PRIMA.CPP: Speeding Up 70B-Scale LLM Inference on Low-Resource Everyday Home Clusters

Zonghang Li¹ Tao Li² Wenjiao Feng² Mohsen Guizani¹ Hongfang Yu²

¹Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi, UAE

²University of Electronic Science and Technology of China, Chengdu, China

Abstract

Emergency of DeepSeek R1 and QwQ 32B have broken through performance barriers for running frontier large language models (LLMs) on home devices. While consumer hardware is getting stronger and model quantization is improving, existing end-side solutions still demand GPU clusters, large RAM/VRAM, and high bandwidth, far beyond what a common home cluster can handle. This paper introduces prima.cpp, a distributed inference system that runs 70B-scale models on everyday home devices using a mix of CPU/GPU, low RAM/VRAM, Wi-Fi, and cross-platform support. It uses mmap to manage model weights and introduces piped-ring parallelism with prefetching to hide disk loading. By modeling heterogeneity in computation, communication, disk, memory (and its management behavior), and OS, it optimally assigns model layers to each device's CPU and GPU, further reducing token latency. An elegant algorithm named Halda is proposed to solve this NP-hard assignment problem. We evaluate prima.cpp on a common four-node home cluster. It outperforms llama.cpp, exo, and dllama on 30B+ models while keeping memory pressure below 6%. This brings frontier 30B-70B models, such as Llama 3, DeepSeek R1, Owen 2.5, and OwQ to home assistants, making advanced AI truly accessible to individuals. The code is open source and available at https://github.com/Lizonghang/prima.cpp.

1 Introduction

Current large language models (LLMs) are mostly cloud-based, but DeepSeek [DeepSeek-AI, 2025] changes this. Its R1 series, spanning from 1.5B to 70B, bring small models closer to frontier performance. Its 70B version even surpasses cloud-based models like GPT-40 and Claude 3.5 Sonnet. This has sparked interest in deploying LLMs locally on users' own devices, especially with the explosion of open-source LLMs. However, limited by weak chips and small RAM, user devices struggle to run anything beyond 10B, even with 4-bit quantization. For example, running Qwen 2.5-14B (Q4K) on a Mac M1 with 8 GiB RAM takes a staggering 10 seconds per token. As a result, most end-side LLM systems prefer smaller models, e.g., run 7B models on phones and browsers [MLC, 2023, Lugaresi et al., 2019] and 3.8B models on an Android device [Ghorbani, 2024].

That's not too bad, users have many devices, e.g., laptops, desktops, phones, and tablets. Some PCs have high-end GPUs like the NVIDIA 20/30/40 series, and Mac M-series have Apple Metal GPUs. By pooling the computing power and memory of home devices, it's possible to run larger models. In pipeline parallelism, the model is split into segments and assigned to devices. Exo [Exo, 2024] distributes model layers based on memory size, devices with more RAM/VRAM handle more layers. While this prevents OOM, it can slow speed if high-memory devices have weak CPUs/GPUs. This raises a question: *How should model layers be assigned?* A simple alternative is to assign model layers based on computing power, with stronger devices handling more layers, but also with high OOM risk. Ye et al. [2024] offers an idea: first, assign model layers based on computing power,

then migrate layers from OOM devices to those with free memory. However, this also slows down inference, as weaker CPUs handle more layers.

So new questions arise: Can stronger devices handle these overloaded layers instead? If we require the cluster memory to meet the model's needs [Ye et al., 2024, Exo, 2024, Tadych, 2024, Zhang et al., 2025, Lee et al., 2024, Zhao et al., 2023, Zhang et al., 2024], the answer is no, as stronger devices have reached their OOM limits. But what if we relax this requirement? For example, we can free up processed layers on stronger devices and load the extra layers to continue computation. With a fast SSD, this could be more efficient than using weak-CPU devices. This adds complexity: whose disks are faster than weak CPUs? And how many extra layers can be migrated to maximize inference speed? This is tricky because disk I/O introduces a new bottleneck: disk loading latency, which depends on data size to be (re-)load and disk read speed. These two factors vary with hardware and OS differences. For example, macOS with Metal reclaims memory more aggressively than Linux, causing more reloads, while Linux optimizes sequential reads, making reloading faster than on macOS. This heterogeneity makes disk loading latency hard to estimate.

This paper introduces a device profiler to capture system heterogeneity across computation, memory, disk, communication, and OS. We mathematically model token latency from these factors and propose Halda, an efficient algorithm to solve this layer-to-device assignment (LDA) problem. Halda provides an optimal workload distribution across CPU and GPU (if available) for each device. It determines how many model layers each device should handle and how to allocate them between CPU and GPU. To hide disk loading, we design piped-ring parallelism, where devices form a ring and pass their results to the next for subsequent processing. Unlike existing systems, in our design, devices can complete a token with multiple rounds, and with prefetching, we overlap disk loading with other devices to hide latency. We call it *prima.cpp*, as it uses piped-ring parallelism and builts on llama.cpp [Gerganov, 2024]. The main contributions are summarized as follows:

- We propose prima.cpp, a distributed inference system designed for low-resource home clusters. It uses mmap to lazily load model weights, and piped-ring parallelism with prefetching to hide disk latency. It prevents OOM and reduces token latency.
- We model the LDA problem and develop a device profiler to capture the system heterogeneity. It minimizes token latency by modeling delays in computation, memory access, disk loading, and communication, while optimizing the use of RAM and VRAM.
- We propose the Halda algorithm to solve the LDA problem. Halda breaks the NP-hard problem into a set of simple ILPs, so we can find the optimal solution in polynomial time.
- We implement prima.cpp with 20K LoC modifications. Evaluation on a real home cluster shows that prima.cpp is 15× faster than llama.cpp on 70B models, with memory pressure below 6% per device. It also surpasses distributed alternatives like exo [Exo, 2024] and dllama [Tadych, 2024] in both speed and memory efficiency across all 7B-72B models.

In our experiments, a small, heterogeneous, and budget-friendly home cluster (2 laptops, 1 desktop, 1 phone) was used. Our prima.cpp achieves $\sim\!600$ milliseconds per token and a time-to-first-token (TTFT) below 2 seconds for a 70B model, making it accessible for voice chat apps like home Siri. For 45B-70B models, the speed matches those of audiobook apps. It now supports hot models including Llama 3, Qwen 2.5, QwQ, and DeepSeek R1 (distilled versions).

2 Related Work

On-device LLM systems. Most of these systems are limited to small models. MLC-LLM [MLC, 2023] and MediaPipe [Lugaresi et al., 2019] bring 7B models to mobile phones and browsers. PocketPal AI [Ghorbani, 2024], which runs on Android using llama.cpp [Gerganov, 2024], supports models up to 3.8B. Some efforts push for larger models, like AirLLM [Li, 2023], which loads only needed layers to save memory but at the cost of speed. Others turn to high-end hardware, such as the Apple M2 Ultra (192 GiB RAM) for a 65B model or kTransformers [kvcache ai, 2025] (382 GiB RAM) for a 671B model (~75 GiB for 70B). These setups (large RAM, advanced CPUs with specific instruction sets) go far beyond common home devices and are inaccessible to most users.

Distributed LLM systems at the user edge. Distributed systems break the limits of a single device. We focus on edge LLM inference and categorize recent efforts into tensor and pipeline parallelism.

Table 1: Comparison of distributed LLM inference systems at the edge.

	Type ¹	Backends ²	Mem ³	Quantization	Mem Pressure ⁴	Speed	Heterogeneity ⁵
dllama [Tadych, 2024]	TP	CPU	RAM	Q4	Critical	Slow	Х
Zhang et al. [2025]	TP	CPU	RAM	FP32	Critical	Slow	✓
Hepti [Lee et al., 2024]	TP	CPU	RAM	FP32	Critical	Slow	✓
Galaxy [Ye et al., 2024]	TP+SP	CPU / GPU	RAM / VRAM	FP32	Critical	Slow	✓
TPI-LLM [Li et al., 2024]	TP	CPU	RAM	FP32	Medium	Slow	X
exo [Exo, 2024]	PP	CPU / GPU	RAM / VRAM	Q4+FP32	Critical	Slow	<u> ✓</u>
LinguaLinked [Zhao et al., 2023]	PP	CPU	RAM	Q8 / FP32	Critical	Slow	✓
EdgeShard [Zhang et al., 2024]	PP	CPU / GPU	RAM / VRAM	FP32	Critical	Fast	✓
prima.cpp (ours)	PRP	CPU & GPU	RAM & VRAM	Q4 / IQ1	Low	Fast	✓

¹ "TP" is tensor parallelism, "PP" is pipeline parallelism, "SP" is sequential parallelism, "PRP" is piped-ring parallelism.

- *Tensor Parallelism*. Tensor parallelism splits weight tensors (e.g., attention heads and FFNs) across devices to share the load [Shoeybi et al., 2019]. To reduce all-reduce costs, dllama [Tadych, 2024] uses USB4 and Thunderbolt 5 for fast connections, and [Zhang et al., 2025] uses wireless analog superposition to perform all-reduce over the air. Due to device heterogeneity, Hepti [Lee et al., 2024] optimizes workload partitioning with three slicing strategies for different memory budgets, and Galaxy [Ye et al., 2024] prioritizes compute power first, then memory, to maximize speed and avoid OOM. These systems load the full model into cluster memory. With limited total memory, only small models can run. TPI-LLM [Li et al., 2024] loads model layers on demand and hides disk loading with prefetching. This allows poor devices with only 4 GiB RAM to run a 70B model, but it's still slow.
- *Pipeline Parallelism*. Due to Wi-Fi's high latency, pipeline parallelism is better for home clusters as its fewer P2P communication. Exo [2024], Zhao et al. [2023], Zhang et al. [2024] split the model into segments and assign them to devices based on memory, compute, and network conditions. Each device computes its segment and passes the result to the next, until the last device outputs the next token. Exo [2024] partitions model segments based on memory ratio; LinguaLinked [Zhao et al., 2023] uses linear optimization to solve the device assignment problem; and EdgeShard [Zhang et al., 2024] uses dynamic programming. Some of them are efficient, but rely on specialized devices like Jetson AGX/Nano, which are not common in household.

These distributed systems, except TPI-LLM, have severe limitation: they require cluster memory to meet the model's needs. When cluster memory is limited, only small models can run. However, users have few devices with less mem_available (could be much less than mem_total), and allocating too much for a LLM app can freeze devices and degrade user experience. Additionally, as shown in Table 1, some systems lack GPU support, while others don't use CPU offloading. This restricts devices to either RAM or VRAM and further limiting cluster memory. Most also skip quantization and reside model weights in memory, making OOM highly likely for larger models.

Prima.cpp addresses these issues with four key features: (a) support *disk offloading* to allow cluster memory to be less than the model's needs. While this causes disk latency, prima.cpp minimizes it through optimal workload distribution and piped-ring parallelism with prefetching; (b) support *GPU & CPU offloading* to combine RAM and VRAM into cluster memory, with optimal workload assignment to alleviate CPU slowdowns; (c) support *quantization* like Q4K (used by default) and IQ1 to cut memory needs; and (d) use *mmap* to cache model weights, so the OS can free them as needed.

3 Prima.cpp: Parallel Architecture and Scheduler Design

We first introduce the piped-ring parallelism with prefetching adopted by prima.cpp in Section 3.1. Section 3.2 then mathematically models the layer-to-device assignment problem and Section 3.3 proposes a polynomial-time algorithm to solve it, so that the inference latency is minimized.

² Backends simultaneously used on one device. "CPU / GPU" indicates only one backend is used at a time on a device, and if a GPU is available, only the GPU will be used. "CPU & GPU" indicates that both CPU and GPU are used if a GPU is available.

Memory simultaneously used on one device. The definitions of "RAM / VRAM" and "RAM & VRAM" are similar to the above.

⁴ Memory pressure when the cluster is just enough to run the model. Low: memory can be reclaimed normally. Medium: Potential for page swapping. Critical: High pressure, likely to freeze or be killed.

⁵ Whether be optimized for heterogeneous devices.

3.1 Piped-ring Parallelism with Prefetching

Pipeline parallelism is common in distributed inference, where the model is split into layer-based segments and assigned to a chain of devices. Each device computes its segment and passes the result to the next, this process continues until the last device outputs the next token. Exo [2024], Zhang et al. [2024], Zhao et al. [2023] take this a step further by linking the last device back to the first, forming a ring. This allows input and output to be processed on the head device, providing enhanced privacy.

Pipeline parallelism works well for batch inference when resources are abundant, but it's not a good fit for resource-limited home clusters due to: (a) home users have few devices with low specs (see Table 2); (b) LLM apps should use mem_available, not mem_total, or other apps will slow down, degrading user experience; and (c) batch inference demands more RAM. These limitations make it hard to gather enough devices to meet the memory needs of larger models. In this study, we focus on single-request inference to tackle the immediate challenge of fast inferencing 70B models in resource-scarce clusters. Batch inference can be developed once this foundation is solid.

