EasyChair Preprint

№ 16034

# NearSEE: Near-Storage Expert Execution for Edge Inference of Mixture-of-Experts LLMs

Prapti Panigrahi, Wahid Uz Zaman, Mahmut Kandemir and Vijaykrishnan Narayanan

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

November 11, 2026

# NearSEE: Near-Storage Expert Execution for Edge Inference of Mixture-of-Experts LLMs

Prapti Panigrahi*, Wahid Uz Zaman*, Mahmut Kandemir*, and Vijaykrishnan Narayanan*†
*Department of Computer Science and Engineering, The Pennsylvania State University, USA
†School of Electrical and Computer Engineering, Georgia Institute of Technology, USA
{pmp5425, wzz5219, mtk2}@psu.edu, vnarayanan67@gatech.edu

***Abstract*—Mixture-of-Experts (MoE) language models reduce per-token computation by routing each token to a small subset of expert FFNs. However, sparse activation does not reduce the total expert-parameter footprint since the selected expert weights must still be accessed during inference. On edge-class systems with constrained host memory, autoregressive decoding repeatedly fetches these weights through SSD-backed offload, shifting the bottleneck from arithmetic throughput to data movement across the storage–memory hierarchy. This paper presents NearSEE, a computational-storage-based *Near*-*S*torage *E*xpert *E*xecution model for memory-constrained MoE inference. We implement NearSEE on Samsung SmartSSD devices for FP32 Mixtral-8x7B-Instruct and evaluate it under 32 GB and 64 GB host-memory budgets. Compared with the CPU/disk offload baseline, NearSEE-1CSD improves decode throughput by up to 2.14× under 32 GB and 1.84× under 64 GB, while NearSEE-2CSD further improves decode throughput by up to 3.78× and 3.09×, respectively. Under the 64 GB configuration, NearSEE reduces measured host-and-CSD energy by up to 52% relative to the CPU/disk offload baseline. These results show that near-storage expert execution directly targets the decode-stage offload bottleneck in memory-constrained MoE inference.**

## I. Introduction

Mixture-of-Experts (MoE) large language models improve scaling efficiency by activating only a small subset of expert feed-forward networks (FFNs) per token instead of the full path at each layer [1], [2]. While this sparsity reduces active computation, it does not reduce the total expert-parameter footprint, since the full expert pool must remain available during inference [3]. Because routing varies across tokens and layers, the selected experts change dynamically during decoding, making performance depend not only on arithmetic throughput but also on efficient access to selected expert weights. This is especially relevant for widely used models like Mixtral-8x7B [4], which offer strong quality and high effective capacity.

This challenge becomes particularly acute in emerging *edge and resource-constrained* environments, where inference must be performed close to data sources due to latency, privacy, and bandwidth constraints. These deployments differ from datacenter serving in two key ways. First, they typically operate at small batch sizes because local applications have limited request multiplexing and cannot accumulate large batches without increasing response latency. Small-batch inference provides limited weight reuse, making data movement a dominant cost. Second, edge-class systems provide limited memory capacity and bandwidth, so a single CPU, GPU, or host-memory hierarchy often cannot hold the full expert-parameter pool of a full-precision MoE model. Unlike datacenter systems, such deployments provide limited GPU memory ($\leq$ 24 GB VRAM) and modest host memory ($\leq$ 64 GB DRAM), making *full-precision* MoE inference difficult to sustain. Prior approaches [3], including those targeting deployments on low-VRAM GPUs, often rely on large host memory (e.g., $\geq$ 192 GB) to stage model weights—an assumption that does not hold in these settings. As a result, such systems rely on SSD-backed offloading and demand paging to enable execution, introducing significant data movement overheads across the storage–memory hierarchy. When the required model state exceeds available host memory, selected expert weights must repeatedly move from storage into the execution path, increasing data movement and reducing inference efficiency. KV cache growth can further increase memory pressure as sequence length grows. This memory pressure not only degrades efficiency but also limits the effectiveness of low-end GPUs; on systems with 32–64 GB host memory and low-VRAM GPUs, GPU execution may fail to deliver expected speedups and can even underperform CPU-only inference at small batch sizes due to excessive transfers across storage, memory, and GPU. These observations highlight a key insight: memory pressure and data movement can become primary performance bottlenecks for MoE inference in constrained environments.

A natural question is whether quantization alone can address this problem. While quantization reduces model footprint and weight traffic, storage-backed expert access is still required when the model exceeds available host-memory capacity. We therefore focus on FP32 inference to study this memory-constrained regime. Another direction is optimizing the KV cache, but this has already been widely studied due to its dynamic, input-dependent nature [5]–[7]. In contrast, model weights are static, reused across queries, and remain a major source of data movement overhead in memory-constrained MoE inference, motivating a focus on restructuring how expert weights are accessed and executed rather than further optimizing dynamic runtime state.

Recent advances in computational storage, such as SmartSSD [8], reflect a broader shift toward moving computation closer to where data resides. By combining storage with programmable logic such as FPGAs, these systems enable near-storage processing and reduce data transfers between

drives and host memory. Prior work [6], [7], [9], [10] has demonstrated the effectiveness of near-storage processing in data analytics and machine learning workloads, including emerging applications in LLM training and inference, where data movement and memory pressure are major bottlenecks. Together, these trends indicate that near-storage computation is becoming a practical and increasingly deployable element of modern systems.

Motivated by these observations, we propose NearSEE, a near-storage expert-execution system for memory-constrained MoE inference. NearSEE offloads routed expert FFN execution to FPGA-equipped computational storage devices (CSDs), while the host retains the token-dependent transformer path. Instead of paging selected expert weights into host DRAM, NearSEE stages them in device-local CSD memory and processes them near storage. This avoids routing large volumes of data through host memory, reducing DRAM pressure, KV cache interference, and redundant data movement, while mitigating page faults. Since the primary bottleneck is weight movement rather than compute, relocating execution closer to storage provides a natural and efficient solution. While GPUs excel at compute-intensive workloads, our results highlight a complementary regime where memory pressure and data movement dominate on capacity-constrained edge systems. NearSEE targets memory-constrained MoE inference in this regime by executing routed experts near storage and reducing repeated expert-weight transfers through host DRAM. At larger batch sizes, a single CSD can become saturated by the increased routed expert workload, motivating multi-CSD execution for higher-throughput MoE inference.

The main contributions of this paper are as follows:

1) We systematically characterize MoE inference under constrained host memory (32 to 64 GB) and low GPU memory (11 to 24 GB). Our measurements show that storage-backed weight movement can limit the benefit of GPU acceleration when model capacity exceeds available memory.
2) We propose NearSEE, a near-storage expert FFN execution mechanism that avoids staging selected expert weights in host memory, reducing host I/O and DRAM traffic while improving throughput in memory-constrained regimes.
3) We show that larger batch sizes increase routed expert work and can exceed the throughput available from a single CSD, and demonstrate that scaling expert execution across multiple CSDs improves throughput.

