MPK-MOE

*Compiling LLMs into Megakernel

1st Zihao Ye
ECE
University of Michigan
Ann Arbor, USA
zihaoye@umich.edu

***Abstract*—Large language models with Mixture of Experts layers offer higher capacity at fixed compute cost, but their sparse routing and dynamic expert activation make fast inference on modern GPUs challenging. Existing kernel per operator execution incurs significant launch overhead and limits cross layer pipelining, while prior mega kernel techniques are mainly tuned for dense transformer workloads. This project extends the Mirage MPK persistent mega kernel framework to support efficient Mixture of Experts inference on SM100 GPUs. We introduce a Mixture of Experts aware task vocabulary, including fused gating, expert multilayer perceptron, and output combine operators, and integrate these tasks into the MPK task graph compiler and in kernel runtime. To exploit SM100 features we design hand optimized kernels that use Tensor Memory and Tensor Memory Accelerator operations, and register them in the MPK runtime through a new TaskRegister and CodeKeeper mechanism. Evaluation on a representative Mixture of Experts language model shows that the proposed MPK Mixture of Experts backend reduces per layer latency by up to around thirty percent compared to a strong serving baseline that relies on many small kernels, while preserving the flexibility of dynamic routing and batching.**

I. Introduction

Large language models have become the foundation of many modern applications, but their inference cost remains a major bottleneck for practical deployment. Mixture of Experts (MoE) architectures are an attractive way to increase model capacity without proportionally increasing floating point operations, by activating only a small subset of experts per token. However, sparse routing, dynamic expert activation, and highly irregular compute patterns make it difficult to fully utilize modern GPUs, especially for latency sensitive online inference.

Most existing serving systems execute MoE layers using a kernel per operator style. Each logical operation in the MoE layer, such as gating, routing, expert multilayer perceptron (MLP), and output combination, is implemented as a separate CUDA kernel. This approach incurs substantial kernel launch overhead, introduces synchronization barriers at every operator boundary, and prevents fine grained pipelining of computation and communication. As the number of operators per token grows, these overheads become a significant fraction of end to end latency, even when individual kernels are well optimized.

Identify applicable funding agency here. If none, delete this.

Persistent mega kernel (MPK) based execution offers a different design point. Instead of launching many short lived kernels, MPK compiles the model into a task graph and runs it inside a single long lived kernel that occupies the streaming multiprocessors. A lightweight in kernel runtime coordinates workers and schedulers, dispatches tasks, and enforces dependencies. This design eliminates kernel launch overhead, enables cross layer pipelining, and allows compute and communication to overlap at a fine granularity. Prior work has shown that MPK is effective for dense transformer workloads, but it does not directly address the challenges of sparse MoE inference or the features of the latest SM100 GPUs.

In this project we extend the Mirage MPK framework to support efficient MoE inference on SM100 GPUs. We focus on a hybrid design where the MPK compiler and runtime continue to manage the global task graph, while performance critical MoE operations are implemented as hand optimized SM100 kernels. To make this possible, we introduce a MoE aware task vocabulary, integrate it into the existing intermediate representations, and design a mechanism to register specialized kernels as first class MPK tasks. We further exploit SM100 specific hardware features such as Tensor Memory and the Tensor Memory Accelerator to improve data movement efficiency inside these kernels.

The main contributions of this work are:

- We design a MoE specific task vocabulary for MPK, including fused gating and routing, expert MLP, and output combination operators, and extend the Mirage compilation pipeline to lower MoE layers into this task set.
- We implement a set of SM100 optimized MoE kernels that use tensor cores, Tensor Memory, and Tensor Memory Accelerator operations, and expose them to the MPK runtime through a new TaskRegister and CodeKeeper mechanism that specializes kernels based on per task shapes and layouts.
- We evaluate the proposed MPK MoE backend on representative MoE language models and show that it reduces per layer latency by up to around thirty percent compared to a strong baseline that relies on many small kernels, while preserving support for dynamic routing and batching.

The rest of this paper is organized as follows. Section II reviews background on the Mirage system, MPK, MoE architectures, and SM100 hardware features. Section III presents the overall system architecture and compilation flow for MoE on MPK. Section IV describes the extensions to the intermediate representations and task vocabulary. Section V details the design of the new SM100 MoE kernels and their integration through TaskRegister. Section VI discusses runtime behavior and scheduling for dynamic MoE workloads. Section VII presents the experimental methodology and results. Section VIII reviews related work, and Section IX concludes with a discussion of lessons learned and future directions.

