ROLV Official Hardware Verification & Benchmarking Kit

Purpose

This test establishes a performance and energy baseline on your specific hardware. ROLV uses the data generated here to create a “Us vs. Them” report showing exactly how much faster and more efficient your workload becomes when processed via ROLV Sparse primitives.

The role of SHA-256 hashes

To prove the results are legitimate, the verifier uses SHA-256 hashing as a mathematical proof of equivalence:

1. The Baseline Your hardware generates a unique fingerprint (hash) of the calculation result.
2. The Match When ROLV processes the same data, we must produce the exact same hash.
3. The Proof This guarantees that ROLV is not skipping math or reducing precision. We deliver the exact same high-fidelity results, just significantly faster and with less power.

In rare cases, hashes may vary slightly across CUDA versions or hardware due to floating-point precision. We confirm equivalence numerically (e.g., atol=1e-5) to ensure legitimacy.
Energy measurement (v2.0 — real hardware readings)
Version 2.0 replaces the previous fixed wattage estimate with live hardware power measurements:

- NVIDIA GPUs
  - pynvml polls the GPU power rail every 50 ms
  - Joules computed via trapezoidal integration of real samples
- AMD GPUs
  - pyrsmi provides equivalent live readings where supported
- CPU / Apple Silicon
  - Energy estimated from: psutil CPU utilization × TDP
  - Clearly labeled as an estimate in the output JSON
  -

The energy_measurement_method field in the output JSON always records which method was used.
Python verification script

