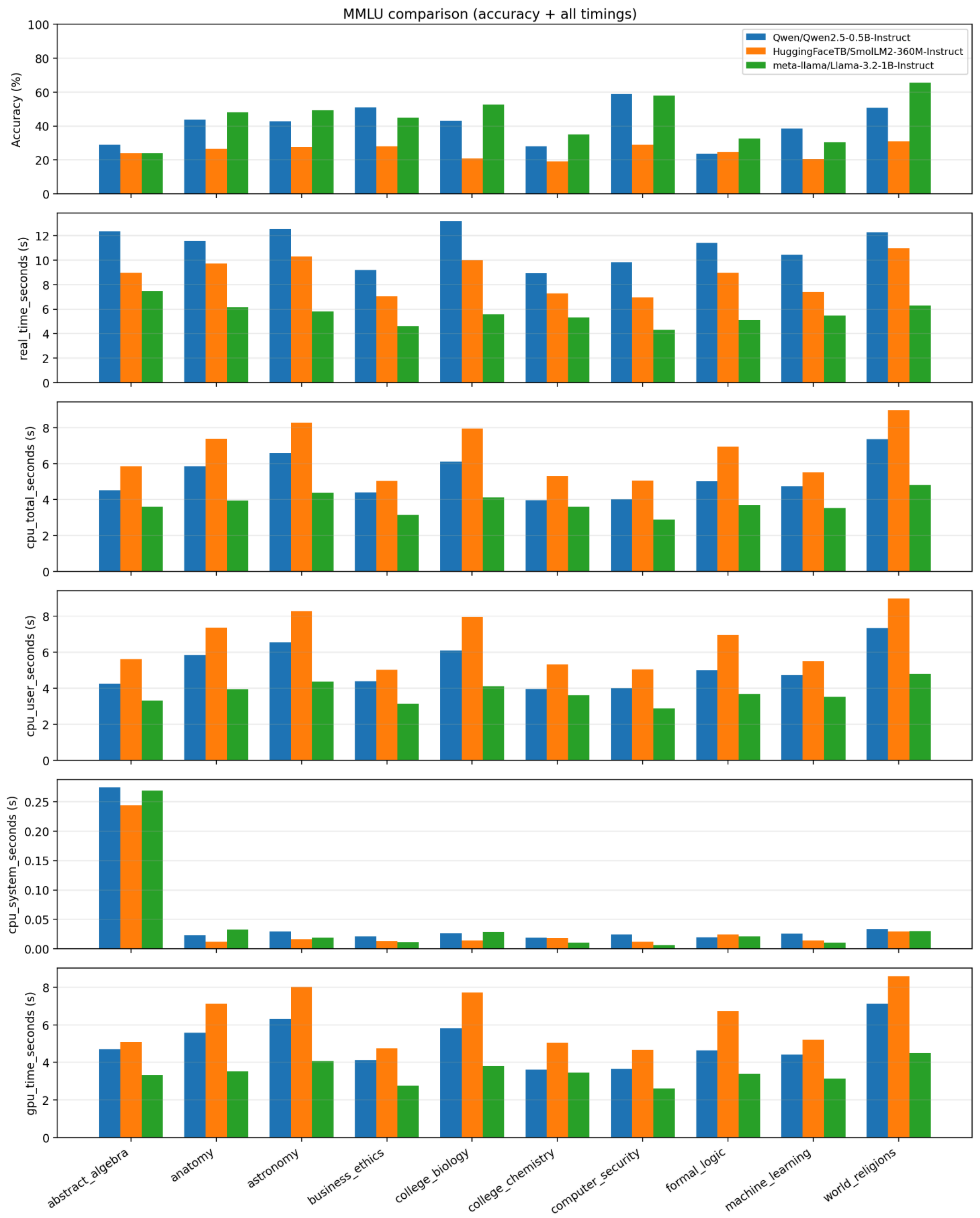
Topic 1: Running an LLM

Introduction

This report presents a comparative analysis of three Small Language Models (SLMs): meta-llama/Llama-3.2-1B-Instruct, HuggingFaceTB/SmolLM2-360M-Instruct, and Qwen/Qwen2.5-0.5B-Instruct. The primary objective is to evaluate their performance across ten distinct subsets of the Massive Multitask Language Understanding (MMLU) benchmark. Furthermore, the report includes a dedicated comparative study on the impact of quantization techniques applied to the meta-llama/Llama-3.2-1B-Instruct model, specifically focusing on the 'Abstract Algebra' subject within the MMLU dataset.

Models Comparison



I compare three models:

- meta-llama/Llama-3.2-1B-Instruct
- HuggingFaceTB/SmolLM2-360M-Instruct
- Qwen/Qwen2.5-0.5B-Instruct

On ten subjects:

- Abstract_algebra
- Anatomy
- Astronomy
- Business_ethics
- College_biology
- College_chemistry
- Computer_security
- Formal_logic
- Machine_learning
- World_religions

Overall, llama has the highest accuracy and the fastest inference.

