AIOS Instance SDK - LLM Utilities

The AIOS LLM Utilities library provides methods and tools built on top of frameworks such as `open-llm`, `llama.cpp`, `ollama`, and `torch transformers`. These frameworks are used for loading and running inference on LLM models. With this library, you can easily load and serve LLM models as AIOS instances.

Using transformers utilities - useful for large LLM models across multiple GPUs with torch transformer models:

The `TransformersUtils` class is a utility wrapper around Hugging Face's `transformers` library designed for inference-only usage. It supports both single-GPU and multi-GPU execution using `tp_plan` for tensor parallelism. The class also provides optional integration for performance metrics and supports text generation, tokenization, and embedding extraction.

Installation:

```
cd services/block/llma-utils/aios_transformers/
```

```
pip3 install -e .
```

And then the library can be imported like:

```
import aios_transformers
```

The library exports two classes:

```
# interface for troch transformers for all kinds of LLM model functionalities
from aios_transformers import TransformersUtils

# using the LLM metrics (explained in the later section of documentation)
from aios_transformers import LLMMetrics
```

Initialization

```
TransformersUtils(
    model_name: str = "gpt2",
    device: str = None,
    metrics=None,
    tensor_parallel: bool = False,
    quantize: bool = False,
    generation_config: dict = {}
)
```

Parameters:

- `model_name`: Hugging Face model identifier. Defaults to `"gpt2"`.
- `device`: Device string (`"cuda"`, `"cpu"`). If `None`, auto-detects.
- `metrics`: Optional metrics tracker object with logging methods.
- `tensor_parallel`: If `True`, enables multi-GPU inference using `device_map="auto"`.
- `quantize`: Currently unused placeholder for quantization support.
- `generation_config`: Default generation parameters for text generation.

Model Loading

`load_model(model_name: str, extra_args: dict = {})`

Loads the model and tokenizer using Hugging Face's `AutoModelForCausalLM`.

- If `tensor_parallel=True`, uses `device_map="auto"` for automatic GPU sharding.
- Otherwise, loads the model onto the specified device (`cuda` or `cpu`).
- Initializes a `text-generation` pipeline.

`reload_model()`

Reloads the model using the last used `model_name`.

`set_generator(generator: Pipeline)`

Sets a pre-initialized pipeline as the generator.

Generation Configuration

`set_generation_config(**kwargs)`

Updates the default generation parameters such as `max_new_tokens`, `top_k`, `temperature`, etc.

Text Generation

`generate(prompt: str, **kwargs)`

Generates text from a given prompt using the initialized pipeline.

- Merges default `generation_config` with any user-specified parameters.
- Automatically logs prompt/response metrics and inference time if metrics are enabled.

`generate_tokens(prompt: str, **kwargs)`

Generates tokens directly using `model.generate()` instead of the pipeline.

Tokenizer Helpers

`tokenize(text: str)`

Returns tokenized output using the loaded tokenizer.

`decode(token_ids)`

Decodes a list of token IDs into text using the tokenizer.

Embeddings

`get_embeddings(text: str)`

Extracts the last hidden state (embeddings) from the transformer model for a given input.

- Uses `AutoModel` for embedding extraction.
- Supports tensor parallel or single-device inference.

Chat Interface

Supports chat-style session tracking using session IDs.

`create_chat_session(session_id: str, system_message: str = "")`

Initializes a new chat session with an optional system prompt.

`add_message_to_chat(session_id: str, message: str, role: str = "user")`

Appends a user or assistant message to an active chat session.

`run_chat_inference(session_id: str, **kwargs)`

Generates a model response based on the full conversation history.

- Builds the prompt from session messages.
- Updates chat history with assistant's response.

`remove_chat_session(session_id: str)`

Deletes a chat session by ID.

