C Artifact Appendix

C.1 Abstract

This artifact contains the open-source ExecuTorch framework and scripts to reproduce the key experimental results presented in the paper: Qwen3 0.6B, Llama 3.2 1B, and Phi4 Mini LLM inference, and MobileNetV3, ResNet50, ViT, and Swin-T vision model inference—all on CPU (XNNPACK), GPU (Vulkan), and NPU (QNN) backends on a Samsung Galaxy S25 Ultra, with additional Core ML results on iPhone 15 Pro. The artifact allows reviewers to (1) export models from PyTorch to on-device `.pte` format for each backend, (2) build C++ runner binaries for Android, and (3) execute and benchmark models on-device to reproduce the LLM and vision benchmark tables from the Performance Evaluations section of the paper.

Repository: `https://github.com/pytorch/executorch`
Archived: `https://zenodo.org/records/18988192` (DOI: 10.5281/zenodo.18988192)

C.2 Artifact check-list (meta-information)

- **Algorithm:** No new algorithm; this is a systems paper evaluating an on-device inference framework.
- **Program:** ExecuTorch (PyTorch on-device inference framework)
- **Compilation:** CMake ≥3.19, Android NDK r27c+, Python 3.10–3.11
- **Transformations:** None required.
- **Binary:** `llama_main` (LLM runner), `executor_runner` (vision runner); QNN backend uses `qnn_llama_runner` and `qnn_executor_runner`
- **Data set:** Qwen3 0.6B, Llama 3.2 1B Instruct, and Phi4 Mini weights (from HuggingFace); MobileNetV3, ResNet50, ViT, and Swin-T use torchvision pretrained weights (downloaded automatically)
- **Run-time environment:** Android API 28+ on ARM64 device
- **Hardware:** Samsung Galaxy S25 Ultra (Snapdragon 8 Elite / SM8750) or Galaxy S24 (Snapdragon 8 Gen 3 / SM8650); Apple iPhone 15 Pro for Core ML. Host: Linux x86_64 with ≥32 GB RAM (≥80 GB for QNN hybrid mode export). macOS host is supported for XNNPACK, Vulkan, and Core ML backends only; QNN requires Linux x86_64 (Ubuntu 22.04+, CentOS Stream 9, or WSL).
- **Run-time state:** Results are sensitive to thermal throttling. Let the device cool between runs and use `--warmup` for LLM benchmarks. Expect ±5–10% run-to-run variance.
- **Execution:** On-device model inference.
- **Metrics:** Tokens/second (LLM prefill and decode), inference latency in ms (vision)
- **Output:** Console benchmark results matching the LLM and vision benchmark tables from the Performance Evaluations section
- **Experiments:** Export + on-device execution for 3 backends × 3 LLMs (at group sizes 32 and 128 for XNNPACK/Vulkan) + 3 backends × 4 vision models; Core ML for 4 vision models on iOS
- **How much disk space required (approximately)?** ∼40 GB
- **How much time is needed to prepare workflow (approximately)?** 1–2 hours (environment setup + build); QNN: additional 1–4 hours for export
- **How much time is needed to complete experiments (approximately)?** ∼30 minutes for XNNPACK/Vulkan; ∼4 hours including QNN
- **Publicly available?** Yes
- **Code licenses (if publicly available)?** BSD-style license
- **Workflow framework used?** CMake, Python, PyTorch, ADB
- **Archived (provide DOI)?** 10.5281/zenodo.18988192

C.3 Description

C.3.1 How delivered

The artifact is the open-source ExecuTorch repository at `https://github.com/pytorch/executorch`, which includes all export scripts, backend implementations, and runner binaries needed to reproduce the results.

C.3.2 Hardware dependencies

Host machine (export and build): Linux x86_64 (Ubuntu 22.04+, CentOS Stream 9, or WSL with Ubuntu 22.04) with ≥32 GB RAM (≥80 GB for QNN hybrid mode export) and ≥40 GB free disk. The QNN backend requires a Linux x86_64 host.
Target device (on-device execution): Android phone with Snapdragon 8 Elite (SM8750) or Snapdragon 8 Gen 3 (SM8650). The paper uses a Samsung Galaxy S25 Ultra. The device must support Vulkan 1.1+ (all modern Snapdragon devices qualify) and be connected via ADB. For Core ML benchmarks: Apple iPhone 15 Pro or comparable iOS device (iOS 17+).

C.3.3 Software dependencies

- Python 3.10 or 3.11
- CMake ≥3.19, GNU Make or Ninja
- Android NDK r27c or later
- **XNNPACK:** No additional dependencies (bundled with ExecuTorch)
- **Vulkan:** Vulkan SDK (1.4.321.0 used; any recent version should work). Available from LunarG.
- **QNN** (Linux x86_64 only)**:** Qualcomm AI Engine Direct SDK v2.37.0 ; download from `https://softwarecenter.qualcomm.com` or use the automated installer (`backends/qualcomm/scripts/install_qnn_sdk.sh`). Set `QNN_SDK_ROOT` and `LD_LIBRARY_PATH` environment variables. Requires g++ 13 or higher. The QNN backend build script (`backends/qualcomm/scripts/build.sh`) must be run separately—it is not included in `install_executorch.sh`.
- **Core ML:** macOS 13+ host with Xcode 15+. Target device requires iOS 17+.

C.3.4 Data sets

LLM weights must be downloaded from HuggingFace: Qwen3 0.6B (`Qwen/Qwen3-0.6B`), Llama 3.2 1B Instruct (`meta-llama/Llama-3.2-1B-Instruct`), and Phi4 Mini (`microsoft/Phi-4-mini-instruct`). Qwen3 and Phi4 weights require conversion to Meta checkpoint format using bundled scripts. Vision models (MobileNetV3, ResNet50, ViT, Swin-T) use `torchvision` pretrained weights, downloaded automatically during export.

