REVIVING TURING FOR DEEPSEEK-V4-FLASH:
HETEROGENEOUS W8A8 INFERENCE OF A 284B MOE ON FOUR RTX 2080 TI

A PREPRINT

Yufeng Lyu

May 2026

ABSTRACT

Frontier mixture-of-experts (MoE) language models such as DeepSeek-V4-Flash (284B total, 13B active parameters) are designed for accelerators with BF16, FP8, and FP4 tensor cores, NVLink, and modern kernel libraries [3, 29, 26]. Existing options for running such models without frontier hardware fall into two camps: ktransformers [13] offers heterogeneous CPU/GPU execution but its fast paths assume newer Intel CPUs with AMX-class matrix instructions, while llama.cpp [10] relies on aggressive low-bit quantization that can introduce additional model-quality risk. Neither targets the case of cheap, widely available legacy GPUs paired with older CPUs. This work starts from the opposite premise: **low-cost legacy hardware should be able to run today's strongest open MoE models without changing their routing semantics or resorting to sub-INT8 quantization**, so that executing frontier-scale open MoE models is not gated on access to frontier accelerators. We show this is achievable. On a four-GPU RTX 2080 Ti server assembled in 2023 for under CNY 20,000, whose 2018-vintage Turing GPUs lack BF16/FP8/FP4 tensor cores, have no NVLink, and connect only over PCIe Gen3 alongside an AVX2-only dual-socket CPU, an INT8 W8A8 runtime serves DeepSeek-V4-Flash end-to-end. The key insight is that the bottleneck on this class of hardware is **PCIe staging and 88 GiB of aggregate GPU memory, not raw GEMM throughput**. The runtime therefore keeps routed experts in a pinned per-layer CPU arena, runs attention and shared-expert paths on GPU with custom INT8 GPU kernels built for Turing GPUs and a fused attention-decode prefuse kernel, and overlaps expert staging with compute through a logical prefill/decode scheduler, bucketed MoE prefill kernels, and cross-layer routed-expert prefetch. The runtime validates **the 65,536-token prompt path at about 255 prefill tok/s** and reaches a mean **3.16 decode tok/s** on a short-decode benchmark. **Frontier-scale MoE execution on legacy hardware is possible** when the runtime is designed around capacity and staging rather than GEMM peak. Code is available at https://github.com/lvyufeng/deepseek-v4-2080ti.

1 Introduction

Running frontier-scale open language models is increasingly tied to the availability of recent accelerator features. DeepSeek-V4-Flash, a 284B-parameter MoE with 13B active parameters per token, is designed for hardware that supports FP8 and FP4 tensor cores, sparse and indexed attention, NVLink-class GPU interconnect, and modern kernel libraries such as FlashAttention [3]/FlashInfer [29] or compiler stacks such as TileLang [26]. Hardware in this class is expensive, supply-constrained, and unevenly distributed. The implicit consequence is that the strongest open models effectively become inaccessible on widely available, low-cost hardware — not because the math does not fit, but because the runtime stack does not target it.

This work starts from the opposite assumption. The motivating goal is that a 2023-era four-GPU RTX 2080 Ti server, assembled for under CNY 20,000 and built around 2018-vintage Turing GPUs, an older AVX2-only dual-socket CPU, no NVLink, and only PCIe Gen3, should be able to run DeepSeek-V4-Flash end-to-end while preserving the model's routing semantics and avoiding additional sub-INT8 compression risk. Two existing camps fall short of this target. ktransformers [13] performs heterogeneous CPU/GPU execution, but its high-throughput paths depend on newer Intel CPUs with AMX-class matrix instructions that the target machine lacks. llama.cpp [10] can ingest very large models on legacy hardware, but typically does so through aggressive low-bit quantization, which is a different tradeoff from

preserving a W8A8 execution path. Neither path matches the combination of legacy GPUs, legacy CPUs, and exact top-6 MoE routing.

The runtime described here closes that gap. On the target machine it serves DeepSeek-V4-Flash as INT8 W8A8 with exact top-6 routing, validates the 65,536-token prompt path at about 255 prefill tok/s, and reaches a mean 3.16 decode tok/s on a 29-prompt-token / 127-decode-token short-decode benchmark. The result is less a raw-GEMM story than a placement and movement story: every effective optimization either reduces PCIe staging cost, hides it under compute, or keeps a latency-sensitive path off PCIe altogether.

Contributions.

- An INT8 W8A8 inference path for DeepSeek-V4-Flash that preserves the model's exact top-6 routing and avoids additional sub-INT8 compression while keeping the heavy matrix paths in an execution regime supported by the target hardware.
- Custom INT8 GPU kernels for Turing (`sm_75`), organized as four `__dp4a`-based families that match the recurring decode and prefill shapes (per-token decode GEMM, paired shared-expert GEMM, long-context chunked `wo_a`, and grouped/bucketed routed-MoE GEMM), together with a fused attention-decode prefuse kernel that collapses sixteen eager PyTorch dispatches per layer into two CUDA launches.
- A heterogeneous placement strategy that uses a pinned per-layer CPU arena as the sole physical store for routed-expert weights and stages only the active experts to GPU, while keeping attention and shared-expert paths resident on GPU.
- A logical prefill/decode scheduler that switches resource policy inside one model process and overlaps expert movement with useful compute.
- Prefill- and decode-specific MoE staging optimizations: a bucketed MoE GEMM kernel that processes contiguous expert ranges to remove per-expert padding, and a cross-layer routed-expert prefetch policy that pre-stages the next layer's likely-local experts during the current layer's compute.
- An OpenAI-compatible service path validated end to end on the 65,536-token prompt path on the target machine.
- A documented set of negative results — cuBLAS IMMA INT8, PMADDUBSW sign-trick CPU dot, fused inverse RoPE, KV-cache fold into the fused KV kernel, TileLang/native FP4, and MTP speculative decode under default scheduling — reported because they explain why the adopted system looks the way it does.

2 Background and Constraints

2.1 Hardware platform

The platform combines substantial aggregate GPU arithmetic throughput with three limiting factors: device-memory capacity, PCIe-only data movement, and an older datatype/ISA envelope. The target machine was assembled in 2023 and uses four RTX 2080 Ti GPUs plus a dual-socket Intel Xeon E5-2696 v4 host. Table 1 lists the numerical specification used throughout the report.

Table 1: Hardware specification of the target machine.

Device	Count	Property	Per unit	Aggregate	Unit
RTX 2080 Ti	4	FP32 throughput	13.4	53.8	TFLOPS
		FP16 tensor-core peak	107.6	430.4	TFLOPS
		Device memory	22	88	GiB
		DRAM bandwidth	616	2,464	GB/s
PCIe Gen3 x16	4	One-way payload	15.75	63.0[†]	GB/s
Xeon E5-2696 v4	2	Physical cores	22	44	cores
		Logical threads	44	88	threads
DDR4-2400 (2 NUMA nodes)	2	Capacity	512	1,024	GiB
		Peak DRAM bandwidth	76.8	153.6	GB/s

[†] PCIe aggregate is the theoretical peak over four independent x16 links; effective bandwidth depends on PCIe host-bridge and cross-socket topology as well as host-bridge contention. Other throughput/bandwidth entries are specification peaks.

The four GPUs provide roughly 430 TFLOPS of aggregate FP16 tensor-core peak, so the deployment problem is better characterized as capacity- and movement-constrained than purely compute-limited. The limiting resources are asymmetric: 88 GiB of aggregate device memory versus 1 TiB of host memory, implemented in this runtime as a pinned per-layer CPU arena for routed experts; PCIe Gen3 plus host bridges rather than NVLink for cross-rank synchronization and CPU–GPU staging; and Turing/Broadwell-generation arithmetic support, with FP16 tensor cores and `dp4a` on GPU, AVX2/FMA/F16C on CPU, but no native BF16, FP8, FP4, AVX-512, or AMX matrix path. These constraints define the hardware side of the design space: dense compute should remain on GPU, routed-expert capacity must use host memory, and expert-weight movement must be scheduled explicitly.

2.2 DeepSeek-V4-Flash requirements

Four model-derived requirement categories are relevant for this work.

Capacity. DeepSeek-V4-Flash has 43 Transformer layers, 284B total parameters, and 13B activated parameters per token [5]. Each MoE layer contains one shared expert and 256 routed experts, and each token activates 6 routed experts, following the broader top-k sparse MoE routing lineage [23, 15, 2]. Although execution is sparse, the routed-expert pool is too large to replicate inside the 22 GiB local memory of each RTX 2080 Ti rank. Even an INT8 representation of all parameters would require roughly 284 GB of raw payload before metadata, KV cache, activations, and runtime buffers, exceeding the 88 GiB aggregate device memory. This rules out a simple replicated-GPU placement and requires a non-replicated expert layout.

Datatype. The DeepSeek-V4 report uses low-precision formats that match modern accelerators: routed-expert parameters are stored in FP4; KV entries use BF16 for RoPE dimensions and FP8 for the remaining dimensions; and the lightning-indexer attention path uses FP4 precision. These choices are intended to reduce memory traffic and cache footprint in the intended stack, but none of BF16, FP8, or FP4 is natively accelerated on Turing. Supporting the model on this platform therefore requires datatype substitution, not merely kernel substitution.

Long-context attention. DeepSeek-V4-Flash targets million-token contexts through hybrid CSA/HCA attention. For V4-Flash, CSA uses compression rate $m{=}4$, 64 indexer query heads, indexer head dimension 128, and sparse-attention top-$k{=}512$; HCA uses compression rate $m'{=}128$ [5]. These long-context mechanisms require sparse and indexed attention semantics that must be preserved even when the original FP4-oriented kernels cannot be used as hardware-native kernels on `sm_75`.

Infrastructure assumptions. The DeepSeek-V4 infrastructure discussion describes expert-parallel communication/computation overlap, TileLang-based fused-kernel development, and the DeepSeek-V3 MTP setup with depth 1 [4, 26]. These assumptions are reasonable on the intended accelerator stack, but they become nontrivial when the accelerator is a PCIe-only Turing GPU and the host is AVX2-only. The runtime must preserve these model semantics while replacing the original infrastructure assumptions.

