Overview

This guide shows a repeatable, apple-to-apple workflow to compile a common ONNX model (I use MobileNetV2 for classification and YOLOv5s for detection as examples) and run inference on TI AM68A and Renesas RZ/V2N evaluation kits, measure latency and throughput, and trace buffer lifecycles through your display pipeline. Assumptions made: you have one EVK for each platform, host (build) machine with Linux, and basic SDKs installed (TI Edge AI SDK/TIDL and Renesas DRP-AI toolchain). All steps are self-contained and focused on deterministic, reproducible comparison.

---

1. Prepare baseline assets and environment

- **Choose models**
  - MobileNetV2 ONNX (classification) and YOLOv5s ONNX (detection) — both widely supported and small enough to run on both targets.
- **Host tools (on a Linux build machine)**
  - Python 3.8+ with pip, numpy, onnx, onnxruntime (for baseline validation).
  - tflite/onnx utilities if needed for preprocessing.
- **Platform SDKs**
  - TI: Edge AI SDK with TIDL compiler and Edge AI Studio CLI installed on host.
  - Renesas: DRP-AI Translator + TVM toolchain and DRP-AI runtime/tools installed on host.
- **Cross toolchains and SDKs on target**
  - Ensure runtime libraries supplied by each vendor are installed on the EVK rootfs, and that you can copy compiled blobs and run vendor sample apps.
- **Reproducibility**
  - Use git to store scripts and exact SDK versions.
  - Record kernel, libc, SDK versions, and calibration data in a single experiment manifest file.

Files to create and keep in a repo:

- models/ (original ONNX files)
- scripts/compile_ti.sh, scripts/compile_renesas.sh
- scripts/deploy_and_run_ti.sh, scripts/deploy_and_run_renesas.sh
- tests/ (inference client, latency logger)
- manifest.txt (platform kernel, SDK versions, commit hashes, hardware IDs)

---

2. Validate and normalize ONNX model on host

- Sanity run with onnxruntime
    - Run an inference using onnxruntime on the host to verify model correctness and get baseline latency/outputs.
- Standardize input preprocessing
    - Fixed input size, normalization, color order (RGB/BGR), cropping/resizing policy.
    - Save a deterministic test dataset (N images, with fixed seeds) used for all runs.
- Export model metadata
    - Input shape, input dtype, expected output format, anchors (for YOLO), mean/std values.

Example host validation steps:

1. Load model in onnxruntime, feed N deterministic images, save outputs to results/baseline.json.
2. Record model FLOPs/parameters with tools like onnx-simplifier or custom script.

Record baseline CPU inference numbers (onnxruntime) for reference.

---

3. Quantization and operator compatibility

- Decide quantization strategy
    - Use INT8 post-training quantization for both platforms for fair efficiency comparison unless one platform lacks stable INT8 support; otherwise compare FP16 vs INT8 separately.
- Calibration dataset
    - Use the same small calibration set (256–1024 images) for post-training quantization on both toolchains.
- Operator coverage
    - Run an op-support check for each toolchain; identify unsupported ops and prepare replacements or model surgery (fuse conv+bn, replace unsupported activations).
- Quantization-aware training (optional)
    - If INT8 accuracy drop is unacceptable, run QAT on host and export ONNX QAT result before compilation.

Deliverables after this step:

- models/mobilenetv2_fp32.onnx, models/mobilenetv2_int8.onnx
- models/yolov5s_fp32.onnx, models/yolov5s_int8.onnx
- op_support_report_ti.txt, op_support_report_renesas.txt

---

4. Compile for each platform

TI AM68A (TIDL)

- Steps (scripted in scripts/compile_ti.sh):
    1. Run TIDL model import tool to convert ONNX to TIDL IR.
    2. Apply quantization flags (use the same calibration dataset).
    3. Run TIDL codegen to generate .bin/.net artifacts for the C7x/MMA.
    4. Generate runtime metadata and ensure memory map matches target constraints (static buffer sizes).
- Key flags to capture:
    - target core (C7x vs DSP), quantization mode, optimization level, memory pool sizes.
- Validate compiled model on host emulator (if provided) or run a quick dry run on the EVK and compare outputs to host baseline (tolerance thresholds captured in manifest).

Renesas RZ/V2N (DRP-AI + TVM)

- Steps (scripted in scripts/compile_renesas.sh):
    1. Use DRP-AI Translator to convert ONNX to DRP-AI intermediate format.
    2. Use TVM compile and apply quantization using the same calibration set.
    3. Run DRP-AI packager to produce binary + metadata for runtime.
- Key flags:
    - memory pool allocation, tile sizes, scheduling strategy, operator-specific tuning flags.
- Validate output on EVK; compare outputs within tolerances.