C.4 Installation

```
# Clone the repository
git clone https://github.com/pytorch/executorch.git
cd executorch
git submodule sync && git submodule update --init --recursive

# Create and activate conda environment
conda create -n et python=3.11 -y
conda activate et

# Install ExecuTorch in editable mode
./install_executorch.sh --editable

# Install model-specific requirements
bash examples/models/llama/install_requirements.sh

# Set Android NDK path
export ANDROID_NDK=<path_to_android_ndk>

# For Core ML backend (macOS only)
bash backends/apple/coreml/scripts/install_requirements.sh
```

Optional workaround (macOS, AppleClang ≥17): If `install_executorch.sh` fails during the flatcc build with `-Werror` diagnostics (e.g. `-Wimplicit-int-conversion-on-negation`, `-Wunterminated-string-initialization`), re-run it with `-Werror` suppressed:

```
CFLAGS="-Wno-error" CXXFLAGS="-Wno-error" ./install_executorch.sh --editable
```

Note: The above instructions check out the `main` branch, which is required to export Qwen and Phi models to the Vulkan backend. If you encounter any issues, you may instead check out the `v1.2.0` release tag. Run `git checkout v1.2.0` after cloning, then re-run the submodule update and install steps. If the `v1.2.0` tag is used, then LLM repros will likely not work for the Vulkan backend. The instructions below have been validated on commit `34663328b8` (April 5, 2026).

C.4.1 QNN Backend Setup (Linux x86_64 only)

The QNN backend is *not* built by `install_executorch.sh`. The following additional steps are required to export or run QNN models. Without these steps, QNN export scripts will fail with `ModuleNotFoundError`.

```
# --- QNN SDK + NDK Setup ---

# Option A: Automated download
source backends/qualcomm/scripts/install_qnn_sdk.sh
install_qnn        # Downloads QNN SDK 2.37.0 to /tmp/qnn/
setup_android_ndk  # Downloads Android NDK r26c to /tmp/android-ndk/
export QNN_SDK_ROOT=/tmp/qnn/2.37.0.250724
export ANDROID_NDK_ROOT=/tmp/android-ndk/ndk

# Option B: Manual download
# Download QNN SDK 2.37.0 from the URL in Software Dependencies
# and Android NDK r26c from Google, then set:
#   export QNN_SDK_ROOT=<path_to_qairt/2.37.0.250724>
#   export ANDROID_NDK_ROOT=<path_to_android_ndk>

# Set LD_LIBRARY_PATH (required for host-side QNN compilation)
```

```
export LD_LIBRARY_PATH=$QNN_SDK_ROOT/lib/x86_64-linux-clang/:$LD_LIBRARY_PATH

# Install QNN Python dependencies
pip install py-cpuinfo pydot graphviz "numpy<2"
pip install accelerate lm_eval

# Build QNN backend (x86_64 host + Android arm64)
# Produces build-x86/ and build-android/ directories
./backends/qualcomm/scripts/build.sh
```

C.4.2 Download and Convert Model Weights

All three LLM models must be downloaded from HuggingFace. Qwen3 and Phi4 require conversion to Meta checkpoint format; Llama 3.2 1B includes Meta-format weights in its `original/` subdirectory.

```
mkdir -p hf_checkpoints

# Download into the canonical HuggingFace Hub cache
hf download meta-llama/Llama-3.2-1B-Instruct
hf download Qwen/Qwen3-0.6B
hf download microsoft/Phi-4-mini-instruct

# Resolve each repo's snapshot dir:
SD='from huggingface_hub import snapshot_download as s; import sys; print(s(sys.argv[1],
    local_files_only=True))'
LLAMA_HF_DIR=$(python -c "$SD" meta-llama/Llama-3.2-1B-Instruct)
QWEN3_HF_DIR=$(python -c "$SD" Qwen/Qwen3-0.6B)
PHI4_HF_DIR=$(python -c "$SD" microsoft/Phi-4-mini-instruct)

# Llama 3.2 1B Instruct: Meta-format weights ship inside the
# repo at original/{consolidated.00.pth, params.json,
# tokenizer.model}. No conversion needed.

# Qwen3 0.6B (requires conversion to Meta checkpoint format)
python -m examples.models.qwen3.convert_weights $QWEN3_HF_DIR hf_checkpoints/qwen3_0_6b_meta.pth

# Phi4 Mini (requires conversion)
python -m examples.models.phi_4_mini.convert_weights $PHI4_HF_DIR hf_checkpoints/phi4_mini_meta.pth
```

Set convenience variables for subsequent commands:

```
export LLAMA_CKPT=$LLAMA_HF_DIR/original/consolidated.00.pth
export LLAMA_PARAMS=$LLAMA_HF_DIR/original/params.json
export LLAMA_TOK=$LLAMA_HF_DIR/original/tokenizer.model
export QWEN3_CKPT=hf_checkpoints/qwen3_0_6b_meta.pth
export QWEN3_PARAMS=examples/models/qwen3/config/0_6b_config.json
export PHI4_CKPT=hf_checkpoints/phi4_mini_meta.pth
export PHI4_PARAMS=examples/models/phi_4_mini/config/config.json
```

C.5 Experiment workflow

The workflow consists of four steps: (1) export models, (2) build Android runners, (3) push artifacts to device, and (4) benchmark on-device.