Given this setup, the drawback of pipeline becomes apparent: when a device is running, it loads required layers (due to limited mem_available, we swap layers from disk instead of keeping a full model) and computes them, meanwhile, others sit idle. To improve this, we use prefetching.

Pipeline parallelism with prefetching. Prefetching loads layers in advance, ensuring that when computing starts, the required layers are already (or partially) in place. This overlaps disk loading with ongoing operations on other devices, so part of the latency can be hidden. We use mmap advice to achieve this: once a device finishes computing, we mark upcoming

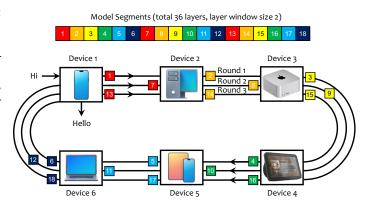


Figure 1: Piped-ring parallelism. In this case, 6 devices handle a 36-layer model. With a layer window size of 2, the model is splitted into 18 segments, which are assigned to 6 devices in a ring order, so each device needs 3 rounds to predict one token.

layers as WILLNEED to advise the OS to prefetch them in the background when the load permits, so it does not block the main process. It continues even after computing starts until page faults occur.

This vanilla approach has a flaw: if the current device has a high disk speed, or other devices run for a long time, prefetching may load too many layers, exceeding its available memory. As a result, later-loaded layers will overwrite those prefetched earlier. We call this effect "prefetch-release" (see Appendix A.1), which results in all layers being loaded twice, adding unnecessary disk overhead without any benefit from prefetching. To address this, we propose the piped-ring parallelism.

Piped-ring parallelism with prefetching. In piped-ring parallelism, devices are connected in a ring structure, but unlike Zhang et al. [2024], Exo [2024], Zhao et al. [2023], we can run multiple rounds to predict one token. We define layer window size as the number of model layers a device should handle in each round. For example, in Figure 1, each device takes 3 rounds to predict one token. In each round, device 1 should handle 2 model layers, so its window size is 2. Layer window sizes vary by device capability—stronger devices have larger windows. However, finding the optimal window size setup is complex. In section 3.2, we mathematically model this problem, followed by an efficient algorithm in section 3.3 to solve it. By setting the layer window size small, we ensure the model layers stays within memory limits, avoiding "prefetch-release" during prefetching. We visualize this magic and explain how disk loading is overlapped in Appendix A.2.

3.2 Layer-to-device Assignment Problem

An implicit assumption was made in section 3.1: all devices, even heterogeneous ones, use the same layer window size, which is poor practice. High-end devices should handle more layers than

low-end ones. As shown in Figure 6(f), if we offload some layers from devices 2, 3 to devices 1, 4, "bubbles" (device idle time) can be smaller, and more rounds can be executed within the same time. So here comes the question: how should the model layers be distributed among resource-limited and heterogeneous devices, and how many layers should run on the GPU (if any)? To answer this question, we define the layer-to-device assignment (LDA) problem as follows.

Definition 1 (Layer-to-Device Assignment Problem, LDA). Assume there are M devices, w_m is the layer window size on device d_m and n_m is the number of GPU layers within w_m . Let the decision variables be $\mathbf{w}^T = [w_1, w_2, \cdots, w_M]$ and $\mathbf{n}^T = [n_1, n_2, \cdots, n_M]$. Our objective is to find the optimal \mathbf{w} and \mathbf{n} that minimizes the token latency:

$$\min_{\boldsymbol{w},\boldsymbol{n}} L \cdot \frac{\boldsymbol{a}^{\mathrm{T}} \cdot \boldsymbol{w} + \boldsymbol{b}^{\mathrm{T}} \cdot \boldsymbol{n} + \boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{c}}{\boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{w}} + \kappa, \tag{1}$$

s.t.
$$w_m \in \mathbb{Z}_{>0}, n_m \in \mathbb{Z}_{>0}, n_m \le w_m \le L,$$
 (2)

$$L - k(\boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{w}) = 0, k \in \mathbb{Z}_{>0}, \tag{3}$$

$$P_w \cdot w' + P_n \cdot n' + e^{\mathrm{T}} \cdot w \cdot z < 0, \tag{4}$$

$$-P_n^{gpu} \cdot z^{gpu} \cdot e^{\mathrm{T}} \cdot w + P_n^{gpu} \cdot n \le 0.$$
 (5)

where L is the number of model layers; a, b, c are constant vectors determined by computation latency, memory access latency, disk loading latency, and communication latency on each device; κ is a constant coefficient; k is the number of rounds to predict one token; \mathbf{w}' and \mathbf{n}' are the extended vectors of \mathbf{w} and \mathbf{n} ; \mathbf{z} and \mathbf{z}^{gpu} are constraint vectors for RAM and VRAM; $\mathbf{P}_w, \mathbf{P}_n, \mathbf{P}_n^{\text{gpu}}$ are diagonal matricies that activate or deactivate the decision variables.

For the derivation of this definition, please see Appendix A.3. Simply put, constraint (2) ensures that each device handles at least one layer, with the GPU layers not exceeding the total. Constraint (3) enforces all devices are assigned an equal number of windows and all the windows are filled. This is not mandatory in our implementation, but can simplify the problem model. Constraints (4) and (5) ensure that RAM and VRAM usage stay within limits. Table 5 summarizes the key symbols.

To construct $a, b, c, \kappa, w', n', z, P_w, P_n$, we should categorize devices into one of the following cases: Case 1 (set \mathcal{M}_1): macOS with Metal disabled and insufficient RAM; Case 2 (set \mathcal{M}_2): macOS with Metal enabled and insufficient RAM; Case 3 (set \mathcal{M}_3): Linux and Android with insufficient RAM; Case 4 (set \mathcal{M}_4): OS with sufficient RAM or low disk speed. For example, for cases 1–3 where devices should overload, RAM usage must stay above a lower bound, but for case 4 where overloading is not allowed, RAM usage must stay below a upper bound.

Whether a device is overloaded depends on w and n, e.g., a large $w_m - n_m$ will overload RAM. However, we cannot determine a device's case before solving the problem, and without knowing the case, we cannot solve the problem. This traps us in a circular dependency. Besides, since $W = e^T \cdot w$ appears in both the objective and constraints, this LDA model is not a standard integer linear fractional programming (ILFP) problem, making the problem more challenging.

3.3 Our Halda Scheduler

To solve the LDA problem, our core ideas include: (i) transform the NP-hard original problem into a set of standard integer linear programming (ILP) problems by enumerating over all possible k, and (ii) search for optimal set assignments $\mathcal{M}_1 \sim \mathcal{M}_4$ by iterative optimization.

Transform into standard ILP problems. Given that the number of layers L in typical LLMs is less than 100, the integer k has a limited range of values — at most 11 valid factors for any $k \le 100$. By enumerating over these factors, we can treat k and W as constants, then the problem becomes:

min
$$k(\boldsymbol{a}^{\mathrm{T}} \cdot \boldsymbol{w} + \boldsymbol{b}^{\mathrm{T}} \cdot \boldsymbol{n} + \boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{c}) + \kappa,$$
 (6)

s.t.
$$w_m \in \mathbb{Z}_{>0}, n_m \in \mathbb{Z}_{>0}, n_m \le w_m \le L,$$
 (7)

$$\boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{w} = W,\tag{8}$$

$$P_w \cdot w' + P_n \cdot n' + Wz < 0, \tag{9}$$

$$P_n^{\text{gpu}} \cdot \boldsymbol{n} - W P_n^{\text{gpu}} \cdot \boldsymbol{z}^{\text{gpu}} \le 0. \tag{10}$$

Algorithm 1: <u>H</u>eterogeneity-<u>A</u>ware <u>L</u>ayer-to-<u>D</u>evice <u>A</u>llocation (HALDA)

```
1 Initialize layer windows w proportionally to devices' memory budgets and the number of GPU
    layers n \leftarrow 0;
2 Calculate platform-specific coefficients \alpha_m, \beta_m, \xi_m for each device m;
3 Calculate valid factors \mathcal{K}_L of L (excluding L);
4 while true do
        Calculate W = e^{\mathrm{T}} \cdot \boldsymbol{w} and k = L/W;
        Re-assign devices to sets \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4 based on the latest w, n, k and \mathcal{M}_4^{\text{force}};
6
        if the assignment sets \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4 remain unchanged then
        Calculate the objective coefficients a, b, c, \kappa, the RAM upper bound z, and the
          VRAM/shared memory upper bound z^{gpu} according to the updated assignment sets;
        foreach k \in \mathcal{K}_L do
10
             Solve the ILP problem (1-5) with fixing k using a ILP solver;
11
            Update best solution (w^*, n^*) if the current objective is smaller;
12
        if any device has free VRAM but another device is overloaded then
13
             Force the device m_s from \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 with lowest disk read speed into \mathcal{M}_4^{\text{force}};
14
            continue;
15
        Update w \leftarrow w^* and n \leftarrow n^*;
16
17 return w^*, n^*;
```

Hence, for each fixed k, the objective and constraints boil down to linear functions/inequalities, and the problem becomes an ILP. Then, we can run a standard ILP solver (e.g., HiGHS [Huangfu and Hall, 2018]) to obtain the optimum w, n.

Iterative optimization on set assignments. The problem remains unsolvable because the set assignments $\mathcal{M}_1 \sim \mathcal{M}_4$ are unknown. To resolve the circular dependency, we adopt an iterative optimization approach. Initially, \boldsymbol{w} is set proportionally based on the memory budget (d_m^{avail}) for macOS without Metal and Linux, $d_{m,\text{metal}}^{\text{avail}}$ for macOS with Metal, and $d_m^{\text{avail}} + d_m^{\text{swapout}}$ for Android), and \boldsymbol{n} is initialized to 0. This gives an initial division of devices into sets $\mathcal{M}_1 \sim \mathcal{M}_4$. Then, we solve the ILP problems to update \boldsymbol{w} and \boldsymbol{n} , re-assign the devices to their new sets, and repeat this process until the set assignment remain unchanged.

This approach still has defects. The ILP model only constrains the upper bound on the number of layers that can be allocated to the GPU n. In certain cases, such as when the total number of layers L is small and when the initial set assignment falls into a local optimum, the number of layers assigned to the GPU may be insufficient. For example, if a device m is assigned $w_m = 30$ layers and its GPU can handle up to 20 layers, but due to memory constraints in $\mathcal{M}_1 \sim \mathcal{M}_3$, only 10 layers go to the GPU, leading to underutilization of GPU resources. This issue is caused by the initial set assignment, which allocates too many devices to $\mathcal{M}_1 \sim \mathcal{M}_3$, leaving an insufficient number of layers for GPUs. Therefore, in Algorithm 1, we apply a calibration step: if a GPU is underutilized (its VRAM is not full) while another device is overloaded (i.e., its VRAM is full and has offloaded some layers to the CPU, or it has exceeded the RAM limit), we force the device $m \in \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$ with lagging disk into \mathcal{M}_4 , re-construct the sets $\mathcal{M}_1 \sim \mathcal{M}_4$, and re-solve the problem. This way, we can converge to the optimal set assignment and then determine the optimal setup for w, n and k.

Complexity Analysis. The main loop has two key tasks: set assignment and solving the LDA problem. It runs for T iterations, which in the worst case is T = O(M). Set assignment also takes O(M) time. For LDA, we solve an ILP for each valid factor of L, with $K = O(\log L)$ factors in total. Our ILP is usually tiny and sparse, taking $O((2M)^{3.5})$ to solve it. Thus, the total time complexity is $O(T(M + K(2M)^{3.5}))$, which can be solved in polynomial time.

4 Experiments

We built the experimental platform on common home devices detailed in Table 2. On Mate40Pro (HarmonyOS) and Honor Pad (Android), we run prima.cpp inside a Termux-simulated Linux. These

Table 2: Configurations on experimental device

	D1	D2	D3	D4	D5	D6
Device	Mac M1	Laptop	Desktop	Mate40Pro	Honor Pad	Mac Air
OS	MacOS (UMA)	Linux	Linux	Linux (on HarmonyOS)	Linux (on Android)	MacOS (NUMA)
CPU	Apple M1	Intel i9	Intel i9	Kirin 9000	Dimensity 8100	Intel i5
CPU Cores	8	8	16	8	8	4
RAM (available)	2.4 GiB	4.1 GiB	9.7 GiB	1.9 GiB	5.1 GiB	6.8 GiB
Disk Read Speed	0.72 GB/s	2.98 GB/s	3.17 GB/s	1.37 GB/s	2.00 GB/s	0.39 GB/s
GPU Type	Apple Metal	3070	2080TI	-	-	-
VRAM (available)	-	8 GiB	11 GiB	-	-	-

devices were connected to a local Wi-Fi router. By default, we used 4 devices (D1–D4) with a total available RAM+VRAM of 37 GiB (not enough for a Q4K-quantized 70B model). We evaluated Llama models from 8B to 70B (in Q4K format) in terms of token latency, TTFT, and memory pressure, and compared with llama.cpp [Gerganov, 2024], exo [Exo, 2024], and dllama [Tadych, 2024]. The results showed the significant advantages of prima.cpp in both speed and memory pressure.

