AsymServe: Demystifying and Optimizing LLM Serving Efficiency on CPU Acceleration Units

Xinkai Wang[1][0000−0003−3764−8065], Yiming Zhuansun[1], Chao Li[1][0000−0001−6218−4659], Jing Wang[1][0000−0001−7260−0521], Xiaofeng Hou[1][0000−0003−4372−7851], Lingyu Sun[1], Luping Wang[2], and Minyi Guo[1]

[1] School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
{unbreakablewxk, zsym2019, jing618, sunlingyu}@sjtu.edu.cn
{lichao, hou-xf, guo-my}@cs.sjtu.edu.cn
[2] Alibaba Group, Hangzhou, China
chamu.wlp@alibaba-inc.com

Abstract. Current data centers are accommodating more AI-based workloads, especially large-language model (LLM) training and serving in recent years. Given the limited count and significant energy consumption of expensive GPUs, cloud providers tend to utilize more cost-efficient processors for LLM serving, such as Intel scalable CPU equipped with acceleration units AMX. To understand the improvements, bottlenecks, and opportunities on this new platform, we first undertake a comprehensive characterization of LLM serving using AMX on two generations of modern CPUs with various memory devices. Our characterization reveals that the hardware and software behaviors of LLM serving on CPU are distinct from conventional cloud workloads and vary greatly. In this paper, we propose AsymServe to maximize LLM serving efficiency on scalable CPU platforms via handling software and hardware asymmetry. It adjusts hardware allocation and software configurations adaptively to maximize CPU performance-per-watt. Through extensive evaluation, we show that AsymServe improves LLM serving performance. Specifically, it achieves up to 1.71x faster first-token generation, 3.13x greater throughput, and 11.09x better energy efficiency.

Keywords: LLM Serving · CPU Efficiency · Intel AMX.

1 Introduction

Recent advancement in Large Language Models (LLMs), such as GPT [1] and LLaMA [17], marks a pivotal moment in the generative artificial intelligence (AI) era. These trends have led to the widespread adoption of LLM in various domains, ranging from personal assistants [3] to search engines [18]. With increasing LLM queries from more users, accommodating LLM inference (i.e., LLM serving) more efficiently grows in importance despite heavy-weight once-for-all LLM training [34]. To meet the substantial demands of LLM serving, despite high-performance Nvidia GPU, general, accessible, and cost-efficient CPUs are emerging for cloud providers to deploy LLM serving [7].

Using general-purpose CPUs for LLM serving seems counterintuitive, but this choice is promising and necessary. Promisingly, modern scalable CPUs are incorporating specialized acceleration units (AU), such as Intel AMX [20] and ARM SVE [8] that accelerate key LLM matrix multiplication operations [13], and larger memory capacities that support bigger models and higher batching sizes without offloading, significantly closing the performance gap with GPUs [19]. The performance promise and better cost-efficiency makes CPU a competitive alternative to GPUs for LLM serving [19, 24].

Given the performance improvements of LLM serving on CPU with AU [19, 13], the underlying bottlenecks are not well studied. We first undertake a comprehensive characterization of two generations of scalable CPUs and two types of LLMs in order to understand the obscure inner asymmetry of LLM and AU. Firstly, LLM serving on AU has asymmetric software behaviors due to unbalanced prefill and decode phases and sensitive configuration parameters, requiring disaggregated and divergent management for maximal LLM performance. Secondly, LLM serving on AU exhibits asymmetric microarchitectural properties from scalar functional units owing to its unique affinity on pipeline and uncore resources, requiring precise and adaptive decisions for maximal CPU efficiency

To this end, we argue that optimizing LLM serving on CPU architectures requires addressing system stack asymmetries, manifested through unbalanced workloads and unique hardware units. We take the first to propose **AsymServe**, an asymmetry-aware LLM serving framework designed to maximize CPU efficiency with LLM performance guarantee. AsymServe bridges LLM token scheduling and hardware allocation through two coordinated components. The *Offline Profiler* jointly models software inference sensitivity and hardware criticality into an *Asymmetry Model*. The *Online Scheduler* uses LAG analysis to dynamically set the most energy-efficient configurations based on real-time constraints and the *Asymmetry Model*.

To evaluate AsymServe, we have implemented and evaluated it with various workloads on scalable CPU platforms. Evaluation results show that AsymServe achieves better CPU performance per watt efficiency by up to 11.09x compared with state-of-the-art (SOTA) CPU LLM serving methods. Meanwhile, it outperforms SOTA methods in guaranteeing LLM performance maximum 3.13x throughput improvement and 1.71x latency reduction.

In summary, this paper makes the following contributions:

- **Analysis**: We comprehensively characterize LLM serving on CPU with acceleration units in terms of software-layer and hardware-layer behaviors, bottlenecks, and opportunities.
- **Design**: We introduce AsymServe, an asymmetry-aware LLM serving framework designed to maximize CPU efficiency. We design cascaded offline and online components to handle the asymmetric system stacks.
- **Evaluation**: We implement and evaluate AsymServe to prove its better efficiency and performance compared to SOTA methods.