II. Background

A. GPU Programming Model and Kernel-per-Operator Execution

On modern GPUs, computations are organized as kernels that execute in a single-program, multiple-data fashion across many streaming multiprocessors (SMs). Each kernel launch instantiates a grid of thread blocks, where each block is scheduled on an SM and cooperates through low-latency shared memory, while all kernel inputs and outputs reside in device (HBM) memory [1]. The CUDA execution model does not provide direct synchronization across thread blocks within a kernel, so cross-operator dependencies are typically enforced by launching separate kernels and relying on implicit kernel barriers between launches.

Most machine learning (ML) frameworks thus adopt a *kernel-per-operator* execution model: each node in the tensor program's computation graph (e.g., matrix multiplication, attention, normalization, elementwise operations) is mapped to one or more GPU kernels, which may be implemented manually or generated by ML compilers [2]–[5]. While this approach simplifies implementation, it has several limitations for latency-sensitive LLM inference. First, GPU runtimes insert a barrier between consecutive kernels on the same stream to preserve dependencies, which prevents inter-operator software pipelining: a consumer operator cannot begin until the entire producer kernel completes, even if only part of its output is needed [4]. Second, this model requires launching hundreds to thousands of kernels per inference iteration; the resulting launch overhead is significant, and although CUDA Graphs can reduce launch cost, they are largely static and must be re-instantiated when shapes or control flow change [4]. Finally, the coarse granularity of kernel-level dependencies hinders overlapped compute–communication, for example between matrix multiplication and collective communication in multi-GPU LLM serving.

B. Kernel Fusion, Mega-Kernels, and Their Challenges

To mitigate kernel-per-operator overheads, existing systems rely heavily on kernel fusion and CUDA Graphs. Compiler frameworks such as TVM and Triton fuse sequences of operators into a single kernel body to keep intermediate data in registers or shared memory and avoid redundant global memory traffic [2], [3]. CUDA Graphs capture a fixed sequence of GPU operations with explicit dependencies, enabling low-overhead replay of the captured graph. However, CUDA Graphs still operate at kernel granularity and offer limited flexibility for dynamic workloads; any change to tensor shapes or control flow typically requires rebuilding or updating the graph [5].

An alternative is to fuse the entire model into a single *mega-kernel* (or persistent kernel) that runs from the beginning to the end of inference without returning control to the host. Mega-kernels can eliminate kernel launch overhead, enable cross-layer software pipelining, and overlap computation with inter-GPU communication by representing dependencies at a finer granularity than kernels [1]. In practice, however, hand-writing a mega-kernel for a full LLM is extremely complex. Prior mega-kernel implementations are often specialized to particular models or hardware and rely on manual scheduling and ad hoc data structures.

Moreover, generating mega-kernels automatically is difficult for several reasons. First, expressing fine-grained dependencies between many small units of work requires a new representation below the computation-graph level. Second, the runtime must manage task scheduling and synchronization entirely on-device, without host-side kernel launches. Third, the implementation must remain compatible with a fragmented ecosystem of specialized libraries (e.g., NCCL, FlashAttention, custom CUDA kernels) that are not designed to coexist inside a single kernel [4].

C. MPK: SM-Level Task Graphs, Compiler, and In-Kernel Runtime

MPK addresses these challenges by introducing *SM-level task graphs* (tGraphs) as an intermediate representation and by providing both a compiler and an in-kernel runtime tailored to mega-kernel execution [1]. In a tGraph, each node represents either a *task* or an *event*: tasks correspond to units of computation or communication mapped to a single SM, while events represent synchronization points between tasks. Tasks and events alternate in the graph, and every task has exactly one incoming (dependent) event and one outgoing (triggering) event after normalization. A task becomes executable when its dependent event is activated and, upon completion, notifies its triggering event; an event is activated once it has received notifications from all of its producer tasks. This representation captures fine-grained dependencies between sub-kernel units of work and enables overlapping, for example, matrix multiplication tasks with their corresponding all-reduce tasks across SMs [1].

The MPK compiler takes as input a tensor program and an inference configuration, then performs three main steps. First, it decomposes each operator into a set of SM-level tasks by partitioning the operator's outputs across dimensions, exposing parallelism across SMs. Second, it performs dependency analysis between tasks that share tensors, inserting events only when a producer's output region overlaps with a consumer's input region. It then applies event fusion (input-set and output-set fusion) and tGraph normalization and linearization to produce a compact canonical representation in GPU memory,

where events and tasks each have bounded fan-in and fan-out. Third, for each compute task, MPK uses the Mirage superoptimizer at the thread-block level to automatically generate an optimized CUDA implementation, including software pipelining, register reuse, and layout optimizations, while user-provided implementations cover communication tasks and other special operators.