Code

Drawing image:

```
python plot_mmlu_compare.py --all-timings --out
compare_3models_all_timings.png
llama_3.2_1b_mmlu_results_full_20260123_213230.json
llama_3.2_1b_mmlu_results_full_20260123_211656.json
llama_3.2_1b_mmlu_results_full_20260123_195436.json
```

```
import argparse
import json
import os
from collections import Counter

DEFAULT_TIMING_KEYS = [
    "real_time_seconds",
    "cpu_total_seconds",
    "cpu_user_seconds",
    "cpu_system_seconds",
    "gpu_time_seconds",
]


def _load_results(path):
    with open(path, "r", encoding="utf-8") as f:
        data = json.load(f)

    subject_results = data.get("subject_results") or []
    subjects_in_order = [r.get("subject") for r in subject_results
if r.get("subject")]
```

```
    subject_map = {r.get("subject"): r for r in subject_results if
r.get("subject")}

    model = data.get("model", "model")
    ts = data.get("timestamp") or os.path.basename(path)
    quant = data.get("quantization_bits")
    device = data.get("device")

    return {
        "path": path,
        "label_base": model,
        "timestamp": ts,
        "quant": quant,
        "device": device,
        "subjects_in_order": subjects_in_order,
        "subject_map": subject_map,
        "overall_accuracy": data.get("overall_accuracy"),
        "timing": data.get("timing") or {},
    }


def _make_unique_labels(runs):
    base_counts = Counter(r["label_base"] for r in runs)
    labels = []
    for r in runs:
        if base_counts[r["label_base"]] == 1:
            labels.append(r["label_base"])
        else:
            labels.append(f"{r['label_base']} ({r['timestamp']})")
    return labels


def _get_subject_list(runs, *, subjects=None):
    if subjects:
        return subjects
    first = runs[0]["subjects_in_order"]
    if not first:
        union = sorted({s for r in runs for s in
r["subjects_in_order"]})
        return union
    return first


def create_mmlu_model_comparison_chart(
    json_paths,
    *,
    output_path=None,
    subjects=None,
    timing_key="real_time_seconds",
    title=None,
):
    """
    Create one PNG comparing multiple runs/models:
    - Subplot 1: per-subject accuracy (%)
    - Subplot 2: per-subject timing (seconds) using `timing_key`

    Works with the JSON produced by `llama_mmlu_eval.py`.
    """
    try:
        import matplotlib

        matplotlib.use("Agg")
        import matplotlib.pyplot as plt
    except ModuleNotFoundError as exc:
        raise SystemExit(
            "Missing dependency: matplotlib\n\nInstall it with:\n"
            "  python -m pip install matplotlib\n"
        ) from exc

    if len(json_paths) < 2:
        raise ValueError("Pass 2+ JSON files to compare.")

    runs = [_load_results(p) for p in json_paths]
    labels = _make_unique_labels(runs)
    subject_list = _get_subject_list(runs, subjects=subjects)
    if not subject_list:
        raise ValueError("No subjects found in the provided JSON
files.")

    if output_path is None:
        output_path = "mmlu_compare.png"

    if title is None:
        title = "MMLU comparison (accuracy + timing)"

    n_models = len(runs)
    n_subjects = len(subject_list)
    x = list(range(n_subjects))
    width = min(0.8 / max(1, n_models), 0.25)

    fig_width = max(12, n_subjects * 0.7)
    fig_height = 9
    fig, (ax_acc, ax_time) = plt.subplots(nrows=2, ncols=1,
figsize=(fig_width, fig_height), sharex=True)
    fig.suptitle(title)

    for model_index, (run, label) in enumerate(zip(runs, labels)):
        offset = (model_index - (n_models - 1) / 2) * width
```

```python
        xs = [xi + offset for xi in x]

        acc = []
        tsec = []
        for subject in subject_list:
            sr = run["subject_map"].get(subject) or {}
            acc.append(float(sr.get("accuracy", 0.0) or 0.0))
            timing = sr.get("timing") or {}
            val = timing.get(timing_key)
            tsec.append(float(val) if val is not None else 0.0)

        ax_acc.bar(xs, acc, width=width, label=label)
        ax_time.bar(xs, tsec, width=width, label=label)

    ax_acc.set_ylabel("Accuracy (%)")
    ax_acc.set_ylim(0, 100)
    ax_acc.grid(axis="y", alpha=0.25)
    ax_acc.legend(loc="upper right", fontsize="small")

    ax_time.set_ylabel(f"{timing_key} (s)")
    ax_time.grid(axis="y", alpha=0.25)

    ax_time.set_xticks(x)
    ax_time.set_xticklabels(subject_list, rotation=35, ha="right")

    fig.tight_layout()
    fig.savefig(output_path, bbox_inches="tight", dpi=200)
    return output_path


def create_mmlu_model_comparison_all_timings_chart(
    json_paths,
    *,
    output_path=None,
    subjects=None,
    timing_keys=None,
    title=None,
):
    """
    Create one PNG comparing multiple runs/models:
    - Subplot 1: per-subject accuracy (%)
    - Subplots 2..N: per-subject timings (seconds) for all timing
keys
    """
    try:
        import matplotlib

        matplotlib.use("Agg")
        import matplotlib.pyplot as plt
    except ModuleNotFoundError as exc:
        raise SystemExit(
            "Missing dependency: matplotlib\n\nInstall it with:\n"
            "  python -m pip install matplotlib\n"
        ) from exc

    if len(json_paths) < 2:
        raise ValueError("Pass 2+ JSON files to compare.")

    runs = [_load_results(p) for p in json_paths]
    labels = _make_unique_labels(runs)
    subject_list = _get_subject_list(runs, subjects=subjects)
    if not subject_list:
        raise ValueError("No subjects found in the provided JSON
files.")

    if timing_keys is None:
        timing_keys = list(DEFAULT_TIMING_KEYS)

    if output_path is None:
        output_path = "mmlu_compare_all_timings.png"

    if title is None:
        title = "MMLU comparison (accuracy + all timings)"

    n_models = len(runs)
    n_subjects = len(subject_list)
    x = list(range(n_subjects))
    width = min(0.8 / max(1, n_models), 0.25)

    nrows = 1 + len(timing_keys)
    fig_width = max(12, n_subjects * 0.7)
    fig_height = 2.2 * nrows + 2.0
    fig, axes = plt.subplots(nrows=nrows, ncols=1,
figsize=(fig_width, fig_height), sharex=True)
    fig.suptitle(title)

    ax_acc = axes[0]
    for model_index, (run, label) in enumerate(zip(runs, labels)):
        offset = (model_index - (n_models - 1) / 2) * width
        xs = [xi + offset for xi in x]
        acc = []
        for subject in subject_list:
            sr = run["subject_map"].get(subject) or {}
            acc.append(float(sr.get("accuracy", 0.0) or 0.0))
        ax_acc.bar(xs, acc, width=width, label=label)

    ax_acc.set_ylabel("Accuracy (%)")
    ax_acc.set_ylim(0, 100)
    ax_acc.grid(axis="y", alpha=0.25)
    ax_acc.legend(loc="upper right", fontsize="small")

    for plot_index, timing_key in enumerate(timing_keys, start=1):
        ax = axes[plot_index]
        for model_index, (run, label) in enumerate(zip(runs,
labels)):
            offset = (model_index - (n_models - 1) / 2) * width
            xs = [xi + offset for xi in x]
            vals = []
            for subject in subject_list:
                sr = run["subject_map"].get(subject) or {}
                timing = sr.get("timing") or {}
                v = timing.get(timing_key)
                vals.append(float(v) if v is not None else 0.0)
            ax.bar(xs, vals, width=width, label=label)

        ax.set_ylabel(f"{timing_key} (s)")
        ax.grid(axis="y", alpha=0.25)

    axes[-1].set_xticks(x)
    axes[-1].set_xticklabels(subject_list, rotation=35, ha="right")

    fig.tight_layout()
    fig.savefig(output_path, bbox_inches="tight", dpi=200)
    return output_path


def main():
    parser = argparse.ArgumentParser(description="Compare multiple
MMLU results JSON files in one chart.")
    parser.add_argument("json_paths", nargs="+", help="2+ result
JSON files (from llama_mmlu_eval.py).")
    parser.add_argument("--out", default=None, help="Output PNG path
(default: mmlu_compare.png).")
    parser.add_argument(
        "--timing-key",
        default="real_time_seconds",
        help="Which per-subject timing field to plot (default:
%(default)s).",
    )
    parser.add_argument(
        "--all-timings",
        action="store_true",
        help="Plot accuracy plus all timing keys in one PNG.",
    )
    parser.add_argument(
        "--subjects",
        nargs="+",
        default=None,
        help="Optional subject list to plot (otherwise uses the
first JSON's subject order).",
    )
    args = parser.parse_args()

    if args.all_timings:
        out = create_mmlu_model_comparison_all_timings_chart(
            args.json_paths,
            output_path=args.out,
            subjects=args.subjects,
        )
    else:
        out = create_mmlu_model_comparison_chart(
            args.json_paths,
            output_path=args.out,
            subjects=args.subjects,
            timing_key=args.timing_key,
        )
    print(out)


if __name__ == "__main__":
    main()
```