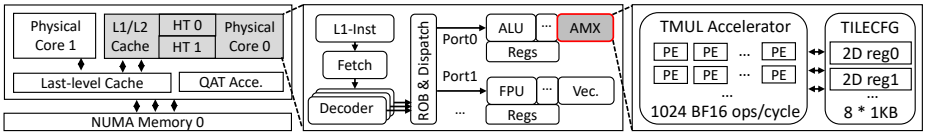

Fig. 1: Layout of scalable CPU with acceleration units.

2 Background and Related Work

2.1 Acceleration Units on Scalable CPU

To improve the performance of AI applications on CPU platforms, hardware vendors have integrated domain-specific acceleration units (AU) into recent processors, such as Intel Advanced Matrix Extensions (AMX) [20] and ARM Scalable Vector Extensions (SVE) [8]. It follows the single-instruction-multiple-data (SIMD) paradigm and extends CPU from traditional vector extensions (e.g., AVX-512 in 2013 [23]) to matrix multiplication acceleration. Different from dedicated accelerators (e.g., QAT [31]) outside the pipeline for specific operations, AUs on every physical core share the instruction flow and data access path with normal functional units as shown in Figure 1. Every AMX unit contains an array of eight 2-dimensional registers (TILECFG) with the size of 1KB and a matrix multiply accelerator (TMUL) that performs 1024 ops per cycle. Compared to expensive and limited GPU, scalable CPU equipped with AUs is becoming a competitive option in the AI era for its generality and cost-efficiency [19, 28].

Hardware asymmetry (i.e., heterogeneity) is widely studied for accelerating domain-specific tasks. Regardless of accelerator designs [11] and CPU-accelerator cooperation [29], there are three main asymmetries inside CPU. First is ISA asymmetry that fully exploits heterogeneous ISA in one chip via design space exploration [26] and execution migration [2]. Second is processor asymmetry that adopts big.LITTLE architecture for both high performance and low power consumption [21]. Third is component asymmetry, which considers specialized functional units within the CPU as AsymServe does. Prior to AMX usage [13], AVX usage has been studied and optimized via core specialization [5, 4]. *To the best of our knowledge, AsymServe is the first to study the usage and optimization of more asymmetric and complex AMX.*

2.2 LLM Serving Systems

Large language models (LLM) have shown impressive achievements and great potential in various creative tasks, leading the boom of the AI era [17, 1]. Besides once-for-all heavyweight LLM training, optimizing the performance and efficiency of LLM serving (i.e., inference) is gaining more attention in both academia and industry [15, 7, 16]. LLM inference contains two phases: 1) The prefill phase processes all tokens of the input prompt simultaneously to generate the first new token. 2) The decode phase uses the previously generated token

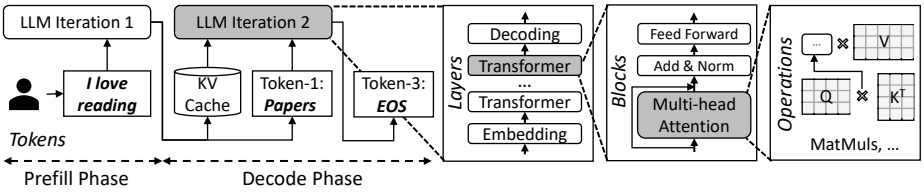

Fig. 2: LLM serving workflow at different granularities.

as input to produce the next tokens one by one until an end-of-sequence (EOS) token is produced [34]. Two phases are relatively independent and have different characteristics, motivating disaggregated optimizations [22, 33], as shown in Figure 2. To generate every subsequent token, LLM iteration passes through repeated transformer layers composed of multiple blocks like multi-head attention. The heaviest attention block contains multiple matrix-multiply and dot-product operations on the query (Q), key (K), and value (V) matrices, which can be accelerated using AUs on scalable CPU [13, 19].

LLM serving systems are gaining popularity in both academia and industry. Regardless of well-crafted system and framework designs like continuous batching [15] and prefill/decode disaggregation [22, 33] on GPUs, LLM serving communities are exploring other cost-efficient hardware platforms like FPGA [32] and CPU [24]. Regarding CPU-based LLM serving, current solutions focus on KV cache optimization [7] and usage of specialized units [19]. Compared to GPU platform, LLM serving performance on modern CPUs is preliminarily studied [19, 13]. *To the best of our knowledge, AsymServe is the first to characterize CPU-based LLM serving asymmetry and optimize its efficiency systematically.*

3 Performance Analysis of LLM with CPU AUs

3.1 Characterization Methodology

Hardware: In Sections 3 and 4, we characterize LLM serving on two commercial off-the-shelf CPU platforms: Intel 4th Sapphire Rapids (SPR) [20], 6th Granite Rapids (GNR) [10] scalable CPUs released in 2022 and 2024, respectively. The hardware specifications are shown in Table 1. The main differences are two-fold: (1) GNR has better AMX support and FP16 precision; (2) our SPR systems use DDR (SPR-DDR) and DDR+HBM (SPR-HBM) memory configurations, while GNR system adopts MCR memory (GNR-MCR). To avoid inter-node memory access cost, we only use one socket (48 cores) to serve LLM on SPR-DDR and SPR-HBM. On GNR-MCR, we use all 120 cores.

