HiNa-MoE: High-Performance, Non-Intrusive MoE Inference on CPUs with Matrix Engines

Weiling Yang*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
w.yang@nudt.edu.cn

Junwen Zhang*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
zhang_junwen@nudt.edu.cn

Dezun Dong†
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
dong@nudt.edu.cn

Jianbin Fang†
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
j.fang@nudt.edu.cn

Enda Yu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
yuenda@nudt.edu.cn

Zhe Bai
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
baizheo3o@nudt.edu.cn

Xiaopeng Deng
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
dengxiaopeng25@nudt.edu.cn

ABSTRACT

Mixture-of-Experts (MoE) inference is increasingly deployed in local and on-premise environments, where expert parameters often exceed GPU memory capacity. In latency-sensitive, low-concurrency settings, repeatedly staging routed-expert weights from CPU memory to the GPU can be prohibitive, leaving routed-expert feed-forward networks (FFNs) on the critical path of multi-socket CPUs. Existing CPU accelerations often rely on intrusive, hardware- or topology-specific requirements, such as AMX-specific weight layouts or manual NUMA-aware placement. These requirements reduce portability and complicate integration with standard CPU–GPU offloading pipelines. We present HiNa-MoE, a high-performance, non-intrusive operator library for MoE inference on CPUs with matrix engines. HiNa-MoE (1) exploits AMX with an optimized micro-kernel that keeps expert weights in standard layouts and instead fuses lightweight layout transforms into token gathering and stores; (2) applies NUMA-aware task partitioning that minimizes cross-socket traffic under a simple page-interleaved policy without modifying the framework allocator; and (3) converts decode-phase memory-bound matrix–vector operations into small matrix–matrix execution to better utilize AMX. Across multiple server-class CPUs and real-world MoE models, HiNa-MoE achieves up to 3.37× speedup for FFN kernels and up to 2.09× end-to-end inference speedup over state-of-the-art baselines, while remaining plug-and-play with existing frameworks and deployment workflows.

KEYWORDS

Mixture-of-Experts, CPU Inference, NUMA Optimization, AMX, Non-Intrusive Design

ACM Reference Format:
Weiling Yang, Junwen Zhang, Dezun Dong, Jianbin Fang, Enda Yu, Zhe Bai, and Xiaopeng Deng. 2026. HiNa-MoE: High-Performance, Non-Intrusive MoE Inference on CPUs with Matrix Engines. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19-22, 2026, Chicago, IL, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3838684.3846846

1 INTRODUCTION

Mixture-of-Experts (MoE) large language models (LLMs) offer an appealing quality–efficiency trade-off by using conditional computation: each token activates only a small subset of experts in the feed-forward network (FFN) layers [20, 24, 28, 35]. This sparsity reduces arithmetic work relative to dense LLMs while preserving high model capacity, and it has become a common design point in production models such as Mixtral [8], Qwen-MoE [12], and DeepSeek [1].

*Equal Contribution
†Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '26, October 19-22, 2026, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846846

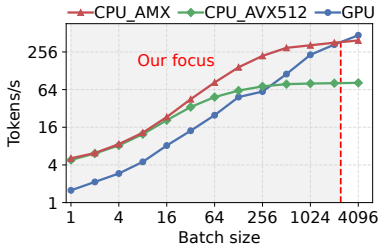


Figure 1: Decoding throughput of CPU- and GPU-based FFN execution for Mixtral-8x7B across batch sizes. CPU execution remains highly competitive over a broad batch regime. Results are measured on an Intel Xeon Gold 6430 and an NVIDIA A6000.

Problem scope. In local and on-premise deployments, MoE inference increasingly runs on CPU–GPU nodes with limited GPU memory [3, 15, 18, 23, 29, 38, 39]. Since the complete set of expert parameters can exceed GPU memory capacity, practical systems often keep some or all routed experts in CPU DRAM and execute different experts on the CPU and GPU [3, 18, 29, 38]. Thus, MoE inference in this setting is inherently a heterogeneous CPU–GPU execution problem: depending on the inference workload, a routed expert may be executed either on the CPU or on the GPU.

A natural strategy is to execute routed experts on the GPU whenever they are activated. However, the benefit of GPU execution depends strongly on the number of tokens processed by each expert. At large batch sizes, the cost of transferring expert weights can be amortized over more tokens, making GPU execution attractive [17, 30]. In contrast, during low-concurrency inference, each expert typically processes only a small number of tokens, while its weights remain large. Repeatedly transferring expert weights from CPU memory to the GPU can therefore incur substantial data-movement overhead relative to the computation [18, 29, 38]. In such cases, retaining routed experts in CPU memory and executing their FFNs on the CPU can be preferable. Figure 1 illustrates this trade-off: CPU execution remains competitive at small and moderate batch sizes, whereas GPU execution becomes increasingly advantageous as the batch size grows.

This workload-dependent execution behavior defines the scope of our work. We do not attempt to redesign the CPU–GPU offloading or scheduling policy. Instead, we focus on the CPU execution path, which remains critical when routed experts are retained in CPU memory under low-concurrency workloads. Our goal is to make CPU-side expert execution highly efficient and portable, while preserving the standard weight layouts required by existing GPU libraries. In this way, HiNa-MoE serves as a CPU-side building block for heterogeneous MoE inference: existing or future offloading policies can continue to choose between CPU and GPU execution, while HiNa-MoE provides an efficient CPU backend when CPU execution is selected.

Challenge. CPU-based MoE inference is constrained by (1) small/irregular matrix multiplications (few tokens per expert) and (2) cross-socket memory traffic when expert weights reside in CPU DRAM. To exploit matrix engines such as Intel AMX, systems including KTransformers [18] and Intel Extension for PyTorch (IPEX) [3] store expert weights in AMX-specific layouts. While effective for CPU execution, such layouts are incompatible with standard GPU libraries, complicating CPU–GPU offloading and forcing reconversion, duplicated storage, or specialized kernels. Meanwhile, to mitigate cross-NUMA traffic, KTransformers further partitions weight tensors across NUMA nodes, requiring deep modifications to the inference framework and its memory allocation behavior. Together, these choices improve speed but reduce portability and raise the engineering barrier for real deployments.

Our approach. We present HiNa-MoE, a CPU-optimized library for MoE operators that delivers high performance while remaining **non-intrusive**: it requires no offline weight reformatting, no framework-level weight sharding, and no custom NUMA-aware allocator (Section 3.1). HiNa-MoE preserves general-purpose matrix layouts (row- or column-major) and requires no changes to the framework's weight storage or memory allocation policies; adopting it amounts to replacing a small, well-defined set of operator implementations.

Contributions. HiNa-MoE achieves these goals via three complementary techniques:

- **AMX-efficient micro-kernels for general layouts**. Based on observations of MoE FFN computation patterns and AMX data layout constraints, we design micro-kernels that keep expert weights in standard layouts and instead fuse lightweight layout transforms into token gathering and stores ("transform intermediate matrix, not weight matrix"), enabling efficient AMX utilization without any weight reformatting (Section 4).
- **NUMA-aware task partitioning without framework modifications**. Leveraging the fact that expert dimensions in modern MoE models are typically multiples of 128 [1, 8, 12], we propose a task-partitioning strategy that reduces cross-socket traffic under a simple page-interleaved policy (e.g., `numactl -interleave=all`) without restructuring weight storage (Section 5).
- **Attention optimization via MV-to-MM conversion**. During decoding, we convert memory-intensive matrix–vector operations into compute-intensive matrix–matrix execution to better utilize AMX (Section 6).

Results. We evaluate HiNa-MoE on three representative multi-core CPUs: Intel Xeon Gold 6430 [6], 6448H [7], and Max 9462 [5]. Across real-world MoE models [1, 8, 12], HiNa-MoE consistently outperforms state-of-the-art baselines [2, 3, 18], achieving up to 3.37× FFN-kernel speedup and up to 2.09× end-to-end inference speedup. These results make HiNa-MoE a practical and portable building block for high-performance MoE inference on CPU–GPU nodes.

