Sandwich: Joint Configuration Search and Hot-Switching for Efficient CPU LLM Serving

Juntao Zhao*
The University of Hong Kong
Hong Kong, China

Jiuru Li*
The University of Hong Kong
Hong Kong, China

Chuan Wu
The University of Hong Kong
Hong Kong, China

Abstract

CPUs are critical for LLM serving due to their availability, cost efficiency, and edge applicability. However, efficient CPU serving is hindered by conflicting prefill/decode resource demands under non-disaggregated deployment constraints—existing solutions fail to avoid cross-phase interference, ignore sub-NUMA hardware structures, and deliver suboptimal dynamic-shape kernel performance. We propose Sandwich, a full-stack CPU LLM serving system with three core innovations addressing these challenges: (1) seamless phase-wise plan switching to eliminate cross-phase interference; (2) TopoTree, a tree-based hardware abstraction for automated substructure-aware (e.g., LLC slices) partial core allocation; (3) fast-start-then-finetune dynamic-shape tensor program generation. Evaluated on five x86/ARM CPU platforms, Sandwich achieves average 2.01× end-to-end speedup and up to 3.40× latency reduction over SOTA systems. Its kernels match static compiler performance with three orders of magnitude less tuning cost.

ACM Reference Format:
Juntao Zhao, Jiuru Li, and Chuan Wu. 2026. Sandwich: Joint Configuration Search and Hot-Switching for Efficient CPU LLM Serving. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803931

1 Introduction

The scarcity of high-end GPUs [38], combined with the demand for greater availability and flexibility in data centers [12, 32], has increased interest in using CPUs for language model deployment, owing to their widespread availability and cost efficiency. The emergence of capable small- and medium-scale language models (SLMs) [2] for specialized tasks, along with power-constrained edge environments [35], further solidifies the CPU's role as a vital deployment platform.

Efficient CPU-based LLM serving faces two primary challenges. As prior work [43] establishes, LLM inference consists of two phases with divergent resource demands: a compute-intensive *prefill* phase that processes variable-length inputs, and a memory-intensive *decode* phase that is bottlenecked by memory bandwidth and cache contention. First, strict memory constraints in many CPU deployment scenarios (e.g., edge devices) mandate **non-disaggregated** generation: prefill and decode must be colocated within a single instance to avoid redundant model weight storage, which amplifies the need to unify optimizations for their conflicting resource requirements. At the same time, modern server CPUs exhibit complex hierarchical memory architectures featuring Non-Uniform Memory Access (NUMA) and shared cache hierarchies (e.g., multiple Last-Level caches (LLCs)). Thus, low-latency efficient serving requires four key capabilities: (1) optimize variable-length prefill tensor programs, (2) alleviate decode's memory bottlenecks, (3) align optimizations with underlying CPU hardware, and (4) minimize prefill-decode cross-phase interference in colocated instances.

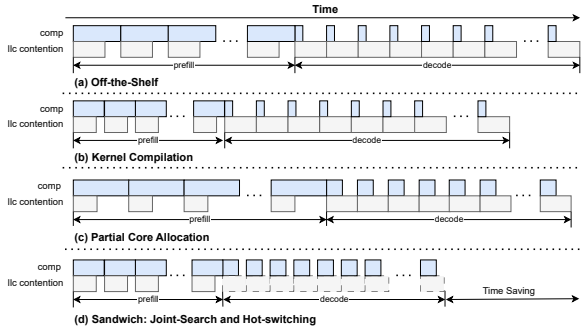


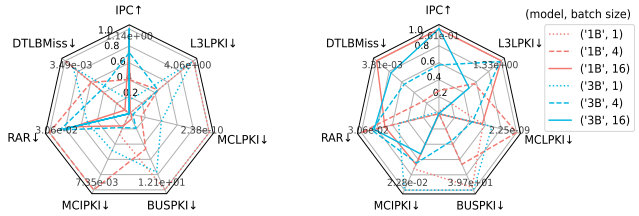
Figure 1: Execution time comparison of CPU serving solutions. The prefill stage benefits from kernel compilation due to higher kernel efficiency, while the decode stage improves with partial core allocation, which reduces last-level cache (LLC) contention. By dynamically switching computation graphs, Sandwich avoids interference from partial core allocation, enabling fine-grained control and better performance (dashed line) than either approach alone.

Existing systems like CPU backend of vLLM [22], llama.cpp [11], and xFasterTransformer [14]—often rely on vendor-specific or hand-tuned solutions with limited adaptability to dynamic shapes. As shown in Fig. 1(b), automatic tensor program schedulers like TVM [6] can generate dynamic-shape results, but require substantial auto-tuning effort. To reduce search overhead, recent dynamic-shape compilers [36, 39, 44] encode hardware primitives into micro-kernels (MKs), which are programs that partially compute results, and apply *polymerization schemes*, which define a selection of MKs to use and how they are assigned to computing resources under different input shapes.

