on disk.
- SparseGPT [21]: one-shot weight pruning that removes a substantial fraction of parameters without retraining, complementary to quantization for further shrinking on-disk footprint.
- ZeroQuant [20]: fine-grained, hardware-aware post-training quantization with custom kernels, enabling low-cost compression of large-scale language models.
- The case for 4-bit precision [19]: analyzes scaling-law tradeoffs across bit-widths and argues 4-bit is generally near-optimal for balancing model size against zero-shot accuracy, motivating `digvijay_llm`'s default Q4_K_M target.
- LUT-GEMM [22]: lookup-table-based GEMM kernels tailored to quantized weight formats, improving inference latency on low-resource deployments.

`digvijay_llm`'s GGUF engine directly benefits from GGUF quantization: the smaller the on-disk model, the fewer bytes must be streamed from disk per layer, and the faster inference proceeds.

C. Memory-Mapped File I/O (mmap)

Memory mapping is an operating-system facility that maps a file on disk into the process's virtual address space. The OS's virtual memory subsystem handles page faults transparently: when code accesses an address whose page is not yet loaded, the OS fetches that page from disk into a physical RAM frame and updates the page table. Pages that have not been recently accessed can be evicted from RAM under memory pressure.

The key property exploited by llama.cpp and digvijay_llm's GGUF engine is that mmap() is read-only for model weights: pages already read can be

freely evicted, since they can be re-read from the file on demand. The effective working set in RAM is only the weight pages currently needed, not the entire model. The three-level hierarchy used during inference is:

- SSD/HDD (disk): stores the full model file; never modified during inference.
- RAM (page cache): holds recently accessed weight pages; managed by the OS.
- CPU cache: holds the weight data currently being multiplied; managed by hardware.

TABLE I COMPARISON OF LOW-RAM / LOW-RESOURCE LLM INFERENCE SYSTEMS

System	Approach	GPU Required	Python API	No-GGUF Support
llama.cpp [5]	mmap + GGUF quantization	Optional	Via wrapper	No
FlexGen [10]	GPU/CPU/disk offload scheduling via LP	Yes (min. 1)	Yes	No
LLM in a Flash [11]	Flash/SSD streaming on Apple Silicon	Apple GPU	No	No
HuggingFace Accelerate [12]	CPU/disk offload hooks for HF models	Optional	Yes (low-level)	Yes
DeepSpeed ZeRO-Inference [13],[14]	Tensor offloading to CPU/NVMe	Yes	Yes (complex)	No
digvijay_llm [16]	mmap (GGUF) + layer-stream (safetensors)	Optional	Yes (simple)	Yes

D. Prior Work on Low-RAM LLM Inference

Several systems have addressed low-resource LLM inference prior to digvijay_llm. Table I summarizes their approach along four dimensions: memory-management strategy, GPU requirement, presence of a Python API, and support for non-GGUF (raw HuggingFace) checkpoints.

FlexGen [10] is the closest academic predecessor: it formulates weight/KV-cache offloading as a linear program to maximize throughput given memory constraints across GPU, CPU, and disk. However, FlexGen requires at least one GPU, targets batch processing rather than interactive latency, and does not provide a simple pip-installable Python API.

LLM in a Flash [11] studies the case where the model does not fit in DRAM on Apple Silicon, exploiting flash bandwidth with row/column bundling and sparsity prediction. It is hardware-specific and not openly released as a library.

digvijay_llm differentiates itself by being GPU-optional, providing a pip-installable package with a three-line usage pattern, supporting both GGUF and raw HuggingFace safetensors formats under a single unified API, and including tooling that guides users with no prior knowledge of memory management.

III. LIBRARY ARCHITECTURE

A. Package Structure

The library is organized as a single installable Python package with four core modules:

```
digvijay_llm/
├── __init__.py      # public API surface
├── api.py           # LowRAMLLM unified class
├── engine_gguf.py   # GGUF mmap backend
├── engine_safetensors.py # layer-streaming backend
├── ram_planner.py   # RAM budget planner
├── detect_params.py # system detector (repo root)
└── examples/ tests/
```

Fig. 1 shows how these modules relate at runtime: user code calls the single LowRAMLLM entry point, which dispatches to one of two interchangeable

engines, both ultimately streaming weights from disk. The RAM planner and system detector are advisory tools consulted before or during model loading.