We evaluate NearSEE across multiple batch sizes and host memory configurations, demonstrating that near-storage expert execution reduces host-side weight movement and improves inference efficiency under constrained resources.

## II. Background

### A. Mixture-of-Experts

Transformer language models are built from a stack of layers, where each layer contains an attention block followed by a feed-forward network (FFN). In a dense model, every token passes through the same FFN in every layer. Mixture-of-Experts (MoE) models replace this single FFN with a set of expert FFNs and a router, as shown in Figure 4(a). For each token, the router selects a small number of experts, the selected experts process the token representation, and their outputs are combined using the router scores. This allows MoE models to increase total parameter capacity without executing all parameters for every token. However, the full expert pool must still be stored and made available because different tokens, layers, and batch elements can select different experts. Thus, MoE reduces active computation per token, but shifts inference toward a data-movement bottleneck dominated by fetching and staging sparsely activated expert weights.

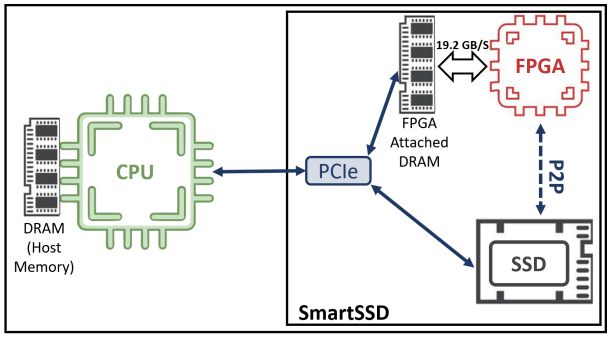



Fig. 1: Samsung SmartSSD architecture with an NVMe SSD, FPGA, FPGA-attached DRAM, and the device-local data path enabled by the onboard PCIe switch.

### B. LLM Inference

LLMs generate output text one token at a time. Given an input prompt, inference first runs a *prefill* stage, where the model processes the prompt tokens in parallel and builds the key-value (KV) cache used by attention. After prefill, the model enters the *autoregressive decode* stage, where it generates one new token per step. During each decode step, the model reads the KV cache, executes the transformer layers, updates the KV cache, and produces the next token. Since each generated token depends on the previous one, decode has limited parallelism across time, and any per-token cost is repeated throughout generation. In MoE models, each decode step also invokes the routed expert FFNs. The KV cache is dynamic and grows with sequence length and batch size, whereas expert weights are static but accessed according to router decisions. This work targets repeated expert-weight movement during MoE decode.

### C. SmartSSD Computational Storage

Figure 1 shows the SmartSSD architecture. SmartSSD is a 3.84-TB NVMe SSD integrating an AMD Kintex™ UltraScale+ KU15P FPGA with FPGA-attached DRAM. The host connects through a PCIe Gen3x4 interface (approximately 4 GB/s), while SSD media bandwidth is roughly 3 GB/s. SmartSSD supports peer-to-peer (P2P) transfers between the SSD and FPGA-local DRAM via an onboard PCIe switch [11], enabling SSD-resident data to be staged directly for FPGA execution without traversing host DRAM. The FPGA accesses its local DRAM at up to 19.2 GB/s [12], a bandwidth internal

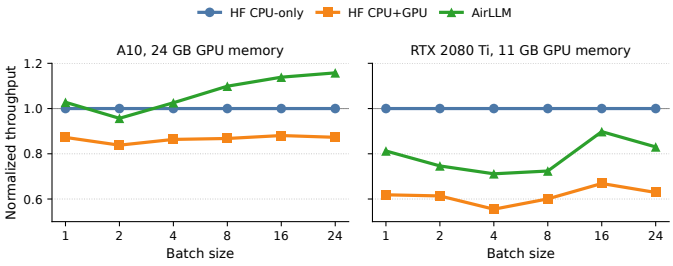



Fig. 2: Memory-constrained Mixtral decode throughput on two GPU systems, normalized to HF CPU-only execution.

TABLE I: GPU-side power and utilization during GPU-based offloading.

| GPU mem. | Method | $P_{\text{avg}}$ (W) | $U_{\text{avg}}$ (%) | $U_{\text{peak}}$ (%) | $E_{\text{GPU}}$ (kJ/run) |
|---|---|---|---|---|---|
| A10 24GB | HF CPU+GPU | 91.1 | 7.9 | 67 | 1302.2 |
| A10 24GB | AirLLM | 93.2 | 10.4 | 78 | 1072.1 |
| 2080Ti 11GB | HF CPU+GPU | 66.7 | 19.3 | 96 | 1104.6 |
| 2080Ti 11GB | AirLLM | 73.7 | 16.8 | 49 | 941.3 |

to the SmartSSD and distinct from the host–device PCIe link. Because this local bandwidth exceeds both the host interface and SSD bandwidths, P2P primarily improves data locality and reduces host-memory traffic rather than increasing host-link bandwidth.

## III. Motivation

On memory-constrained systems, the full MoE model cannot reside entirely in GPU memory or host DRAM, so inference commonly relies on CPU/disk offloading, CPU/GPU/disk offloading, or layer-streaming frameworks such as AirLLM [13]. We evaluate Mixtral-8x7B-Instruct-v0.1 using a 17-token input prompt and 128 generated tokens per sequence. We compare three commodity offloading baselines: Hugging Face Accelerate CPU/disk offload, Hugging Face Accelerate CPU/GPU/disk offload [14], and AirLLM [13]. We evaluate these baselines on two constrained-GPU systems: an NVIDIA A10 with 24 GB GPU memory and an NVIDIA RTX 2080 Ti with 11 GB GPU memory.

Figure 2 reports generated-token throughput normalized to Accelerate-CPU/disk on the same platform and batch size. Values above 1.0 indicate that a GPU-based path overcomes its additional staging and transfer overhead. On the A10 system, AirLLM exceeds Accelerate-CPU/disk only at larger batch sizes, reaching up to 1.16× normalized throughput, while Accelerate-CPU/GPU/disk remains below the CPU-only baseline at 0.84–0.88×. On the RTX 2080 Ti system, both GPU-based paths underperform Accelerate-CPU/disk: Accelerate-CPU/GPU/disk reaches only 0.55–0.67×, and AirLLM reaches 0.71–0.90×. These results show that limited GPU memory alone does not make MoE decode faster. When expert weights are still streamed through the host offload path, GPU compute is underfed, and batching amortizes but does not remove the underlying data-movement bottleneck.