2.3 Model–platform mismatch

The mismatch between DeepSeek-V4-Flash and the four-RTX 2080 Ti platform can be grouped into eight constraint classes. Table 2 summarizes the mismatch induced by combining the model requirements above with the hardware limits in Table 1. Capacity and activation sparsity determine placement; expert datatype, KV storage, and sparse attention determine datatype and kernel choices; communication determines scheduling; and kernel-stack plus speculation constraints motivate selective reimplementation of upstream infrastructure components.

The implementation sections use this decomposition as a guide: placement and EP sharding are described in Section 3, datatype and kernels in Section 4, scheduling and overlap in Section 6, and long-context service behavior in Section 7.

2.4 Gap in existing inference stacks

Existing inference approaches each address part of this design space, but none directly targets the complete constraint set. Heterogeneous CPU/GPU runtimes address capacity but often depend on faster host-side matrix paths than this Broadwell platform provides. Aggressive low-bit runtimes address fit by reducing model precision, which conflicts with avoiding the quality loss associated with aggressive quantization. Modern serving and kernel stacks address scheduling, KV management, or kernel productivity on recent accelerators, but they do not resolve the combined absence of FP4/FP8/BF16 support, NVLink, and sufficient GPU memory on this platform. Section 10 discusses representative systems in each category.

Table 2: Constraint decomposition for DeepSeek-V4-Flash on the target platform. DeepSeek-V4 details are from the official technical report [5].

Constraint class	Model-side constraint	Platform limit
Capacity	284B MoE, 256 experts/layer	88 GiB VRAM
Activation sparsity	top-6 experts/token	dynamic active set
Expert datatype	FP4 routed experts	no FP4 MMA
KV storage	BF16/FP8 KV	no BF16/FP8
Sparse attention	CSA/HCA, FP4 indexer	no FP4 / modern kernels
Communication	EP dispatch/combine	PCIe, no NVLink
Kernel stack	TileLang-based kernel stack	no `sm_75` FP4 target
Speculation	MTP depth 1	sync-sensitive decode

The system design in Section 3 follows this constraint decomposition: dense latency-sensitive paths remain on GPU, routed-expert capacity is placed in the pinned per-layer CPU arena, and PCIe transfers are scheduled to overlap with useful compute.

3 System Overview

This section describes the four-rank placement, the per-layer collective sequence, and the phase-dependent resource policy. The system follows three placement invariants derived from Section 2: dense latency-sensitive paths remain on GPU, routed-expert capacity is held in a pinned per-layer CPU arena, and expert-weight PCIe transfers are issued through explicit copy streams rather than delegated to framework-level offload heuristics.

Figure 1 shows the resulting four-rank layout. Each rank owns one GPU and one expert-parallel shard of the routed-expert pool: with 256 routed experts and four ranks, this is 64 experts per rank. The dense/output projection and selected attention-side projections (`wo_a`, `wo_b`, and `wq_b` when the lightning indexer runs sharded) are tensor-parallel across the four GPUs, following the same model-parallel principle used by large Transformer systems [24]; `wq_a`, `wkv`, and the shared expert are replicated because the small-latent components on the compressed side of MLA gain little from sharding and the shared expert is small relative to the routed experts.

In the configuration used for the reported measurements, a layer issues three All-Reduces through NCCL collectives [20]: after the attention-side TP projections, after the routed-expert partial outputs, and after the dense/output projection. The shared-expert output is replicated and added locally on each rank rather than passing through a collective. Optional variants are configuration-dependent: a sharded lightning indexer contributes one further per-layer All-Reduce on the indexer score, and CPU MoE can replace the routed-expert reduce with a CPU-side host reduce.

The runtime uses one distributed model instance and switches resource policy through the prefill/decode (PD) scheduler, matching the broader observation that Transformer serving has phase-dependent compute and memory behavior [21]. In prefill, once the token batch exceeds a configurable threshold (default 64 tokens), routed experts are copied from the per-rank shard of the CPU pinned arena into per-rank GPU staging buffers and MoE runs on GPU alongside attention, the shared expert, and the TP projections; below the threshold, or when the GPU prefill path is disabled, routed MoE stays on the CPU path. In decode, routed MoE is dispatched to the CPU MoE path described in Section 5 while the GPU runs the shared expert. When cross-layer prefetch is enabled, the predicted local experts for the next layer are staged on a copy stream during current-layer compute.

Because the platform has no NVLink, cross-rank All-Reduces share the PCIe fabric with expert staging and would serialize on the same physical link absent overlap. When the runtime is configured to issue them on a dedicated CUDA stream, the CPU MoE worker, GPU shared-expert compute, and the routed-expert reduce overlap rather than serialize. The primary decode-best configuration in Section 8 enables GPU prefill MoE, cross-layer prefetch, and the asynchronous All-Reduce stream. Sections 4–7 elaborate this layout at the kernel, expert-storage, scheduling, and service layers.

4 Datatype and Kernel Design

This section describes the runtime's INT8 W8A8 precision policy, the four custom Turing INT8 kernel families that carry it, and the decode-side attention prefusion that collapses the per-layer launch overhead.

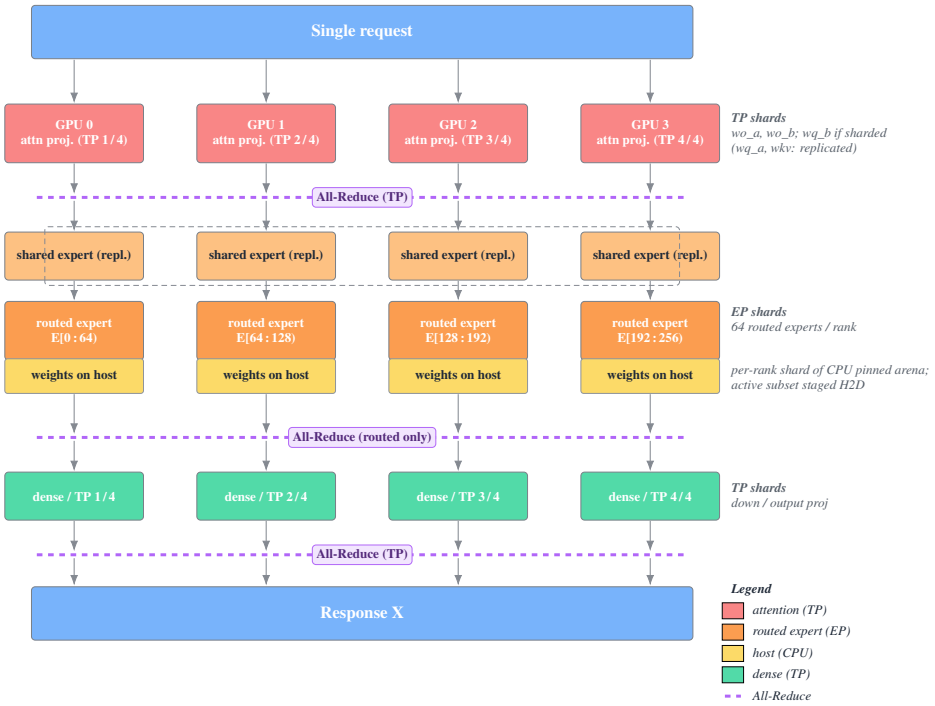

Figure 1: System layout on the four-rank RTX 2080 Ti server. Tensor parallelism shards the dense/output projection and selected attention-side projections (`wo_a`, `wo_b`, and sharded `wq_b`); `wq_a`, `wkv`, and the shared expert are replicated; the 256 routed experts are split expert-parallel as 64 per rank. INT8 routed-expert weights live in a per-rank shard of the CPU pinned per-layer arena, with the active subset staged to GPU over PCIe. The three All-Reduce bars are, top to bottom: TP after attention-side projections, routed-expert reduce, TP after dense.

4.1 Precision policy

The reference DeepSeek-V4-Flash design assumes datatypes that Turing does not implement natively: routed experts use FP4, KV storage mixes FP8 and BF16, and several kernels depend on tensor-core paths that `sm_75` does not expose. The runtime cannot inherit any of this directly. Instead, it applies a module-selective INT8 W8A8 policy, a regime related to prior large-model INT8 and W8A8 quantization work [7, 28]: the heavy matrix paths move to a datatype both the target GPU (via `__dp4a`) and the CPU MoE backend can execute efficiently, while routing, vocabulary, and compression-sensitive tensors stay in higher precision. Table 3 summarizes the resulting assignment.

Table 3: Runtime precision policy.

Module class	Scope	Runtime precision	Runtime treatment
Routed experts	Per-layer EP shard	INT8 W8A8	CPU pinned arena; GPU staged when active
Shared expert	Replicated GPU path	INT8 W8A8	GPU resident; paired INT8 GEMM
Attention projections	`wq_a`, `wkv`, `wo_a`, `wo_b`	INT8 W8A8	Online/preloaded INT8; `wo_a` FP16 in prefill, INT8 in decode
Indexer projection	`wq_b`	INT8 W8A8	Sharding-gated custom INT8 path
Gates	Router logits	FP16/FP32	Kept high precision for routing
Embeddings / LM head	Vocabulary path	FP16	Kept FP16 on vocabulary path
RMSNorm / compressor / sink	Norm scales, `Compressor.ape`, `attn_sink`	FP32	Held in FP32 throughout

Quantization scheme. Weights use per-row symmetric INT8: each output row carries an FP32 scale, $w_q = \mathrm{round}(w/s)$ with $s = \max|w_{\mathrm{row}}|/127$. Activations are quantized online per token in the same kernel that consumes them (`quantize_rows_kernel`), producing an INT8 tensor and a per-row FP32 scale. The GEMM accumulates INT8 $\times$ INT8 products into INT32 and the dequantization step folds the per-row activation scale and per-column weight

scale into a single multiply that emits the FP16/BF16 output, so no separate scale-application launch follows. There is no zero point, no per-channel weight quantization, and no static activation calibration; this keeps the kernels tight and avoids running an offline calibration pass against the converted checkpoint.