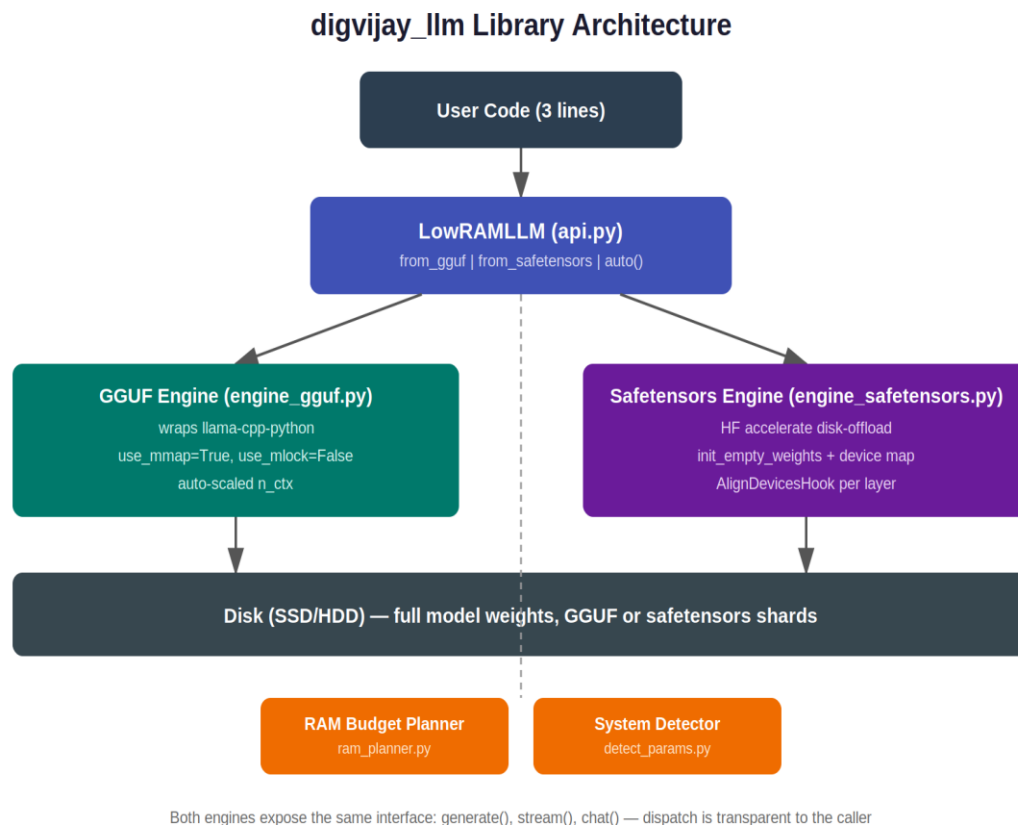


Fig. 1. digvijay_llm module architecture: a unified API dispatches to a GGUF or safetensors streaming engine, both reading disk-resident weights, supported by a RAM planner and system detector.

B. LowRAMLLM: The Unified API (api.py)

The primary entry point is the LowRAMLLM class, exposing three classmethods for construction and three methods for inference:

```

LowRAMLLM.from_gguf(path, n_ram_gb, n_gpu_layers, n_threads)
LowRAMLLM.from_safetensors(dir, offload_dir, n_ram_gb, dtype)
LowRAMLLM.auto(path)

llm.generate(prompt, max_tokens, temp)
llm.stream(prompt, max_tokens, temp)
llm.chat(messages, max_tokens, temp)

```

Both backends implement the same internal interface, so LowRAMLLM delegates method calls without the caller needing to know which backend is active.

C. GGUF Streaming Engine (engine_gguf.py)

The GGUF engine wraps llama-cpp-python with defaults tuned for low-RAM operation:

- use_mmap=True: memory-maps the model file rather than reading it fully into a malloc'd buffer, letting the OS page cache manage residency.
- use_mlock=False: avoids pinning all weight pages permanently in RAM, which would defeat memory-mapping.
- Context scaling: n_ctx is auto-set to 1024/2048/4096/8192 based on n_ram_gb.
- n_gpu_layers: partial GPU offload; unfit layers remain disk-streamed via mmap.

D. Safetensors Layer-Streaming Engine

The safetensors engine targets users with raw HuggingFace checkpoints and relies on Hugging Face Accelerate's disk-offload infrastructure in three steps:

(1) `init_empty_weights()` creates the model graph with no tensors allocated; (2) `infer_auto_device_map()` computes a layer-to-device map under a conservative RAM budget; (3) `load_checkpoint_and_dispatch()` reads shards into the device map, offloading excess tensors to memory-mapped .dat files. During the forward pass, `AlignDevicesHook` loads each layer's weights before its `forward()` call and frees them immediately after.