Software: To deploy LLM serving on CPUs using AMX, we use Intel xFasterTransformer (xft) [7] framework with the latest AMX support. We use xft to serve the open-sourced llama-2-7b [17] and llama-2-13b. To study the cascaded yet different phases (prefill and decode) in LLM serving [22, 33], we mimic them in xft with varying batch sizes, ranging from 1 to 32. Prefill has an input sequence

Table 1: Hardware specifications of evaluated CPUs.

	SPR-DDR	**SPR-HBM**	**GNR-MCR**
Generation	Sapphire Rapids	Sapphire Rapids	Granite Rapids
CPU	Xeon 8475B	Xeon 9468	Xeon 6982P-C
#cores / sockets	48 / 2	48 / 2	120 / 1
Frequency	2.7 GHz	2.1 GHz	2.8 GHz
L1-I / core	32 KB	32 KB	64 KB
L1-D / core	48 KB	48 KB	48 KB
L2 / core	2 MB	2 MB	2 MB
Shared LLC	97.5 MB	105 MB	504 MB
Memory	DDR5 1TB	DDR5 1TB+HBM 128GB	DDR5 768GB
Memory BW	614 GB/s	973 GB/s	845 GB/s

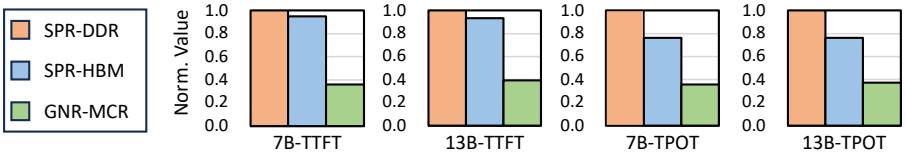

Fig. 3: Prefill and decode performance across models and platforms.

length of 512 and an output sequence length of 1, while the input and output for decode are both 512. As previous work [33], we evaluate three representative LLM applications with distinct input-output characteristics: (1) ChatGPT-like *chatbot* with 512-token input and 200-token output; (2) Cursor-like *code completion* with 128-token input and 98-token output; (3) *summarization* with 2016-token input and 32-token output. We use Linux perf [6], pmu-tools [14], and pqos [9] tools to characterize LLM and CPU behaviors.

Metrics: To measure the performance of LLM serving, we select various metrics widely used in previous studies [19, 34]. For prefill performance, we use time-to-first-token (TTFT), indicating the time to generate the first token. For decode performance, we use time-per-output-token (TPOT), indicating the average time taken to generate subsequent tokens. Also, the throughput of the two phases is measured as generated tokens per second. Different metrics are critical for various use cases of LLM serving [34].

3.2 Decomposed LLM Serving Performance Investigation

We benchmark the performance of the phases using varying model sizes (7B and 13B) of llama-2 across the three platforms. We use task chatbot and set the batch size as 16. Figure 3 shows multidimensional performance variations in LLM serving with the chatbot task and batch size 16.

Our performance decomposition analysis reveals three critical factors affecting LLM serving efficiency: phase characteristics, model size, and hardware re-

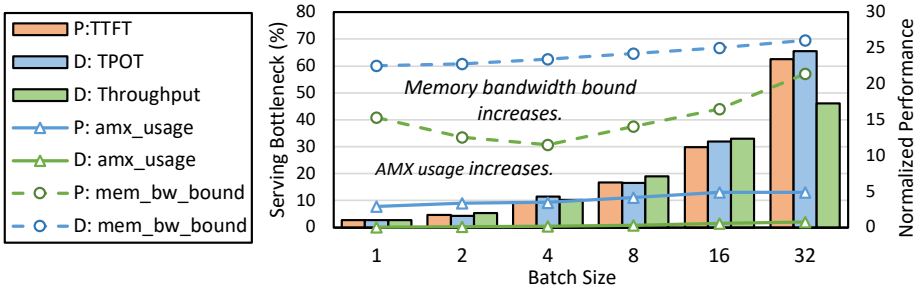

Fig. 4: Influence of batch size on LLM serving with AMX.

source configurations. Prefill and decode are two processing phases in LLM serving, each with different characteristics. The prefill phase initializes contextual states and processes input sequences, while the decode phase iteratively generates output tokens. Though both phases are memory-bound due to their reliance on the model, their different compute patterns affect the performance. In the prefill stage, the full-sequence operations make the phase more computationally intensive, which results in comparable performance between SPR-DDR and SPR-HBM platforms with similar computing resources. In the decode stage, autoregressive token generation makes the performance more susceptible to memory bandwidth. Therefore, as Figure 3 shows, SPR-HBM improves TPOT performance compared with SPR-DDR. Model size also affects LLM serving performance. Larger models require more computing resources and consume greater memory bandwidth. Hardware resources, mainly computing and bandwidth, critically influence serving performance. Table 1 shows that SPR-HBM exceeds SPR-DDR in bandwidth, while GNR-MCR surpasses SPR-HBM with enhanced computational power. Figure 3 demonstrates that increased computing and bandwidth resources effectively accelerate LLM serving.