2 BACKGROUND AND RELATED WORK

2.1 MoE Architecture

MoE models are built by stacking Transformer layers [17], where each layer consists of an attention module followed by a feed-forward network (FFN). In standard dense models (e.g., OPT [40] and LLaMA-3 [31]), every token is processed by the same FFN. In contrast, an MoE layer replaces this single FFN with multiple independent expert FFNs and uses a gating function to route each token to only a small subset of experts. As shown in Figure 2, tokens

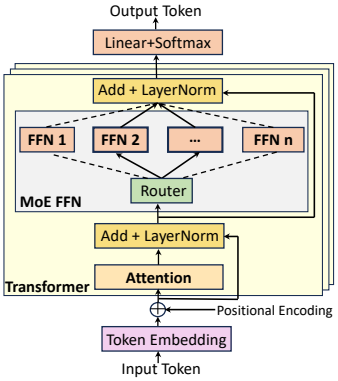


Figure 2: The architecture of MoE model.

are first processed by attention and then dispatched to the selected experts, which introduces structured sparsity in both computation and memory traffic.

2.2 Advanced CPU Compute Unit

Modern CPUs integrate matrix engines such as Intel AMX [13] and Arm SME [10], which provide substantially higher throughput than SIMD units (e.g., AVX-512) for matrix workloads. However, these gains often rely on specific data layouts and incur packing/transform overheads [18]. In this work, we focus on AMX, which is widely available in current server CPUs (where SME is not yet broadly deployed).

Figure 3 illustrates the BF16 fused multiply–add (FMA) workflow of AMX. For simplicity, matrices are shown as 4×16 bytes. Each core provides eight tile registers (`tmm0`~`tmm7`), each of size 16×64 bytes. In the figure, matrix **A** matches the layout expected by `tmm0` (row-major, reduction along columns), which is also a common layout in both CPU and GPU libraries. In contrast, matrix **B** does not match the layout required by `tmm1`; consequently, **B** is typically transformed into an AMX-specific layout prior to computation. We refer to the layout expected by `tmm0` as the *general layout* and the layout expected by `tmm1` as the *AMX-specific layout*.

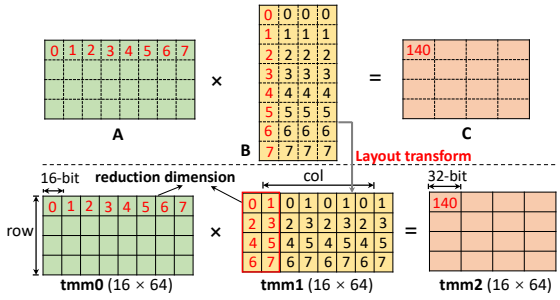


Figure 3: The computational process of AMX's BF16-precision FMA instructions (all matrices stored in row-major order).

2.3 NUMA Architecture

Non-Uniform Memory Access (NUMA) is the dominant architecture for multi-socket servers. In a NUMA system, a core can access memory attached to its local node with higher bandwidth and lower latency than memory attached to a remote node (cross-NUMA access) [4]. NUMA systems typically expose three memory allocation policies:

- *First-touch*: the default OS policy; a page is allocated on the node of the thread that first writes it. This is convenient but can lead to non-deterministic and suboptimal placement.
- *Interleaved*: pages are allocated round-robin across all nodes. This policy is deterministic and easy to deploy (e.g., via `numactl -interleave=all`) without code modifications.
- *NUMA-aware*: the application explicitly allocates pages on specific nodes to optimize locality, but this typically requires invasive code changes.

HiNa-MoE shows that, when combined with NUMA-aware task partitioning, the simple interleaved policy can mitigate cross-NUMA bottlenecks without modifying the inference framework's weight storage or memory allocator.

2.4 Related Work

In resource-constrained scenarios, MoE inference systems typically combine (i) a high-level policy for placing experts and scheduling transfers across CPU and GPU and (ii) a low-level implementation of operators (FFN and attention). We therefore organize related work along three axes: *offloading/scheduling*, *CPU operator kernels and data layouts*, and *NUMA locality management* [3, 17, 18, 22, 23, 29, 30, 34, 42].

2.4.1 Offloading and scheduling. LIA [30] adapts offloading decisions based on prompt length, batch size, and inference phase for dense LLMs. MoE-Lightning [17] pages expert weights to GPU memory and overlaps transfers with GPU FFN execution. Fiddler [29] proposes a dynamic offloading strategy and finds that substantial GPU offloading gains occur only beyond a batch-size threshold. HybriMoE designs prefetching to proactively fetch frequently invoked ("hot") experts to increase GPU residency [42]. KTransformers [18] uses a static placement strategy, retaining routed experts on the CPU while placing shared experts and attention layers on the GPU.

Key differentiation. HiNa-MoE does not propose a new offloading policy. Instead, we focus on making CPU-side operators both fast and portable so they can serve as drop-in building blocks under existing and future policies.

2.4.2 CPU operator optimization and data layouts. KTransformers [18] and IPEX [3] accelerate CPU-side MoE operators by leveraging AMX, typically through pre-converting expert weights into AMX-specific layouts (e.g., `tmm1` in Figure 3). T-MAC [34] targets low-bit dense LLM inference on embedded CPUs by transforming matrix multiply–accumulate operations into table lookups.

Why "layout-first" can be intrusive. AMX-specific layouts can improve CPU throughput, but they hinder heterogeneous execution because standard GPU libraries expect general layouts. In practice, this often forces extra weight conversions (AMX-specific layout→general layout) or the use of specialized GPU kernels. Retaining general layouts is therefore essential for seamless CPU–GPU collaborative execution of routed expert FFNs.

Table 1: Differentiation from prior work.

Name	General Layout	Cross-NUMA Optimization	Non-Intrusive
Fiddler [29]	✓	×	✓
MoE-lightning [17]	✓	×	✓
IPEX [3]	×	×	×
vLLM [14]	×	NUMA-aware pre-allocation	×
KTransformers [18]	×	NUMA-aware pre-allocation	×
HiNa-MoE	✓	**interleaved**	✓

2.4.3 NUMA locality management. On multi-socket servers, cross-NUMA bandwidth and latency can dominate MoE decoding. KTransformers reduces remote traffic by partitioning each weight tensor across NUMA nodes and using tensor-parallel execution so each socket primarily accesses local expert weights [18].

Key differentiation. KTransformers and IPEX are the closest baselines to HiNa-MoE for CPU-side MoE execution with AMX. HiNa-MoE differs in two key ways. First, instead of requiring AMX-specific *weight* layouts, we preserve general layouts and apply on-the-fly transformations to the intermediate matrix during kernel execution, maintaining compatibility with standard GPU libraries and CPU–GPU offloading frameworks. Second, on multi-socket servers, we avoid framework-level custom NUMA-aware allocators by combining OS-level page interleaving with NUMA-aware task partitioning to reduce cross-socket traffic (Table 1).

3 MOTIVATION

3.1 The Case for a Non-Intrusive Design

In local deployments, MoE inference often runs on multi-socket CPU–GPU nodes, where expert parameters reside in CPU DRAM and CPU-side execution becomes performance-critical. Meanwhile, the operator stack must remain easy to integrate across CPU-only, GPU-offload, and hybrid settings [3, 30]. We argue that this requires a **non-intrusive** design with two principles:

- **General layouts**: expert weights remain in standard row- or column-major formats compatible with CPU and GPU libraries.
- **Framework agnosticism**: no changes to weight storage, tensor semantics, or memory allocation.

These principles preserve deployment flexibility: models can move between CPU and GPU without layout conversion or duplication, and NUMA effects can be handled via OS-level mechanisms (e.g., `numactl -interleave=all`) rather than intrusive changes.

Non-intrusive criterion. A method is *non-intrusive* if it requires no (1) offline weight reformatting, (2) framework-level weight sharding/replication changes, or (3) custom NUMA-aware allocators. HiNa-MoE satisfies these by preserving general layouts, using page interleaving, and applying NUMA-aware task partitioning (Table 1).

Table 1 shows that existing systems trade off performance, portability, and complexity: some retain generality but underutilize CPU performance (e.g., Fiddler), while others achieve high performance via intrusive, hardware-specific optimizations (e.g., IPEX, KTransformers). HiNa-MoE breaks this trade-off without sacrificing deployability.

3.2 Limitations of Previous Works

To quantify the cost of this trade-off, we evaluate representative systems along two hardware-critical dimensions: (i) AMX utilization and (ii) NUMA locality on multi-socket nodes.