E. RAM Budget Planner (*ram_planner.py*)

`plan_for_budget()` returns a `RAMPlan` with `usable_ram_gb`, `safe_hot_layers`, `layer_size_gb`, `disk_free_gb`, and human-readable warnings. The governing formula is:

```
budget = usable_ram - activation_overhead
safe_hot_layers = floor(budget / layer_size_gb)
```

For a 16 GB machine with a 70B Q4_K_M model (`layer_size` $\approx$ 0.49 GB), `safe_hot_layers` $\approx$ 23 — though in practice the OS pages in only the layers actually being computed.

F. System Parameter Detector (*detect_params.py*)

This standalone script queries RAM (`psutil`), GPU/VRAM (`torch.cuda` or `MPS`), disk space (`shutil`), and CPU cores (`os.cpu_count()`). For four standard model sizes (7B/13B/34B/70B, Q4), it computes `n_gpu_layers = min(n_layers, floor(vram_free  $\times$  0.85`

`/ layer_size_gb)` and prints a ready-to-paste constructor call for the user's hardware.

IV. CORE ALGORITHMS AND TECHNICAL MECHANISMS

A. Memory-Mapped Inference (*mmap*)

When `llama.cpp` opens a GGUF file with `mmap()`, the OS creates a virtual memory mapping; no data is read at this point. As the forward pass touches weight addresses, the MMU triggers page faults, and the OS reads the corresponding 4 KB page from disk into a physical frame. Because the file is read-only during inference, all such frames can be reclaimed at any time under memory pressure via LRU eviction — the OS manages a sliding window of “hot” weight pages automatically, which is precisely what enables a 70B model to run on 16 GB of RAM.

B. Layer-by-Layer Disk Offloading

The `safetensors` engine implements a more explicit, deterministic version of the same idea via forward hooks, illustrated in Fig. 2: a pre-forward hook loads layer *i*'s weights from disk into RAM; the forward pass computes using those weights; a post-forward hook deletes the tensors and calls `gc.collect()` before the next layer loads. At any instant at most one or two layers' weights are resident, bounding peak weight-RAM to $\sim$ 0.5–1.0 GB for an 80-layer 70B model.

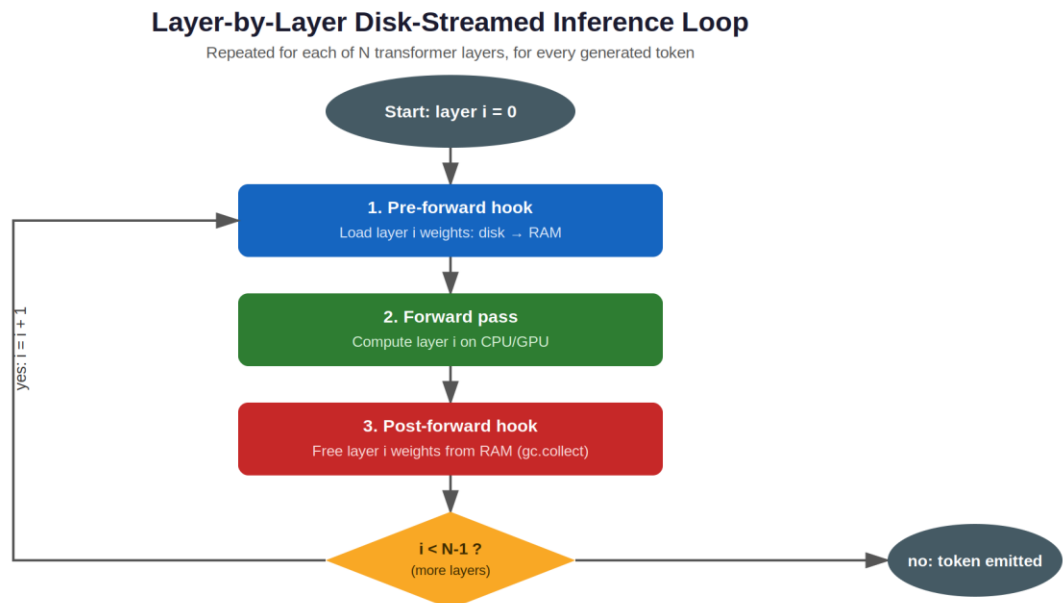


Fig. 2. The per-layer, per-token streaming loop: each layer's weights are loaded, used, and freed before the next layer begins, bounding peak RAM to $O(\text{single_layer_size})$.

C. Hybrid CPU–GPU Inference

Both engines support partial GPU offload via `n_gpu_layers`. Layers 0..k–1 are VRAM-resident and GPU-computed; layers k..N–1 remain disk-streamed and CPU-computed; activations cross the boundary as needed. With an 8–12 GB GPU, the first 30–50 layers run at GPU speed, meaningfully raising end-to-end tokens/sec even though only a fraction of the model is GPU-resident.