Paste the following into Jupyter Notebook or save as rolv-verifier.py. Both square and rectangular matrices are supported — set TEST_ROWS and TEST_COLS independently.
Python script (copy-paste ready)

```
#!/usr/bin/env python3

# =============================================================================
# ROLV Baseline Verifier - Hardware Identification & Baseline Collection
# Version 2.0
# =============================================================================
# PURPOSE:
# This script runs a dense matrix multiplication on YOUR hardware and captures
# a cryptographic fingerprint (SHA-256) of the result along with your full
# hardware environment. You email the JSON output to rolv@rolv.ai and we run
# the same computation through ROLV -- on identical inputs, producing an
# identical output hash -- and return a full comparison report.
#
# ENERGY MEASUREMENT:
# NVIDIA GPUs: real joule readings via NVML (pynvml).
# AMD GPUs: real power via pyrsmi where available.
# CPU/Apple: estimated from psutil CPU utilization x TDP.
# =============================================================================

import torch, numpy as np, time, hashlib, json, os, sys
import platform, psutil, re, subprocess

# =============================================================================
# CONFIGURATION BLOCK — YOU CAN CHANGE THESE PARAMETERS
# =============================================================================

TEST_ROWS = 20000  # Matrix Rows (M)
TEST_COLS = 20000  # Matrix Columns (K)
TEST_BATCH = 5000  # Batch Size / Tokens (N)
TEST_SPARSITY = 0.70  # Percentage of zeros (0.0 to 1.0)
TEST_ITERS = 1000  # Number of benchmark iterations
TEST_MODE = "baseline_dense"

def pip_install(*pkgs):
    subprocess.check_call([sys.executable, "-m", "pip", "install", "--quiet", *pkgs])

# =============================================================================
# POWER MONITORS
# =============================================================================

class NvmlMonitor:
    """Polls NVIDIA GPU power via pynvml for accurate joule measurement."""
    def init(self):
        self.available = False
        try:
            import pynvml
            pynvml.nvmlInit()
            self.handle = pynvml.nvmlDeviceGetHandleByIndex(0)
            self.pynvml = pynvml
            self.available = True
        except Exception:
            pass

    def power_watts(self):
        if not self.available: return None
        try:
            return self.pynvml.nvmlDeviceGetPowerUsage(self.handle) / 1000.0
        except Exception:
            return None

    def measure_joules(self, fn, poll_interval=0.05):
        if not self.available: return fn(), None, "unavailable"
        samples, stop_flag, result_box = [], [False], [None]
        import threading

        def worker():
            result_box[0] = fn()
            stop_flag[0] = True

        def poller():
            while not stop_flag[0]:
                w = self.power_watts()
                if w is not None:
                    samples.append((time.perf_counter(), w))
                time.sleep(poll_interval)

        t_worker = threading.Thread(target=worker)
        t_poller = threading.Thread(target=poller)
        t_poller.start()
        t0 = time.perf_counter()
        t_worker.start()
        t_worker.join()
        stop_flag[0] = True
        t_poller.join()
        t1 = time.perf_counter()

        if len(samples) >= 2:
            joules = sum(
                (samples[i][0] - samples[i-1][0]) *
                (samples[i][1] + samples[i-1][1]) / 2
                for i in range(1, len(samples))
            )
        elif len(samples) == 1:
            joules = samples[0][1] * (t1 - t0)
        else:
            joules = None

        return result_box[0], joules, "nvml_integrated"


class AmdMonitor:
    """Polls AMD GPU power via pyrsmi where available."""
    def init(self):
        self.available = False
        try:
            from pyrsmi import rocml
            rocml.smi_initialize()
            self.rocml = rocml
            self.device_id = 0
            self.available = True
        except Exception:
            pass

    def power_watts(self):
        if not self.available: return None
        try:
            return self.rocml.smi_get_device_average_power(self.device_id)
        except Exception:
            return None

    def measure_joules(self, fn, poll_interval=0.05):
        if not self.available: return fn(), None, "unavailable"
        samples, stop_flag, result_box = [], [False], [None]
        import threading

        def worker():
            result_box[0] = fn()
            stop_flag[0] = True

        def poller():
            while not stop_flag[0]:
                w = self.power_watts()
                if w is not None:
                    samples.append((time.perf_counter(), w))
                time.sleep(poll_interval)

        t_worker = threading.Thread(target=worker)
        t_poller = threading.Thread(target=poller)
        t_poller.start()
        t0 = time.perf_counter()
        t_worker.start()
        t_worker.join()
        stop_flag[0] = True
        t_poller.join()
        t1 = time.perf_counter()

        if len(samples) >= 2:
            joules = sum(
                (samples[i][0] - samples[i-1][0]) *
                (samples[i][1] + samples[i-1][1]) / 2
                for i in range(1, len(samples))
            )
        elif len(samples) == 1:
            joules = samples[0][1] * (t1 - t0)
        else:
            joules = None

        return result_box[0], joules, "amd_smi_integrated"


def cpu_joules_estimate(duration_s, tdp_watts=None):
    if tdp_watts is None:
        cores = psutil.cpu_count(logical=False) or 4
        tdp_watts = min(15.0 * cores, 280.0)
    util = psutil.cpu_percent(interval=None) / 100.0
    est_watts = tdp_watts * max(util, 0.1)
    return est_watts * duration_s, f"cpu_utilization_estimate_{tdp_watts:.0f}W_tdp"

# ============================================================================
# ENVIRONMENT CAPTURE
# ============================================================================

def get_full_env():
    env = {
        "processor_model": platform.processor(),
        "cpu_architecture": platform.machine(),
        "cpu_cores_physical": psutil.cpu_count(logical=False),