4.1 Faster Inference on Larger Models

As llama.cpp is an on-device system, we ran it on the most powerful desktop. Meanwhile, we ran exo on devices D1-D3, as they are equipped with Apple/NVIDIA GPUs, and running exo on D4 require root access, which is not permitted. Instead, dllama and prima.cpp used devices D1-D4. Table 3 presents the token latency and TTFT across different model sizes. As of now, exo and dllama do not support Llama models from 14B to 65B, so we put a "-" in the table.

As illustrated in Figure 9a (in Appendix A.6), for smaller models (\leq 14B), the 11 GiB VRAM on the desktop is sufficient for full GPU inference, at this time, llama.cpp is the best choice. However, at 30B, VRAM runs out, forcing layers onto the CPU, thus inference speed becomes limited. As the model size increases to 45B, even the 9.7 GiB of available RAM is exhausted. Since llama.cpp uses mmap to lazily load model weights, the OS frees inactive mmap-ed pages and reloads them as needed, which introduces disk loading latency. At this stage, only a few pages are released, so efficiency loss is small. However, at 60B, more active mmap-ed pages are labeled as inactive earlier and then released, leading to a sharp increase in token latency and TTFT. This indicates that llama.cpp is not well-suited for running larger models on home devices.

For exo and dllama, despite only data for the 8B model is available, their limitations are already evident. Llama 3-8B (Q4K) requires just 5.3 GiB of VRAM, so all 32 layers could fit entirely on D3-GPU. However, exo allocates model layers proportional to each device's memory. D1 (8 GiB RAM), D2 (8 GiB VRAM), and D3 (11 GiB VRAM) are assigned 9, 10, 13 layers, respectively. While D1 has an Apple Metal GPU, its efficiency is much lower than D3-GPU, making it an efficiency bottleneck. For dllama, inference is performed using tensor parallelism, and the workload is evenly distributed across devices. This forces high- and low-performance devices to handle the same load and causes blocking during all-reduce, leading to limited efficiency.

Prima.cpp addresses these limitations effectively, which uses piped-ring parallelism with prefetching to collaborate multiple devices. Unlike existing systems using solely CPU or GPU, prima.cpp uses both CPU and GPU on each device. It also adapts better to heterogeneous home devices by analyzing CPU/GPU power, available RAM/VRAM, memory management behavior, and disk speed to assign workloads, with the goal to minimize token latency. As shown in Table 3, prima.cpp has significantly lower token latency and TTFT than exo and dllama across all model sizes and outperforms llama.cpp for models larger than 30B. Specifically, compared to llama.cpp, prima.cpp improves token latency by up to 17× and TTFT by up to 8×. Against exo and dllama, it speeds up token latency by 5–8× and TTFT by 12–24×. It also effectively prevents OOM errors.

Appendix A.4 shows the results on more hot models, including Qwen 2.5, QwQ and DeepSeek R1. In current implementation, each device is assigned at least one model layer, for example, leading to a 1:1:29:1 split for Llama 3-8B. This restriction is unnecessary and we will remove it in future updates. Then, we will have a 0:0:32:0 split and idle devices removed, making llama.cpp a special case of prima.cpp when serving small models. Appendix A.5 explains why we use D1-D4 instead of D1-D6:

Table 3. Token rateley and 1111 (in minisceond/token) for nama.epp, exo, anama, and prima.epp.										
Model	llama.cpp		exo		dllama		prima.cpp (w/o halda)	prima.cpp (w/o prefetch)	prima.cpp	
1,10001	Latency	TTFT	Latency	TTFT	Latency	TTFT	Latency	Latency	Latency	TTFT
Llama 3-8B	15	18	263	960	459	1845	78	53	54	78
Llama 3-14B	20	25	-	-	-	-	131	62	65	134
Llama 1-30B	202	611	-	-	-	-	258	79	72	214
Llama 3-45B	328	712	-	-	-	-	409	263	233	440
Llama 3-60B	7965	8350	_	-	_	-	7053	532	468	990
Llama 1-65B	8807	9662	-	-	-	-	12253	688	569	1770
Llama 3-70B	10120	10806	OOM	OOM	OOM	OOM	20848	755	674	1793

Table 3: Token latency and TTFT (in millisecond/token) for llama.cpp, exo, dllama, and prima.cpp.

more is not always better, and the cluster memory does not need to match the model's needs. It also shows how prima.cpp can pick a subset of devices to build a best-performing cluster.

4.2 Ablation Study on Prefetching, Halda and Piped-ring Parallelism

Table 3 also includes an ablation study on Halda and prefetching. For a fair comparison, we set prima.cpp (w/o halda) to use exo's strategy - assigning model layers proportional to device RAM/VRAM, but improves it by using available RAM/VRAM instead of total and offloading overloaded layers from GPU to CPU to prevent OOM. Instead, prima.cpp uses our Halda strategy.

Prefetching. To test prefetching, we disable it in prima.cpp (w/o prefetch) and enable it in prima.cpp. Prefetching has no effect on small models, as they fit entirely in RAM/VRAM without triggering memory reclamation. However, for larger models, memory reclamation triggers frequent page faults as early layers have been released by the OS, but computation starts with them. To mitigate this, prima.cpp prefetches released layers immediately after computation, ensuring they are already in memory before the next round of computation begins to reduce page faults. This overlaps disk loading with other devices' operations, thus lowering token latency by 9%~17%.

Halda. A proper layer assignment is crucial for fast inference. For small models, RAM/VRAM on D1-D4 is sufficient, so prima.cpp (w/o halda) and prima.cpp perform similarly. However, prima.cpp (w/o halda) assigns more layers to weaker CPUs/GPUs, while prima.cpp prioritizes powerful GPUs, the latter achieves lower latency. For larger models, prima.cpp (w/o halda) overloads non-GPU devices, triggering memory reclamation and reloading, causing a latency spike. In contrast, prima.cpp avoids memory overload on slow-disk devices through optimized layer assignment, achieving up to 30× speedup for 70B-scale models.

Piped-ring Parallelism. To evaluate it in isolation, we built a CPU cluster with 4 Linux devices, each having 8 cores, 8 GiB RAM, and SSD of 2 GB/s. We tested models from 8B to

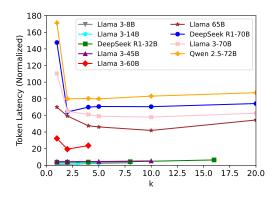


Figure 2: Normalized token latency over k.

72B, and assigned model layers evenly across devices. For example, for 65B-72B models with 80 layers, at k=1 (k denotes the rounds to predict one token), layers were split 20:20:20:20; at k=2, 10:10:10:10, and so on. The results in Figure 2 show that when total available RAM is insufficient (>60B), piped-ring parallelism reduces latency by nearly half. However, at k=1, it offers no benefit due to the "prefetch-release" effect. Thus, the layer window size should be set smaller, and piped-ring parallelism becomes essential (see Appendices A.1 and A.2). However, when memory is sufficient (<45B), no disk loading occurs, but increasing k slightly raises latency due to overhead from model fragmentation and reduced computational parallelism. To sum up, piped-ring parallelism excels under high disk-loading conditions, but k=1 is better when memory is sufficient.

Table 4: Memory	pressure for llama.cp	n exo dllama	and prima cpp	on each device
Table T. Michiel y	pressure for mama.cp	p, cao, unama,	and prima.cpp	on cach acvice.

Model	llama.cpp	llama.cpp exo			dllama				prima.cpp			
Model	D3	D1	D2	D3	D1	D2	D3	D4	D1	D2	D3	D4
Llama 3-8B	2.0%	20.0%	51.3%	42.5%	13.5%	12.8%	55.8%	12.8%	5.3%	5.4%	2.7%	≤1.0%
Llama 3-14B	2.5%	-	-	-	-	-	-	-	5.3%	4.3%	2.2%	$\leq 1.0\%$
Llama 1-30B	8.0%	-	-	-	-	-	-	-	3.0%	5.7%	2.9%	≤1.0%
Llama 3-45B	3.9%	-	-	-	-	-	-	-	4.9%	$\leq 1.0\%$	6.0%	$\leq 1.0\%$
Llama 3-60B	5.5%	-	-	-	-	-	-	-	6.3%	4.7%	4.7%	$\leq 1.0\%$
Llama 1-65B	15.6%	-	-	-	-	-	-	-	3.9%	$\leq 1.0\%$	$\leq 1.0\%$	$\leq 1.0\%$
Llama 3-70B	6.0%	OOM	OOM	OOM	OOM	OOM	OOM	OOM	4.7%	4.8%	4.8%	≤1.0%

4.3 Low Memory Pressure for Better Experience

Memory pressure is crucial for user experience because high pressure can slow down apps or even crash the device. Imagine if an LLM app will cause home devices to freeze, would you use it? Clearly, no. However, this is ignored by existing systems. Appendix A.6 shows devices' memory footprint to illustrate the workload distribution and explain why prima.cpp is faster. However, a higher value does not indicate higher pressure, as reclaimable memory like page cache is included but can be freed instantly by the OS. To better measure memory pressure, we define it as the reduction in mem_available during runtime relative to mem_total. For example, 2 GiB decrease in mem_available on an 8 GiB device results in 25% pressure. As mem_available includes free and reclaimable pages, its reduction contains only non-reclaimable pages, so it is memory pressure.

Table 4 shows the memory pressure caused by llama.cpp, exo, dllama, and prima.cpp on each device. Prima.cpp and llama.cpp have low memory pressure, using only a small amount of mem_used for key-value cache and compute buffer. Since they load model weights via mmap, the majority of resident memory is held in the page cache and can be instantly released by the OS without affecting other apps. In contrast, exo and dllama keep model weights in mem_used, causing high memory pressure. This forces the OS to free inactive pages, compress memory, swap data to disk, potentially slowing down apps, leading to system lag, or OOM for larger models. Overall, exo and dllama prioritize resources for the LLM app at the cost of other apps; prima.cpp and llama.cpp prioritize user experience, keeping low memory pressure and thus better suited for user devices. Therefore, prima.cpp is the best choice for running larger models on home devices (>30B in our case).

5 Conclusion

This paper introduces prima.cpp, a distributed inference system to run 70B-scale LLMs on everyday home devices. The core design is a piped-ring parallelism with prefetching and a scheduler to split model layers to heterogeneous devices. Prima.cpp uses mmap to manage model weights, preventing OOM for any model size but introducing disk latency. To hide this latency, prima.cpp employs prefetching and piped-ring parallelism, which overlaps disk loading and avoiding the "prefetch-release" effect. To further speed up inference, we mathematically model token latency by considering the heterogeneous nature of home devices. We propose Halda to solve this NP-hard LDA problem, which assigns model layers to device CPUs and GPUs and minimizes token latency. On a small, heterogeneous, and budget-friendly home cluster, prima.cpp outperforms llama.cpp, exo, and dllama in speed, model size, and memory pressure. This work requires lower-end hardwares and has a better support for cross-platform deployment. With prima.cpp, we are able to collaborate phones, tablets, laptops, desktops etc. to bring 70B LLMs (Llama 3, DeepSeek R1, Qwen 2.5, QwQ) to home Siri.

Limitations: (a) Limited device types and quantity restrict our exploration of diverse home clusters. Readers are welcome to share their results on our Github repo. (b) Though much faster than other on-device systems, 70B models remain much slower than those on the cloud. Future work will integrate IQ1/Q4K for higher efficiency. (c) In low-RAM clusters without SSDs or GPUs, larger models will be extremely slow. (d) Token latency is heavily affected by memory competition. If there are other processes, prima.cpp slows down to free RAM for them and speeds up when they stop, so we use a stable value from multiple runs instead of an error bound. (e) Prima.cpp unlocks larger-scale open-source LLMs on user devices, where malicious content may not be filtered. The open-source community should enhance oversight of their models to prevent misuse.