Phase-split for `wo_a`. The `wo_a` projection is the one module whose precision is chosen per phase rather than once at load time. Prefill is memory- and launch-geometry constrained at long sequence lengths, where keeping the FP16 `wo_a` path avoids per-row INT8 quantization cost on already large activation tiles; decode is dominated by small-batch dispatch and per-token projection overhead, where INT8 `wo_a` is a clear net win. The default service configuration enforces this split with `DEEPSEEK_PD_PREFILL_WO_A_INT8=0` and `DEEPSEEK_PD_DECODE_WO_A_INT8=1`, switched by the PD scheduler (Section 6) at phase boundary.

Reusing the dispatcher rather than rebuilding the graph. A subtle point is that conversion correctness and runtime efficiency are decoupled. Offline-quantized full-checkpoint INT8 tensors match online quantization for the targeted modules, but loading them through a parallel "native INT8" graph rebuilt on top of library kernels is consistently slower, because that graph picks library shapes that do not match the small-batch decode and grouped-MoE geometry of the serving path. The runtime therefore feeds full-checkpoint INT8 tensors into the existing online/preloaded dispatcher rather than rewriting the graph, which keeps the proven control flow and lets the custom kernels described next decide the launch shapes.

Runtime validation. INT8 kernels are bit-exact against their library reference where one exists: `tests/test_int8_gemm_imma.py` reports `max_abs=0` for `int8_gemm_rows_kernel` versus cuBLAS IMMA on all decode shapes, and the fused decode prefuse kernels match the eager PyTorch reference within bf16 ULP (`tests/test_fused_attn_prefuse.py`). End-to-end, enabling INT8 on `wo_a` produces byte-identical greedy completions to the FP16 `wo_a` path on both a 5-prompt smoke set and a 16K-context prompt; the 64K long-context regression round-trips the canonical `OK` response as a smoke check against obvious long-context corruption. Tighter perplexity-style quality numbers are not part of this report; the present checks are sufficient for the *runtime* claim that the W8A8 path preserves the tested execution semantics without changing routing decisions.

4.2 INT8 execution on Turing

Turing exposes the `__dp4a` integer dot-product instruction and an INT8 IMMA (integer matrix multiply-accumulate) tensor-core path through CUDA/cuBLAS facilities [19, 18]; both give INT8 a usable arithmetic lane on `sm_75`. Library INT8 kernels, however, are mostly tuned for Ampere-or-newer IMMA throughput and medium-to-large M batches; on the geometry this runtime presents – per-token decode projections (small M), paired shared-expert GEMMs, long-context low-rank attention, and grouped routed-MoE – they leave a large fraction of SM time on the table or pay launch and layout overheads that exceed their arithmetic advantage. The runtime therefore implements four custom INT8 kernel families in `src/csrc/cuda_kernel_impl.cu`, each matched to one of the recurring shapes:

- **Per-token decode GEMM** (`int8_gemm_rows_kernel`). Drives the small-M attention projections `wq_a`, `wq_b` (the lightning-indexer projection), `wkv`, and `wo_b`. Decode rows are quantized in the same launch and the kernel runs row-major directly, avoiding the layout transposes a generic library kernel would impose.
- **Paired shared-expert GEMM** (`int8_gemm_pair_forward_cuda`). Issues the two shared-expert projections (`w1`, `w3`) back to back over a single quantized activation buffer; the SwiGLU and `w2` stages are handled by separate kernels in the same path. Sharing the input quantization across the pair removes one quantize launch per layer.
- **Long-context `wo_a` projection** (`wo_a_int8_forward_cuda`). At 65,536-token prefill, a single launch needs rows $\times$ groups in the grid Y dimension, which exceeds the CUDA `gridDim.y` cap of $65{,}535$ by exactly one and returns `cudaErrorInvalidConfiguration`. The kernel therefore slices rows into chunks of at most $65{,}535$ and dispatches one grouped INT8 GEMM launch per chunk, with row offsets folded into the input pointers.
- **Grouped / bucketed routed-MoE GEMM** (`moe_prefill_int8_grouped_gemm_*`, prefill only). Used by the GPU-prefill MoE path; one launch covers all active experts of a layer with per-expert ranges, with a bucketed variant for skewed routing.

Beyond the four families above, a cuBLAS IMMA INT8 path (`int8_gemm_imma_cuda`) is also implemented for the per-token decode and shared-expert shapes, which all satisfy IMMA's M/N/K-aligned-to-16 constraint on `sm_75`, but it is gated off by default (`DEEPSEEK_INT8_GEMM_IMMA=0`).

The reason for keeping IMMA off by default is worth measuring directly. Table 4 reports per-call CUDA-event time on the actual decode shapes (TP = 4, $M=1$): IMMA matches or beats the dp4a kernel on every shape, with up to -28% GPU time on `wq_a`, so per-kernel arithmetic is not the issue. End-to-end, however, two long/long decode runs with the CPU frequency governor pinned to performance changed from $\{2.057,\ 2.097\}$ to $\{2.030,\ 1.847\}$ tok/s under IMMA, a mean -6.6% regression across the small A/B sample.

The end-to-end regression is consistent with the production trace, which shows the per-token decode GEMMs are not the binding constraint. On `decode_step2_rank0` (one decode step with the default INT8 dp4a path), the trace spans 501.2 ms wall and accumulates 294.1 ms of self-CUDA time, i.e. the GPU is $\sim 58.7\%$ active and $\sim 41.3\%$ idle. Of the active fraction, the dominant single slice is the cross-rank `ncclDevKernel_AllReduce_Sum_bf16` stage at 76.6 ms (26.1% of self-CUDA), with the dp4a `int8_gemm_rows_kernel` at 22.4 ms (7.6%). IMMA reduces part of that 22.4 ms slice but adds a per-call `dequant_int32_to_bf16_kernel` launch and an INT32 accumulator allocation; on a path where the binding constraints are All-Reduce, the CPU MoE worker, and dispatcher overhead, those extra launches outweigh the arithmetic gain.

This is a dispatcher-bound regime rather than an arithmetic-bound one, so the comparison is worth revisiting only after launch overhead is reduced. The IMMA code path is preserved behind `DEEPSEEK_INT8_GEMM_IMMA=1` so the experiment can be re-run after CUDA Graph capture collapses the per-launch cost; on a graph-captured replay the per-call dequant launch becomes amortized rather than serial, and IMMA's per-kernel advantage in Table 4 could plausibly turn net-positive. Section 8.4 returns to this comparison alongside other negative results.

Table 4: Per-call kernel time of `int8_gemm_rows_kernel` (dp4a) versus `int8_gemm_imma_cuda` (cuBLAS IMMA) on the actual decode shapes ($M=1$, TP = 4), measured on RTX 2080 Ti via CUDA events over 400 steady-state iterations. "Wall" is the per-call `perf_counter` delta around `torch.cuda.synchronize()`-bracketed iterations; "GPU" is the `cudaEvent` elapsed time inside the same call. Both columns are measured around the public `int8_gemm_forward()` entry point, so the IMMA columns include the per-call `dequant_int32_to_bf16_kernel` dispatch and INT32 accumulator allocation, while the dp4a columns include only the fused single-launch path. For very small kernels (`idx.wq_b`, `shared.w2`) wall is below GPU because the launch is queued before the previous kernel finishes, so launch overhead overlaps with execution; the two columns are ordered consistently here for readability.

Projection	K	N	dp4a wall (μs)	dp4a GPU (μs)	IMMA wall (μs)	IMMA GPU (μs)
`wq_a`	4096	1024	118	119	85	86
`wq_b`	1024	8192	53	49	43	48
`wkv`	4096	512	85	87	85	86
`wo_b`	2048	4096	45	49	45	48
`idx.wq_b`	1024	2048	24	38	24	38
`shared.w2`	512	4096	15	38	15	38
`shared.w1+w3` (paired)	4096	512	166	167	166	167

Kernel implementation notes. A few details from `src/csrc/cuda_kernel_impl.cu` are worth recording, because they are what makes the four families above small and fast on this geometry:

- **Per-token decode GEMM.** `int8_gemm_rows_kernel` stages the quantized activation row into shared memory once per CTA as packed `int` (each `int` holds four INT8 elements), and the inner loop is a tight `__dp4a(int, int, int)` reduction over $K/4$ packs into a single INT32 accumulator. The per-row activation scale and per-column weight scale fold in only after the loop, so no FP arithmetic appears on the critical path. The launch grid is $(\lceil N/\text{block} \rceil, \text{rows}, \text{groups})$, which lets the same kernel cover both the ungrouped per-token decode shapes and the grouped `wo_a` long-context shape from a single source.
- **Paired shared-expert GEMM.** `int8_gemm_pair_forward_cuda` quantizes the activation once and issues two back-to-back `int8_gemm_rows_kernel` launches against `w1` and `w3`, sharing the `x_q`/`x_scale` buffers. The only additional cost over a single GEMM is the second kernel launch, which is exactly what cuBLAS would otherwise charge twice for separate `w1` and `w3` calls and would also require materializing the activation INT8 buffer twice.
- **Long-context `wo_a` projection.** `wo_a_int8_forward_cuda` performs the row-chunk slicing described above by folding `row_offset` into the pointer arithmetic of `x_q`, `x_scale`, and `out`; the kernel itself is unchanged, so a 65,536-token prefill issues a small handful of identical launches rather than one launch that would exceed the `gridDim.y` cap.
- **Grouped / bucketed routed-MoE GEMM.** `moe_prefill_int8_grouped_gemm_*` processes contiguous expert ranges in a single launch with per-expert segment pointers, and the bucketed variant pads each bucket to

the maximum token-count inside that bucket rather than to the global maximum, which keeps padded compute within a few percent of useful compute on skewed routings (Figure 4, Algorithm 1).

