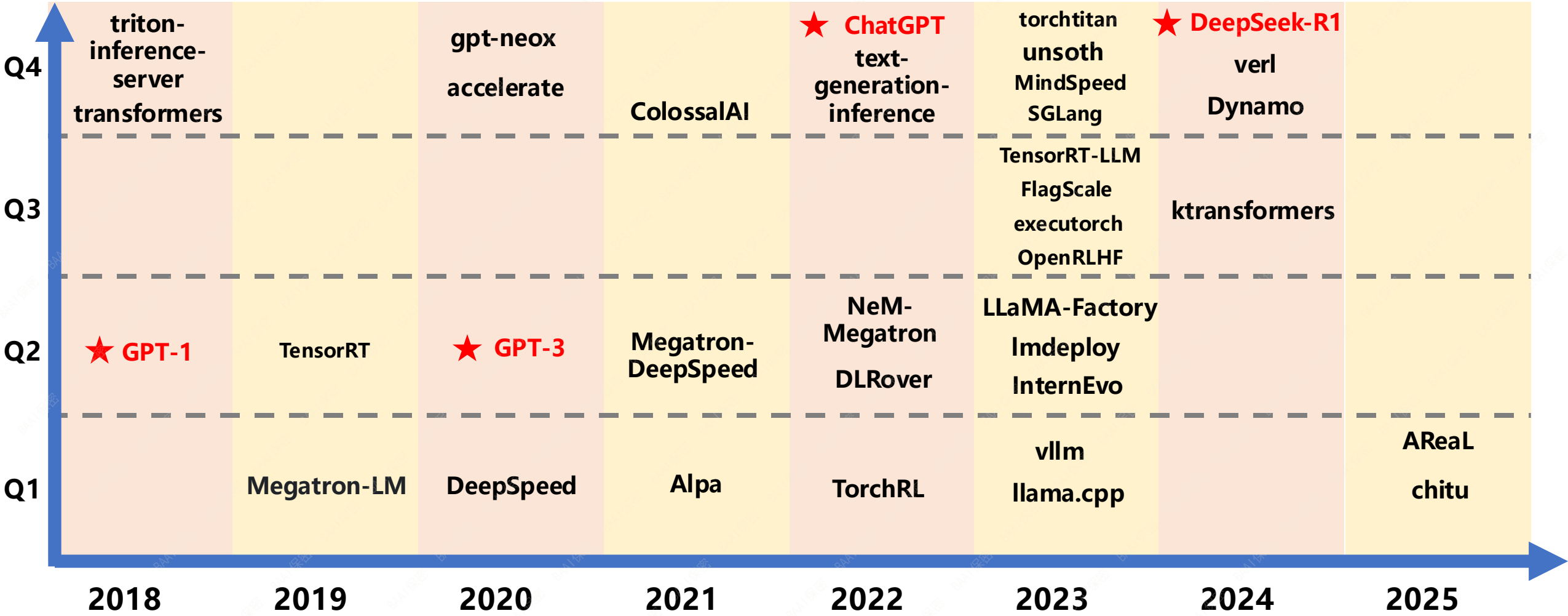
北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

