AUTOMATED TENSOR SCHEDULING FOR HYBRID CPU-GPU LLM INFERENCE ON CONSUMER DEVICES

Anonymous authors
Paper under double-blind review

ABSTRACT

Running large language models (LLMs) on consumer devices such as laptops and desktops is challenging because model weights often exceed GPU memory capacity, making offloading necessary to extend effective model capacity with CPU memory. However, existing offloading systems typically rely on coarse layer-level or expert-level scheduling, which overlooks substantial heterogeneity among tensors within the same layer and adapts poorly to changing hardware load conditions on such devices. In this paper, we design a novel system, called automated tensor scheduling for hybrid CPU-GPU LLM inference on consumer devices (ATSINFER), that performs offloading at tensor granularity. ATSINFER combines static tensor placement with load-aware dynamic transfer, and introduces asynchronous CPU-GPU coordination to efficiently schedule hardware storage, data movement, and computation across heterogeneous backends. We implement ATSINFER and evaluate it on representative consumer platforms using both dense and MoE models. Compared with existing systems, ATSINFER improves prefill throughput by up to 1.94× and decode throughput by up to 3.29×, while also increasing GPU utilization and making more effective use of PCIe bandwidth. These results show that ATSINFER can substantially improve the user experience of local LLM deployment on personal consumer devices.

1 INTRODUCTION

Local deployment of large language models (LLMs) supports privacy, offline availability, and interactive applications on laptops and desktops (Achiam et al., 2024; Yang et al., 2025; Liu et al., 2025; Meta, 2025). Unlike server serving, local inference usually runs at low concurrency, and hence per-request latency matters more than aggregate throughput. However, model weights often exceed consumer GPU memory even after quantization, making offloading necessary (Sheng et al., 2023).

GPU-centric offloading keeps computation on the GPU and fetches weights from host memory (Sheng et al., 2023; Hugging Face, 2022; Rasley et al., 2020; Kwon et al., 2023; Zheng et al., 2024). Its performance is constrained by PCIe bandwidth which is much lower than on-device GPU memory bandwidth. To reduce PCIe traffic, hybrid CPU-GPU inference is proposed to execute some operators on the CPU, replacing repeated weight transfers with smaller activation transfers (Gerganov, 2023; Na et al., 2025; Song et al., 2024; Alizadeh et al., 2024; Chen et al., 2025).

Existing hybrid CPU-GPU inference systems commonly place weights at layer granularity, or at expert granularity for Mixture-of-Experts (MoE) models (Gerganov, 2023; Sheng et al., 2023; Chen et al., 2025; Zhong et al., 2025b; Zhu et al., 2026; Kamahori et al., 2025; Cao et al., 2025; Song et al., 2025; Xue et al., 2025; Zhong et al., 2025a). Operators within a layer or expert differ in size and backend efficiency, and hence their benefit from GPU residency is not uniform. Furthermore, scheduling policies in existing hybrid CPU-GPU inference systems are often fixed or respond primarily to sequence length and expert activation. But consumer devices also experience changing CPU, GPU, and PCIe pressure from co-running applications and thermal limits. These changes can invalidate an otherwise effective placement or transfer policy. In sum, existing hybrid CPU-GPU inference systems typically rely on coarse layer-level or expert-level scheduling, which overlooks substantial heterogeneity among tensors within the same layer and adapts poorly to changing hardware load conditions on such devices.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To solve these problems, in this paper we design a novel system, called automated tensor scheduling for hybrid CPU-GPU LLM inference on consumer devices (ATSINFER), that performs offloading at tensor granularity. The contributions of this paper are outlined as follows:

- We conduct an empirical analysis of tensor heterogeneity and hardware-load variation in hybrid CPU-GPU LLM inference on consumer devices, which motivates the necessity of offloading at tensor granularity.
- In ATSINFER, a novel tensor-granularity placement strategy is proposed to combine static placement with load-aware dynamic transfer.
- In ATSINFER, a novel asynchronous CPU-GPU coordination mechanism is proposed to efficiently schedule hardware storage, data movement, and computation across heterogeneous backends.
- We implement ATSINFER by extending llama.cpp with approximately 15,000 lines of C++ code. ATSINFER supports both dense and MoE models.
- ATSINFER outperforms other widely used systems including llama.cpp (Gerganov, 2023), vLLM (Kwon et al., 2023), and KTransformers (Chen et al., 2025). In particular, ATSINFER improves prefill throughput by up to 1.94× and decode throughput by up to 3.29× over llama.cpp under the same VRAM budget.

2 BACKGROUND

Inference on consumer devices. Local inference on consumer devices must execute both prefill and decode on the same device. Server systems can separate these phases and optimize for large batches and concurrent requests (Kwon et al., 2023; Zheng et al., 2024; Patel et al., 2024; Qin et al., 2025). At batch size one which is a typical setting on consumer devices, model weights dominate storage in the workload. Furthermore, the KV-cache stays with the backend executing the corresponding attention operator to avoid additional transfers.

Prefill processes many prompt tokens together and exposes parallel matrix multiplication, whereas autoregressive decode processes one new token per step with less parallel work. Their different compute demands and transfer-overlap opportunities motivate phase-specific scheduling for interactive local deployment.

Heterogeneous execution. Consumer CPU kernels exploit AVX-family SIMD extensions, whereas AMX-optimized kernels in some systems target server-class processors (Intel, 2026b;a;c; Chen et al., 2025). These differences make measured backend latency more useful than tensor size alone when deciding where an operator should execute. GPU acceleration must also compensate for any additional activation or weight transfers.

3 ATSINFER DESIGN

3.1 MOTIVATION

We first conduct an empirical analysis of tensor heterogeneity and hardware-load variation in hybrid CPU-GPU LLM inference on consumer devices, which motivates the necessity of offloading at tensor granularity in ATSINFER.