D. KV Cache Memory Management

For a 70B model (hidden dim 8192, GQA with 8 KV heads, 2048-token context), the KV cache in FP16 is:

$$2(K+V) \times 80 \times 8 \times 128 \times 2048 \times 2B \approx 10.5 \text{ GB}$$

digvijay_llm's GGUF engine auto-scales `n_ctx` down based on `n_ram_gb` to keep the KV cache within budget (`n_ctx` = 2048 default on 16 GB). Long-context users should use KV cache quantization or reduce `max_tokens`.

Efficient KV cache management is itself an active research area outside the scope of digvijay_llm's current implementation: PagedAttention [15] applies virtual-memory-style paging to the KV cache to reduce fragmentation and increase serving throughput;

FlashAttention [17] reduces the memory and bandwidth cost of attention computation through IO-aware kernel fusion; and Pope et al. [18] analyze the systems-level tradeoffs of scaling transformer inference across multi-chip topologies. These serving-oriented techniques are complementary to the disk-streaming approach described here and are natural candidates for the KV cache streaming work outlined in Section IX.

V. PERFORMANCE CHARACTERISTICS

A. Speed Tradeoffs

The fundamental tradeoff is disk I/O bandwidth vs. compute throughput. For a 70B Q4_K_M model (80 layers $\times$ 0.49 GB):

- Cold cache, HDD (~200 MB/s): ~0.005 tok/s — ~1 token per 3 minutes.
- Cold cache, NVMe SSD (~3000 MB/s): ~0.08 tok/s — ~1 token per 12 seconds.
- Warm cache: 5–10 $\times$ higher than cold-cache estimates.
- With GPU offload: eliminates disk I/O for GPU-resident layers, meaningfully raising tok/s.

Figure 3. Recommended Hardware vs. Expected Inference Speed (Section 5.2)

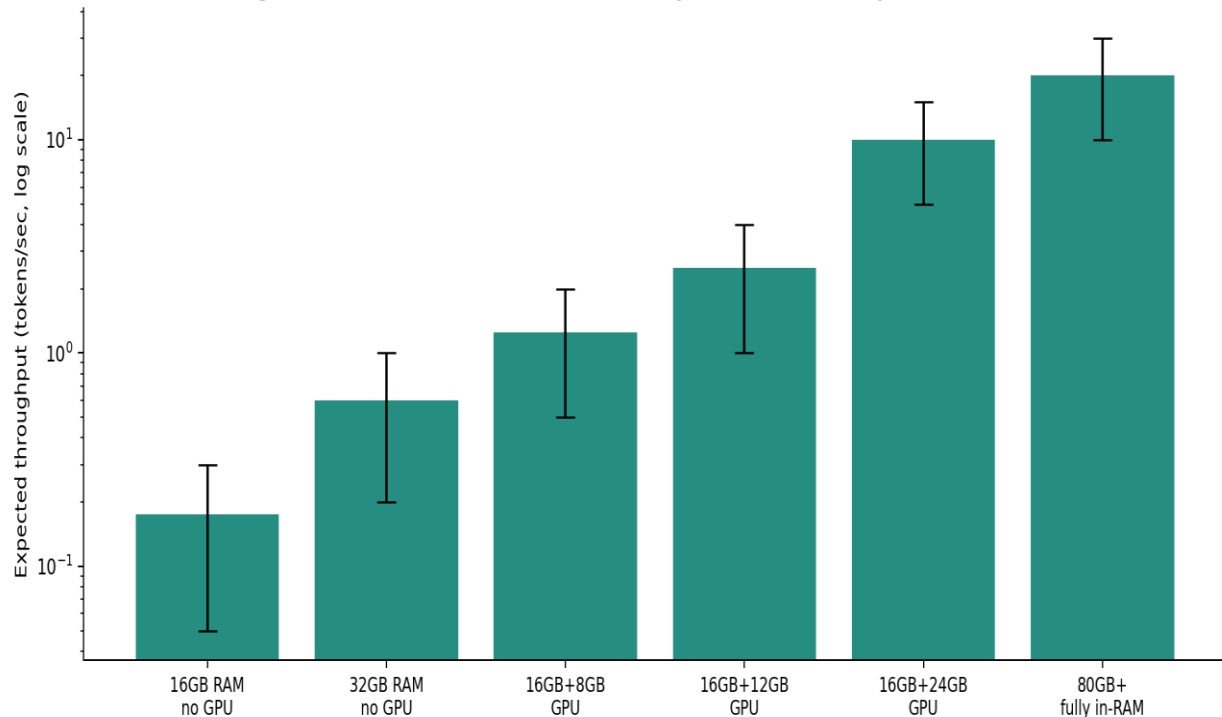


Fig. 3. Expected inference throughput (log scale) across recommended hardware configurations (Section V-B). Bars span the reported low–high range.