Findings #1: *The decomposed two LLM serving phases show distinct characteristics, necessitating separate consideration and optimization.*

3.3 Impact of Software Configurations on LLM Performance

Impact of batch size on AMX usage To accommodate LLM serving, batch size, the number of parallel inputs for LLM, is a crucial hyperparameter to optimize LLM serving performance. Continuous batching with varying batch sizes achieves better performance [15], but how batch size impacts AMX usage is understudied. Despite lower cache misses, higher core utilization, and more load/store instructions with higher batch sizes [19], we further compare the performance and AMX usage of prefill and decode phases in Figure 4. The performance is normalized to batch size=1. Obviously, a larger batch size leads to better throughput but worse per-token latency. We find that AMX performs more computations under larger batch sizes due to larger matrix dimensions and more multiply computations, especially for the prefill phase.

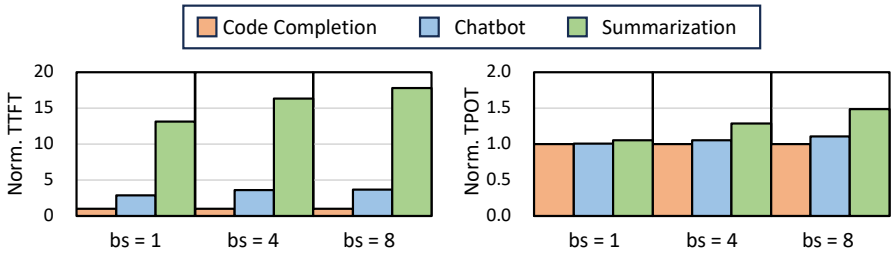

Fig. 5: Influence of sequence length on LLM serving performance.

Impact of sequence length on performance Sequence length, the number of input tokens for LLM, is a critical parameter in task design. Longer input sequences require significantly more compute resources during the prefill phase, whereas the impact on the decode phase remains relatively marginal. Figure 5 illustrates how TTFT and TPOT evolve across three workloads on the SPR-DDR platform as the batch size increases from 1 to 8, where the llama-2-7b model was employed. The prefill phase demonstrates near-linear TTFT scaling with sequence length, while TPOT in the decode stage remains relatively stable.

Findings #2: *Software configuration impacts system resource utilization in LLM serving, resulting in observable performance variations.*

4 In-depth Resource Sensitivity Analysis

4.1 Impact of Microarchitectural Resources

With an understanding of AU usage in LLM serving, we further analyze the resource bottlenecks from the microarchitectural perspective. We use the top-down analysis methodology [30] with a concentration on AU-related insights rather than general cycle/function distributions in prior works [12, 25]. This section investigates AU's affinity for various resources and where AU is bounded.

Frontend resources are over-supplied Figure 6 shows the CPU cycle distributions of two phases on three platforms, compared with *mcf* benchmark from SPECCPU and *ads* services from Google [12, 25]. Frontend bound (in blue) includes stall cycles at the L1 instruction (L1-I) cache, fetch unit, and decoder. LLM serving on AMX follows SIMD paradigm with a smaller instruction working set and fewer i-cache misses, leading to significantly lower fetch latency than before ($\sim$ 5% stall cycles due to fetch latency [12]). As for fetch bandwidth, contention on the decoded μop cache *tma_ dsb* is more severe than the legacy decode pipeline *tma_ mite*, but fetch bandwidth is excessive for AMX usage. The prefill and decode phases behave similarly at the frontend. Meanwhile, SPR is hardly bounded at the frontend but AMX on GNR requires the frontend resources

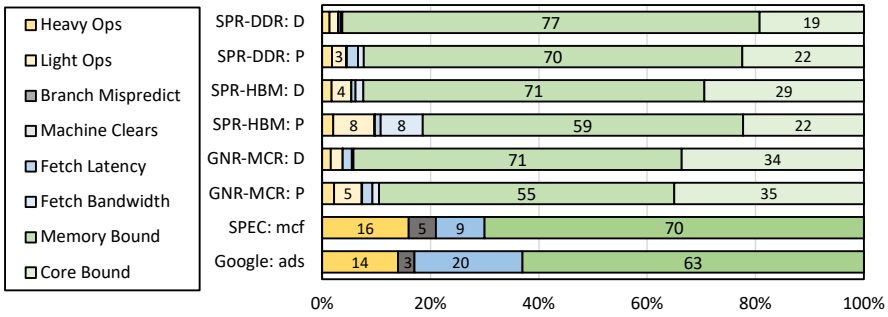

Fig. 6: Cycle distributions for LLM serving on AMX. Yellow, gray, blue, and green categories are retiring, bad speculation, frontend bound, and backend bound, respectively.