References

- DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.
- Exo. exo: Run your own ai cluster at home with everyday devices. https://github.com/exo-explore/exo, 2024.
- Georgi Gerganov. llama.cpp: Llm inference in c/c++. https://github.com/ggerganov/llama.cpp, 2024.
- Asghar Ghorbani. Pocketpal ai: An app that brings language models directly to your phone. https://github.com/a-ghorbani/pocketpal-ai, 2024.
- Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.
- kvcache ai. ktransformers: A flexible framework for experiencing cutting-edge llm inference optimizations, 2025. URL https://github.com/kvcache-ai/ktransformers.
- Juhyeon Lee, Insung Bahk, Hoseung Kim, Sinjin Jeong, Suyeon Lee, and Donghyun Min. An autonomous parallelization of transformer model inference on heterogeneous edge devices. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 50–61, 2024.
- Gavin Li. Airllm: scaling large language models on low-end commodity computers, 2023. URL https://github.com/lyogavin/airllm/.
- Zonghang Li, Wenjiao Feng, Mohsen Guizani, and Hongfang Yu. Tpi-llm: Serving 70b-scale llms efficiently on low-resource edge devices. *arXiv preprint arXiv:2410.00531*, 2024.
- Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, et al. Mediapipe: A framework for perceiving and processing reality. In 3rd Workshop on Computer Vision for AR/VR at CVPR, 2019.
- MLC. Mlc-llm: Universal llm deployment engine with ml compilation. https://github.com/mlc-ai/mlc-llm, 2023.
- Mohammad Shoeybi, Mostofa Patwary, Raul Puri, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- Bartłomiej Tadych. Distributed llama. https://github.com/b4rtaz/distributed-llama, 2024.
- Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 1001–1010. IEEE, 2024.
- Kai Zhang, Hengtao He, Shenghui Song, Jun Zhang, and Khaled B Letaief. Distributed on-device llm inference with over-the-air computation. *arXiv* preprint arXiv:2502.12559, 2025.
- Mingjin Zhang, Jiannong Cao, Xiaoming Shen, et al. Edgeshard: Efficient llm inference via collaborative edge computing. *arXiv preprint arXiv:2405.14371*, 2024.
- Junchen Zhao, Yurun Song, Simeng Liu, Ian G Harris, and Sangeetha Abdu Jyothi. Lingualinked: A distributed large language model inference system for mobile devices. *arXiv preprint arXiv:2312.00388*, 2023.

A Appendix

A.1 Prefetch-release Effect

As illustrated in Figure 3. Before computation starts, the OS prefetches 3 model layers to the available memory limit. However, it does not stop and continues to load the 4th layer, causing the 1st layer to be released. This "prefetch-release" cycle repeats, so by the end, the last 3 layers are in memory, while the first 3 are not. Then, when computation begins, the 1st layer, which is not in memory, triggers a page fault, prompting the OS to reload it and the 4th layer to be released. Finally, all layers are loaded twice, adding unnecessary disk overhead without any benefit from prefetching.

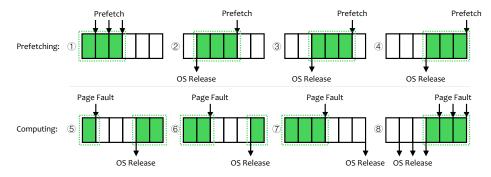


Figure 3: Illustration of model layers loaded into memory in pipeline parallelism with prefetching. In this case, the device handles 6 model layers but its available memory can only hold 3. The green blocks show the layers loaded into memory, while white blocks indicate those not yet loaded.

A.2 How Piped-ring Parallelism Solve the Prefetch-release Effect?

To illustrate, Figure 4 considers a fast-disk device where prefetching is fast enough to complete before computation begins. In this case, with a fast disk and a layer window size of 2, ① prefetching is fast enough to load 2 layers before computation begins, then ② computation runs without page faults. Then, ③ the next round of 2 layers are prefetched, replacing the used ones. Steps ②-⑦ repeat until inference is complete. Prefetching overlaps with other devices' operations, so its latency does not contribute to inference time. Here, with no page faults, total latency comes only from computation. In other words, disk loading latency is fully overlapped.

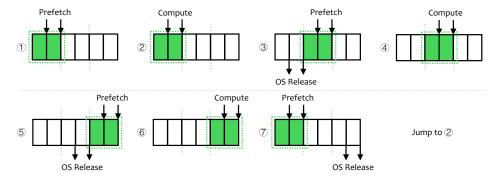


Figure 4: Illustration of model layers loaded into memory in piped-ring parallelism with fast disk.

Figure 5 shows a common case with a slow disk. In this case, ① prefetching loads only one layer then ② computation begins, ③ a page fault triggers when reaching the 2nd layer, blocking until it loads. After computation, ④ the device prefetches the next round of 2 layers, but only one layer loads due to the slow disk and releasing the oldest layer. Then, ⑤ the next round of computation begins, and ⑥ at the 6th layer, another page fault occurs. This cycle of "loading (prefetch) - computing - loading (page fault) - computing" repeats until inference completes. While page fault-induced loading blocks computation, prefetching helps overlap some latency.

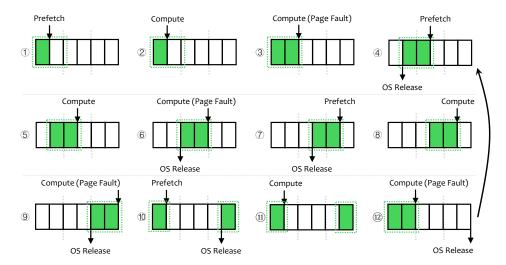


Figure 5: Illustration of model layers loaded into memory in piped-ring parallelism with slow disk.

We use a timeline to visualize this overlap. In Figure 6, green blocks show prefetching that is overlapped, and orange blocks show page fault-induced loading that is not overlapped. In Figure 6(a), with a fast disk, disk loading is fully overlapped. In Figures 6(b) and 6(c), with a device has a slow disk, only part of the disk loading is overlapped, while others are fully overlapped. In Figures 6(c), 6(d) and 6(e), while disk loading is not fully hidden, piped-ring parallelism significantly reduces token latency compared to vanilla pipeline parallelism. In Figure 6(e), while prefetching is used, it exceeds memory limits and triggers "prefetch-release", where the OS releases the earlier prefetched layers as new ones load, adding disk cost with no benefit. This underscores the need to combine piped-ring parallelism with prefetching for higher efficiency.

A.3 Layer-to-Device Assignment: From Latency Analysis to Vectorized Model

Assume there are M devices, where the layer window size for device d_m is w_m . On device d_m , the number of GPU layers n_m is defined as: within a layer window of size w_m , n_m layers run on the GPU, while the remaining $w_m - n_m$ layers run on the CPU (w_m and n_m can vary across devices). Our objective is to find a vector $\mathbf{w} = \{w_1, \cdots, w_M\}$ and a vector $\mathbf{n} = \{n_1, \cdots, n_M\}$ to minimize the token latency T, which is the sum of latencies from computation T_m^{comp} , memory access T_m^{mem} , disk loading T_m^{disk} , and communication T_m^{comm} on each device.

$$T = \sum_{m=1}^{M} \left(T_m^{\text{comp}} + T_m^{\text{mem}} + T_m^{\text{disk}} + T_m^{\text{comm}} \right). \tag{11}$$

Here, we minimize $T = \sum_{m=1}^{M} T_m$ instead of $T = \max\{T_m\}$ because Figure 6(f) is an idealized visualization. In practice, the OS does not start prefetching immediately after computation, and the timing is unknown. As a result, device 4 experiences more "bubbles" and higher page fault-induced latency than expected. This uncertainty prevents solving $T = \max\{T_m\}$ before deployment (it is also hard to measure), and historical data is useless due to fluctuating device conditions. Thus, we take a worst-case approach, assuming the OS hasn't started prefetching when computation begins, leading to our objective $T = \sum_{m=1}^{M} T_m$. Next, we analyze these latencies in detail.

Estimation of computation latency T_m^{comp} . The computation latency on device d_m is defined as the time taken to process l_m model layers and the output layer (if d_m is the head device), where l_m^{gpu} layers run on the GPU, and the remaining $l_m - l_m^{\text{gpu}}$ layers and output layer run on the CPU. Here, we have $l_m = \left\lfloor \frac{L}{W} \right\rfloor w_m + \min(w_m, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R)))$, $l_m^{\text{gpu}} = \left\lfloor \frac{L}{W} \right\rfloor n_m + \min(n_m, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R)))$, where $n_m \leq w_m$, $W = \sum_{m=1}^{M} w_m$, and $R = L \mod W$. Since the input layer adopts a look-up table, it does not contribute to computation latency.

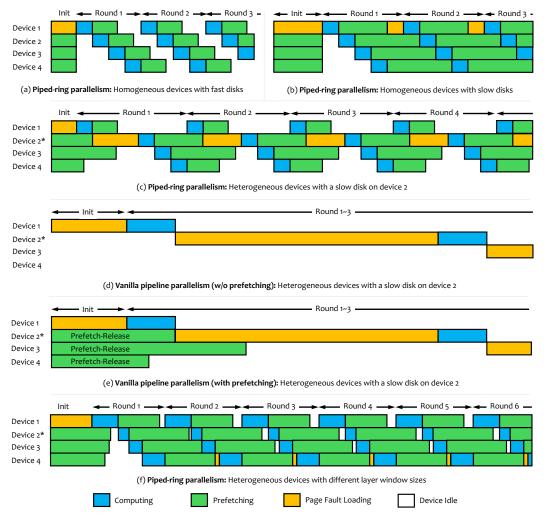


Figure 6: Timeline of (a,b) piped-ring parallelism on homogeneous devices with fast and slow disks; (c,e) piped-ring parallelism on heterogeneous devices with same and different window sizes; and (d,e) vanilla pipeline parallelism on heterogeneous devices with and without prefetching.

To estimate the computation time, we develop a model profiler to count the number of floating-point operations (FLOPs) for each model layer and a device profiler to measure the floating-point throughput (FLOPS) of each device. Taking Q4K as an example, the model weights are primarily quantized in the Q4K format, though some weights use other formats. Specifically, we consider $\mathcal{Q} = \{Q4_K, Q5_K, Q6_K, Q8_0, F16, F32\}$, as well as three types of backends CPU, CUDA, and Metal. The FLOPs for each layer \mathcal{F}_m and output layer $\mathcal{F}_m^{\text{out}}$ consists of 6 values $\mathcal{F}_m = \{f_m^{q4k}, f_m^{q5k}, f_m^{q6k}, f_m^{q80}, f_m^{f16}, f_m^{f32}\}$, $\mathcal{F}_m^{\text{out}} = \{f_{m,\text{out}}^{q4k}, f_{m,\text{out}}^{q5k}, f_{m,\text{out}}^{q80}, f_{m,\text{out}}^{f16}, f_{m,\text{out}}^{f32}\}$, with each represents the FLOPs under a specific quantization format. The FLOPS \mathcal{S}_m consists of 3 sets $\{\mathcal{S}_m^{\text{cpu}}, \mathcal{S}_m^{\text{cuda}}, \mathcal{S}_m^{\text{metal}}\}$, with each set consists of 6 values (e.g., for CPU, $\mathcal{S}_m^{\text{cpu}} = \{s_m^{\text{cpu},q4k}, s_m^{\text{cpu},q5k}, s_m^{\text{cpu},q6k}, s_m^{\text{cpu},q6k}, s_m^{\text{cpu},f32}\}$) representing the floating-point throughput under specific device and quantization format. \mathcal{F}_m , $\mathcal{F}_m^{\text{out}}$ and \mathcal{S}_m can be easily extended.

With these profilers, we can estimate the computation time as follows:

$$T_m^{\text{comp}} = (l_m - l_m^{\text{gpu}}) \sum_{q \in \mathcal{Q}} \frac{f_m^q}{s_m^{\text{cpu}}, q} + l_m^{\text{gpu}} \sum_{q \in \mathcal{Q}} \frac{f_m^q}{s_m^{\text{gpu}}, q} + \mathbb{I}_{m=1} \cdot \sum_{q \in \mathcal{Q}} \frac{f_{m, \text{out}}^q}{s_m^{\text{cpu}}, q}, \tag{12}$$

Here, $s_m^{\mathrm{gpu},q}$ refers to GPU FLOPS. If prima.cpp is compiled with CUDA support, $s_m^{\mathrm{gpu},q}$ corresponds to $s_m^{\mathrm{cuda},q}$. If it runs on an Apple device with Metal enabled, $s_m^{\mathrm{gpu},q}$ corresponds to $s_m^{\mathrm{metal},q}$. In our current implementation, the output layer is executed only on CPUs by the master node m=1.

Estimation of memory access latency $T_m^{\rm mem}$. This latency consists of three components: a) kvcache $copy time \ T_m^{kv_cpy}$: the time taken to copy the new token cache to the key-value cache storage within a specific device; b) $device \ copy \ time \ T_m^{dev_cpy}$: the time taken to copy hidden states between the CPU and the GPU (CUDA or Metal); c) $device \ loading \ time \ T_m^{dev_load}$: the time taken to load data from RAM or VRAM into the processing cores of the CPU or GPU.