C.5.1 Vision export helper script

At the time of writing, the ExecuTorch `main` branch does not yet include Swin-T in the model registry (`examples/models/`), and the Vulkan export script (`examples/vulkan/export.py`) does not yet support the `-q`/`--quantize` flag for int8 static quantization. The following standalone helper script (`export_vision_models.py`) loads models directly from `torchvision` and uses the ExecuTorch XNNPACK/Vulkan partitioner APIs to export them. Save this file in the ExecuTorch root directory before running the vision export commands.

```
#!/usr/bin/env python3
"""export_vision_models.py -- Export vision models
to XNNPACK/Vulkan .pte files."""

import argparse, logging, torch
from executorch.backends.vulkan.partitioner.vulkan_partitioner import VulkanPartitioner
from executorch.backends.xnnpack.partition.xnnpack_partitioner import XnnpackPartitioner
from executorch.backends.xnnpack.quantizer.xnnpack_quantizer import (
    get_symmetric_quantization_config,
    XNNPACKQuantizer,
)
from executorch.exir import (
    EdgeCompileConfig, ExecutorchBackendConfig,
    to_edge_transform_and_lower,
)
from executorch.extension.export_util.utils import save_pte_program
from torch.export import export
from torchao.quantization.pt2e.quantize_pt2e import convert_pt2e, prepare_pt2e
from torchvision import models

logging.basicConfig(level=logging.INFO)

MODELS = {
  "swin_t": lambda: models.swin_t(
    weights=models.Swin_T_Weights.IMAGENET1K_V1),
  "resnet50": lambda: models.resnet50(
    weights=models.ResNet50_Weights.IMAGENET1K_V1),
  "mv3": lambda: models.mobilenet_v3_small(
    weights=\
      models.MobileNet_V3_Small_Weights.IMAGENET1K_V1),
  "vit": lambda: models.vit_b_16(
    weights=models.ViT_B_16_Weights.IMAGENET1K_V1),
}

def get_example_inputs():
    return (torch.randn(1, 3, 224, 224),)

def quantize_model(model, example_inputs,
                   is_per_channel=True,
                   is_dynamic=False):
    exported = export(model, example_inputs).module()
    quantizer = XNNPACKQuantizer()
    config = get_symmetric_quantization_config(
        is_per_channel=is_per_channel,
        is_dynamic=is_dynamic)
    quantizer.set_global(config)
    prepared = prepare_pt2e(exported, quantizer)
    with torch.no_grad():
        prepared(*example_inputs)
```

```
    return convert_pt2e(prepared)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model_name",
      required=True, choices=list(MODELS.keys()))
    parser.add_argument("--xnnpack",
      action="store_true")
    parser.add_argument("--vulkan",
      action="store_true")
    parser.add_argument("-q", "--quantize",
      action="store_true")
    parser.add_argument("-fp16", "--force_fp16",
      action="store_true")
    parser.add_argument("-o", "--output_dir",
      default=".")
    args = parser.parse_args()

    model = MODELS[args.model_name]()
    model.eval()
    example_inputs = get_example_inputs()

    if args.xnnpack:
        if args.quantize:
            q = quantize_model(model, example_inputs)
            ep = export(q, example_inputs)
        else:
            ep = export(model, example_inputs)
        edge = to_edge_transform_and_lower(ep,
          partitioner=[XnnpackPartitioner()],
          compile_config=EdgeCompileConfig(
            _check_ir_validity=not args.quantize,
            _skip_dim_order=True))
        prog = edge.to_executorch(
          config=ExecutorchBackendConfig(
            extract_delegate_segments=False))
        tag = "q8" if args.quantize else "fp32"
        save_pte_program(prog,
          f"{args.model_name}_xnnpack_{tag}",
          args.output_dir)

    if args.vulkan:
        if args.quantize:
            q = quantize_model(model, example_inputs,
                               is_per_channel=True)
            ep = export(q, example_inputs)
        else:
            ep = export(model, example_inputs)
        opts = {}
        if args.force_fp16:
            opts["force_fp16"] = True
        edge = to_edge_transform_and_lower(ep,
          partitioner=[VulkanPartitioner(opts)])
        prog = edge.to_executorch()
        name = f"{args.model_name}_vulkan"
        if args.quantize:
            name += "_q8"
        save_pte_program(prog, name, args.output_dir)
```

```
if __name__ == "__main__":
    with torch.no_grad():
        main()
```

C.6 Step 1: Export Models

All LLMs are exported with 8da4w quantization (8-bit dynamic activations, 4-bit weights) and 4-bit embedding quantization. Results for group sizes 32 and 128 are reported in the paper; the commands below show group size 32. To switch to the other configuration, update *both* the weight group size (`quantization.group_size=32` for XNNPACK, `--group_size 32` for Vulkan) *and* the embedding-quantization group size (`quantization.embedding_quantize='4,32'` for XNNPACK, `--embedding-quantize 4,32` or `-E 4,32` for Vulkan), replacing `32` with `128` in each. The two must match.

C.6.1 XNNPACK Backend (CPU) — LLMs

XNNPACK LLM exports use the Hydra-based export pipeline:

```
# IMPORTANT: output directory must exist before export script is invoked
mkdir -p artifacts

# Llama 3.2 1B -- XNNPACK
python -m extension.llm.export.export_llm \
  --config examples/models/llama/config/llama_xnnpack.yaml \
  ++base.model_class=llama3_2 \
  ++base.checkpoint=$LLAMA_CKPT \
  ++base.params=$LLAMA_PARAMS \
  ++quantization.group_size=32 \
  "++quantization.embedding_quantize='4,32'" \
  ++export.max_seq_length=2048 \
  ++export.max_context_length=2048 \
  ++export.output_name=artifacts/llama3_2_1b_xnnpack_8da4w_g32.pte

# Qwen3 0.6B -- XNNPACK
python -m extension.llm.export.export_llm \
  --config examples/models/qwen3/config/qwen3_xnnpack_q8da4w.yaml \
  ++base.model_class=qwen3_0_6b \
  ++base.checkpoint=$QWEN3_CKPT \
  ++base.params=$QWEN3_PARAMS \
  ++quantization.group_size=32 \
  "++quantization.embedding_quantize='4,32'" \
  ++export.max_seq_length=2048 \
  ++export.max_context_length=2048 \
  ++export.output_name=artifacts/qwen3_0_6b_xnnpack_8da4w_g32.pte

# Phi4 Mini -- XNNPACK
python -m extension.llm.export.export_llm \
  --config examples/models/phi_4_mini/config/phi_4_mini_xnnpack.yaml \
  ++base.checkpoint=$PHI4_CKPT \
  ++base.params=$PHI4_PARAMS \
  ++quantization.qmode=8da4w \
  ++quantization.group_size=32 \
  "++quantization.embedding_quantize='4,32'" \
  ++export.max_seq_length=2048 \
  ++export.max_context_length=2048 \
  ++export.output_name=artifacts/phi4_mini_xnnpack_8da4w_g32.pte
```

C.6.2 XNNPACK Backend (CPU) — Vision Models

```
mkdir -p artifacts

# MV3, ResNet50, ViT (registered in upstream)
python -m executorch.examples.xnnpack.aot_compiler \
  --model_name mv3 --delegate --quantize -o artifacts
python -m executorch.examples.xnnpack.aot_compiler \
  --model_name resnet50 --delegate --quantize -o artifacts
python -m executorch.examples.xnnpack.aot_compiler \
  --model_name vit --delegate --quantize -o artifacts

# Swin-T (not yet registered on main; use helper)
python export_vision_models.py \
  -m swin_t --xnnpack --quantize -o artifacts
```

