```
%%capture
import torch
major_version, minor_version =
torch.cuda.get_device_capability()
# Must install separately since Colab has torch 2.2.1, which
breaks packages
!pip install "unsloth[colab-new] @ git+https://github.com/
unslothai/unsloth.git"
if major_version >= 8:
# Use this for new GPUs like Ampere, Hopper GPUs (RTX 30xx, RTX
40xx, A100, H100, L40)
    !pip install --no-deps packaging ninja einops flash-attn
xformers trl peft accelerate bitsandbytes
else:
    # Use this for older GPUs (V100, Tesla T4, RTX 20xx)
    !pip install --no-deps xformers trl peft accelerate
bitsandbytes
pass
```

This snippet installs the necessary packages based on the GPU version in Google Colab:

1. **torch.cuda.get_device_capability:** Retrieves the compute capability (major_version and minor_version) of the current GPU to determine its generation.

Version features and specifications [edit]

<table>
<tr><th rowspan="2">Feature support (unlisted features are supported for all compute capabilities)</th><th colspan="13">Compute capability (version)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th><th>6.x</th><th>7.x</th><th>8.0</th><th>8.6</th></tr>
<tr><td>Integer atomic functions operating on 32-bit words in global memory</td><td rowspan="2">No</td><td colspan="12" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory</td></tr>
<tr><td>Integer atomic functions operating on 32-bit words in shared memory</td><td colspan="2" rowspan="4">No</td><td colspan="11" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory</td></tr>
<tr><td>Integer atomic functions operating on 64-bit words in global memory</td></tr>
<tr><td>Warp vote functions</td></tr>
<tr><td>Double-precision floating-point operations</td><td colspan="3">No</td><td colspan="10">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory</td><td colspan="4" rowspan="7">No</td><td colspan="9" rowspan="7">Yes</td></tr>
<tr><td>Floating-point atomic addition operating on 32-bit words in global and shared memory</td></tr>
<tr><td>_ballot()</td></tr>
<tr><td>_threadfence_system()</td></tr>
<tr><td>_syncthreads_count(), _syncthreads_and(), _syncthreads_or()</td></tr>
<tr><td>Surface functions</td></tr>
<tr><td>3D grid of thread block</td></tr>
<tr><td>Warp shuffle functions , Unified Memory</td><td colspan="5">No</td><td colspan="8">Yes</td></tr>
<tr><td>Funnel shift</td><td colspan="6">No</td><td colspan="7">Yes</td></tr>
<tr><td>Dynamic parallelism</td><td colspan="7">No</td><td colspan="6">Yes</td></tr>
<tr><td>Half-precision floating-point operations:
addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="8">No</td><td colspan="5">Yes</td></tr>
<tr><td>Atomic addition operating on 64-bit floating point values in global memory and shared memory</td><td colspan="9">No</td><td colspan="4">Yes</td></tr>
<tr><td>Tensor core</td><td colspan="10">No</td><td colspan="3">Yes</td></tr>
<tr><td>Mixed Precision Warp-Matrix Functions</td><td colspan="10">No</td><td colspan="3">Yes</td></tr>
<tr><td>Hardware-accelerated async-copy</td><td colspan="11">No</td><td colspan="2">Yes</td></tr>
<tr><td>Hardware-accelerated Split Arrive/Wait Barrier</td><td colspan="11">No</td><td colspan="2">Yes</td></tr>
<tr><td>L2 Cache Residency Management</td><td colspan="11">No</td><td colspan="2">Yes</td></tr>
</table>

2. **Installing** unsloth: Installs the unsloth package from a GitHub repository, specifically configured for Colab (colab-new).
3. **Conditional package installation**:
   New GPUs (Ampere, Hopper): Installs additional dependencies like packaging, flash-attn, and bitsandbytes, which are optimized for newer GPU architectures (RTX 30xx, RTX 40xx, etc.).
   Older GPUs: A lighter set of dependencies is installed, compatible with older GPUs like V100 or Tesla T4.