My MMLU Evaluation Script:

```
python llama_mmlu_eval.py --gpu --quant none --model
meta-llama/Llama-3.2-1B-Instruct --subjects abstract_algebra anatomy
astronomy business_ethics college_biology college_chemistry
computer_security formal_logic machine_learning world_religions
--verbose
python llama_mmlu_eval.py --gpu --quant none --model
HuggingFaceTB/SmolLM2-360M-Instruct --subjects abstract_algebra anatomy
astronomy business_ethics college_biology college_chemistry
computer_security formal_logic machine_learning world_religions
--verbose
python llama_mmlu_eval.py --gpu --quant none --model
Qwen/Qwen2.5-0.5B-Instruct --subjects abstract_algebra anatomy
astronomy business_ethics college_biology college_chemistry
computer_security formal_logic machine_learning world_religions
--verbose
```

```
"""
Llama 3.2-1B MMLU Evaluation Script (Laptop Optimized with
Quantization)

This script evaluates Llama 3.2-1B on the MMLU benchmark.
Optimized for laptops with 4-bit or 8-bit quantization to reduce
memory usage.
```

```
Quantization options:
- 4-bit: ~1.5 GB VRAM/RAM (default for laptop)
- 8-bit: ~2.5 GB VRAM/RAM
- No quantization: ~5 GB VRAM/RAM

Usage:
1. Install: pip install transformers torch datasets accelerate tqdm
bitsandbytes
2. Login: huggingface-cli login
3. Run: python llama_mmlu_eval_quantized.py

Set QUANTIZATION_BITS below to choose quantization level.
"""

import argparse
from datetime import datetime
import json
import os
import platform
import sys
import time
import resource

torch = None
AutoTokenizer = None
AutoModelForCausalLM = None
BitsAndBytesConfig = None
load_dataset = None
tqdm = None

#
====================================================================
========
# CONFIGURATION - Modify these settings
#
====================================================================
========

MODEL_NAME = "meta-llama/Llama-3.2-1B-Instruct"

# GPU settings
# If True, will attempt to use the best available GPU (CUDA for
NVIDIA, MPS for Apple Silicon)
# If False, will always use CPU regardless of available hardware
USE_GPU = True  # Set to False to force CPU-only execution

MAX_NEW_TOKENS = 1
```

```
# Quantization settings
# Options: 4, 8, or None (default is None for full precision)
#
# To enable quantization, change QUANTIZATION_BITS to one of the
following:
#   QUANTIZATION_BITS = 4   # 4-bit quantization: ~1.5 GB memory
(most memory efficient)
#   QUANTIZATION_BITS = 8   # 8-bit quantization: ~2.5 GB memory
(balanced quality/memory)
#   QUANTIZATION_BITS = None  # No quantization: ~5 GB memory (full
precision, best quality)
#
# Notes:
# - Quantization requires the 'bitsandbytes' package: pip install
bitsandbytes
# - Quantization only works with CUDA (NVIDIA GPUs), not with Apple
Metal (MPS)
# - If using Apple Silicon, quantization will be automatically
disabled

QUANTIZATION_BITS = None  # Change to 4 or 8 to enable quantization

# For quick testing, you can reduce this list
MMLU_SUBJECTS = [
    "abstract_algebra", "anatomy",
    "astronomy", "business_ethics",
    "clinical_knowledge", "college_biology", "college_chemistry",
    "college_computer_science", "college_mathematics",
"college_medicine",
    # "college_physics", "computer_security", "conceptual_physics",
    # "econometrics", "electrical_engineering",
"elementary_mathematics",
    # "formal_logic", "global_facts", "high_school_biology",
    # "high_school_chemistry", "high_school_computer_science",
    # "high_school_european_history", "high_school_geography",
    # "high_school_government_and_politics",
"high_school_macroeconomics",
    # "high_school_mathematics", "high_school_microeconomics",
    # "high_school_physics", "high_school_psychology",
"high_school_statistics",
    # "high_school_us_history", "high_school_world_history",
"human_aging",
    # "human_sexuality", "international_law", "jurisprudence",
    # "logical_fallacies", "machine_learning", "management",
"marketing",
    # "medical_genetics", "miscellaneous", "moral_disputes",
"moral_scenarios",
    # "nutrition", "philosophy", "prehistory",
"professional_accounting",
    # "professional_law", "professional_medicine",
"professional_psychology",
    # "public_relations", "security_studies", "sociology",
"us_foreign_policy",
    # "virology", "world_religions"
]

def import_deps():
    global torch, AutoTokenizer, AutoModelForCausalLM,
BitsAndBytesConfig, load_dataset, tqdm
    try:
        import torch as _torch
        from transformers import AutoTokenizer as _AutoTokenizer
        from transformers import AutoModelForCausalLM as
_AutoModelForCausalLM
        from transformers import BitsAndBytesConfig as
_BitsAndBytesConfig
        from datasets import load_dataset as _load_dataset
        from tqdm.auto import tqdm as _tqdm
    except ModuleNotFoundError as exc:
        missing = getattr(exc, "name", "a required package")
        print(
            f"Missing dependency: {missing}\n\n"
            "Install required packages, then re-run:\n"
            "  python -m pip install transformers torch datasets
accelerate tqdm huggingface_hub bitsandbytes\n",
            file=sys.stderr,
        )
        raise SystemExit(1) from exc

    torch = _torch
    AutoTokenizer = _AutoTokenizer
    AutoModelForCausalLM = _AutoModelForCausalLM
    BitsAndBytesConfig = _BitsAndBytesConfig
    load_dataset = _load_dataset
    tqdm = _tqdm


def _cpu_times_seconds():
    usage = resource.getrusage(resource.RUSAGE_SELF)
    return float(usage.ru_utime), float(usage.ru_stime)


class TimingAccumulator:
    def __init__(self, *, enable_gpu_timing=False):
        self.enable_gpu_timing = bool(enable_gpu_timing)
        self.wall_start = None
        self.wall_end = None
        self.cpu_user_start = None
        self.cpu_sys_start = None
        self.cpu_user_end = None
        self.cpu_sys_end = None
        self.gpu_ms = 0.0

    def start(self):
        self.wall_start = time.perf_counter()
        self.cpu_user_start, self.cpu_sys_start =
_cpu_times_seconds()

    def stop(self):
        self.wall_end = time.perf_counter()
        self.cpu_user_end, self.cpu_sys_end = _cpu_times_seconds()

    def add_gpu_ms(self, ms):
        if self.enable_gpu_timing:
            self.gpu_ms += float(ms)

    def to_dict(self):
        wall = None
        cpu_user = None
        cpu_sys = None
