```
from dotenv import load_dotenv
import base64
import os
from io import BytesIO
from typing import List, Optional, Tuple

import torch
import uvicorn
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from loguru import logger
from PIL import Image
from sse_starlette.sse import EventSourceResponse
from transformers import (
    AutoModelForCausalLM,
    BitsAndBytesConfig,
    LlamaTokenizer,
    PreTrainedModel,
    PreTrainedTokenizer,
    TextIteratorStreamer,
)

from swarms_cloud.schema.cog_vlm_schemas import (
    ChatCompletionRequest,
    ChatCompletionResponse,
    ChatCompletionResponseChoice,
```

```
    ChatCompletionResponseStreamChoice,
    ChatMessageInput,
    ChatMessageResponse,
    DeltaMessage,
    ImageUrlContent,
    ModelCard,
    ModelList,
    TextContent,
    UsageInfo,
)

# from exa.structs.parallelize_models_gpus import prepare_model_for_ddp_inference

# Load environment variables from .env file
load_dotenv()

# Environment variables
MODEL_PATH = os.environ.get("COGVLM_MODEL_PATH", "THUDM/cogvlm-chat-hf")
TOKENIZER_PATH = os.environ.get("TOKENIZER_PATH", "lmsys/vicuna-7b-v1.5")
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
QUANT_ENABLED = os.environ.get("QUANT_ENABLED", True)

# Create a FastAPI app
app = FastAPI(debug=True)
```

```
## Verify the API key



# Load the middleware to handle CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Load the tokenizer and model
tokenizer = LlamaTokenizer.from_pretrained(TOKENIZER_PATH, trust_remote_code=True)

if torch.cuda.is_available() and torch.cuda.get_device_capability()[0] >= 8:
    torch_type = torch.bfloat16
else:
    torch_type = torch.float16

print(f"========Use torch type as:{torch_type} with device:{DEVICE}========\n\n")

quantization_config = {
    "load_in_4bit": True,
    "bnb_4bit_use_double_quant": True,
    "bnb_4bit_compute_dtype": torch_type,
```

```
}

bnb_config = BitsAndBytesConfig(**quantization_config)

model = AutoModelForCausalLM.from_pretrained(
    MODEL_PATH,
    trust_remote_code=True,
    torch_dtype=torch_type,
    low_cpu_mem_usage=True,
    quantization_config=bnb_config,
    load_in_4bit=True,
).eval()

# model = prepare_model_for_ddp_inference(model)

# Torch type
if torch.cuda.is_available() and torch.cuda.get_device_capability()[0] >= 8:
    torch_type = torch.bfloat16
else:
    torch_type = torch.float16



@app.get("/v1/models", response_model=ModelList)
async def list_models():
    """
    An endpoint to list available models. It returns a list of model cards.
```

```
    This is useful for clients to query and understand what models are available for use.
    """
    model_card = ModelCard(
        id="cogvlm-chat-17b"
    )  # can be replaced by your model id like cogagent-chat-18b
    return ModelList(data=[model_card])


@app.post("/v1/chat/completions", response_model=ChatCompletionResponse)
async def create_chat_completion(
    request: ChatCompletionRequest,  # token: str = Depends(authenticate_user)
):
    try:
        if len(request.messages) < 1 or request.messages[-1].role == "assistant":
            raise HTTPException(status_code=400, detail="Invalid request")

        # print(f"Request: {request}")

        gen_params = dict(
            messages=request.messages,
            temperature=request.temperature,
            top_p=request.top_p,
            max_tokens=request.max_tokens or 1024,
            echo=False,
            stream=request.stream,
        )
```

```
if request.stream:
    generate = predict(request.model, gen_params)
    return EventSourceResponse(generate, media_type="text/event-stream")

# Generate response
response = generate_cogvlm(model, tokenizer, gen_params)

usage = UsageInfo()

# ChatMessageResponse
message = ChatMessageResponse(
    role="assistant",
    content=response["text"],
)