FlagScale多后端管理与多硬件适配机制

大模型框架的开源发展趋势

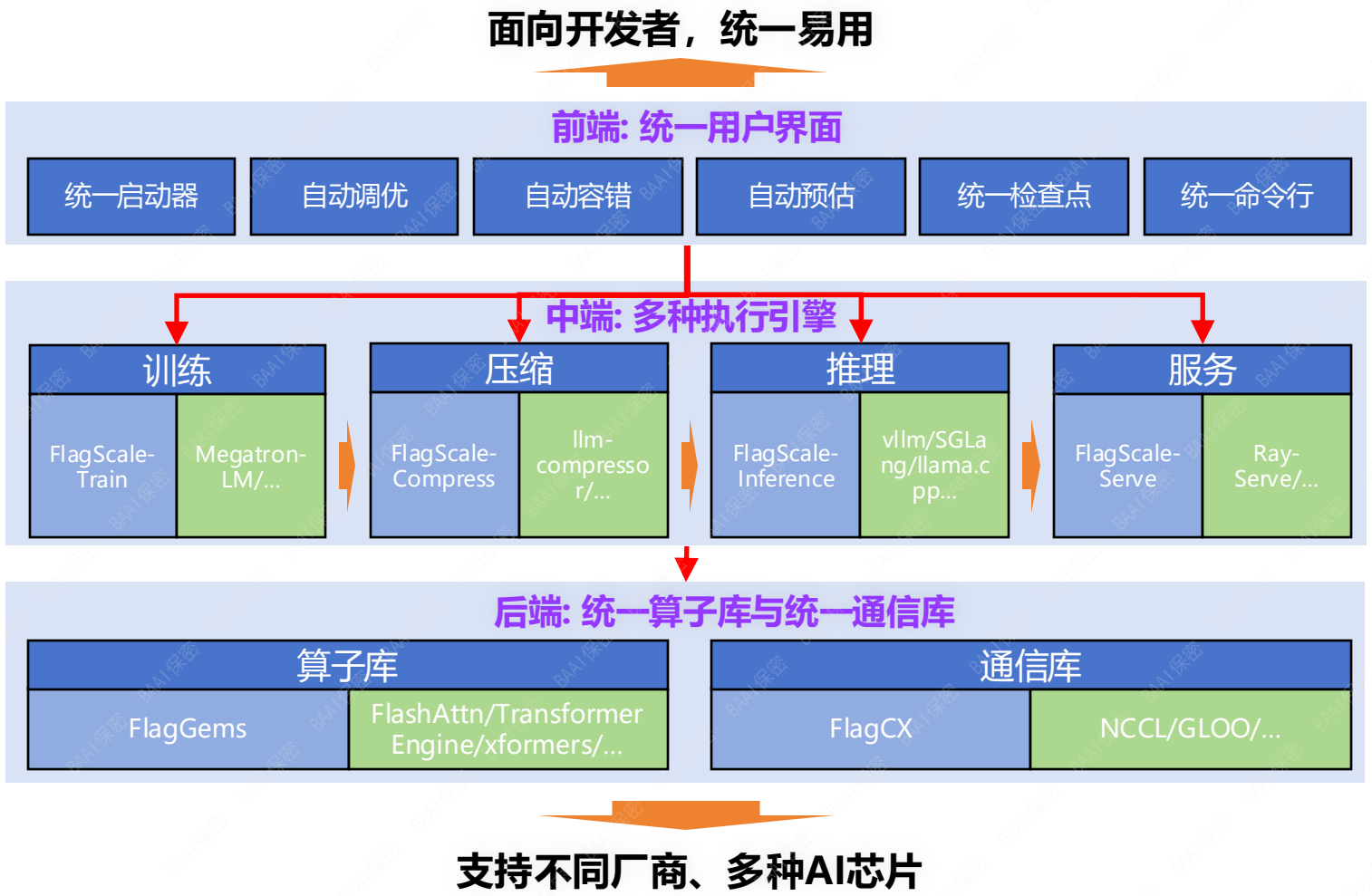
基于GitHub初始提交commit (可能不是非常准确)