        "cpu_cores_logical": psutil.cpu_count(logical=True),
        "total_system_ram_gb": round(psutil.virtual_memory().total / (1024**3), 2),
        "os_version": f"{platform.system()} {platform.release()}",
        "python_version": sys.version.split()[0],
        "torch_version": torch.__version__,
    }
    if torch.cuda.is_available():
        env["accelerator"] = torch.cuda.get_device_name(0)
        env["vram_gb"] = round(
            torch.cuda.get_device_properties(0).total_memory / (1024**3), 2
        )
    elif hasattr(torch.backends, "mps") and torch.backends.mps.is_available():
        env["accelerator"] = "Apple Silicon Integrated GPU"
    else:
        env["accelerator"] = "CPU only"
    return env

# ============================================================================
# BASELINE RUNNER
# ============================================================================

def run_rolv_baseline(user_email, rows, cols, batch, zeros, iters):
    try:
        pip_install("pynvml")
    except Exception:
        pass

    device = torch.device(
        "cuda" if torch.cuda.is_available()
        else ("mps" if hasattr(torch.backends, "mps") and torch.backends.mps.is_available()
              else "cpu")
    )

    env_specs = get_full_env()

    # GPU power monitor selection
    monitor = None
    if device.type == "cuda":
        m = NvmlMonitor()
        m.init()
        if m.available:
            print("[] NVML power monitoring: active (real joule measurement)")
            monitor = m
        else:
            print("[!] NVML unavailable -- falling back to estimate")
        try:
            pip_install("pyrsmi")
            m = AmdMonitor()
            m.init()
            if m.available:
                monitor = m
        except Exception:
            pass

    print(f"\n[] Initializing data on {device}...")
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':4096:8'
    torch.manual_seed(12345)

    density = 1.0 - zeros
    A = (torch.rand(rows, cols, device=device) *
         (torch.rand(rows, cols, device=device) < density))
    V = torch.rand(cols, batch, device=device)

    print("[] Warming up (10 iterations)...")
    for _ in range(10):
        torch.matmul(A, V)
    if device.type == "cuda":
        torch.cuda.synchronize()

    print(f"[] Benchmarking {iters} iterations...")
    psutil.cpu_percent(interval=None)

    def bench_fn():
        t0 = time.perf_counter()
        for _ in range(iters):
            result = torch.matmul(A, V)
        if device.type == "cuda":
            torch.cuda.synchronize()
        return result, time.perf_counter() - t0

    if monitor is not None:
        (Y, duration), joules, energy_method = monitor.measure_joules(bench_fn, 0.05)
        if joules is None:
            joules, energy_method = cpu_joules_estimate(duration)
    else:
        Y, duration = bench_fn()
        if device.type == "cuda":
            joules, energy_method = duration * 300.0, "gpu_tdp_estimate_300W"
        else:
            joules, energy_method = cpu_joules_estimate(duration)
    avg_sec = duration / iters
    tflops = ((2 * rows * cols * batch) / avg_sec) / 1e12
    tok_s = (batch * iters) / duration
    h = hashlib.sha256(Y.cpu().numpy().tobytes()[:4_000_000]).hexdigest()

    return {
        "user_info": {"email": user_email},
        "metadata": {
            "mode": TEST_MODE,
            "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
            "verifier_version": "2.0",
            "purpose": (
                "Baseline fingerprint for ROLV comparison. "
                "This hash is produced by dense torch.matmul -- "
                "ROLV will reproduce this exact hash on your configuration."
            ),
        },
        "environment": env_specs,
        "parameters": {
            "rows": rows,
            "cols": cols,
            "batch": batch,
            "sparsity": zeros,
            "iterations": iters
        },
        "metrics": {
            "latency_ms_per_iter": round(avg_sec * 1000, 4),
            "throughput_tokens_sec": round(tok_s, 2),
            "compute_tflops": round(tflops, 4),
            "total_joules_consumed": round(joules, 4),
            "energy_measurement_method": energy_method,
            "sha256_result_hash": h,
        },
    }

# ============================================================================
# MAIN ENTRY
# ============================================================================

if __name__ == "__main__":
    print("\n" + "=" * 60)
    print(" ROLV BASELINE VERIFIER v2.0")
    print(" Hardware Identification & Baseline Collection")
    print("=" * 60)
    print(
        "\n This script runs a dense matmul on your hardware and captures"
        "\n a SHA-256 fingerprint of the result. Email the JSON output to"
        "\n rolv@rolv.ai -- we run ROLV against identical inputs and return"
        "\n a full comparison report showing your speedup and energy savings."
        "\n\n Energy is measured via real hardware power APIs where available"
        "\n (NVML on NVIDIA, AMD SMI on AMD), not estimated from a fixed value.\n"
    )

    user_email = input("Please enter your work email: ").strip()
    if not user_email:
        user_email = "anonymous_user"

    clean_email = re.sub(r"[^a-zA-Z0-9]", "", user_email)
    filename = f"rolv_baseline_{clean_email}_{time.strftime('%Y%m%d%H%M')}.json"

    results = run_rolv_baseline(
        user_email,
        TEST_ROWS,
        TEST_COLS,
        TEST_BATCH,
        TEST_SPARSITY,
        TEST_ITERS
    )

    print("\n[ GENERATED DATA PACKAGE ]")
    print(json.dumps(results, indent=4))

    with open(filename, "w") as f:
        json.dump(results, f, indent=4)

    method = results["metrics"]["energy_measurement_method"]

    print("\n" + "=" * 60)
    print(f" SUCCESS: Results saved to {filename}")
    print(f" Performance : {results['metrics']['throughput_tokens_sec']:,.2f} tokens/s")
    print(f" Energy      : {results['metrics']['total_joules_consumed']:.2f} J [{method}]")
    print(f" Hash        : {results['metrics']['sha256_result_hash'][:32]}...")
    print("=" * 60)
    print(f"\n ACTION: Email '{filename}' to rolv@rolv.ai")
    print(" or paste the JSON above into an email.\n")
```