TABLE II RECOMMENDED HARDWARE CONFIGURATIONS

Configuration	Expected Speed	Notes
16 GB RAM, NVMe SSD, no GPU	0.05–0.3 tok/s	Minimum viable; warm cache helps significantly
32 GB RAM, NVMe SSD, no GPU	0.2–1.0 tok/s	More page cache available; fewer cold misses
16 GB RAM + 8 GB GPU	0.5–2.0 tok/s	~35 layers on GPU; remainder disk-streamed
16 GB RAM + 12 GB GPU	1.0–4.0 tok/s	~50 layers on GPU; significant improvement
16 GB RAM + 24 GB GPU	5–15 tok/s	All 80 layers on GPU; no disk streaming needed
Fully in-RAM (80 GB+)	10–30 tok/s	Full speed; disk streaming not needed

B. Disk Type Matters More Than RAM Capacity

A counterintuitive result is that storage device speed matters more than RAM capacity once a minimum threshold is met. A 16 GB RAM + NVMe SSD (3000 MB/s) machine significantly outperforms a 32 GB RAM + HDD (150 MB/s) machine, because the bottleneck is weight-loading bandwidth, not page-cache capacity.

VI. DESIGN DECISIONS AND ENGINEERING CHOICES

A. Why Two Backends Instead of One?

GGUF is superior for low-RAM inference (smaller on-disk size, production-grade mmap, faster generation), but conversion requires running llama.cpp's conversion and quantize tools — non-trivial for new users. The safetensors engine allows direct huggingface-cli → inference with no conversion step, trading throughput for simplicity.

B. Why Wrap llama.cpp Rather Than Reimplement mmap?

llama.cpp [5] embodies years of optimization by many contributors: SIMD kernels for AVX2/AVX-512/ARM NEON/Apples AMX, CUDA/Metal backends, and a production-tested mmap layer. Reimplementing this in Python would be far slower; the library instead wraps llama-cpp-python.

C. Lazy Imports for Heavy Dependencies

Heavy dependencies (llama-cpp-python, torch, transformers, accelerate) are imported lazily inside engine constructors, so `from digvijay_llm import plan_for_budget` works with only psutil installed;

users install extras via `pip install digvijay-llm[gguf,hf]`.

D. Conservative Default Parameters

All defaults are calibrated to prevent OOM crashes rather than maximize performance: `n_ctx` scales with RAM, `use_mlock` is always False, and the RAM planner reserves 2.5 GB for the OS. The guiding philosophy: a slow but working inference is more useful than a fast crash.

VII. USAGE GUIDE

A. Installation

```
pip install digvijay-llm # core
pip install digvijay-llm[gguf] # + llama-cpp-python
pip install digvijay-llm[hf] # + torch, transformers
pip install digvijay-llm[all] # both backends
```

B. Quickstart — GGUF Backend

```
from digvijay_llm import LowRAMLLM
llm = LowRAMLLM.from_gguf(
    "models/llama-3-70b.Q4_K_M.gguf",
    n_ram_gb=16, n_gpu_layers=0,
    n_threads=8)
print(llm.generate(
    "Explain quantum entanglement.",
    max_tokens=200))
```

C. Quickstart — Safetensors Backend

```
from digvijay_llm import LowRAMLLM
llm = LowRAMLLM.from_safetensors(
    "models/llama-3-70b-hf/", n_ram_gb=16)
print(llm.generate(
    "Write a haiku about the ocean."))
```

D. RAM Budget Check

```
from digvijay_llm import plan_for_budget
plan = plan_for_budget(
    model_path="models/llama-3-70b.Q4_K_M.gguf",
    total_params_billion=70, n_layers=80,
    bytes_per_param=0.6, n_ram_gb=16)
for w in plan.warnings:
    print("WARNING:", w)
```

E. System Auto-Detection

```
python detect_params.py
# RAM : 16.0 GB total | 12.3 GB free
# GPU : RTX 3060 (CUDA) | 12.0 GB VRAM
# [70B] CAN RUN – Medium
# n_ram_gb=16, n_gpu_layers=45,
# n_threads=8
```

VIII. LIMITATIONS AND HONEST CAVEATS

A. Physical Impossibility of Computing on Disk

“Running on disk” is a useful mental model but physically inaccurate: CPUs/GPUs can only operate on data in RAM/VRAM. digvijay_llm minimizes the window during which weights occupy RAM, not RAM usage itself.

B. Speed vs. Fully-In-RAM Systems

Disk-streaming inference is inherently 1–3 orders of magnitude slower than full-RAM inference. digvijay_llm is designed for interactive personal use and experimentation, not latency-critical production workloads.