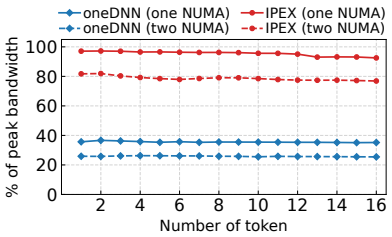


Figure 4: Memory-bandwidth utilization of MoE FFNs in existing methods (IPEX defaults to using TPP). The peak bandwidth refers to the measured peak bandwidth.

AMX layout overhead. To satisfy AMX's layout requirements, frameworks typically adopt one of two strategies. *Runtime conversion* (e.g., oneDNN) stores weights in a general layout and transforms them on the fly during expert execution; this preserves compatibility, but introduces a recurring conversion overhead. *Pre-conversion* (e.g., IPEX and KTransformers) stores weights in an AMX-specific layout, maximizing AMX throughput but making the weights incompatible with standard GPU libraries; CPU–GPU offloading then requires a costly reverse transformation, duplicated weight storage, or bespoke GPU kernels.

Cross-socket (NUMA) penalty. Figure 4 shows that memory-bandwidth utilization drops by about 20% when scaling from one to two NUMA nodes. Systems without NUMA-aware execution (e.g., IPEX and oneDNN) incur substantial cross-socket traffic, reducing effective bandwidth available to memory-bound MoE operators. KTransformers mitigates this by pre-partitioning weights across NUMA nodes to enforce locality, but doing so is intrusive and requires modifying tensor storage as well as the framework's memory allocation behavior.

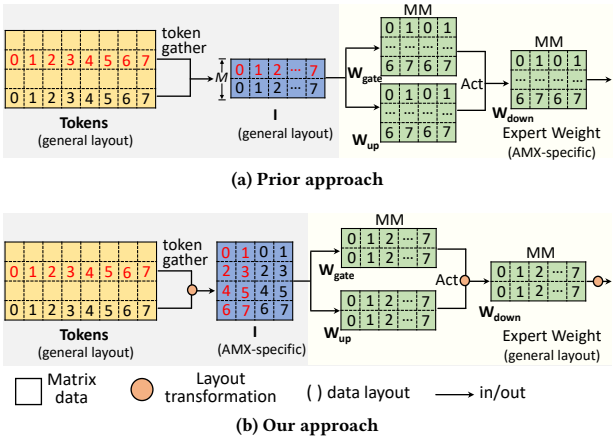
Overall, current approaches either leave performance untapped or bake in hardware-specific assumptions that reduce portability and increase engineering cost. This motivates an operator stack that delivers **high performance without intrusion**.

Motivated by this gap, HiNa-MoE is designed to pursue three goals simultaneously: (1) preserve general matrix layouts to maximize offloading flexibility, (2) fully exploit AMX compute capability, and (3) avoid cross-NUMA accesses without framework modifications.

4 EXPLOITING AMX WITH GENERAL LAYOUTS

4.1 Overview and Design Philosophy

Figure 5 illustrates the computation pattern of an expert FFN, where each expert is activated by only a subset of tokens. To improve efficiency, inference frameworks first *gather* the activated tokens into an intermediate matrix **I**, so that the expert computation becomes a small number of matrix–matrix multiplications (MM) instead of many matrix–vector multiplications. Token gathering is a standard



(a) Prior approach

(b) Our approach

Figure 5: The computation workflow of expert FFN in the MoE.

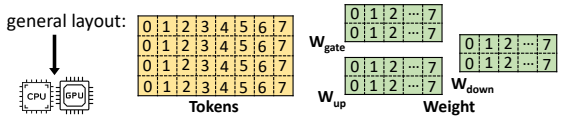


Figure 6: The native data layout of matrices in the inference framework is unified across both CPU and GPU.

step in modern MoE runtimes and is essential for efficient FFN execution on both CPU and GPU [3, 14, 18].

Given $\mathbf{I}$, the expert FFN computes two projections (with $\mathbf{W_{gate}}$ and $\mathbf{W_{up}}$), applies a nonlinearity (e.g., SiLU), combines the two branches via element-wise multiplication, and then applies $\mathbf{W_{down}}$. We write the expert FFN as

$$\mathrm{FFN}(\mathbf{I}) = \mathrm{SiLU}(\mathbf{I} \times \mathbf{W_{gate}}) \odot (\mathbf{I} \times \mathbf{W_{up}}) \times \mathbf{W_{down}}$$

where $\odot$ denotes element-wise multiplication.

Why general layouts matter. In mainstream MoE inference frameworks [11], token matrices, expert weights, and output tensors are stored in a unified, general-purpose layout (typically row-major) so the same model state can support both CPU and GPU execution (Figure 6). However, AMX reaches peak throughput only when one MM operand is in an AMX-specific (packed) layout (Section 2.2). To avoid per-inference layout conversion, IPEX and KTransformers pre-convert each expert weight matrix $\mathbf{W}$ into an AMX-specific layout (Figure 5a). While this improves CPU performance, the resulting weight format is incompatible with standard GPU libraries, which makes CPU–GPU offloading inflexible and can force reconversion or special-purpose kernels.

Our approach: transform I, not W. HiNa-MoE achieves high AMX performance while preserving the framework's general layouts (Figure 5b). This design remains largely unexplored [3, 18] due to the unclear interaction between AMX and MoE FFN computation. Instead of pre-converting weights, we perform lightweight runtime layout transformations on the intermediate matrix $\mathbf{I}$ and fuse them into data movement that the framework already must perform. Concretely, during token gathering, the selected tokens are loaded into registers and written into $\mathbf{I}$. We fuse a register-level transformation into this path so that $\mathbf{I}$ is produced directly in an AMX-specific layout (the first transformation in Figure 5b). The overhead is small because (i) $\mathbf{I}$ is much smaller than the expert weight matrices and (ii) the transform consists of only a handful of register operations.

Specifically, matrix $\mathbf{I}$ corresponds to the tile register labeled `tmm1` in Figure 3; this mapping preserves the mathematical form $\mathbf{I} \times \mathbf{W}$. After computing $\mathbf{I} \times \mathbf{W_{gate}}$ and $\mathbf{I} \times \mathbf{W_{up}}$, we similarly fuse the store path to emit outputs in the layout required by the next AMX MM (the second transformation). Finally, after multiplying by $\mathbf{W_{down}}$, we apply a register-level transpose before writing the output back in the standard row-major layout (the third transformation).

In summary, we keep expert weights in the general layout (`tmm0` in Figure 3) and align $\mathbf{I}$ with the AMX-specific operand (`tmm1`), fusing necessary transforms into unavoidable gather/store operations. This redesign is confined to the FFN operator implementation and requires no changes to the framework's matrix layouts. HiNa-MoE therefore enables the framework to fully exploit AMX without requiring the framework to be aware of AMX-specific layout constraints.

4.2 Micro-kernel Design

4.2.1 Determining Micro-kernel Size. Expert FFN performance is dominated by its three MM operators, so micro-kernel efficiency is a first-order concern. We target high efficiency by maximizing the compute-to-memory ratio (CMR) of the micro-kernel [27, 33, 37], which captures how much arithmetic we can perform per byte loaded from memory.

Many prior designs are primarily tuned for vector registers and fall back to vector units (e.g., AVX-512) when the row dimension of $\mathbf{I}$ is small; for example, KTransformers uses AMX only when the row dimension exceeds four [18]. In contrast, HiNa-MoE uses AMX for *all* row dimensions and focuses on maximizing reuse within AMX tile registers even in the decoding regime.

We describe the micro-kernel in BF16, but the same sizing methodology applies to other precisions. Let the micro-kernel tile sizes be m_r (rows), n_r (columns), and k_r (reduction). AMX tiles support up to 64 bytes along the reduction dimension, corresponding to 32 BF16 elements. Because MoE weight dimensions are typically multiples of 32 [1, 8, 12], we set $k_r = 32$ to maximize tile utilization (e.g., DeepSeek-V3 uses 7,168 and 2,048; Mixtral-8x7B uses 4,096 and 14,336).