Device Info

`get_device_info()`

Returns a dictionary with runtime hardware and model information:

```
{
    "device": "cuda" or "cpu",
    "cuda_available": True or False,
    "num_gpus": <int>,
    "tensor_parallel": True or False,
    "model": "<model_name>"
}
```

Example:

```
from aios_transformers import TransformersUtils  # assuming your class is saved as transfor

# Initialize with single-GPU inference (default behavior)
utils = TransformersUtils(
    model_name="Qwen/Qwen1.5-0.5B-Chat",        # small, fast model for testing
    tensor_parallel=True     # disable multi-GPU tensor parallelism
)

utils.load_model()

# Define a simple prompt
prompt = "hey!"

# Generate text
generated = utils.generate(prompt)
print("Generated Text:\n", generated)

# Tokenize and decode for demonstration
tokens = utils.tokenize(prompt)
print("\nToken IDs:", tokens["input_ids"])

decoded = utils.decode(tokens["input_ids"][0])
print("Decoded Text:", decoded)

# Get raw token output
```

```
token_output = utils.generate_tokens(prompt, max_new_tokens=20)
print("\nGenerated Token IDs:", token_output.tolist()[0])

print(utils.get_device_info())

utils.create_chat_session('123', "You are a chat bot, you respond to only what is asked")

utils.add_message_to_chat('123', "Tell me about yourself")

data = utils.run_chat_inference('123')

print(data)
```

Using LLAMA.cpp utilities - useful for simple model/single GPU inference with GGUF models

A utility wrapper around the `llama_cpp` library to simplify model loading, tokenization, streaming and batch inference, and managing chat-style interactions with support for performance metrics logging and GPU usage.

```
cd services/block/llma-utils/aios_llama_cpp/
```

```
pip3 install -e .
```

And then the library can be imported like:

```
import aios_llama_cpp
```

The library exports two classes:

```
# interface for troch transformers for all kinds of LLM model functionalities
from aios_llama_cpp import LLAMAUtils

# using the LLM metrics (explained in the later section of documentation)
from aios_llama_cpp import LLMMetrics
```

Initialization

```
LLAMAUtils(
    model_path: str,
    use_gpu: bool = False,
    gpu_id: int = 0,
    metrics = None
)
```

Parameters:

- **`model_path`**: Path to the `.gguf` or `.bin` model file.
- **`use_gpu`**: Whether to use GPU acceleration (if supported by `llama_cpp`).
- **`gpu_id`**: Index of the GPU to use if `use_gpu` is `True`.
- **`metrics`**: Optional object with custom metric logging hooks for tokens and latency.

Model Loading

`load_model() -> bool`

Loads the model from `model_path` using the specified device configuration.

- Returns `True` on success, `False` otherwise.
- Handles GPU configuration and logs status.

Chat Support

`supports_chat() -> bool`

Checks whether the loaded model supports chat-style generation via `create_chat_completion()`.

Prompt Inference

`run_inference(prompt: str, stream: bool = False, **kwargs)`

Performs inference on a given prompt.

- If `stream=True`, yields token chunks in real-time using `stream_inference()`.
- Tracks and logs generation latency and token usage via the metrics module (if provided).
- Supports runtime overrides of generation parameters.

`generate_text(prompt: str, num_sequences: int = 1, **kwargs)`

Generates multiple completions for a given prompt.

- Useful for sampling multiple outputs from the same prompt.
- Collects metrics per completion.

`stream_inference(prompt: str, **kwargs)`

Streams inference output token-by-token to stdout.

- Meant for CLI or interactive use.

- Handles exceptions and streaming errors gracefully.

Tokenization

```
tokenize(text: str)
```

Returns token IDs for a given text string.

```
detokenize(tokens: List[int])
```

Converts a list of token IDs back into human-readable text.

Model Utilities

```
save_model(save_path: str) -> bool
```

Saves the current model state to a given path (if supported).

```
get_model_info() -> dict
```

Returns metadata about the loaded model such as size, number of layers, or architecture (if available).

```
set_seed(seed: int) -> bool
```

Sets the random seed for deterministic inference.

Chat Interface

Maintains multi-turn conversation context per session ID.

```
create_chat_session(session_id: str, system_message: str = "",
tools_list: list = None, tools_choice: dict = None)
```

Creates a new session with optional system message and tool configuration (for tool-augmented models).

```
add_message_to_chat(session_id: str, message: str, role: str =
"user")
```

Appends a message to the session's conversation history.

run_chat_inference(session_id: str, **kwargs) -> str

Runs inference for the current session using `create_chat_completion()`.