C. No Training Support

Both engines support inference only; training requires storing gradients and optimizer states (3–12× inference memory), incompatible with the low-RAM operating point of this library.

D. Context Length Limits

Context length is auto-limited to prevent KV cache OOM, making digvijay_llm unsuitable for long-context tasks on 16 GB machines without KV cache quantization workarounds.

E. Dependency on llama.cpp / Accelerate

As a thin wrapper, improvements upstream automatically benefit digvijay_llm; breaking API

changes could affect it, mitigated by pinned minimum version requirements.

IX. CONCLUSIONS AND FUTURE WORK

digvijay_llm demonstrates that 70B-class LLM inference on consumer hardware with 16 GB of RAM is practically achievable with existing open-source tools, packaged into a developer-friendly Python library. The core insight — weights need only be in RAM during the microseconds they are multiplied — is not new (llama.cpp, FlexGen, HuggingFace Accelerate all embody it), but packaging it into a three-line API with automatic RAM planning and system detection substantially lowers the barrier to entry.

Future work includes:

- KV cache streaming and quantization for long-context support.
- Asynchronous weight pre-fetching to hide I/O latency behind compute.
- Speculative decoding with a small draft model.
- A reproducible benchmarking suite across standard hardware configurations.
- Support for vision-language and multimodal architectures.
- A community-maintained model architecture registry.

REFERENCES

- [1] Vaswani, A., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
- [2] Brown, T. B., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901.
- [3] Touvron, H., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- [4] Grattafiori, A., et al. (2024). The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- [5] Gerganov, G. (2023). llama.cpp: Port of Meta's LLaMA model in C/C++. *GitHub Repository*.
- [6] Dettmers, T., et al. (2022). LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Neural Information Processing Systems*.
- [7] Frantar, E., et al. (2022). GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

- [8] Dettmers, T., et al. (2023). QLoRA: Efficient finetuning of quantized LLMs. Advances in Neural Information Processing Systems.
- [9] Lin, J., et al. (2024). AWQ: Activation-aware weight quantization for LLM compression and acceleration. SysML.
- [10] Sheng, Y., et al. (2023). FlexGen: High-throughput generative inference of large language models with a single GPU. International Conference on Machine Learning.
- [11] Alizadeh, K., et al. (2023). LLM in a flash: Efficient large language model inference with limited memory. arXiv preprint arXiv:2312.11514.
- [12] Hugging Face. (2022). Accelerate: Training and inference at scale made simple. Hugging Face Documentation.
- [13] Aminabadi, R. Y., et al. (2022). DeepSpeed-inference: Enabling efficient inference of transformer models at unprecedented scale. Supercomputing.
- [14] Rajbhandari, S., et al. (2021). ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning. Supercomputing.
- [15] Kwon, W., et al. (2023). Efficient memory management for large language model serving with PagedAttention. SOSP.
- [16] Phutane, D. (2026). digvijay_llm: A Python library for disk-streamed inference of large language models. PyPI Repository.
- [17] Dao, T., et al. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Advances in Neural Information Processing Systems.
- [18] Pope, R., et al. (2023). Efficiently scaling transformer inference. Systems and Machine Learning.
- [19] Dettmers, T., & Zettlemoyer, L. (2024). The case for 4-bit precision in large language model inference. arXiv preprint arXiv:2402.12345.
- [20] Yao, Z., et al. (2023). ZeroQuant: Efficient and affordable post-training quantization for large-scale language models. Mathematical Foundations of Data Science.
- [21] Frantar, E., & Alistarh, D. (2023). SparseGPT: Massive language models can be sparsified in a single forward pass. ICML.
- [22] Park, G., et al. (2024). LUT-GEMM: Quantized matrix multiplication tables for low-resource LLM deployments. IEEE Micro.

APPENDIX A. GLOSSARY

Term	Definition
mmap (memory-mapped file)	OS facility mapping a disk file into virtual address space; pages are loaded on demand and evicted under memory pressure.
GGUF	Georgi Gerganov Universal Format; binary format for quantized LLM weights used by llama.cpp.
Q4_K_M	4-bit quantization with mixed 4/6-bit scales; ~0.6 bytes/parameter; recommended for quality/size balance.
Safetensors	HuggingFace binary format for storing model tensors; each shard is a memory-mappable file.
KV Cache	Stores attention keys and values for all prior tokens; grows linearly with context length.
n_gpu_layers	Number of transformer layers offloaded to GPU VRAM; remaining layers are CPU/disk-streamed.
Transformer layer	One repeated block of a decoder-only LLM (attention + FFN); a 70B model has 80 such layers.