Output artifacts to keep:

- deploy/ti/mobilenetv2.bin, deploy/ti/mobilenetv2.meta
- deploy/renesas/mobilenetv2.drp, deploy/renesas/mobilenetv2.json

---

5. Deploy, integrate, and build the test harness

- Deployment
    - Copy compiled blobs and runtime binaries to each EVK under a consistent path (e.g., /opt/ai_test/).
    - Install any required permissions, udev rules for camera devices, and ensure CPU governors are pinned and DVFS disabled for repeatability.
- I/O path for apples-to-apples
    - Use identical capture sources: either a hardware genlock source, recorded video file piped through the same camera ISP settings, or a frame generator

producing identical frames to both boards.
    - Fix framerate (e.g., 30 FPS) and pixel format (e.g., RGB24), and use identical scaling/resizing on-device or in the model input pipeline.
- GStreamer (recommended)
    - Build or reuse small pipelines to feed frames into vendor runtime:
        - Source (v4l2src or filesrc) → identity sync=false → appsink/appsink-structured to feed inference.
    - Use plugins only present and consistent across both targets or emulate the same behavior in user-space clients.
- Test harness components
    - Inference client that:
        - Reads frames, timestamps them at capture (t0).
        - Sends frame to vendor runtime, records t_infer_start and t_infer_end.
        - Writes outputs to disk and prints a concise per-frame CSV line: frame_id, t0_capture_ns, t_enqueue_ns, t_infer_start_ns, t_infer_end_ns, t_display_present_ns, inference_result_hash.
    - A display client (or compositor hook) that logs when a frame buffer is presented to scanout/vblank (t_display_present_ns).
    - A buffer lifecycle tracer that logs buffer allocations, imports/exports, reuse counts, and reference drops. Instrument either vendor runtime and compositor (Weston) to log buffer handles with unique IDs.
- Timing synchronization
    - Synchronize EVK system clocks (use PTP or set them manually). At minimum, use monotonic clocks for internal timing and ensure all timestamps are on the same clock on a given board.
    - If comparing across boards, use host-side timestamps where possible by streaming telemetry back to host and aligning by sequence numbers.
- Automated run script
    - Deploy and run fixed-length experiments (e.g., 5 runs × 1,000 frames) to get statistical certainty.
    - The run script must:
        - Set CPU affinity and governor, disable throttling.
        - Clear caches between runs (where possible).
        - Capture perf counters (CPU util, memory usage, DRP/TIDL accelerator utilization if accessible).
        - Save logs and CSVs into a timestamped experiment directory.

---

6. Metrics, logs to collect, and analysis

- Primary metrics
  - **Inference latency**: t_infer_end − t_infer_start (per-frame).
  - **End-to-end latency**: t_display_present − t0_capture (capture to visible presentation).
  - **Throughput**: frames per second sustained.
  - **Jitter**: 50/90/99th percentile latencies.
  - **Accuracy**: compare outputs to host baseline using IoU (detection) or top-1/top-5 (classification).
- Secondary telemetry
  - Accelerator utilization (vendor counters), CPU usage per core, memory footprint, DRAM bandwidth if available.
  - Buffer lifecycle events: allocation → enqueue → dequeue → release with time deltas.
- Logs
  - Per-frame CSV (as described).
  - Accelerator trace logs from TIDL or DRP runtime.
  - Compositor logs (Weston repaint times, vblank timestamps).
  - dmesg/syslog for kernel warnings during runs.
- Analysis steps
  - Produce per-run summary (mean, std, p50/90/99).
  - Plot latency histograms and CDFs for both platforms.
  - Correlate spikes with buffer reuse or compositor repaint jitter by merging CSVs on frame_id.
  - Report determinism: how many frames missed target presentation deadlines.
- Comparison table to fill
  - Model (name, input size), quantization (FP32/INT8), inference mean latency, p90 latency, throughput, top-1 accuracy delta vs baseline, power draw (if measured), notes on operator fallbacks.

Practical tips and pitfalls

- **Operator mismatch**: If an op falls back to CPU on one platform, it will skew latency — detect this via op-support reports and runtime logs.
- **Memory layout**: Ensure buffer alignment and contiguous memory constraints are handled; mismatches cause copies and high latency.

- **Quantization differences**: INT8 implementations vary; validate accuracy before speed tests.
- **Clock and DVFS effects**: Pin frequencies; otherwise measurements will be noisy.
- **Compositor interference**: Disable unrelated services, and isolate compositor surface used for test to avoid unexpected repaints.
- **Deterministic I/O**: Prefer file-based frame source or hardware genlock; camera jitter ruins comparisons.
- **Buffer tagging**: Add an opaque frame_id field to metadata passed through your pipeline so you can correlate logs across processes and the compositor.

Quick example scripts and CSV format (concise)

- CSV format per-frame:
  - frame_id, t0_capture_ns, t_enqueue_ns, t_infer_start_ns, t_infer_end_ns, t_display_present_ns, result_hash
- Minimal test flow (pseudocode):

1. capture_frame() → stamp t0, save frame_id in metadata
2. preprocess() → enqueue to inference runtime; stamp t_enqueue
3. runtime starts → stamp t_infer_start
4. runtime ends → stamp t_infer_end, write result
5. compositor presents → stamp t_display_present (logged by compositor hook)

- Use the per-frame result_hash to ensure output determinism between platforms (hash final post-NMS bounding boxes or class probabilities).

Final checklist before full experiments

- [ ] Same exact ONNX file used for both compile toolchains (document filename and checksum).
- [ ] Same calibration dataset and quantization mode.
- [ ] CPU frequency pinned and thermals stable.
- [ ] Deterministic input source (genlocked or filesrc).
- [ ] Buffer lifecycle instrumentation enabled on both runtime and compositor.
- [ ] Scripts to parse and merge CSV logs by frame_id.
- [ ] Experiment manifest recorded with all versions and hardware IDs.

If you want, I will now:

- Provide concrete shell script templates for TI TIDL and DRP-AI compilation (compile_ti.sh, compile_renesas.sh),

- Produce a ready-to-run Python inference client used on both boards that emits the CSV format and does the host baseline check,
- Or generate an instrumentation plan to patch Weston and vendor runtimes to emit the exact buffer lifecycle logs (showing header names and sample log lines).

Pick one and I will produce the requested scripts and code.