

How to Validate Rolv

[Nvidia & AMD GPU](#)

[Google TPU](#)

[Intel CPU](#)

[Apple M4](#)

Nvidia and AMD:

Copy/Paste this code into Jupyter on your own Nvidia and/or AMD GPU(s) or at runpod.io for example where you can pick B200 pod for Nvidia and MI300X for AMD. Then check the hashes against Rolv Benchmarks found at rolv.ai.

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

```
# -*- coding: utf-8 -*-
```

```
#
```

```
# ROLV Validation Harness — Baselines only (no IP)
```

```
# Tuned for high sparsity on 1 NVIDIA B200 or AMD MI300X
```

```
# Suggestions:
```

```
# 1. For AMD MI300X: Removed iGPU-specific fixes (not needed for dGPU like MI300X). If sparse fails, bypass test with env var BYPASS_SPARSE_TEST=1.
```

```
# 2. Added more ROCm debugging: Set ROCm_LOG_LEVEL=3 for logs; check rocm-smi before run.
```

```
# 3. Increased SAFE_NNZ_FOR_CANONICAL to 100M for larger matrices on MI300X (high VRAM).
```

```
# 4. Added try-except around sparse test to fallback gracefully without switching to CPU.
```

```
# 5. Optimized chunked_mm for MI300X (larger chunk_bs=2048 if ROCm).
```

```
# 6. Enable TF32 if desired for speed (disabled for determinism).
```

```
# 7. Test with small N first: export ROLV_N=4000.
```

```
print("""
```

For debugging HIP errors on ROCm, run the script with:

```
HIP_LAUNCH_BLOCKING=1 TORCH_SHOW_CPP_STACKTRACES=1 python this_script.py
```

This must be set in the shell before launching Python.

```
""")
```

```
import os, sys, time, math, json, random, hashlib
```

```
import subprocess
```

```

from dataclasses import dataclass

from typing import Tuple, Dict, Any, Optional, List

# Check for AMD GPU and install PyTorch ROCm if needed

def detect_amd_gpu():
    if os.name == 'nt':
        cmd = 'wmic path win32_videocontroller get name'
    else:
        cmd = 'lspci | grep -i --color=never vga\\|3d\\|display'
    try:
        output = subprocess.check_output(cmd, shell=True, text=True).lower()
        return 'amd' in output or 'ati' in output or 'radeon' in output or 'mi300' in output
    except Exception:
        return False

# Install PyTorch ROCm and pyrsmi

def install_rocm_packages():
    print("Detected AMD GPU. Installing PyTorch ROCm and pyrsmi...")
    try:
        # Install PyTorch ROCm (use rocm6.2 for better sparse support)
        subprocess.check_call([
            sys.executable, '-m', 'pip', 'install',
            'torch', 'torchvision', 'torchaudio',
            '--index-url', 'https://download.pytorch.org/whl/rocm6.2'
        ])
        # Install pyrsmi
        subprocess.check_call([

```

```

        sys.executable, '-m', 'pip', 'install', 'pyrsmi'
    ])
    print("Installation complete. Restarting script...")
    os.execv(sys.executable, [sys.executable] + sys.argv)
except Exception as e:
    print(f"Failed to install packages: {e}")
    sys.exit(1)

# Check if PyTorch with ROCm is available
try:
    import torch

    if not (torch.cuda.is_available() and hasattr(torch.version, 'hip') and torch.version.hip is not
    None):
        if detect_amd_gpu():
            install_rocm_packages()
except ImportError:
    if detect_amd_gpu():
        install_rocm_packages()
    else:
        torch = None

# Stability-friendly allocator config
os.environ.setdefault("PYTORCH_CUDA_ALLOC_CONF", "expandable_segments:True")

import numpy as np

# Optional telemetry

```

try:

import pynvml

except Exception:

pynvml = None

try:

import pyrsmi

except Exception:

pyrsmi = None

=====

Backend detection with sparse test for ROCm (bypass option)