The %capture magic suppresses the output of these commands in Jupyter/Colab cells.

```
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048 # Choose any! We auto support RoPE Scaling
internally!
dtype = None # None for auto detection. Float16 for Tesla T4,
V100, Bfloat16 for Ampere+
load_in_4bit = True # Use 4bit quantization to reduce memory
usage. Can be False.
# 4bit pre quantized models we support for 4x faster downloading
+ no OOMs.
# fourbit_models = [
#     "unsloth/mistral-7b-bnb-4bit",
#     "unsloth/mistral-7b-instruct-v0.2-bnb-4bit",
#     "unsloth/llama-2-7b-bnb-4bit",
#     "unsloth/gemma-7b-bnb-4bit",
#     "unsloth/gemma-7b-it-bnb-4bit", # Instruct version of
Gemma 7b
#     "unsloth/gemma-2b-bnb-4bit",
#     "unsloth/gemma-2b-it-bnb-4bit", # Instruct version of
Gemma 2b
#     "unsloth/llama-3-8b-bnb-4bit", # [NEW] 15 Trillion token
Llama-3
# ] # More models at https://huggingface.co/unsloth

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/llama-3-8b-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
    # token = "hf_...", # use one if using gated models like
meta-llama/Llama-2-7b-hf
```

)
This snippet demonstrates how to load a quantized language model using the unsloth library:

1. **Importing FastLanguageModel:** FastLanguageModel is a utility for efficiently loading and working with language models, including those with quantization optimizations.

2. **Configuration Parameters:**
   - **max_seq_length:** Maximum sequence length the model can process. RoPE scaling is supported internally for larger inputs.
   - **dtype:** Data type for computation. It is auto-detected unless manually specified (float16 for older GPUs, bfloat16 for newer ones like Ampere).
   - **load_in_4bit:** Enables 4-bit quantization, significantly reducing memory usage while maintaining decent performance.

3. **Pre-Quantized Models:**
   - The comment lists several 4-bit pre-quantized models available on Hugging Face (e.g., llama-2-7b-bnb-4bit, gemma-7b-it-bnb-4bit).
   - These models download faster and avoid out-of-memory (OOM) issues.

4. **Loading the Model:**
   - **model_name:** Specifies the model to load (unsloth/llama-3-8b-bnb-4bit in this case).
   - **Other Parameters:** max_seq_length, dtype, and load_in_4bit customize the model's behavior. A Hugging Face token (token) may be required for gated models.

```
model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Choose any number > 0 ! Suggested 8, 16, 32, 64
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0, # Supports any, but = 0 is optimized
    bias = "none",    # Supports any, but = "none" is optimized
    # "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
    use_gradient_checkpointing = “unsloth",
     # True or "unsloth" for very long context
    random_state = 3407,
    use_rslora = False,  # We support rank stabilized LoRA
```

```
    loftq_config = None)
```

PEFT (Parameter-Efficient Fine-Tuning) approaches only fine-tune a small number of (extra) model parameters while freezing most parameters of the pre-trained LLMs, thereby greatly decreasing the computational and storage costs. This also overcomes the issues of catastrophic forgetting, a behavior observed during the full fine-tuning of LLMs. PEFT approaches have also shown to be better than fine-tuning in the low-data regimes and generalize better to out-of-domain scenarios. It can be applied to various modalities, e.g., image classification and stable diffusion dreambooth.

LoRA (Low-Rank Adaptation of Large Language Models) is a lightweight training technique that significantly reduces the number of trainable parameters. It works by inserting a smaller number of new weights into the model and only these are trained. This makes training with LoRA much faster, memory-efficient, and produces smaller model weights (a few hundred MBs), which are easier to store and share. LoRA can also be combined with other training techniques like DreamBooth to speedup training.

This snippet configures Parameter-Efficient Fine-Tuning (PEFT) for the loaded model using LoRA (Low-Rank Adaptation):

Key Parameters:

1. **r:** The rank of the low-rank adaptation matrix (e.g., 16). Higher values increase flexibility but require more memory.

2. **target_modules:** Specifies which layers (e.g., q_proj, v_proj, etc.) will be fine-tuned using LoRA, targeting key projection layers of the transformer.

3. **lora_alpha:** A scaling factor controlling the impact of LoRA updates. Larger values make adaptation stronger.

4. **lora_dropout:** Dropout applied during fine-tuning to improve generalization. 0 optimizes memory and speed.

5. **bias:** Specifies bias term handling. "none" is the most memory-efficient option.

6. **use_gradient_checkpointing:** "unsloth" reduces VRAM usage by ~30% for long sequences, allowing larger batch sizes.
7. **random_state:** Ensures reproducibility during fine-tuning.

8. **use_rslora:** Enables Rank Stabilized LoRA (optional), which improves robustness for certain tasks.

9. **loftq_config:** Optional integration with LoFT-Q (LoRA with quantization), combining memory efficiency with quantization.