C.6.3 Vulkan Backend (GPU) — LLMs

PyTorch nightly override (and ExecuTorch rebuild): Vulkan LLM export from `main` requires a bug fix that is not yet in the pinned PyTorch installed by `install_executorch.sh`. Override with the nightly build *only at this point* — doing it earlier breaks the Qwen3 / Phi4 weight conversion scripts, and it is not required for XNNPACK LLM export or any vision export. After upgrading torch you must also rebuild the ExecuTorch wheel against it; use `pip install . --no-build-isolation` for the rebuild — *not* `./install_executorch.sh`, which would repin torch back to `2.11.0` and undo the override. If you are building from the `v1.2.0` release tag, skip this override step (but note that LLM Vulkan export may not work on this tag).

```
pip uninstall -y torch executorch
pip install torch --index-url https://download.pytorch.org/whl/nightly/cpu
pip install . --no-build-isolation
```

```
# IMPORTANT: output directory must exist before export script is invoked
mkdir -p artifacts

# Llama 3.2 1B -- Vulkan
python -m examples.models.llama.export_llama \
  --model llama3_2 \
  --use_kv_cache --use_sdpa_with_kv_cache \
  --checkpoint $LLAMA_CKPT --params $LLAMA_PARAMS \
  --quantization_mode 8da4w --group_size 32 \
  --embedding-quantize 4,32 \
  --vulkan --dtype fp32 \
  --max_seq_length 2048 --max_context_length 2048 \
  --output_name artifacts/llama3_2_1b_vulkan_8da4w_g32

# Qwen3 0.6B -- Vulkan
python -m examples.models.llama.export_llama \
  --model qwen3_0_6b \
  --use_kv_cache --use_sdpa_with_kv_cache \
  --params $QWEN3_PARAMS \
  --quantization_mode 8da4w --group_size 32 \
  -E 4,32 \
  --vulkan --dtype fp32 \
  --max_seq_length 2048 --max_context_length 2048 \
  --output_name artifacts/qwen3_0_6b_vulkan_8da4w_g32
```

```
# Phi4 Mini -- Vulkan
python -m examples.models.llama.export_llama \
  --model phi_4_mini \
  --use_kv_cache --use_sdpa_with_kv_cache \
  --checkpoint $PHI4_CKPT --params $PHI4_PARAMS \
  --quantization_mode 8da4w --group_size 32 \
  -E 4,32 \
  --vulkan --dtype fp32 \
  --max_seq_length 2048 --max_context_length 2048 \
  --output_name artifacts/phi4_mini_vulkan_8da4w_g32
```

Restore the pinned PyTorch version: after the three Vulkan LLM exports complete. Subsequent vision exports may use the pinned version. The simplest reliable way is to re-run the installer, which re-installs the pinned `torch` version:

```
pip uninstall -y torch
./install_executorch.sh --editable
```

C.6.4 Vulkan Backend (GPU) — Vision Models

```
# MV3, ResNet50 fp16, ViT (registered in upstream)
python -m examples.vulkan.export -m mv3 -fp16 -o artifacts
python -m examples.vulkan.export -m resnet50 -fp16 -o artifacts
python -m examples.vulkan.export -m vit -fp16 -o artifacts

# ResNet50 int8 (-q not yet on main; use helper)
python export_vision_models.py -m resnet50 --vulkan --quantize -o artifacts

# Swin-T (not yet registered on main; use helper)
python export_vision_models.py -m swin_t --vulkan -fp16 -o artifacts
```

C.6.5 QNN Backend (NPU) — LLMs

Requires a Linux x86_64 host with the QNN SDK and QNN backend build completed (see Installation). Ensure the following environment is set before running any QNN export:

```
export QNN_SDK_ROOT=<path_to_qnn_sdk>
export LD_LIBRARY_PATH=$QNN_SDK_ROOT/lib/x86_64-linux-clang/:$LD_LIBRARY_PATH
```

The `build-android/` directory should already exist from running `backends/qualcomm/scripts/build.sh` during installation.

Note: QNN hybrid mode LLM export requires $\geq$80 GB host RAM and can take 1–4 hours per model.

Important: every QNN LLM export writes to the same two paths — `llama_qnn/hybrid_llama_qnn.pte` and `llama_qnn/tokenizer.json` — so each export overwrites the previous one. Rename both after each step so all three models survive to the benchmark phase. (Llama uses sentencepiece, so its tokenizer comes from `$LLAMA_TOK` rather than the `tokenizer.json` the script emits.)

```
# Llama 3.2 1B -- QNN (hybrid mode)
python examples/qualcomm/oss_scripts/llama/llama.py \
  -a llama_qnn -b build-android -m SM8750 \
  --decoder_model llama3_2-1b_instruct \
  --checkpoint $LLAMA_CKPT \
  --tokenizer_model $LLAMA_TOK \
```

```
  --params $LLAMA_PARAMS \
  --model_mode hybrid --prefill_ar_len 128 \
  --max_seq_len 2048 \
  --prompt "What is AI?" --compile_only

mv llama_qnn/hybrid_llama_qnn.pte llama_qnn/hybrid_llama32_1b.pte

# Qwen3 0.6B -- QNN (hybrid mode)
python examples/qualcomm/oss_scripts/llama/llama.py \
  -a llama_qnn -b build-android -m SM8750 \
  --decoder_model qwen3-0_6b \
  --model_mode hybrid --prefill_ar_len 128 \
  --max_seq_len 2048 \
  --prompt "What is AI?" --compile_only

mv llama_qnn/hybrid_llama_qnn.pte llama_qnn/hybrid_qwen3_06b.pte

mv llama_qnn/tokenizer.json llama_qnn/qwen3_tokenizer.json

# Phi4 Mini -- QNN (hybrid mode)
python examples/qualcomm/oss_scripts/llama/llama.py \
  -a llama_qnn -b build-android -m SM8750 \
  --decoder_model phi_4_mini \
  --model_mode hybrid --prefill_ar_len 128 \
  --max_seq_len 2048 \
  --prompt "What is AI?" --compile_only

mv llama_qnn/hybrid_llama_qnn.pte llama_qnn/hybrid_phi4_mini.pte

mv llama_qnn/tokenizer.json llama_qnn/phi4_tokenizer.json
```

When `--compile_only` is used, no device connection (`-s`) is required. After the three exports finish, you should have `hybrid_llama32_1b.pte`, `hybrid_qwen3_06b.pte`, `hybrid_phi4_mini.pte`, plus `qwen3_tokenizer.json` and `phi4_tokenizer.json` all under `llama_qnn/`.