Tensor heterogeneity. Figure 1 compares major weight tensors in Qwen3-14B and Qwen3-30B-A3B. We can find that tensors of comparable size can have different CPU and GPU execution times because their operators differ in compute intensity, memory access, and kernel efficiency. Consequently, moving an entire layer or expert to the GPU can spend scarce memory on tensors with little latency benefit. Placement should account for the latency reduction per byte and the activation-transfer cost introduced by fragmented backend assignments.

For example, the gate and up projections in the dense model have much larger CPU execution times than the down projection, although all are substantial weight tensors. Normalizing latency by storage exposes the benefit of assigning each tensor to the GPU, which will be formalized in Section 3.4. An ordering by tensor size alone misses the backend-dependent benefit, while an ordering by benefit per

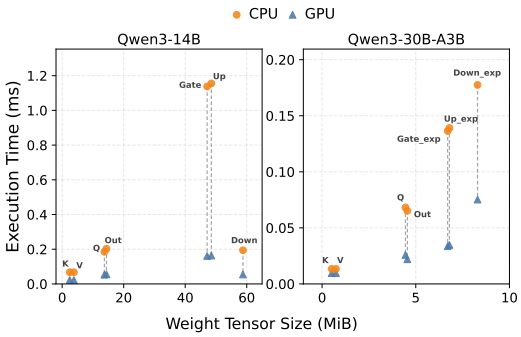


Figure 1: Tensor sizes and decode latencies on the laptop. MoE sizes count activated experts.

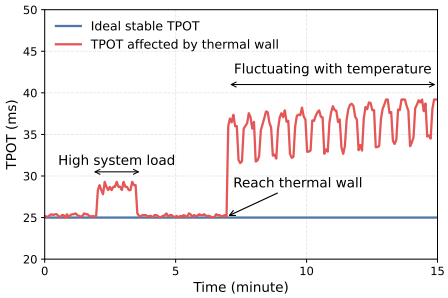


Figure 2: TPOT under background load and subsequent thermal throttling on a consumer device.

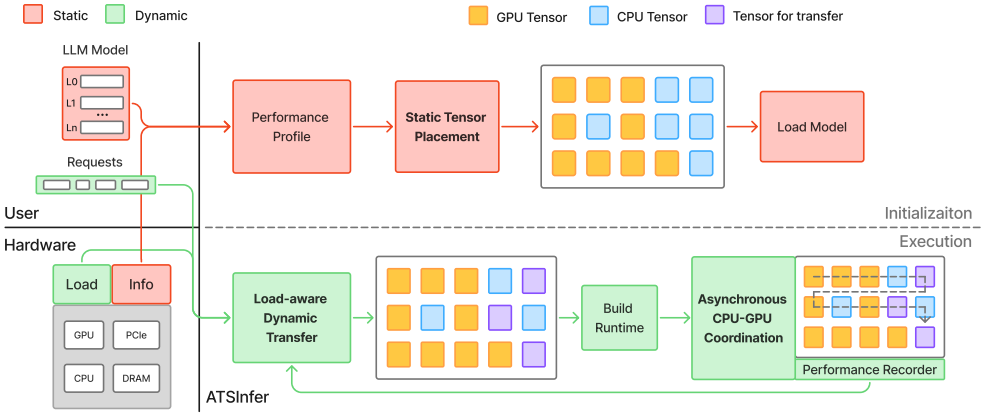
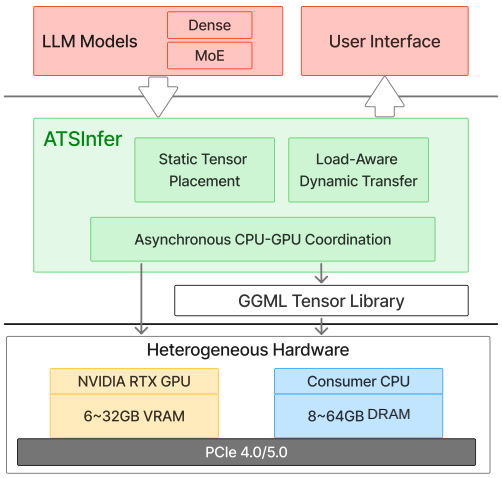


Figure 3: ATSINFER workflow. Initialization profiles execution and transfer costs and determines static residency. Online execution performs load-aware transfer and asynchronous CPU-GPU coordination, with performance feedback for later scheduling decisions.

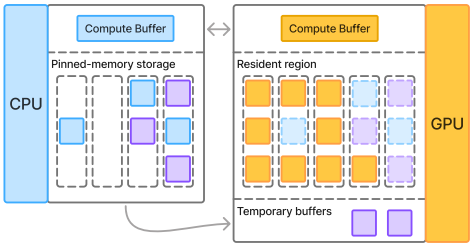
byte alone ignores the cost of switching backends between operators. These observations motivate a joint placement objective rather than independent per-tensor decisions.

Hardware-load variation. Input-aware expert caching can adapt to routing patterns (Zhong et al., 2025b; Cao et al., 2025; Song et al., 2025; Xue et al., 2025; Zhong et al., 2025a), but this does not directly address device contention or thermal throttling. Background applications can change the relative cost of CPU execution, GPU execution, and PCIe transfer during a request. Figure 2 shows the resulting variation in time per output token (TPOT), in which a period of background load raises latency and sustained execution later encounters a thermal limit. A profitable runtime decision should transfer more work when CPU pressure rises, and reduce promotion when GPU or PCIe pressure makes transfer unattractive. Section 4 evaluates this behavior under controlled load changes.