- Supports structured tool inputs and streaming.
- Logs inference metrics.
- Returns only the assistant's generated message content.

remove_chat_session(session_id: str)

Deletes the specified chat session and updates active session metrics.

Default Generation Configuration

Set during initialization, can be overridden per call:

```
{
    "max_tokens": 50,
    "temperature": 1.0,
    "top_p": 1.0,
    "stop": ["Q:", "\n"]
}
```

You can override these by passing arguments like `max_tokens=100` during any inference call.

Example:

```
from aios_llama_cpp import LLAMAUtils  # Assuming your class is saved as llama_utils.py


# Initialize
model_path = "<path>/qwen1_5-0_5b-chat-q2_k.gguf"  # Update this to the real path
llama = LLAMAUtils(model_path=model_path, use_gpu=True, gpu_id=0, metrics=None)

# Load model
if llama.load_model():
    # Run basic inference
    result = llama.run_inference("What is the capital of France?")
    print("\nResult:", result["choices"][0]["text"].strip() if result else "No result")

    # Run streaming inference
    print("\nStreaming:")
    llama.run_inference("Write a haiku about space.", stream=True)

    # Chat session
    session_id = "chat123"
```

```
llama.create_chat_session(session_id, system_message="You are a helpful assistant.")
llama.add_message_to_chat(session_id, "What's the weather like on Mars?")
response = llama.run_chat_inference(session_id)
print("\n\nChat Response:", response)

llama.remove_chat_session(session_id)
```

Using `LLMMetrics` for custom LLM metrics:

Metric Name	Type	Description	Unit / Buckets
`llm_prompts_total`	Counter	Total number of prompts received and processed	Count
`llm_tokens_generated_total`	Counter	Total number of tokens generated by the LLM	Count
`llm_prompt_tokens_total`	Counter	Total number of tokens received in prompts	Count
`llm_active_sessions`	Gauge	Current number of active chat sessions	Count
`llm_inference_duration_seconds`	Histogram	Time taken for full inference	Seconds — `[0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10]`
`llm_time_to_first_token_seconds`	Histogram	Time from prompt receipt to first token generated (TTFT)	Seconds — `[0.01, 0.05, 0.1, 0.2, 0.5, 1, 2]`
`llm_time_per_output_token_seconds`	Histogram	Time taken to generate each output token (TPOT)	Seconds — `[0.01, 0.05, 0.1, 0.2, 0.5]`
`llm_tokens_per_second`	Gauge	Rate of tokens generated per second	Float (tokens/sec)
`llm_cpu_utilization`	Gauge	CPU utilization percentage during inference	Percentage (0–100)
`llm_gpu_utilization`	Gauge	GPU utilization percentage during inference	Percentage (0–100)
`llm_memory_usage_bytes`	Gauge	Memory consumed during inference	Bytes
`llm_inference_errors_total`	Counter	Total number of errors encountered during inference	Count

Public Method Documentation

`__init__(metrics, block_id=None)`

Initializes the metrics handler and registers all required metrics.

- **`metrics`**: An instance of a Prometheus-compatible metrics manager.
- **`block_id`** *(optional)*: Unique identifier for the block (for context or filtering).

`log_prompt(prompt_token_count: int)`

Logs the reception of a prompt and increments the token count received.

- **`prompt_token_count`**: Number of tokens in the incoming prompt.

`log_response(generated_token_count: int)`

Logs the number of tokens generated in response.

- **`generated_token_count`**: Number of tokens generated by the model.

`observe_inference_time(start_time: float)`

Measures total time taken for model inference.

- **`start_time`**: Timestamp when inference started (`time.time()`).

`observe_time_to_first_token(start_time: float)`

Measures time to first token generation after inference begins (TTFT).

- **`start_time`**: Timestamp when inference started.

`observe_time_per_output_token(start_time: float, token_count: int)`

Calculates average time spent per output token.

- **`start_time`**: Timestamp when inference started.
- **`token_count`**: Number of tokens generated.