        if self.wall_start is not None and self.wall_end is not
None:
            wall = self.wall_end - self.wall_start
        if (
            self.cpu_user_start is not None
            and self.cpu_user_end is not None
            and self.cpu_sys_start is not None
            and self.cpu_sys_end is not None
        ):
            cpu_user = self.cpu_user_end - self.cpu_user_start
            cpu_sys = self.cpu_sys_end - self.cpu_sys_start

        return {
            "real_time_seconds": wall,
            "cpu_user_seconds": cpu_user,
            "cpu_system_seconds": cpu_sys,
            "cpu_total_seconds": (None if cpu_user is None or
cpu_sys is None else cpu_user + cpu_sys),
            "gpu_time_seconds": (self.gpu_ms / 1000.0) if
self.enable_gpu_timing else None,
        }


def parse_args():
    parser = argparse.ArgumentParser(
        description="Evaluate a (small) causal LLM on MMLU
multiple-choice subjects."
```

```
    )

    parser.add_argument(
        "--model",
        type=str,
        default=MODEL_NAME,
        help="Hugging Face model id (default: %(default)s)",
    )

    parser.add_argument(
        "--subjects",
        nargs="+",
        default=None,
        help=(
            "MMLU subject names. Examples: "
            "--subjects astronomy business_ethics OR --subjects
astronomy,business_ethics. "
            "If omitted, uses the script's MMLU_SUBJECTS list."
        ),
    )

    device_group = parser.add_mutually_exclusive_group()
    device_group.add_argument(
        "--use-gpu",
        "--gpu",
        dest="use_gpu",
        action="store_true",
        help="Attempt to use an available GPU (CUDA on NVIDIA, MPS
on Apple Silicon).",
    )
    device_group.add_argument(
        "--cpu",
        action="store_true",
        help="Force CPU-only execution.",
    )

    parser.add_argument(
        "--quant",
        type=str,
        default=("none" if QUANTIZATION_BITS is None else
str(QUANTIZATION_BITS)),
        choices=["none", "4", "8"],
        help="Quantization bits: none, 4, or 8 (default:
%(default)s). CUDA-only.",
    )

    parser.add_argument(
        "--verbose",
        action="store_true",
        help="Print each question, choices, predicted answer,
correct answer, and RIGHT/WRONG.",
    )

    return parser.parse_args()


def detect_device():
    """Detect the best available device (CUDA, MPS, or CPU)"""

    # If GPU is disabled, always use CPU
    if not USE_GPU:
        return "cpu"

    # Check for CUDA
    if torch.cuda.is_available():
        return "cuda"

    # Check for Apple Silicon with Metal
    if torch.backends.mps.is_available():
        # Check if we're actually on Apple ARM
        is_apple_arm = platform.system() == "Darwin" and
platform.processor() == "arm"

        if is_apple_arm:
            # Metal is available but incompatible with quantization
            if QUANTIZATION_BITS is not None:
                print("\n" + "="*70)
                print("ERROR: Metal and Quantization Conflict")
                print("="*70)
                print("Metal Performance Shaders (MPS) is
incompatible with quantization.")
                print(f"You have USE_GPU = True and
QUANTIZATION_BITS = {QUANTIZATION_BITS}")
                print("")
                print("Please choose one of the following options:")
                print("  1. Set USE_GPU = False to use CPU with
quantization")
                print("  2. Set QUANTIZATION_BITS = None to use
Metal without quantization")
                print("="*70 + "\n")
                sys.exit(1)
            return "mps"

    # Default to CPU
    return "cpu"
```

```
def check_environment():
    """Check environment and dependencies"""
    global QUANTIZATION_BITS
    print("="*70)
    print("Environment Check")
    print("="*70)

    # Check if in Colab
    try:
        import google.colab
        print("✓ Running in Google Colab")
        in_colab = True
    except:
        print("✓ Running locally (not in Colab)")
        in_colab = False

    # Check system info
    print(f"✓ Platform: {platform.system()}
({platform.machine()})")
    if platform.system() == "Darwin":
        print(f"✓ Processor: {platform.processor()}")

    # Detect and set device
    device = detect_device()

    # Check device
    if device == "cuda":
        gpu_name = torch.cuda.get_device_name(0)
        gpu_memory =
torch.cuda.get_device_properties(0).total_memory / 1e9
        print(f"✓ GPU Available: {gpu_name}")
        print(f"✓ GPU Memory: {gpu_memory:.2f} GB")
    elif device == "mps":
        print("✓ Apple Metal (MPS) Available")
        print("✓ Using Metal Performance Shaders for GPU
acceleration")
    else:
        print("⚠️  No GPU detected - running on CPU")

    # Check quantization support (CUDA-only for this script)
    # if QUANTIZATION_BITS is not None and device != "cuda":
    #     print("⚠️  Quantization requested, but this script only
supports quantization on CUDA.")
    #     print("✓ Quantization disabled - loading full precision
model")
    #     QUANTIZATION_BITS = None

    if QUANTIZATION_BITS is not None:
        try:
            import bitsandbytes  # noqa: F401
            print(f"✓ bitsandbytes installed - 
{QUANTIZATION_BITS}-bit quantization available (CUDA)")
        except ImportError:
            print("❌ bitsandbytes NOT installed - cannot use 
quantization")
            sys.exit(1)
    else:
        print("✓ Quantization disabled - loading full precision 
model")

    # Check HF authentication
    try:
        from huggingface_hub import HfFolder
        token = HfFolder.get_token()
        if token:
            print("✓ Hugging Face authenticated")
        else:
            print("⚠️  No Hugging Face token found")
            print("Run: huggingface-cli login")
    except:
        print("⚠️  Could not check Hugging Face authentication")

    # Print configuration
    print("\n" + "="*70)
    print("Configuration")
    print("="*70)
    print(f"Model: {MODEL_NAME}")
    print(f"Device: {device}")
    if QUANTIZATION_BITS is not None:
        print(f"Quantization: {QUANTIZATION_BITS}-bit")
        if QUANTIZATION_BITS == 4:
            print(f"Expected memory: ~1.5 GB")
        elif QUANTIZATION_BITS == 8:
            print(f"Expected memory: ~2.5 GB")
    else:
        print(f"Quantization: None (full precision)")
        if device == "cuda":
            print(f"Expected memory: ~2.5 GB (FP16)")
        elif device == "mps":
            print(f"Expected memory: ~2.5 GB (FP16)")
        else:
            print(f"Expected memory: ~5 GB (FP32)")
    print(f"Number of subjects: {len(MMLU_SUBJECTS)}")

    print("="*70 + "\n")
    return in_colab, device
```

```
def get_quantization_config():
    """Create quantization config based on settings"""
    if QUANTIZATION_BITS is None:
        return None

    if QUANTIZATION_BITS == 4:
        # 4-bit quantization (most memory efficient)
        config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.float16,
            bnb_4bit_use_double_quant=True,  # Double quantization
for extra compression
            bnb_4bit_quant_type="nf4"  # NormalFloat4 - better for
LLMs
        )
        print("Using 4-bit quantization (NF4 + double quant)")
        print("Memory usage: ~1.5 GB")
    elif QUANTIZATION_BITS == 8:
        # 8-bit quantization (balanced)
        config = BitsAndBytesConfig(
            load_in_8bit=True,
            llm_int8_threshold=6.0,
            llm_int8_has_fp16_weight=False
        )
        print("Using 8-bit quantization")
        print("Memory usage: ~2.5 GB")
    else:
        raise ValueError(f"Invalid QUANTIZATION_BITS:
{QUANTIZATION_BITS}. Use 4, 8, or None")

    return config


def load_model_and_tokenizer(device):
    """Load Llama model with optional quantization"""
    print(f"\nLoading model {MODEL_NAME}...")
    print(f"Device: {device}")

    try:
        # Load tokenizer
        tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
        print("✓ Tokenizer loaded")

        # Get quantization config
        quant_config = get_quantization_config()

        # Load model
        print("Loading model (this may take 2-3 minutes)...")
        if quant_config is not None:
            # Quantized model loading (only works with CUDA)
            model = AutoModelForCausalLM.from_pretrained(
                MODEL_NAME,
                quantization_config=quant_config,
                device_map="auto",
                low_cpu_mem_usage=True
            )
        else:
            # Non-quantized model loading
            if device == "cuda":
                model = AutoModelForCausalLM.from_pretrained(
                    MODEL_NAME,
                    dtype=torch.float16,
                    device_map="auto",
                    low_cpu_mem_usage=True
                )
            elif device == "mps":
                model = AutoModelForCausalLM.from_pretrained(
                    MODEL_NAME,
                    dtype=torch.float16,
                    low_cpu_mem_usage=True
                )
                model = model.to(device)
            else:  # CPU
                model = AutoModelForCausalLM.from_pretrained(
                    MODEL_NAME,
                    dtype=torch.float32,
                    low_cpu_mem_usage=True
                )
                model = model.to(device)

        model.eval()

        # Print model info
        print("✓ Model loaded successfully!")
        print(f"  Model device: {next(model.parameters()).device}")
        print(f"  Model dtype: {next(model.parameters()).dtype}")

        # Check memory usage
        if torch.cuda.is_available():
            memory_allocated = torch.cuda.memory_allocated(0) / 1e9
            memory_reserved = torch.cuda.memory_reserved(0) / 1e9
            print(f"  GPU Memory: {memory_allocated:.2f} GB
allocated, {memory_reserved:.2f} GB reserved")

            # Check if using quantization
            if quant_config is not None:
                print(f"  Quantization: {QUANTIZATION_BITS}-bit 
active")
        elif device == "mps":
            print(f"  Running on Apple Metal (MPS)")

        return model, tokenizer

    except Exception as e:
        print(f"\n❌ Error loading model: {e}")
        print("\nPossible causes:")
        print("1. No Hugging Face token - Run: huggingface-cli 
login")
        print("2. Llama license not accepted - Visit: 
https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct")
        print("3. bitsandbytes not installed - Run: pip install 
bitsandbytes")
        print("4. Out of memory - Try 4-bit quantization or smaller 
model")
        raise


def format_mmlu_prompt(question, choices):
    """Format MMLU question as multiple choice"""
    choice_labels = ["A", "B", "C", "D"]
    prompt = f"{question}\n\n"
    for label, choice in zip(choice_labels, choices):
        prompt += f"{label}. {choice}\n"
    prompt += "\nAnswer:"
    return prompt


def get_model_prediction(model, tokenizer, prompt):
    """Get model's prediction for multiple-choice question"""
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

    with torch.no_grad():
        timing_start = None
        timing_end = None
        is_cuda = str(model.device).startswith("cuda")
        if is_cuda:
            timing_start = torch.cuda.Event(enable_timing=True)
            timing_end = torch.cuda.Event(enable_timing=True)
            timing_start.record()

        outputs = model.generate(
            **inputs,
            max_new_tokens=MAX_NEW_TOKENS,
            pad_token_id=tokenizer.eos_token_id,
            do_sample=False,
            temperature=1.0
        )

        if timing_start is not None and timing_end is not None:
            timing_end.record()
            torch.cuda.synchronize()
            gpu_ms = timing_start.elapsed_time(timing_end)
        else:
            gpu_ms = None

    generated_text = tokenizer.decode(
        outputs[0][inputs['input_ids'].shape[1]:],
        skip_special_tokens=True
    )

    answer = generated_text.strip()[:1].upper()

    if answer not in ["A", "B", "C", "D"]:
        for char in generated_text.upper():
            if char in ["A", "B", "C", "D"]:
                answer = char
                break
        else:
            answer = "A"

    return answer, gpu_ms


def evaluate_subject(model, tokenizer, subject, *, verbose=False,
gpu_timing_accumulators=None):
    """Evaluate model on a specific MMLU subject"""
    print(f"\n{'='*70}")
    print(f"Evaluating subject: {subject}")
    print(f"{'='*70}")

    enable_gpu_timing = str(model.device).startswith("cuda")