The distinction matters even when the request and model are unchanged. For example, a browser or other local application can consume resources after a schedule has been selected, and sustained generation can encounter a thermal limit. Adapting only to prompt length or expert activation does not capture these changes. The scheduler must also account for the cost of changing its own execution plan, since frequent reconfiguration can erase the benefit of responding to brief load fluctuations.



(a) System architecture and backends.



(b) Weight storage and reusable buffers.

Figure 4: ATSINFER implementation: the three scheduling components share CPU-GPU backends (a). static GPU weights, pinned CPU weights, and temporary GPU copies occupy separate regions (b). CPU and GPU compute buffers are managed by ggml.

3.2 SYSTEM OVERVIEW

Built on llama.cpp (Gerganov, 2023), ATSINFER adds a tensor-granularity execution path organized around static placement, load-aware dynamic transfer, and asynchronous CPU-GPU coordination. These components share the ggml tensor library and its heterogeneous backends, preserving support for dense and MoE models. Figure 3 and Figure 4 show the two-timescale workflow and its implementation.

During initialization, ATSINFER profiles per-operator CPU and GPU latency, records tensor sizes, and estimates transfer costs from measured bandwidth. The resulting placement is specialized to the target model, runtime, and hardware. It then loads the static GPU-resident set and allocates pinned host storage and temporary GPU buffers.

During online execution, the runtime executes a phase-specific plan with backend assignments, transfer events, and synchronization points, while recording computation and transfer times. When the re-scheduling condition is met, these measurements guide a new selection of CPU-resident tensors for temporary GPU execution. Thus static placement determines long-lived residency, while dynamic transfer adapts execution to changing device conditions without rerunning the optimizer for every token.

This organization separates the capacity decision from the execution decision. A CPU-resident weight keeps its host copy even when it is temporarily executed on the GPU, and promotion therefore does not require replacing the static resident set. Reusing temporary storage also prevents the memory demand of a round from growing with the total number of promoted tensors. The runtime assigns buffer space and synchronization points together so that a new transfer cannot overwrite weights still needed by an earlier operator.

Temporary storage is organized by tensor lifetime, and tensors in different layers can share a buffer when their live ranges do not overlap. This is common for tensors playing the same role in sequential layers. Pinned host storage supports asynchronous weight movement, while preallocated CPU and GPU compute buffers remain under ggml's management. Figure 4b makes these distinct storage roles explicit.

3.3 ASYNCHRONOUS CPU-GPU COORDINATION

CUDA operations within a stream execute in order and different streams can overlap when resources and dependencies permit (NVIDIA, 2026). Weight and activation transfers can nevertheless contend

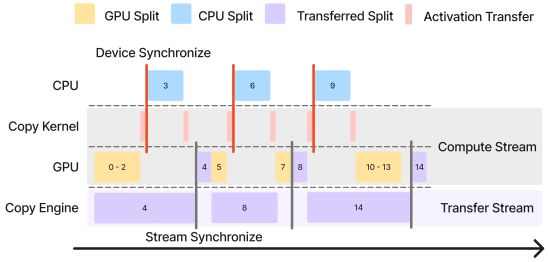


Figure 5: Execution pipeline: activation copies follow computation. A separate stream moves weights with explicit synchronization.

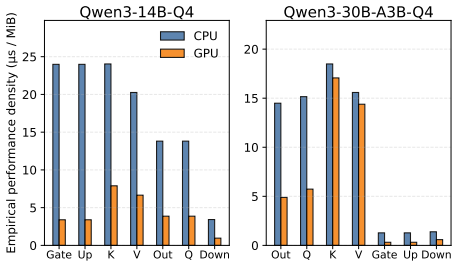


Figure 6: Decode performance density. Static MoE residency counts all experts, rather than only activated experts.

for the same copy-engine path. ATSINFER uses SM-driven copy kernels accessing mapped, pinned host memory for activations and a dedicated copy-engine stream for weights. This prevents short activation copies from waiting behind long weight transfers on the same engine, although both paths still share PCIe bandwidth.

Before an execution round, ATSINFER partitions the computation graph at backend switches and operators consuming temporary GPU tensors. A compute stream orders GPU operators and activation copies, while a transfer stream moves selected weights through the CUDA copy engine. Figure 5 shows the resulting pipeline, and detailed hardware transfer paths are provided in Appendix C.

Weight movement starts as soon as dependencies and temporary-buffer availability permit. For MoE models, expert transfer waits for routing so that only activated experts are moved. GPU events enforce cross-stream dependencies and GPU-to-CPU transitions synchronize the compute stream before the CPU consumes transferred activations. A temporary buffer is reused only after its prior consumer has finished. These rules preserve execution order while exposing opportunities to overlap communication and computation.

3.4 STATIC TENSOR PLACEMENT

For tensor i, let s_i be its memory footprint and t_i^c, t_i^g the profiled CPU and GPU operator latencies. The empirical performance density is $k_i^b = t_i^b/s_i$ for backend $b \in \{c, g\}$. The residency benefit is $r_i = t_i^c - t_i^g$, and $k_i^c - k_i^g = r_i/s_i$ is the saved latency per unit of resident memory. Measured latency captures both hardware and kernel behavior, a tensor's size alone does not determine the value of keeping it on the GPU. Figure 6 compares these densities. Appendix A gives the placement pseudocode.