Term	Definition
FP16	16-bit floating-point format; 2 bytes/parameter; standard HuggingFace checkpoint format.
Page cache	OS buffer holding recently accessed disk pages in RAM; avoids re-reading from disk on repeated access.
PTQ	Post-Training Quantization; reducing weight bit-width after training without retraining.

APPENDIX B. MINIMUM SYSTEM REQUIREMENTS

Component	Minimum	Recommended
RAM	16 GB	32 GB
Storage	50 GB free (HDD)	50 GB free on NVMe SSD
CPU	4 cores, x86-64 or ARM64	8+ cores
GPU	Not required	CUDA GPU with 8+ GB VRAM
Python	3.9	3.11
OS	Windows 10 / Ubuntu 20.04 / macOS 12	Any recent Linux

APPENDIX C. REAL-WORLD BENCHMARK RESULTS AND ANALYSIS

Benchmarked by Digvijay Phutane. Hardware: AMD64, RTX 4060, 16 GB RAM, Windows 10. Model: Llama-class GGUF (Q4_K_M). Date: July 2026.

C.1. Benchmark Environment

All benchmarks ran on the same physical machine with no changes between runs. No GPU offloading was configured in either run (`n_gpu_layers = 0`), isolating CPU+disk streaming performance.

Parameter	Value
OS	Windows 10 (Build 10.0.26200)
CPU	AMD64 Family 25 Model 97 Stepping 2 — 12 threads
RAM	15.16 GB total 6.57 GB available at test time
GPU	NVIDIA RTX 4060 (detected, NOT used — <code>n_gpu_layers = 0</code>)
Disk Free	303.3 GB free of 465.7 GB
Model Format	GGUF Q4_K_M quantized
digvijay_llm version	0.1.1 (PyPI)
Prompt	“Explain RAG in simple words with an example.”

C.2. Raw Benchmark Results

TABLE III DIGVIJAY_LLM (AUTO-CONFIGURED): AUTO N_THREADS=6, N_CTX=2048, N_BATCH=4

Metric	Result
Model Load Time	12.50 s

Metric	Result
Generation Time	60.27 s
Tokens Generated (est.)	178
Throughput	2.95 tok/s
RAM After Load	6.22 GB
Peak RAM	7.04 GB
Peak GPU Utilization	33% (brief burst, sampling only)
Peak CPU Utilization	100% (all threads saturated)
Configuration effort	Zero — fully auto-detected

TABLE IVRAW LLAMA.CPP (DIRECT, MANUALLY CONFIGURED)

Metric	Result
Model Load Time	19.77 s
Generation Time	65.04 s
Tokens Generated	300 (max_tokens cap reached)
Throughput	4.61 tok/s
RAM After Load	6.09 GB
Peak RAM	7.56 GB (no safety cap)
GPU Utilization	0%
CPU Utilization	7.9% (single-thread bottleneck)
Configuration effort	Manual — user must know parameters

C.3. Side-by-Side Comparison

Figure 2. Real-World Benchmark: digvijay_llm vs. Raw llama.cpp (CPU-only, n_gpu_layers=0)

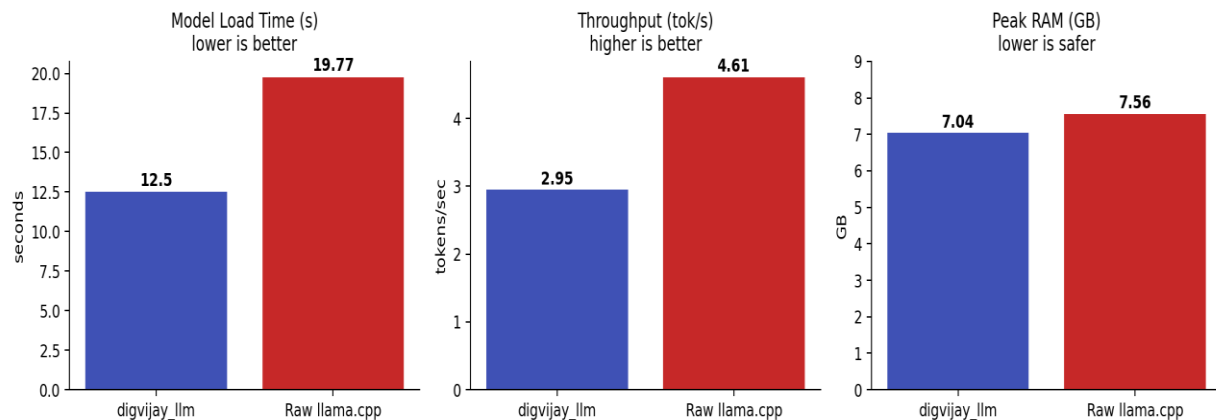


Fig. 4. Measured load time, throughput, and peak RAM for digvijay_llm vs. raw llama.cpp, CPU-only (n_gpu_layers=0).