`update_tokens_per_second(tokens_generated: int, duration_seconds: float)`

Updates the token throughput metric (tokens/sec).

- **`tokens_generated`**: Number of tokens produced.
- **`duration_seconds`**: Time taken to generate them.

`update_resource_utilization(cpu_percent=None, gpu_percent=None, memory_bytes=None)`

Sets resource usage gauges.

- **`cpu_percent`**: CPU usage in percentage.
- **`gpu_percent`**: GPU usage in percentage.
- **`memory_bytes`**: Memory usage in bytes.

`increment_inference_errors()`

Increments the counter for inference errors encountered.

`increase_active_sessions()`

Increments the number of active chat sessions.

`decrease_active_sessions()`

Decrements the number of active chat sessions.

Using the `LLMMetrics` class:

```
# LLMMetrics class needs to be instantiated by passing the object
# of AIOSMetrics provided in the context object passed in the constructor of your AIOS Insta
llm_metrics = LLMMetrics(context.metrics)
```

Using the docker images for building:

Docker images for llama.cpp base and torch transformers base are built on top of `aios_instance:v1` docker image.

Here is how the docker images can be built:

```
cd services/block/llma-utils/aios_llama_cpp
docker build . -t aios_llama_cpp:v1
```

And

```
cd services/block/llma-utils/aios_transformers
docker build . -t aios_transformers:v1
```

Later, these images can be used as base to build instance components which can be on-boarded into the components registry.

Integration with AIOS Instance SDK:

Here is an example of using LLMAUtils to build a simple chat server with `qwen1.5-0.5B-chat` model:

```python
import time
import json
from llama_cpp import Llama
from aios_instance import AIOSMetrics  # assume this exists
from aios_instance import AIOSPacket, PreProcessResult, OnDataResult
from aios_llma_cpp import LLAMAUtils, LLMMetrics

class ChatBlock:
    def __init__(self, context):
        """
        Initialize chat block with LLaMA model and session context.
        """
        self.context = context

        llm_metrics = LLMMetrics(context.metrics)

        self.llama = LLAMAUtils(
            model_path=context.block_init_parameters.get("model_path", "./"),
            use_gpu=True,
            metrics=llm_metrics
        )

        loaded = self.llama.load_model()
        if not loaded:
            raise RuntimeError("Failed to load LLaMA model.")

        self.default_temperature = context.block_init_parameters.get("temperature", 0.8)

    def on_preprocess(self, packet):
        """
        Parse user message and session ID.
        """
        try:
            data = packet.data
            if isinstance(data, str):
                data = json.loads(data)

            if "inputs" in data:
                results = []
                for item in data["inputs"]:
                    results.append(PreProcessResult(packet=packet, extra_data={"input": ite
                return True, results
            else:
                return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
        except Exception as e:
            return False, str(e)

    def on_data(self, preprocessed_entry):
        """
        Run LLaMA chat inference using LLAMAUtils.
        """
        try:
            input_data = preprocessed_entry.extra_data["input"]
            user_message = input_data.get("message")
            session_id = input_data.get("session_id", "default")

            # Create chat session if not exists
            if session_id not in self.llama.chat_sessions:
                self.llama.create_chat_session(session_id, system_message="You are a helpful

            self.llama.add_message_to_chat(session_id, user_message, role="user")

            response = self.llama.run_chat_inference(
                session_id,
                temperature=self.default_temperature
            )

            return True, OnDataResult(output={"reply": response})
        except Exception as e:
            return False, str(e)

    def on_update(self, updated_parameters):
        try:
            if "temperature" in updated_parameters:
                self.default_temperature = updated_parameters["temperature"]
            return True, updated_parameters
        except Exception as e:
            return False, str(e)
```

```python
def health(self):
    return {"status": "healthy", "model_loaded": self.llama.model is not None}

def management(self, action, data):
    try:
        if action == "reset":
            self.llama.chat_sessions.clear()
            return {"message": "Chat sessions cleared."}
        return {"message": f"Unknown action '{action}'"}
    except Exception as e:
        return {"error": str(e)}

def get_muxer(self):
    return None  # or Muxer(N=3) if using packet merging
```