In one micro-kernel step, AMX tiles hold $m_r \cdot 32$ elements of $\mathbf{I}$, $n_r \cdot 32$ elements of $\mathbf{W}$, and $m_r \cdot n_r$ FP32 accumulators. Each core provides eight 1,024-byte tile registers, which constrains m_r and n_r as follows:

$$\begin{cases} m_r \cdot 64 + n_r \cdot 64 + m_r \cdot n_r \cdot 4 \leqslant 8 \cdot 1024 \\ m_r \leqslant M \end{cases} \tag{1}$$

where M is the row dimension of $\mathbf{I}$. Each AMX FMA consumes BF16 inputs and accumulates into FP32 (4 bytes), which explains the $\cdot 4$ term.

Each step performs $2 \cdot m_r \cdot n_r \cdot 32$ operations (a multiplication and an addition) and loads only $\mathbf{I}$ and $\mathbf{W}$ because the accumulators remain resident in tile registers. Thus,

$$CMR = \frac{2 \cdot m_r \cdot n_r \cdot 32}{m_r \cdot 64 + n_r \cdot 64} = \frac{m_r \cdot n_r}{m_r + n_r} \tag{2}$$

Combining Equations 1 and 2, the CMR is maximized at $m_r = n_r = 32$. When $M < 32$, we set $m_r = M$ to match the available rows.

Algorithm 1: Micro-kernel for MM

```
1 for k = 0 → K step 32 do
2     (tmm0 ~ tmm1) ← I (m : m + 32, k : k + 32)          ▷ tileloadd
3     (tmm2 ~ tmm3) ← W (n : n + 32, k : k + 32)          ▷ tileloaddt1/tileloadd
4     (tmm4 ~ tmm7) ← FMA (tmm0~tmm1, tmm2~tmm3)
5 Layout transformation          ▷ the second or third layout transform in Fig. 5b
6 store (tmm4-tmm7)
```

4.2.2 Implementation Details. Algorithm 1 outlines the implementation of our AMX MM micro-kernel. When $M \leq 32$, expert weights exhibit limited temporal reuse in the cache; we therefore load **W** using the `tileloadt1` instruction. `tileloadt1` performs non-temporal loads that reduce cache pollution and do not trigger hardware prefetching, improving the effective use of main-memory bandwidth under decoding-style access patterns.

The second and third layout transformations in Figure 5b are fused into the micro-kernel (line 5 of Algorithm 1). These transformations execute within registers and the L1 data cache, and therefore introduce negligible overhead, which we also validate experimentally.

5 NON-INTRUSIVE CROSS-NUMA OPTIMIZATION

5.1 Performance Bottleneck and Opportunity

In CPU-centric MoE inference on multi-socket servers, the dominant NUMA bottleneck is remote access to expert weight matrices [18]. In contrast, accesses to the gathered token matrix **I** are typically not performance-critical. During prefill, M is determined by batch size, prompt length, and expert activation rate, and in common local deployments the batch size is small (often 1). When prompts are short, **I** is much smaller than the expert weights; when prompts are long, prefill becomes compute-bound and is commonly addressed via CPU-GPU or multi-GPU parallelism, which is outside our scope. During decoding, M equals the batch size times the activation rate, and **I** remains small.

To avoid cross-socket traffic on weights, prior systems such as KTransformers use *numa_alloc_onnode* to partition each weight matrix across NUMA nodes so that each socket primarily accesses its local portion (Figure 7(a)). However, this approach is intrusive because it requires modifying the framework's tensor storage and memory allocator.

HiNa-MoE instead relies on a standard OS mechanism: interleaving the physical pages of weight matrices across NUMA nodes (e.g., via `numactl -interleave=all`). With 4 KB pages, each page holds 2,048 BF16 elements; for example, a weight matrix with 6,144 columns occupies three pages per row (Figure 7(b)). Given this page-level interleaving, HiNa-MoE uses task partitioning so that each socket predominantly touches pages local to its NUMA node, while preserving the framework's original weight storage.

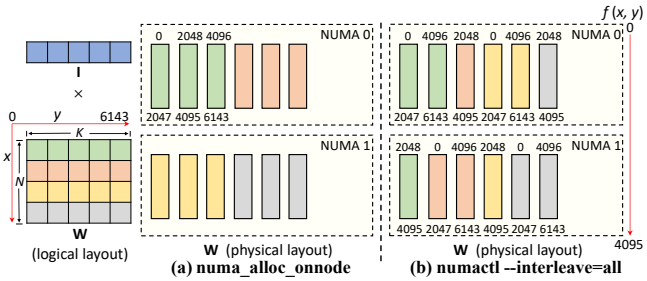


Figure 7: The physical layout of the weight matrix in main memory. The numbers in (a) and (b) denote the coordinates of the elements along the reduction dimension.

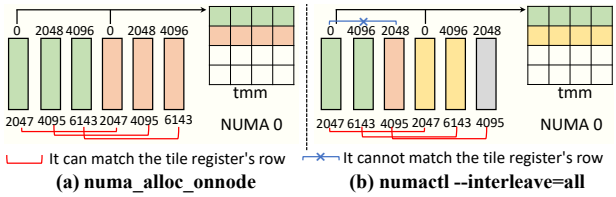


Figure 8: The workflow for mapping data segments from the NUMA 0 (Figure 7) onto tile registers.

5.2 Task Partitioning Strategy

Under an interleaved page layout, high MM performance requires two properties: (1) each socket should primarily read weight elements that are local to its NUMA node, and (2) each socket must still be able to assemble contiguous tiles that efficiently feed AMX (in particular, the operand mapped to `tmm0` in Figure 3).

With explicit NUMA placement (e.g., *numa_alloc_onnode*), these properties are straightforward to satisfy: adjacent matrix rows stored on the same node can be mapped directly to AMX tile rows (Figure 8(a)). Under page interleaving, however, adjacent rows can reside on different NUMA nodes. For example, in Figure 7(b), the first 2,048 elements of row 0 are on NUMA 0, while the first 2,048 elements of row 1 are on NUMA 1.

As a result, as shown in Figure 8(b), on NUMA 0 we may need to combine segments from non-adjacent rows to fill AMX tile rows (e.g., elements $0 \sim 2047$ of row 0 and elements $0 \sim 2047$ of row 2), while selecting multiple segments within a row to cover the reduction dimension (e.g., elements $0 \sim 2047$ and $4096 \sim 6143$ of row 0). Next, we show how to systematically identify and organize these segments.

Concretely, we address three subproblems: (1) identifying which weight segments are local to each NUMA node under interleaving; (2) selecting local segments that share the same y-range so they can form AMX tile rows; and (3) selecting local segments that share the same x so they provide contiguous ranges along the reduction dimension. Subproblem (1) enforces locality, while (2) and (3) ensure efficient utilization of AMX.

For ease of description, we define the following terms.

***x* and *y*:** the coordinates of an element in the weight matrix along the row and reduction dimensions, respectively. For example, the labels 0, 2047, and 4096 in Figure 7 indicate y-coordinates.

Data segment $(x,\ y_{start} \sim y_{end})$**:** a contiguous range of elements in the same row, from y_{start} to y_{end} along the reduction dimension. For example, the first segment in Figure 7(b) is $(0, 0 \sim 2047)$.

Function $f(x, y)$**:** the page-level "vertical" coordinate used to determine the NUMA placement under interleaving. We define

$$f(x, y) = (x \cdot K + y) \bmod (2048 \cdot N_{numa}),\ x \in [0, N),\ y \in [0, K) \quad (3)$$

Here, N and K are the row and reduction dimensions of the weight matrix, and N_{numa} is the number of NUMA nodes. The function $f(x, y)$ is periodic with period

$$\begin{aligned} T_x &= \frac{2048 \cdot N_{numa}}{\gcd(K, 2048 \cdot N_{numa})} \in [0, N), \\ T_y &= 2048 \cdot N_{numa} \in [0, K) \end{aligned} \quad (4)$$

where T_x, T_y are the periods in x and y. Based on the characteristics of the periodic function, namely $f(x, y) = f(x + T_x, y + T_y)$, it can be concluded that the data segment $(x, y_{start} \sim y_{end})$ and the data segment $(x + T_x, y_{start} + T_y \sim y_{end} + T_y)$ are guaranteed to reside on the same NUMA node. Therefore, we only need to identify the data segments within the first period of x; the remaining data segments can be derived based on the properties of the periodic function. All data on each NUMA node must satisfy the following constraints:

$$\begin{aligned} id_{numa} \cdot 2048 <= f(x, y) < (id_{numa} + 1) \cdot 2048 \\ id_{numa} \in [0, N_{numa}), x \in [0, T_x) \end{aligned} \quad (5)$$

id_{numa} denotes the id of a NUMA node.