    subject_timing =
TimingAccumulator(enable_gpu_timing=enable_gpu_timing)
    subject_timing.start()

    try:
        dataset = load_dataset("cais/mmlu", subject, split="test")
    except Exception as e:
        print(f"❌ Error loading subject {subject}: {e}")
        return None

    correct = 0
    total = 0
    iterator = dataset if verbose else tqdm(dataset, desc=f"Testing
{subject}", leave=True)

    for example in iterator:
        question = example["question"]
        choices = example["choices"]
        correct_answer_idx = example["answer"]
        correct_answer = ["A", "B", "C", "D"][correct_answer_idx]

        prompt = format_mmlu_prompt(question, choices)
        predicted_answer, gpu_ms = get_model_prediction(model,
tokenizer, prompt)
        if gpu_ms is not None:
            subject_timing.add_gpu_ms(gpu_ms)
            if gpu_timing_accumulators is not None:
                for acc in gpu_timing_accumulators:
                    acc.add_gpu_ms(gpu_ms)

        is_correct = predicted_answer == correct_answer

        if verbose:
            choice_labels = ["A", "B", "C", "D"]
            print()
            print("Question:")
            print(question)
            print()
            print("Choices:")
            for label, choice in zip(choice_labels, choices):
                print(f"{label}. {choice}")
            print()
            print(f"predicted = {predicted_answer}")
            print(f"correct   = {correct_answer}")
            print(f"result    = {'RIGHT' if is_correct else
'WRONG'}")
            print("-"*70)

        if is_correct:
            correct += 1
        total += 1

    accuracy = (correct / total * 100) if total > 0 else 0
    print(f"✓ Result: {correct}/{total} correct = {accuracy:.2f}%")

    subject_timing.stop()

    return {
        "subject": subject,
        "correct": correct,
        "total": total,
        "accuracy": accuracy,
        "timing": subject_timing.to_dict(),
    }


def main():
    """Main evaluation function"""
    print("\n" + "="*70)
    print("Llama 3.2-1B MMLU Evaluation (Quantized)")
    print("="*70 + "\n")

    args = parse_args()

    import_deps()

    global MODEL_NAME, USE_GPU, QUANTIZATION_BITS, MMLU_SUBJECTS

    MODEL_NAME = args.model

    if args.cpu:
        USE_GPU = False
    elif args.use_gpu:
        USE_GPU = True

    if args.quant == "none":
        QUANTIZATION_BITS = None
    else:
        QUANTIZATION_BITS = int(args.quant)

    if args.subjects is not None:
        if len(args.subjects) == 1 and "," in args.subjects[0]:
            MMLU_SUBJECTS = [s.strip() for s in
args.subjects[0].split(",") if s.strip()]
        else:
            MMLU_SUBJECTS = [s.strip() for s in args.subjects if
s.strip()]

    # Check environment
    in_colab, device = check_environment()

    # Load model
    model, tokenizer = load_model_and_tokenizer(device)

    # Evaluate
    results = []
    total_correct = 0
    total_questions = 0

    is_cuda = next(model.parameters()).device.type == "cuda"
    overall_timing = TimingAccumulator(enable_gpu_timing=is_cuda)

    print(f"\n{'='*70}")
    print(f"Starting evaluation on {len(MMLU_SUBJECTS)} subjects")
    print(f"{'='*70}\n")

    start_time = datetime.now()
    overall_timing.start()

    for i, subject in enumerate(MMLU_SUBJECTS, 1):
        print(f"\nProgress: {i}/{len(MMLU_SUBJECTS)} subjects")
        result = evaluate_subject(
            model,
            tokenizer,
            subject,
            verbose=args.verbose,
            gpu_timing_accumulators=[overall_timing] if is_cuda else
None,
        )
        if result:
            results.append(result)
            total_correct += result["correct"]
            total_questions += result["total"]

    end_time = datetime.now()
    overall_timing.stop()
    duration = (end_time - start_time).total_seconds()

    # Calculate overall accuracy
    overall_accuracy = (total_correct / total_questions * 100) if
total_questions > 0 else 0

    # Print summary
    print("\n" + "="*70)
    print("EVALUATION SUMMARY")
    print("="*70)
    print(f"Model: {MODEL_NAME}")
    print(f"Quantization: {QUANTIZATION_BITS}-bit" if
QUANTIZATION_BITS else "None (full precision)")
    print(f"Total Subjects: {len(results)}")
    print(f"Total Questions: {total_questions}")
    print(f"Total Correct: {total_correct}")
    print(f"Overall Accuracy: {overall_accuracy:.2f}%")
    print(f"Duration: {duration/60:.1f} minutes")
    timing_summary = overall_timing.to_dict()
    if timing_summary["real_time_seconds"] is not None:
        print(f"Real time: {timing_summary['real_time_seconds']:.2f}
s")
    if timing_summary["cpu_total_seconds"] is not None:
        print(
            "CPU time: "
            f"{timing_summary['cpu_total_seconds']:.2f} s "
            f"(user {timing_summary['cpu_user_seconds']:.2f} s,
system {timing_summary['cpu_system_seconds']:.2f} s)"
        )
    if timing_summary["gpu_time_seconds"] is not None:
        print(f"GPU time (CUDA events):
{timing_summary['gpu_time_seconds']:.2f} s")
    print("="*70)

    # Save results
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    quant_suffix = f"_{QUANTIZATION_BITS}bit" if QUANTIZATION_BITS
else "_full"
    output_file =
f"llama_3.2_1b_mmlu_results{quant_suffix}_{timestamp}.json"

    output_data = {
        "model": MODEL_NAME,
        "quantization_bits": QUANTIZATION_BITS,
        "timestamp": timestamp,
        "device": str(device),
        "duration_seconds": duration,
        "timing": timing_summary,
        "overall_accuracy": overall_accuracy,
        "total_correct": total_correct,
        "total_questions": total_questions,
        "subject_results": results
    }

    with open(output_file, "w") as f:
        json.dump(output_data, f, indent=2)

    print(f"\n✓ Results saved to: {output_file}")

    # Print top/bottom subjects
    if len(results) > 0:
        sorted_results = sorted(results, key=lambda x:
x["accuracy"], reverse=True)

        print("\n📊 Top 5 Subjects:")
        for i, result in enumerate(sorted_results[:5], 1):
            print(f"  {i}. {result['subject']}:
{result['accuracy']:.2f}%")

        print("\n📉 Bottom 5 Subjects:")
        for i, result in enumerate(sorted_results[-5:], 1):
            print(f"  {i}. {result['subject']}:
```