# # # Log the entry to supabase
# entry = ModelAPILogEntry(
#     user_id=fetch_api_key_info(token),
#     model_id="41a2869c-5f8d-403f-83bb-1f06c56bad47",
#     input_tokens=count_tokens(request.messages, tokenizer, request.model),
#     output_tokens=count_tokens(response["text"], tokenizer, request.model),
#     all_cost=calculate_pricing(
#         texts=[message.content], tokenizer=tokenizer, rate_per_million=15.0
#     ),
#     input_cost=calculate_pricing(
```

```
#         texts=[message.content], tokenizer=tokenizer, rate_per_million=15.0
#     ),
#     output_cost=calculate_pricing(
#         texts=response["text"], tokenizer=tokenizer, rate_per_million=15.0
#     )
#     * 5,
#     messages=request.messages,
#     # temperature=request.temperature,
#     top_p=request.top_p,
#     # echo=request.echo,
#     stream=request.stream,
#     repetition_penalty=request.repetition_penalty,
#     max_tokens=request.max_tokens,
# )

# # Log the entry to supabase
# log_to_supabase(entry=entry)

# ChatCompletionResponseChoice
logger.debug(f"==== message ====\n{message}")
choice_data = ChatCompletionResponseChoice(
    index=0,
    message=message,
)

# task_usage = UsageInfo.model_validate(response["usage"])
```

```
        task_usage = UsageInfo.parse_obj(response["usage"])
        for usage_key, usage_value in task_usage.dict().items():
            setattr(usage, usage_key, getattr(usage, usage_key) + usage_value)

        out = ChatCompletionResponse(
            model=request.model,
            choices=[choice_data],
            object="chat.completion",
            usage=usage,
        )

        return out
    except Exception as e:
        logger.error(f"Error: {e}")
        raise HTTPException(status_code=500, detail="Internal Server Error")


async def predict(model_id: str, params: dict):
    """
    Handle streaming predictions. It continuously generates responses for a given input stream.
    This is particularly useful for real-time, continuous interactions with the model.
    """

    choice_data = ChatCompletionResponseStreamChoice(
        index=0, delta=DeltaMessage(role="assistant"), finish_reason=None
    )
```

```
chunk = ChatCompletionResponse(
    model=model_id, choices=[choice_data], object="chat.completion.chunk"
)

# Log to supabase
# supabase_logger.log(chunk)

yield f"{chunk.model_dump_json(exclude_unset=True)}"

previous_text = ""
for new_response in generate_stream_cogvlm(model, tokenizer, params):
    decoded_unicode = new_response["text"]
    delta_text = decoded_unicode[len(previous_text) :]
    previous_text = decoded_unicode
    delta = DeltaMessage(
        content=delta_text,
        role="assistant",
    )
    choice_data = ChatCompletionResponseStreamChoice(
        index=0,
        delta=delta,
    )

    chunk = ChatCompletionResponse(
        model=model_id, choices=[choice_data], object="chat.completion.chunk"
```

```
        )

        yield f"{chunk.model_dump_json(exclude_unset=True)}"

    choice_data = ChatCompletionResponseStreamChoice(
        index=0,
        delta=DeltaMessage(),
    )

    chunk = ChatCompletionResponse(
        model=model_id, choices=[choice_data], object="chat.completion.chunk"
    )

    yield f"{chunk.model_dump_json(exclude_unset=True)}"


def generate_cogvlm(
    model: PreTrainedModel, tokenizer: PreTrainedTokenizer, params: dict
):
    """
    Generates a response using the CogVLM model. It processes the chat history and image data, if any,
    and then invokes the model to generate a response.
    """

    for response in generate_stream_cogvlm(model, tokenizer, params):
```

```
        pass
    return response


def process_history_and_images(
    messages: List[ChatMessageInput],
) -> Tuple[Optional[str], Optional[List[Tuple[str, str]]], Optional[List[Image.Image]]]:
    """
    Process history messages to extract text, identify the last user query,
    and convert base64 encoded image URLs to PIL images.

    Args:
        messages(List[ChatMessageInput]): List of ChatMessageInput objects.
    return: A tuple of three elements:
            - The last user query as a string.
            - Text history formatted as a list of tuples for the model.
            - List of PIL Image objects extracted from the messages.