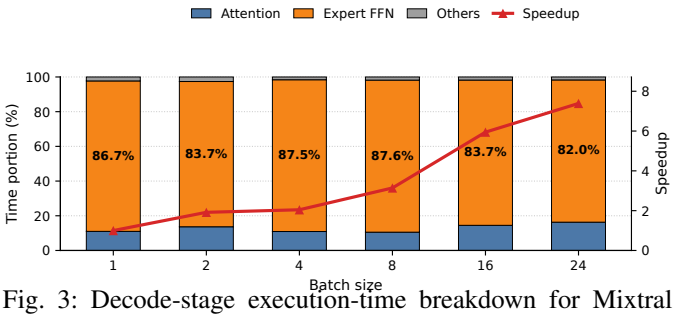



Fig. 3: Decode-stage execution-time breakdown for Mixtral inference across batch sizes.

Table I shows the same limitation from the GPU side. Values are summarized over batch sizes 1, 2, 4, 8, 16, and 24. $P_{\text{avg}}$, $U_{\text{avg}}$, and $E_{\text{GPU}}$ are averaged over the six batch runs, while $U_{\text{peak}}$ is the maximum observed GPU utilization across the sweep. GPU-based offloading consumes active GPU power, but average utilization remains low, ranging from 7.9% to 19.3%, indicating that the GPU is often waiting on the offload path rather than being fully utilized for sustained computation.

To determine the appropriate offload granularity, we further profile Mixtral decode using Hugging Face Accelerate CPU/disk offload and decompose execution time into attention, expert FFN execution, and remaining operations. Figure 3 shows that routed expert FFNs dominate decode across all evaluated batch sizes. Expert FFN execution accounts for 82.0%–87.6% of decode time, whereas attention contributes only 10.5%–16.3%. The remaining operations, including routing and runtime overheads, account for a small fraction of execution time. Although batching improves decode throughput, as shown by the speedup line, the expert FFN remains the largest component. Thus, increasing batch size alone does not eliminate the bottleneck; reducing the cost of routed expert FFN execution is necessary for improving memory-constrained MoE decode. Together, Figure 2, Table I, and Figure 3 identify the target for near-storage execution. Commodity offloading repeatedly moves expert weights through the host storage hierarchy, GPU-based offloading leaves accelerator compute underutilized, and routed expert FFNs account for most of the decode-stage execution time. These observations motivate moving routed expert FFN execution closer to where expert weights reside, so that memory-constrained MoE inference can reduce host-side weight movement while accelerating the dominant decode component.

## IV. NearSEE Design

### A. Design Overview

Figure 4(b) shows the NearSEE execution model. The host retains the token-dependent transformer path, including embedding lookup, attention, KV-cache management, routing, normalization, residual connections, output projection, and token selection. SmartSSD devices execute only the routed expert feed-forward networks (FFNs), which dominate the expert-parameter footprint. For each token activation $x_i$, the

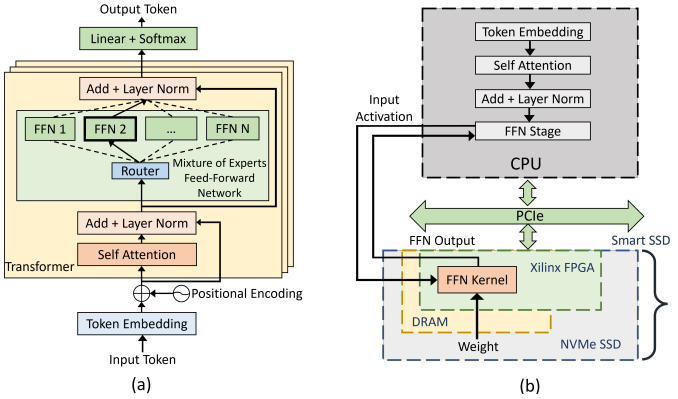



Fig. 4: Overview of MoE inference and NearSEE. (a) Architecture of a Mixture-of-Experts layer in large language models. (b) NearSEE near-storage execution model, where the host retains the token-dependent transformer path and SmartSSD devices execute routed expert FFNs.

MoE router selects a top-$k$ expert set $\mathcal{T}_i$ and produces normalized routing weights $\alpha_{i,e}$. The MoE output is

$$y_i = \sum_{e \in \mathcal{T}_i} \alpha_{i,e} E_e(x_i), \tag{1}$$

where $E_e(\cdot)$ is the FFN of expert $e$. For Mixtral with $k = 2$, this partitioning preserves the original MoE computation: the host computes $\mathcal{T}_i$ and $\alpha_{i,e}$, the SmartSSD computes the selected expert outputs $E_e(x_i)$, and the host applies the routing weights and accumulates the final MoE output. Each expert FFN computes

$$E_e(x) = (\mathrm{SiLU}(xW_1) \odot xW_3)\, W_2, \tag{2}$$

where $W_1$, $W_3$, and $W_2$ are the expert-specific gate, up, and down projection matrices. Expert FFNs dominate the parameter footprint of Mixtral-style MoE models, and each expert call is specified by the selected layer, expert identifier, and routed activation batch. The remaining transformer path stays on the host because it manages token-dependent state and control flow during autoregressive decoding. Therefore, NearSEE offloads only routed expert FFNs rather than moving the full transformer stack into the CSD.

### B. Hybrid CPU-Prefill and CSD-Decode Pipeline

NearSEE uses a hybrid CPU-prefill and CSD-decode pipeline. During prefill, the full Mixtral model runs through the CPU/disk offload path, processes all prompt tokens, constructs the KV cache, and produces the logits for the first generated token. We keep this phase on the CPU because prefill exposes more token-level parallelism and, in our experiments, is more efficiently handled by the host runtime than by the SmartSSD expert path. After prefill, the KV cache is retained on the CPU, and the full CPU model is released. The decode model is then loaded with expert tensors removed from the host checkpoint; for Mixtral-8x7B, this removes the three FFN matrices for every expert in every MoE layer. During autoregressive decode, SmartSSD calls supply the missing expert computations whenever an MoE layer invokes a selected expert FFN. This division targets the repeated and memory-bound portion of generation: prefill executes once per prompt, whereas decode invokes routed expert FFNs at every generated token and every MoE layer.

### C. Host–CSD Expert Interface

The NearSEE runtime intercepts execution at the MoE boundary. In the single-CSD path, each Mixtral expert module is replaced by a drop-in FPGA-backed expert module. The replacement receives selected token activations with shape $[N, d_{\mathrm{model}}]$, converts them to contiguous FP32 tensors in CPU memory, calls the CSD runtime, and returns an output tensor with the same shape. In the multi-CSD path, the runtime replaces the MoE forward function. The host router computes the top-$k$ experts and routing weights, after which the runtime groups token activations by selected expert and creates jobs of the form $(\ell, e, X)$, where $\ell$ is the layer index, $e$ is the expert index, and $X$ is the activation batch assigned to that expert. After the CSD workers return the expert outputs, the host applies the routing weights and accumulates the final MoE result. The software interface to the CSD backend is

$$\texttt{run_expert}(\ell, e, X, Y, N), \tag{3}$$

where $Y$ is the output buffer and $N$ is the number of routed token activations. This interface exposes only the selected expert invocation to the CSD backend. Routing, weighting, residual computation, attention, and KV-cache updates remain in the host model.