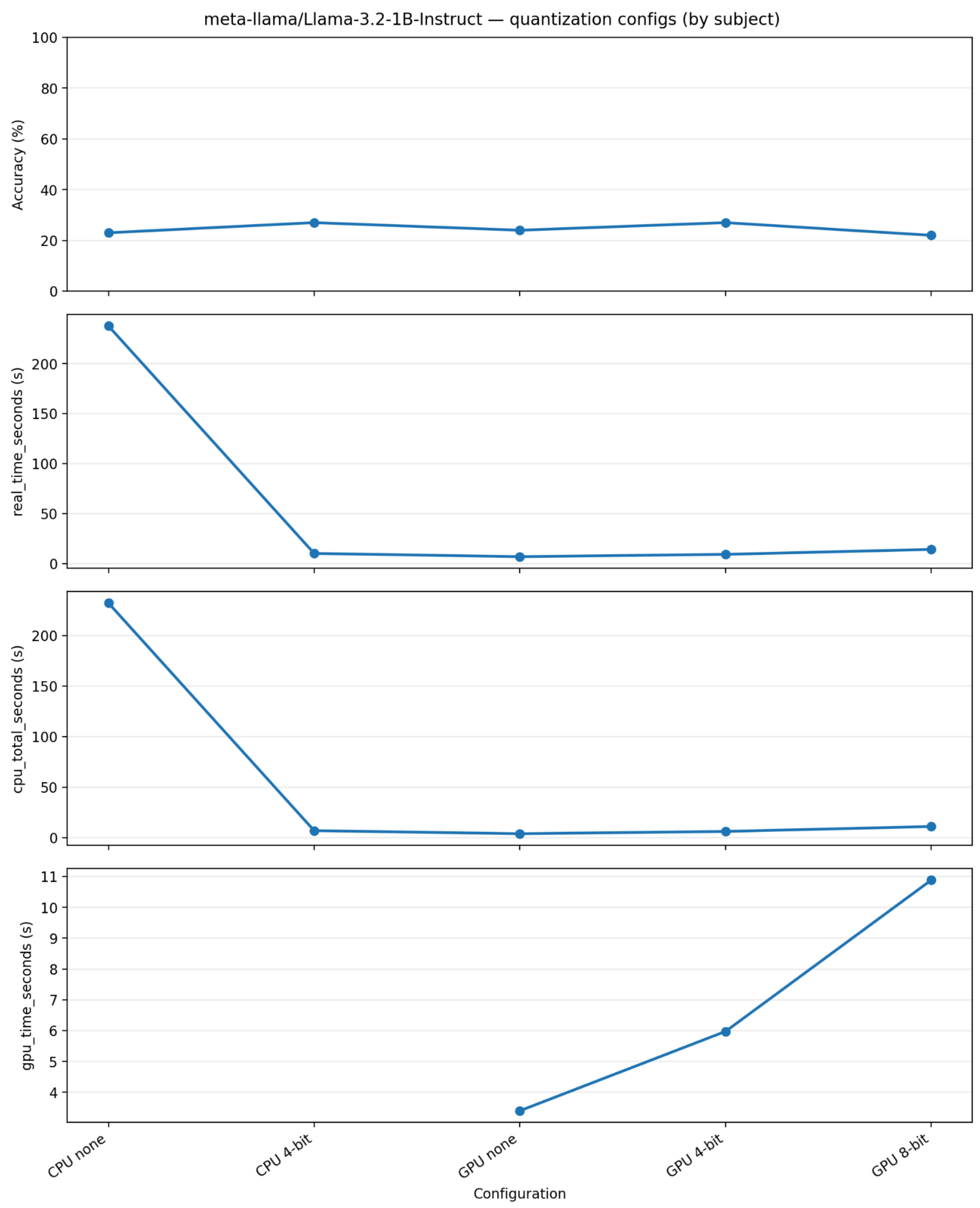
```
{result['accuracy']:.2f}%")

    # Colab-specific instructions
    if in_colab:
        print("\n" + "="*70)
        print("💾 To download results in Colab:")
        print("="*70)
        print(f"from google.colab import files")
        print(f"files.download('{output_file}')")

    print("\n✅ Evaluation complete!")
    return output_file


if __name__ == "__main__":
    try:
        output_file = main()
    except KeyboardInterrupt:
        print("\n\n⚠️  Evaluation interrupted by user")
    except Exception as e:
        print(f"\n❌ Error during evaluation: {e}")
        import traceback
        traceback.print_exc()
```

