IK2221 GPU Inferencing Project

Grading

The project grading will be based on the completion and quality of the four main tasks described in the assignment. The distribution of points is as follows:

- **Task 1**: Evaluation of LLM inference with and without KV caching — **20 points**.
- **Task 2**: Implementation and analysis of a scheduler for batched requests — **20 points**.
- **Task 3**: Implementation of RAG and integration into the system pipeline — **20 points**.
- **Task 4**: Clustering based on KV cache representations — **10 bonus points**.

The bonus points from Task 4 can be used to compensate for points lost in the earlier tasks. In other words, if you have missed points in Tasks 1–3 or in project assignment 1, a good performance in Task 4 can help recover up to 10 points.

Submission Instructions:
Each group must submit:

- The full **source code** of the project.
- A **written report** (one per group), containing:
  - Clear instructions on how to set up and test the code.
  - A description of your implemented blocks and how they solve each task.
  - Detailed answers to the assignment questions, including explanations, motivations, and experimental results supporting your answers.
  - The contribution of each group member.

Important: The deadline for submission can be found on the course Canvas page. The final project evaluation will be individual, and we will assess each member contribution during the project discussions. In general, every member should be able to identify personal contributions, and be familiar with the whole project, ready to explain any part of it if asked. Every team member should be present during the project discussions.

Access to Computing Resources

Each group will be granted access to a dedicated server via SSH, equipped with a **NVIDIA Tesla T4 GPU**. Due to limited availability of hardware resources, each GPU will be shared between two groups on a time-scheduled basis.

The typical usage plan is as follows:

- One group will have access during the **morning slot** (**09:00–13:00**).
- Another group will have access during the **afternoon slot** (**13:00–17:00**).

Students are expected to primarily work during their assigned time slots, within the official working hours of **09:00–17:00**. However, if students wish to continue working beyond 17:00, it is permitted, and two additional evening slots are available:

- **17:00–19:00**
- **19:00–21:00**

Note that work after 21:00 is not supported.

Each group will receive by email:

- A unique **username** for SSH access (one per group).
- An associated **password**.
- The **IP address** of the assigned server.

To connect to your assigned server, use the following command in your terminal:

```
ssh -J username@nslabgw.it.kth.se username@IP_ADDRESS
```

For example, if your username is `group7` and the server IP address is `192.168.2.27`, you would connect using:

```
ssh -J group7@nslabgw.it.kth.se group7@192.168.2.27
```

Important: Detailed information about your assigned time slot and server assignment can be found on Canvas, in the scheduling sheet for the project. Please make sure to respect your assigned slot and to log out once you are finished, in order to allow fair and efficient resource sharing among all groups.

Background

As we know, Large Language Models (LLMs) are being widely used in different areas from AI healthcare to code development and analysis. However, the increasing size and complexity of state of the art models requires a lot of computational resources to respond queries. Consequently, one critical dimension of machine learning systems is optimizing the inference processing and minimize computation in runtime. In this project we are going to focus on this area by deploying a state of the art LLM inferencing framework on a server and examine the resource utilization benefit of having a proper prompt scheduler in a system.

Generative decoder-only LLMs

Generative decoder-only LLMs are based on the attention mechanism, which enables efficient parallel computation of dependencies between different parts of a sequence. This mechanism is fundamental to transformers and generative LLMs, as it allows each token to attend to other tokens within the same sequence or across different sequences, capturing contextual relationships effectively (see Figure 1).

In a generative LLM, each attention block operates on a transformed space of **Keys (K), Values (V), and Queries (Q)** and performs **scaled dot-product attention**, defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where d_k is the dimensionality of the key vectors, ensuring stable gradients during training.

The output of each attention block consists of **weighted representations of the input tokens**, where the weights are determined by the learned attention scores. These values are then passed through multiple stacked layers, forming deep hierarchical feature representations.

During **training**, generative decoder-only models use **causal self-attention**, meaning each token can only attend to previous tokens. This constraint, implemented via masking, ensures that the model learns to predict the next token without seeing future tokens.

During **inference**, which is the focus of this project, causal self-attention remains crucial, but the model no longer requires backpropagation. The inference process proceeds as follows:

1. The model receives an initial context (prompt) and generates the first token.
2. The generated token is appended to the context and used to predict the next token.
3. This process continues iteratively until an **end-of-sequence (EOS) token** is produced or a predefined length limit is reached.