On the other hand, since CPUs offer fine-grained core control via APIs like OpenMP [25], several serving systems [22] advocate partial core allocation to limit parallelism and reduce memory contention during the memory-bound decode phase [5], as illustrated in Fig. 1(c). To avoid cross-NUMA memory accesses [19], many systems employ NUMA-aware thread scheduling [11, 16, 20] or partition models evenly across NUMA nodes [14].

*Equal contribution.

This work is licensed under a Creative Commons Attribution 4.0 International License.
DAC '26, Long Beach, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803931



(a) Prefill **(b) Decode**

Figure 2: Prefill and decode workload characterization using different performance counters for serving Llama3.2-1B and Llama3.2-3B.

Despite these optimizations, current approaches remain suboptimal for CPU-based serving. For the decode phase, partial core allocation reduces prefill parallelism, hurting its performance. Moreover, the derived core allocation schemes often ignore sub-NUMA structures, such as the four-core clusters sharing an LLC slice in AMD EPYC 7H11 processors [1]. For the polymerization scheme at prefill, dynamic-shape compilers either adopt a scale-up-and-out strategy [44], i.e., expanding MK shapes greedily per cache level and partitioning across processors, or use a cost-model-based approach with fixed-size MKs and runtime polymerization. While larger MKs improve kernel efficiency, they reduce parallelizability, constraining the polymerization search space. Existing methods fail to model this trade-off, leading to suboptimal performance.

To address these challenges, we propose Sandwich, a full-stack LLM serving system for CPUs that enables program hot-switching between prefill and decode phases and efficiently explores the combinatorial design space of core allocations and dynamic-shape inputs. As illustrated in Fig. 1(d), for core allocation, Sandwich represents CPU topology as a tree (TopoTree), systematically enumerating core allocation plans that account for shared hardware resources like LLC slices. For tensor program generation, Sandwich produces multiple MK shapes and explores polymerization schemes via a fast-start-then-finetune strategy, jointly optimizing computation slices and parallelization plans. Our contributions are:

▹ We design and implement a runtime hot-switching mechanism for CPU-based LLM serving, enabling separate execution plans for prefill and decode phases without requiring duplicate model copies.

▹ We introduce TopoTree, a tree-based hardware abstraction that uses grouping and removal transformations to efficiently explore core allocation strategies, maximizing synergy and minimizing resource contention.

▹ We develop a fast-start-then-finetune method for generating dynamic-shape tensor programs, which jointly optimizes MKs and polymerization schemes. This reduces tuning overhead and improves prefill kernel performance.

▹ We evaluate Sandwich extensively on multiple CPU platforms (Xeon Gold 6151, 6230, Platinum 8272CL, EPYC 7H12, and Kunpeng 920) using diverse chatbot traces. Sandwich reduces latency by up to 3.40× and achieves an average 2.01× end-to-end speedup over the best existing systems, while matching TVM's kernel performance with three orders of magnitude less tuning cost.

2 Background

2.1 Prefill-decode workload difference in CPU

To quantify the distinct workload characteristics of prefill and decode phases on many-core CPUs, we adopt performance metrics from [9, 26]: Instructions Per Cycle (IPC), Last-level Cache Loads Per Thousand Instructions (L3LPKI), Data TLB Miss Ratio (DTLBMiss), Remote Access Ratio (RAR), Memory Controller Loads Per Thousand Instructions (MCLPKI), Memory Controller Imbalance (MCIPKI), and Bus Cycles Per Thousand Instructions (BUSPKI).

We evaluate BF16 Llama3.2-1B and 3B models on a dual-socket Intel Xeon Platinum 8275CL system (24 cores per socket, two NUMA nodes) using vLLM with NUMA-aware model partitioning. Requests are sampled from the ShareGPT dataset with batch sizes of 1, 4, and 16. As shown in Fig. 2, the prefill phase exhibits higher IPC and lower BUSPKI, indicating a compute-bound workload where memory accesses are spaced apart by computation, reducing pressure on memory controllers. In contrast, the decode phase shows lower IPC with significantly higher BUSPKI and MCLPKI, reflecting memory-bound behavior. Thus, prefill benefits from greater computational throughput, while decode requires memory contention mitigation.

2.2 CPU Autonomy

CPUs provide fine-grained core management through mechanisms like OpenMP, enabling precise control over core binding and affinity. As demonstrated in Sec. 2.1, memory bus congestion can be mitigated by strategically deactivating CPU cores, which improves end-to-end token generation throughput while reducing power consumption. Existing systems such as vLLM typically address this by evenly distributing models and computation across NUMA nodes, combined with limited core deactivation (e.g., 2 cores) to enhance overall efficiency.

2.3 Dynamic-shape Tensor Program Generation

Static-shape compilers [6] require unacceptable tuning times to produce dynamic shape tensor programs, because they need to tune every input shape. Dynamic-shape compilers construct tensor programs using *micro kernels*, which are single-threaded programs that cover certain tensor shapes, and *polymerization schemes*, which are schedules to place the computation slices onto parallel computation resources. They can be categorized into two approaches.