T_x is usually very small, and this follows from two observations. First, in modern MoE models, the dimensions of expert weight matrices (i.e., `hidden_size` and `intermediate_size`) are typically multiples of 128. This is not accidental: widely used model families such as Qwen [12], DeepSeek [1], and Mixtral [8] all exhibit this property. Second, mainstream server platforms most commonly expose two NUMA nodes, while four-node systems are comparatively rare. For example, in Mixtral-8x7B, K takes values 4,096 or 14,336; on a dual-socket system, Equation 4 yields $T_x = 1$ and $T_x = 2$, respectively. Similarly, in Qwen3-30B-A3B, $K = 2,048$ or 768, corresponding to $T_x = 2$ and $T_x = 16$. Because the period T_x is small, we only need to traverse the first period of x. For each fixed x in this interval, we can identify the corresponding y-ranges. The lower and upper bounds of each such y-range are exactly the y_{start} and y_{end} values of a local data segment.

In summary, the data segments assigned to each NUMA node can be derived in two steps. First, compute T_x and T_y from the weight shape and the number of NUMA nodes using Equation 4. Second, determine the data segments for each NUMA node within the first period of x using Equation 5, which solves subproblem (1). Segments with the same x are then used to assemble contiguous ranges along the reduction dimension of the AMX tile registers, while segments sharing the same y_{start} and y_{end} are used to assemble the tile rows. This addresses subproblems (2) and (3). Taking the weight matrix in Figure 7(b) as an example, we have $T_x = 2$ and $T_y = 4096$. For NUMA 0, when $x = 0$ we obtain $y \in [0, 2048)$ and $y \in [4096, 6144)$, and when $x = 1$ we obtain $y \in [2048, 4096)$. Based on these three segments, all segments on NUMA 0 can be derived. In the implementation, we do not need to explicitly materialize the remaining segments; the kernel can index the weight tensor using only these segments together with T_x, which is sufficient to ensure correct and local computation.

Although our design leverages the facts that current MoE models use 128-aligned dimensions and that commercial servers expose only a small number of NUMA nodes, these are not arbitrary assumptions; rather, they arise naturally from both hardware constraints and common model design choices. Overall, HiNa-MoE enables the framework to minimize cross-NUMA accesses to expert weights without requiring the framework to be aware of NUMA details.

5.3 Design Details

To fit the task partitioning in Section 5.2, we make several adjustments to the micro-kernel, as shown in Algorithm 2. We use the array S_{index} to store the indices of the data segments assigned to each node, containing only the data segments within a single period of x (i.e., $x < T_x$). Each entry of the array contains three elements: x, y_{start}, and y_{end}.

Line 1 of Algorithm 2 iterates over one period of x. For a fixed x, all local data segments are processed along the reduction dimension, which is implemented by the loop in Line 3. Specifically, segments that share the same x are multiplied with the corresponding elements of **I**, and the partial results are accumulated into tile registers `tmm4~tmm7` (Lines 2 and 7). To fully utilize the two tile registers used for the **W** operand, 32 segments of **W** that share the same y coordinate are loaded from the local NUMA node in each iteration (Line 6), where the x coordinate of adjacent segments differs by T_x. Since the reduction dimension of **W** may span multiple NUMA nodes (Figure 7(b)), each node computes a partial result using only its local segments. After all NUMA nodes finish, the partial results are aggregated to produce the final output.

Algorithm 2: NUMA-optimized micro-kernel for MM

```
1 for x = 0 → T_x step 1 do
2     (tmm4 ~ tmm7) ← 0                                  ▷ zeroing for accumulation
3     for (y_start, y_end) = S_index[x](y_start, y_end) → K step T_y do
4         for y = y_start → y_end step 32 do
5             (tmm0 ~ tmm1) ← I (m : m + 32, y : y + 32)
              /* load 32 data segments                                   */
6             (tmm2 ~ tmm3) ← W (x : x + 32 · T_x, y : y + 32)
7             (tmm4 ~ tmm7) ← FMA (tmm0~tmm1, tmm2~tmm3)
```

6 ATTENTION MODULE OPTIMIZATION

The attention module consists of (i) input projections (three matrix multiplications, producing **Q**, **K**, and **V**), (ii) scaled dot-product attention (SDPA), and (iii) an output projection (one matrix multiplication). The projection MMs are optimized by our AMX micro-kernels (Section 4) together with the NUMA-aware task partitioning strategy (Section 5.2), and therefore we focus here on SDPA. Prior work has largely optimized SDPA for the prefill phase [19, 25], where computation dominates and batching is higher. We therefore target SDPA optimization in decoding.

6.1 SDPA Module Optimization

Modern MoE models commonly use grouped-query attention (GQA) [16] to reduce KV-cache size while maintaining accuracy

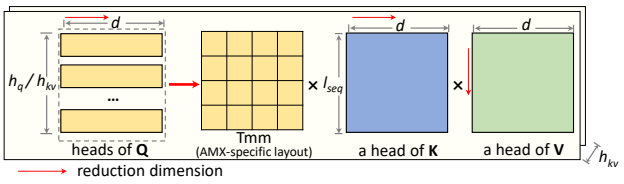


Figure 9: The computational workflow of SDPA during the decoding phase.

Table 2: Main characteristics of each platform

	6430	6448H
Peak BF16 TFLOPS	134.4	300.4
Cores	32x2	32x4
Frequency	2.1 GHz	2.4 GHz
Number of NUMA nodes	2	4
Local Bandwidth	234 GB/s	158 GB/s
Cross-NUMA Bandwidth	95 GB/s	32 GB/s

(e.g., Qwen and Mixtral). During decoding, the query **Q** is split into h_q heads, where each head is a length-d vector. The key **K** and value **V** are split into h_{kv} heads, where each head is a matrix of size $l_{seq} \times d$ (Figure 9). Here, d denotes the head dimension and l_{seq} denotes the current sequence length.

In GQA, each key/value head is shared by a group of $\frac{h_q}{h_{kv}}$ query heads. Consequently, computing attention scores for one KV head can be expressed as multiplying a small block of query heads (shape $\frac{h_q}{h_{kv}} \times d$) with the corresponding key head (shape $l_{seq} \times d$), i.e., a MM. The same structure appears again when multiplying attention probabilities by the value head, and thus the decode-phase SDPA kernel naturally exposes a sequence of small MMs.

To exploit AMX in these decode-phase MMs, HiNa-MoE performs lightweight, on-the-fly layout transformations on the small query-head block and keeps the transformed block resident in cache across the subsequent multiplies. Because $\frac{h_q}{h_{kv}}$ is small (e.g., 8 in Qwen3-30B-A3B [12]), the transformation cost is negligible relative to the MM work. For the subsequent multiplication with **V**, HiNa-MoE similarly converts the accessed value-head tiles into an AMX-specific layout on demand, enabling high AMX utilization without changing the framework's native tensor layouts.

7 EXPERIMENTAL SETUP

7.1 Hardware Details

We evaluate HiNa-MoE on two modern multi-socket server platforms: a dual-socket Intel Xeon Gold 6430 (**6430**) [6] and a quad-socket Intel Xeon Gold 6448H (**6448H**) [7] (Table 2). Memory bandwidth is measured using STREAM [4]. The local (per-socket) bandwidth and cross-socket bandwidth quantify the cost of NUMA-remote accesses. Although both CPUs support eight memory channels per socket, our 6448H machine is populated with only six memory modules per socket, resulting in lower measured local bandwidth than the fully populated 6430. The 6430 system also includes an NVIDIA RTX A6000 GPU [9] for hybrid CPU–GPU experiments [18], whereas the 6448H system is CPU-only.

Table 3: Expert weight matrix dimensions and routing strategies of evaluated MoE models

	Gate, Up	Down	Routing Strategy
Mixtral-8x7B	$4,096 \times 14,336$	$14,336 \times 4,096$	Top-2
Mixtral-8x22B	$6,144 \times 16,384$	$16,384 \times 6,144$	Top-2
DeepSeek-V3	$7,168 \times 2,048$	$2,048 \times 7,168$	Top-8
Qwen3-235B	$4,096 \times 1,536$	$1,536 \times 4,096$	Top-8

7.2 Competitive Baselines

We compare HiNa-MoE against four widely used CPU(-centric) inference stacks. All baseline configurations follow the official recommendations.