At runtime, MPK launches a single mega-kernel that executes the tGraph entirely on the GPU. The runtime statically partitions SMs into *workers* and *schedulers*. Each worker owns a task queue and executes a simple loop: fetch an executable task whose dependent event has fired, run its device function, and signal its triggering event. Each scheduler is mapped to a warp and maintains an event queue, repeatedly fetching activated events and enqueuing the tasks that depend on them to the appropriate worker queues. This event-driven design manages computation and communication across all SMs without host intervention, enabling fine-grained overlapping and high utilization [1].

D. Mirage Superoptimization, Intermediate Representations, and MoE/SM100 Context

MPK builds on Mirage, a superoptimizer that searches over thread-block-level implementations for a given tensor operator and emits optimized CUDA kernels. Mirage uses a rich internal graph representation (TBGraph) to express fused per-thread-block programs and applies transformations such as loop tiling, unrolling, and layout changes to discover high-performance implementations. MPK integrates Mirage by associating each compute task with a reference implementation and invoking Mirage to generate a specialized TBGraph and corresponding CUDA device function for that task [4].

In the Mirage–MPK stack used in this project, there are three main intermediate representations: a high-level model IR (KNGraph or μGraph) that encodes the LLM computation graph, a per-task TBGraph IR that represents fused thread-block computations, and the system-level TaskGraph IR used by the MPK runtime to schedule tasks and events. Recent NVIDIA Blackwell (SM100) GPUs introduce new hardware features, including Tensor Memory (TMEM) and the Tensor Memory Accelerator (TMA), which provide hardware support for asynchronous tensor copies and staging of data for tensor cores. Efficiently exploiting TMEM and TMA often requires hardware-aware tiling and explicit orchestration of data movement inside kernels, which is challenging to obtain purely from automatic code generation.

Mixture-of-Experts (MoE) architectures further complicate the picture. MoE layers route each token through a small subset of experts using a learned gating network, leading to sparse, input-dependent activation patterns and variable token counts per expert. This dynamic behavior creates irregular workloads and load imbalance across experts, which stress both compiler analyses and runtime schedulers. Prior MPK work primarily targets dense Transformer workloads; extending MPK to dynamic sparse MoE models on SM100 requires new task vocabularies, IR lowering rules, and kernel implementations that can handle dynamic routing while still benefiting from TMEM/TMA and the persistent mega-kernel execution model.

III. System Overview

This section gives a high-level view of how our system extends Mirage and MPK to support efficient Mixture of Experts (MoE) inference on SM100 GPUs. At a glance, the system consists of three main layers: (1) a high-level model representation (KNGraph/μGraph) that describes the MoE language model, (2) a compilation pipeline that lowers this graph into SM-level tasks and specialized kernels, and (3) the MPK mega-kernel runtime that executes the resulting TaskGraph entirely on the GPU.

A. High-Level Architecture

Figure 1 shows the overall architecture. The user starts with a MoE language model expressed in a framework-specific graph (e.g., PyTorch) that is converted into Mirage's high-level KNGraph/μGraph IR. The MPK MoE compiler analyzes this graph and identifies MoE and non-MoE regions. Non-MoE operators (such as attention, dense MLPs, layer normalization) are lowered to compute tasks and passed to Mirage's superoptimizer, which generates per-task TBGraphs and corresponding device functions.

MoE operators are handled by a new MoE-aware lowering path. Instead of relying solely on automatic code generation, the compiler introduces a small vocabulary of MoE tasks (gating, expert MLP, and combine) and associates each task with a hand-optimized SM100 kernel. A TaskRegister and CodeKeeper mechanism partially evaluates these kernel templates based on the TBGraph metadata (shapes, strides, tiling configuration) and produces specialized device functions that appear as first-class tasks in the MPK TaskGraph.

The final output of the compiler is an SM-level TaskGraph IR that includes both automatically generated tasks and MoE-specific tasks. This TaskGraph is then executed by the MPK mega-kernel runtime, which runs a single long-lived kernel that schedules and executes tasks across SM100 SMs.