Scale-up-and-out [44] is a bottom-up approach that gradually increases the size of the current MK by repeating a smaller MK along some dimension and selecting the scale-up dimension that yields the most performance gain through profiling. When the performance improvement saturates at a cache level, the same process is repeated on the new MK. Eventually, a tiled MK is constructed for the LLC and it is replicated among CPU cores. A cost-model-based method [36, 39] is a top-down approach that uses fixed-shape pre-compiled MKs, whose polymerization schedules are determined at runtime by a cost model. Our observation shows that both approaches fail at providing the optimal tensor program.

3 Motivations

Current CPU-based LLM serving systems exhibit three key limitations: interference from partial core allocation, neglection of core

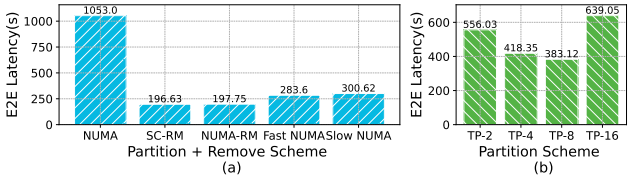
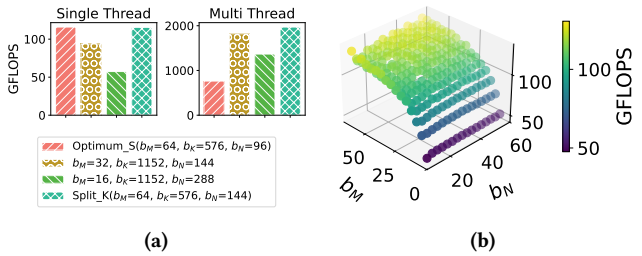


Figure 3: (a) Kunpeng: i) NUMA: using all the NUMA nodes; ii) SC-RM: removing one core from every 4 cores; iii) NUMA-RM: removing the same number of cores as SC-RM, but the last cores in each NUMA node; iv) Fast NUMA: using fast NUMA nodes only; v) Slow NUMA: using slow NUMA node; (b) AMD EPYC 7H21: comparison among different TP degrees.



(a) (b)

Figure 4: GEMM: (a) GFLOPS under different combinations of compute block and threading schemes; (b) Block size vs. GFLOPS when increasing b_M, b_N.

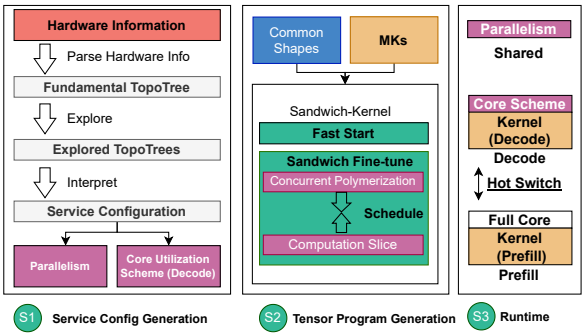


Figure 5: Workflow of Sandwich.

topology, and suboptimal dynamic-shape kernel performance. We detail these issues below.

Observation #1: Interference Between Prefill and Decode Phases Partial core allocation is optimized for memory-bound decode phases harms compute-intensive prefill performance by limiting parallelism. While disaggregation [43] alleviates this, it duplicates model weights—infeasible under memory constraints for common CPU environments (e.g., edge devices). A lightweight alternative that avoids this overhead is necessary.

Observation #2: Core Topology and Placement Affect Performance Identical core counts yield significantly different performance depending on topology. Fig. 3(a) compares core allocation strategies on Kunpeng-920 (8 NUMA nodes, 192 cores). Removing one core per LLC cluster (SC-RM) achieves 196.63s latency—outperforming NUMA-level removal (197.75s) and fast/slow NUMA selection (283.5s/300.62s). This confirms that sub-NUMA structures (e.g., LLC slices, as shown in Fig. 6) significantly impact performance, yet remain invisible to standard tools like "lstopo".

Similarly, optimizing data locality *within NUMA nodes* improves efficiency. Fig. 3(b) shows TP-8 achieving lowest latency (383.12s) on EPYC 7H12 by aligning model partitions with LLC groups (4 cores/share), reducing cross-group cache coherence overhead. TP-16 degrades performance (639.05s) due to excessive fragmentation.

Opportunity: Execution Graph Hot-Switching and Topology-Aware Core Allocation To address phase interference, Sandwich leverages CPU autonomy to dynamically switch execution graphs between partial and full core sets at runtime, eliminating interference without model duplication.

For topology-aware optimization, we introduce **TopoTree**—a mutable tree representation of all core utilization plans under hierarchical memory architectures. Through *group* and *remove* transformations, TopoTree systematically explores configurations where model partitions align with latent shared structures (e.g., LLC slices). This enables efficient search of core allocation strategies that maximize data locality while minimizing resource contention.