A major computational challenge in inference is the repeated computation of **Keys (K) and Values (V)** for each new token. Since **K and V depend only on previous tokens and remain unchanged during generation**, recomputing them at every step is inefficient. To address this, a **KV cache** is used, storing previously computed **K and**

V values. This reduces redundant computation and improves inference speed, trading computational cost for memory usage.

The goal of this project is to explore the implementation of **KV caching** and develop optimized techniques to maximize the throughput of an inference engine based on a pre-trained generative decoder-only LLM.

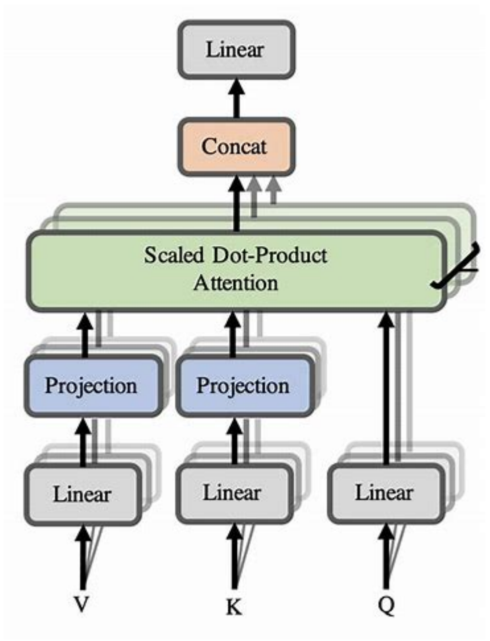

Figure 1: Multi-head attention mechanism. Each attention head independently computes attention scores, and the outputs are concatenated and projected to form the final attention representation.

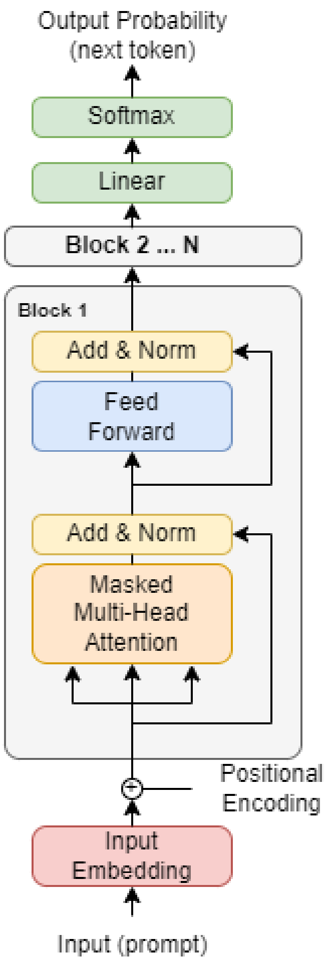

Figure 2: Decoder-only transformer architecture. The model consists of stacked self-attention and feedforward layers, generating tokens autoregressively.

KV-Caches in LLMs

In LLMs using long contexts increases the response generation latency significantly, as no response can be generated until the whole context is loaded and processed by the LLM. While some recent works increase the throughput of processing long context, the delay of processing the context can still be several seconds for long contexts (2 seconds for a 3K context). KV cache (Key-Value cache) in LLMs is a technique used to optimize inference speed by storing previously computed key and value representations of past tokens in the model's attention layers.

LMCache: Efficient Management of KV-Caches

While KV caching significantly improves inference latency, efficiently managing and scaling KV caches during inference remains a major challenge, particularly in systems constrained by GPU memory. **LMCache** is a specialized system designed to address this issue by enabling dynamic caching, retrieval, and eviction of KV tensors without redundant computation.

LMCache is composed of two main components:

- **LMCache Engine**: A local module integrated with the LLM inference engine, responsible for managing the GPU memory allocated for KV caches. The engine dynamically stores newly computed KV pairs, monitors memory usage, and evicts entries when necessary according to a cache policy. When an incoming request requires KV data that is no longer available locally, the engine coordinates retrieval from an external server to avoid full recomputation.
- **LMCache Server**: A lightweight remote storage server that acts as an extended memory for KV tensors. Evicted KV caches are offloaded asynchronously to the server, and can be retrieved on demand when required for future inference steps. The server enables efficient management of larger workloads and longer contexts by expanding the effective capacity of the system beyond the limits of GPU memory.