The cuBLAS IMMA reference path differs from the dp4a families in one additional dispatch: `cublasGemmEx` produces an INT32 accumulator with no scale fold, so a separate `dequant_int32_to_bf16_kernel` launch is required after it. This is a structural cost, not an artifact of how it is wired up, and is part of why the IMMA wall figures in Table 4 sit at or above their dp4a counterparts even when the GPU column is faster.

4.3 Decode-side attention prefusion

The decode attention path is dominated by launch overhead rather than arithmetic intensity. The front-end work per layer is on the order of 10^5 FLOPs (RMSNorm [30] and RoPE [25] on a 16×512 q-tile and a 512-dim KV row, plus a per-row INT8 quantization). The relevant peak is the FP32 throughput of the RTX 2080 Ti CUDA cores, $\sim$13.45 TFLOPS, since these elementwise and reduction kernels do not touch the tensor cores; against that peak, the front-end arithmetic is sub-microsecond per layer, while the eager reference spends close to a millisecond. The remainder is dispatcher cost. Eager PyTorch issues roughly sixteen small dispatches per layer for the q-side (RMSNorm + RoPE) and the KV-side (RMSNorm + RoPE + activation quantization) of the attention front end. The decode-side attention prefusion kernel preserves the same math but collapses this front end into two CUDA launches per layer (Figure 2); it is enabled in the reported configuration with `DEEPSEEK_FUSED_ATTN_PREFUSE=1`.

The per-layer effect is direct. Table 5 measures one decode-step front end on RTX 2080 Ti (B = 1, S = 1, H = 16, D = 512): the fused 2-kernel path takes 84.6 μs wall and 29.3 μs GPU per layer, against 1,089.6 μs wall and 996.1 μs GPU for the eager 16-dispatch reference – a 13$\times$ wall and 34$\times$ GPU reduction on this slice alone. Across the 43 layers of one decode step that is $\approx$ 43 ms of dispatcher cost removed from the critical path. Benchmark names use prompt/decode length pairs: short/short, short/long, long/short, and long/long combine short or long prompts with short or long decode targets, while the short-decode headline case uses 29 prompt tokens and 127 decode tokens. End-to-end this lifted decode from 1.962 to 2.099 tok/s on short/short and from 1.750 to 1.828 tok/s on long/long with the CPU frequency governor pinned to performance; both data points are part of the optimization history summarized later in Section 8. The conversion is consistent with this microbench budget: 1.962 $\rightarrow$ 2.099 tok/s is $\sim$33 ms saved per decode step, against the $\sim$43 ms predicted at the GPU layer; the residual $\sim$10 ms gap is the part of the front-end window that already overlapped with concurrent work in production. It is the largest fusion in the runtime that consistently produced wall-clock improvement; finer per-op fusions (Section 8.4) tended to be bit-exact but throughput-neutral, because once dispatcher count is no longer the binding constraint, removing a handful of additional launches falls within the run-to-run noise of the benchmark.

Table 5: Per-layer attention front end on RTX 2080 Ti, measured via CUDA events over 400 steady-state iterations (B = 1, S = 1, H = 16, D = 512). The fused path replaces the eager 16-dispatch reference with two CUDA launches. Wall time includes a per-iteration `q.clone()/kv.clone()` for both rows so that the in-place fused kernels and the destructive eager path are measured on identical fresh inputs; the production decode trace, where this clone is unnecessary, shows roughly 7 μs/layer for the fused path.

Path	Launches/layer	Wall (μs)	GPU (μs)	Per-step total wall (43 layers, ms)
Eager PyTorch reference	$\sim$16	1,089.6	996.1	46.9
Fused CUDA prefuse	2	84.6	29.3	3.6

These datatype and kernel choices define the GPU-resident compute path; the next section describes how the routed-expert capacity is supplied from host memory.

5 CPU-Resident Routed Experts

This section describes the pinned per-layer arena that serves as the sole backing store for routed-expert weights, the active-expert CPU execution path that consumes it, and the async overlap that hides routed-MoE wall time behind shared-expert and All-Reduce work.

5.1 Pinned per-layer arena

In the reported in-process backend, routed-expert weights live in a per-layer pinned CPU arena that is the sole physical store for those tensors. The arena uses CUDA pinned-memory behavior to make host-to-device staging explicit and asynchronous [19]. Each layer owns six contiguous pinned tensors for the rank-local routed-expert shard: `w1.weight`,

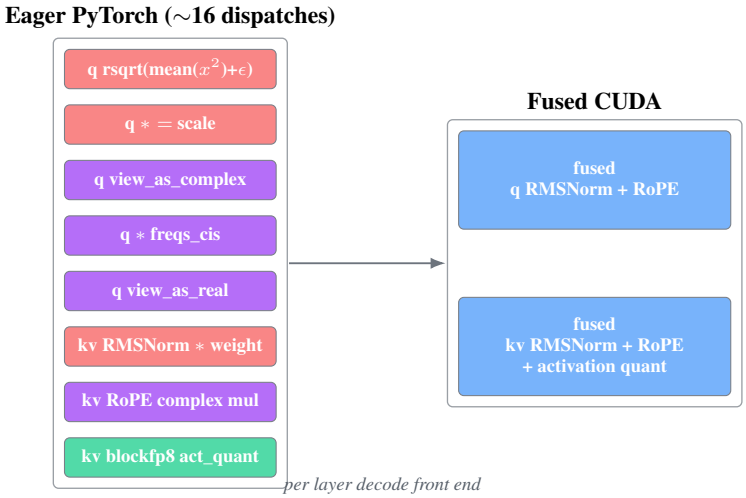

Figure 2: Decode-side attention prefusion. Sixteen eager PyTorch dispatches per layer collapse into two fused CUDA launches; per-call timings are reported in Table 5.

`w1.scale`, `w2.weight`, `w2.scale`, `w3.weight`, and `w3.scale`. For the shipped W8A8 configuration (dim $= 4096$, moe_inter_dim $= 2048$), one routed expert occupies about 25.2 MB of INT8 weights plus FP32 row scales. With 64 local experts and 43 layers, that is about 69.3 GB (64.6 GiB) of pinned expert payload per rank, or 258 GiB across four ranks. A second full staging copy would push routed-expert storage toward half a terabyte, before KV cache, activations, process overhead, and OS page cache; the arena therefore has to be the backing store, not a cache layered on top of loaded parameters.

Earlier prototypes left duplicate parameter storage beside the pinned arena and paid `cudaHostRegister` cost during long prefill. Materializing INT8 weights directly into the arena, then repointing each expert's CPU weight/scale buffers at arena slices, released the duplicate storage, reduced measured host RSS by roughly 10.5 GB in the long-prefill A/B, and removed about 29 s of registration overhead at the start of a 64K prefill.

The 10.5 GB RSS delta is the measured reduction in that benchmark window, not the theoretical full-arena payload; the payload calculation above explains the capacity pressure that makes duplicate backing stores unacceptable. The per-layer layout also turns GPU prefill staging into a small number of large H2D copies rather than many scattered per-expert copies, which is what keeps PCIe staging within budget.

5.2 Active-expert CPU execution

During decode, the reported path keeps routed-expert weights on CPU and moves only the token activations, selected expert ids, routing weights, and final routed output across PCIe. The CPU backend (`src/moe/cpu_backend.py`) copies these small tensors into double-buffered pinned slots (`buffer_depth=2`, indexed by layer id), then runs the rank-local top-6 routed MoE in a native OpenMP extension. Routing semantics are preserved: the runtime keeps top-6 expert selection rather than pruning experts to fit the machine; faster but semantically risky shortcuts (e.g., routing top-k trimming) were prototyped and rejected.

The native INT8 CPU kernel is deliberately AVX2-shaped. Broadwell lacks VNNI / VPDPBUSD, so `src/csrc/deepseek_cpu_moe_ext.cpp` widens signed INT8 lanes with `_mm256_cvtepi8_epi16`, multiplies with `_mm256_mullo_epi16`, and reduces adjacent products with `_mm256_madd_epi16`. Thread-local scratch buffers hold the quantized input, hidden activation, hidden scales, and partial reductions, eliminating per-call vector allocation in the decode hot path. The PMADDUBSW sign-trick variant reported later in Section 8.4 improved the inner dot microbenchmark, but did not move end-to-end TPS because the overlap path described next already absorbs most of the residual CPU wait, so a faster inner dot does not surface at the decode-step level.

The implementation contains several execution modes; the reported service defaults to the in-process executor unless explicitly configured otherwise:

- **In-process executor (default).** `submit_forward` copies inputs into pinned slots and schedules one Python `ThreadPoolExecutor` worker, which calls the OpenMP native INT8 kernel. This starts CPU MoE early so the main thread can launch GPU shared-expert work in parallel.
- **Persistent worker.** A daemon `_PersistentCPUWorker` accepts one task through a condition-variable queue; tested but slower than the standard executor, because the handoff cost outweighed the thread-reuse benefit.
- **Persistent native team.** A C++ `PersistentParallelTeam` keeps `std::thread` workers alive across native calls instead of re-entering an OpenMP fork/join region; useful when OpenMP team construction dominates.
- **External / rank-0 CPU server.** CPU MoE runs in a separate process or on rank 0, with request/ack state in POSIX shared memory; used for service-isolation experiments and to reduce duplicated CPU-side execution.
- **Shared weight arena.** A header-validated POSIX shared-memory segment (names of the form `dsv4_moe_..._rankN`) stores the layer-by-expert INT8 tensors and scales for `w1`, `w2`, and `w3`, so server-style deployments read a single page-cached image instead of each worker holding a private expert-weight copy.

The same pinned arena also feeds optional GPU execution paths. For long prefill, Section 6 describes how the scheduler stages bounded layer slabs into GPU buffers and runs grouped MoE GEMM. For decode experiments that choose a GPU active-expert path, only the selected experts are staged, not the full 64-expert local shard. These variants change where the arithmetic runs, but they reuse the same top-6 routing decisions and the same per-layer arena layout.