Observation #3: Limitations in Both Top-down and Bottom-up Tensor Program Generation We evaluate top-down (cost-model-based) and bottom-up (scale-up) methods using a two-level GEMM based on BLAS. Macro-kernel (b_M, b_N, b_K) targets per-core caches, while micro-kernel (μ_M, μ_N, μ_K) handles vector registers, with $(\mu_M, \mu_N) = (4, 4)$ and $\mu_K = b_K$.

Fig. 4(a) shows that the bottom-up method yields $(64, 96, 576)$, achieving peak single-thread performance but only using 24 threads, underutilizing 32 cores. Meanwhile, $(32, 144, 1152)$ significantly outperforms $(16, 288, 1152)$, yet cost models treat them identically. The true optimum $(64, 144, 576)$ is missed by both methods.

Furthermore, existing scale-up approaches use fixed step sizes. Fig. 4(b) reveals that scaling b_M yields rapid initial GFLOPS growth before convergence, indicating that coarse-grained early steps can skip fine decisions without sacrificing optimization.

Opportunity: Joint Optimization and Fast Start Bottom-up scaling excels at single-thread efficiency, while top-down cost models optimize parallel utilization. We propose a joint method that preserves both strengths. Inspired by momentum optimization [21] and TCP congestion control [17], we introduce a *fast-start* phase that aggressively expands initial programs, followed by a *finetune* stage that enumerates polymerization schemes and applies finer steps to reach optimal solutions.

4 Sandwich Overview

Sandwich comprises two core components, as illustrated in Fig. 5: service configuration generation (S1) and kernel orchestration (S2), both executed offline. For service configuration generation, Sandwich first parses the CPU (NUMA) architecture via system tools to construct the fundamental *TopoTree*. An exploration phase then enumerates all potential latent sub-structures among tree nodes using group transforms; it further employs remove transforms to discover all valid core utilization plans—wherein a subset of cores is reserved for decode operations. The resulting TopoTree is translated into two key schemes: a shared parallelism scheme and a core utilization scheme tailored for the decode phase. For tensor

program generation, Sandwich produces two sets of dedicated kernels: prefill kernels aligned with the prefill service configuration, and decode kernels matched to the decode service configuration. At runtime, these paired configurations and their corresponding optimized kernels are executed via hot-switching (S3).

5 Service Configuration Generation

5.1 TopoTree Abstraction

TopoTree is a multi-level tree abstraction for NUMA systems with hierarchical shared resources, representing core utilization plans. As shown in Fig. 6(a) ①, the *fundamental TopoTree* is built from hardware info (e.g., `lstopo` [23]): its leaf nodes correspond to Processing Units (PUs, physical/virtual CPU cores) dedicated to computation, non-leaf nodes represent shared resources (e.g., LLC cache for Super Core Cluster (SCCL) PUs), and color annotation— inspired by LLVM register allocation [4]— denotes domains/locality conflicts derived from profiling or architectural knowledge.

TopoTree enables automated tree generation/optimization and maps PUs to shared resource hierarchies. It represents a family of service configurations (process count, core distribution, NUMA association), making the configuration search space isomorphic to viable TopoTree variants.

5.2 TopoTree Transformation

To explore service configurations from the fundamental TopoTree, we design two transformations guided by key observations within color domains: 1. **Symmetric & Tileable**: SPMD-aligned intra-op parallelism [10, 22, 42] implies isomorphic subtrees per color, with equal PU counts per child node. 2. **Latent Shared Structure**: Tools like `lstopo` hide microarchitectural details (e.g., Kunpeng 920's LLC slices per 4-core CCL [33]), causing contention if unmodeled.

5.2.1 Transformation Definitions. 1. `group`(n, t, d): Inserts $L(d)/n$ new nodes at depth d, grouping $n \geq 2$ nodes with stride t. New nodes inherit colors based on subnode color sets. 2. `remove`(n, d): Removes n right-most children of depth-$d-1$ nodes (parallelized over same-color nodes).

5.2.2 Exploration Workflow. 1. `group` explores latent structures (e.g., LLC slices, Fig. 6(a) ②). 2. `remove` mitigates contention (e.g., CCL/PU pruning, Fig. 6(a) ③). 3. Enumeration stops when no new candidates emerge; only `remove`-derived TopoTrees are retained (group order is irrelevant).

5.2.3 Complexity & Pruning. 1. `group`: Bounded by visible subgraph core count ($n_s \ll n$), completes in 1.18–600s (converges to $O(1)$ for data center CPUs). 2. `remove`: $O(n^2)$ complexity is reduced through *Equivalent TopoTrees* - Merkle tree hashing [18] filters duplicates.