### D. SmartSSD Runtime and Expert Weight Streaming

Each SmartSSD is managed by an independent XRT device handle. The runtime loads the FPGA binary, allocates device buffers, stages expert weights, launches kernels, and synchronizes outputs with the host. Expert weights are stored separately from the host decode checkpoint in packed FP32 files. The layout is indexed by layer and expert identifier and stores the three matrices required by the fused FFN. Each CSD stores the full expert-weight set on its local SSD. This replication lets the scheduler place tasks by load rather than expert ownership and avoids inter-CSD weight transfers. During decode, the host model does not materialize these expert tensors. Instead, the CSD runtime uses the selected layer–expert pair to locate the corresponding packed weights and stage them into SmartSSD device buffers before kernel execution. All SmartSSD experiments use peer-to-peer-enabled communication for expert-weight movement between the SSD-resident packed weight file and the FPGA execution path. This keeps expert weights outside the host-loaded PyTorch decode model and lets the CSD runtime stage the selected expert weights directly for FPGA execution. In the current token-by-token decode path, expert calls frequently move across different layers and experts, so the system is best characterized as SSD-resident expert streaming with device-local staging. Since the FPGA-attached DRAM cannot hold

**Algorithm 1** Fused FP32 Expert FFN Kernel

**Input:** activation tile $X$; weights $W_1$, $W_3$, $W_2^T$
**Output:** expert output tile $Y$
Load $X$ into local buffer $X_b$; initialize $Y_b \leftarrow 0$
**for** each intermediate index $h = 0$ to $d_{\text{ff}} - 1$ **do**
  Initialize $g[0:T-1] \leftarrow 0$ and $u[0:T-1] \leftarrow 0$
  **for** each vector block $v = 0$ to $d_{\text{model}}/V - 1$ **do**
    Load $p_1 \leftarrow W_1[h, v]$ and $p_3 \leftarrow W_3[h, v]$
    **for** each token $i = 0$ to $T - 1$ **do**
      $g[i] \leftarrow g[i] + \textsc{Dot}(X_b[i, v], p_1)$
      $u[i] \leftarrow u[i] + \textsc{Dot}(X_b[i, v], p_3)$
    **end for**
  **end for**
  **for** each token $i = 0$ to $T - 1$ **do**
    $a[i] \leftarrow \text{SiLU}(g[i]) \cdot u[i]$
  **end for**
  **for** each vector block $v = 0$ to $d_{\text{model}}/V - 1$ **do**
    Load $p_2 \leftarrow W_2^T[h, v]$
    **for** each token $i = 0$ to $T - 1$ **do**
      $Y_b[i, v] \leftarrow Y_b[i, v] + a[i] \cdot p_2$
    **end for**
  **end for**
**end for**
Store $Y_b$ to output tile $Y$

the full Mixtral expert pool, the SmartSSD acts as a near-storage streaming executor rather than a fully weight-resident LLM accelerator.

### *E. FPGA Expert Engine*

The SmartSSD executes the routed expert FFN computation from Eq. (2) using a Vitis HLS kernel specialized for Mixtral. The kernel is implemented in FP32 and fuses the gate projection, up projection, SiLU activation, elementwise product, and down projection into one FPGA-side execution path. Thus, the host exchanges only routed input activations and final expert outputs with the CSD runtime. The top-level kernel exposes five memory arguments: input activation $X$, output activation $Y$, and three expert weight buffers for $W_1$, $W_3$, and $W_2^T$. These buffers are mapped to separate AXI master interfaces. $W_1$ and $W_3$ are stored in row-major order as $[d_{\text{ff}}, d_{\text{model}}]$. The down-projection matrix is stored as $W_2^T$ using the same traversal order. This layout lets the kernel stream the rows of $W_1$, $W_3$, and $W_2^T$ for the same intermediate index before advancing to the next one. Algorithm 1 shows the FPGA schedule for one routed activation tile. The kernel loads the tile into on-chip storage, scans the FFN intermediate dimension, and immediately consumes each intermediate activation in the down projection. As a result, the $d_{\text{ff}}$-dimensional intermediate vector is never written to external memory.

The implementation uses $d_{\text{model}} = 4096$ and $d_{\text{ff}} = 14336$. The token tile size is $T = 4$, and the vector width is $V = 16$, corresponding to 16 FP32 values per 512-bit memory access. Since $d_{\text{model}}/V = 256$, each expert-weight row is streamed as 256 vector words. Each expert call streams the three FP32 expert matrices:

$$3 \times d_{\text{ff}} \times d_{\text{model}} \times 4 = 704.6 \text{ MB} \approx 672 \text{ MiB}. \quad (4)$$

The routed input and output tiles are only $T \times d_{\text{model}} \times 4$ bytes each. Therefore, the FPGA kernel is dominated by expert-weight streaming rather than activation movement. Each SmartSSD bitstream instantiates two identical FFN kernel instances. The runtime partitions routed activations into four-token tiles, stripes independent tiles across the two kernels, and uses ping-pong input/output buffers across successive expert calls. This keeps the host interface simple: the host provides the selected layer, expert identifier, input tile, and output buffer, while the FPGA executes the selected expert FFN. Table II summarizes the kernel configuration and per-instance resource utilization. The resource numbers are reported for one FFN kernel instance; the bitstream contains two such instances. The tile size $T$ trades weight reuse against on-chip activation storage, while kernel replication trades FPGA resources for tile-level parallelism. The FPGA expert engine can be reprogrammed by rebuilding the HLS kernel and loading the resulting hardware binary through XRT. Porting NearSEE requires adapting the kernel, buffers, and runtime to a programmable CSD with a local weight-transfer path that bypasses host DRAM. PCIe P2P is not SmartSSD-specific but requires platform support. For systems with more than two CSDs, partitioning or selective replication could reduce storage duplication, with task assignment restricted to devices holding the selected expert.

TABLE II: SmartSSD FP32 expert FFN kernel configuration and per-instance resource utilization.

| Metric | Value | Unit |
|---|---|---|
| *Kernel configuration* | | |
| Kernel instances per SmartSSD | 2 | – |
| Model dimension, $d_{\text{model}}$ | 4096 | elements |
| Expert dimension, $d_{\text{ff}}$ | 14336 | elements |
| Token tile size, $T$ | 4 | tokens |
| Vector width, $V$ | 16 | FP32 values |
| AXI vector width | 512 | bits |
| Clock frequency | 260.7 | MHz |
| Activation precision | FP32 | – |
| Weight precision | FP32 | – |
| *Per-kernel resource utilization* | | |
| LUT | 50,034 | 9.57% |
| Register | 80,384 | 11.39% |
| BRAM | 132 | 13.41% |
| URAM | 0 | N/A |
| DSP | 348 | 17.68% |

### *F. Scaling Across Multiple CSDs*

The multi-CSD NearSEE implementation uses a worker-pool runtime. Each worker owns one SmartSSD device handle, one FPGA binary instance, and one packed expert-weight file. At each MoE layer, the host constructs per-worker job queues and dispatches routed expert tasks in parallel.