5.3 Async CPU MoE / shared-expert overlap

A naive decode loop runs CPU routed MoE inline before GPU shared-expert work, serializing two phases that have no real dependency. The adopted path decomposes one MoE layer into four cooperating lanes: (i) a CUDA copy stream moves the token activation, expert ids, and routing weights into pinned CPU buffers; (ii) the CPU worker runs the OpenMP routed-MoE kernel once the copy event is ready; (iii) the main GPU compute stream launches the shared-expert INT8 GEMMs immediately after `submit_forward`; and (iv) the async All-Reduce stream waits for the CPU output, copies it back to GPU, and drains the routed-expert reduce. Figure 3 shows these lanes and contrasts them with the serial baseline. The critical path goes from

$$T_{\text{CPU routed}} + T_{\text{GPU shared}} \quad \text{to} \quad \max(T_{\text{CPU routed}}, T_{\text{GPU shared}}) + T_{\text{sync residual}}.$$

The useful window is the GPU shared-expert plus side-stream collective work that overlaps the CPU routed-MoE interval. In the overlap A/B, shared expert plus NCCL side-stream work covered roughly 1.7 ms/layer, consistent with the trace in Section 4, where the 76.6 ms All-Reduce slice amortizes to 1.8 ms over 43 layers. The OpenMP CPU MoE was about 3 ms/layer, so $T_{\text{CPU routed}}$ is the longer arm of the max; the optimization removes the serial GPU-shared and side-stream window rather than reducing CPU MoE itself.

With the CPU governor pinned to performance, enabling this overlap improved decode across all four benchmark cases: short/short $2.099 \rightarrow 2.424$ tok/s, short/long $1.800 \rightarrow 2.466$ tok/s, long/short $1.720 \rightarrow 2.366$ tok/s, and long/long $2.057 \rightarrow 2.378$ tok/s. Three fresh long/long runs sustained 2.417, 2.370, and 2.392 tok/s (mean 2.393, $\sigma < 1\%$). A host-side reduction variant implements the CPU-side host reduce anticipated in Section 3: when routed-MoE output remains on CPU, `host_float_allreduce` can reduce through POSIX shared memory and perform one H2D copy after the reduce, avoiding NCCL for that routed-output path; the reported service default keeps the GPU All-Reduce stream and uses the host-reduce path only when explicitly enabled.

With the expert arena and decode overlap mechanisms fixed, the remaining question is scheduling: which phase owns the accelerator, which MoE kernel runs, and how early active experts are staged.

6 Scheduling and Overlap

Section 5 established the CPU-resident expert arena; this section turns that placement into a timing policy. The runtime combines a logical prefill/decode scheduler, a bucketed grouped-GEMM path for long-prefill MoE, and a cross-layer prefetch policy for decode-time active-expert staging. The common objective is to keep routed-expert capacity in the pinned per-layer CPU arena while issuing PCIe transfers early enough that they overlap with useful compute rather than appearing as serialized stalls.

6.1 Logical prefill/decode scheduler

Prefill and decode want very different resource policies, but running them as separate model processes would duplicate weights and triple memory pressure. The logical PD scheduler in `src/runtime/pd_scheduler.py` is therefore a

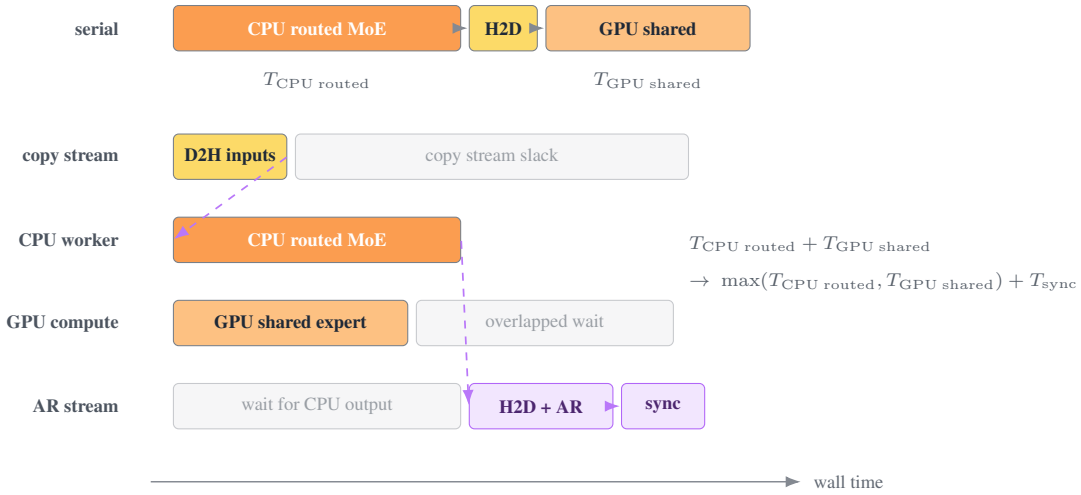

Figure 3: Async CPU MoE / GPU shared-expert overlap during decode. Yellow blocks denote copy / host-staging work, orange is CPU routed MoE, light orange is GPU shared-expert compute, and purple is H2D plus All-Reduce synchronization. The serial path runs CPU routed MoE before GPU shared-expert work; the adopted path launches CPU work, GPU shared-expert compute, H2D, and All-Reduce on separate lanes so the critical path becomes a max plus a small synchronization residual.

phase state machine, not a separate serving stack. It keeps prefill and decode queues, moves a request to the decode queue once prefill owns the KV state, gives decode priority when both queues are nonempty, and switches the process-wide active phase through `DEEPSEEK_PD_ACTIVE_PHASE`. On phase changes it can apply phase-specific attention INT8 gates, OpenMP thread counts, CPU affinity, and NUMA node placement without reloading the model.

The canonical benchmark wrapper `scripts/run_best_scheduler.sh` freezes the adopted policy, and the OpenAI service installs the same defaults before runtime imports. Decode uses the in-process CPU routed-MoE executor with `DEEPSEEK_PD_DECODE_OMP_THREADS=12`, `DEEPSEEK_CPU_DECODE_INLINE_THRESHOLD=0`, async All-Reduce, shared-expert INT8, and fused attention prefuse, while centralized CPU-MoE server and shared-weight-server variants are disabled. Prefill enables GPU routed-MoE only outside the decode phase and only when the token count crosses the default 64-token threshold; below that threshold, or when the GPU prefill path is disabled, routed MoE stays on the CPU path.

The GPU-prefill path is resource-managed rather than a direct kernel substitution. The backend stages local experts through a per-device CUDA copy stream, reuses a two-slot pinned CPU staging workspace, evicts staged layers through a three-layer LRU, stages the whole 64-expert local shard as a dense copy once the active set exceeds 75% of the shard, and caps compact expert slabs with a process-wide 3 GiB budget. These controls keep GPU prefill from turning a short-lived speedup into persistent VRAM pressure. With the phase policy in place, the next bottleneck is the prefill MoE kernel itself.

6.2 Bucketed MoE GEMM for long prefill

Long-context prefill is dominated by routed-expert MoE compute. The first grouped MoE GEMM path padded work by expert group, wasting both memory and compute on sparse route distributions. The adopted path first groups local token-to-expert routes into `route_tokens`, `route_weights`, and `seg_starts` using the CUDA `moe_group_routes` binding, then launches a bucketed grouped-GEMM kernel over contiguous expert ranges (Figure 4, Algorithm 1). The default bucket width is 16 experts, so padding is bounded inside each bucket rather than across the full 64-expert local shard.

The 64K ablation isolates the bucketed kernel effect: a stable non-bucketed default took about 289.52 s, while the adopted bucketed default reduced the run to 259.87 s. A more aggressive bucketed+chunk-512 run reached 250.76 s, but the report keeps 259.87 s as the reproducible bucketed default. The separately reported 257.45 s run in Section 7 is the validated service-path content-check run under current defaults, not a contradictory point on the same ablation curve.

A later binding fix addressed route grouping itself rather than GEMM padding. Moving grouping out of Python avoids a `tolist()`-driven GPU-to-CPU synchronization; the effect is largest at shorter prefill lengths, where the sync is large

relative to compute. That change reduced a 2K prompt from 13.39 s to 7.99 s without regressing 16K or 64K. Bucketed GEMM addresses the prefill compute side; decode needs a different policy because active-expert staging happens one token at a time.

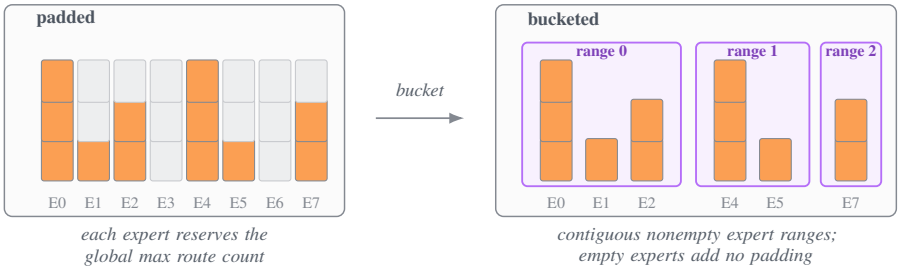

Figure 4: Padded vs bucketed grouped MoE GEMM. Padded scheduling reserves the maximum per-expert route count across the whole local shard; bucketed scheduling restricts padding to contiguous nonempty expert ranges, so empty experts no longer inflate the grouped-GEMM workspace.