5.3 Interpret TopoTree to Service Configuration

As illustrate in Fig. 6(b), we map TopoTrees to service configurations via level-wise horizontal cross-sections: cross-section nodes determine process count and NUMA mapping, while subtrees define core locality and utilization. This enables thread/process-level partitioning for diverse configurations—e.g., 144 cores can be allocated as 8 processes × 16 cores (TP=8) or 48 processes × 3 cores. With $O(\log k)$ complexity (bounded by tree height $\leq \log_2(k)$ cores) and lightweight pruning, optimal configurations are selected via sampled token trace latency simulation using a default tensor schedule [39], with top-$k = 10$ candidates retained for subsequent kernel orchestration.

6 Tensor Program Generation

In LLM inference, the model partition plan fixes all input shape dimensions, with only the input sequence length determined at runtime. Inspired by Roller [44], Sandwich adopts a three-tier framework to generate tensor programs for dynamic shapes, striking a balance between execution performance and tuning overhead.

Micro-Kernel Generation Vectorized micro-kernels (MKs) of size $\mu_M \times \mu_N$ are customized for CPU architectures to accelerate core operations such as GEMM. These MKs leverage SIMD registers for multiply-accumulate computations, where μ_M and μ_N are selected to avoid register spilling, and the reduction dimension b_K is aligned with cache line sizes [44]. Regular access patterns enable hardware prefetchers to optimize memory latency, eliminating the need for manual prefetch instructions. MKs are generated via IR programming templates, following a design analogous to the tile-size auto-tuning mechanism in Triton [24].

Scheduling Plan Generation Building on prior MK polymerization approaches [36, 39, 44], Sandwich expands MKs into full tensor programs through a two-phase strategy that leverages CPU cache hierarchy. In the fast start phase, computation slice dimensions are expanded with an exponential step size ($2^{t_D} \times$ tilesize), inspired by TCP congestion control [17]; expansion is rolled back if no performance gain is observed, and scale is constrained to preserve thread parallelization potential. For finetuning, Sandwich enumerates all thread polymerization schemes across available CPU cores, expands computation slices along dimensions yielding optimal performance under each scheme, and selects the globally best tensor program while accounting for thread synergy and interference tradeoffs inherent to service configuration [5].

Sliding Window & Tensor Schedule Reuse To optimize for dynamically varying input sequence lengths [44], Sandwich employs two key optimizations. For the sliding window technique, input shapes differing only in sequence length are grouped and sorted: all MK candidates are used for the first σ shapes, and only the most recent σ optimal MKs are retained for subsequent shapes (with $\sigma = 16$ as the empirical optimal value). For tensor schedule reuse, Sandwich caches stable computation slice shapes and thread polymerization schemes once sequence lengths are sufficiently large [39, 44]; an early-stopping criterion halts further tuning, and cached schedules are reused to rapidly generate programs for unseen large shapes without redundant search.

7 Runtime Optimization

Sandwich is implemented with 14.4k Python and 6.7k C++ lines, fully integrated with vLLM (supporting continuous batching [37]) and incorporating kernel optimizations (positional embedding [29], FlashAttention [8], PagedAttention [22]) via IR parsing for multi-device compatibility. For inter-process communication, we use shared memory (SHM), where traditional SHM all-reduce [14] suffers from blocking synchronization and traffic contention due to

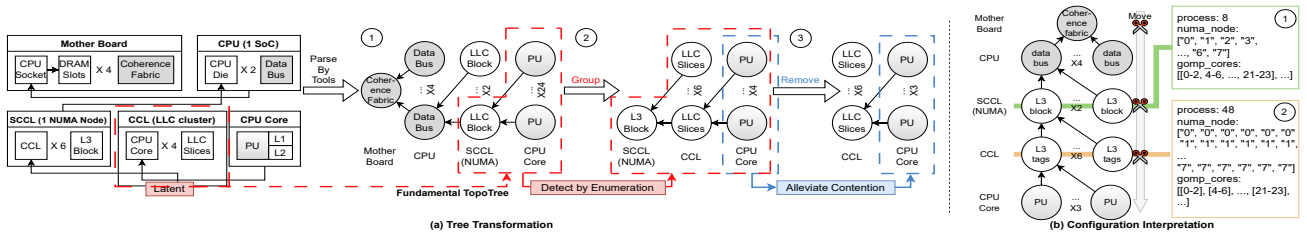


Figure 6: TopoTree transformation and interpretation on a Kunpeng 920 CPU to model its latent LLC cache hierarchy. The cache is partitioned per four-core cluster (CCL), creating a non-uniform domain where inter-CCL communication is costly, while intra-CCL access benefits from high-performance synergy.

fixed chunk sizes; inspired by [27], we propose a rank-shifted adaptive SHM approach that employs unique rank-based offsets for concurrent non-blocking accumulation, dynamically adjusts block sizes to exceed cache-line size (avoiding false sharing [3]), and limits threads for small inputs to reduce data movement overhead. Notably, all kernel executions and communication mechanisms are built on our custom backend, granting us full control over thread management to allow hot-switching.