For the kvcache copy time, in each token step, new key and value caches are generated with dimensions $(h_k e_k, 1)$ and $(h_v e_v, 1)$, respectively. Here, h_k and h_v are the number of attention heads for the key and value caches, e_k and e_v are the embedding size per head for the key and value vectors. Thus, for generating one token, each layer needs to copy $h_k e_k + h_v e_v$ values to the key-value cache storage. If the values are stored in the F16 format, each value takes 2 bytes, so the total number of bytes to be copied is $2(h_k e_k + h_v e_v)$. In the device profiler module, we measure the time of copying $2(h_k e_k + h_v e_v)$ bytes within CPU, CUDA, and Metal to obtain $t_m^{\rm kv_cpy,cpu}$ and $t_m^{\rm kv_cpy,gpu}$. Then, the kvcache copy time $T_m^{\rm kv_cpy}$ can be estimated by $(l_m - l_m^{\rm gpu})t_m^{\rm kv_cpy,cpu} + l_m^{\rm gpu}t_m^{\rm kv_cpy,gpu}$.

For the device copy time, this latency arises when the GPU is enabled, as it involves copying the input from RAM to VRAM and then copying the output from VRAM back to RAM. Both the input and output have dimensions (e,1), where e represents the embedding size. These values are typically stored in the F32 format. In the device profiler module, we measured the latency for two operations: the time taken to copy 4e bytes of data from RAM to VRAM, denoted as $t_m^{\text{ram-vram}}$, and the time taken to copy 4e bytes of data from VRAM to RAM, denoted as $t_m^{\text{vram-ram}}$. For a sequence of layers within a window, one RAM-to-VRAM copy and one VRAM-to-RAM copy are needed, so the device copy time for one window is $t_m^{\text{ram-vram}} + t_m^{\text{vram-ram}}$. For device d_m , it was assigned $\mathcal{W}_m = \left\lfloor \frac{L}{W} \right\rfloor + \min(1, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R)))$ windows. Thus, the device copy time for device d_m is $T_m^{\text{dev_cpy}} = \mathcal{W}_m(t_m^{\text{ram-vram}} + t_m^{\text{vram-ram}})(1 - \mathbb{I}_m^{\text{UMA}})$, where $\mathbb{I}_m^{\text{UMA}} = 1$ indicates that device d_m uses a unified memory architecture (UMA, e.g., Apple with M1 chip) and the CPU and GPU use the shared memory, so no explicit RAM-VRAM copy is needed.

For the device loading time, processing cores must load data from RAM/VRAM into registers before executing instructions, which incurs latency. However, the theoretical bandwidth of memory cards cannot estimate this latency because apps often fail to fully utilize the bandwidth, and multi-level caching also has a significant influence. To capture these effects, our device profiler implements an operator to read data from RAM/VRAM into registers. By measuring its latency with data volumes similar to the tensor sizes, we have the practical throughput, denoted as $\{\mathcal{T}_m^{\text{cpu}}, \mathcal{T}_m^{\text{cuda}}, \mathcal{T}_m^{\text{metal}}\}$. Next, we count the data volume that needs to be loaded into registers during each token step, which typically consists of the weight data and the key-value cache. In the model profiler, we record the total bytes of weight data for the input and output layer as b_i, b_o , and for each layer as b. Additionally, the key-value cache size for each layer is $2(h_k e_k + h_v e_v) n_{kv}$, where n_{kv} is the number of tokens for which the cache is stored. Then, the device loading time $\mathcal{T}_m^{\text{dev_load}}$ can be expressed as $\mathcal{T}_m^{\text{dev_load}} = (\frac{l_m - l_m^{\text{gpu}}}{\mathcal{T}_m^{\text{pu}}})(b + 2(h_k e_k + h_v e_v) n_{kv}) + \frac{b_i/V + b_o}{\mathcal{T}_m^{\text{pu}}} \cdot \mathbb{I}_{m=1}$, where $\mathcal{T}_m^{\text{gpu}}$ depends on the hardware: it equals to $\mathcal{T}_m^{\text{metal}}$ if the GPU uses Metal, or $\mathcal{T}_m^{\text{cuda}}$ for CUDA, and V is the vocabulary size

Now we can combine the three latency components and give the formal definition of the memory access latency T_m^{mem} :

$$T_{m}^{\text{mem}} = (l_{m} - l_{m}^{\text{gpu}})t_{m}^{\text{kv_cpy,cpu}} + l_{m}^{\text{gpu}}t_{m}^{\text{kv_cpy,gpu}} + \mathcal{W}_{m}(t_{m}^{\text{ram-vram}} + t_{m}^{\text{vram-ram}})(1 - \mathbb{I}_{m}^{\text{UMA}})$$
(13)
+
$$(\frac{l_{m} - l_{m}^{\text{gpu}}}{\mathcal{T}_{m}^{\text{cpu}}} + \frac{l_{m}^{\text{gpu}}}{\mathcal{T}_{m}^{\text{gpu}}})(b + 2(h_{k}e_{k} + h_{v}e_{v})n_{kv}) + \frac{b_{i}/V + b_{o}}{\mathcal{T}_{m}^{\text{cpu}}} \cdot \mathbb{I}_{m=1}.$$
(14)

Estimation of disk loading latency $T_m^{\rm disk}$. Prima.cpp is designed to run on memory-constrained home devices, so it cannot load the entire model weights into physical RAM. To address this, prima.cpp uses mmap to manage model weights. By using mmap, model weights are loaded into memory from disk only when needed for computation, and the OS will release inactive mmap-ed pages when memory pressure is high. This prevents OOM issues but incurs significant disk I/O latency because mmap must reload the model weights when they are accessed again after being released. To estimate this disk loading latency, it is necessary to determine the data volume that mmap needs to load in each

token step. This is a challenging task because different OSs have very different memory management behaviors and great dynamics.

On macOS (without Metal) and Linux, the OS gradually reclaims memory. When memory pressure is moderate, i.e., when $b_{lio}+2(h_ke_k+h_ve_v)n_{kv}(l_m-l_m^{\rm gpu})+c^{\rm cpu}>d_m^{\rm avail}$, mmap-ed pages are released incrementally until the pressure is alleviated. As a result, some weight data remain in the page cache, and the amount of data that map needs to reload is $\max(b_{lio} + 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\rm gpu}) + c^{\rm cpu} - d_m^{\rm avail}, 0)$, where $b_{lio} = (l_m - l_m^{\rm gpu})b + (b_i/V + b_o) \cdot \mathbb{I}_{m=1}, 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\rm gpu})$ is the key-value cache size, and $c^{\rm cpu}$ is the total computing buffer size. If CUDA is enabled on Linux, the model weights in private VRAM are locked by the CUDA driver, keeping them resident so no disk I/O occurs. Therefore, the disk loading latency for macOS (without Metal) and Linux can be estimated as

$$T_{m,\mathrm{macOS(no\ Metal)}}^{\mathrm{disk}} = T_{m,\mathrm{Linux}}^{\mathrm{disk}} = \frac{\max\left(b_{lio} + 2(h_k e_k + h_v e_v)n_{kv}(l_m - l_m^{\mathrm{gpu}}) + c^{\mathrm{cpu}} - d_m^{\mathrm{avail}}, b_i/V\right)}{s_m^{\mathrm{disk}}}.$$
 (15) For $T_{m,\mathrm{macOS(no\ Metal)}}^{\mathrm{disk}}$, $l_m^{\mathrm{gpu}} = 0$ and s_m^{disk} is the random read throughput. On Linux, mmap is configured for sequential access, so s_m^{disk} is now the sequential read throughput.

However, when Metal is enabled on macOS, the behavior changes. Metal loads mmap pages into the shared memory, and the OS prioritizes retaining these pages. That is, the OS is more inclined to swap out or compress active pages while keeping mmap-ed model weight pages in shared memory intact. However, when memory is exhausted (with free and inactive pages exhausted, the compression pool nearing saturation, and heavy swap usage), macOS will release these mmap-ed pages in a more aggressive manner. This may cause the entire model weights to be repeatedly loaded and released. As a result, when the required memory exceeds the total available memory, i.e., when $l_m b + (b_i/V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_k e_k + h_v e_v) n_{kv} l_m + c^{\text{cpu}} + c^{\text{gpu}} > d_{m,\text{metal}}^{\text{avail}}$, device d_m needs to reload $l_m b + (b_i/V + b_o) \cdot \mathbb{I}_{m=1}$ bytes in each token step. Here, $d_{m,\text{metal}}^{\text{avail}}$ denotes the maximum working set size recommended by Metal. By measuring the random read throughput $s_m^{\rm disk}$ of disk, we can calculate the disk loading latency for macOS (with Metal) as:

$$T_{m,\text{macOS (with Metal)}}^{\text{disk}} = \max\left(\frac{l_m b + (b_i/V + b_o) \cdot \mathbb{I}_{m=1}}{s_m^{\text{disk}}} \cdot \mathbb{I}\left(l_m b + (b_i/V + b_o) \cdot \mathbb{I}_{m=1}\right) + 2(h_k e_k + h_v e_v) n_{kv} l_m + c^{\text{cpu}} + c^{\text{gpu}} - d_{m,\text{metal}}^{\text{avail}}\right), b_i/V\right).$$
(16)

When running on Android devices, the OS prioritizes swapping out inactive pages to disk, such as memory used by background apps, to ensure that the active app runs smoothly. As a result, the available RAM for prima.cpp can be higher than initially expected, as the OS will swap out cold pages to disk, freeing up additional space in memory. Thus, on Android, the number of byte that mmap needs to reload is $\max(b_{bio}+2(h_ke_k+h_ve_v)n_{kv}(l_m-l_m^{\rm gpu})+c^{\rm cpu}-d_m^{\rm avail}-d_m^{\rm swapout},0)$, where $d_m^{\rm swapout}=\min(\max(0,b_{bio}+2(h_ke_k+h_ve_v)n_{kv}(l_m-l_m^{\rm gpu})+c^{\rm cpu}-d_m^{\rm avail}),\min(d_m^{\rm bytes_can_swap},d_m^{\rm swap_avail}))$ represents the data bytes that are swapped out to disk, $d_m^{\rm bytes_can_swap}$ is the data bytes of currently used memory that can be swapped out, and $d_m^{\text{swap_avail}}$ is the total available swap space on the device. Then

$$T_{m,\text{Android}}^{\text{disk}} = \frac{\max(b_{bio} + 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\text{gpu}}) + c^{\text{cpu}} - d_m^{\text{avail}} - d_m^{\text{swapout}}, b_i / V)}{s_m^{\text{disk}}}.$$
 (17)

By aggregating them, we obtain a unified expression compatible for cross-platform devices:

$$T_{m}^{\text{disk}} = T_{m,\text{macOS (no Metal)}}^{\text{disk}} \cdot \mathbb{I}_{\text{macOS (no Metal)}} + T_{m,\text{macOS (with Metal)}}^{\text{disk}} \cdot \mathbb{I}_{\text{macOS (with Metal)}}$$

$$+ T_{m,\text{Linux}}^{\text{disk}} \cdot \mathbb{I}_{\text{Linux}} + T_{m,\text{Android}}^{\text{disk}} \cdot \mathbb{I}_{\text{Android}},$$

$$(18)$$

where $\mathbb{I}_{macOS\ (no\ Metal)}, \mathbb{I}_{macOS\ (with\ Metal)}, \mathbb{I}_{Linux}, \mathbb{I}_{Android}$ are indicator functions. This expression can be easily extended to include new OSs, e.g., Windows will be added in future updates.