IPEX [3]: Intel Extension for PyTorch, which optimizes PyTorch execution on Intel CPUs. By default, it uses Tensor Processing Primitives (TPP) [26, 32].

KTransformers [18]: A framework for locally deployed MoE models with static CPU–GPU offloading: routed experts run on the CPU, while attention and shared experts run on the GPU. We use the `SOSP'25` release.

vLLM [14]: A widely used LLM serving framework; we use its CPU-only variant [15].

oneDNN [2]: Intel's high-performance DNN primitive library; we use oneDNN `3.10.1`.

7.3 Workloads

We evaluate HiNa-MoE at two levels: **single-operator (FFN) kernels** and **end-to-end** model inference.

FFN Kernel Evaluation. We extract the expert-matrix shapes from four MoE models: Mixtral-8x7B, Mixtral-8x22B [8], DeepSeek-V3 [1], and Qwen3-235B-A22B [12]. Their expert dimensions and routing configurations are summarized in Table 3. We report the geometric mean over 100 runs.

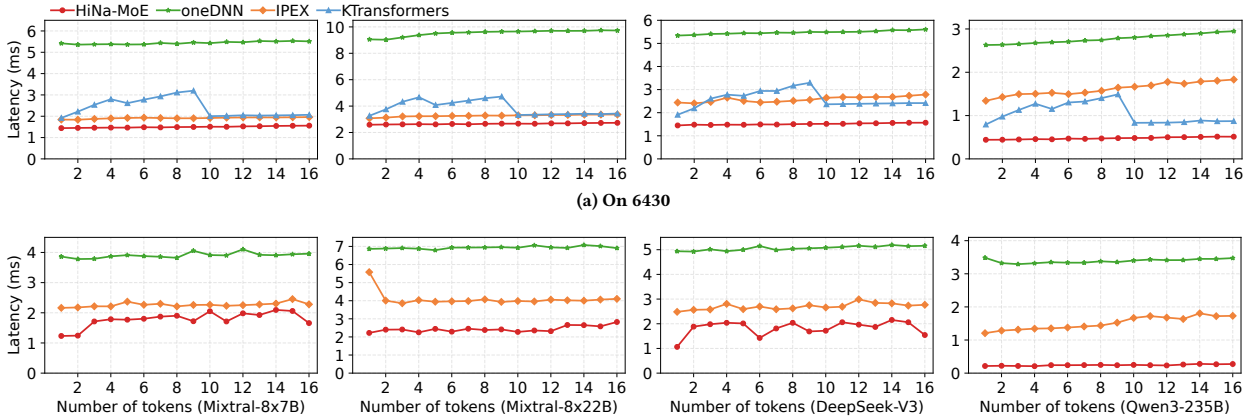
End-to-End Inference Evaluation. We measure end-to-end inference latency for Mixtral-8x7B, Qwen3-30B-A3B, and Qwen2-57B-A14B [36]. We integrate HiNa-MoE by replacing the corresponding CPU operators in IPEX and KTransformers, yielding two configurations: *HiNa-MoE (IPEX)* and *HiNa-MoE (KTrans)*. Unless otherwise stated, we use batch size 1 to reflect low-concurrency, latency-oriented deployments and to match prior work [18, 34]. We report the geometric mean over 5 runs.

8 EVALUATION RESULTS

8.1 FFN Kernel Performance

We begin by evaluating the core MoE FFN operator under varying per-expert token counts, using the exact expert matrix shapes of real-world MoE models (Table 3).

Figure 10 shows that HiNa-MoE consistently outperforms all baselines. Averaged across all workloads, it achieves 1.73× and 1.68× speedups over IPEX and KTransformers, respectively, with maximum observed speedups of 3.37× and 2.16×. On the 6430, HiNa-MoE reaches up to 96% of the measured peak local memory bandwidth, suggesting that performance is close to the platform's bandwidth ceiling. These gains come from (i) NUMA-aware task partitioning that avoids remote weight traffic and (ii) AMX



(a) On 6430

(b) On 6448H

Figure 10: Performance of the FFN operator on the 6430 and 6448H platforms. The X-axis represents the number of tokens assigned to each expert, and the Y-axis represents the latency (in ms) of the MoE FFN.

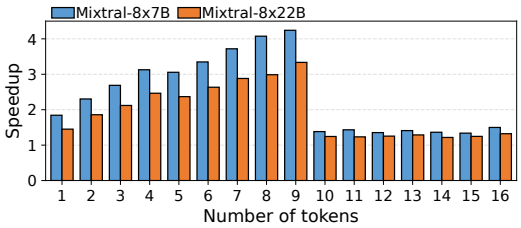


Figure 11: Speedup of HiNa-MoE over KTransformers for the INT8-quantized FFN kernel on the dual-socket 6430 platform.

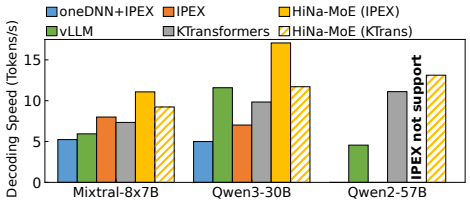


Figure 12: Comparison of decoding speed between HiNa-MoE and the state-of-the-art baselines.

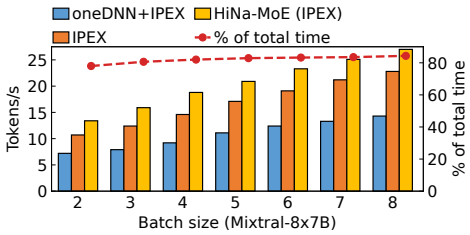


Figure 13: Comparison of batch decoding performance (left: throughput; right: % of total runtime for optimized FFNs).

micro-kernels that sustain high utilization while preserving general-purpose layouts. In contrast, oneDNN is slower due to the need to convert the weight matrices into an AMX-specific layout.

MoE FFN latency is largely insensitive to small changes in per-expert token count (e.g., 1∼16 tokens). The dominant cost is streaming large expert weights from memory, whereas the gathered-token matrix **I** contributes little to the total traffic. KTransformers shows a slight throughput drop around 4 tokens, as it uses AMX only when an expert processes more than 4 tokens; otherwise, it falls back to AVX-512 [18]. HiNa-MoE uses AMX efficiently even for single-token decoding.

HiNa-MoE's advantage is more pronounced on the 6448H (4 sockets) than on the 6430 (2 sockets) because the 6448H has much lower cross-socket bandwidth (32 GB/s vs. 95 GB/s). Avoiding remote weight traffic therefore yields larger relative benefits as NUMA topologies become more complex.

We further evaluate the INT8-quantized FFN kernel. For a fair comparison, both HiNa-MoE and KTransformers use the same symmetric group-wise quantization scheme. As shown in Figure 11, HiNa-MoE achieves an average speedup of 2.18× over KTransformers, demonstrating the effectiveness of our approach under INT8 precision as well.

8.2 End-to-end Inference Performance

Decoding Phase. We first evaluate end-to-end decoding performance for Mixtral-8x7B, Qwen3-30B-A3B, and Qwen2-57B-A14B with batch size 1, using 128 input and output tokens on a dual-socket 6430 CPU + RTX A6000 system. Figure 12 shows that HiNa-MoE achieves the highest decoding throughput, with average speedups of 1.99×, 1.70×, and 1.22× over vLLM (CPU-only), IPEX, and KTransformers, respectively, and maximum speedups of 2.87×, 2.09×, and 1.25×. For IPEX and KTransformers, we use their HiNa-MoE-integrated counterparts to isolate the impact of the CPU operator implementation. These gains are consistent with our FFN-kernel analysis: decoding is dominated by streaming expert weights from memory, making memory bandwidth and kernel efficiency critical. HiNa-MoE reduces cross-socket traffic through NUMA-aware task

Table 4: Prefill Throughput on Qwen2-57B-A14B (tokens/s)

Input Length	SGL + HiNa-MoE + GPU Scheduling	SGL + HiNa-MoE	SGL + KTransformers
2K	1526.66	1293.36	1187.43
4K	2065.21	1348.27	1317.39
8K	1977.82	1357.80	1377.77
16K	2134.25	1221.91	1343.52
32K	2085.47	1187.70	1252.54

partitioning and improves kernel efficiency for small per-expert token counts compared with IPEX and KTransformers, respectively.