C.6.6 QNN Backend (NPU) — Vision Models

```
# MobileNetV3 -- QNN
python examples/qualcomm/scripts/mobilenet_v3.py \
  -m SM8750 -b build-android \
  -s placeholder -a ./artifacts/mv3_qnn \
  --ci --compile_only

# ViT -- QNN
python examples/qualcomm/scripts/torchvision_vit.py \
  -m SM8750 -b build-android \
  -s placeholder -a ./artifacts/vit_qnn \
  --ci --compile_only

# Swin-T -- QNN
python examples/qualcomm/oss_scripts/\
  swin_transformer.py \
  -m SM8750 -b build-android \
  -s placeholder -a ./artifacts/swin_qnn \
  --ci --compile_only
```

ResNet50 QNN export does not have a dedicated upstream script. Save the following as `export_resnet50_qnn.py` and run it from the ExecuTorch root:

```
#!/usr/bin/env python3
"""export_resnet50_qnn.py -- Export ResNet50 to QNN.
import os

import torch

from executorch.backends.qualcomm.export_utils import (
    build_executorch_binary,
    make_quantizer,
    QnnConfig,
    setup_common_args_and_variables,
    SimpleADB,
)
from executorch.backends.qualcomm.quantizer.quantizer \
  import QuantDtype
from executorch.backends.qualcomm.serialization.qc_schema import (
    QnnExecuTorchBackendType,
)
from executorch.examples.models import MODEL_NAME_TO_MODEL
from executorch.examples.models.model_factory \
  import EagerModelFactory


def main(args):
    qnn_config = QnnConfig.load_config(
        args.config_file if args.config_file else args)
    os.makedirs(args.artifact, exist_ok=True)

    if args.ci:
        inputs = [(torch.rand(1, 3, 224, 224),)]
    else:
        raise SystemExit("This script only supports --ci.")

    pte_filename = "resnet50_qnn"
    model, example_inputs, _, _ = \
        EagerModelFactory.create_model(
            *MODEL_NAME_TO_MODEL["resnet50"])
    model = model.eval()

    quantizer = {
        QnnExecuTorchBackendType.kGpuBackend: None,
        QnnExecuTorchBackendType.kHtpBackend: make_quantizer(
            quant_dtype=QuantDtype.use_8a8w,
            backend=qnn_config.backend,
            soc_model=qnn_config.soc_model,
        ),
    }[qnn_config.backend]

    build_executorch_binary(
        model=model,
        qnn_config=qnn_config,
        file_name=f"{args.artifact}/{pte_filename}",
        dataset=inputs,
        custom_quantizer=quantizer,
    )
```

```
    if args.compile_only:
        return

    adb = SimpleADB(
        qnn_config=qnn_config,
        pte_path=f"{args.artifact}/{pte_filename}.pte",
        workspace=
          f"/data/local/tmp/executorch/{pte_filename}",
    )
    adb.push(inputs=inputs)
    adb.execute()


if __name__ == "__main__":
    parser = setup_common_args_and_variables()
    parser.add_argument(
        "-a", "--artifact",
        default="./artifacts/resnet50_qnn", type=str)
    args = parser.parse_args()
    main(args)
```

```
python export_resnet50_qnn.py \
  -m SM8750 -b build-android \
  -s placeholder -a ./artifacts/resnet50_qnn \
  --ci --compile_only
```

C.6.7 Core ML Backend (Apple Neural Engine)

Requires macOS with Xcode. Exports vision models delegated to Core ML with FP16 precision and automatic compute unit selection (CPU/GPU/ANE).

```
# MobileNetV3
python3 -m examples.apple.coreml.scripts.export \
    --model_name mv3 --use_partitioner \
    -c all -precision float16

# ResNet50
python3 -m examples.apple.coreml.scripts.export \
    --model_name resnet50 --use_partitioner \
    -c all -precision float16

# ViT
python3 -m examples.apple.coreml.scripts.export \
    --model_name vit --use_partitioner \
    -c all -precision float16

# Swin-T
python3 -m examples.apple.coreml.scripts.export \
    --model_name swin_t --use_partitioner \
    -c all -precision float16
```

This produces `<model>_coreml_all.pte` files for each model.

C.7 Step 2: Build Android Runners

Required on macOS: the host-side Vulkan shader compile will use Python multiprocessing by default, which may fail with `PermissionError: [Errno 1] Operation not permitted` when

running on macOS. Set `ETVK_SHADER_COMPILE_NTHREADS=1` below to force sequential shader compilation.

```
export ETVK_SHADER_COMPILE_NTHREADS=1
# Core libraries with XNNPACK + Vulkan
cmake . \
    -DCMAKE_INSTALL_PREFIX=cmake-out-android \
    -DCMAKE_TOOLCHAIN_FILE=\
      $ANDROID_NDK/build/cmake/android.toolchain.cmake \
    -DANDROID_SUPPORT_FLEXIBLE_PAGE_SIZES=ON \
    --preset "android-arm64-v8a" \
    -DANDROID_PLATFORM=android-28 \
    -DPYTHON_EXECUTABLE=python \
    -DCMAKE_BUILD_TYPE=Release \
    -DEXECUTORCH_PAL_DEFAULT=posix \
    -DEXECUTORCH_BUILD_EXTENSION_NAMED_DATA_MAP=ON \
    -DEXECUTORCH_BUILD_VULKAN=ON \
    -DEXECUTORCH_BUILD_XNNPACK=ON \
    -DEXECUTORCH_BUILD_TESTS=OFF \
    -DEXECUTORCH_BUILD_EXECUTOR_RUNNER=ON \
    -DEXECUTORCH_BUILD_EXTENSION_EVALUE_UTIL=ON \
    -DEXECUTORCH_VULKAN_SHADER_COMPILE_NTHREADS=$ETVK_SHADER_COMPILE_NTHREADS \
    -Bcmake-out-android && \
cmake --build cmake-out-android -j16 \
    --target install --config Release

# Llama runner
cmake examples/models/llama \
    -DCMAKE_INSTALL_PREFIX=cmake-out-android \
    -DCMAKE_TOOLCHAIN_FILE=\
      $ANDROID_NDK/build/cmake/android.toolchain.cmake \
    -DANDROID_SUPPORT_FLEXIBLE_PAGE_SIZES=ON \
    -DEXECUTORCH_ENABLE_LOGGING=ON \
    -DANDROID_ABI=arm64-v8a \
    -DANDROID_PLATFORM=android-28 \
    -DCMAKE_BUILD_TYPE=Release \
    -DPYTHON_EXECUTABLE=python \
    -Bcmake-out-android/examples/models/llama && \
cmake --build cmake-out-android/examples/models/llama \
    -j16 --config Release
```

C.7.1 QNN Runners (built separately)

The QNN backend uses its own runner binaries (`qnn_executor_runner` and `qnn_llama_runner`). These are built by `backends/qualcomm/scripts/build.sh` during installation (see Section A.4). Verify they exist:

```
ls build-android/examples/qualcomm/executor_runner/qnn_executor_runner
ls build-android/examples/qualcomm/oss_scripts/llama/qnn_llama_runner
ls build-android/lib/executorch/backends/qualcomm/libqnn_executorch_backend.so
```

C.7.2 Build Core ML Runner (macOS)

The Core ML runner is a macOS binary built with Xcode:

```
./examples/apple/coreml/scripts/build_executor_runner.sh
```

This produces `./coreml_executor_runner`.