The multi-CSD path uses routed-task-parallel scheduling. The runtime first groups routed token activations by selected expert, then splits each expert group into bounded-size tasks and assigns those tasks to the least-loaded CSD worker. Here, the least-loaded CSD is the device with the fewest routed rows assigned within the current MoE layer. Figure 5 and Algorithm 2 illustrate this policy. By scheduling at task granularity rather than assigning whole experts to fixed devices,

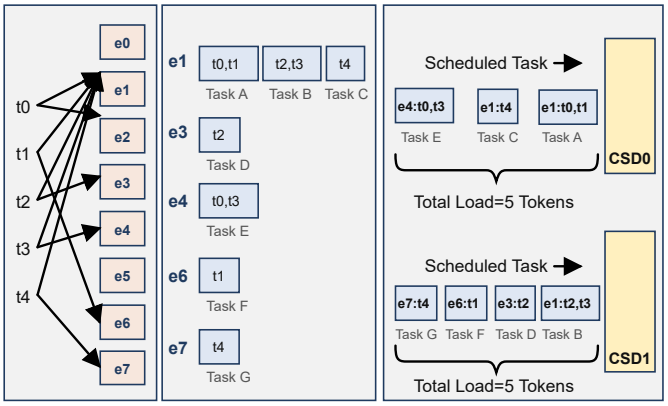



Fig. 5: Multi-CSD routed-task-parallel scheduling.

**Algorithm 2** Routed-task-parallel scheduling for multi-CSD expert execution

**Input:** hidden states $X$; selected experts $E$; routing weights $R$; max task rows $\tau$; CSD workers $\mathcal{D}$
**Output:** MoE output $Y$
Initialize $Y \leftarrow 0$, job queues $Q_d \leftarrow \emptyset$, and worker loads $\mathrm{load}[d] \leftarrow 0$
**for** each expert $e$ with routed rows in $E$ **do**
  Gather routed activations $X_e$, token indices $I_e$, and top-$k$ slots $S_e$
  **for** each chunk $(X_{e,c}, I_{e,c}, S_{e,c})$ with at most $\tau$ rows **do**
    $d^\star \leftarrow \arg\min_{d \in \mathcal{D}} \mathrm{load}[d]$
    Enqueue $(e, X_{e,c}, I_{e,c}, S_{e,c})$ on $Q_{d^\star}$
    $\mathrm{load}[d^\star] \leftarrow \mathrm{load}[d^\star] + |X_{e,c}|$
  **end for**
**end for**
**for** each CSD worker $d \in \mathcal{D}$ in parallel **do**
  **for** each job $(e, X_{e,c}, I_{e,c}, S_{e,c})$ in $Q_d$ **do**
    Compute $Z_{e,c} \leftarrow \mathrm{FFN}_e(X_{e,c})$ on CSD $d$
    Return $(Z_{e,c}, I_{e,c}, S_{e,c})$ to the host
  **end for**
**end for**
**for** each returned $(Z_{e,c}, I_{e,c}, S_{e,c})$ **do**
  Accumulate $R[I_{e,c}, S_{e,c}] \odot Z_{e,c}$ into rows $Y[I_{e,c}]$
**end for**
**return** $Y$

the runtime exposes finer-grained parallelism and reduces load imbalance under skewed expert routing. After execution, the host gathers the returned expert outputs and performs the same routing-weighted accumulation as in the single-CSD path. This preserves the host-side MoE interface while scaling routed expert FFN execution across multiple CSDs.

## V. Experimental Methodology

### A. Experimental Platform and Workloads

Table III summarizes the SmartSSD evaluation platform, where the host executes the transformer runtime and orchestrates SmartSSD execution through XRT. All systems use the same model checkpoint and tokenizer. Experiments use FP32 Mixtral-8x7B-Instruct-v0.1 [4] with 128 generated tokens, greedy decoding, and batch sizes 1, 2, 4, 8, 16, and 24. EOS early stopping is disabled so that all systems execute the same number of decode steps. The 32-GiB and 64-GiB memory-budget experiments use a 17-token prompt. To evaluate routing diversity, we additionally perform a 32-GiB sweep using up to 24 distinct AlpacaEval prompts [15] ranging from 18 to 60 tokens. For each batch size, all compared systems use the same input prompts.

TABLE III: NearSEE evaluation platform.

| Component | Configuration |
|---|---|
| Host processor | Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz |
| Host memory budget | 32 GB and 64 GB configurations |
| CPU threads | 12 |
| CSD device | Samsung SmartSSD with AMD Kintex UltraScale+ KU15P FPGA |
| CSD count | 1-CSD and 2-CSD configurations |
| FPGA Software Stack | XRT 2.12.427 (runtime), Vitis 2021.2 (Kernel Development) |
| Software stack | PyTorch 2.0.1+cu118, Transformers 4.46.3, Accelerate 1.0.1 |
| Linux kernel | 5.4.0-234-generic |

TABLE IV: Execution model of host-centric offload baselines and NearSEE.

| Framework / design | Weight location | Expert FFN execution |
|---|---|---|
| HF CPU/disk | CPU DRAM + disk offload | CPU |
| HF GPU/CPU/disk | GPU, CPU DRAM, disk | GPU if mapped; CPU otherwise |
| AirLLM [13] | Layer-wise disk shards | GPU after layer load |
| llama.cpp | GGUF/mmap model layout | CPU |
| **NearSEE** | SSD-resident expert weights | **SmartSSD FPGA** |

### B. Compared Systems

We compare NearSEE against host-side offload baselines using the same workload configuration. Table IV summarizes where model weights reside and where routed expert FFNs execute in each path. The primary architectural comparison is between Hugging Face Accelerate CPU/disk offload and NearSEE, which use the same host-side framework while differing in the execution path for routed expert FFNs. We additionally report llama.cpp as an optimized CPU-only baseline, while Hugging Face CPU+GPU/disk and AirLLM are used only for the GPU-based motivation study. The CPU/disk baseline uses the full model with Hugging Face Accelerate [14] offload. NearSEE uses CPU prefill and redirects decode-stage routed expert FFNs to one SmartSSD in NearSEE-1CSD and to two SmartSSDs in NearSEE-2CSD. This keeps the comparison focused on the decode-stage expert execution path. We enforce the 32 GiB and 64 GiB host-memory budgets using cgroup v2 `memory.max`. For Hugging Face model placement, the CPU memory limits are set to 28 GiB and 56 GiB, respectively, reserving nominal headroom for the KV cache, activations, worker processes, and runtime state. Parameters exceeding these placement limits are explicitly offloaded to storage by the runtime.