This configuration applies LoRA for lightweight and efficient fine-tuning, ideal for adapting large models with limited resources.

```
alpaca_prompt = """Below is an instruction that describes a
task, paired with an input that provides further context. Write
a response that appropriately completes the request.

### Instruction:
{}

### Input:
{}

### Response:
{}"""

EOS_TOKEN = tokenizer.eos_token # Must add EOS_TOKEN
def formatting_prompts_func(examples):
    instructions = examples["instruction"]
    inputs       = examples["input"]
    outputs      = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs,
outputs):

        text = alpaca_prompt.format(instruction, input, output)
+ EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
pass

from datasets import load_dataset
dataset = load_dataset("yahma/alpaca-cleaned", split = "train")
```

```
dataset = dataset.map(formatting_prompts_func, batched = True,)
```

This snippet prepares a dataset for fine-tuning a language model using Alpaca-style prompts:

Key Steps:

1. **alpaca_prompt:** Template for formatting data. Includes placeholders for Instruction, Input, and Response to create a complete training example.

2. **EOS_TOKEN:** Ensures every example ends with the tokenizer's end-of-sequence token. Prevents infinite generation during inference.

3. formatting_prompts_func:
- Function to format the dataset:
- Extracts instructions, inputs, and outputs from the dataset.
- Fills the alpaca_prompt with these values.
- Appends EOS_TOKEN to each formatted text for proper termination.

4. Dataset Handling:
- load_dataset: Loads the cleaned Alpaca dataset (yahma/alpaca-cleaned), a high-quality instruction-following dataset.
- map: Applies formatting_prompts_func to format all training examples with Alpaca-style prompts.

Output:

The formatted dataset includes a text field containing properly formatted examples with the Alpaca prompt structure and end-of-sequence token. These are ready for fine-tuning the language model.

```
#@title Show current memory stats
gpu_stats = torch.cuda.get_device_properties(0)
start_gpu_memory = round(torch.cuda.max_memory_reserved() / 1024
/ 1024 / 1024, 3)
max_memory = round(gpu_stats.total_memory / 1024 / 1024 / 1024,
3)
print(f"GPU = {gpu_stats.name}. Max memory = {max_memory} GB.")
print(f"{start_gpu_memory} GB of memory reserved.")
trainer_stats = trainer.train()
```

(WandB API key might be asked in prompt)

Weights & Biases (WandB) is a tool for machine learning experiment tracking and model management. It helps track and visualize:

1. **Metrics:** Loss, accuracy, learning rate, etc.
2. **Hyperparameters:** Changes and their impact.
3. **Logs:** Training progress in real-time.
4. **Model versions:** Save and compare model checkpoints.
5. **Collaborations:** Share experiments with teams.

It's widely used for experiment reproducibility and integrates with popular ML frameworks like PyTorch, TensorFlow, and Hugging Face.

```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster
inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64,
use_cache = True)
tokenizer.batch_decode(outputs)
```

This snippet generates text from the fine-tuned language model using Alpaca-style prompting. Here's the breakdown:

Preparation:

1. **FastLanguageModel.for_inference(model):** Optimizes the model for inference by enabling faster decoding (2x speed-up).

2. **alpaca_prompt:** A structured prompt template (defined earlier) used to frame the task for the model.

3. **tokenizer():**
   - Converts the input text into tokenized tensors:

- return_tensors = "pt": Returns tensors in PyTorch format.
- .to("cuda"): Moves the tokenized input to the GPU for faster computation.

Input Prompt:

- Instruction: "Continue the Fibonacci sequence."
- Input: "1, 1, 2, 3, 5, 8"
- Output: Left blank ("") to let the model generate it.

Generation:

4. model.generate():

- Generates text based on the input tokens.
- inputs: The tokenized prompt as input.
- max_new_tokens = 64: Limits the generated output to 64 tokens.
- use_cache = True: Speeds up generation by caching key-value pairs of past attention layers.

Decoding:

5. tokenizer.batch_decode(outputs):

- Converts the generated tokens back into human-readable text.
- Outputs a list of decoded strings, where each entry corresponds to a generated response.

Purpose:

This snippet frames a task (Fibonacci continuation), tokenizes it, generates the completion, and decodes the result into readable text. It uses optimized inference settings for faster and efficient output generation.

```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster
inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")
```

```python
from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer,
max_new_tokens = 128)
model.save_pretrained("lora_model") # Local saving
```