By decoupling KV cache management from direct inference execution, LMCache enables substantial improvements in system throughput and latency, particularly in workloads where context reuse is common. The design introduces a trade-off between memory, network communication, and computational savings, and provides new opportunities for scheduling, batching, and optimizing the interaction between the LLM engine and its cache hierarchy.

LMCache is particularly well-suited for high-throughput LLM serving scenarios, retrieval-augmented generation pipelines, and systems where inference over shared or repetitive contexts is frequent.

Project Description

The project focuses on implementing a GPU-based inferencing pipeline leveraging modern frameworks and tools. The GPU hosts a large language model from Hugging Face, such as Qwen-2.5B, which can be found in the literature and resources section. The main goal is to implement a scheduler that allows fast generation of LLM responses with infrequent KV cache swaps.

Tasks:

- **Task 1**: Evaluation of the LLM model inference performance with and without KV caching. Define metrics such as latency and the number of requests that can be processed. Analyze the impact of local cache size.

- **Task 2**: Scheduler implementation for batched requests. Requests should be ordered according the their context, so that requests with same context are served as sequentially as possible. Analyze performance when varying local cache size.
- **Task 3**: Implementation of RAG (retrieval-augmented generation), which will include:
    1. The creation of a database of contexts and a corresponding embedding space from provided texts (some samples can be found in Canvas).
    2. Implementation of similarity search between query and stored contexts.
    3. Implementation of a scheduler that orders classified requests from RAG in a way that requests with the same contexts are grouped sequentially to minimize redundant KV cache loads. Analyze performance when varying local cache size.
- **Task 4**: Analysis of KV cache similarities. Investigate potential similarities between caches using PCA and visualize results in a 2D space.

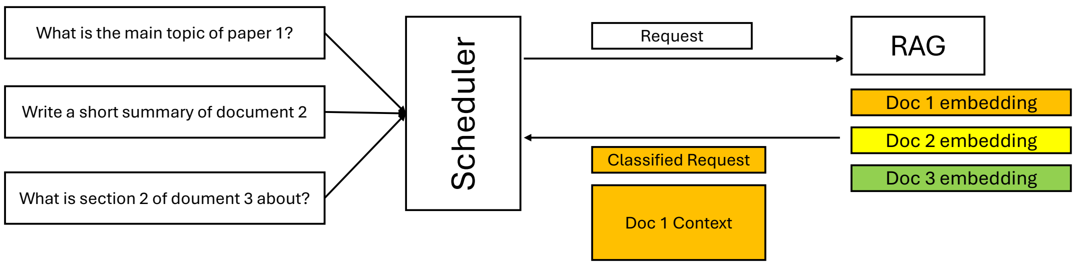
What is the main topic of paper 1?
Write a short summary of document 2
What is section 2 of doument 3 about?
Scheduler
Request
Classified Request
Doc 1 Context
RAG
Doc 1 embedding
Doc 2 embedding
Doc 3 embedding

Figure 3: RAG search

Task 1: LMCache and LMCache Server

In this task, you will run and analyze the behavior of the system implementation shown in Figure 4. A simple scheduler is already implemented for you, available at:

`https://github.com/hamidgh09/lmcache-vllm-extended/blob/ik2221/lmcache_vllm/custom_api.py`

This baseline scheduler simply forwards incoming requests to the model without any reordering.