三大关键碎片化挑战

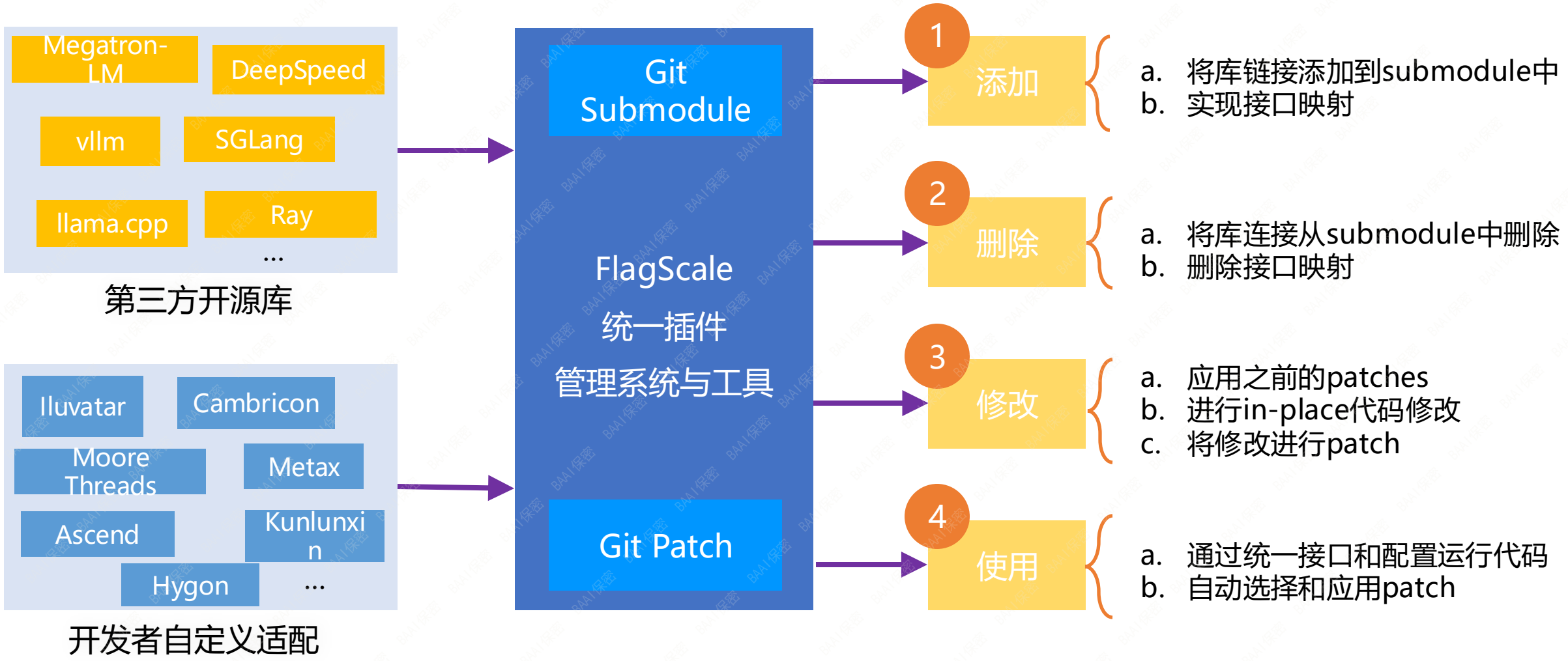
- **挑战一:** LLM 全生命周期的端到端支持碎片化
  - DeepSpeed 和 Megatron-LM 等框架主要聚焦训练，虽已支持推理，但缺乏全生命周期集成。
  - NeMo-Megatron 尝试提供从预训练到部署的全周期支持，但与 NVIDIA 生态深度绑定，限制了通用性。
- **挑战二:** 框架能力重叠且专业化导致的碎片化
  - vLLM 和 SGLang 等框架在不同场景中各有优势，选择难度大。
  - llama.cpp 等更多选项进一步分化生态，增加了框架选择和集成的复杂度。
- **挑战三:** AI 硬件碎片化与缺乏标准化
  - GPU、NPU 和定制加速器等 AI 硬件种类日益增多，用户需在性能、兼容性和成本间权衡。
  - 每个硬件平台通常需要专属工具链和优化策略，跨平台开发部署需大量工程投入。

“多芯片+多后端”的三层灵活架构设计



- 通过自动化技术实现跨芯片的自适应计算
- 通过多种可扩展的执行引擎后端支持大模型的全生命周期
- 通过统一底层库实现不同芯片间计算与通信兼容

通过插件化实现不同后端灵活管理



努力打造LLM系统开发者的“VSCode”