Let $b_i \in \{\text{CPU}, \text{GPU}\}$ be the static backend and M the memory budget for resident weights. A backend switch between adjacent operators incurs activation-transfer cost $c_i = S_{\text{in},i}/B_{\text{pcie}}$, estimated from input size and measured bandwidth. Static placement maximizes saved execution time minus switching cost:

$$\max_b \sum_{i=1}^{n} r_i \mathbf{1}\{b_i = \text{GPU}\} - \sum_{i=2}^{n} c_i \mathbf{1}\{b_i \neq b_{i-1}\}, \quad \sum_{i=1}^{n} s_i \mathbf{1}\{b_i = \text{GPU}\} \leq M. \tag{1}$$

For dense models, a knapsack-style dynamic program tracks memory usage and the preceding backend, and backtracking recovers the placement. The implementation discretizes memory at MB granularity. With Q memory-budget bins, the solver requires $O(nQ)$ time and space and runs during initialization.

To make the switching penalty explicit, let $F_i(q, b)$ be the best saved latency for the first i tensors, using q memory bins and ending on backend b. Let $\hat{s}_i$ be the discretized tensor size and $x_b = \mathbf{1}\{b = \text{GPU}\}$. The recurrence is

$$F_i(q, b) = r_i x_b + \max_{b' \in \{\text{CPU},\text{GPU}\}} \left[F_{i-1}(q - \hat{s}_i x_b, b') - c_i \mathbf{1}\{b \neq b'\}\right]. \tag{2}$$

We set $c_1 = 0$, initialize $F_0(0, b) = 0$, and treat other initial states and out-of-budget states as infeasible. The final value is $\max_{q \le Q, b} F_n(q, b)$. Keeping both ending backends allows the solver to retain a locally less attractive assignment when it avoids a costly boundary later. Thus performance density explains the opportunity, while the dynamic program accounts for dependencies between neighboring assignments.

For MoE models, placement operates on all expert weights of an operator because routing is unknown at model-loading time. The solver first partitions expert and non-expert tensors. If memory permits, non-expert tensors remain on the GPU and the residual budget is allocated among expert tensors. Otherwise, expert tensors remain on the CPU and the budget is assigned to non-expert tensors. This residency footprint differs from the activated-expert transfer volume used online.

Non-expert weights are used irrespective of the routing outcome, whereas any particular expert is activated only for a subset of tokens. The partitioning policy reflects this asymmetry before applying the placement solver. Its result is a default residency plan, not a prediction of the next token's experts. At runtime, routing determines which portions of a CPU-resident expert tensor actually need to move.

3.5 Load-Aware Dynamic Transfer

Static placement avoids repeated transfers for high-value weights, but CPU-resident tensors can still dominate latency. Dynamic transfer selects some of these tensors for temporary GPU execution, using available overlap to hide weight movement. GPU-resident tensors retain their assignment. For each round, let rb_i denote the runtime backend, w_i the weight-transfer time, and $\mathcal{G} = \{i : b_i = \mathrm{CPU}, rb_i = \mathrm{GPU}\}$ the promoted set. For MoE decode, w_i includes only activated experts.

The scheduler estimates an overlap window from computation and activation transfers between a preceding CPU-side endpoint j and tensor i along the default path:

$$seg(j, i) = \sum_{k=j+1}^{i-1} t_k^{b_k} + \sum_{k=j+1}^{i-1} c_k \mathbf{1}\{b_{k-1} \neq b_k\}. \tag{3}$$

The exposed part of weight transfer is $\delta_i = \max(w_i - seg(j, i), 0)$. It minimizes the modeled round latency

$$T(rb) = \sum_{i=1}^{n} t_i^{rb_i} + \sum_{i=2}^{n} c_i \mathbf{1}\{rb_{i-1} \neq rb_i\} + \sum_{i \in \mathcal{G}} \delta_i. \tag{4}$$

Execution and transfer estimates are refreshed from runtime measurements. The dynamic program visits tensors in execution order, retains CPU- and GPU-ending states, and evaluates legal preceding transfer endpoints for each promotion. Its worst-case time is $O(n^2)$, full pseudocode and the overlap expression are in Appendix A.1. The model guides scheduling, while explicit runtime synchronization enforces dependencies and buffer lifetimes.

For a GPU-resident tensor, only the GPU-ending state is feasible. For a CPU-resident tensor, the scheduler compares retaining CPU execution with transferring the weight and executing on the GPU. Enumerating preceding endpoints accounts for the fact that two transfers of the same size can have different exposed costs depending on how much intervening work is available. If $w_i \le seg(j, i)$, the model hides the weight movement. Otherwise, the remaining delay must be justified by faster execution and any change in activation-transfer cost. The objective is an estimate of the execution pipeline, and does not assume that all PCIe traffic can overlap without contention.

Prefill generally offers more computation to hide transfer and thus favors more promotion. Decode has smaller overlap windows and benefits from selective promotion. Resource contention changes both decisions: a CPU slowdown increases the potential benefit of GPU execution, whereas slower GPU execution or PCIe transfer reduces it.

The dynamic program is not rerun for every token and re-scheduling requires both a sufficient measurement deviation and a minimum interval since the previous decision. Re-optimization itself has a cost. For Qwen3-14B on the RTX 3060, TPOT is approximately 40 ms, graph rebuilding takes 7 ms, and the dynamic program takes 9 ms. ATSInfer therefore re-schedules only when measurement deviation reaches 15% and at least five recent TPOTs have elapsed since the last re-scheduling

Table 1: Evaluation hardware. Both platforms use PCIe Gen4 x16.