Estimation of network communication latency T_m^{comm} . In prima.cpp, devices are logically interconnected in a ring structure, where each device receives input from its predecessor, processes a layer window, and sends the output to its successor. After a device completes the computation

for one layer window, it transmits the result (e values in F32 format, totaling 4e bytes) to the next device for further computation on the next layer window. Therefore, during a token step, the number of network communications on device d_m equals the number of layer windows, which is $\mathcal{W}_m = \left\lfloor \frac{L}{W} \right\rfloor + \min(1, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R)))$. By measuring the latency t_m^{comm} for transmitting 4e bytes between adjacent devices, we can estimate the network communication latency on device d_m as:

$$T_m^{\text{comm}} = \left(\left\lfloor \frac{L}{W} \right\rfloor + \min(1, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R))) \right) t_m^{\text{comm}}. \tag{20}$$

By aggregating all these latencies, we have the objective as:

$$\begin{split} T &= \sum_{m=1}^{M} \left[(l_m - l_m^{\rm gau}) \sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} + l_m^{\rm gau} \sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} + \mathbb{I}_{m=1} \cdot \sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} \right] \\ &+ (l_m - l_m^{\rm gau}) t_m^{\rm kv, cpy, cpu} + l_m^{\rm gau} t_m^{\rm kv, cpy, spu} + \mathcal{W}_m(t_m^{\rm cmv, vam} + t_m^{\rm vam-ram})(1 - \mathbb{I}_m^{\rm UMA}) \\ &+ \left(\frac{l_m - l_m^{\rm gau}}{T_m^{\rm gau}} + \frac{l_m^{\rm gau}}{T_m^{\rm gau}} \right) (b + 2(h_k e_k + h_v e_v) n_{kv} + b_v / T_{rm}^{\rm gau}} \cdot \mathbb{I}_{m=1} \right. \\ &+ \frac{\max(l_m b + (b_i / V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_k e_k + h_v e_v) n_{kv} l_m + e^{cpu} - d_m^{\rm gau}, b_i / V)}{s_m^{\rm gav}} \cdot \mathbb{I}_{macOS \, (no \, Metal)} \\ &+ \max \left(\frac{l_m b + (b_i / V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\rm gau}) + e^{cpu} + e^{spu} - d_m^{\rm gau}, b_i / V)}{s_m^{\rm gab}} \cdot \mathbb{I}_{macOS \, (with \, Metal)} \right. \\ &+ \frac{b_i}{s_m^{\rm disk} V} \cdot \mathbb{I}_{macOS \, (with \, Metal)} \\ &+ \frac{\max((l_m - l_m^{\rm gau}) b + (b_i / V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\rm gau}) + e^{cpu} - d_m^{\rm gau}, b_i / V)}{s_m^{\rm gab}} \cdot \mathbb{I}_{macOS \, (with \, Metal)} \\ &+ \frac{\max((l_m - l_m^{\rm gau}) b + (b_i / V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_k e_k + h_v e_v) n_{kv} (l_m - l_m^{\rm gau}) + e^{cpu} - d_m^{\rm gau}, b_i / V)}{s_m^{\rm gau}} \cdot \mathbb{I}_{nadroic} \\ &+ \left(\left\lfloor \frac{L}{W} \right\rfloor + \min(1, \max(0, R - \sum_{j=1}^{m-1} \min(w_j, R))) t_{comm} \right) \\ &= \sum_{m=1}^{M} \left[\left(\sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} + t_m^{\rm kv, cpy, cpu} + \frac{b + 2(h_k e_k + h_v e_v) n_{kv}}{T_m^{\rm gau}} \right) + \mathcal{W}_m \left((t_m^{\rm tam-vram} + t_m^{\rm vran-ram})(1 - \mathbb{I}_m^{\rm tam}) + t_m^{\rm comm} \right) \\ &+ l_m^{\rm gau} \left(\sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} + t_m^{\rm kv, cpy, cpu} + \frac{b + 2(h_k e_k + h_v e_v) n_{kv}}{T_m^{\rm gau}} \right) + \mathcal{W}_m \left((t_m^{\rm tam-vram} + t_m^{\rm vran-ram})(1 - \mathbb{I}_m^{\rm tam}) + t_m^{\rm comm} \right) \\ &+ l_m^{\rm gau} \left(\sum_{q \in Q} \frac{f_m^q}{s_m^{\rm gau}, q} + t_m^{\rm kv, cpy, cpu} + \frac{b + 2(h_k e_k + h_v e_v) n_{kv}}{T_m^{\rm gau}} \right) + \mathcal{W}_m \left(l_m^{\rm tam-vram} + l_m^{\rm vran-ram} \right) + l_m^{\rm tam, cos} \left(l_m^{\rm tam, cos} \right) \\ &+ l_m^{\rm tam, cos} \left(l_m^{\rm tam, cos} \right) \cdot \mathbb{I}$$

To remove the max operator, we decompose the disk loading latency into multiple terms, separately accounting for cases where memory is sufficient or insufficient. Let \mathcal{M} be the set of all devices, and $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ be the subsets of devices that satisfy the respective conditions in Cases 1-4, where $\mathcal{M}_1 \cap \mathcal{M}_2 \cap \mathcal{M}_3 \cap \mathcal{M}_4 = \emptyset$ and $\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 \cup \mathcal{M}_4 = \mathcal{M}$.

 $\begin{array}{lll} \textit{Case 1 (macOS with Metal disabled and insufficient RAM):} & \textit{If $l_mb+(b_i/V+b_o)$} \cdot \\ \mathbb{I}_{m=1} \ + \ 2(h_ke_k + h_ve_v)n_{kv}l_m + c^{\text{cpu}} > d_m^{\text{avail}} & \textit{and } s_m^{\text{disk}} > s_m^{\text{disk}}, & \textit{then } T_m^{\text{disk}} = \frac{l_mb+(b_i/V+b_o)\cdot\mathbb{I}_{m=1}+2(h_ke_k+h_ve_v)n_{kv}l_m+c^{\text{cpu}}-d_m^{\text{avail}}}{s_m^{\text{disk}}}, & m \in \mathcal{M}_1. \end{array}$

Case 2 (macOS with Metal enabled and insufficient RAM): If $l_mb + (b_i/V + b_o) \cdot \mathbb{I}_{m=1} + 2(h_ke_k + h_ve_v)n_{kv}l_m + c^{\text{cpu}} + c^{\text{gpu}} > d_{m,\text{metal}}^{\text{avail}}$ and $s_m^{\text{disk}} > s_{\text{threshold}}^{\text{disk}}$, then $T_m^{\text{disk}} = \frac{l_mb + (b_i/V + b_o) \cdot \mathbb{I}_{m=1}}{s_m^{\text{disk}}}, m \in \mathcal{M}_2$.

Case 3 (Linux and Android with insufficient RAM): If
$$(l_m - l_m^{\text{gpu}}) \left[b + 2(h_k e_k + h_v e_v) n_{kv} \right] + (b_i/V + b_o) \cdot \mathbb{I}_{m=1} + c^{\text{cpu}} > d_m^{\text{avail}} + d_m^{\text{swapout}} \cdot \mathbb{I}_{\text{Android}} \text{ and } s_m^{\text{disk}} > s_{\text{threshold}}^{\text{disk}}, \text{ then } T_m^{\text{disk}} = \frac{1}{s_m^{\text{disk}}} \left((l_m - l_m^{\text{gpu}}) [b + 2(h_k e_k + h_v e_v) n_{kv}] + (b_i/V + b_o) \cdot \mathbb{I}_{m=1} + c^{\text{cpu}} - d_m^{\text{avail}} - d_m^{\text{swapout}} \cdot \mathbb{I}_{\text{Android}} \right), m \in \mathcal{M}_3.$$

Case 4 (OS with sufficient RAM or low disk speed): In these cases, the physical RAM is large enough to hold the model weights or the disk speed is too slow (i.e., $s_m^{\rm disk} < s_{\rm threshold}^{\rm disk}$). As a result, no disk loading is expected, except for the latency incurred during lookup table access, thus $T_m^{\rm disk} = \frac{b_i}{s_m^{\rm disk}V}, m \in \mathcal{M}_4$.

With these cases, we can rewrite the objective function as follows:

$$T = \sum_{q \in \mathcal{Q}} \frac{f_{1,\text{out}}^{q}}{s_{1}^{\text{cpu},q}} + \frac{b_{i}/V + b_{o}}{\mathcal{T}_{1}^{\text{cpu}}} + \frac{b_{i}/V}{s_{1}^{\text{disk}}} + \frac{b_{o}}{s_{1}^{\text{disk}}} \cdot \mathbb{I}_{1 \notin \mathcal{M}_{4}}$$

$$+ \sum_{m \in \mathcal{M}} \left[(l_{m} - l_{m}^{\text{gpu}}) \left(\sum_{q \in \mathcal{Q}} \frac{f_{m}^{q}}{s_{m}^{\text{cpu},q}} + t_{m}^{\text{kv_cpy,cpu}} + \frac{b + 2(h_{k}e_{k} + h_{v}e_{v})n_{kv}}{\mathcal{T}_{m}^{\text{cpu}}} \right) \right.$$

$$+ l_{m}^{\text{gpu}} \left(\sum_{q \in \mathcal{Q}} \frac{f_{m}^{q}}{s_{m}^{\text{gpu},q}} + t_{m}^{\text{kv_cpy,gpu}} + \frac{b + 2(h_{k}e_{k} + h_{v}e_{v})n_{kv}}{\mathcal{T}_{m}^{\text{gpu}}} \right) + \mathcal{W}_{m} \left((t_{m}^{\text{ram-vram}} + t_{m}^{\text{vram-ram}})(1 - \mathbb{I}_{m}^{\text{UMA}}) + t_{m}^{\text{comm}} \right) \right]$$

$$+ \sum_{m \in \mathcal{M}_{2}} \frac{l_{m}b}{s_{m}^{\text{disk}}} + \sum_{m \in \mathcal{M}_{1} \cup \mathcal{M}_{3}} \left[\frac{(l_{m} - l_{m}^{\text{gpu}})[b + 2(h_{k}e_{k} + h_{v}e_{v})n_{kv}]}{s_{m}^{\text{disk}}} + \frac{c^{\text{cpu}} - d_{m}^{\text{avail}} - d_{m}^{\text{avapout}} \cdot \mathbb{I}_{\text{Android}}}{s_{m}^{\text{disk}}} \right]$$

To further simplify the objective function, we make the following assumption.

Assumption 1. Let $\frac{L}{W}$ be an integer, i.e., R = 0, where $W = \sum_{m \in \mathcal{M}} w_m$, all devices are assigned an equal number of windows and all the windows are filled.

Now, we have
$$l_m = \frac{w_m L}{W}$$
, $l_m^{\rm gpu} = \frac{n_m L}{W}$, $\mathcal{W}_m = \frac{L}{W}$. Let $b' = b + 2(h_k e_k + h_v e_v) n_{kv}$, $\alpha_m = \sum_{q \in \mathcal{Q}} \frac{f_m^q}{s_m^{\rm spu}, q} + t_m^{\rm kv_cpy,cpu} + \frac{b'}{\mathcal{T}_m^{\rm cpu}}$, $\beta_m = \sum_{q \in \mathcal{Q}} \frac{f_m^q}{s_m^{\rm spu}, q} - \sum_{q \in \mathcal{Q}} \frac{f_m^q}{s_m^{\rm spu}, q} + t_m^{\rm kv_cpy,cpu} - t_m^{\rm kv_cpy,cpu} + \frac{b'}{\mathcal{T}_m^{\rm cpu}} - \frac{b'}{\mathcal{T}_m^{\rm cpu}}$, $\xi_m = (t_m^{\rm ram-vram} + t_m^{\rm vram-vram})(1 - \mathbb{I}_m^{\rm UMA}) + t_m^{\rm comm}$, $\kappa = \sum_{q \in \mathcal{Q}} \frac{f_{1, \rm out}^q}{s_1^{\rm cpu}, q} + \frac{b_i/V + b_o}{\mathcal{T}_1^{\rm cpu}} + \frac{b_i/V}{s_1^{\rm disk}} + \frac{b_o}{s_1^{\rm disk}} \cdot \mathbb{I}_{1 \not\in \mathcal{M}_4} + \sum_{m \in \mathcal{M}_1 \cup \mathcal{M}_3} \frac{c^{\rm cpu} - d_m^{\rm avail} - d_m^{\rm swapout} \cdot \mathbb{I}_{\rm Android}}{s_m^{\rm disk}}$, where α_m, β_m, ξ_m are platform-specific constants and κ is a global constant. Then, we add the first general term to the three platform-specific terms and obtain:

$$T = \frac{L}{W} \sum_{m \in \mathcal{M}_1} \left[(\alpha_m + \frac{b'}{s_m^{\text{disk}}}) w_m + \xi_m \right] + \frac{L}{W} \sum_{m \in \mathcal{M}_2} \left[(\alpha_m + \frac{b}{s_m^{\text{disk}}}) w_m + \beta_m n_m + \xi_m \right]$$

$$+ \frac{L}{W} \sum_{m \in \mathcal{M}_3} \left[(\alpha_m + \frac{b'}{s_m^{\text{disk}}}) w_m + (\beta_m - \frac{b'}{s_m^{\text{disk}}}) n_m + \xi_m \right] + \frac{L}{W} \sum_{m \in \mathcal{M}_4} \left[\alpha_m w_m + \beta_m n_m + \xi_m \right] + \kappa.$$

This objective is a sum over three sets, $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$. Each summand involves expressions linear in w_m and n_m , plus constant terms specific to the platform. To clarify the form, we define a linear function $f(a,b,c)=aw_m+bn_m+c$, where the platform-specific constants a,b,c are independent of the decision variables w_m and n_m . Consequently, the objective can be rearranged to a combination

of linear functions:

$$T = \frac{L}{W} \left[\sum_{m \in \mathcal{M}_1} f(\alpha_m + \frac{b'}{s_m^{\text{disk}}}, 0, \xi_m) + \sum_{m \in \mathcal{M}_2} f(\alpha_m + \frac{b}{s_m^{\text{disk}}}, \beta_m, \xi_m) \right]$$
$$+ \sum_{m \in \mathcal{M}_2} f(\alpha_m + \frac{b'}{s_m^{\text{disk}}}, \beta_m - \frac{b'}{s_m^{\text{disk}}}, \xi_m) + \sum_{m \in \mathcal{M}_2} f(\alpha_m, \beta_m, \xi_m) \right] + \kappa$$

Noted that the objective function T is not linear because $W = \sum_{m \in \mathcal{M}} w_m$ is the sum of decision variables, and the term $\frac{1}{W}$ introduces a nonlinear dependency, which makes the problem a nonlinear optimization problem.