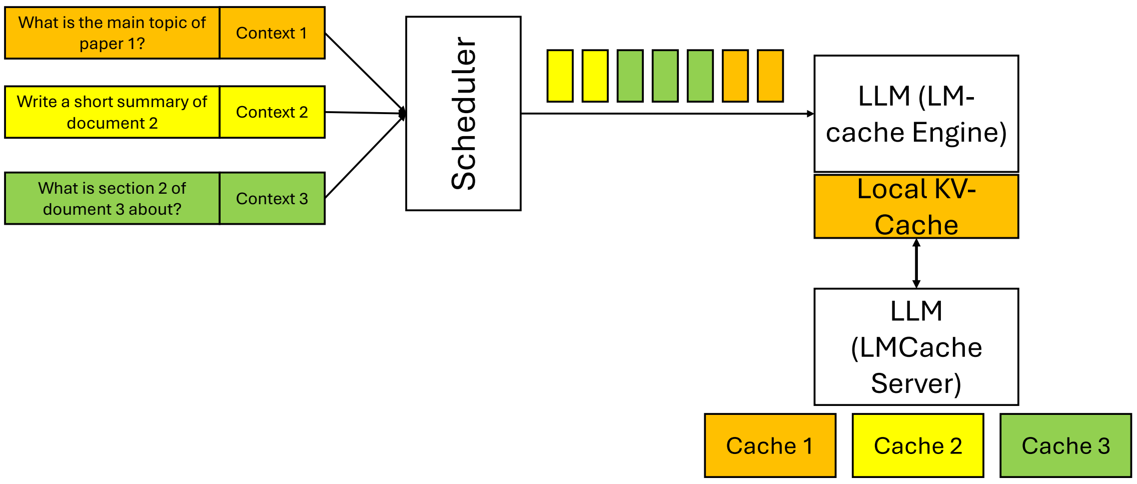

Figure 4: LMCache scheduling

First, install the required libraries as defined in:
`https://github.com/hamidgh09/lmcache-vllm-extended/tree/ik2221`
Ensure that inference from the frontend interface is working correctly before proceeding to implement the request generator and perform your analysis.

You are required to implement the following component:

- A **request generator** that creates a set of requests over multiple contexts provided in the resources. The generator should create a stream of unordered, different, requests that are handled sequentially to the LLM. Make sure that you know in advance which request corresponds to which context.

Resources: You can find a collection of `.txt` files related to the course papers on Canvas. These documents can be used as baseline contexts for your analysis and for subsequent tasks throughout the project. You can extend such set with additional contexts for your experiments.

Once your request generator is ready, you will analyze the performance of the inference pipeline by measuring:

- The number of handled requests per second (throughput).
- The request latency (response time).

while **varying the amount of allocated memory** in the local cache of the LLM model.

Questions to answer:

- What happens to the latency as the combined context + question length increases? Plot a graph showing the relationship between sequence length and response time.

- What happens if an old (previously seen) request is fed again to the model? Are the performances better in terms of latency? Analyze how the size of the local cache affects these results.
- What happens to the performance when the diversity (e.g., increasing the number of contexts, making the order of the requests more random) of the requests increases?

After completing your experiments and answering the questions, you should be able to determine the optimal amount of allocated memory needed to perform inference with low latency and high throughput. Once you have configured the system accordingly, proceed to the next task.

Task 2: Scheduler for batch request

In this task you will implement a more complex scheduler to handle batched requests. You will update your request generator to provide batches of requests instead of a request at a time to the scheduler. The scheduler should now handle such batched requests and order them in so that requests which share the same context are processed sequentially.

You will implement the following blocks:

- an extension of the request generator from the previous task that allows sending multiple requests in a single batch to the scheduler.
- a scheduler that orders the requests obtained in the batch in order to optimize the latency and throughput of the inferencing system.

Based on the metrics defined in the previous task, you will analyze the performance of your new system when you vary the amount of local cache:

- Does your scheduler improve the metrics you defined? Why?
- What happens if the local cache has a very high size? Does your scheduler has any impact in that case? How does the batch size of the grouped requests affect such impact?
- What happens when the diversity of the requests increases?
- What happens when requests have larger contexts sizes?

Task 3: RAG

RAG is a powerful tool to store embeddings and retrieve corresponding context based on similarities between query inputs and stored embeddings. As the original text for extracting embeddings, you will use summarized versions of the papers reviewed during the course (found in the resource section). You will generate the embeddings using any model (e.g., Llama or Qwen) from the Hugging Face library. For instance, you could use:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

def get_llm_embeddings(text, model, tokenizer):
    # Tokenize input text
    inputs = tokenizer(text, return_tensors="pt").to(device)

    with torch.no_grad():
        # Forward pass through the model to get hidden states
        outputs = model(**inputs, output_hidden_states=True)

    # Extract the last hidden state (embeddings)
    hidden_states = outputs.hidden_states
    last_hidden_state = hidden_states[-1]