基于SubTree的后端管理机制


```
FlagScale/
├── flagscale/
│   └── inference/
│       └── core
│           └── llm_engine.py
│           └── model_runner.py
│           └── ......
│
├── vllm/
│   └── vllm/
│       └── engine/
│           └── llm_engine.py
│       └── worker/
│           └── model_runner.py
│   └── ......
```

FlagScale / flagscale / inference / core / **model_runner.py**

Code | Blame | 2078 lines (1819 loc) · 91.4 KB · | Raw

```
79
80  # Know more about FlagGems: https://github.com/FlagOpen/FlagGems
81  if os.getenv("USE_FLAGGEMS", "false").lower() in ("1", "true", "yes"):
82      try:
83          import flag_gems
84          flag_gems.enable()
85          logger.info("Successfully enabled flag_gems as default ops implementation.")
86      except ImportError:
87          logger.warning("Failed to import 'flag_gems'. Falling back to default implementation.")
88      except Exception as e:
89          logger.warning(f"Failed to enable 'flag_gems': {e}. Falling back to default implementation.")
90
```

- Subtree方式管理后端，完整源码存放在FlagScale中，体积大
- 自研组件运行时替换，但文件映射不清晰，核心不突出
- 与上游同步时，解冲突困难，同步不及时

基于SubModule的后端管理机制


```
FlagScale/
├── flagscale/
│       └── backends/
│           └── vllm/
│               └── vllm/
│                   └── engine/
│                       └── llm_engine.py
│                   └── worker/
│                       └── model_runner.py
│               └── ......
│
├── third_party/
│       └── vllm @ ba41cc9
```

unpatch：同步FlagScale适配至Submodule

```
FlagScale/
├── flagscale/
│       └── backends/
│           └── vllm/
│               └── vllm/
│                   └── engine/
│                       └── llm_engine.py
│                   └── worker/
│                       └── model_runner.py
│               └── ......
│
├── third_party/
│       └── vllm/
│           └── vllm/
│               └── engine/
│                   └── llm_engine.py
│               └── worker/
│                   └── model_runner.py
│           └── ......
```

- Submodule方式管理多后端，仅以commit链接形式存在。
- 文件路径一一对应，突出核心自研组件。
- 修改统一标记，与上游快速同步。

FlagScale / flagscale / backends / vllm / vllm / worker / model_runner.py

Code | Blame | 2101 lines (1836 loc) · 92.1 KB · Raw

```
77
78
79  # --- FLAGSCALE MODIFICATION BEG ---
80  # Know more about FlagGems: https://github.com/FlagOpen/FlagGems
81  import os
82  if os.getenv("USE_FLAGGEMS", "false").lower() in ("1", "true", "yes"):
83      try:
84          import flag_gems
85          flag_gems.enable()
86          logger.info("Successfully enabled flag_gems as default ops implementation.")
87      except ImportError:
88          logger.warning("Failed to import 'flag_gems'. Falling back to default implementation.")
89      except Exception as e:
90          logger.warning(f"Failed to enable 'flag_gems': {e}. Falling back to default implementation.")
91  # --- FLAGSCALE MODIFICATION END ---
```

用户使用和开发流程

- 易用开发流程，方便用户进行使用和二次开发：
  1. unpatch：同步FlagScale适配至submodule
     - `python tools/patch/unpatch.py --backend vllm`
  2. Submodule内inplace修改
  3. patch：同步submodule内修改至FlagScale
     - `python tools/patch/patch.py --backend vllm`

用户使用和开发流程

- 极简两步后端接入流程，几十行代码完成：

1. 添加submodule

   - *git submodule add https://github.com/ggml-org/llama.cpp.git third_party/llama.cpp*
   - *git add .gitmodules third_party/llama.cpp*