### C. Metrics

We report latency, throughput, host-side activity, and energy metrics. Prefill time measures prompt processing, KV-cache construction, and computation of the logits used to start generation. Decode time measures autoregressive generation after prefill, where each generated token executes the transformer layers, reads and updates the KV cache, and invokes routed expert FFNs. Generation time is measured as the sum of prefill

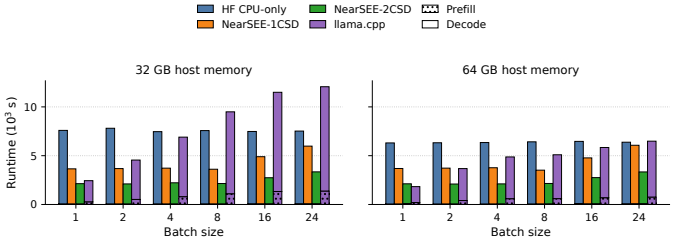



Fig. 6: Runtime breakdown under 32 GB and 64 GB host-memory budgets. Stacked bars separate prefill and decode time.

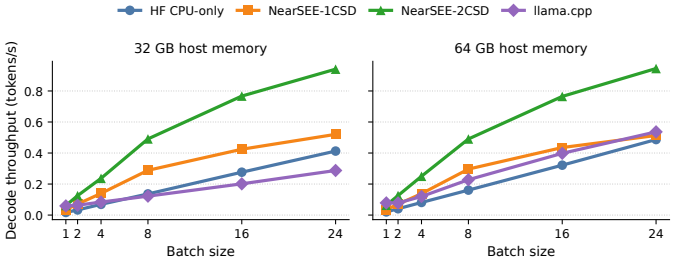



Fig. 7: Decode throughput under 32 GB and 64 GB host-memory budgets.

time and decode time. In the following equations, generated tokens denote the total number of output tokens generated across the batch. Decode throughput isolates the autoregressive phase and is computed as

$$\text{DecodeTok/s} = \frac{\text{generated tokens}}{\text{decode time}}. \tag{5}$$

End-to-end generation throughput includes both prefill and decode and is computed as

$$\text{GenTok/s} = \frac{\text{generated tokens}}{\text{prefill time} + \text{decode time}}. \tag{6}$$

For host-side behavior, we record memory usage, page faults, swap activity, and storage read traffic. For NearSEE execution, we additionally record FPGA expert calls, processed token tiles, processed tokens, kernel time, I/O time, weight-staging time, read time, synchronization time, and device staging-cache counters. For energy measurements, host energy is measured using RAPL counters and includes CPU package energy and DRAM energy. Thus, the CPU-only and `llama.cpp` energy values reported in this paper correspond to measured host CPU-package-plus-DRAM energy. For NearSEE, we additionally include SmartSSD device energy when SmartSSD power logs are available. SmartSSD power is sampled through the XRT utility interface during the run, and average SmartSSD power is multiplied by runtime to estimate SmartSSD energy. Energy per generated token is computed by dividing the corresponding measured energy by the total number of generated tokens. Since SSD/NVMe device energy is not measured for the CPU-only and `llama.cpp` baselines, we report measured host energy for host-only baselines and measured host-and-CSD energy for NearSEE, rather than claiming full-system energy.

### D. Run Control and Fairness

All compared systems use the same input workload, tokenizer, precision, batch size, decoding policy, and fixed generated-token count. For NearSEE, CPU prefill is measured directly, and the resulting KV cache is used for CSD decode. For the diverse-prompt experiments, swap is disabled and model-weight files are evicted from the OS page cache before each measured run to ensure a cold-start condition.

## VI. Evaluation

This section evaluates NearSEE on memory-constrained MoE inference and quantifies its impact on decode latency, throughput, device-level read traffic, and measured energy. We compare HF Accelerate CPU-only inference with disk offload, NearSEE-1CSD, NearSEE-2CSD, and a llama.cpp CPU-only baseline. All reported NearSEE-2CSD results use routed-task-parallel scheduling.

### A. Generation Runtime

Figure 6 shows that autoregressive decode dominates generation runtime across all configurations, confirming that decode-stage expert execution is the primary optimization target. Under the 32 GB memory budget, HF CPU-only decode takes 7.40–7.75$\times 10^3$ s across batch sizes. NearSEE-1CSD reduces decode time to 3.55–5.90$\times 10^3$ s, reaching up to 2.14$\times$ speedup. NearSEE-2CSD further reduces decode time to 2.05–3.27$\times 10^3$ s and reaches up to 3.78$\times$ speedup over HF CPU-only. The 64 GB configuration improves the HF CPU-only baseline because more parameters can remain in host memory, reducing the relative gain of NearSEE-1CSD. Even in this less memory-constrained setting, NearSEE-1CSD reaches up to 1.84$\times$ speedup. NearSEE-2CSD reaches up to 3.09$\times$ speedup over HF CPU-only. The llama.cpp baseline shows that optimized host-side execution can be competitive, especially under 64 GB, while NearSEE-2CSD achieves the lowest decode time in this evaluation.

### B. Decode Throughput

Figure 7 reports generated-token throughput during decode. Under 32 GB, NearSEE-1CSD improves throughput by up to 2.14$\times$ over HF CPU-only and remains close to 2$\times$ through batch size 8. At larger batches, the single-CSD gain narrows as routed expert work increases, while NearSEE-2CSD sustains the advantage, reaching 0.941 tokens/s at batch size 24 compared with 0.413 tokens/s for HF CPU-only and 0.520 tokens/s for NearSEE-1CSD. Under 64 GB, NearSEE-1CSD nearly converges with HF CPU-only at batch size 24, reaching 0.513 tokens/s compared with 0.487 tokens/s. In contrast, NearSEE-2CSD reaches 0.945 tokens/s, corresponding to 1.94$\times$ speedup over HF CPU-only and 1.84$\times$ over NearSEE-1CSD. At batch size 24, the llama.cpp baseline reaches 0.287 tokens/s under 32 GB and 0.536 tokens/s under 64 GB, remaining below NearSEE-2CSD in both configurations.

TABLE V: Two-SSD no-FPGA ablation at batch size 8 under the 32 GiB memory budget.

| Configuration | Decode (s) | E2E (s) |
|---|---|---|
| llama.cpp, 1 SSD | 5467.20 | 6224.81 |
| llama.cpp, 2 SSDs | 5413.68 | 6162.46 |
| NearSEE-2CSD | 2085.89 | 2147.80 |

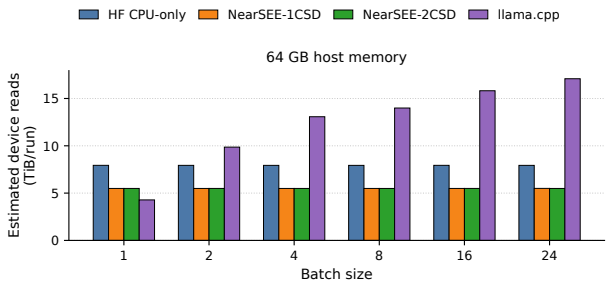



Fig. 8: Estimated device-level read volume under the 64 GB host-memory configuration.