    # Mean pooling for sentence-level embedding
    embeddings = last_hidden_state.mean(dim=1)

    return embeddings
```

Differently from the previous context, you cannot assume that the packets requests include the id of the document to be used as context for the LLM inferencing. Instead, the request should be sent to the RAG as shown in 3 and the context should be retrieved from that. In order to detect the most fitting context (i.e., the paper that the request is referring to) you can use cosine similarity or another distance-based metric on the embeddings of request prompt and available papers. The additional architecture blocks will consist of:

- A request generator based on the available summarized papers provided as input in the resources.
- A RAG database that maps summarized papers to their embeddings, computed with a Hugging Face transformer.
- A RAG search-and-retrieve mechanism that provides the most similar paper given a request, based on computed embeddings.

Analyze the performance of your RAG model and answer the following questions:

- What is the accuracy of the RAG module in detecting the correct context for the requests? Employ the information from the request generator to address this question.
- How does the RAG pipeline determine which document is most relevant to a given prompt?
- How does context retrieval time impact the overall inference latency?
- How does increasing the number of documents in the RAG database affect accuracy and response time?

Task 4 (Bonus): Clustering Requests Based on KV Cache Representations

In the previous task (RAG), the goal was to retrieve the correct document context (e.g., a paper ID) from the request prompt using semantic similarity in embedding space. In this task, we go a step further and investigate whether it is possible to identify the underlying topic or source document of a request **after inference**, using only the cached Key-Value (KV) tensors stored in the LMCache engine.

If similar requests produce similar internal representations, we can use those to cluster and group semantically related requests. This could enable downstream optimization strategies, such as intelligent cache reuse, request scheduling, or cache eviction prioritization, even when the original request is no longer available. You will have to perform the following steps.

- Extract the KV cache tensors for a set of processed requests. Ensure that the original request topic (e.g., paper ID) is known for evaluation purposes.
- For each request, compute a fixed-size vector representation by aggregating KV values. For instance, average across heads and/or layers, or apply layer weighting.
- Apply a dimensionality reduction method (e.g., PCA, Truncated SVD) to obtain a compact feature vector for each request. Note: do not limit the reduced space to 2D or 3D; select a dimensionality that retains meaningful structure.
- Perform clustering (e.g., KMeans, DBSCAN) on the reduced KV cache representations.
- Evaluate the quality of the clustering by comparing predicted cluster labels with the known request topics from the request generator or RAG labels.

Questions to Answer:

- Are requests referring to the same document or topic clustered together based solely on their KV cache?
- What are the performance of your KV-clustering method compared to the previous RAG implementation?
- How could this representation-based clustering be used to improve request scheduling or prefetching mechanisms in LLM inference pipelines?

Implementation Tools

- **PyTorch**: Used for computation of model embeddings
- **Hugging Face Transformers**: Provides pre-trained models such as Qwen and LLama.
- **Project repository**: provodies the instructions to run the LLMCache code, including VLLM, LMCache and custom API:

  `https://github.com/hamidgh09/lmcache-vllm-extended/blob/ik2221/`
- **LMCache server**: The server that stores the LLM KV caches, which you can install from here: `https://github.com/LMCache/lmcache-server`
- **LMCache engine**: The main component of LMCache, which includes also a local cache: `https://github.com/LMCache/LMCache`

Getting Started

You will use virtual environments such as *conda* (`https://www.anaconda.com/download/success`) or *venv* in this project.

Follow the instructions at `https://github.com/hamidgh09/lmcache-vllm-extended/blob/ik2221/lmcache_vllm/custom_api.py` to set up the three required repositories for this project. Also, install the necessary packages using *pip*, as mentioned in the repository.

You will need to run three scripts in separate terminals inside your virtual environment. For instance, open *tmux* by executing `tmux`. In each terminal, you should include the correct python path (if not already set) by running:

```
PYTHONPATH="$(pwd)/LMCache:$(pwd)/lmcache-vllm-extended:$PYTHONPATH"
export PYTHONPATH
```

Then, proceed as follows:

- In the first terminal, run the **lmcache server**:

  ```
  cd lmcache-server/
  python -m lmcache_server.server \
      192.168.2.XX 65432 ./
  ```

  Make sure to use the correct IP address and port, as specified in the `configuration.yaml` file and in `lmcache-vllm-extended/frontend/frontend.py`.

- In the second terminal, run the **lmcache engine**:

  ```
  cd lmcache-vllm-extended
  LMCACHE_CONFIG_FILE="configuration.yaml" \
  CUDA_VISIBLE_DEVICES=0 \
  python lmcache_vllm/script.py serve \
      Qwen/Qwen2.5-1.5B-Instruct \
      --gpu-memory-utilization 0.8 \
      --dtype half \
      --port 8000
  ```

- In the third terminal, run the **frontend** (note that you might need port forwarding if you want to access it from your local machine):

  ```
  streamlit run frontend.py
  ```

You should now be able to perform inference through the frontend. Please refer to the next section for troubleshooting instructions.

Technical details

Make sure that you are using Python version greater than 3.10, for instance Python 3.12. You can check your current Python version by running:

```
python --version
```

or

```
python3 --version
```

On the server where you have received an account, you will find a Python 3.12 installation available under `/usr/bin` or `/usr/local/bin`.

To create an isolated environment, you have two options:

- Using **venv** (recommended for simplicity, as it does not require any additional installation):

  ```
  python3.12 -m venv venv
  source venv/bin/activate
  ```

- Alternatively, using **conda** (requires conda to be installed):

  ```
  conda create --name ik2221-env python=3.12 -y
  conda activate ik2221-env
  ```

Remember to add your data folder inside the frontend subdirectory of `lmcache-vllm-extended`. For instance: `data -> paper summaries from Canvas in txt format`.

When you need to implement batched requests and reordering, you will most likely have to update the function `create_chat_completion` located in:

```
lmcache-vllm-extended/lmcache_vllm/custom_api.py
```

You must also configure the correct IP addresses:

- In the `lmcache-vllm-extended` repository, edit the `configuration.yaml` file by updating the `remote_url` field:

  ```
  remote_url: "lm://192.168.2.XX:65432"
  ```

  where `XX` corresponds to your assigned server (e.g., 27 for server `nslrack27`).

- When running the server, you must match the IP address and port:

  ```
  python -m lmcache_server.server 192.168.2.27 65432 ./
  ```

- In the frontend script `lmcache-vllm-extended/frontend/frontend.py`, update the IP address accordingly:

  ```
  IP1 = "192.168.2.27"
  ```

Troubleshooting Guide

Git-related problems: Install repositories with HTTPS instead of SSH

Use the following commands:

```
git clone https://github.com/LMCache/LMCache.git
cd LMCache || return
git checkout v0.1.4-alpha
cd ..

git clone https://github.com/LMCache/lmcache-server.git
cd lmcache-server || return
git checkout v0.1.1-alpha
cd ..

git clone https://github.com/hamidgh09/lmcache-vllm-extended.git
```

Cannot find `cuda.h` or `nvcc` when installing pip requirements or building `torchac_cuda`

1. Create `.bashrc` if it does not exist:

   ```
   touch ~/.bashrc
   ```

2. Open `.bashrc` in a text editor:

   ```
   nano ~/.bashrc
   ```

3. Add the following lines to the file:

   ```
   # --- CUDA environment setup ---
   export PATH=/usr/local/cuda/bin:$PATH
   export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
   export CUDA_HOME=/usr/local/cuda
   export TORCH_CUDA_ARCH_LIST="7.5"
   ```

4. Apply the updated `.bashrc`:

   ```
   source ~/.bashrc
   ```

404 errors when sending requests to LMCache (Cannot see v2 API)

This is likely due to a misconfigured `PYTHONPATH`. Set it properly by running:

```
PYTHONPATH="$(pwd)/LMCache:$(pwd)/lmcache-vllm-extended:$PYTHONPATH"
export PYTHONPATH
```

`torchac_cuda` installation error

If you encounter problems with `torchac_cuda`, reinstall it manually:

```
python -m pip uninstall torchac-cuda -y
git clone https://github.com/LMCache/torchac_cuda.git
cd torchac_cuda
python setup.py install
```

SQLite3 errors

If you encounter errors related to SQLite3:

- First, try installing through pip inside your virtual environment:

  ```
  pip install sqlite3
  ```

- If that does not work, contact a Teaching Assistant (TA) for assistance. Students typically do not have `sudo` permissions. The TA can install the required library using:

  ```
  sudo apt-get -y install libsqlite3-dev
  ```

  Afterward, Python may need to be reconfigured to link against the updated SQLite3 library.