Algorithm 1 Bucketed grouped MoE GEMM scheduling

Require: routed token-to-expert assignment R, local expert range $[E_l, E_r)$, bucket budget B, chunk C

group local routes on GPU, building expert-range table $\mathcal{E}$
BUCKETS $\leftarrow \emptyset$; $q \leftarrow [E_l, E_l)$; $w \leftarrow 0$
for each contiguous nonempty expert run $[a, r) \in \mathcal{E}$ in $[E_l, E_r)$ **do**
  if $w + |[a, r)| \leq B$ **then**
    extend current bucket q to $[q.\text{lo}, r)$; $w \mathrel{+}= |[a, r)|$
  else
    emit q to BUCKETS; $q \leftarrow [a, r)$; $w \leftarrow |[a, r)|$
  end if
end for
emit final q to BUCKETS
for each bucket $b \in$ BUCKETS **do**
  for each chunk of up to C tokens routed into b **do**
    launch grouped INT8 `dp4a` GEMM over expert range b on the chunk
  end for
end for

6.3 Cross-layer routed-expert prefetch

The last major decode improvement predicts the next layer's local experts from the current layer's gate proxy and stages likely local experts on the copy stream during the current layer's compute (Figure 5, Algorithm 2), following the cross-layer-gate prefetch idea in Fate [8]. The predictor $\hat{P}$ is not a learned auxiliary model: for hash-routed layers it reuses the token-id-to-expert table, while for score-routed layers it applies the next layer's gate projection to the current layer's normalized FFN proxy, then uses the same gate score function, bias, and top-K selection as the true gate. The optimization is decode-only and token-count-limited (default at most four tokens), because it targets the regime where per-token H2D staging latency dominates arithmetic.

Figure 5 makes the timing intent explicit. The upper lane is the compute stream: layer ℓ exposes a gate proxy, continues through FFN work, and only later reaches layer $\ell{+}1$ MoE. The lower lane is the copy stream: as soon as the proxy for layer ℓ is available, the runtime predicts likely-local experts for layer $\ell{+}1$ and begins staging them out of the pinned per-layer CPU arena. The overlap window is therefore the current layer's FFN and surrounding decode work, not a separate prefetch phase inserted ahead of MoE. If the prediction hits, the next layer consumes already-staged experts; if it misses, the exact route still falls back to on-demand staging before execution.

The adopted policy is $K = 10$ with a per-rank local limit of 2. The top-K budget gives the next layer enough candidate experts to find local hits, while the local limit caps copy-stream pressure by staging at most two predicted experts per rank. The prefetch is best-effort and does not change routing semantics: predicted-but-unused experts only waste an H2D copy, and missed actual experts are still staged by the exact current-route path before MoE execution, so routing remains the original top-6 decision.

In the short-decode ablation used for the current-best OpenAI number (29 prompt tokens, 127 decode tokens), baseline decode was 2.736 tok/s and $K{=}10$ / local-limit-2 reached 2.926 tok/s in the initial sweep. Three fresh trimmed runs measured 3.135, 3.168, and 3.170 tok/s, with mean 3.158. This is the source of the 3.16 tok/s headline number reported in the abstract.

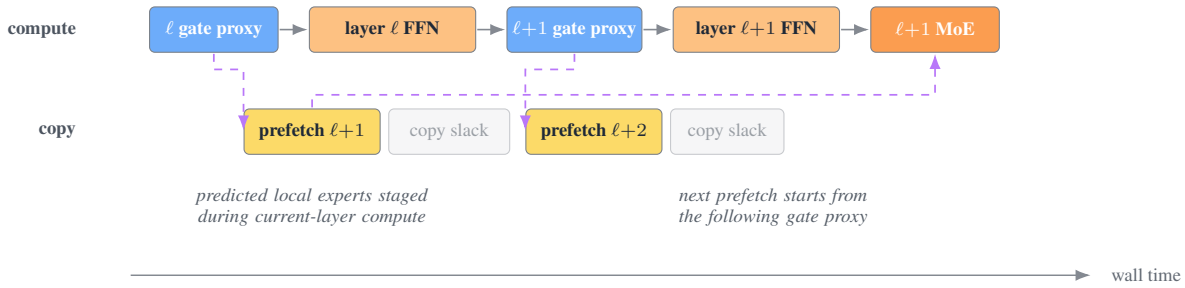

Figure 5: Cross-layer routed-expert prefetch during decode. The compute lane exposes the gate proxy for layer ℓ, continues through current-layer FFN work, and later reaches layer $\ell{+}1$ MoE; the copy lane uses that early proxy to stage likely-local experts for the next layer out of the pinned per-layer CPU arena. The copy stream therefore spends the current layer's compute window on useful staging work, so the next routed-MoE step waits only for misses or residual copies rather than a full cold start. Colors follow the report palette: blue for gate proxies, light orange for FFN compute, yellow for copy-stream work, and orange for routed MoE.

Algorithm 2 Cross-layer routed-expert prefetch (per rank, per decode step)

Require: decode phase, gate proxy g_ℓ, next-layer gate predictor $\hat{P}$, top-K budget K, per-rank local limit L, local expert set $\mathcal{E}_{\text{loc}}$, copy stream S_c, compute stream S_g

predict next-layer expert scores: $\hat{s} \leftarrow \hat{P}(g_\ell)$
$\hat{T} \leftarrow$ indices of top-K predicted experts in $\hat{s}$
$\hat{T}_{\text{loc}} \leftarrow \hat{T} \cap \mathcal{E}_{\text{loc}}$
$\hat{T}_{\text{loc}} \leftarrow$ first $\min(L, |\hat{T}_{\text{loc}}|)$ entries of $\hat{T}_{\text{loc}}$ ranked by $\hat{s}$
for each predicted expert $e \in \hat{T}_{\text{loc}}$ not already on GPU **do**
    on S_c: enqueue async H2D copy from CPU pinned arena to the rank's GPU staging
end for
on S_g: continue layer ℓ compute
on S_g: at start of layer $\ell{+}1$ MoE, wait on copies for actually-routed experts, staging any missed experts on demand

Together, the logical PD scheduler, bucketed prefill kernels, and cross-layer prefetch policy form the timing layer around the CPU-resident expert arena. The next section exercises the resulting long-context and OpenAI service paths end to end.

7 Long-Context and Service Path

Section 6 described the timing policies that make CPU-resident experts usable; this section validates that those policies survive the two execution surfaces that stress them most. The first is the 65,536-token context path, which exercises compressed attention state, long prefill MoE geometry, and generation-loop correctness. The second is the service path: the OpenAI-compatible HTTP wrapper around the same scheduler, kernels, and CPU expert arena. The current service target is single-request chat completion, with a 4K default serving configuration and an explicit 64K validation mode; both streaming and non-streaming completions use the same distributed runtime underneath.

7.1 64K context stabilization

The 65,536-token path is not just a larger allocation target. It stresses CUDA launch geometry, compressed-attention cache sizing, intermediate activation lifetimes, and the test harness itself. The `wo_a_int8_forward` kernel hit CUDA grid limits for long-sequence launches and had to be chunked. Some long-context tests were also false positives when `MAX_MODEL_LEN` equaled the prompt length and the generation loop produced no useful decode; later tests reserve headroom beyond the prompt so the path must execute an actual continuation.

The remaining fixes were memory and MoE-geometry issues. Large intermediate FP32 buffers caused OOMs in the HCA heavy-compression attention path, shared experts, online INT8 projections, MoE reduction/finalization, and final heads; these were addressed with chunking and last-position-only output where appropriate. MoE grouped GEMM chunk size 512 was initially too memory-hungry for effective 64K prefill, so chunk 384 stabilized the path; after bucketed GEMM reduced peak workspace, chunk 512 became viable again under the service defaults. After these fixes, 64K became a stable target. The validation endpoint reported later in Table 6 is a 65,536-token prompt with 257.45 s prefill, approximately 255 prefill tok/s, and a smoke content check that returns `OK`; this validates the long-context execution path, not model-quality parity.

7.2 OpenAI-compatible service

The runtime is exposed through `src/server/openai.py`. Its compatibility surface is intentionally scoped to chat-completion serving rather than full OpenAI API parity: `GET /health` returns status, rank, world size, and model id; `GET /v1/models` returns a single local model entry; and `POST /v1/chat/completions` is the only generation endpoint. The server supports non-streaming chat completions and token-level SSE streaming, but leaves embeddings, legacy completions, multi-sample $n > 1$, and logprobs outside the current scope.

The service installs the same best-runtime policy as the benchmark path before constructing the model. `_set_best_env_defaults()` uses `setdefault`, so explicit environment overrides survive; it is called before `torch` and runtime imports and again inside `_init_runtime()` before model construction. In practice, that policy enables the Section 6 choices—GPU prefill MoE with bucketed GEMM, decode active-expert staging with $K{=}10$/limit-2 prefetch, async All-Reduce, fused attention prefuse, and the 12-thread decode OpenMP setting. The launch wrapper `scripts/run_openai_server.sh` mirrors the same posture with four torchrun ranks, routed experts on CPU, `PD_MODE=scheduler`, and `MAX_MODEL_LEN=4096` by default.

Initialization order matters because many feature gates are read while modules and the `Transformer` are constructed. The service therefore installs defaults before runtime imports, ensuring that decode-side gates such as fused prefuse, async All-Reduce, and cross-layer prefetch match the benchmark path. This keeps OpenAI long/long decode aligned with the historical scheduler measurements under both 4K and 64K `MAX_MODEL_LEN` settings.

The serving model is deliberately serialized. `DeepSeekHTTPServer` uses a threaded HTTP server, but chat completions acquire a process-level request lock, and `_init_runtime()` forces `max_batch_size=1`. Rank 0 broadcasts each request to worker ranks through a Gloo control group, and the workers execute or drain the same generation path to keep collectives synchronized. Health and model-list requests remain lightweight and do not acquire the generation lock; multi-request concurrency, continuous batching, and paged KV-cache management as used in modern serving stacks [14, 31, 11] are left outside the current scope.

Non-streaming responses include standard usage counts plus a nonstandard `deepseek_timings` object with prefill/decode time, token counts, and decode tok/s. Streaming responses emit an initial assistant-role chunk, token deltas, optional final usage/timing when `stream_options.include_usage` is set, and a final `[DONE]` sentinel. Tool calls are parsed and emitted only after generation finishes rather than as fine-grained tool-call deltas. If a client disconnects during streaming, rank 0 stops writing to the socket but continues draining the generator so all ranks finish the same distributed step sequence.