	Laptop	Desktop
CPU	Intel i7-11800H	Intel i7-11700
GPU	RTX 3060 (6GB)	RTX 4090 (24GB)
RAM	32GB DDR4	64GB DDR4

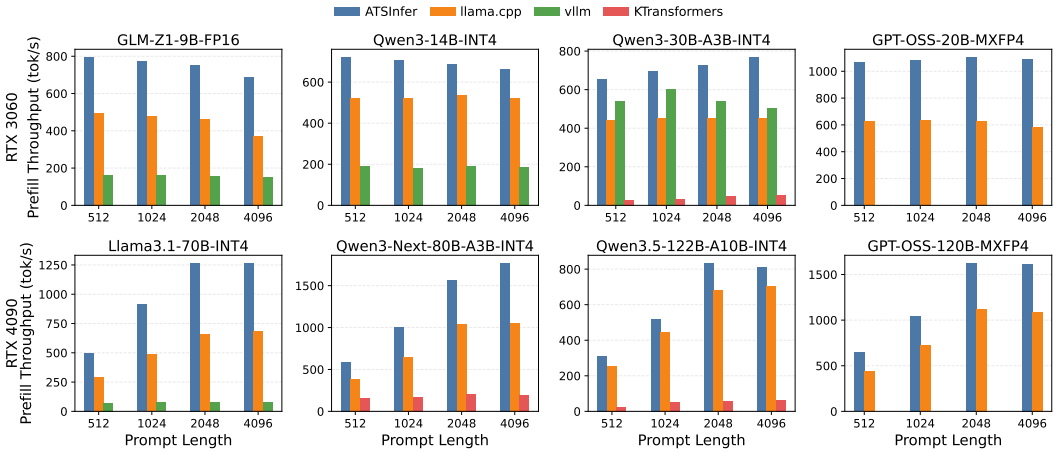


Figure 7: Prefill throughput for all eight models: RTX 3060 laptop (top row) and RTX 4090 desktop (bottom row). All systems use the same platform-specific chunk size; missing baseline bars indicate unsupported configurations.

event. This rate limit avoids repeatedly paying reconfiguration costs for short-lived fluctuations. Algorithm 3 in Appendix details the trigger.

4 Evaluation

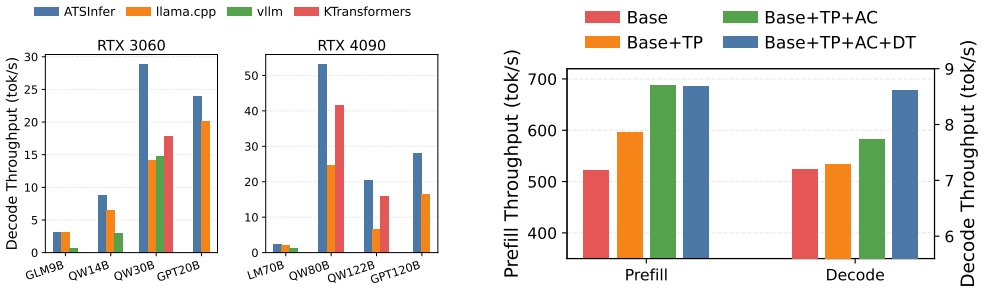
4.1 Experimental Setup

We evaluate at batch size one on the two platforms in Table 1. Laptop models are GLM-Z1-9B (Glm et al., 2024), Qwen3-14B, Qwen3-30B-A3B (Yang et al., 2025), and GPT-OSS-20B (Agarwal et al., 2025). Desktop models are Llama3.1-70B (Grattafiori et al., 2024), Qwen3-Next-80B-A3B, Qwen3.5-122B-A10B (Qwen Team, 2026), and GPT-OSS-120B. GLM-Z1-9B uses FP16, GPT-OSS uses native MXFP4, and the others use INT4.

Baselines are llama.cpp (Gerganov, 2023), vLLM v0.18.0 (Kwon et al., 2023), and KTransformers v0.5.2 (Chen et al., 2025). vLLM lacks offloading support for Qwen3-Next, Qwen3.5, and GPT-OSS; KTransformers supports selected MoE models and lacks MXFP4 CPU kernels. For each supported comparison, systems use the same model, prompt length, and KV-cache size, with maximal GPU-memory use and prefill chunks of 512 tokens on the laptop and 2048 on the desktop.

PowerInfer and baseline scope. PowerInfer (Song et al., 2024) combines neuron placement with predicted activation sparsity, but its public ReLU-family checkpoints and predictors do not cover our model suite. ATSInfer preserves the supplied operators and activations and supports dense and MoE execution without predictive neuron skipping. A shared-ReLU comparison would require separate speed and quality evaluation because sparse execution and backend differences confound scheduling.

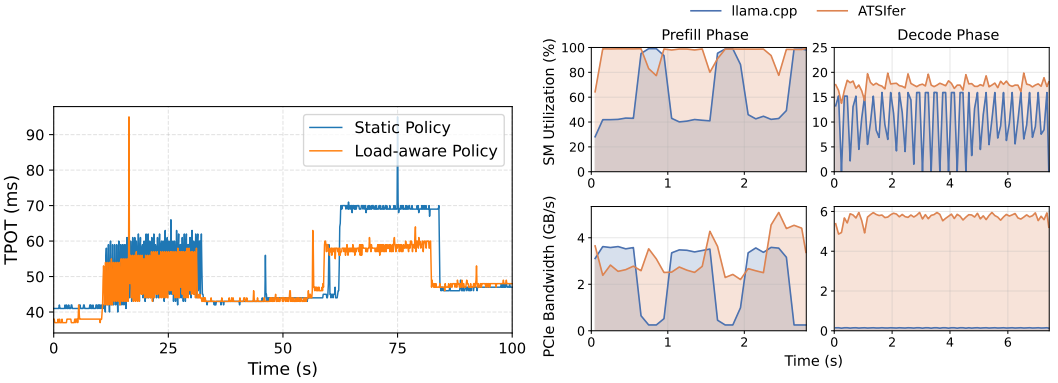
Prefill prompts range from 512 to 4096 tokens, with the main comparison summarized in Figure 7. Decode uses a 2048-token prompt and generates 128 tokens. Additional tests vary generation length from 32 to 2048 tokens and apply external resource pressure. Appendix B provides generation-length curves. Figure 9b uses NVIDIA Nsight Systems traces (NVIDIA, 2018). We report through-



(a) Decode throughput on RTX 3060 (left) and RTX 4090 (right).