2. 添加对应的启动器

flagscale/serve/run_inference_engine.py +36 -1

```
+ def llama_cpp_serve(args):
+     common_args = args.get("engine_args", {})
+     llama_cpp_args = args.get("engine_args_specific", {}).get("llama_cpp", {})
+
+     command = ["./third_party/llama.cpp/build/bin/llama-server"]
+     if common_args.get("model", None):
+         converted_args = ARGS_CONVERTER.convert("llama_cpp", common_args)
+         command.extend(["--model", converted_args["model"]])
+         common_args_flatten = flatten_dict_to_args(converted_args, ["model"])
+         command.extend(common_args_flatten)
+         llama_cpp_args_flatten = flatten_dict_to_args(llama_cpp_args, ["model"])
+         command.extend(llama_cpp_args_flatten)
+     else:
+         raise ValueError("Either model must be specified in vllm_model.")
+
+     # Start the subprocess
+     logger.info(f"[Serve]: Starting llama-cpp serve with command: {' '.join(command)}")
+
+     process = subprocess.Popen(command,
```

多元算力适配的挑战

✗ **挑战1:** 集成多个开源仓库作为后端

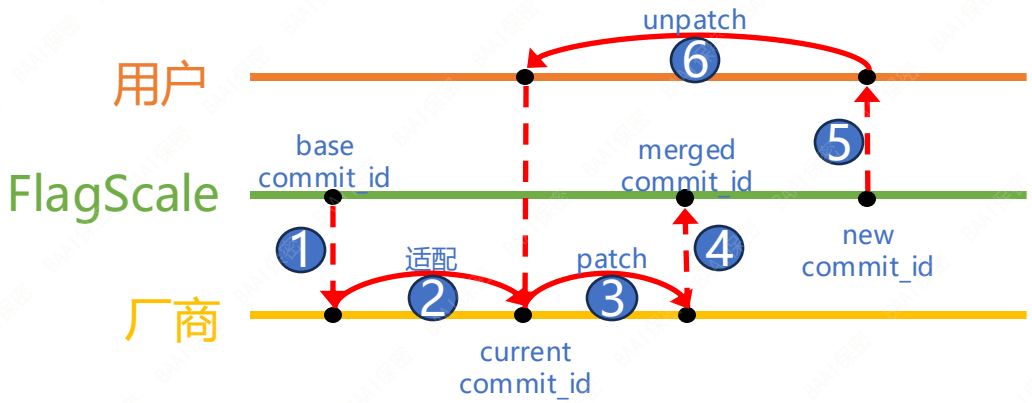
- 需保证与上游仓库兼容和及时同步。
- 可能会动态集成新的高效后端。

✗ **挑战2:** 包含多种语言代码的修改

- 除Python外，还有C++和CUDA等语言。

✗ **挑战3:** 多厂商适配与贡献

- Monkey Patch方式仅适用于单一厂商。
- Plugin方式尚未成为厂商共识。
- 对训推框架侵入式修改，影响框架本身迭代更新。



✓ **解决方案: FlagScale多硬件适配机制**

多硬件适配机制


```
FlagScale/
├── hardware/
│   └── device_A/
│       └── vllm
│           └── diff.patch
│           └── diff.yaml
│       └── FlagScale
│           └── diff.patch
│           └── diff.yaml
│   └── device_B/
│       └── ……
│   ……
│   └── patch_history.yaml
```

- 新增backend目录，分后端存放patch文件，每个后端下有且仅存在一个patch。
- 新增patch元信息yaml文件，存储任务场景、后端版本、支持模型等关键元信息。

FlagScale / hardware / Cambricon_MLU / vllm / **diff.yaml**

cifar10 [MLU] Support qwen infer (#551) ✓ 10faa30 · last week History

Code Blame 13 lines (13 loc) · 252 Bytes · Raw

```yaml
backends_commit:
  vllm: 296c6572dd1f76b31b93be19e550790afcfb8843
backends_version:
  FlagScale: main
  vllm: v0.8.3
commit: 39c0adcf0db48417ded4842c92067626bd55562f
contact: ''
device_type: Cambricon_MLU
models:
- qwen3
- qwen2_5_vl
task:
- inference
```