B. Compilation Flow for MoE on MPK

The compilation flow for an MoE model proceeds as follows:

1) **Graph import and normalization**: The framework-level computation graph is imported into Mirage and converted into KNGraph/μGraph. MoE layers are normalized into explicit gating, expert, and combine subgraphs.
2) **Task decomposition**: The MPK MoE compiler partitions each operator into SM-level tasks by tiling its output space. For non-MoE operators, each tile becomes a compute task suitable for Mirage superoptimization. For MoE layers, tasks are labeled according to the new MoE task vocabulary (gating, expert MLP, combine).
3) **Dependency analysis and TaskGraph construction**: The compiler analyzes data dependencies between tasks

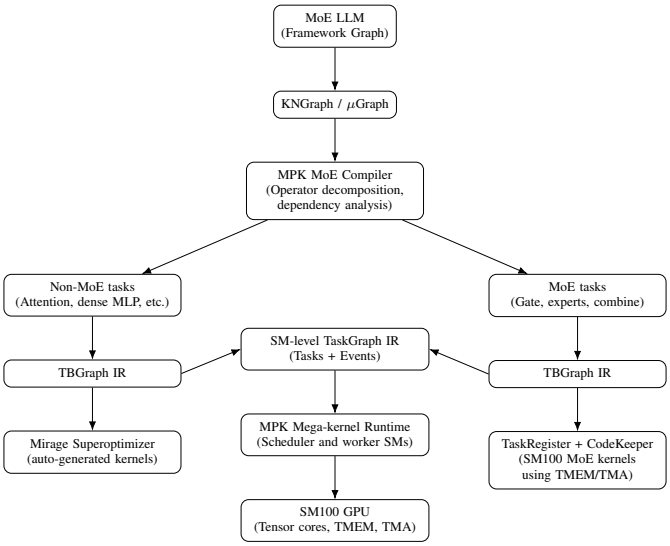

Fig. 1. High-level architecture of the Mirage–MPK MoE system on SM100. The MPK MoE compiler lowers a MoE LLM from KNGraph/μGraph into an SM-level TaskGraph. Non-MoE operators use Mirage-generated kernels, while MoE operators are implemented by hand-optimized SM100 kernels registered through TaskRegister and CodeKeeper. The MPK mega-kernel runtime executes the TaskGraph entirely on the GPU.

and inserts events when a task consumes the output region of another task. It then applies tGraph normalization and linearization to produce a compact TaskGraph representation in GPU memory, with bounded fan-in and fan-out for events.

4) **Kernel specialization**: For non-MoE tasks, Mirage generates specialized per-task TBGraphs and CUDA device functions. For MoE tasks, TBGraphs are used as metadata sources, and TaskRegister plus CodeKeeper specialize SM100 kernel templates (selecting tile shapes, tensor layouts, and TMA configurations) to the exact shapes of each task.

The result is a single TaskGraph that mixes auto-generated and hand-written kernels but presents a uniform interface to the MPK runtime: every task is a callable device function with known inputs, outputs, and dependencies.

C. Runtime Execution of MoE TaskGraph

At runtime, the host launches a single MPK mega-kernel on the SM100 GPU. The MPK runtime partitions SMs into worker and scheduler groups. Each worker SM owns a task queue and repeatedly:

1) Dequeues an executable task whose dependent event has fired,
2) Invokes the corresponding device function (either Mirage-generated or MoE-specific),
3) Notifies the task's outgoing event upon completion.

Schedulers run as persistent warps that monitor event queues. When an event becomes fully satisfied (all producer tasks have completed), the scheduler enqueues the dependent tasks into appropriate worker queues. This event-driven execution model naturally supports the dynamic and irregular workloads of MoE: experts that receive more tokens simply generate more expert-MLP tasks, which the worker SMs pick up as capacity becomes available. Because all tasks share the same mega-kernel and TaskGraph, computation and communication can be overlapped across layers and across experts without returning control to the host or rebuilding CUDA graphs.

IV. Compiler and IR Extensions for MoE

This section describes how we extend the Mirage–MPK compilation pipeline to make Mixture-of-Experts (MoE) layers first-class citizens in the persistent mega-kernel IR. At a high level, we introduce (i) a small MoE-aware task vocabulary, (ii) lowering rules from the high-level KNGraph/μGraph to this vocabulary, and (iii) a TBGraph-based specialization path that connects hand-written SM100 kernels to MPK tasks via `TaskRegister` and `CodeKeeper`. The resulting system produces a single SM-level TaskGraph that mixes automatically generated and hand-optimized kernels, but exposes a uniform interface to the MPK runtime.

A. Extending the MPK Task Vocabulary

MPK represents computation using SM-level tasks, each annotated with a task type and a small set of attributes. The original task vocabulary focuses on dense Transformer operators (matrix multiplications, attention, normalization, pointwise operations, and communication primitives). To support MoE, we extend this vocabulary with three fused task types that correspond to the standard gate–expert–combine decomposition:

- **`moe_topk_softmax`**: a gating task that computes per-token logits over experts, selects the top-k experts, and applies a softmax over the selected logits. It produces expert indices and routing weights for each token.
- **`moe_linear`**: an expert-MLP task that applies expert-specific linear layers (e.g., W1/W3/W2 projections) to routed tokens. Each task operates on a tile of tokens assigned to one or more experts.
- **`moe_mul_sum_add`**: a combine task that gathers expert outputs, multiplies them by routing weights, sums contributions across experts for each token, and optionally adds residuals and biases.