C.8 Step 3: Push Artifacts and Benchmark

```
export DEVICE_DIR=/data/local/tmp/et_bench
adb shell mkdir -p $DEVICE_DIR

# Push XNNPACK/Vulkan runners
adb push cmake-out-android/executor_runner $DEVICE_DIR/
adb push cmake-out-android/examples/models/llama/llama_main $DEVICE_DIR/
adb shell chmod +x $DEVICE_DIR/executor_runner
adb shell chmod +x $DEVICE_DIR/llama_main

# Push model artifacts
adb push artifacts/ $DEVICE_DIR/
adb push *.pte $DEVICE_DIR/

# Push tokenizers (paths derived from the HF snapshot dirs
# resolved during Step "Download and Convert Model Weights")
adb push $LLAMA_TOK $DEVICE_DIR/tokenizer.model
adb push $QWEN3_HF_DIR/tokenizer.json $DEVICE_DIR/tokenizer.json
adb push $PHI4_HF_DIR/tokenizer.json $DEVICE_DIR/phi4_tokenizer.json
```

C.8.1 LLM benchmarks (XNNPACK / Vulkan)

Each model uses a chat-template-formatted prompt. The `adb shell` command re-interprets special characters (|, <, >), so we write each prompt to a file on the device and pass it to `llama_main` via `--prompt_file`, which sidesteps shell-quoting entirely. Define the system and user text, then construct per-model prompts:

```
export SYSTEM_TEXT="You are a highly capable, \
helpful, and honest AI assistant designed to \
provide clear, accurate, and thoughtful responses \
to a wide range of questions. Your primary goal \
is to assist users by offering information, \
explanations, and guidance in a manner that is \
respectful, unbiased, and safe. Always strive to \
be as helpful as possible, but never provide \
content that is harmful, unethical, offensive, or \
illegal. If a question is unclear, nonsensical, \
or based on incorrect premises, politely explain \
the issue rather than attempting to answer \
inaccurately. If you do not know the answer to a \
question, it is better to admit uncertainty than \
to provide false or misleading information. When \
appropriate, include examples, analogies, or \
step-by-step reasoning to enhance understanding. \
Your responses should be positive, inclusive, and \
supportive, fostering a constructive and \
informative interaction."

export USER_TEXT="Please answer the following \
question in detail and provide relevant context, \
examples, and explanations where possible: What \
are some of the most important considerations \
when designing a machine learning system for \
```

```
real-world applications? Discuss potential \
challenges, best practices, and how to ensure \
ethical and responsible use."
```

Construct the per-model prompts. Llama 3 uses its own chat template; Qwen3 and Phi4 use ChatML format:

```
# Llama 3.2 1B prompt
export LLAMA_PROMPT="<|begin_of_text|>\
<|start_header_id|>system<|end_header_id|>
${SYSTEM_TEXT} ${USER_TEXT}
<|eot_id|><|start_header_id|>assistant<|end_header_id|>"

# Qwen3 0.6B prompt (ChatML)
export QWEN3_PROMPT="<|im_start|>system
${SYSTEM_TEXT}<|im_end|>
<|im_start|>user
${USER_TEXT}<|im_end|>
<|im_start|>assistant
"

# Phi4 Mini prompt (ChatML, same format as Qwen3)
export PHI4_PROMPT="<|im_start|>system
${SYSTEM_TEXT}<|im_end|>
<|im_start|>user
${USER_TEXT}<|im_end|>
<|im_start|>assistant
"
```

Write prompts to files and push to device:

```
echo "$LLAMA_PROMPT" > /tmp/llama_prompt.txt
echo "$QWEN3_PROMPT" > /tmp/qwen3_prompt.txt
echo "$PHI4_PROMPT"  > /tmp/phi4_prompt.txt
adb push /tmp/llama_prompt.txt $DEVICE_DIR/
adb push /tmp/qwen3_prompt.txt $DEVICE_DIR/
adb push /tmp/phi4_prompt.txt  $DEVICE_DIR/
```

Then run the benchmarks. Repeat for each backend (XNNPACK / Vulkan) and group size (32 / 128):

```
# Llama 3.2 1B -- XNNPACK (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
  --model_path=llama3_2_1b_xnnpack_8da4w_g32.pte \
  --tokenizer_path=tokenizer.model \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=llama_prompt.txt"

# Llama 3.2 1B -- Vulkan (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
  --model_path=llama3_2_1b_vulkan_8da4w_g32.pte \
  --tokenizer_path=tokenizer.model \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=llama_prompt.txt"

# Qwen3 0.6B -- XNNPACK (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
```

```
  --model_path=qwen3_0_6b_xnnpack_8da4w_g32.pte \
  --tokenizer_path=tokenizer.json \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=qwen3_prompt.txt"

# Qwen3 0.6B -- Vulkan (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
  --model_path=qwen3_0_6b_vulkan_8da4w_g32.pte \
  --tokenizer_path=tokenizer.json \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=qwen3_prompt.txt"

# Phi4 Mini -- XNNPACK (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
  --model_path=phi4_mini_xnnpack_8da4w_g32.pte \
  --tokenizer_path=phi4_tokenizer.json \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=phi4_prompt.txt"

# Phi4 Mini -- Vulkan (g32)
adb shell "cd $DEVICE_DIR && ./llama_main \
  --model_path=phi4_mini_vulkan_8da4w_g32.pte \
  --tokenizer_path=phi4_tokenizer.json \
  --temperature=0 --warmup=1 --seq_len=2048 \
  --max_new_tokens=257 \
  --prompt_file=phi4_prompt.txt"
```