(b) Qwen3-14B ablation.

Figure 8: Decode performance (a) and component contributions (b). Missing bars in (a) indicate unsupported configurations. TP: tensor placement; AC: asynchronous coordination; DT: dynamic transfer. Base is llama.cpp.



(a) Load adaptation.

(b) GPU SM utilization and effective PCIe bandwidth.

Figure 9: Load adaptation (a) and Nsight GPU/PCIe traces (b).

put in tokens/s and time per output token (TPOT). The ablation uses a 1024-token prompt and 128-token output.

4.2 End-to-End Performance

Prefill. Figure 7 reports results for all four models on each platform. Compared with llama.cpp, ATSInfer improves prefill throughput by 1.28–1.88× on the laptop and 1.15–1.94× on the desktop across the evaluated configurations.

Decode. Figure 8a reports decode throughput across both platforms. In these configurations, the per-step weight-transfer volumes are approximately 1.0 GB for Qwen3-30B-A3B, 4.9 GB for Qwen3-14B, and 13.5 GB for GLM-Z1-9B.

ATSInfer's maximum laptop decode speedups are 3.29× over llama.cpp, 4.35× over vLLM, and 3.15× over KTransformers. Desktop maxima are 3.12×, 2.03×, and 1.33×, respectively. These values are maxima over supported comparisons, not averages.

4.3 Adaptation and Component Analysis

Load adaptation. Figure 9a compares a manually tuned static policy with the online policy under external CPU and GPU/PCIe load. At 60% external CPU load, the online policy reduces the TPOT increase by 13% relative to the static policy; at 60% external GPU and PCIe load, it reduces the increase by 38%.

Ablation. Figure 8b incrementally adds static placement, asynchronous coordination, and dynamic transfer to llama.cpp, using a 1024-token prompt and 128-token output. For prefill, the configurations with placement and coordination, with and without dynamic transfer, achieve nearly identical throughput. For decode, throughput increases at each stage, with the largest increase occurring after dynamic transfer is added.

Hardware utilization. Figure 9b reports GPU SM utilization and PCIe bandwidth for llama.cpp and ATSINFER during prefill and decode. In the decode trace, ATSINFER's average GPU SM utilization is approximately 70% higher than llama.cpp's.

5 RELATED WORK

GPU-centric and hybrid inference. GPU-centric systems extend effective GPU capacity by moving model weights or runtime state through a CPU-GPU storage hierarchy. FlexGen, DeepSpeed Offload, and Hugging Face Accelerate focus on memory-limited execution while retaining the GPU as the primary compute backend (Sheng et al., 2023; Rasley et al., 2020; Hugging Face, 2022). vLLM and SGLang primarily target high-throughput server serving, where batching, KV-cache management, and request scheduling are central concerns (Kwon et al., 2023; Zheng et al., 2024). In these systems, performance depends strongly on host-device bandwidth and on how effectively transfers can be hidden behind GPU computation.

Hybrid runtimes allow the CPU to participate directly in inference. llama.cpp supports broad local deployment across commodity processors, while FlexInfer explores CPU-GPU cooperation and PowerInfer and LLM in a Flash exploit selective residency or sparsity to reduce data movement (Gerganov, 2023; Na et al., 2025; Song et al., 2024; Alizadeh et al., 2024). These systems establish the practicality of hybrid execution, but commonly use layer-level placement, fixed partitions, or coarse heuristics. ATSINFER instead uses tensors as the placement unit and combines an initialization-time resident set with runtime transfer decisions driven by measured device costs.

MoE scheduling. Mixture-of-Experts models expose additional scheduling opportunities because only a subset of experts is activated for each token. Existing runtimes distribute expert storage and execution across CPU and GPU, defer selected expert computation, or cache experts according to observed routing behavior (Chen et al., 2025; Zhong et al., 2025b; Cao et al., 2025; Zhu et al., 2026; Song et al., 2025; Xue et al., 2025; Zhong et al., 2025a). Several systems update expert residency online as the activated-expert distribution changes, reducing unnecessary movement and improving the GPU hit rate.

These approaches adapt primarily to input-dependent routing. ATSINFER addresses a different source of variation: changes in measured CPU execution, GPU execution, and transfer time while a request is running. It applies the same tensor-level placement and load-aware transfer formulation to dense and MoE models; for MoE decode, only the activated experts contribute to the online transfer estimate. Expert caching and routing prediction are therefore complementary to the hardware-load adaptation studied here.

Model reduction. Model compression reduces deployment cost from the model side. Low-bit quantization decreases parameter storage and transfer volume and is widely used for consumer inference (Xiao et al., 2023; Lin et al., 2024; Frantar et al., 2023). Sparsity, pruning, and sparse attention reduce the parameters, activations, or attention work involved in execution (Liu et al., 2023; Jiang et al., 2024), while knowledge distillation transfers capability to a smaller model (Hinton et al., 2015). These methods change the amount of storage or computation required but do not determine how the remaining work should be divided between CPU and GPU. They can therefore be combined with ATSINFER's heterogeneous scheduling rather than serving as alternatives to it.

6 CONCLUSION

In this paper, we present ATSINFER, a system for efficient hybrid CPU-GPU inference of large language models on consumer devices. By combining tensor-granularity static placement, load-aware dynamic transfer, and asynchronous CPU-GPU coordination, ATSINFER achieves substantially higher throughput than existing systems, with up to 1.94× higher prefill throughput and 3.29× higher decode throughput than llama.cpp.