8 Evaluation

8.1 Experimental Settings

Platform. We evaluate Sandwich across various NUMA systems. For x86 architectures, we use (i) Intel Xeon Gold 6151, (ii) Xeon Gold 6230, (iii) Xeon Platinum 8272CL, and (iv) AMD EPYC 7H12 (all 2 NUMA nodes). For ARM (aarch64), we assess Kunpeng 920 (4 CPUs, 8 NUMA nodes).

Model and Workload. We use the Llama series [31] (LLMs like QWen [30] has similar structure). For dynamic-shape benchmarks, a payload generator produces all feasible operator shapes via sampled sequence lengths. Experiments use single-NUMA-node cores (no interference), tuning all shapes up to max sequence length seq_{max} for compiler comparison. For serving, we evaluate single-request (90 sequential samples from ShareGPT/LMsys-Chat-1M [40]) and batched scenarios. Batched arrivals follow Poisson distributions (variable rates [22, 43]) with vLLM's default batching/scheduling; greedy sampling is used for token generation. We employ BF16 for AVX-512 CPUs, and FP32 for AVX-2 and Kunpeng platforms.

Metrics. Dynamic-shape benchmarks measure operator latency speed-ups (normalized to TVM) and tuning time. Serving uses 90% SLO attainment (P90, per [43]) as primary metric, plus single-sequence throughput and batched goodput (max SLO-meeting request rate). SLO parameters are detailed in Sec. 8.2.

Baselines. *Dynamic-shape:* DNN framework/vendor solutions (MKL-DNN [15], OpenVINO [16] for x86; Bolt [13], Torch+XNNPACK [34] for ARM) and compilers (TVM Ansor [41], DietCode, Roller, MikPoly [36, 39, 44]—reimplemented for CPU). AutoTVM [7] is excluded (no official CPU linear op support). *Serving:* (1) Vendor: ipex [28], OpenVINO [16]; (2) Open-sourced: vLLM [22] (CPU backend, no partition, NUMA-aware), llama.cpp [11]; (3) Optimized Version: vllmpp [43] (even NUMA/core partition, best TP configuration). Sandwich uses topk=10 configs (best performance selected).

Table 1: Kernel execution comparison between Sandwich and other vendor solutions. We select MKL as the x86 baseline and XNN pack as the ARM baseline to evaluate speedup.

Processor	Sandwich Avg. Speedup	openVINO Avg. Speedup	Bolt Avg. Speedup	Baseline
Xeon Gold 6230 (AVX-512)	1.29	0.90	-	x86: MKL
Xeon Gold 6151 (AVX-512)	1.33	0.87	-	x86: MKL
AMD EPYC 7H12 (AVX-2)	2.26	1.33	-	x86: MKL
KunPeng (ARM-NEON)	1.79	-	0.32	ARM: XNN Pack

8.2 Serving Benchmark

Single Request Serving. We conduct extensive serving experiments under two model sizes (1.3B, and 8B) on four platforms. As shown in Fig. 7, Sandwich can sustain 90% SLO attainment with up to 3.40×, and 4.45× more stringent SLOs, compared to OpenVINO and vLLM. As for token generation throughput, Sandwich achieves an average 2.01x higher throughput, shown in Fig. 8. vendor and open-source solutions. The tuning time of service configurations used in the experiments of Sandwich ranges from 78s to 3279s, with an average of 1309s. The kernel tuning time from $[1, seq_{max}]$ ranges from 2022s to 23115s, averaging 10155s.

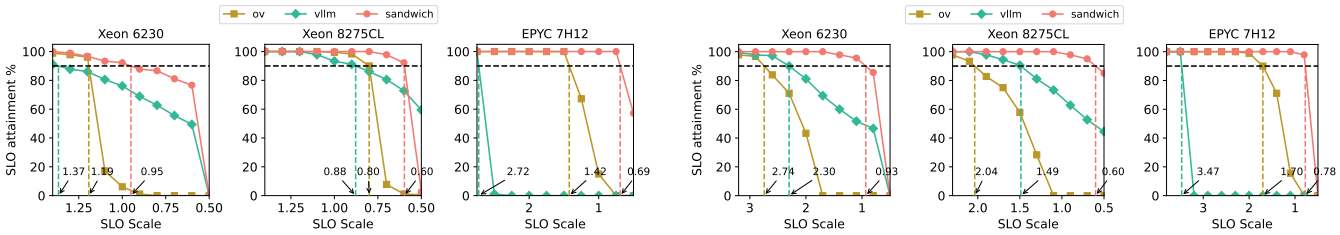
Batch Serving. We evaluate batch serving on 160M and 1.3B Llama models. Since vLLM supports only BF16 and AVX-512 inference, we conduct experiments on the Xeon 6151 and 6230 processors. As we gradually increase the request rate, P90 TPOT and TTFT results are collected. Fig. 9 shows that Sandwich can handle a higher request rate (1.84 and 0.92 requests per second) while meeting the TPOT and TTFT requirements. In contrast, the original vLLM consistently fails to satisfy the TTFT requirement.