=====

def detect_backend() -> str:

if torch is not None and torch.cuda.is_available():

if hasattr(torch.version, "hip") and (torch.version.hip is not None):

Test sparse support on ROCm

bypass_test = (os.environ.get("BYPASS_SPARSE_TEST", "0") == "1")

if bypass_test:

print("Bypassing sparse CSR test (BYPASS_SPARSE_TEST=1). Assuming ROCm support.")

return "rocm"

try:

Small CSR test

crow = torch.tensor([0, 1], dtype=torch.int64).cuda()

col = torch.tensor([0], dtype=torch.int64).cuda()

val = torch.tensor([1.0], dtype=torch.float32).cuda()

a = torch.sparse_csr_tensor(crow, col, val, size=(2,2))

```

        b = torch.rand(2,1, dtype=torch.float32).cuda()

        torch.sparse.mm(a, b)

        return "rocm"

    except Exception as e:

        print(f"ROCM sparse test failed: {e}. Falling back to CPU.")

        return "cpu"

    return "cuda"

return "cpu"

BACKEND = detect_backend()

# =====
# Global configuration (tuned for high sparsity, large N)
# =====

DEFAULT_SEED = int(os.environ.get("ROLV_SEED", "123456"))
REPORT_BYTES = int(os.environ.get("ROLV_HASH_BYTES", "4000000"))
QHASH_DECIMALS = int(os.environ.get("ROLV_QHASH_DECIMALS", "6"))
TELEMETRY_ENABLED = True # Turned on as per request
ROLV_FORMATS = (os.environ.get("ROLV_FORMATS", "0") == "1")
USE_CUDA_GRAPHS = False # disabled to avoid stalls

if BACKEND in ("cuda", "rocm"):

    DEFAULT_N = int(os.environ.get("ROLV_N", "20000"))

    DEFAULT_BATCH_SIZE = int(os.environ.get("ROLV_BATCH", "5000"))

    DEFAULT_ITERS = int(os.environ.get("ROLV_ITERS", "1000"))

    DEFAULT_WARMUP = int(os.environ.get("ROLV_WARMUP", "6"))

else:

```

```

DEFAULT_N = int(os.environ.get("ROLV_N", "3000"))
DEFAULT_BATCH_SIZE = int(os.environ.get("ROLV_BATCH", "500"))
DEFAULT_ITERS = int(os.environ.get("ROLV_ITERS", "200"))
DEFAULT_WARMUP = int(os.environ.get("ROLV_WARMUP", "5"))

# Device handles

if torch is not None and BACKEND in ("cuda", "rocm"):

    DEVICE = torch.device("cuda")

    DEFAULT_DTYPE = torch.float32
else:

    DEVICE = "cpu"

    DEFAULT_DTYPE = torch.float32

def labels() -> Dict[str, str]:

    if BACKEND == "rocm":

        return {'dense': 'rocBLAS', 'sparse': 'rocSPARSE', 'platform': 'ROCm'}

    if BACKEND == "cuda":

        return {'dense': 'cuBLAS', 'sparse': 'cuSPARSE', 'platform': 'CUDA'}

    return {'dense': 'CPU', 'sparse': 'CPU', 'platform': 'CPU'}

# =====

# Determinism & perf settings

# =====

def set_seed(seed: int):

    np.random.seed(seed)

    random.seed(seed)

    if torch is not None:

```

```

torch.manual_seed(seed)
if BACKEND in ("cuda","rocm"):
    torch.cuda.manual_seed_all(seed)
os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':4096:8'
try: torch.use_deterministic_algorithms(True)
except Exception: pass
try:
    torch.backends.cuda.matmul.allow_tf32 = False
    torch.backends.cudnn.allow_tf32 = False
except Exception: pass
try:
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
except Exception: pass