Reproducibility & system requirements

To ensure consistent results and hash matches:

- PyTorch version Use PyTorch 2.5.0 or later with CUDA support. Example:

bash

```
pip install torch==2.5.1 --index-url https://download.pytorch.org/whl/cu121
```

- CUDA version CUDA 12.1 is ideal for exact reproducibility.

Check with:

bash

```
nvcc --version
python -c "import torch; print(torch.version.cuda)"
```

If using a different version (e.g., 12.8), minor floating-point differences may occur.
If hashes differ, this can happen due to:

- GPU architecture differences (e.g., V100 vs H100)
- CUDA version differences
- Library optimizations
-

ROLV verifies numerical equivalence (atol=1e-5) on our side.

Recommended cloud environments for consistent testing

To avoid mismatched CUDA versions, driver differences, or processors we do not have direct access to, we strongly recommend running the verifier on standardized cloud hardware. This ensures that SHA-256 hashes, timing, and energy measurements closely match our internal validation systems.

- RunPod.io — NVIDIA & AMD GPU testing
  - Ideal for A100, H100, B200, MI210, MI300X
  - Clean CUDA/ROCm stacks
  - Accurate NVML/AMD SMI power telemetry
  - No vendor-modified drivers or background GPU workloads
- Google Colab — Intel Xeon CPU & Google TPU testing
  - Standardized Intel Xeon CPU environments
  - TPU v2/v3 support for CPU vs TPU comparisons
  - Clean PyTorch/XLA setups
- Google Cloud — AMD EPYC CPU testing
  - AMD EPYC Rome/Milan/Genoa instances
  - Stable OS images and predictable performance
  - No laptop power throttling or hidden BIOS limits
  -

Using these environments minimizes variability from:
- Different CUDA/ROCm versions
- Different GPU architectures
- Different BLAS libraries
- Local power-management quirks

and gives you baselines we can reproduce and validate precisely.

How to run & submit

Advanced users
- Save as rolv-verifier.py
- Run:

```bash
python rolv-verifier.py
```
- Email the generated .json file to rolv@rolv.ai
-

Novice users (recommended)
1. Install Anaconda
2. Open Jupyter Notebook
3. Paste the script into a cell
4. Press Shift + Enter
5. Email the generated JSON

Important notes

- Output file name format: rolv_baseline_<your_email>_<timestamp>.json
- This script does not contain ROLV — it only captures your hardware baseline.
- All ROLV computation happens on our infrastructure.
- If NVML or hash warnings appear, verify your PyTorch/CUDA versions or consider using the recommended cloud environments above.