Findings. Shown in Fig. 10, we found a dichotomy across platforms with respect to the optimal service configuration composition for 160M, 1.3B, 3B, and 8B models. On the Intel Xeon 8272, core reduction (`remove`) is used infrequently, and non-NUMA partitioning (`group`) does not consistently yield performance gains. For AMD and Kunpeng processors, however, their sub-cluster architecture makes core grouping and reduction scheme essential for optimal performance.

8.3 Dynamic-Shape Benchmark

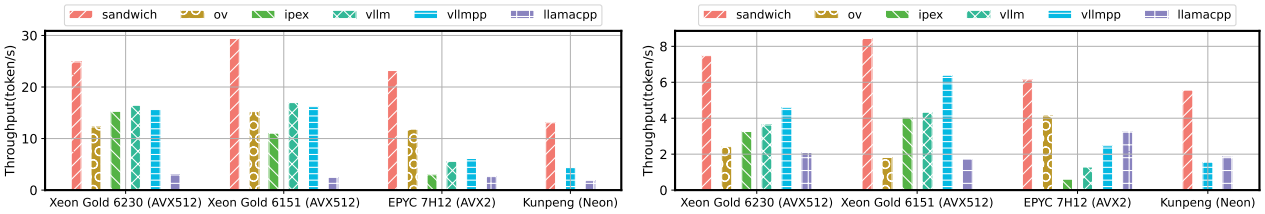
In this section, we compare the performance of Sandwich-generated kernels against baseline methods on commonly used shapes, focusing on GEMM (the primary computational operator in LLMs).

Vendors. As shown in Table 1, we compare our solution with vendor-optimized libraries (MKL for x86, XNNPack for ARM) and other third-party libraries. Sandwich achieves a 1.29–2.26 × speedup over the vendor baselines.



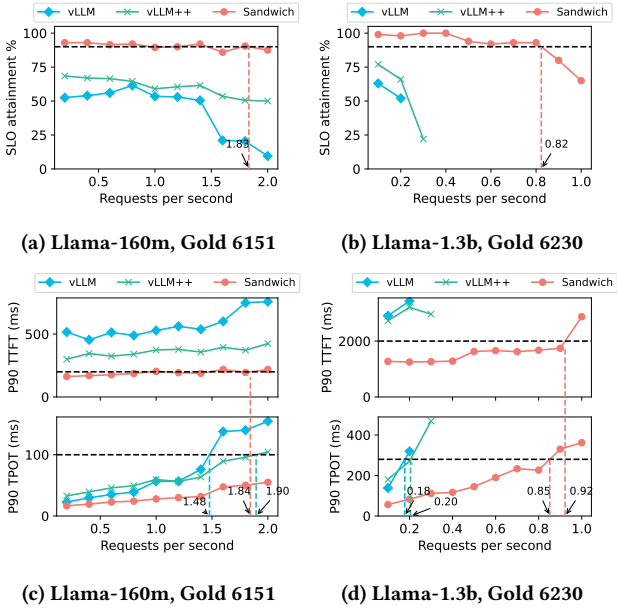
(a) Llama-1.3b, ShareGPT (b) Llama3-8b, ShareGPT

Figure 7: Comparison of SLO attainment percentage under different SLO scales.



(a) Llama-1.3b, ShareGPT (b) Llama3-8b, LMsys-Chat

Figure 8: Comparison of output token generation throughput for single request serving.



(a) Llama-160m, Gold 6151 (b) Llama-1.3b, Gold 6230

(c) Llama-160m, Gold 6151 (d) Llama-1.3b, Gold 6230

Figure 9: SLO and Goodput comparison for batched serving.

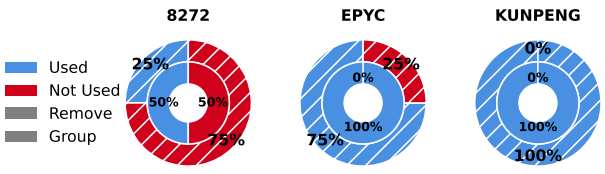


Figure 10: Final distribution of 'Remove' and 'Group' optimizations, where Grouping involves either non-NUMA partitioning or core sub-clustering (latent structure).

Compilers. As shown in Fig. 11, We compare kernel performance and tuning time across Gold 6151, EPYC 7H12, and Kunpeng platforms. Sandwich achieves performance comparable to TVM while

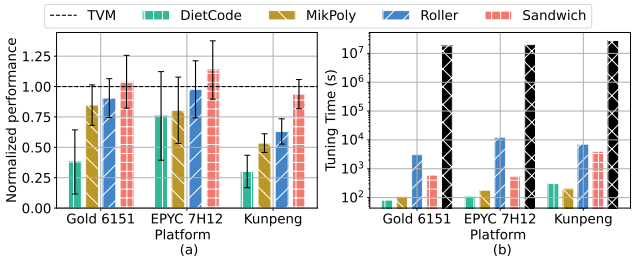


Figure 11: Kernel generation comparison among Sandwich and other compilers: (a) Normalized kernel performance; (b) Tuning time.

reducing tuning time by over 90%, outperforming Roller in tuning efficiency via tensor-schedule reuse. Cost-model-based methods (DietCode, MikPoly) are fast but lack competitive performance, while Sandwich excels by jointly optimizing concurrent polymerization and parallelizable schemes.