TABLE VI: Diverse-prompt decode throughput under the 32 GiB host-memory configuration.

| Batch | HF tok/s | NearSEE-1CSD tok/s (×HF) | NearSEE-2CSD tok/s (×HF) |
|---|---|---|---|
| 1 | 0.0091 | 0.0375 (4.12×) | 0.0683 (7.49×) |
| 2 | 0.0167 | 0.0425 (2.55×) | 0.0652 (3.91×) |
| 4 | 0.0305 | 0.0571 (1.87×) | 0.0914 (2.99×) |
| 8 | 0.0580 | 0.0893 (1.54×) | 0.1482 (2.55×) |
| 16 | 0.1120 | 0.1453 (1.30×) | 0.2220 (1.98×) |
| 24 | 0.1692 | 0.2269 (1.34×) | 0.2911 (1.72×) |

TABLE VII: Routing characteristics of homogeneous- and diverse-prompt batches at batch size 24 under the 32 GiB host-memory budget.

| Metric | Homogeneous | Diverse |
|---|---|---|
| Active experts / layer-step | 2.00 | 7.95 |
| Normalized routing entropy | 0.333 | 0.958 |
| Mean rows / active expert group | 24.00 | 6.04 |

### C. Two-SSD No-FPGA Ablation

To isolate the benefit of the second storage device from near-storage execution, we compare llama.cpp with the FP32 model placed on one SmartSSD and striped across two SmartSSDs, with FPGA execution disabled in both cases. As Table V shows, moving from one to two SSDs improves decode and end-to-end performance by only 1.01×. In contrast, NearSEE-2CSD achieves 2.60× and 2.87× speedup in decode and end-to-end execution, respectively, relative to the two-SSD no-FPGA configuration. Thus, the observed multi-CSD gain cannot be attributed to additional storage bandwidth alone.

### D. Evaluation with Diverse Prompts

We further evaluate the HF CPU/disk baseline, NearSEE-1CSD, NearSEE-2CSD, and the optimized FP32 llama.cpp host baseline using a diverse-prompt workload comprising up to 24 distinct AlpacaEval prompts [15]. For each batch size, all systems use the same input prompts. As shown in Table VI, NearSEE-2CSD achieves 1.72×–7.49× the decode throughput of the HF CPU/disk baseline. Increasing from one to two CSDs provides a further 1.28×–1.82× throughput improvement, showing that the benefit of multi-CSD expert execution persists under diverse-prompt routing.

Compared with the optimized FP32 llama.cpp host baseline, NearSEE-2CSD achieves comparable decode throughput at B1 (0.0683 versus 0.0648 tokens/s) and remains close at B8 (0.1482 versus 0.1582 tokens/s). At B24, llama.cpp reaches 0.4052 tokens/s compared with 0.2911 tokens/s for NearSEE-2CSD. This differs from the homogeneous-prompt evaluation, where NearSEE-2CSD achieves 4.02× and 3.27× the decode throughput of llama.cpp at B8 and B24, respectively. We next examine the routing characteristics of the two workloads to explain this change in relative performance.

Table VII compares the routing characteristics of homogeneous- and diverse-prompt batches at B24 under the same 32 GiB host-memory budget. We use *routing concentration* [16] to denote the extent to which routed assignments are concentrated among a subset of experts at a given layer and decode step. Normalized routing entropy is computed as $H(p)/\log E$, where $H(p) = -\sum_e p_e \log p_e$, $p_e$ is the fraction of routed assignments to expert $e$, and $E = 8$. Homogeneous-prompt batching activates 2.00 experts per layer and decode step with a normalized entropy of 0.333 and 24.0 routed rows per active expert group. In contrast, diverse-prompt batching activates 7.95 experts, increases routing entropy to 0.958, and reduces the mean expert-group size to 6.04 rows. Thus, prompt diversity increases the number of concurrently active experts by nearly 4× while reducing the work aggregated per active expert by approximately 4×, which lowers the opportunity to amortize expert-weight movement over routed computation.

### E. Device Read Traffic

Figure 8 reports estimated device-level read volume from `iostat` under the 64 GB configuration. HF CPU-only reads approximately 7.93 TiB per run, reflecting repeated movement of offloaded model data through the storage-backed execution path. NearSEE-1CSD and NearSEE-2CSD reduce the observed read volume to approximately 5.49 TiB per run by moving routed expert execution closer to SSD-resident expert weights and avoiding expert tensor materialization in the host-loaded decode model.

The llama.cpp baseline uses a memory-mapped GGUF model format that avoids the Hugging Face tensor offload path, giving lower device reads at batch size 1. However, its reads still grow from 4.28 TiB to 17.09 TiB by batch size 24, so we treat it as an optimized host-side reference rather than the primary architectural comparison for the NearSEE data path.

### F. Energy Breakdown

Figure 9 reports measured energy under 64 GB, normalized to HF CPU-only host energy. Host energy is the sum of CPU package and DRAM energy from RAPL counters; for

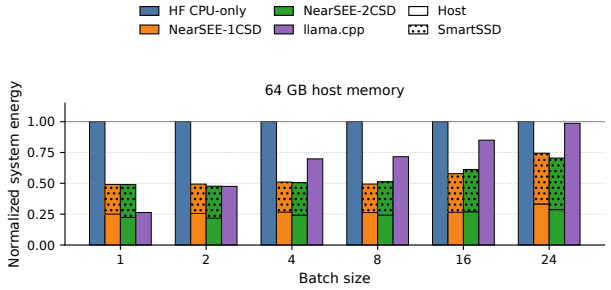



Fig. 9: Normalized measured energy under the 64 GB host-memory configuration. Stacked bars separate host energy and SmartSSD device energy where applicable.

NearSEE configurations, SmartSSD device energy is added using device power logs. We do not claim full-platform energy because SSD/NVMe energy for host-side disk offload is not measured. NearSEE-1CSD reduces measured host-and-CSD energy by 26–51% across batch sizes, while NearSEE-2CSD reduces it by 29–52%. Although two SmartSSDs draw more device power, their shorter runtime offsets the additional device energy; at batch size 24, NearSEE-2CSD improves the energy reduction from 26% to 29%. The llama.cpp baseline reduces measured host energy by 1–74%, following its host-side runtime behavior, but it does not move expert execution into near-storage hardware.