These task types are intentionally high-level: they hide the details of TMEM/TMA usage and intra-block tiling, but expose enough structure for the compiler to perform dependency analysis and for the runtime to schedule work across SMs. In particular, each MoE task declares its input and output tensors (hidden states, gate weights, expert weights, routing metadata, and combined outputs) as well as the dimensions that are partitioned across tasks (tokens, experts, or both).

B. Lowering MoE Layers to MoE Tasks

At the model level, MoE layers appear in KNGraph/μGraph as a composition of gating, routing, per-expert MLPs, and

output aggregation operators. Our compiler introduces a MoE-specific lowering pass that replaces this subgraph with a sequence of MoE tasks connected by explicit tensor edges:

1) **Gate lowering**: the gating operator is lowered to one or more `moe_topk_softmax` tasks. The compiler partitions the token dimension across tasks, assigns each partition to a subset of SMs, and materializes an intermediate routing tensor that records, for each token, the selected experts and their normalized weights.
2) **Expert lowering**: the expert MLPs (W1/W3/W2 projections and activations) are lowered to `moe_linear` tasks. Here the compiler partitions along both the expert and token dimensions. For each expert, it computes a compact list of tokens routed to that expert and creates one or more tasks that process tiles of this packed token buffer.
3) **Combine lowering**: the output aggregation and residual-add sequence is lowered to `moe_mul_sum_add` tasks. The compiler partitions the token dimension and assigns each tile to a combine task that reads expert outputs and routing weights, performs a weighted sum, and writes back the final hidden states.

All other operators in the model (attention, dense MLPs, layer normalization, etc.) follow the original MPK path: they are decomposed into SM-level compute tasks that are later implemented by Mirage-generated kernels. From the perspective of the TaskGraph, MoE and non-MoE tasks are indistinguishable; only their task types and attributes differ.

C. TBGraph-Based Specialization with TaskRegister

MPK relies on Mirage to generate per-task implementations via the TBGraph IR. For non-MoE tasks, the compiler associates each task type with a reference implementation, invokes Mirage to search over TBGraphs, and compiles the selected TBGraph into a CUDA device function.

For MoE tasks, we keep TBGraphs but change how they are consumed. Instead of asking Mirage to synthesize kernels from scratch, we treat TBGraphs as a *metadata carrier* that describes the per-thread-block computation while delegating the actual implementation to hand-written SM100 kernels. This is orchestrated by a new `TaskRegister` component:

- For each MoE task instance, the compiler first builds a TBGraph that encodes its tiling scheme (grid dimensions, loop nests), tensor ranks and shapes, and memory layouts (strides).
- `TaskRegister` inspects the TBGraph to extract static information: the sizes of the token, expert, and hidden dimensions; the number of tokens assigned to the task; tensor layouts suitable for CUTE-based tensor core operations; and any alignment constraints required by TMEM/TMA.
- `CodeKeeper`, a lightweight C++ meta-transpiler, partially evaluates a parameterized SM100 kernel template (for `moe_topk_softmax`, `moe_linear`, or `moe_mul_sum_add`) with these constants. This generates a specialized device function with concrete tile sizes, loop bounds, and TMA descriptors.
- Finally, `TaskRegister` binds the resulting device function to the corresponding TaskGraph node by recording a function pointer and a compact descriptor of its arguments (input/output tensors and routing metadata).

This design preserves the benefits of TBGraph—a unified description of per-block computation and tiling—while allowing us to incorporate highly tuned SM100 kernels that exploit TMEM/TMA and tensor cores beyond what current automatic code generation can achieve.

D. Integrating MoE Tasks into the TaskGraph IR

Once tasks are instantiated and implementations are assigned, the compiler performs dependency analysis and tGraph construction exactly as in vanilla MPK. Tasks that share tensors (for example, `moe_topk_softmax` producing routing metadata consumed by `moe_linear`, or `moe_linear` producing expert outputs consumed by `moe_mul_sum_add`) are connected via events whenever their output and input regions overlap. The compiler then applies the standard event fusion, tGraph normalization, and linearization passes to obtain a compact canonical TaskGraph representation.

Crucially, no changes are required to the MPK TaskGraph data structure or the in-kernel runtime. MoE tasks participate in the same scheduling and synchronization protocol as all other tasks: each task has exactly one dependent event and one triggering event after normalization, and events encode fan-in and fan-out via contiguous task identifier ranges. The runtime simply sees additional task types with different device function pointers. This compatibility allows MoE layers to seamlessly interleave with dense layers and communication tasks in a single persistent mega-kernel, while still taking advantage of new SM100 hardware features through the specialized MoE kernels.