C.8.2 QNN (NPU) LLM benchmarks

Push QNN SDK runtime libraries, the ExecuTorch QNN backend, and the runner manually. The required HTP libraries depend on the target SoC (see Table 1).

```
# Push QNN SDK runtime libs (SM8750 = Hexagon v79)
adb push $QNN_SDK_ROOT/lib/aarch64-android/libQnnHtp.so $DEVICE_DIR/
adb push $QNN_SDK_ROOT/lib/aarch64-android/libQnnSystem.so $DEVICE_DIR/
adb push $QNN_SDK_ROOT/lib/aarch64-android/libQnnHtpV79Stub.so $DEVICE_DIR/
adb push $QNN_SDK_ROOT/lib/hexagon-v79/unsigned/libQnnHtpV79Skel.so $DEVICE_DIR/
adb push $QNN_SDK_ROOT/lib/aarch64-android/libQnnHtpPrepare.so $DEVICE_DIR/
adb push $QNN_SDK_ROOT/lib/aarch64-android/libQnnModelDlc.so $DEVICE_DIR/

# Push ExecuTorch QNN backend and runner
adb push build-android/lib/executorch/backends/qualcomm/libqnn_executorch_backend.so $DEVICE_DIR/
adb push build-android/examples/qualcomm/oss_scripts/llama/qnn_llama_runner $DEVICE_DIR/
adb shell chmod +x $DEVICE_DIR/qnn_llama_runner

# Push the .pte files renamed during Step 1's QNN LLM export.
adb push ./llama_qnn/hybrid_llama32_1b.pte $DEVICE_DIR/
adb push ./llama_qnn/hybrid_qwen3_06b.pte $DEVICE_DIR/
adb push ./llama_qnn/hybrid_phi4_mini.pte $DEVICE_DIR/

# Tokenizers: Llama 3 uses sentencepiece ($LLAMA_TOK); Qwen3 and
# Phi-4 use HuggingFace JSON tokenizers (renamed in Step 1).
adb push $LLAMA_TOK $DEVICE_DIR/tokenizer.model
adb push ./llama_qnn/qwen3_tokenizer.json $DEVICE_DIR/qwen3_tokenizer.json
adb push ./llama_qnn/phi4_tokenizer.json $DEVICE_DIR/phi4_tokenizer.json
```

```
# Run (Llama 3.2 1B)
adb shell "cd $DEVICE_DIR \
  && export LD_LIBRARY_PATH=$DEVICE_DIR \
  && export ADSP_LIBRARY_PATH=$DEVICE_DIR \
  && ./qnn_llama_runner \
       --model_path ./hybrid_llama32_1b.pte \
       --tokenizer_path ./tokenizer.model \
       --decoder_model_version llama3 \
       --system_prompt \"$SYSTEM_TEXT\" \
       --prompt \"$USER_TEXT\" \
       --temperature 0 --seq_len 600 \
       --num_iters 1"

# Qwen3 0.6B
adb shell "cd $DEVICE_DIR \
  && export LD_LIBRARY_PATH=$DEVICE_DIR \
  && export ADSP_LIBRARY_PATH=$DEVICE_DIR \
  && ./qnn_llama_runner \
       --model_path ./hybrid_qwen3_06b.pte \
       --tokenizer_path ./qwen3_tokenizer.json \
       --decoder_model_version qwen3 \
       --system_prompt \"$SYSTEM_TEXT\" \
       --prompt \"$USER_TEXT\" \
       --temperature 0 --seq_len 600 \
       --num_iters 1"

# Phi4 Mini
adb shell "cd $DEVICE_DIR \
  && export LD_LIBRARY_PATH=$DEVICE_DIR \
  && export ADSP_LIBRARY_PATH=$DEVICE_DIR \
  && ./qnn_llama_runner \
       --model_path ./hybrid_phi4_mini.pte \
       --tokenizer_path ./phi4_tokenizer.json \
       --decoder_model_version phi_4_mini \
       --system_prompt \"$SYSTEM_TEXT\" \
       --prompt \"$USER_TEXT\" \
       --temperature 0 --seq_len 600 \
       --num_iters 1"
```

Table 1: SoC to Hexagon HTP library mapping.

SoC	Snapdragon	Hexagon	Stub/Skel libraries
SM8450	8 Gen 1	v69	`libQnnHtpV69{Stub,Skel}.so`
SM8550	8 Gen 2	v73	`libQnnHtpV73{Stub,Skel}.so`
SM8650	8 Gen 3	v75	`libQnnHtpV75{Stub,Skel}.so`
SM8750	8 Elite	v79	`libQnnHtpV79{Stub,Skel}.so`

C.8.3 Vision model benchmarks (XNNPACK / Vulkan)

Vision models are benchmarked with `executor_runner` using `--num_executions` for timing:

```
adb shell "$DEVICE_DIR/executor_runner \
  --model_path=$DEVICE_DIR/artifacts/<model>.pte \
  --num_executions=200"
```

Repeat for each model/backend artifact:

```
# MobileNetV3
  mv3_xnnpack_q8.pte        # XNNPACK int8
  mv3_vulkan.pte             # Vulkan fp16

# ResNet50
  resnet50_xnnpack_q8.pte    # XNNPACK int8
  resnet50_vulkan.pte        # Vulkan fp16
  resnet50_vulkan_q8.pte     # Vulkan int8

# ViT
  vit_xnnpack_q8.pte         # XNNPACK int8
  vit_vulkan.pte             # Vulkan fp16

# Swin-T
  swin_t_xnnpack_q8.pte      # XNNPACK int8
  swin_t_vulkan.pte          # Vulkan fp16
```

C.8.4 QNN vision model benchmarks:

Push the QNN runner and `.pte`s (after pushing the QNN SDK libraries as shown in the LLM section above):

```
# Push QNN vision runner
adb push build-android/examples/qualcomm/\
  executor_runner/qnn_executor_runner $DEVICE_DIR/
adb shell chmod +x $DEVICE_DIR/qnn_executor_runner
adb push ./artifacts/mv3_qnn/*.pte $DEVICE_DIR/
adb push ./artifacts/resnet50_qnn/*.pte $DEVICE_DIR/
adb push ./artifacts/vit_qnn/*.pte $DEVICE_DIR/
adb push ./artifacts/swin_qnn/*.pte $DEVICE_DIR/

# Run (example for MobileNetV3). Note that qnn_executor_runner
# uses --iteration / --warm_up, NOT --num_executions like the
# XNNPACK/Vulkan executor_runner. With no --input_list_path,
# inputs are filled with default (zero) values.
adb shell "cd $DEVICE_DIR \
  && export LD_LIBRARY_PATH=$DEVICE_DIR \
  && export ADSP_LIBRARY_PATH=$DEVICE_DIR \
  && ./qnn_executor_runner \
       --model_path ./mv3_qnn.pte \
       --iteration 200 --warm_up 10"
```

Repeat for each model: `mv3_qnn.pte`, `resnet50_qnn.pte`, `vit_qnn.pte`, `swin_qnn.pte`. Allow the device to cool $\geq$30 s between runs.

