

vLLM Models

vLLM has long pursued the goal of building a unified, fast inference engine across diverse models and hardware. We remain committed to this goal, but we have come to believe that vLLM's current model implementation principles need to be revisited in order to improve both performance and development velocity. Two lessons stand out:

- We were overly reluctant to modify model code directly for op fusion and kernel dispatching. Instead, we accumulated abstractions and compiler passes that ultimately made the code opaque and hard to optimize.
- We pursued the unrealistic ideal of a single model definition that works well on every hardware. In practice, each hardware backend benefits from its own model implementation, since different hardware favors different fusion strategies.
- AI coding agents have made manual op fusion and kernel writing much easier, and the agents work best with raw model code.

To address these issues, we are making three fundamental changes to how vLLM implements and optimizes models:

1. Removing full-graph `torch.compile` requirement
2. Separation of model code across hardware backends
3. A clear model interface

1. Removing full-graph `torch.compile` requirement

vLLM enables full-graph `torch.compile` by default and has relied on it for model-specific optimizations. Two problems have emerged:

- **Performance.** Achieving speed-of-light performance on target models on the latest hardware within a tight timeline requires manual optimization. Compiler-driven optimization alone has not been sufficient, and timely performance gains are critical to vLLM's adoption.
- **Developer experience.** `torch.compile` increases the vLLM startup time and imposes the opacity and constraints inherent to any compiler, both of which slow iteration on model code.

We are therefore **removing vLLM's reliance on full-graph `torch.compile`** and restructuring model definitions accordingly. The migration must **preserve the current user-facing UX and introduce no performance regressions**.

Note that we will keep using `torch.compile` to fuse adjacent ops locally, whenever we find it to be useful. The removal is only about full-graph compilation.

1-1. Compiler fusion → manual fusion

Historically, vLLM kept model code minimal (pure PyTorch ops) and relied on compiler passes to fuse them. We are reversing this convention: fusion should be expressed directly in model code. Concretely, model code should look like:

Python

```
class Attention(nn.Module):
    def __init__(self, ...):
        ...
        self.o_proj = RowParallelLinear(..., reduce_results=False)

    def forward(self, ...):
        ...
        hidden_states = self.o_proj(attn_out) # pre-allreduce
        return hidden_states

class Layer(nn.Module):
    def __init__(self, ...):
        self.attn = Attention(...)
        self.rms_norm = RMSNorm(...)
        self.mlp = MLP(...)

    def forward(self, hidden_states):
        ...
        hidden_states = self.attn(hidden_states) # pre-allreduce
        # Call the fused kernel here.
        residual, quantized_inputs = add_rms_norm_quant(
            hidden_states,
            residual,
            self.rms_norm.weights,
            self.rms_norm.eps,
            # act_quant_kwargs depends on the quantization scheme
            self.mlp.act_quant_kwargs,
            do_allreduce=self.tp_size > 1,
        )
        hidden_states = self.mlp(quantized_inputs) # pre-allreduce
```

```
return hidden_states
```

Here, `add_rms_norm_quant` is the call site for the fused kernel. We will introduce such "big fused ops" directly in model code wherever manual fusion is beneficial. This way, we can replace all fusions in vLLM's compiler passes and even further improve the performance in complex cases such as DeepSeek V3.2 that the current compiler optimizations do not handle well.

1-2. CustomOp / vLLM IR

With manual fusion in place — and the hardware-backend code separation discussed below — there is little reason to retain the `CustomOp` and vLLM IR abstractions. **We will remove both**, and each op will define its own dispatching logic where necessary (for example, when different kernels must be launched on Hopper vs. Blackwell) possibly with some shared utility methods. For example, the code may look like:

Python

```
def add_rms_norm_quant(
    hidden_states: torch.Tensor,
    residual: torch.Tensor,
    rms_weights: torch.Tensor,
    rms_eps: float,
    mlp_act_quant_kwargs: dict[str, Any],
    do_allreduce: bool,
) -> tuple[torch.Tensor, Any]:
    if not do_allreduce:
        # No comm. Launch a simple fused kernel.
        return ...

    device_info = get_device_info()
    if device_info.name in ("GB200", "GB300"):
        # Launch FlashInfer kernel
        ...
    elif device_info.name in ("H100", "H200"):
        # Launch some other kernels
        ...
```

1-3. Piecewise CUDA graph

To preserve the piecewise CUDA graph feature without relying on `torch.compile`, we can adopt the "breakable CUDA graph" approach from [sgl-project/sglang#19102](https://github.com/sgl-project/sglang/pull/19102). We expect this to match the performance of the current compiler-based approach without imposing many constraints on model code.

1-4. Hacks for compiler compatibility

vLLM has accumulated several hacks to keep model code compatible with `torch.compile`. One example is `forward_context`, a global variable used to bypass the compiler's argument checking. These workarounds have hurt readability and introduced rough edges — model inputs are scattered across multiple places, and some global objects are not freed when the vLLM engine is deleted. Removing these hacks will improve the project's maintainability.

2. Hardware separation and model isolation

vLLM currently uses a single model definition for all in-tree hardware backends and relies on `CustomOp` and `torch.compile` to apply different fusions and dispatch kernels. This has made the code unnecessarily complex and fragile: a change made for one backend can easily break or regress another.

Going forward, **we will separate model code by hardware vendor**. Each model definition can then evolve independently for its target hardware without risk of breaking the others. Just as importantly, each vendor can design fusions around the kernels available to them, rather than conforming to a shared abstraction that fits no one perfectly.