Now, we put it all together to present a final model:

$$\min_{w_m, n_m} \quad T \tag{22}$$

$$s.t. w_m \in \mathbb{Z}_{>0}, n_m \in \mathbb{Z}_{>0}, n_m \le w_m \le L, (23)$$

$$L = kW, k \in \mathbb{Z}_{>0},\tag{24}$$

$$W = \sum_{m \in \mathcal{M}} w_m,\tag{25}$$

$$f(a,b,c) = aw_m + bn_m + c, (26)$$

$$\bigcap_{i=1}^{4} \mathcal{M}_i = \emptyset, \bigcup_{i=1}^{4} \mathcal{M}_i = \mathcal{M}, \tag{27}$$

$$w_m > \frac{W}{Lb'}(d_m^{\text{avail}} - b_m^{cio}), m \in \mathcal{M}_1, \tag{28}$$

$$w_m > \frac{W}{Lb'} (d_{m,\text{metal}}^{\text{avail}} - b_m^{cio} - c^{\text{gpu}}), m \in \mathcal{M}_2, \tag{29}$$

$$w_m - n_m > \frac{W}{Lh'} (d_m^{\text{avail}} + d_m^{\text{swapout}} \cdot \mathbb{I}_{\text{Android}} - b_m^{cio}), m \in \mathcal{M}_3,$$
 (30)

$$w_m \cdot \mathbb{I}_{\text{macOS (no Metal)}} < \frac{W}{I, b'} (d_m^{\text{avail}} - b_m^{cio}), m \in \mathcal{M}_4,$$
 (31)

$$w_m \cdot \mathbb{I}_{\text{macOS (with Metal)}} < \frac{W}{I.b'} (d_{m,\text{metal}}^{\text{avail}} - b_m^{cio} - c^{\text{gpu}}), m \in \mathcal{M}_4,$$
 (32)

$$(w_m - n_m)(\mathbb{I}_{\text{Linux}} + \mathbb{I}_{\text{Android}}) < \frac{W}{Lb'}(d_m^{\text{avail}} + d_m^{\text{swapout}} \cdot \mathbb{I}_{\text{Android}} - b_m^{cio}), m \in \mathcal{M}_4,$$
 (33)

$$b_m^{cio} = (b_i/V + b_o) \cdot \mathbb{I}_{m=1} + c^{\text{cpu}},$$
 (34)

$$n_m \cdot \mathbb{I}_{\text{cuda}} \le \frac{W}{Lb'} (d_{m,\text{cuda}}^{\text{avail}} - c^{\text{gpu}}) \cdot \mathbb{I}_{\text{cuda}},$$
 (35)

$$n_m \cdot \mathbb{I}_{\text{metal}} \le \frac{W}{Lb'} (d_{m,\text{metal}}^{\text{avail}} - c^{\text{gpu}} - b_o \cdot \mathbb{I}_{m=1}) \cdot \mathbb{I}_{\text{metal}}, \tag{36}$$

$$n_m = 0$$
, if $\mathbb{I}_{\text{cuda}} = 0$ and $\mathbb{I}_{\text{metal}} = 0$. (37)

Constraint (23) requires that the window size w_n must be a positive integer, the number of GPU layers n_m must be a non-negative integer, and n_m cannot exceed w_m . Constraint (24) requires that all devices are assigned an equal number of windows and all the windows are filled. This is not mandatory in our implementation, but can simplify the problem model. Constraint (28-30) ensure that devices categorized into sets $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ meet the memory condition outlined in Cases 1-3. Similarly, Constraints (31-33) ensure that devices assigned to set \mathcal{M}_4 meet the memory condition outlined in Case 4. b_m^{cio} in eq. (34) is a platform-independent constant. Constraints (35-36) ensure that the VRAM used by CUDA or the shared memory used by Metal does not exceed the available capacity. Here, $d_{m,\text{cuda}}^{\text{avail}}$ denotes the available GPU private memory for CUDA, and $d_{m,\text{metal}}^{\text{avail}}$ denotes the maximum working set size recommended by Metal. Note that the output layer weights (of b_o bytes) are kept in Metal's shared memory but run on the CPU by default.

This is an integer linear fractional programming (ILFP) problem because the numerator is a linear function of the decision variables w_m, n_w and the denominator W is also a linear function of w_m . Moreover, the constraints are linear inequalities. Even when indicator variables

 $\mathbb{I}_{\text{macOS}}$, $\mathbb{I}_{\text{Linux}}$, $\mathbb{I}_{\text{Android}}$, \mathbb{I}_{cuda} , $\mathbb{I}_{\text{metal}}$, $\mathbb{I}_{m=1}$ appear, each device's platform is known in advance, so these indicators are fixed, they just activate or deactivate certain linear constraints for each device.

Next, we transform the above model into its vectorized form. Let the decision variables be $\mathbf{w}^{\mathrm{T}} = [w_1, w_2, \cdots, w_M], \mathbf{n}^{\mathrm{T}} = [n_1, n_2, \cdots, n_M],$ and the coefficients $\mathbf{a}, \mathbf{b}, \mathbf{c}$ be:

$$\boldsymbol{a} = \begin{bmatrix} \alpha_{m} + \frac{b'}{s_{m}^{\text{disk}}} \mid m \in \mathcal{M}_{1} \\ \hline \alpha_{m} + \frac{b}{s_{m}^{\text{disk}}} \mid m \in \mathcal{M}_{2} \\ \hline \alpha_{m} + \frac{b'}{s_{m}^{\text{disk}}} \mid m \in \mathcal{M}_{3} \\ \hline \hline \alpha_{m} \mid m \in \mathcal{M}_{4} \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 0 \mid m \in \mathcal{M}_{1} \\ \hline \beta_{m} \mid m \in \mathcal{M}_{2} \\ \hline \beta_{m} - \frac{b'}{s_{m}^{\text{disk}}} \mid m \in \mathcal{M}_{3} \\ \hline \beta_{m} \mid m \in \mathcal{M}_{4} \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} \xi_{m} \mid m \in \mathcal{M}_{1} \\ \hline \xi_{m} \mid m \in \mathcal{M}_{2} \\ \hline \xi_{m} \mid m \in \mathcal{M}_{3} \\ \hline \xi_{m} \mid m \in \mathcal{M}_{4} \end{bmatrix}.$$

To apply constraints to the subset of \boldsymbol{w} and \boldsymbol{n} corresponding to $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$, we define diagonal matricies $\boldsymbol{P}_w = \operatorname{diag}(-\boldsymbol{I}_{\mathcal{M}_1}, -\boldsymbol{I}_{\mathcal{M}_2}, -\boldsymbol{I}_{\mathcal{M}_3}, \boldsymbol{P}_{\mathcal{M}_4}^1, \boldsymbol{P}_{\mathcal{M}_4}^2, \boldsymbol{P}_{\mathcal{M}_4}^3)$, $\boldsymbol{P}_n = \operatorname{diag}(\boldsymbol{0}_{\mathcal{M}_1}, \boldsymbol{0}_{\mathcal{M}_2}, \boldsymbol{I}_{\mathcal{M}_3}, \boldsymbol{0}_{\mathcal{M}_4}^1, \boldsymbol{0}_{\mathcal{M}_4}^2, -\boldsymbol{P}_{\mathcal{M}_4}^3)$, where $\boldsymbol{I}_{\mathcal{M}_1}, \boldsymbol{I}_{\mathcal{M}_2}, \boldsymbol{I}_{\mathcal{M}_3}$ are identity matrices and $\boldsymbol{0}_{\mathcal{M}_1}, \boldsymbol{0}_{\mathcal{M}_2}, \boldsymbol{0}_{\mathcal{M}_3}$ are zero matrices corresponding to the subsets $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$, and $\boldsymbol{P}_{\mathcal{M}_4}^1, \boldsymbol{P}_{\mathcal{M}_4}^2, \boldsymbol{P}_{\mathcal{M}_4}^3$ are diagonal binary matricies (i.e., selection matricies) corresponding to the three constraints (31-33) within the subset \mathcal{M}_4 . To construct $\boldsymbol{P}_{\mathcal{M}_4}^1, \boldsymbol{P}_{\mathcal{M}_4}^2, \boldsymbol{P}_{\mathcal{M}_4}^3$, we define a binary vector $\boldsymbol{p}_{\text{macOS}}$, where a value of 1 indicates that the current device is running on macOS and a value of 0 indicates otherwise. The number of elements in $\boldsymbol{p}_{\text{macOS}}$ matches the number of devices in the set \mathcal{M}_4 . Similarly, we define binary vectors $\boldsymbol{p}_{\text{Linux}}, \boldsymbol{p}_{\text{Android}}, \boldsymbol{p}_{\text{metal}}$. Thus, we have $\boldsymbol{P}_{\mathcal{M}_4}^1 = \operatorname{diag}(\boldsymbol{p}_{\text{macOS}} \odot (1-\boldsymbol{p}_{\text{metal}})), \boldsymbol{P}_{\mathcal{M}_4}^2 = \operatorname{diag}(\boldsymbol{p}_{\text{macOS}} \odot \boldsymbol{p}_{\text{metal}}), \boldsymbol{P}_{\mathcal{M}_4}^3 = \boldsymbol{p}_{\text{Linux}} + \boldsymbol{p}_{\text{Android}}$.

To handle constraints (35-36), we define P_n^{gpu} as a similar diagonal binary matrix, with elements set to one for devices with CUDA or Metal support. Specifically, we let $P_n^{\text{gpu}} = P_n^{\text{cuda}} + P_n^{\text{metal}}$, where $P_n^{\text{cuda}} = \text{diag}(\mathbf{0}_{\mathcal{M}_1}, \mathbf{0}_{\mathcal{M}_2}, P_{\mathcal{M}_3}^{\text{cuda}}, P_{\mathcal{M}_4}^{\text{cuda}})$ and $P_n^{\text{metal}} = \text{diag}(\mathbf{0}_{\mathcal{M}_1}, I_{\mathcal{M}_2}, \mathbf{0}_{\mathcal{M}_3}, P_{\mathcal{M}_4}^{\text{metal}})$.