- 新增厂商所有历史信息yaml文件，由FlagScale统一维护。

```yaml
Cambricon_MLU:
  inference:
    FlagScale+vllm:
      - 15ea26dfdb0bcd
```

厂商接入与用户使用

- 厂商接入流程：
  1. unpatch：同步FlagScale适配至submodule
     - *python tools/patch/unpatch.py --backend vllm --mode copy*
  2. Submodule内inplace修改
  3. patch：生成patch文件
     - *python tools/patch/patch.py --backend vllm --task train --device-type Chip_Vendor --commit e6e5643776f12058418fbb70db3a0f95273083d8*

- 用户一步使用：
  1. unpatch：应用厂商的patch文件
     - *python tools/patch/unpatch.py --backend vllm FlagScale --deice-type Chip_Vendor –task inference*

训练适配情况

- 适配天数、寒武纪、摩尔线程、沐曦、昆仑芯、清微智能、海光、华为共8家国产芯片。
- 支持MoE和Dense模型架构，覆盖十亿和百亿大模型规模，涵盖语言和多模态类型。
- 版本从0.10.0至0.12.0，大部分厂商已跟进到0.12.0。

天数智芯 Iluvatar CoreX　摩尔线程 MOORE THREADS　昆仑芯 KUNLUNXIN　HYGON

Cambricon 寒武纪　MetaX 沐曦　清微智能 TSING MICRO　Ascend

推理适配情况

- 适配天数、寒武纪、摩尔线程、沐曦、昆仑芯、清微智能、海光、华为共8家国产芯片。
- 推理版本从0.7.2到最新的0.9.2不等。
- 适配方式：
  - Inplace： 直接修改vLLM原始代码，安装修改后的vLLM。
  - Plugin： 通过vLLM自身的插件机制添加后端相关的注册机制和相关代码，安装vLLM和厂商自身的vLLM。

天数智芯 Iluvatar CoreX　摩尔线程 MOORE THREADS　昆仑芯 KUNLUNXIN　HYGON

Cambricon 寒武纪　MetaX 沐曦　清微智能 TSING MICRO　Ascend

多后端多芯片统一的安装设计

- 源码编译安装：

```
PYTHONPATH=$FLAGSCALE_PATH:$PYTHONPATH pip install . --config-settings=backend=<backend>
--config-settings=device=<device> --no-build-isolation
```

- 摩尔AI PC上安装llama.cpp

```
PYTHONPATH=$FLAGSCALE_PATH:$PYTHONPATH pip install . --config-settings=backend=llama.cpp
--config-settings=device=musa --no-build-isolation
```

- GPU上安装vllm、llama.cpp、sglang三个推理后端

```
PYTHONPATH=$FLAGSCALE_PATH:$PYTHONPATH pip install . --config-
settings=backend=vllm,llama.cpp,sglang --config-settings=device=gpu --no-build-isolation
```

- 寒武纪上安装vllm

```
PYTHONPATH=$FLAGSCALE_PATH:$PYTHONPATH pip install . --config-settings=backend=vllm --
config-settings=device=Cambricon_MLU --no-build-isolation
```

多后端多芯片统一的安装设计

- whl包安装:

```
pip install flag_scale -i https://test.pypi.org/simple/
```

- GPU上安装vllm

```
flagscale install vllm
```

- 沐曦上安装vllm

```
flagscale install vllm --device metax
```

- ……

FlagScale CLI命令行体系

- flagscale pull <image> <ckpt>: 一键拉取镜像和模型参数文件。
- flagscale test: 一键执行单元测试和功能测试。
- flagscale serve <model>: 一键进行服务部署，在不同芯片上命令相同。
- flagscale train <model>: 一键进行模型训练，在不同芯片上命令相同。
- flagscale install <backend> --device <device>: 一键安装指定芯片上的指定后端。

谢谢聆听，
欢迎合作与建议。

https://github.com/FlagOpen/FlagScale