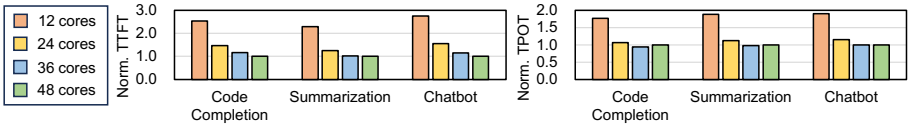

Fig. 7: Influence of multicore scaling on LLM serving performance.

more due to improved backend resources. Compared to traditional workloads using general units, AU has a much lower affinity for frontend resources. The over-supplied frontend resource for LLM serving using AUs is wasted and could be aggressively shared to co-running workloads.

Backend resources are overloaded Despite abundant frontend resources for LLM serving, backend resources are under strain and restrict performance. As shown in Figure 6, the backend bound (in green) includes stall cycles at execution ports (core bound) and memory access path (memory bound). In both prefill and decode phases, backend bound dominates the majority of slots. However, within the backend bound category, the two stages exhibit distinct characteristics. The decode phase reports higher memory bound compared to the prefill phase, indicating increased memory demands during this stage.

Findings #3: *Different microarchitectural bottlenecks of AU running LLM serving present intra-AU resource asymmetry on frontend and backend resources.*

4.2 Impact of Multicore Scaling

The core count represents the available AU resources for LLM serving. We indirectly control the actual core usage by adjusting the number of threads in xft

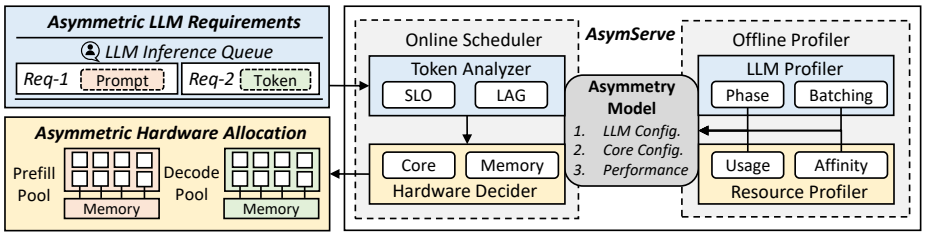

Fig. 8: Design overview of AsymServe.

inference. Figure 7 shows the performance of three tasks under llama-2-7b with a batch size of 8 as core count increases from 12 to 48. On the whole, higher core counts improve both prefill and decode performance, with prefill showing more significant gains. Interestingly, in some decode scenarios (like code completion and summarization in Figure 7) and certain prefill cases (not shown in the figure), LLM serving performs better using 3/4 of the available cores rather than all cores. This occurs because fewer cores reduce bandwidth contention between different cores, improving AU utilization and overall efficiency.

Findings #4: *Due to high contention for non-AU resources in LLM serving, greedy usage of AU can lead to its underutilization.*

5 Design of AsymServe

Based on the analysis above, we argue that optimizing LLM serving on acceleration units must consider the asymmetric system stacks. As shown in Figure 8, the asymmetric LLM requirements and hardware allocation require the cloud providers to consider and optimize jointly for better efficiency. Therefore, we propose **AsymServe**, an asymmetry-aware LLM serving framework designed to maximize the efficiency of the CPU with acceleration units. It operates at the operation system layer to determine the optimal configurations for a given serving service-level-objective (SLO).

AsymServe contains two cooperative components. The *Offline Profiler* works in the background to understand the software inference behaviors with *LLM Profiler* and hardware resource allocation with *Resource Profiler*. The profiles model the impact of LLM and hardware configurations on inference performance, synthesized into a unified *Asymmetry Model*. The key module of AsymServe is *Online Scheduler* that makes an allocation decision in real-time for a newly submitted request. It determines real-time execution lag with *Token Analyzer* and selects proper resources that can maximize CPU efficiency with *Hardware Decider* based on the *Asymmetry Model*.

5.1 Offline Profiling

LLM profiler To model the LLM serving performance with varied software configurations, the *LLM Profiler* builds a performance profile for every serving parameter individually. For every serving request of LLM m, it quantifies the performance with four-dimensional software features. The input features include model size s, execution phase p, input length i, and batch size bs. The performance indicators for prefill and decode phases are TTFT and TPOT, respectively. In order to improve the accuracy and reduce the profiling overhead, we discretize the input length and batch size with discrete values. Specifically, AsymServe divides the space of features into multiple buckets and then takes the upper bound of the bucket as the final value.

Resource profiler To model the LLM serving performance with varied resource usages, the *Resource Profiler* captures LLM affinity on acceleration units and memory selections. Every LLM configuration further quantifies the performance with five more resource dimensions. The input features include the number of cores n, core frequency f, memory device m, cache allocation c, and memory bandwidth allocation mb. We record the 50% mean performance p_{mean} and 90% tail performance p_{tail} of repeated inferences to reduce model biases. The LLM serving performance models the software and hardware features: $p_{mean}, p_{tail} = f_m(s, p, i, bs, n, f, m, c, mb)$. Similarly, we use discrete resource allocation buckets for overhead reduction and further adopt transfer learning between models to obtain the execution performance agilely.