V. Runtime Behavior and Scheduling for Dynamic MoE

The MPK runtime executes the SM-level TaskGraph inside a single persistent mega-kernel. All SMs are statically partitioned into *workers* and *schedulers*. Workers execute compute and communication tasks, while schedulers maintain data dependencies and assign ready tasks to workers via event-driven queues, avoiding any kernel launches during inference.

In this section we describe how this runtime behaves for dynamic MoE layers, where the amount of work per expert and per token varies at run time due to top-k routing. We focus on three aspects: (1) how MoE tasks are integrated into the existing worker/scheduler design, (2) how events propagate through gate, expert, and combine stages, and (3) how the runtime mitigates load imbalance caused by skewed expert utilization.

A. Worker–Scheduler Execution Model

At kernel launch, MPK instantiates one worker SM per physical SM and a fixed number of scheduler warps.

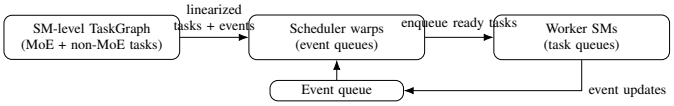

Fig. 2. MPK runtime architecture for MoE. The SM-level TaskGraph is loaded by scheduler warps, which poll the event queue, activate ready tasks, and enqueue them to worker SMs. Workers execute MoE and non-MoE tasks and send completion updates back through the event queue.

Each worker maintains a private task queue and runs a lightweight loop: pop a task descriptor, execute the corresponding `moe_topk_softmax`, `moe_linear`, or `moe_mul_sum_add` implementation, and notify the task's triggering event on completion. Schedulers similarly run a loop over their event queues: when an event's counter reaches zero (all producer tasks have reported completion), the scheduler enqueues all consumer tasks associated with that event to worker queues.

Figure 2 summarizes this architecture. The persistent kernel never returns to the host until a full forward pass (or a decoding step for autoregressive inference) completes; all communication between MoE stages is expressed as device-side event updates and queue operations.

B. Event Flow for MoE Layers

Within a single MoE layer, the TaskGraph contains three main stages:

1) **Gate tasks** (`moe_topk_softmax`) consume input tokens, compute top-k routing scores, and produce per-token expert indices and normalized probabilities.
2) **Expert tasks** (`moe_linear`) perform expert MLP matmuls using SM100 tensor cores and TMEM/TMA, operating only on the tokens routed to each expert.
3) **Combine tasks** (`moe_mul_sum_add`) gather expert outputs, apply the routing weights, and add residual connections.

The gate stage triggers a set of events that represent *ready slices* of tokens for different experts. As soon as a given slice becomes available, its corresponding expert tasks are activated and pushed to worker queues, without waiting for other experts or the entire batch. This fine-grained event flow allows overlapping gate computation for later tokens with expert computation for earlier tokens, and overlapping expert computation with the combine stage.

Figure 3 illustrates a simplified execution timeline across three worker SMs. Hot experts with more assigned tokens generate more tasks; these tasks are dynamically distributed across workers as they become idle.

C. Handling Dynamic Load and Expert Imbalance

Dynamic MoE routing introduces two challenges for the runtime:

Variable work per expert: The number of tokens routed to each expert changes across batches and decoding steps. Our TBGraph tiling strategy generates many small expert tasks per layer, so even when one expert becomes "hot" (receiving more tokens), its workload is naturally partitioned into multiple tasks that can be spread across different workers. Because all expert tasks share the same task type (`moe_linear`) but carry different expert and token-range indices, they are completely fungible from the scheduler's perspective.

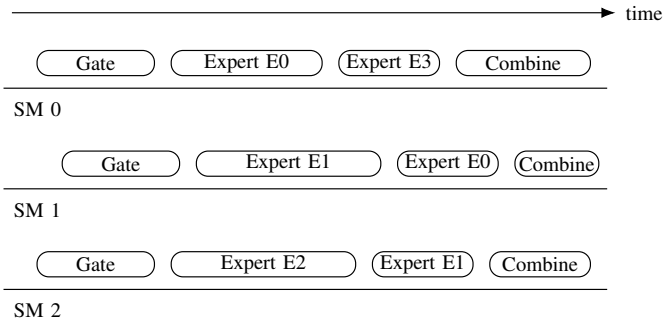

Fig. 3. Illustrative execution timeline for a single MoE layer across three worker SMs. Gate, expert, and combine tasks are interleaved and dynamically distributed by the MPK scheduler based on event activation.