Let the decision variables be $\boldsymbol{w}_{\mathcal{M}_4}^{\mathrm{T}} = [w_m \mid m \in \mathcal{M}_4], \ \boldsymbol{n}_{\mathcal{M}_4}^{\mathrm{T}} = [n_m \mid m \in \mathcal{M}_4], \ \boldsymbol{w}'^{\mathrm{T}} = [\boldsymbol{w}^{\mathrm{T}}, \boldsymbol{w}_{\mathcal{M}_4}^{\mathrm{T}}, \boldsymbol{w}_{\mathcal{M}_4}^{\mathrm{T}}], \ \boldsymbol{n}'^{\mathrm{T}} = [\boldsymbol{n}^{\mathrm{T}}, \boldsymbol{n}_{\mathcal{M}_4}^{\mathrm{T}}, \boldsymbol{n}_{\mathcal{M}_4}^{\mathrm{T}}],$ the RAM upper bound be

$$z = rac{1}{Lb'} egin{array}{c} d_{\mathcal{M}_4}^{ ext{avail}}, n_{\mathcal{M}_4}^{ ext{avail}}, & \text{the RAM upper bound be} \ d_m^{ ext{avail}} - b_m^{cio} \mid m \in \mathcal{M}_1 \ \hline d_{m, ext{metal}}^{ ext{avail}} - b_m^{cio} - c^{ ext{gpu}} \mid m \in \mathcal{M}_2 \ \hline d_m^{ ext{avail}} + d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} - b_m^{cio} \mid m \in \mathcal{M}_3 \ \hline -d_m^{ ext{avail}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{swapout}} \cdot \mathbb{I}_{ ext{Android}} + b_m^{cio} \mid m \in \mathcal{M}_4 \ \hline -d_m^{ ext{avail}} - d_m^{ ext{avail}} - d_m^{ ext{avail}} - d_m^{ ext{avail}} - d_m^{ ext{avail}} + d_m^{ ext{avail}} - d_m^{ ext{ava$$

and the VRAM/shared memory upper bound be $m{z}^{
m gpu} = [z_1^{
m gpu}, \cdots, z_M^{
m gpu}]$, where

$$z_m^{\rm gpu} = \frac{1}{Lb'} \cdot \begin{cases} 0, & \text{if } \mathbb{I}_{\rm cuda} = 0 \text{ and } \mathbb{I}_{\rm metal} = 0, \\ d_{m,{\rm cuda}}^{\rm avail} - c^{\rm gpu}, & \text{if } \mathbb{I}_{\rm cuda} = 1, \\ d_{m,{\rm metal}}^{\rm avail} - c^{\rm gpu}, & \text{if } \mathbb{I}_{\rm metal} = 1 \text{ and } m \neq 1, \\ d_{m,{\rm metal}}^{\rm avail} - c^{\rm gpu} - d_o, & \text{if } \mathbb{I}_{\rm metal} = 1 \text{ and } m = 1. \end{cases}$$

The problem model can then be reformated as:

$$\min_{\boldsymbol{w},\boldsymbol{n}} \quad L \cdot \frac{\boldsymbol{a}^{\mathrm{T}} \cdot \boldsymbol{w} + \boldsymbol{b}^{\mathrm{T}} \cdot \boldsymbol{n} + \boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{c}}{\boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{w}} + \kappa, \tag{38}$$

$$t. w_m \in \mathbb{Z}_{>0}, n_m \in \mathbb{Z}_{>0}, n_m \le w_m \le L, (39)$$

$$L - k(\boldsymbol{e}^{\mathrm{T}} \cdot \boldsymbol{w}) = 0, k \in \mathbb{Z}_{>0}, \tag{40}$$

$$P_w \cdot w' + P_n \cdot n' + e^{\mathrm{T}} \cdot w \cdot z < 0, \tag{41}$$

$$-P_n^{\text{gpu}} \cdot \boldsymbol{z}^{\text{gpu}} \cdot \boldsymbol{e}^{\text{T}} \cdot \boldsymbol{w} + P_n^{\text{gpu}} \cdot \boldsymbol{n} \le 0.$$
 (42)

Table 5: Summary of key symbols and their explanations.

Symbol	Explanation
\overline{M}	Number of devices.
w_m	Layer window size on device d_m .
n_m	Number of GPU layers on device d_m .
\overline{T}	Token latency.
l_m	Total model layers processed by device d_m .
$\frac{l_m^{\rm gpu}}{L}$	Total GPU layers processed by device d_m .
L	Total number of model layers.
\overline{W}	Total layer window size across all devices $(W = \sum_{m=1}^{M} w_m)$.
h_k, h_v	Number of attention heads for keys and values.
e_k, e_v	Embedding size per attention head.
\overline{e}	Embedding size.
b, b_i, b_o	Bytes of weight data for each layer, input, and output.
n_{kv}	Number of tokens stored in key-value cache.
\overline{V}	Vocabulary size.
$d_m^{ m avail}$	Available memory on device d_m .
$c^{\text{cpu}}, c^{\text{gpu}}$	Buffer sizes for CPU/GPU computations.
$egin{array}{c} d_m^{ ext{avail}} & & & & & & & & & & & & & & & & & & $	Disk read throughput for device d_m .
S ^{disk} threshold	A threshold for disk speed. If the disk speed is below this threshold, it is
	considered too slow.
$\overline{\mathcal{M}_1,\mathcal{M}_2,\mathcal{M}_3,\mathcal{M}_4}$	Set assignments, corresponding to cases 1-4.
a, b, c	Coefficient vectors for the objective function.
$oldsymbol{P}_w, P_n$	Constraint coefficients for w_m and n_m , should be diagonal matrices.
$egin{aligned} oldsymbol{P_n^{ ext{gpu}}} \ oldsymbol{w}', oldsymbol{n}' \end{aligned}$	A diagonal binary matrix that indicates whether a device uses a GPU.
$oldsymbol{w}', \overline{oldsymbol{n}'}$	Extended vectors for w and n .
$oldsymbol{z},oldsymbol{z}^{ ext{gpu}}$	Vectors of RAM/VRAM upper bounds for constraints.

Table 5 summarizes the key symbols used in this paper.

A.4 Run Prima.cpp with More Hot Models: Llama, Qwen, QwQ and DeepSeek

Figure 7 provides a clear comparison, showing that for models larger than 30B, our prima.cpp always has the lowest token latency and TTFT. We also support Qwen-2.5, QwQ, and DeepSeek-R1 in prima.cpp. The results are given in Table 6.

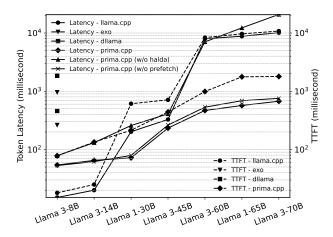


Figure 7: Comparison of token latency and TTFT for llama.cpp, exo, dllama, and prima.cpp.

A.5 Select Devices to Build the Most Powerful Cluster

Existing systems require the cluster with enough RAM/VRAM, users have to gather more devices to run larger models. However, collecting enough devices is challenging. Here we raise two questions:

Table 6: Token latency (in millisecond/token) for Qwen, QwQ and distilled DeepSeek R1 models.

Model	llama.cpp	exo ¹	dllama	prima.cpp
Qwen-2.5-7B	14	86	-	44
DeepSeek-R1-Distill-Qwen-7B	14	68^{2}	-	52
DeepSeek-R1-Distill-Llama-8B	14	77^{2}	435	59
Qwen-2.5-14B	23	31710^{3}	-	65
DeepSeek-R1-Distill-Qwen-14B	24	23475^3	-	76
Qwen-2.5-32B and QwQ-32B	224	OOM	-	89
DeepSeek-R1-Distill-Qwen-32B	232	OOM	-	93
DeepSeek-R1-Distill-Llama-70B	10978	OOM	-	724
Qwen-2.5-72B	12227	OOM	-	867

¹ Exo supports only the MLX backend for these models, so only D1 works.

(a) should we collect enough devices to meet the model's needs? (b) Is more devices always better? The answer is no. Figure 8 shows how layer assignment adjusts and token latency varies on Llama 3-70B as devices reduce from 6 to 1.

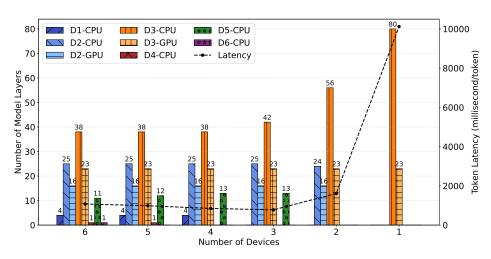


Figure 8: Layer assignment and token latency over different number of devices.

For question (a), with only 3 devices (D2, D3, and D5), the total available RAM+VRAM is 38 GiB, which is insufficient to hold the 40 GiB Llama 3-70B (Q4K) model. However, thanks to the fast SSDs on D2 and D3, mmap can swap model layers in a flash, and prima.cpp achieves the lowest latency. Thus, prima.cpp does not require enough memory to hold the entire model.

For question (b), when we increase the number of devices to 6, the total available RAM+VRAM reaches 50 GiB, enough to hold the entire model. However, token latency is lower with just 3 devices, as the additional devices (D4 and D6) have weak CPUs and slow disks, creating bottlenecks. This shows that more devices do not always result in faster inference.

This raises a new question: (c) If user has a device with a weak CPU or a slow disk, should it be added to the cluster? Intuitively, such a weak device would be a bottleneck. However, in cases of severe memory shortage, it actually help. For example, D2 and D3 are devices with a GPU, strong CPU, and fast disk, while D5 is a weak device. As shown in Figure 8, adding D5 reduced token latency by half, as the disk loading latency on D3 (which was heavily overloaded) became more problematic than D5's computing. This raises more complex questions: if users have some weak devices, which ones should be added? More generally, (d) given a set of heterogeneous devices, how can we select a subset to build the best-performing cluster? This is challenging due to the uncertain number of devices to be selected, the highly heterogeneous cluster, and the various factors like CPU, GPU,

² Latency decreases because exo provides full-precision Qwen models but 3-bit quantized DeepSeek models.

Latency spikes because exo is swapping data in/out from disk.

RAM, VRAM, disk, network, and even OS that significantly affect inference speed. Fortunately, prima.cpp offers an easy solution: start by including all devices in the cluster, then remove those with only one assigned layer or fewer than a set threshold, as Halda identifies them as drags. Future updates will automate this process for easier to use.

A.6 Efficient Workload Distribution and Memory Usage

Figure 9 shows each device's RAM and VRAM usage to illustrate why prima.cpp achieves faster speed and prevents OOM. As exo and dllama don't support Llama 14B-65B and encounter OOM at 70B, the memory usage for 14B-70B in Figures 9b and 9c is estimated based on system behavior and memory load at 8B. Figure 9a has been discussed in Section 4.1. Figures 9b and 9c show that exo and dllama consume high memory. Exo mixes multiple backends, using MLX on macOS for 4-bit computation and Tinygrad on Linux, where model weights load in 16-bit on the CPU and decoded to 32-bit on the GPU. In our case, D1(8 GiB UMA RAM) and D2 (8 GiB VRAM) get the same number of model layers, yet D2-CPU uses 4× more RAM and D2-GPU 8× more VRAM than D1-GPU. This results in high memory usage on Linux devices, increasing the risk of OOM. For dllama, it uses tensor parallelism and Q40 quantization to distribute and compress memory usage but lacks GPU support, so all memory load is on RAM and inference speed is limited. It has similar memory usage across devices due to its uniform tensor splitting, which causes problems on low-memory devices. In Figure 9c, when running a 30B model, D2/D3 have more available RAM, while D1/D4 have less. To allocate enough memory, D1/D4 must free more active pages or swap out app data, slowing user apps or even the system. In such cases, OOM might be the better outcome. Additionally, D3 (the head device) loads the entire model before slicing and distributing it, taking significant RAM and making it more prone to OOM.

In contrast, prima.cpp optimizes workload distribution with Halda and prevents memory waste with mmap. Though the solution to (1-5) is unobvious, we can observe Halda's preference from Figure 9d: powerful GPUs > weak GPUs > powerful CPUs > fast disks. For example, at 8B–30B, Halda first fills D2-GPU and D3-GPU. At 45-65B, it fills D1-CPU to D4-CPU. Lastly, the remaining layers are placed on D2-CPU and D3-CPU because they have fast disks. This assignment prevents weak CPUs and slow disks from being used. Finally, only D2-CPU and D3-CPU experience RAM overload, but this does not cause OOM because the OS will free inactive mmap-ed pages instantly and prefetch model layers in advance. With fast disk reads, disk loading latency stays low, ensuring minimal token latency, which is exactly the result of our optimization goal (1). Beyond the advanced workload distribution, prima.cpp also prevents memory waste. With mmap, it loads only required model layers instead of the full model, eliminating the need for model slicing. Additionally, it supports model inference in Q4K format across heterogeneous platforms, eliminating the need to decode back to 16/32-bit, so RAM/VRAM usage is further reduced.

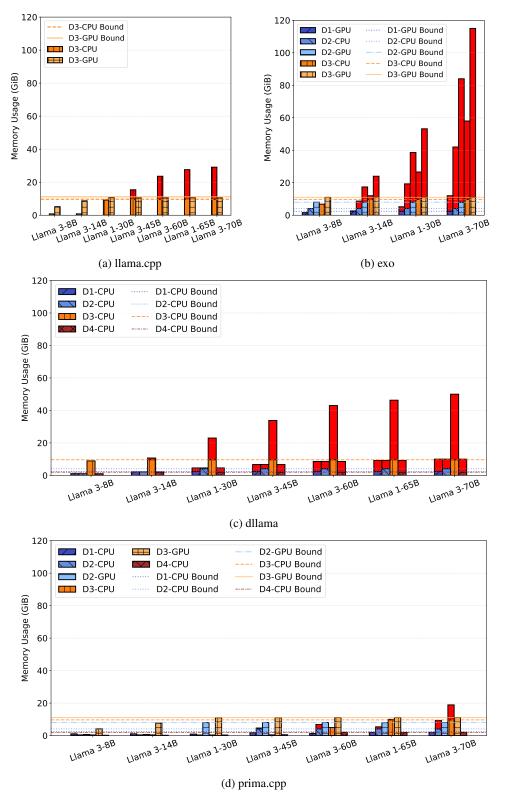


Figure 9: Comparison of memory usage on each device for llama.cpp, exo, dllama, and prima.cpp.