Together, the 64K validation path and the OpenAI service path define the operating envelope for the measurements that follow. Section 8 turns that envelope into headline results, adopted and rejected ablations, and correctness validation.

8 Evaluation

8.1 Experimental setup and measurement discipline

All headline measurements use the target four-RTX 2080 Ti server described in Section 2, the converted DeepSeek-V4-Flash W8A8 checkpoint, four distributed ranks, CPU-resident routed experts, and the scheduler or OpenAI service defaults described in Sections 6–7. Decode measurements use greedy generation paths so that timing changes are not confounded by sampling. CLI runs report `prefill time`, `decode time`, token counts, and decode tok/s from the generation loop; OpenAI runs report the same quantities through the nonstandard `deepseek_timings` field described in Section 7.

The numbers below intentionally separate timing from diagnostics. Profiler traces, kernel microbenchmarks, and CPU-MoE profiling runs explain bottlenecks, but they are not used as final throughput numbers because profiling inserts synchronization and print overhead. The main text reports only the curated measurements needed to compare scenarios and design choices.

8.2 End-to-end performance

Table 6 reports representative headline runs on the target machine. The rows are different scenarios rather than a single sweep: the 64K row validates the long-context execution path, the long-prompt row checks OpenAI service parity on the 2,148-prompt-token / 63-decode-token long/long workload, and the current-best row reports the 29-prompt-token / 127-decode-token short-decode prefetch setting that produces the headline 3.16 tok/s number. The 64K row returned the expected `OK` smoke content, and the current-best row uses the $K{=}10$/local-limit-2 cross-layer prefetch setting. Figure 6 is an optimization-history view: the first segment uses long/long decode, while the final segment uses the short-decode benchmark selected for the headline prefetch result.

Table 6: Representative end-to-end performance on the target 4×RTX 2080 Ti server.

Scenario	Prompt tok.	Decode tok.	Prefill time (s)	Prefill tok/s	Decode tok/s
Maximum validated context	65,536	2	257.45	≈255	–
Long prompt decode reference	2,148	63	7.85	–	2.66
Current decode best	29	127	–	–	3.16

Dashes denote metrics that are not the primary measurement for that scenario; the three current-best decode repeats were 3.135 / 3.168 / 3.170 tok/s.

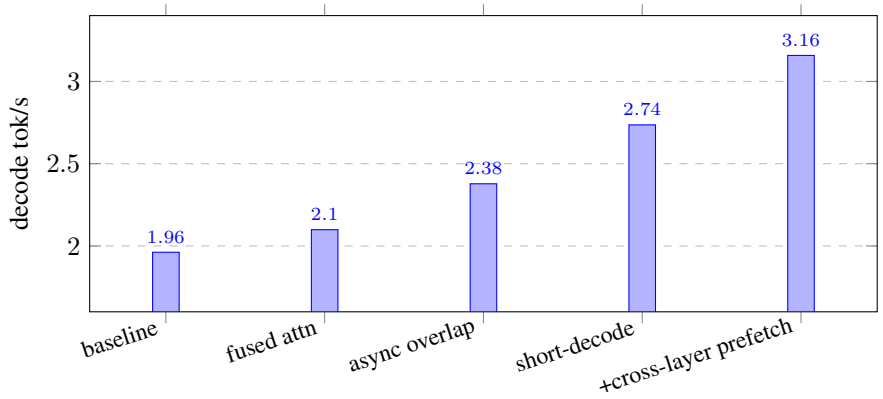

Figure 6: Optimization-history view of adopted decode improvements. The first three bars use the long/long benchmark with the CPU frequency governor pinned to performance; the last two use the short-decode benchmark selected for the headline prefetch result, so the figure shows progression rather than a single controlled sweep.

8.3 Adopted optimization ablations

Table 7 reports only A/B measurements with comparable scalar metrics. Enabling decisions without a clean single-row ablation—the W8A8 checkpoint format, CPU-resident routed experts, native CPU MoE runner, pinned arena, and logical PD scheduler—are discussed in Sections 4–7 rather than repeated as prose inside the table.

Table 7: Quantitative ablations for adopted optimizations on the target machine. Positive deltas improve decode tok/s; negative deltas reduce elapsed time.

Change	Scenario / metric	Baseline	After	Delta
CPU governor pinning	Long/long decode (tok/s)	≈1.63	1.85–1.93	+13–18%
Fused attention prefuse	Short/short decode (tok/s)	1.962	2.099	+7.0%
Fused attention prefuse	Long/long decode (tok/s)	1.750	1.828	+4.5%
Async CPU MoE / shared overlap	Long/long decode (tok/s)	2.057	2.378	+15.6%
Cross-layer MoE prefetch	Short-decode sweep (tok/s)	2.736	2.926	+6.9%
Bucketed MoE prefill GEMM	64K elapsed time (s)	289.52	259.87	-10.2%
Route grouping binding	2K prefill time (s)	13.39	7.99	-40.3%

8.4 Negative results

Rejected optimizations are part of the evaluation because they identify the bottlenecks that simple microbenchmarks miss. The negative results fall into four recurring failure modes:

- **Inner-kernel wins became slack.** PMADDUBSW roughly doubled the CPU inner-dot microbenchmark, but long/long decode moved only 2.393→2.396 tok/s after CPU/GPU overlap. cuBLAS IMMA was bit-exact on tested shapes, but long/long mean throughput fell by ≈6.6% once dequantization, allocation, and dispatcher overhead were included.
- **Small fusions were below system noise.** Fused inverse RoPE was bit-exact but only +1.7% mean, and folding the KV-cache write into the fused KV path measured -2.0% in A/B. Both removed local operations without shortening the dominant coordination path.
- **Speculation and centralization serialized distributed decode.** MTP reached 83.9% acceptance and length-2 verification was bit-identical, echoing the broader speculative-decoding and DeepSeek-MTP literature [27, 16, 1, 17, 4], but the scheduler path regressed because verify/draft synchronization serialized the step. External or rank-0 CPU MoE server variants similarly looked promising in microbenchmarks but failed on full distributed correctness, synchronization, or all-rank communication.
- **Unsupported datatype paths or semantic shortcuts were out of scope.** TileLang/native FP4 failed on `sm_75`, and route top-k trimming was rejected because it changes exact routing semantics.

The unifying signal is that on this hardware the GPU is often idle while Python dispatcher and CPU/MoE coordination dominate wall-clock time. Every optimization that does not reduce dispatcher count, staging cost, or distributed synchronization is either absorbed by run-to-run variance or hidden inside slack that already sits off the critical path.

8.5 Correctness and measurement hygiene

The project used four layers of correctness and measurement hygiene rather than relying on throughput alone:

1. **Conversion integrity.** Checkpoint conversion preserves shard names, tensor keys, index mapping, and conservative precision choices before runtime benchmarks are meaningful.
2. **Kernel equivalence.** CUDA kernels are tested against Python/PyTorch fallbacks when a local oracle exists, including fused attention prefuse, HCA post-processing, MoE single-token kernels, and INT8 GEMM variants.
3. **End-to-end smoke coverage.** Greedy smoke prompts catch content regressions, while 16K/64K prompts validate long-context execution; the 64K `OK` check is an execution smoke test, not a model-quality benchmark.
4. **Measurement isolation.** Final numbers exclude profiler traces and debug modes, and common false positives include stale servers, prompts equal to max model length, competing four-GPU runs, and environment variables read after import-time gates are initialized.

8.6 A bandwidth-bound decode ceiling

The profiling results help explain why decode improvements become harder after overlap and prefetch are in place. On this machine each GPU has a theoretical one-direction PCIe Gen3 x16 payload ceiling of about 15.75 GB/s (Table 1), yet diagnostic decode traces still spend most CUDA self time on pinned H2D copies rather than arithmetic. In the current-best decode profile, `Memcpy HtoD (Pinned -> Device)` accounts for 147.8 ms of 294.1 ms self CUDA time (50.3%), while NCCL All-Reduce adds another 76.9 ms (26.1%); the main INT8 GEMM kernel is only 22.4 ms (7.6%). In a more staging-heavy cross-layer sweep, pinned H2D rises to 206.2 ms of 283.6 ms self CUDA time (72.7%), with the same GEMM slice still near 22.5 ms.

These numbers are not a literal roofline model, because they aggregate many small transfers, collectives, and launch gaps rather than one steady streaming copy. They are still useful as an upper-bound diagnostic. Once fused prefuse has removed much of the dispatcher cost and CPU/GPU overlap has hidden part of the host-side routed-MoE interval, the remaining decode step is bounded primarily by how much routed-expert traffic and collective synchronization can be pulled forward on the PCIe fabric. That is why the overlap-only long/long path remains around 1.62–1.67 tok/s in the CPU-overlap diagnostic run, why the current best decode reaches 3.16 tok/s only on a short-decode case with favorable staging locality, and why local arithmetic wins such as PMADDUBSW or cuBLAS IMMA do not reliably surface in end-to-end throughput.

9 Discussion

The evaluation points to a system-level result rather than a single-kernel result: on this machine, the dominant constraints are placement, PCIe staging, and distributed coordination. The design choices that survived end-to-end measurement either make the model fit, remove repeated host-registration or padding waste, reduce dispatcher count at layer scale, or overlap expert movement with useful compute. Optimizations that only improve a local arithmetic kernel often disappear once CPU work, Python dispatch, NCCL synchronization, and PCIe transfers are in the same critical path.

9.1 Limitations

The implementation is intentionally specialized. It targets the converted `DeepSeek-V4-Flash-w8a8` checkpoint format, four tensor-parallel ranks, CPU-resident routed experts, and a single-request service path. It is not a general inference framework, and it does not yet implement continuous batching, multi-request scheduling, KV paging, or fine-grained streaming tool-call deltas. The OpenAI API surface is sufficient for chat-completion compatibility and timing export, but it is not a complete server-product implementation.