### G. Summary

Overall, NearSEE-1CSD improves decode throughput by up to 2.14× under 32 GB and 1.84× under 64 GB, while NearSEE-2CSD improves decode throughput by up to 3.78× and 3.09×, respectively. Under 64 GB, estimated device-level read volume falls from 7.93 TiB/run to 5.49 TiB/run, and measured host-and-CSD energy falls by up to 52%. These results show that NearSEE is most effective when SSD-backed expert-weight transfers dominate decode time, while additional CSD parallelism improves throughput scaling at larger batches. Under diverse-prompt batching, NearSEE-2CSD retains 1.72×–7.49× speedup over HF baseline and 1.28×–1.82× scaling over NearSEE-1CSD. Routing analysis shows that prompt diversity broadens expert activation and reduces per-expert work aggregation, identifying routing concentration as a key factor in near-storage MoE efficiency.

## VII. Related Work

Recent systems have studied ways to reduce the memory and I/O overhead of large-model inference under constrained hardware budgets. MoE-Lightning [3] targets high-throughput MoE inference on memory-constrained GPUs using CPU–GPU–I/O pipelining, paged expert weights, and a hierarchical roofline model for scheduling. It is closely related to our motivation because both works target MoE inference when the full model cannot reside in GPU memory. However, MoE-Lightning is designed for throughput-oriented batched serving and relies on the host-side CPU memory, GPU memory, and storage hierarchy to stage and schedule expert weights. In contrast, NearSEE targets lower-memory edge systems where decode is dominated by repeated expert-weight movement, and reduces this movement by executing routed expert FFNs near SSD-resident weights in computational storage. SLIM [17] also targets edge LLM inference and observes that limited memory capacity and bandwidth make FFN weight movement a major bottleneck. It proposes a heterogeneous DRAM–SSD accelerator that combines processing-in-memory (PIM) for memory-intensive operations with near-storage processing (NSP) for feed-forward network (FFN) computation. While SLIM is conceptually similar, it relies on a custom heterogeneous PIM/NSP architecture. In contrast, NearSEE implements and evaluates MoE expert offloading on a commercially available SmartSSD-based platform, leveraging FPGA-equipped computational storage devices and a practical host–CSD runtime to support full-precision expert execution. HILOS [7] also explores near-storage processing for LLM inference, offloading attention and KV-cache-related operations to computational storage devices in order to reduce interconnect traffic. NearSEE is complementary to HILOS, focusing instead on mitigating the repeated movement of expert weights and accelerating routed expert FFN execution during autoregressive decoding.

## VIII. Conclusion and Future Work

This paper presented NearSEE, a computational-storage-based near-storage execution model for full-precision MoE inference under constrained host memory. NearSEE keeps the token-dependent transformer path on the host while executing routed expert FFNs in computational storage close to SSD-resident expert weights. Our FP32 Mixtral-8x7B-Instruct prototype shows that NearSEE-1CSD improves decode throughput by up to 2.14× under a 32 GB host-memory budget and 1.84× under a 64 GB budget. NearSEE-2CSD further improves decode throughput by up to 3.78× and 3.09×, respectively, and reduces measured host-and-CSD energy by up to 52% under the 64 GB configuration. Our diverse-prompt analysis further identifies routing concentration as an important determinant of near-storage MoE efficiency, as broader expert activation reduces per-expert work aggregation and the opportunity to amortize expert-weight movement. Future work will explore lower-precision expert execution, additional MoE models, and scaling beyond two SmartSSDs. As computational-storage devices evolve with tighter SSD–compute integration and higher-bandwidth device-local paths, the locality advantage of near-storage execution is expected to remain relevant even as host PCIe bandwidth improves.

## Acknowledgment

This work was supported in part by PRISM, one of seven research centers under the JUMP 2.0 program, an initiative of the Semiconductor Research Corporation (SRC) sponsored by DARPA, as well as by the U.S. Department of Energy under Award DE-SC0023186 and the National Science Foundation under Award Nos. 2312886 and 2453653.

## REFERENCES


[1] W. Fedus, B. Zoph, and N. Shazeer, "Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[2] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat *et al.*, "GLaM: Efficient scaling of language models with mixture-of-experts," in *International Conference on Machine Learning*. PMLR, 2022, pp. 5547–5569.

[3] S. Cao, S. Liu, T. Griggs, P. Schafhalter, X. Liu, Y. Sheng, J. E. Gonzalez, M. Zaharia, and I. Stoica, "MoE-Lightning: High-throughput MoE inference on memory-constrained GPUs," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 715–730.

[4] Mistral AI, "Mixtral 8x7B," https://mistral.ai/news/mixtral-of-experts/.

[5] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[6] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "InstAttention: In-storage attention offloading for cost-effective long-context LLM inference," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1510–1525.

[7] H. Jang, J. Song, C. Shin, S. U. Noh, J. Jung, J. Park, and J. Lee, "A cost-effective near-storage processing solution for offline inference of long-context LLMs," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2026, pp. 5–24.

[8] AMD, "SmartSSD computational storage drive product brief," Product brief, 2023. [Online]. Available: https://www.xilinx.com/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf

[9] H. Jang, J. Song, J. Jung, J. Park, Y. Kim, and J. Lee, "Smart-Infinity: Fast large language model training using near-storage processing on a real system," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 345–360.

[10] W. U. Zaman, C. S. Mishra, S. AlSaleh, A. Aghayev, and M. T. Kandemir, "Cord: Parallelizing query processing across multiple computational storage devices," in *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025, pp. 1141–1153.

[11] AMD Xilinx, "PCIe Peer-to-Peer (P2P)," https://xilinx.github.io/XRT/, accessed: September 2025.

[12] AMD, "SmartSSD computational storage drive installation and user guide (UG1382)," 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1382-smartssd-csd

[13] G. Li, "AirLLM: Scaling large language models on low-end commodity computers," 2023. [Online]. Available: https://github.com/lyogavin/airllm/

[14] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable," https://github.com/huggingface/accelerate, 2022.

[15] X. Li, T. Zhang, Y. Dubois, R. Taori, I. Gulrajani, C. Guestrin, P. Liang, and T. B. Hashimoto, "AlpacaEval: An automatic evaluator of instruction-following models," 2023.

[16] R. Huang, Q. Wang, H. Huang, Y. Gao, D. Chen, S. Wang, and W. Wang, "RepetitionCurse: Measuring and understanding router imbalance in mixture-of-experts LLMs under DoS stress," 2026. [Online]. Available: https://arxiv.org/abs/2512.23995

[17] W. Xu, H. Choi, P.-K. Hsu, S. Yu, and T. Simunic, "SLIM: A heterogeneous accelerator for edge inference of sparse large language model via adaptive thresholding," *ACM Transactions on Embedded Computing Systems*, vol. 25, no. 3, pp. 1–28, 2026.