AI USE STATEMENT

AI assistance was used to revise the manuscript's LaTeX formatting, reorganize figures and appendices, and edit the presentation of existing text.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. arXiv, 2024.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. arXiv, 2025.

Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 12562–12584, 2024.

Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 715–730, 2025.

Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, et al. Ktransformers: Unleashing the full potential of cpu/gpu hybrid inference for moe models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 1014–1029, 2025.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.

Georgi Gerganov. llama.cpp: LLM inference in C/C++. Software, 2023.

Team Glm, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools. arXiv, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. arXiv, 2024.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. arXiv, 2015.

Hugging Face. Accelerate: Training and inference at scale made easy. Software, 2022.

Intel. Intel advanced matrix extensions (AMX) overview. Intel technical documentation, 2026a.

Intel. Intel AVX-512 overview. Intel technical documentation, 2026b.

Intel. Intel processor claims. Intel technical documentation, 2026c.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.

Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: CPU-GPU orchestration for fast inference of mixture-of-experts models. In *International Conference on Learning Representations*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6: 87–100, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. arXiv, 2025.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 22137–22176, 2023.

Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. Meta AI Blog, 2025.

Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J. Hughes, Tushar Krishna, and Hyesoon Kim. Flexinfer: Flexible llm inference with cpu computations. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

NVIDIA. NVIDIA Nsight Systems. Software, 2018.

NVIDIA. CUDA C++ best practices guide. NVIDIA documentation, 2026.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, 2025.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents. Qwen Blog, February 2026.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116, 2023.

Xiaoniu Song, Zihang Zhong, Rong Chen, and Haibo Chen. ProMoE: Fast MoE-based LLM serving using proactive caching. arXiv, 2025.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, 2024.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pp. 38087–38099, 2023.

Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. MoE-Infinity: Efficient MoE inference on personal machines with sparsity-aware expert cache. arXiv, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, pp. 62557–62583, 2024.

Shuzhang Zhong, Ling Liang, Yuan Wang, Runsheng Wang, Ru Huang, and Meng Li. Adapmoe: Adaptive sensitivity-based expert gating and management for efficient moe inference. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2025a.

Shuzhang Zhong, Yanfan Sun, Ling Liang, Runsheng Wang, Ru Huang, and Meng Li. Hybrimoe: Hybrid cpu-gpu scheduling and cache management for efficient moe inference. In *2025 62nd ACM/IEEE Design Automation Conference*, pp. 1–7, 2025b.

Zeyu Zhu, Gang Li, Peisong Wang, Zitao Mo, Minnan Pei, Zhuoran Song, Xiaoyao Liang, and Jian Cheng. DALI: A workload-aware offloading framework for efficient MoE inference on local PCs. arXiv, 2026.

A PLACEMENT AND SCHEDULING ALGORITHMS

Figure 6 in the main text reports the empirical performance-density comparison discussed in Section 3.4. For static MoE residency, tensor size includes all experts in the operator, because the routed experts are unknown before loading. This differs from the activated-expert size in Figure 1 and the online transfer model. Algorithm 1 retains the full initialization procedure, including the expert/non-expert budget policy.

Algorithm 1 Static Tensor Placement

Input:
  model $\mathcal{M}$;
  GPU memory budget M
Output:
  placement decision b
$s \leftarrow \text{READTENSORSIZES}(\mathcal{M})$;
$G \leftarrow \text{BUILDCOMPUTATIONGRAPH}(\mathcal{M})$;
$(t^c, t^g, c) \leftarrow \text{PROFILE}(G)$;
$r \leftarrow t^c - t^g$;
if $\mathcal{M}$ is dense **then**
  $b \leftarrow \text{SOLVEKNAPSACKDP}(G, s, r, c, M)$;
else
  $(T_{\text{exp}}, T_{\text{nonexp}}) \leftarrow \text{PARTITIONTENSORS}(G)$;
  $s_{\text{nonexp}} \leftarrow \text{TOTALSIZE}(T_{\text{nonexp}}, s)$;
  if $M \geq s_{\text{nonexp}}$ **then**
    $b_{\text{nonexp}} \leftarrow \text{GPU}$;
    $b_{\text{exp}} \leftarrow \text{SOLVEKNAPSACKDP}(T_{\text{exp}}, s, r, c, M - s_{\text{nonexp}})$;
  else
    $b_{\text{exp}} \leftarrow \text{CPU}$;
    $b_{\text{nonexp}} \leftarrow \text{SOLVEKNAPSACKDP}(T_{\text{nonexp}}, s, r, c, M)$;
  end if
end if
Return b

A.1 DYNAMIC TRANSFER

For a preceding CPU-side endpoint j and candidate tensor i, the scheduler estimates the overlap window along the default path as

$$seg(j,i) = \sum_{k=j+1}^{i-1} t_k^{b_k} + \sum_{k=j+1}^{i-1} c_k \mathbf{1}\{b_{k-1} \neq b_k\}.$$

Algorithm 2 retains the dynamic-programming procedure. The weight-transfer cost w_i for MoE decode includes only activated experts. In the worst case, each CPU-resident tensor considers all legal preceding endpoints, giving $O(n^2)$ time. Explicit interval storage requires $O(n^2)$ space; retaining only states and backtracking information reduces auxiliary state to $O(n)$.