Quantization Comparison



I tested on **meta-llama/Llama-3.2-1B-Instruct** model with same subject (abstract algebra)

Why **GPU with None Quantization (FP16/FP32)** is faster, but **GPU with 8-bit Quantization** is slower than the "None" case?

A:

1. **Quantization/Dequantization Overhead:** When using 8-bit quantization with `bitsandbytes`, the weights are stored in 8-bit format, but for computation, they must often be **dequantized** back to 16-bit floating point, multiplied, and then potentially **re-quantized**. This process adds overhead that can sometimes

negate the speed benefit of moving less data, especially for smaller models or when the GPU's memory bandwidth isn't the primary bottleneck.
2. **Non-Native Operations:** While newer GPUs have dedicated INT8 support, the specific 8-bit implementation used by `bitsandbytes` (which often focuses on storage and memory savings) might not perfectly align with the fastest native INT8 kernels, forcing the framework to use slower, more generic kernels.
3. **Kernel Launch/Setup Time:** The overhead of setting up and launching the specialized quantization kernels can, for a fast-executing task like a single MMLU question, add a non-trivial amount of time compared to using the highly optimized standard FP16 kernels.

Code

```
import argparse
import json
import math
import os
from dataclasses import dataclass


DEFAULT_CONFIGS = [
    ("CPU none", "cpu", None),
    ("CPU 4-bit", "cpu", 4),
    ("GPU none", "cuda", None),
    ("GPU 4-bit", "cuda", 4),
    ("GPU 8-bit", "cuda", 8),
]

DEFAULT_TIMING_KEYS = [
    "real_time_seconds",
    "cpu_total_seconds",
    "gpu_time_seconds",
]


def _load_json(path):
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)


def _quant_bits(value):
    if value is None:
        return None
    try:
        return int(value)
    except Exception:
        return None


def _is_cuda_device(device_value):
```

```python
    if device_value is None:
        return False
    return str(device_value).lower().startswith("cuda")


def _matches_device(device_value, want):
    dv = ("" if device_value is None else str(device_value).lower())
    if want == "cuda":
        return dv.startswith("cuda")
    if want == "cpu":
        return dv == "cpu"
    return dv == want


def _pick_latest_by_timestamp(paths):
    def key(p):
        try:
            d = _load_json(p)
            return d.get("timestamp") or os.path.basename(p)
        except Exception:
            return os.path.basename(p)

    return sorted(paths, key=key)[-1]


def find_latest_run(directory_path, *, want_device, want_quant_bits):
    """
    Finds the latest JSON run matching device + quantization_bits.
    Returns (path, data) or (None, None).
    """
    if not os.path.isdir(directory_path):
        return None, None

    candidates = []
    for name in os.listdir(directory_path):
        if not name.endswith(".json"):
            continue
        path = os.path.join(directory_path, name)
        try:
            d = _load_json(path)
        except Exception:
            continue

        if not _matches_device(d.get("device"), want_device):
            continue

        qb = _quant_bits(d.get("quantization_bits"))
        if qb != want_quant_bits:
            continue

        candidates.append(path)
    if not candidates:
        return None, None

    chosen = _pick_latest_by_timestamp(candidates)
    return chosen, _load_json(chosen)


@dataclass(frozen=True)
class RunSeries:
    label: str
    path: str
    model: str
    device: str
    quant_bits: int | None
    subject_map: dict
    subjects_in_order: list[str]


def _build_run_series(label, path, data):
    subject_results = data.get("subject_results") or []
    subjects_in_order = [r.get("subject") for r in subject_results if
r.get("subject")]
    subject_map = {r.get("subject"): r for r in subject_results if
r.get("subject")}
    return RunSeries(
        label=label,
        path=path,
        model=data.get("model", "model"),
        device=str(data.get("device", "")),
        quant_bits=_quant_bits(data.get("quantization_bits")),
        subject_map=subject_map,
        subjects_in_order=subjects_in_order,
    )


def create_quantization_configs_line_chart(
    *,
    cpu_dir,
    gpu_dir,
    output_path=None,
    configs=None,
    subjects=None,
    include_all_timings=True,
    timing_keys=None,
    title=None,
):
    """
    Line chart comparing these 5 configurations:
      CPU none, CPU 4-bit, GPU none, GPU 4-bit, GPU 8-bit

    If you have multiple subjects:
      - X-axis: subjects
      - Series: one line per configuration

    If you have only one subject:
      - X-axis: configurations
      - Series: single line (points) across configurations

    Subplot 1: accuracy (%)
    Subplots 2..N: timing keys (seconds)
    """
    try:
        import matplotlib

        matplotlib.use("Agg")
        import matplotlib.pyplot as plt
    except ModuleNotFoundError as exc:
        raise SystemExit(
            "Missing dependency: matplotlib\n\nInstall it with:\n"
            "  python -m pip install matplotlib\n"
        ) from exc

    if configs is None:
        configs = list(DEFAULT_CONFIGS)

    runs = []
    for label, dev, qbits in configs:
        directory = gpu_dir if dev == "cuda" else cpu_dir
        path, data = find_latest_run(directory, want_device=dev,
want_quant_bits=qbits)
        if path is None:
            continue
        runs.append(_build_run_series(label, path, data))

    if not runs:
        raise ValueError("No matching JSON runs found for the requested
configurations.")

    if output_path is None:
        output_path = "quantization_configs_by_subject.png"

    if subjects is None:
        # Prefer subject order from the first run; fall back to union.
        if runs[0].subjects_in_order:
            subjects = runs[0].subjects_in_order
        else:
            union = sorted({s for r in runs for s in
r.subjects_in_order})
            subjects = union

    if timing_keys is None:
        timing_keys = list(DEFAULT_TIMING_KEYS if include_all_timings
else ["real_time_seconds"])
```

```
    if title is None:
        title = f"{runs[0].model} -- quantization configs (by subject)"

    single_subject_mode = len(subjects) == 1

    if single_subject_mode:
        subject = subjects[0]
        config_labels = [r.label for r in runs]
        x = list(range(len(config_labels)))
        x_label = "Configuration"
        fig_width = max(10, len(config_labels) * 1.2)
    else:
        x = list(range(len(subjects)))
        x_label = "MMLU subject"
        fig_width = max(12, len(subjects) * 0.7)

    nrows = 1 + len(timing_keys)
    fig, axes = plt.subplots(
        nrows=nrows,
        ncols=1,
        figsize=(fig_width, 2.6 * nrows + 2),
        sharex=True,
    )
    if nrows == 1:
        axes = [axes]

    fig.suptitle(title)

    # Accuracy
    ax_acc = axes[0]
    if single_subject_mode:
        ys = []
        for run in runs:
            sr = run.subject_map.get(subject) or {}
            ys.append(float(sr.get("accuracy", math.nan) or math.nan))
        ax_acc.plot(x, ys, marker="o", linewidth=2)
    else:
        for run in runs:
            ys = []
            for subj in subjects:
                sr = run.subject_map.get(subj) or {}
                ys.append(float(sr.get("accuracy", math.nan) or
math.nan))
            ax_acc.plot(x, ys, marker="o", linewidth=2,
label=run.label)
        ax_acc.legend(loc="upper right", fontsize="small")

    ax_acc.set_ylabel("Accuracy (%)")
    ax_acc.set_ylim(0, 100)
    ax_acc.grid(axis="y", alpha=0.25)

    # Timings
    for i, key in enumerate(timing_keys, start=1):
        ax = axes[i]
        if single_subject_mode:
            ys = []
            for run in runs:
                sr = run.subject_map.get(subject) or {}
                t = (sr.get("timing") or {}).get(key)
                if key == "gpu_time_seconds" and not
_is_cuda_device(run.device):
                    t = None
                ys.append(float(t) if t is not None else math.nan)
            ax.plot(x, ys, marker="o", linewidth=2)
        else:
            for run in runs:
                ys = []
                for subj in subjects:
                    sr = run.subject_map.get(subj) or {}
                    t = (sr.get("timing") or {}).get(key)
                    if key == "gpu_time_seconds" and not
_is_cuda_device(run.device):
                        t = None
                    ys.append(float(t) if t is not None else math.nan)
                ax.plot(x, ys, marker="o", linewidth=2,
label=run.label)
        ax.set_ylabel(f"{key} (s)")
        ax.grid(axis="y", alpha=0.25)

    axes[-1].set_xticks(x)
    axes[-1].set_xticklabels(
        (config_labels if single_subject_mode else subjects),
        rotation=35,
        ha="right",
    )
    axes[-1].set_xlabel(x_label)

    fig.tight_layout()
    fig.savefig(output_path, bbox_inches="tight", dpi=200)
    return output_path, [r.path for r in runs]


def main():
    here = os.path.dirname(__file__)
    parser = argparse.ArgumentParser(
        description="Compare CPU/GPU quantization configurations as
separate line-series across subjects."
    )
    parser.add_argument("--cpu-dir", default=os.path.join(here, "cpu"),
help='CPU JSON dir (default: "<this_dir>/cpu").')
    parser.add_argument("--gpu-dir", default=os.path.join(here, "gpu"),
help='GPU JSON dir (default: "<this_dir>/gpu").')
    parser.add_argument("--out", default=None, help="Output PNG path.")
    parser.add_argument("--timing-keys", nargs="+", default=None,
help="Timing keys to include (seconds).")
    parser.add_argument("--subjects", nargs="+", default=None,
help="Optional explicit subject order.")
    args = parser.parse_args()

    out, used = create_quantization_configs_line_chart(
        cpu_dir=args.cpu_dir,
        gpu_dir=args.gpu_dir,
        output_path=args.out,
        subjects=args.subjects,
        timing_keys=args.timing_keys,
        include_all_timings=(args.timing_keys is None),
    )
    print(out)
    print("Used JSON files:")
    for p in used:
        print(f"  {p}")


if __name__ == "__main__":
    main()
```