The design is also specialized to PCIe-only heterogeneous scaling. Moving from four to eight similar GPUs would not automatically double throughput: the routed-expert shard per rank would shrink, but the number of tensor-parallel participants and collectives would rise, and on this platform those collectives share the same PCIe/host-bridge fabric that decode staging already stresses. More GPUs would therefore help capacity and could improve per-rank expert locality, but only if the communication schedule, NUMA placement, and staging policy were re-tuned for the wider PCIe topology.

The validation boundary is also deliberate. The report verifies checkpoint integrity, kernel equivalence where local references exist, greedy smoke outputs, 16K/64K long-context execution, and exact routing semantics. It does not claim full perplexity parity with the upstream model, broad task-quality equivalence, or production SLOs under concurrent load. The 65,536-token `OK` run is a long-context execution smoke test; it demonstrates that the memory hierarchy and scheduler survive the path, not that a full benchmark suite has been passed.

The measurements are hardware-sensitive. The dual-socket Broadwell host, PCIe host-bridge and cross-socket topology, CPU governor, OpenMP placement, and competing four-GPU jobs all affect short-run throughput. Profiler traces are useful for explaining bottlenecks but are excluded from final throughput because they add synchronization and print overhead. A public release should include a checked-in benchmark harness, fixed prompts, environment manifests, repeat-count metadata, and raw logs.

9.2 What would help most

Working backward from the measurements, three hardware features would each unlock a different bottleneck. NVLink or another high-bandwidth GPU interconnect would reduce the cost of cross-rank synchronization and make expert staging less exposed to PCIe contention. A CPU with AVX-512, VNNI, or AMX would raise the ceiling for routed-expert execution and make more aggressive CPU-side scheduling viable. Native BF16/FP8/FP4 tensor-core support would reduce the need for custom `dp4a` kernels and simplify the precision policy for attention, shared experts, and routed experts.

At the software level, the most promising portability target is not a generic dense LLM but another large MoE whose routed experts dominate capacity and whose inference semantics can tolerate exact-routing execution with explicit staging. The core recipe — keep dense paths resident on GPU, hold routed experts in a pinned per-layer CPU arena, and schedule staging around useful compute — should transfer more naturally to DeepSeek-lineage or Qwen-style MoE systems than to architectures whose bottleneck is KV paging or dense-matrix throughput alone. What would need to change is the expert layout, gate proxy, and collective schedule, not the central placement argument.

Those extensions would make the system broader, but they do not change the main conclusion: on legacy heterogeneous servers, the first-order design problem is still where expert capacity lives and how its movement is scheduled. The related systems below explain which pieces of this design space are shared with prior work and where the legacy-hardware constraint changes the tradeoff.

10 Related Work

Heterogeneous CPU/GPU inference. ktransformers is the closest project-level analogue: it explicitly targets CPU/GPU heterogeneous LLM inference and exposes optimized kernels for consumer or edge-style deployments [13]. This report follows the same broad motivation—using host memory to make large models run outside datacenter-class

GPUs—but makes different engineering commitments: AVX2-only CPU support, Turing-specific INT8 GPU kernels, exact top-6 routing, and measured behavior on a four-RTX 2080 Ti PCIe server rather than an AMX-capable host.

Low-bit and INT8 quantization. llama.cpp and the ggml ecosystem make large models widely accessible through compact CPU/GPU inference and aggressive low-bit formats [10]. Prior LLM quantization work such as LLM.int8() and SmoothQuant shows that 8-bit matrix paths can preserve model behavior while improving efficiency [7, 28]. The runtime here deliberately stays in the INT8 W8A8 regime for heavy matrix paths and avoids sub-INT8 quantization for the model-facing claim: the goal is not maximum compression, but preserving routing semantics and avoiding additional sub-INT8 compression risk on legacy hardware.

Modern GPU serving stacks. vLLM introduced PagedAttention for efficient KV-cache memory management and high-throughput serving [14]; SGLang adds a frontend/runtime for structured language-model programs with KV reuse and structured-output optimizations [31]; DeepSpeed-FastGen studies high-throughput text generation through serving-oriented scheduling [11]. These systems target modern serving workloads, batching, paged KV, and current accelerator assumptions. This report instead targets a single-request legacy server where the dominant problem is not fleet throughput but making a 284B MoE fit and execute over PCIe without NVLink or native BF16/FP8/FP4 support.

Attention and kernel systems. FlashAttention established IO-aware exact attention kernels for modern accelerators [3], while FlashInfer provides customizable inference attention kernels for LLM serving [29]. TileLang provides a tiled programming model for AI kernels [26]. The target platform in this report cannot rely on those assumptions uniformly: Turing lacks native BF16/FP8/FP4 tensor-core paths, and the project's TileLang/native-FP4 experiments failed on `sm_75`. The response is narrower but more controlled: bespoke `dp4a` INT8 kernels and small fused decode-prefuse kernels for the exact shapes observed in this runtime.

Mixture-of-experts systems. Sparse MoE routing traces back to the sparsely-gated MoE layer [23] and large-scale conditional-computation systems such as GShard [15]. System work including DeepSpeed-MoE, Tutel, and MegaBlocks optimizes MoE training or inference at cluster scale [22, 12, 9], while DeepSeekMoE and DeepSeek-V2 are part of the architecture lineage leading to modern DeepSeek MoE models [2, 6]. The four-rank runtime here also inherits the tensor-parallel and collective-communication setting common in large Transformer systems [24, 20], but applies it under PCIe-only constraints. Fate studies edge MoE offloading with cross-layer gate prediction [8]; the prefetch policy here uses the same broad signal but is integrated into a four-rank DeepSeek-V4-Flash runtime with exact routing fallback. This report focuses on exact-routing inference of a fixed large MoE on legacy heterogeneous hardware, where the key systems problem is expert placement and PCIe staging rather than cluster-scale expert parallelism alone.

Speculative decoding and MTP. Speculative decoding and speculative sampling propose accelerating autoregressive generation by verifying draft tokens without changing the accepted distribution [27, 16, 1], and SpecInfer extends the idea to tree-based serving verification [17]. DeepSeek-V3 reports multi-token prediction in the DeepSeek model lineage [4]. The negative result here is therefore not about the promise of speculation itself: the local probe showed high acceptance. The bottleneck was integration with a heterogeneous distributed scheduler, where verify/draft synchronization erased the benefit under the default execution path.

11 Conclusion

This report shows that DeepSeek-V4-Flash can be served end-to-end on a four-GPU RTX 2080 Ti server by designing the runtime around the actual constraints of the machine rather than around modern accelerator assumptions. The system preserves exact top-6 MoE routing, avoids sub-INT8 quantization, validates the 65,536-token prompt path at roughly 255 prefill tok/s, and reaches a 3.16 decode tok/s mean on the current short-decode benchmark. These results are modest compared with frontier GPU clusters, but they are meaningful for the target question: whether inexpensive legacy hardware can run a current frontier-scale open MoE at all without changing the model's routing semantics.

The working recipe is a sequence of compatible constraints-driven decisions. Routed experts live in CPU memory because 88 GiB of aggregate VRAM cannot hold the full model. A pinned per-layer arena and GPU staging cache make that placement usable over PCIe. Turing-aware INT8 `dp4a` kernels cover the attention, shared-expert, dense, and grouped-MoE shapes that generic modern libraries do not target. The logical prefill/decode scheduler switches resource policy by phase, bucketed prefill GEMM reduces long-context padding waste, and cross-layer prefetch hides decode-time expert staging behind current-layer work. The negative results are part of the conclusion: optimizations that improved a local dot product or removed a small operation were not enough unless they shortened the real distributed critical path.

The broader implication is that frontier-model access is partly a systems problem. Legacy GPUs lack BF16/FP8/FP4 tensor cores, NVLink, and current kernel-library support, but they still provide useful memory bandwidth and integer throughput when paired with a runtime that respects their placement and interconnect limits. The same approach should transfer to other heterogeneous low-cost servers: start from capacity and movement, preserve model semantics, validate with end-to-end measurements, and only then decide which kernels deserve bespoke implementation.

References

[1] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[2] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, et al. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*, 2024.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022.

[4] DeepSeek-AI. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[5] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. Technical report, DeepSeek-AI, 2026. Technical report.

[6] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022.

[8] Zhiyuan Fang, Zicong Hong, Yuegui Huang, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Fate: Fast edge inference of mixture-of-experts models via cross-layer gate. *arXiv preprint arXiv:2502.12224*, 2025.

[9] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. *arXiv preprint arXiv:2211.15841*, 2022.

[10] ggml-org. llama.cpp: Llm inference in c/c++. GitHub repository, 2026. Accessed 2026-05-09.

[11] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[12] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *arXiv preprint arXiv:2206.03382*, 2022.

[13] KTransformers Contributors. Ktransformers: A flexible framework for experiencing cutting-edge llm inference/fine-tune optimizations. GitHub repository, 2026. Accessed 2026-05-09.

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

[15] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[16] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[17] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, et al. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[18] NVIDIA Corporation. *cuBLAS Documentation*. NVIDIA Corporation, 2026. Accessed 2026-05-09.

[19] NVIDIA Corporation. *CUDA C++ Programming Guide*. NVIDIA Corporation, 2026. Accessed 2026-05-09.

[20] NVIDIA Corporation. *NVIDIA Collective Communications Library Documentation*. NVIDIA Corporation, 2026. Accessed 2026-05-09.

[21] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

[22] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. *arXiv preprint arXiv:2201.05596*, 2022.

[23] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[24] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[25] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[26] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, et al. Tilelang: A composable tiled programming model for ai systems. *arXiv preprint arXiv:2504.17577*, 2025.

[27] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

[28] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[29] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. In *Proceedings of Machine Learning and Systems*, 2025.

[30] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, 2019.

[31] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2023.