5.2 Online Scheduling

Phase-aware Token Analysis AsymServe adopts token SLO for prefill and decode phases. For prefill tokens, shorter TTFT makes the serving system more responsive. We simply use first-come-first-serve (FCFS) to schedule prompts with the batch size of 1. The deadline for prefill tokens is set as $d_{TTFT} - t_{wait}$, where d_{TTFT} is the TTFT SLO (e.g., $1s$) and t_{wait} is the request waiting time.

For abundant decode tokens in LLM serving requests, AsymServe tracks the performance of tokens at runtime to optimize CPU efficiency at token granularity. We adopt LAG analysis to determine the performance and resource requirements of the next batch of tokens. We quantify the relationship between the partial execution time at time t of serving request i (e_i) and its relative deadline, denoted as LAG_i, as shown in Equation 1.

$$LAG_i(token, T_i(t)) = \sum_{token \in T_i(t)} (d_{TPOT} - e_{token}) \tag{1}$$

in which $T_i(t)$ is the tokens of request i that have completed by time t. For token $token \in T_i(t)$, d_{TPOT} is set as the TPOT SLO (e.g., 100ms), and e_{token} is the recorded execution time for token t, respectively. Since LAG indicates how far behind ($LAG < 0$) or ahead ($LAG > 0$) each serving request is, the resource allocation can be adjusted accordingly for faster or slower generation.

SLO-aware Hardware Decider AsymServe optimizes hardware allocation for LLM serving by dynamically matching resources to real-time performance targets (SLO) while maximizing performance-per-watt efficiency. The system evaluates processing capabilities through tokens-per-second metrics and calculates CPU power consumption using dynamic measurements. Deployment nodes are initially selected based on memory devices capable of meeting minimum SLO requirements. For the compute-intensive prefill phase, all available processor cores are utilized, while the decode phase activates only cores necessary to satisfy runtime SLO, leaving surplus cores idle. Finally, cache allocation and memory bandwidth are tuned using prior asymmetry modeling data, achieving precise resource allocation without additional waste.

6 Evaluation

6.1 Experiment Methodology

Implementation We implement a prototype of AsymServe based on xFasterTransformer [7]. The hardware platforms, LLM serving configurations, and performance metrics are similar to Section 3.1. As for SLO settings, we set diverse performance SLOs for each application [33]: (1) *chatbot*: TTFT SLO=400 ms, TPOT SLO=80 ms; (2) *code completion*: TTFT SLO=125 ms, TPOT SLO=200 ms; (3) *summarization*: TTFT SLO=3s, TPOT SLO=150 ms. These SLOs reflect real-world requirements for different LLM-powered scenarios.
Baselines We compare AsymServe with three baselines: one state-of-the-art CPU LLM serving method, *Exclusive*, that uses all hardware resources for LLM serving exclusively. Two AsymServe variants to validate the decomposed improvements: *AsymServe-H* is AsymServe only with hardware asymmetry, i.e., allocating hardware resources considering AU profiles and setting software configurations same with the *Exclusive* scheme. *AsymServe-S* is AsymServe only with software asymmetry, i.e., adjusting software configurations considering LLM SLOs and allocating hardware resource same with the *Exclusive* scheme. *AsymServe* considers the two-dimensional profiles jointly at runtime.

6.2 Evaluation Results

Performance We begin by evaluating the performance of AsymServe. Our LLM serving evaluation uses three key metrics: TTFT, TPOT, and throughput from Section 3.1. We conducted comprehensive testing using our experimental platform across four schemes, employing various tasks, different LLM models, and multiple software/hardware configurations. The experimental results from testing the llama-2-7b model on GNR-MCR are shown in Figure 9. In this figure, the three tasks are labeled using the following abbreviations: code completion (CC), summarization (Summ), and Chatbot (Chat).

During the prefill phase, baseline performance often violates SLOs (in code completion and chatbot tasks). To address these violations, we experiment with

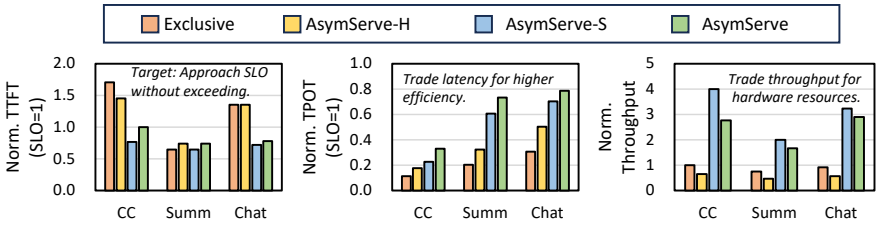

Fig. 9: Performance of AsymServe.