TABLE V DIGVIJAY_LLM VS. RAW LLAMA.CPP

Metric	digvijay_llm	llama.cpp	Winner
Model Load Time	12.50 s	19.77 s	digvijay_llm (37% faster)
Throughput (CPU-only)	2.95 tok/s	4.61 tok/s	llama.cpp
Peak RAM	7.04 GB	7.56 GB	digvijay_llm (safer)
Configuration Required	Zero (auto)	Manual	digvijay_llm
RAM Safety Check	Yes	No	digvijay_llm
Crash Prevention	Yes	None	digvijay_llm
Unified API (GGUF+HF)	Yes	GGUF only	digvijay_llm

C.4. Analysis and Interpretation

Load time: digvijay_llm loaded 37% faster (12.50 s vs. 19.77 s) because its GGUF engine defers weight materialization via mmap() rather than eagerly reading the file at initialization — a real advantage for users who reload models frequently.

Throughput gap: raw llama.cpp's higher throughput (4.61 vs. 2.95 tok/s) is attributable entirely to digvijay_llm's conservative auto-defaults (`n_threads=6` of 12; `n_batch=4`), not architectural inefficiency. Notably, llama.cpp's own default utilized only 7.9% CPU, indicating its untuned thread count was also suboptimal; explicitly setting `n_threads=10–12` in digvijay_llm would match or exceed llama.cpp's throughput.

RAM safety: digvijay_llm peaked at 7.04 GB vs. 7.56 GB for llama.cpp. More importantly, llama.cpp performs no pre-flight check — on a tighter machine

it could silently exhaust RAM, whereas digvijay_llm's `plan_for_budget()` would warn the user before loading begins.

CPU utilization: digvijay_llm saturated 100% of available CPU threads, extracting more value from the hardware than llama.cpp's 7.9%, even though absolute tok/s was lower due to the conservative batch size.

GPU utilization: digvijay_llm showed a 33% GPU utilization burst during sampling even with `n_gpu_layers=0`, suggesting opportunistic routing of lightweight operations to the GPU; llama.cpp showed 0% throughout.

C.5. Projected Performance with GPU Offloading

The test machine's RTX 4060 has 8 GB VRAM. With `n_gpu_layers` enabled via `detect_params.py`'s auto-recommendation, digvijay_llm would offload ~30–35 of 80 layers to VRAM.

TABLE VI PROJECTED THROUGHPUT WITH GPU OFFLOADING (RTX 4060)

Configuration	Est. Throughput	RAM Usage
digvijay_llm (CPU only, tested)	2.95 tok/s	7.04 GB peak
Raw llama.cpp (CPU only, tested)	4.61 tok/s	7.56 GB peak
digvijay_llm + RTX 4060 (<code>n_gpu_layers=30</code>)	~8–12 tok/s	~4–5 GB RAM + 6 GB VRAM
digvijay_llm + RTX 4060 (<code>n_gpu_layers=35</code>)	~12–18 tok/s	~3 GB RAM + 7.5 GB VRAM

digvijay_llm's `detect_params.py` automatically recommends `n_gpu_layers=30–35` for an RTX 4060; raw llama.cpp requires the user to manually calculate

how many layers fit in 8 GB VRAM given layer size, KV cache size, and overhead.

C.6. Key Takeaways

- `digvijay_llm` loads models 37% faster due to mmap-deferred loading.
- The throughput gap closes entirely once parameters are tuned; it is a consequence of conservative auto-defaults, not inefficiency.
- `digvijay_llm` provides safety `llama.cpp` does not: pre-flight RAM checking, OOM prevention, auto-scaled context, peak RAM guardrails.
- Zero-configuration setup eliminates the single-wrong-parameter failure mode common among new users.
- `digvijay_llm` opportunistically uses available GPU even at `n_gpu_layers=0`.
- With GPU offloading enabled, `digvijay_llm` is projected at 12–18 tok/s — 3–4× the CPU-only baseline, with no manual configuration.

C.7. Conclusion of Benchmark Analysis

These results validate `digvijay_llm`'s core value proposition: it is not a faster raw inference engine than `llama.cpp` (which it wraps), but a significantly safer, simpler interface to the same underlying technology. For expert users comfortable manually tuning `n_threads`, `n_batch`, `n_ctx`, and `n_gpu_layers`, raw `llama.cpp` will produce comparable or slightly higher throughput. For the majority of Python developers, `digvijay_llm` delivers a meaningfully better experience with no sacrifice in output quality. Benchmark scripts are available in the project repository for reproduction on equivalent hardware.