The results also highlight the benefit of preserving a general-purpose weight layout. Both oneDNN and HiNa-MoE keep expert weights in the general layout, but oneDNN performs weight-layout transformation at runtime for each inference, whereas HiNa-MoE fuses lightweight transformations into the existing token gather/store paths. Thus, HiNa-MoE avoids the recurring transformation overhead while retaining compatibility with standard CPU–GPU execution.

To examine whether these benefits persist beyond batch size 1, Figure 13 further reports decoding performance across different batch sizes. HiNa-MoE consistently outperforms IPEX over the evaluated range. This is expected because expert FFNs account for approximately 80% of the decoding runtime, making the optimized FFN execution the dominant contributor to end-to-end performance.

Prefill Phase. We next evaluate long-context prefill by integrating different CPU operator backends into SGLang (SGL) [41] on the 6430 CPU + A6000 GPU system, using Qwen2-57B-A14B with batch size 1. During prefill, GQA attention runs on the GPU, while expert FFNs primarily run on the CPU. Table 4 reports the prefill throughput across input sequence lengths.

Without GPU scheduling, SGL+HiNa-MoE achieves performance comparable to SGL+KTransformers, showing that preserving the general-purpose weight layout does not sacrifice CPU-side execution efficiency. More importantly, this general layout also allows GPU scheduling without requiring expert-weight reconversion. We emphasize that we do not introduce a new scheduling policy; rather, HiNa-MoE's layout compatibility enables SGLang to schedule expert execution on the GPU while retaining the same CPU backend. With GPU scheduling, SGL+HiNa-MoE consistently outperforms SGL+KTransformers across all evaluated sequence lengths, achieving speedups from 1.29× at 2K to 1.67× at 32K. These results demonstrate that HiNa-MoE provides both efficient CPU execution and the layout flexibility required for heterogeneous CPU–GPU inference.

Both SGL+HiNa-MoE and SGL+KTransformers exhibit a similar trend as the sequence length increases: their prefill throughput improves from 2K to 8K, peaks at 8K, and gradually decreases at 16K and 32K. This trend arises from the different scaling behaviors of GPU-side GQA attention and CPU-side expert FFN. As shown in Table 5, increasing the sequence length from 2K to 8K improves the effective computational performance of both operators, with attention increasing from 97.88 to 119.97 TFLOPS and FFN from 13.44 to 15.86 TFLOPS, indicating improved hardware utilization. Beyond 8K, however, the performance gains become marginal as both operators approach saturation. Meanwhile, GQA attention grows quadratically with the sequence length ($O(l^2)$), whereas the

Table 5: Effective computational performance of GPU-side attention and CPU-side expert FFN in SGL+KTransformers across different input sequence lengths.

Input Length	Attention (GPU)		Expert FFN (CPU)	
	Time (ms)	TFLOPS	Time (ms)	TFLOPS
2K	1.536	97.88	67.124	13.44
4K	3.224	111.91	119.981	15.04
8K	8.020	119.97	227.537	15.86
16K	23.021	125.38	452.798	15.93
32K	85.466	112.57	919.989	15.69

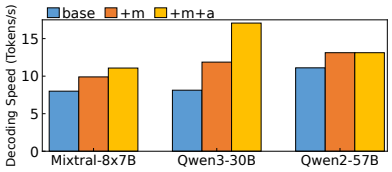


Figure 14: Ablation study on the decoding phase.

expert FFN grows approximately linearly. As a result, the attention latency increases rapidly from 8.02 ms at 8K to 85.466 ms at 32K despite nearly saturated execution performance. Consequently, the rapidly increasing attention cost can no longer be offset by improved execution efficiency. Although FFN remains the dominant component, the growing attention overhead reduces the benefit of processing longer sequences, causing prefill throughput to decrease.

8.3 Ablation Study

We analyze the contribution of each component: the micro-kernel (+m), including FFN optimizations for both the general layout and page-interleaving, and the decode-phase attention optimization (+a). Figure 14 reports the decoding throughput improvement brought by each component at a batch size of 1, with IPEX serving as the baseline for Mixtral-8x7B and Qwen3-30B-A3B, and KTransformers for Qwen2-57B-A14B.

Both optimizations improve performance, with micro-kernel optimization (+m) providing the largest gain and improving performance by 27% on average over the baseline. For Qwen2-57B-A14B, attention optimization provides no additional benefit because KTransformers offloads attention computation to the GPU, whereas (+a) targets the CPU-side attention path. In contrast, Qwen3-30B-A3B exhibits a more pronounced benefit from (+a), as its attention accounts for a larger fraction of the decoding computation than in Mixtral-8x7B, making CPU-side attention optimization more impactful on end-to-end performance. This result highlights that the effectiveness of attention optimization depends on both the relative cost of attention and the underlying offloading strategy.

8.4 Cross-NUMA Data Traffic Analysis

Using Linux `perf`, we quantify cross-socket traffic by measuring the fraction of L3-miss loads served by remote DRAM (`mem_load_l3_miss_retired.remote_dram`) among all DRAM-served L3-miss loads. Figure 15 shows that HiNa-MoE's remote-access fraction is nearly zero, validating that our task partitioning (Section 5) largely eliminates remote accesses to weights. KTransformers also shows low remote traffic due to its manual weight partitioning. While oneDNN and IPEX exhibit similar remote fractions,

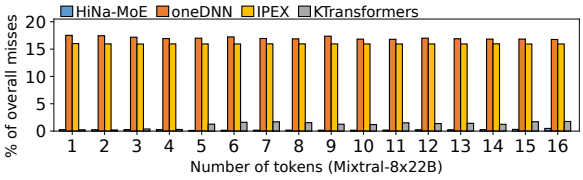


Figure 15: The proportion of remote L3 cache misses to overall L3 cache misses.

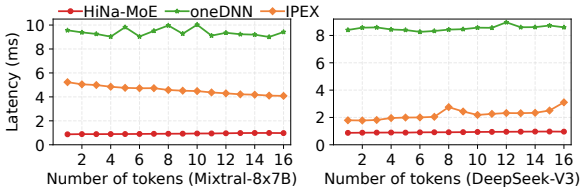


Figure 16: Performance of the MoE FFN on the 9462 CPU.

oneDNN additionally pays runtime overhead for weight-layout transformations, leading to worse overall performance.

8.5 Evaluation on an HBM-based CPU

We evaluate FFN-kernel performance on a dual-socket Intel Xeon CPU Max 9462 (**9462**) [5]. Each socket provides 64 GB of HBM (HBM-only mode). The measured peak bandwidth is 853 GB/s; local NUMA accesses reach 425 GB/s, while cross-socket accesses drop to 89 GB/s.

Because HBM capacity is insufficient for full end-to-end MoE inference on our evaluated models, we report FFN-kernel results only. Figure 16 shows that HiNa-MoE outperforms oneDNN and IPEX by 9.65× and 3.41× on average. The larger gap on 9462 is explained by the higher penalty of remote accesses: the local-to-remote bandwidth ratio is ~5× on 9462 versus ~2.3× on 6430. As the cost of cross-socket accesses increases, HiNa-MoE's NUMA-aware scheduling yields correspondingly larger benefits.

8.6 Integrating HiNa-MoE into MoE Inference Frameworks

Figure 17 summarizes the workflow of existing MoE inference frameworks. Integrating HiNa-MoE requires only two changes: (1) skip the weight pre-conversion step used by implementations based on the AMX-specific layout, and (2) replace the framework's CPU-side attention and FFN operators with HiNa-MoE implementations. For example, in IPEX we replace the *mixtral_moe_tpp_kernl_impl* function responsible for FFN computation; in total, integration requires modifying about 30 lines of framework code.

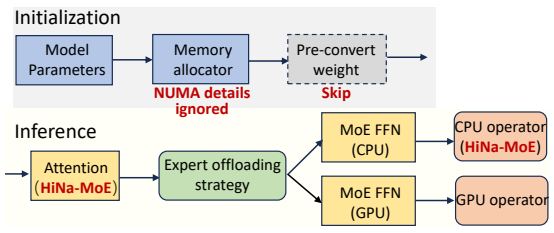


Figure 17: Workflow of the MoE inference framework.