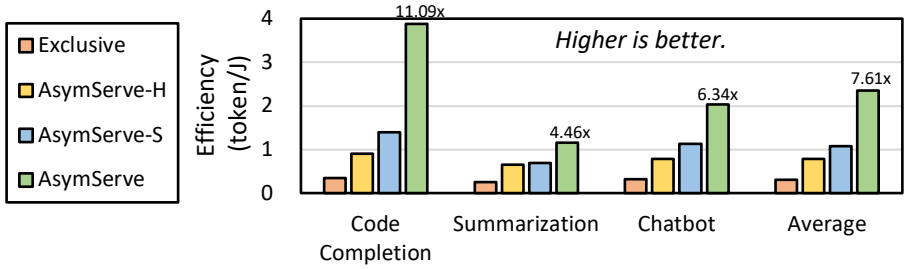

Fig. 10: Energy efficiency of AsymServe.

AsymServe-H, AsymServe-S, and AsymServe. When meeting SLO requirements, the software solution AsymServe-S occasionally achieves lower latency than AsymServe because AsymServe strategically allows increased latency (still below SLO limits) to free up extra resources. Across three tested scenarios in Figure 9, AsymServe achieves up to 1.73x TTFT improvement compared to the baseline.

During the decode phase, the baseline typically shows TPOT values well below SLO limits. As shown in Figure 9, the hardware approach AsymServe-H saves resources but reduces throughput because of limited software flexibility. AsymServe-S maximizes throughput but occupies all hardware resources. AsymServe effectively balances these extremes and dynamically adjusts resource allocation to improve throughput while ensuring SLO compliance and preserving system resources. In this stage, AsymServe achieves 2.23x to 3.13x higher throughput compared to the baseline.

Energy efficiency As discussed, AsymServe not only enhances LLM serving performance but also releases redundant hardware resources, making it more energy-efficient than the other three schemes. We define energy efficiency in LLM serving as the ratio of throughput to power consumption. Figure 10 demonstrates the energy efficiency across different tasks on the GNR-MCR platform running the llama-2-7b model. By optimizing batch sizes and reducing allocated computing cores, AsymServe effectively lowers energy consumption per token, achieving 4.46x to 11.09x (average 7.61x) energy efficiency improvements.

7 Discussion and Future Work

In this paper, we preliminarily demystify and optimize LLM serving efficiency on modern CPU with acceleration units, Intel AMX. However, there are three limitations leaving for future work.

Firstly, larger and more diverse LLMs are unexplored. We study the behaviors of standard small-sized LLaMA models, while leaving the recent Mixture of Experts (MoE) models with larger memory requirements and variable quantization techniques [34] as future work. *Secondly, CPU microarchitectural resource are leaving static.* We adjust the hardware resources with off-the-shelf technologies, while the LLM distinct resource demands at the microarchitectural layer could be grasped with more flexible architecture optimizations [26]. *Thirdly, more cost-efficient sharing LLM deployment is promising.* We optimize LLM serving efficiency without any colocation, which is common practices in datacenters. In the future, we are going to exploiting the underutilized resources of CPU LLM serving to further improve resource utilization and overall efficiency [27].

8 Conclusion

The scalable CPU equipped with acceleration units is becoming a more general and cost-efficient option for LLM serving. We extensively characterize LLM serving using AMX to understand its asymmetric software behaviors and hardware demands. Based on our insights, we design AsymServe, an asymmetry-aware LLM serving framework to handle unbalanced LLM serving on specialized functional units. Through extensive evaluations, AsymServe demonstrates enhanced LLM serving performance, achieving maximum improvements of 1.71x in time-to-first-token latency, 3.13x in throughput, and 11.09x in energy efficiency.

Acknowledgments. We sincerely thank all the anonymous reviewers for their valuable comments that helped us to improve the paper. This work is supported by the National Natural Science Foundation of China (No. U23A6007) and an Alibaba Research Grant. The corresponding authors are Chao Li and Luping Wang.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. DeVuyst, M., Venkat, A., Tullsen, D.M.: Execution migration in a heterogeneous-isa chip multiprocessor. In: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems. pp. 261–272 (2012)
3. Google: Assistant with bard: A step toward a more personal assistant. Available: https://bit.ly/4h3Pti8, 2024
4. Gottschlag, M., Brantsch, P., Bellosa, F.: Automatic core specialization for avx-512 applications. In: Proceedings of the 13th ACM International Systems and Storage Conference. pp. 25–35 (2020)