Algorithm 2 Dynamic Transfer Scheduling

Input:
  default placement b;
  GPU/CPU execution times t^g, t^c;
  activation-transfer costs c;
  weight-transfer times w
Output:
  runtime backend assignment rb
$seg[j,i] \leftarrow \sum_{k=j+1}^{i-1} t_k^{b_k} + \sum_{k=j+1}^{i-1} c_k \, \mathbf{1}\{b_{k-1} \neq b_k\}$;
Initialize dynamic-programming states;
for $i = 1$ to n **do**
  if $b_i = \mathrm{GPU}$ **then**
    Update $dp[i, \mathrm{GPU}]$ by extending $dp[i-1]$;
    Mark $dp[i, \mathrm{CPU}]$ as infeasible;
  else
    Update $dp[i, \mathrm{CPU}]$ by extending $dp[i-1]$;
    $best \leftarrow \min_{\substack{j<i \\ b_j = \mathrm{CPU}}} \big(dp[j, \mathrm{CPU}] + \max\big(w_i, seg[j,i]\big)\big)$;
    $dp[i, \mathrm{GPU}] \leftarrow best + c_i \, \mathbf{1}\{b_{i-1} = \mathrm{GPU}\} + t_i^g$;
  end if
end for
$(T^\star, rb^\star) \leftarrow \arg\min_{rb \in \{\mathrm{GPU},\mathrm{CPU}\}} dp[n, rb]$;
Recover $rb_1, \ldots, rb_n$ by backtracking;
Return rb

A.2 RE-SCHEDULING TRIGGER

Algorithm 3 applies the deviation threshold and minimum interval before rebuilding the execution plan. The implementation uses $\epsilon = 15\%$ and a minimum interval of five recent TPOTs. On Qwen3-14B with RTX 3060, the reported graph-rebuild and scheduling times are approximately 7 ms and 9 ms, compared with approximately 40 ms TPOT; these costs motivate the rate limit.

Algorithm 3 Load-aware Re-scheduling

Input:
  previous measurement snapshot m^{prev};
  current measurement snapshot m^{cur};
  deviation threshold ϵ;
  minimum re-scheduling interval τ;
  time of last re-scheduling t^{last};
  default placement b
Output:
  runtime backend assignment rb
$d \leftarrow \text{COMPUTEDEVIATION}(m^{prev}, m^{cur})$;
if $d < \epsilon$ **or** $t^{now} - t^{last} < \tau$ **then**
  Return previous assignment
else
  $(t^g, t^c, c, w) \leftarrow m^{cur}$;
  $rb \leftarrow \text{DYNAMICTRANSFERSCHEDULING}(b, t^g, t^c, c, w)$;
  $\text{BUILDRUNTIME}(rb)$;
  Update $m^{prev} \leftarrow m^{cur}$ and $t^{last} \leftarrow t^{now}$;
  Return rb
end if

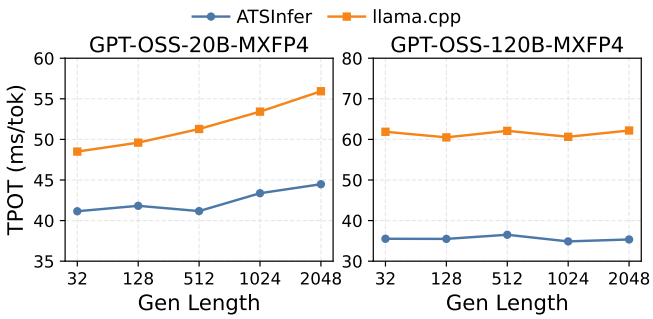


Figure 10: Average decode TPOT versus generation length for GPT-OSS on the laptop and desktop.

B Additional Experimental Results

The complete prefill comparison is presented in Figure 7, and the hardware-utilization traces are now paired with the load-adaptation result in Figure 9. Here we report the complementary sustained-generation measurement described in Section 4.

Sustained generation. Figure 10 reports average TPOT as generation length increases. Both llama.cpp and ATSInfer show rising TPOT on the thermally constrained laptop, whereas the desktop is more stable. This sustained-generation behavior is consistent with the thermal effect in Figure 2.

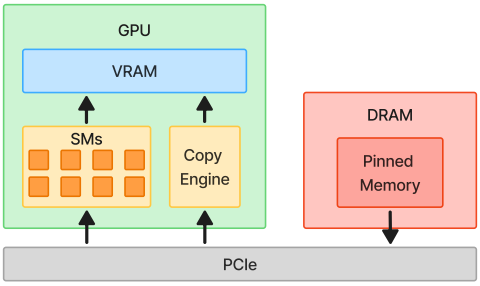


Figure 11: Host-device transfer paths. Weight copies use the CUDA copy engine, while SM-driven copy kernels move activations; both paths share PCIe bandwidth.

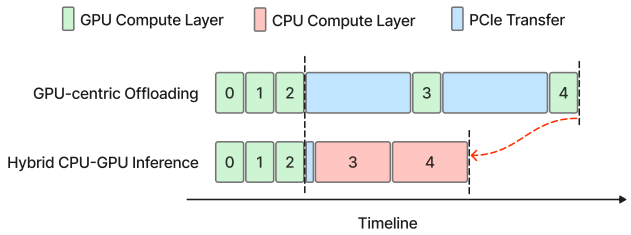


Figure 12: GPU-centric offloading replaces CPU execution with weight transfers; hybrid execution reduces traffic but can expose CPU latency.

C Supporting Execution Diagrams

The main text presents the original workflow, backend architecture, memory organization, and execution pipeline in Figures 3–5. Here we provide the hardware transfer paths supporting Section 3.3 and the conceptual offloading timeline discussed in Section 1.

Transfer engines. Mapped pinned host memory makes CPU storage accessible to the SM-driven activation-copy path. Separating this path from copy-engine weight transfers prevents small activation copies from waiting behind long weight copies on the same engine. The streams still require synchronization at data dependencies and remain subject to shared PCIe capacity.

Bottleneck shift. Figure 12 illustrates why minimizing transfer volume alone does not minimize inference latency. Hybrid execution substitutes CPU computation and small activation transfers for large weight transfers, reducing PCIe traffic but potentially extending the critical path through slow CPU operators.