Concretely, we expect the model code to be organized as follows:

```
None
models/
  deepseek_v4/
    nvidia/
      model.py
      kernels/
        fused_q_kv_norm.py
      tests/
        test_model.py
        test_fused_q_kv_norm.py
    amd/
      model.py
  deepseek_v3_2/
```

```

nvidia/
  model.py
  kernels/
    fused_q.py
  tests/
    test_model.py
    test_fused_q.py
amd/
  ...
tpu/
torch_compile_models/
  ...

```

Here, `kernels` contains model-specific JIT kernels (and maybe AoT kernels depending on our build structure); commonly used kernels remain in their existing locations. Each model directory also includes unit tests, so we can easily verify basic correctness of the model and its kernels. The code organization will help aggressively leverage AI to optimize models, as well as cleanly deprecate old models from the vLLM codebase.

3. Model interface

vLLM relies on many implicit assumptions about models and does not cleanly separate model-agnostic logic from model-dependent logic. This has produced a poor developer experience for users bringing custom in-house models. To address this, we are introducing explicit interfaces for both model definition and configuration.

First, we will define vLLM's own `ModelConfig` to hold all the information vLLM needs about a model. It is essentially a union of the typical HF config and vLLM's model protocol. The idea is similar to [vllm-project/vllm#24384](https://github.com/vllm-project/vllm/issues/24384).

```

Python
@dataclass
class ModelConfig(ABC):
    num_layers: int
    max_model_len: int
    vocab_size: int
    hidden_states_dtype: torch.dtype

```

```

num_kv_heads: int
num_query_heads: int
is_moe: bool
num_shared_experts: int | None
num_routed_experts: int | None
experts_per_tok: int | None
supports_mm: bool
supports_pp: bool
supports_lora: bool
...
model_kwargs: dict[str, Any] # e.g., "rms_norm_eps": 1e-05

```

This brings three benefits:

1. A single source of truth for all model information.
2. We currently spawn a subprocess to read certain fields from the model definition; this config eliminates that overhead, which is often dominated by import time.
3. Model developers no longer need to port their models into HF format to use vLLM. In an extreme case, they can simply hardcode the config with values they already know.

Second, we will introduce vLLM's own model interface to make all contracts explicit. It should look like:

Python

```

class Model(ABC):
    def __init__(
        self,
        config: ModelConfig,
        parallel_state: ParallelState,
        **kwargs,
    ) -> None:
        # Allows the model to manage its own state.
        raise NotImplementedError

    @abstractmethod
    def load_weights(self, use_dummy: bool) -> None:
        ...

```

```

@abstractmethod
def init_kv_cache(self) -> None:
    # KV cache-related API (Needs more thought).
    ...

def optimize_and_warmup(self) -> None:
    return None

def add_requests(self, req_indices: list[int], request_data:
list[Any]) -> None:
    # For initializing model-specific state (if any) when a
    request first arrives.
    return None

@abstractmethod
def prepare_inputs(self, input_batch: InputBatch) -> dict[str,
Any]:
    # Prepare model inputs for a step. May include building
    attention metadata.
    ...

@abstractmethod
def forward(self, **kwargs) -> torch.Tensor:
    # Run the model and return the hidden states.
    ...

def compute_logits(self, hidden_states: torch.Tensor) ->
torch.Tensor:
    raise NotImplementedError

    # More optional APIs for LoRA, spec decoding, and multimodal
    inputs.
    ...

```

The key idea is to let the model manage its own state when needed. For example, Qwen-VL models can manage their pre-computed M-RoPE values within this framework without any

changes to the upper-level model runner. Similarly, `prepare_inputs` lets each model define its own inputs. With this design, the model runner handles only model-agnostic work — bookkeeping for continuous batching, sampling, CUDA graphs, and parts of model parallelism — while all model-dependent work lives in the model layer. Part of this idea was already implemented as `ModelState` in [model runner V2](#).

Any model definition that implements both the `Model` and `ModelConfig` interfaces should be fully supported by vLLM, with no additional requirements.

Torch Compile Migration Plan

May 13th

- Initial implementation of Breakable CUDA graph (by Jiangyun Zhu)
- Initial version of Hardware separation (led by woosuk@inferact.ai) for NVIDIA, AMD + XPU, and out-of-tree hardware
 - And share it with community
- Share this document with sig-torch-compile community on Wednesday May 13 (After confirming with Tyler that we're good)
- CI to make sure the perf is not regressed?
- Enumerate all fusion passes and concretize how to implement the manual fusion
 - [AR +] RMS Norm + Quant (FP8, NVFP4)
 - Attention + Output Quant (FP8 per-tensor, NVFP4)
 - SiLU-Mul + Quant (FP8, NVFP4)
 - Q/K RMSNorm + RoPE
 - RoPE + KV Update
 - MiniMax forward_qk allreduce+norm

May 20th

- Stabilize Breakable CUDA graph
- Code separation for DeepSeek V4 & DeepSeek V3.2

May 22nd

- Remove torch compile requirement for DeepSeek V4 & DeepSeek V3.2 (GLM 5.1)
 - PW CUDA graph → Breakable CUDA graph
 - Compiler fusion → Manual fusion (Ensure no perf regression)

May 27th

- Apply manual fusion for more key models (Qwen, Kimi, Minimax, Llama, NemotronH, Gemma)

Mid June

- Apply manual fusion for all models

End of June

- Remove full-graph `torch.compile` integration in vLLM (Compiler passes, `forward_context`)
- Remove `CustomOp` & vLLM IR