5. Gottschlag, M., Machauer, P., Khalil, Y., Bellosa, F.: Fair scheduling for avx2 and avx-512 workloads. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). pp. 745–758 (2021)
6. Gregg, B.: perf examples. Available: https://www.brendangregg.com/perf.html, 2024
7. He, P., Zhou, S., Huang, W., Li, C., Wang, D., Guo, B., Meng, C., Gui, S., Yu, W., Xie, Y.: Inference performance optimization for large language models on cpus. In: ICML 2024 Workshop on Foundation Models in the Wild (2024)
8. Iliescu, D.A., Petrogalli, F.: Arm scalable vector extension and application to machine learning. Retrieved October (2018)
9. Intel: Intel® rdt software package. Available: https://github.com/intel/intel-cmt-cat, 2024
10. Intel: Intel unveils future-generation xeon with robust performance and efficiency architectures. Available: https://bit.ly/4gobDeL, 2024
11. Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z., Patterson, D.A.: Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In: Proceedings of the 50th Annual International Symposium on Computer Architecture. ISCA '23 (2023)
12. Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.Y., Brooks, D.: Profiling a warehouse-scale computer. In: Proceedings of the 42nd annual international symposium on computer architecture. pp. 158–169 (2015)
13. Kim, H., Ye, G., Wang, N., Yazdanbakhsh, A., Kim, N.S.: Exploiting intel® advanced matrix extensions (amx) for large language model inference. IEEE Computer Architecture Letters (2024)
14. Kleen, A.: Intel pmu profiling tools. Available: https://github.com/andikleen/pmu-tools, 2024
15. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with pagedattention. In: Proceedings of the 29th Symposium on Operating Systems Principles. pp. 611–626 (2023)
16. Liu, J., Tang, P., Hou, X., Li, C., Heng, P.A.: Loraexit: Empowering dynamic modulation of llms in resource-limited settings using low-rank adapters. In: Findings of the Association for Computational Linguistics: EMNLP 2024. pp. 9211–9225 (2024)
17. Meta: Introducing llama 3.2. Available: https://www.llama.com/, 2024
18. Microsoft: Introducing the new bing. the ai-powered assistant for your search. Available: https://bit.ly/3DHVp26, 2024
19. Na, S., Jeong, G., Ahn, B.H., Young, J., Krishna, T., Kim, H.: Understanding performance implications of llm inference on cpus. In: 2024 IEEE International Symposium on Workload Characterization (IISWC). pp. 169–180. IEEE (2024)
20. Nassif, N., Munch, A.O., Molnar, C.L., Pasdast, G., Lyer, S.V., Yang, Z., Mendoza, O., Huddart, M., Venkataraman, S., Kandula, S., et al.: Sapphire rapids: The next-generation intel xeon scalable processor. In: 2022 IEEE International Solid-State Circuits Conference (ISSCC). vol. 65, pp. 44–46. IEEE (2022)
21. Padoin, E.L., Pilla, L.L., Castro, M., Boito, F.Z., Alexandre Navaux, P.O., Méhaut, J.F.: Performance/energy trade-off in scientific computing: the case of arm big. little and intel sandy bridge. IET Computers & Digital Techniques **9**(1), 27–35 (2015)

22. Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., Bianchini, R.: Splitwise: Efficient generative llm inference using phase splitting. In: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). pp. 118–132. IEEE (2024)
23. Reinders, J.R.: Intel® avx-512 instructions. Available: https://bit.ly/3DbYbfL, 2017
24. Shen, H., Chang, H., Dong, B., Luo, Y., Meng, H.: Efficient llm inference on cpus. arXiv preprint arXiv:2311.00502 (2023)
25. Sriraman, A., Dhanotia, A.: Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 733–750 (2020)
26. Venkat, A., Tullsen, D.M.: Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. ACM SIGARCH Computer Architecture News **42**(3), 121–132 (2014)
27. Wang, X., He, H., Li, Y., Li, C., Hou, X., Wang, J., Chen, Q., Leng, J., Guo, M., Wang, L.: Not all resources are visible: Exploiting fragmented shadow resources in shared-state scheduler architecture. In: Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC). pp. 109–124 (2023)
28. Wang, X., Hou, X., Li, C., Li, Y., Liu, D., Xu, G., Yang, G., Zhang, L., Wu, Y., Yuan, X., Chen, Q., Guo, M.: Exist: Enabling extremely efficient intra-service tracing observability in datacenters. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS). p. 355–372 (2025). https://doi.org/10.1145/3676641.3716283
29. Wang, X., Li, C., Sun, L., Lyu, Q., Hou, X., Leng, J., Guo, M.: Sheeo: Continuous energy efficiency optimization in autonomous embedded systems. In: 2024 IEEE 42nd International Conference on Computer Design (ICCD). pp. 496–503 (2024). https://doi.org/10.1109/ICCD63220.2024.00082
30. Yasin, A.: A top-down method for performance analysis and counters architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 35–44. IEEE (2014)
31. Yuan, Y., Wang, R., Ranganathan, N., Rao, N., Kumar, S., Lantz, P., Sanjeepan, V., Cabrera, J., Kwatra, A., Sankaran, R., et al.: Intel accelerators ecosystem: An soc-oriented perspective: Industry product. In: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). pp. 848–862. IEEE (2024)
32. Zeng, S., Liu, J., Dai, G., Yang, X., Fu, T., Wang, H., Ma, W., Sun, H., Li, S., Huang, Z., et al.: Flightllm: Efficient large language model inference with a complete mapping flow on fpgas. In: Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays. pp. 223–234 (2024)
33. Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., Zhang, H.: Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). pp. 193–210 (2024)
34. Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., et al.: A survey on efficient inference for large language models. arXiv preprint arXiv:2404.14294 (2024)