C.8.5 Core ML vision benchmarks (macOS/iPhone):

Run on the macOS host or deploy to an iOS device:

```
# MobileNetV3 -- Core ML
./coreml_executor_runner \
    --model_path mv3_coreml_all.pte \
    --iterations 1000

# ResNet50 -- Core ML
./coreml_executor_runner \
```

```
    --model_path resnet50_coreml_all.pte \
    --iterations 1000

# ViT -- Core ML
./coreml_executor_runner \
    --model_path vit_coreml_all.pte \
    --iterations 200

# Swin-T -- Core ML
./coreml_executor_runner \
    --model_path swin_t_coreml_all.pte \
    --iterations 200
```

C.9 Evaluation and expected result

C.9.1 LLM Benchmarks on Samsung Galaxy S25 Ultra

Reported prefill and decode throughput ranges (tok/s) for ExecuTorch across three models at group size 32 reported in the paper are summarized in Table 2.

Table 2: Expected LLM throughputs for GS = 32

Model	**Backend**	**Prefill (tok/s)**	**Decode (tok/s)**
Qwen3 0.6B	XNNPACK	717–733	72–73
	Vulkan	1206–1246	58
	QNN	1463–1542	61–62
Llama 3.2 1B	XNNPACK	525–529	66–67
	Vulkan	928–931	59
	QNN	2813–2977	47
Phi4 Mini	XNNPACK	144–160	19–20
	Vulkan	191–239	16
	QNN	1161–1229	18–20

C.9.2 Vision Models on Samsung Galaxy S25 Ultra / iPhone 15 Pro

Average inference latencies (ms) reported in the paper are summarized in Table 3.

C.10 Experiment customization

- Different Snapdragon devices (SM8650, SM8750) can be used; expect 10–30% performance variance.
- LLM quantization group size can be changed between 32, 64, 128, and 256 (substitute in the export commands).
- Vision models can be exported with different precision (`--quantize` for INT8, `-fp16` for FP16).
- LLM max context length can be adjusted during export via `--max_seq_length`
- QNN export supports different SoC targets via `-m` flag.

Table 3: Expected latencies for vision models

Model	Backend	Device	Avg (ms)
MobileNetV3	XNNPACK (int8)	S25 Ultra	0.51
MobileNetV3	Vulkan (fp16)	S25 Ultra	2.20
MobileNetV3	QNN (int8)	S25 Ultra	0.24
MobileNetV3	Core ML (fp16)	iPhone 15 Pro	0.40
ResNet50	XNNPACK (int8)	S25 Ultra	4.95
ResNet50	Vulkan (int8)	S25 Ultra	5.44
ResNet50	Vulkan (fp16)	S25 Ultra	22.23
ResNet50	QNN (int8)	S25 Ultra	0.55
ResNet50	Core ML (fp16)	iPhone 15 Pro	1.60
ViT	XNNPACK (int8)	S25 Ultra	64.97
ViT	Vulkan (fp16)	S25 Ultra	136.11
ViT	QNN (int8)	S25 Ultra	3.81
ViT	Core ML (fp16)	iPhone 15 Pro	10.55
Swin-T	XNNPACK (int8)	S25 Ultra	23.06
Swin-T	Vulkan (fp16)	S25 Ultra	36.50
Swin-T	QNN (int8)	S25 Ultra	3.38
Swin-T	Core ML (fp16)	iPhone 15 Pro	8.70

- The benchmark length can be controlled via `--max_new_tokens` (LLM) or `--num_executions` (vision).

C.11 Notes

- QNN backend export can take 1–4 hours for hybrid mode models and requires ≥80 GB host RAM.
- QNN backend requires a Linux x86_64 host for compilation (AOT export). macOS is not supported. The QNN build script (`backends/qualcomm/scripts/build.sh`) must be run before export; `install_executorch.sh` alone does not build the QNN backend or the required `PyQnnManagerAdaptor` Python bindings.
- QNN uses dedicated runner binaries (`qnn_executor_runner`, `qnn_llama_runner`) that differ from the XNNPACK/Vulkan runners (`executor_runner`, `llama_main`). The QNN runners handle loading QNN SDK libraries and initializing the HTP backend.
- On-device QNN execution requires pushing QNN SDK runtime libraries (`libQnnHtp.so`, `libQnnHtpV<X>Stub.so`, `libQnnHtpV<X>Skel.so`, etc.) and setting `LD_LIBRARY_PATH` and `ADSP_LIBRARY_PATH` on the device. The export scripts handle this automatically when run without `--compile_only`. See Table 1 for the SoC-to-library mapping.
- The environment variable for the Android NDK is `ANDROID_NDK_ROOT` (used by the QNN build script), not `ANDROID_NDK` (used by the XNNPACK/Vulkan cmake commands). Set both to the same path for convenience.
- QNN Python bindings require `numpy<2`. If you encounter `RuntimeError: Unable to cast Python instance of type <class 'numpy.ndarray'>` during QNN export, run `pip install "numpy<2"`.

- Qwen3 0.6B and Phi4 Mini weights must be converted from HuggingFace safetensors format to Meta checkpoint format using the bundled `convert_weights.py` scripts. Llama 3.2 1B includes Meta-format weights in its `original/` subdirectory and does not require conversion.
- Quantization configurations (8da4w for LLMs, INT8 for vision) match the paper.
- All experiments use batch size 1.
- LLM models are exported with `--max_seq_length 2048`.
- If `flatc` is not found during export, set `FLATC_EXECUTABLE`:
  `export FLATC_EXECUTABLE=$(find pip-out -name flatc -type f 2>/dev/null | head -1)`

C.12 Methodology

Submission, reviewing and badging methodology:

- `http://cTuning.org/ae/submission-20190109.html`
- `http://cTuning.org/ae/reviewing-20190109.html`
- `https://www.acm.org/publications/policies/artifact-review-badging`