9 Conclusion

This paper introduces Sandwich, a compilation framework for efficient CPU-based LLM serving. Sandwich constructs a hardware-centric search space to optimize CPU core utilization and model partition granularity, thereby enhancing serving performance. Sandwich's three core innovations—seamless phase-wise plan switching, TopoTree-based substructure-aware core allocation, and fast-start-then-finetune kernel generation—enable it to deliver efficient core utilization plans and generate tensor programs with significantly reduced tuning time. These capabilities allow Sandwich to outperform all existing solutions, unlocking the full potential of CPUs for LLM serving. Our extensive experiments demonstrate that Sandwich achieves significant throughput gains, substantial Goodput improvement and can be used to serve under tighter latency SLO setups.

References

[1] AMD. n.d.. Server Processor Specifications.

[2] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small Language Models are the Future of Agentic AI. arXiv:2506.02153

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 117–128.

[4] G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/800230.806984

[5] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. Punica: Multi-Tenant LoRA Serving.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, USA, 579–594.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. Learning to Optimize Tensor Programs.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.

[9] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 381–394.

[10] Jiangsu Du, Jinhui Wei, Jiazhi Jiang, Shenggan Cheng, Dan Huang, Zhiguang Chen, and Yutong Lu. 2024. Liger: Interleaving Intra- and Inter-Operator Parallelism for Distributed Large Model Inference. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 42–54.

[11] Georgi Gerganov and contributors. 2023. llama.cpp: LLaMA inference in C/C++.

[12] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 709–729.

[13] Huawei Noah. Year. Bolt: A Deep Learning Library with High Performance and Heterogeneous Flexibility. https://github.com/huawei-noah/bolt.

[14] Intel. 2023. xFasterTransformer.

[15] Intel Corporation. 2024. Intel Math Kernel Library.

[16] Intel Corporation. 2024. Intel OpenVINO Toolkit.

[17] V. Jacobson. 1988. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols* (Stanford, California, USA) *(SIGCOMM '88)*. Association for Computing Machinery, New York, NY, USA, 314–329.

[18] Kiseok Jeon, Junghee Lee, Bumsoo Kim, and James J. Kim. 2023. Hardware Accelerated Reusable Merkle Tree Generation for Bitcoin Blockchain Headers. *IEEE Computer Architecture Letters* 22, 2 (2023), 69–72.

[19] Jiazhi Jiang, Jiangsu Du, Dan Huang, Dongsheng Li, Jiang Zheng, and Yutong Lu. 2023. Characterizing and Optimizing Transformer Inference on ARM Many-core Processor. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages.

[20] Jiazhi Jiang, Jiangsu Du, Dan Huang, Dongsheng Li, Jiang Zheng, and Yutong Lu. 2023. Characterizing and Optimizing Transformer Inference on ARM Many-core Processor. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages.

[21] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626.

[23] Open MPI. 2023. hwloc. https://github.com/open-mpi/hwloc.

[24] OpenAI. 2021. Introducing Triton: Open-source GPU programming for neural networks.

[25] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf

[26] Hongliang Qu and Zhibin Yu. 2024. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1233–1249.

[27] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2023. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. arXiv:2308.00852 [cs.NI]

[28] Haihao Shen, Hanwen Chang, Bo Dong, Yu Luo, and Hengyu Meng. 2023. Efficient LLM Inference on CPUs. arXiv:2311.00502 [cs.LG]

[29] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2023. RoFormer: Enhanced Transformer with Rotary Position Embedding.

[30] QWen3 Team. 2025. Qwen3 Technical Report.

[31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[32] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2019.00048

[33] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* 41, 5 (2021), 67–75. https://doi.org/10.1109/MM.2021.3085578

[34] XNNPack Contributors. 2024. XNNPack.

[35] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-Device Language Models: A Comprehensive Review. arXiv:2409.00088 [cs.CL] https://arxiv.org/abs/2409.00088

[36] Feng Yu, Guangli Li, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2024. Optimizing Dynamic-Shape Neural Networks on Accelerators via On-the-Fly Micro-Kernel Polymerization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 797–812.

[37] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538.

[38] Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. LLM-PQ: Serving LLM on Heterogeneous Clusters with Phase-Aware Partition and Adaptive Quantization.

[39] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *Proceedings of Machine Learning and Systems*, Vol. 4. Conference on Machine Learning and Systems, USA, 848–863.

[40] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL]

[41] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[42] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578.

[43] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving.

[44] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 233–248.