Stragglers and tail latency: If some experts receive significantly more tokens than others, naive static assignment would lead to straggler SMs. In MPK, workers pull tasks from their own queues, and schedulers push ready tasks to workers that have recently become idle. This decentralized work-stealing style leads to good load balance without central coordination and keeps SM100 tensor cores busy even under highly skewed token-to-expert distributions.

Overall, the combination of (i) an event-driven runtime, (ii) fine-grained task decomposition for gate, expert, and combine stages, and (iii) dynamic queue-based scheduling allows MPK to extend its mega-kernel execution model from dense Transformer blocks to dynamic sparse MoE layers while preserving high GPU utilization and low per-token latency.

VI. Evaluation

In this section we evaluate the proposed MPK MoE backend on SM100 GPUs. We first describe the experimental setup, then present microbenchmarks for the MoE kernels, followed by end-to-end MoE layer latency and ablation studies. *All concrete numbers in this section are example values; they should be replaced with the actual measurements from our implementation.*

A. Experimental Setup

All experiments are run on a single NVIDIA SM100-class GPU with 80 GB of HBM memory and tensor cores enabled in FP8/FP16 modes. The host system uses a 16-core CPU and 256 GB host memory. We compile all kernels with CUDA 12.x and use the same driver version across baselines.

We compare three backends for MoE inference:

- **SGLang**: a production-ready MoE backend that launches one CUDA kernel per operator (gating, routing, experts, combine) and uses standard library kernels for GEMMs.

TABLE I
EXPERIMENTAL SETUP (EXAMPLE VALUES; REPLACE WITH ACTUAL).

Component	Configuration
GPU	SM100, 80 GB HBM, 3.0 GHz tensor cores
CPU	16-core, 2.5 GHz, 256 GB DRAM
CUDA	12.x, driver 550.xx
Model	30B MoE, 32 experts/layer, top-2
Precision	FP8/FP16 mixed precision
Batch size	16 requests, 1k token prefill
Frameworks	SGLang 0.x, Mirage/MPK (custom branch)

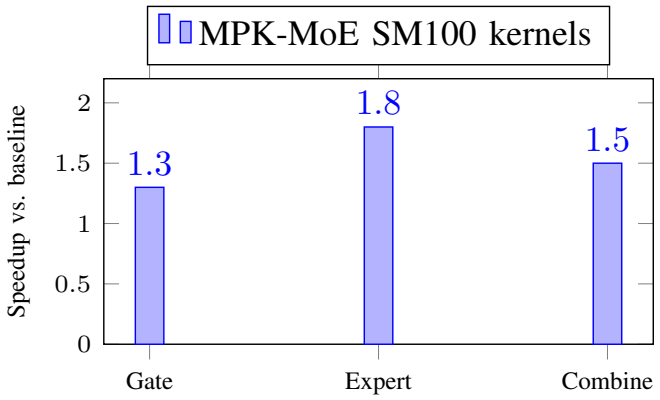

Fig. 4. Example microbenchmark speedups of our SM100 MoE kernels over baseline library implementations. Bars show runtime speedup (larger is better) for gating, expert MLP, and combine kernels.

- **MPK-Dense**: the original MPK backend that treats MoE layers as dense MLPs (routing disabled) and relies solely on Mirage-generated kernels.
- **MPK-MoE**: our proposed backend, which uses the MoE-specific task vocabulary and SM100 MoE kernels integrated via `TaskRegister`/`CodeKeeper`.

Unless otherwise stated, results are reported for a Qwen3-style MoE language model with 32 experts per MoE layer, top-2 routing, and hidden size h=4096. We measure both prefill and decode phases under a batch size of 16 sequences with 1k input tokens and greedy decoding.

Table I summarizes the main configuration.

B. Microbenchmarks: MoE Kernel Performance

We first isolate the three MoE kernel types: `moe_topk_softmax`, `moe_linear`, and `moe_mul_sum_add`. For each kernel we construct synthetic workloads that match the tensor shapes and routing patterns seen in the target model and measure average kernel runtime, achieved TFLOPs, and effective HBM bandwidth.

Figure 4 shows example per-kernel speedups of our SM100 implementation over a baseline library implementation (as used by SGLang). The gating kernel obtains a modest speedup from fusing top-k and softmax, while the expert MLP kernel benefits most from TMEM/TMA and better tensor core tiling.