    """
    formatted_history = []
    image_list = []
    last_user_query = ""

    for i, message in enumerate(messages):
        role = message.role
        content = message.content
```

```
if isinstance(content, list):  # text
    text_content = " ".join(
        item.text for item in content if isinstance(item, TextContent)
    )
else:
    text_content = content


if isinstance(content, list):  # image
    for item in content:
        if isinstance(item, ImageUrlContent):
            image_url = item.image_url.url
            if image_url.startswith(""):
                base64_encoded_image = image_url.split(
                    ""
                )[1]
                image_data = base64.b64decode(base64_encoded_image)
                image = Image.open(BytesIO(image_data)).convert("RGB")
                image_list.append(image)


if role == "user":
    if i == len(messages) - 1:  #
        last_user_query = text_content
    else:
        formatted_history.append((text_content, ""))
elif role == "assistant":
    if formatted_history:
```

```
            if formatted_history[-1][1] != "":
                assert (
                    False
                        ), f"the last query is answered. answer again. {formatted_history[-1][0]}, {formatted_history[-1][1]}, {text_content}"
                formatted_history[-1] = (formatted_history[-1][0], text_content)
            else:
                assert False, "assistant reply before user"
        else:
            assert False, f"unrecognized role: {role}"

    return last_user_query, formatted_history, image_list


@torch.inference_mode()
def generate_stream_cogvlm(
    model: PreTrainedModel, tokenizer: PreTrainedTokenizer, params: dict
):
    """
    Generates a stream of responses using the CogVLM model in inference mode.
        It's optimized to handle continuous input-output interactions with the model in a streaming manner.
    """
    messages = params["messages"]
    temperature = float(params.get("temperature", 1.0))
    repetition_penalty = float(params.get("repetition_penalty", 1.0))
```

```
top_p = float(params.get("top_p", 1.0))
max_new_tokens = int(params.get("max_tokens", 256))
query, history, image_list = process_history_and_images(messages)
logger.debug(f"==== request ====\n{query}")

input_by_model = model.build_conversation_input_ids(
    tokenizer, query=query, history=history, images=[image_list[-1]]
)
inputs = {
    "input_ids": input_by_model["input_ids"].unsqueeze(0).to(DEVICE),
    "token_type_ids": input_by_model["token_type_ids"].unsqueeze(0).to(DEVICE),
    "attention_mask": input_by_model["attention_mask"].unsqueeze(0).to(DEVICE),
    "images": [[input_by_model["images"][0].to(DEVICE).to(torch_type)]],
}
if "cross_images" in input_by_model and input_by_model["cross_images"]:
    inputs["cross_images"] = [
        [input_by_model["cross_images"][0].to(DEVICE).to(torch_type)]
    ]

input_echo_len = len(inputs["input_ids"][0])
streamer = TextIteratorStreamer(
    tokenizer=tokenizer, timeout=60.0, skip_prompt=True, skip_special_tokens=True
)
gen_kwargs = {
    "repetition_penalty": repetition_penalty,
    "max_new_tokens": max_new_tokens,
```

```
    "do_sample": True if temperature > 1e-5 else False,
    "top_p": top_p if temperature > 1e-5 else 0,
    "streamer": streamer,
}
if temperature > 1e-5:
    gen_kwargs["temperature"] = temperature

total_len = 0
generated_text = ""
with torch.no_grad():
    model.generate(**inputs, **gen_kwargs)
    for next_text in streamer:
        generated_text += next_text
        yield {
            "text": generated_text,
            "usage": {
                "prompt_tokens": input_echo_len,
                "completion_tokens": total_len - input_echo_len,
                "total_tokens": total_len,
            },
        }
ret = {
    "text": generated_text,
    "usage": {
        "prompt_tokens": input_echo_len,
        "completion_tokens": total_len - input_echo_len,
```

```
            "total_tokens": total_len,
        },
    }
    yield ret



if __name__ == "__main__":
    uvicorn.run(
        app,
        host="0.0.0.0",
        port=int(os.environ.get("MODEL_API_PORT", 8000)),
        # workers=5,
        log_level="info",
        use_colors=True,
        # reload=True,
    )
```