9 DISCUSSION AND FUTURE WORK

Our evaluation shows that HiNa-MoE achieves consistent speedups across multi-socket servers and real-world MoE models. While the current implementation targets latency-oriented inference at small batch sizes on AMX-capable CPUs, the underlying design principles generalize.

Adaptation to other CPU architectures. HiNa-MoE's core idea—preserve general-purpose weight layouts and adapt the computation kernels—extends beyond AMX. Prior work suggests that Arm's SME can support tile-oriented micro-kernels similar in spirit to AMX [21]. A natural direction is to apply the same "transform intermediate matrix **I**, not **W**" philosophy to SME by fusing lightweight layout transformations into token gathering and kernel execution, with tile sizes chosen to match the available register space and instruction semantics.

Multi-batch and CPU–GPU collaborative inference. For larger batch sizes and long-prompt prefill, CPU-only routed-expert execution can become a bottleneck because GPU compute and bandwidth are typically higher [17, 30]. A promising direction is dynamic expert offloading based on prompt length, batch size, and available GPU memory, overlapping transfers with CPU computation [38]. Because HiNa-MoE preserves general-purpose layouts and avoids framework-level allocator changes, experts can move between CPU and GPU without layout-conversion overhead, making it a natural substrate for such schedulers.

10 CONCLUSION

We have presented HiNa-MoE, a high-performance, non-intrusive operator library for MoE inference on multi-socket, multi-core CPUs. It preserves general-purpose matrix layouts for compatibility with CPU–GPU offloading frameworks, while improving performance through (1) AMX-efficient micro-kernels that fuse token gathering with lightweight, on-the-fly data movement, (2) NUMA-aware task partitioning that exploits the periodicity of interleaved pages without modifying the framework allocator, and (3) decode-phase attention optimization that converts memory-bound matrix–vector operations into compute-intensive matrix–matrix multiplications.

Across multiple server-class CPUs and real-world MoE models, HiNa-MoE achieves up to 3.37× FFN-kernel speedup and up to 2.09× end-to-end MoE inference speedup over state-of-the-art baselines, while maintaining near-zero cross-socket traffic and plug-and-play integration.

ACKNOWLEDGMENTS

We thank Guangnan Feng and Zheng Wang for their helpful discussions. This work is supported by the National Natural Science Foundation of China under Grant No.62421002 and No.U24B20151, and the Hunan Provincial Natural Science Foundation of China under Grant No. 2026JJ20082.

REFERENCES

[1] 2025. DeepSeek v3: Advanced AI Language Model. https://deepseekv3.org/.
[2] 2025. Intel oneDNN. https://github.com/oneapi-src/oneDNN.
[3] 2025. Intel® Extension for PyTorch*. https://github.com/intel/intel-extension-for-pytorch.

[4] 2025. Intel® Memory Latency Checker v3.12. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[5] 2025. Intel® Xeon® CPU Max 9462 Processor. https://www.intel.com/content/www/us/en/products/sku/232597/intel-xeon-cpu-max-9462-processor-75m-cache-2-70-ghz/specifications.html.

[6] 2025. Intel® Xeon® Gold 6430 Processor. https://www.intel.com/content/www/us/en/products/sku/231737/intel-xeon-gold-6430-processor-60m-cache-2-10-ghz/specifications.html.

[7] 2025. Intel® Xeon® Gold 6448H Processor. https://www.intel.com/content/www/us/en/products/sku/232391/intel-xeon-gold-6448h-processor-60m-cache-2-40-ghz/specifications.html.

[8] 2025. MistralAI. https://mistral.ai/news/mixtral-8x22b/.

[9] 2025. NVIDIA RTX A6000. https://www.nvidia.com/en-us/products/workstations/rtx-a6000/.

[10] 2025. Part 1: Arm Scalable Matrix Extension (SME) Introduction. https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction.

[11] 2025. PyTorch. https://github.com/pytorch/pytorch.

[12] 2025. Qwen-3: Explore the Next-Generation Open Source Large Model. https://qwen-3.com/en.

[13] 2025. What Is Intel® Advanced Matrix Extensions (Intel® AMX)? https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html.

[14] 2026. vLLM. https://github.com/vllm-project/vllm.

[15] 2026. vllm-cpu-amxbf16 0.13.0. https://pypi.org/project/vllm-cpu-amxbf16/0.13.0/.

[16] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 4895–4901.

[17] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. 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, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 715–730.

[18] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. 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, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*. ACM, 1014–1029.

[19] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[20] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024).

[21] Chencheng Deng, Weiling Yang, Jianbin Fang, and Dezun Dong. 2026. Demystifying ARM SME to Optimize General Matrix Multiplications. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2026, New Orleans, LA, USA, May 25-29, 2026*. IEEE, 1233–1245.

[22] Hongchao Du, Shangyu Wu, Arina Kharlamova, Nan Guan, and Chun Jason Xue. 2025. FLEXINFER: FLEXIBLE LLM INFERENCE WITH CPU COMPUTATIONS. In *Proceedings of Machine Learning and Systems 2025, MLSys 2025, Santa Clara, CA, USA, August 29 - September 1, 2025*. mlsys.org.

[23] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. 2025. Klotski: Efficient Mixture-of-Expert Inference via Expert-Aware Multi-Batch Pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM, 574–588.

[24] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* 23 (2022), 120:1–120:39.

[25] Xiao Fu, Weiling Yang, Dezun Dong, and Xing Su. 2024. Optimizing Attention by Exploiting Data Reuse on ARM Multi-core CPUs. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*. ACM, 137–149.

[26] Evangelos Georganas, Dhiraj D. Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Jeremy Bruestle, Narendra Chaudhary, Abhisek Kundu, Denise Kutnick, Frank Laub, Md. Vasimuddin, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. 2021. Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 14.

[27] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. IEEE Computer Society, 981–991.

[28] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. *CoRR* abs/2401.04088 (2024).

[29] Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. 2025. Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

[30] Hyungyo Kim, Nachuan Wang, Qirong Xia, Jinghan Huang, Amir Yazdanbakhsh, and Nam Sung Kim. 2025. LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 544–558.

[31] Llama Team. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024).

[32] Arun Thangamani, Md Asghar Ahmad Shahid, Adam Siemieniuk, Rolf Morel, Renato Golin, and Alexander Heinecke. 2025. Library Liberation: Competitive Performance Matmul Through Compiler-composed Nanokernels. *CoRR* abs/2511.13764 (2025).

[33] Feng Wang, Hao Jiang, Ke Zuo, Xing Su, Jingling Xue, and Canqun Yang. 2015. Design and Implementation of a Highly Efficient DGEMM for 64-Bit ARMv8 Multi-core Processors. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE Computer Society, 200–209.

[34] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. 2025. T-MAC: CPU Renaissance via Table Lookup for Low-Bit LLM Deployment on Edge. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 278–292.

[35] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jian Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *CoRR* abs/2505.09388 (2025).

[36] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 Technical Report. *CoRR* abs/2407.10671 (2024).

[37] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 72.

[38] Enda Yu, Dezun Dong, Zhaoning Zhang, Zhe Bai, Weiling Yang, Haojie Wang, Dongsheng Li, Yongwei Wu, and Xiangke Liao. 2026. LayerScope: Predictive Cross-Layer Scheduling for Efficient Multi-Batch MoE Inference on Legacy Servers. In *Proceedings of the 40th ACM International Conference on Supercomputing, ICS 2026, Belfast, United Kingdom, July 6-9, 2026*. ACM, 881–892.

[39] Junwen Zhang, Weiling Yang, Jianbin Fang, Enda Yu, and Dezun Dong. 2026. AsymFlow: Enabling Long-Context LLM Serving via CPU-GPU Prefill-Decode Disaggregation. In *55th International Conference on Parallel Processing, ICPP 2026*. ACM, 359–369.

[40] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *CoRR* abs/2205.01068 (2022).

[41] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez,

Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[42] Shuzhang Zhong, Yanfan Sun, Ling Liang, Runsheng Wang, Ru Huang, and Meng Li. 2025. HybriMoE: Hybrid CPU-GPU Scheduling and Cache Management for Efficient MoE Inference. In *62nd ACM/IEEE Design Automation Conference, DAC 2025, San Francisco, CA, USA, June 22-25, 2025*. IEEE, 1–7.