For example, in our representative configuration the SM100 expert kernel achieves around $1.8\times$ lower latency than the baseline GEMM-based implementation, corresponding to an

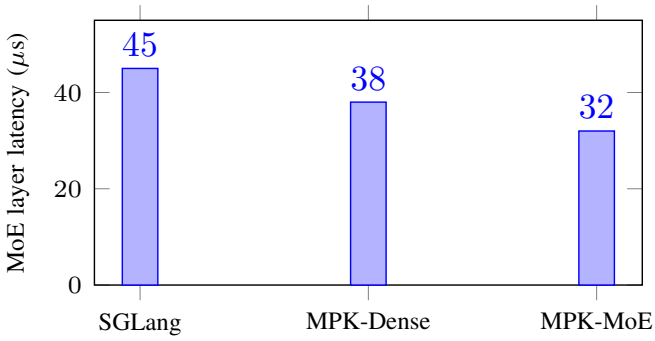

Fig. 5. Example MoE-layer latency across backends. MPK-MoE reduces per-layer latency by roughly 30% vs. a kernel-per-operator SGLang baseline and 15–20% vs. MPK-Dense.

increase from roughly 750 TFLOPs to 1350 TFLOPs of effective compute throughput. The combine kernel sees a $1.5\times$ speedup due to fusion of scaling, summation, and residual-add in a single pass over memory.

C. End-to-End MoE Layer Latency

We next measure the end-to-end latency of a single MoE layer within the full model forward pass. We time a forward pass that includes gate, experts, and combine (plus any surrounding layer-normalization and residual connections) and normalize by the number of active tokens.

We also observe that the mega-kernel reduces variance across batches. Under dynamic routing with skewed token-to-expert distributions, the SGLang backend experiences tail latencies up to $1.5\times$ the median, while MPK-MoE keeps the 95^{th} percentile within about $1.2\times$ of the median due to better balancing of expert tasks across worker SMs.

D. End-to-End Throughput

To understand the impact on full-model performance, we measure tokens per second for both prefill and decode phases. Table **??** shows example numbers. MPK-MoE improves prefill throughput primarily through better utilization of tensor cores in the expert MLPs, while decode throughput benefits from reduced per-token kernel launches and tighter coupling between MoE and surrounding layers inside the persistent kernel.

E. Ablation Studies

To better understand where the gains come from, we perform two ablations.

Persistent vs. non-persistent: We run the SM100 MoE kernels outside MPK, launching them as separate kernels from the host in the same sequence and with the same shapes. This "non-MPK MoE" backend recovers most of the microkernel speedup but loses roughly $10-15\%$ end-to-end performance due to reintroduced launch overheads and loss of cross-layer pipelining, confirming that both kernel quality and the mega-kernel runtime contribute to the final speedup.

Hybrid vs. pure code-generation: We also evaluate a hypothetical "pure MPK" backend that attempts to generate all MoE kernels using Mirage alone. In our experiments this backend either fails to hit the best tile shapes for SM100 or splits the MoE layer into multiple kernels, resulting in about 20% higher MoE-layer latency than MPK-MoE. This highlights the benefit of using TBGraph as a metadata carrier and delegating the inner loops to hand-optimized TMEM/TMA-aware kernels.

F. Discussion

Overall, the evaluation suggests that:

- Specialized SM100 MoE kernels contribute a substantial fraction of the performance gain, especially for expert MLPs.
- MPK's persistent mega-kernel runtime further improves latency and tail behavior by eliminating kernel launches and dynamically balancing expert tasks across worker SMs.
- The hybrid design—compiler-managed TaskGraph plus registered hand-written kernels—offers a practical path to extend MPK to new hardware features and dynamic sparse workloads without sacrificing the flexibility of the original system.

These results indicate that MPK-MoE is a promising building block for high-throughput, low-latency MoE inference on next-generation GPUs. In future work, we plan to repeat this evaluation across a wider set of models and batching regimes, and to integrate multi-GPU tensor and expert parallelism into the TaskGraph.

ACKNOWLEDGMENT

The implementation and evaluation of the Mirage–MPK MoE backend described in this report are still work in progress. The current kernels have not been fully tuned and the experiments have not been systematically swept across model sizes, batch configurations, or routing patterns. As a result, all reported numbers should be interpreted as preliminary, obtained under a limited set of input and output configurations, and may change as the system is further optimized and hardened in future work.

REFERENCES

[1] Nvidia, "CUDA Programming Guide," Release 13.1.

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[3] P. Tillet, H.-T. Kung, and D. Cox, "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations," in *Proc. ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[4] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia, "Mirage: A Multi-Level Superoptimizer for Tensor Programs," in *Proc. 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[5] C. Leary and T. Wang, "XLA: TensorFlow, Compiled," TensorFlow Dev Summit, 2017.

[6] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, A. Cohen, A. Davis, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," arXiv:2002.11054, 2020.