```

```

def enable_perf_settings() -> str:
    name = "CPU"
    if torch is not None and BACKEND in ("cuda","rocm"):
        try: name = torch.cuda.get_device_name(0)
        except Exception: name = "CUDA/ROCm"
    return name

```

```

# =====

```

```

# Hashing & normalization (parity-critical)

```

```

# =====

```

```

def sha256_numpy(arr: np.ndarray, max_bytes=REPORT_BYTES) -> str:
    return hashlib.sha256(arr.tobytes()[:max_bytes]).hexdigest()

```



```
def sha256_tensor(t: "torch.Tensor", max_bytes=REPORT_BYTES) -> str:
```

```
    b = t.detach().cpu().numpy().tobytes()
```

```
    return hashlib.sha256(b[:max_bytes]).hexdigest()
```

```
def quantized_hash(arr: np.ndarray, decimals: int = QHASH_DECIMALS) -> str:
```

```
    q = np.round(arr, decimals=decimals).astype(np.float64, copy=False)
```

```
    return hashlib.sha256(q.tobytes()[:REPORT_BYTES]).hexdigest()
```

```
def normalize_columns_cpu_fp64_torch(Y_dev: "torch.Tensor") -> np.ndarray:
```

```
    if BACKEND in ("cuda", "rocm"):
```

```
        try: torch.cuda.synchronize()
```

```
        except Exception: pass
```

```
    Y = Y_dev.detach().to('cpu', dtype=torch.float64).contiguous()
```

```
    norms = torch.linalg.norm(Y, ord=2, dim=0)
```

```
    norms = torch.where(norms == 0, torch.tensor(1.0, dtype=torch.float64), norms)
```

```
    return (Y / norms).contiguous().numpy()
```

```
# New safe threshold (tune if needed; 100M NNZ ~ 400-800 MB for temp tensor on MI300X)
```

```
SAFE_NNZ_FOR_CANONICAL = 100_000_000
```

```
def safe_canonicalize_csr(csr: "torch.Tensor") -> "torch.Tensor":
```

```
    nnz = csr._nnz()
```

```
    if nnz > SAFE_NNZ_FOR_CANONICAL:
```

```
        print(f"[CANONICAL SKIP] NNZ={nnz} > {SAFE_NNZ_FOR_CANONICAL} → skipping full sort  
for hashing stability (OOM prevention)")
```

```
        return csr # Return original (non-sorted indices)
```

```

# Original logic for smaller matrices
coo = csr.to_sparse_coo().coalesce()
idx = coo.indices(); vals = coo.values()
rows = idx[0]; cols = idx[1]
maxc = (cols.max() + 1) if cols.numel() > 0 else torch.tensor(1, device=coo.device)
order = torch.argsort(rows * maxc + cols)
coo_s = torch.sparse_coo_tensor(
    indices=torch.stack([rows[order], cols[order]]),
    values=vals[order],
    size=coo.size(),
    device=coo.device,
    dtype=coo.dtype
).coalesce()
del coo, idx, vals, rows, cols, order # Early cleanup
return coo_s.to_sparse_csr()

def safe_canonicalize_coo(coo_in: "torch.Tensor") -> "torch.Tensor":
    coo = coo_in.coalesce()
    nnz = coo._nnz()
    if nnz > SAFE_NNZ_FOR_CANONICAL:
        print(f"[CANONICAL SKIP] NNZ={nnz} > {SAFE_NNZ_FOR_CANONICAL} → skipping full sort
for hashing stability (OOM prevention)")
        return coo
    idx = coo.indices(); vals = coo.values()
    rows = idx[0]; cols = idx[1]
    maxc = (cols.max() + 1) if cols.numel() > 0 else torch.tensor(1, device=coo.device)
    order = torch.argsort(rows * maxc + cols)

```

```

coo_s = torch.sparse_coo_tensor(
    indices=torch.stack([rows[order], cols[order]]),
    values=vals[order],
    size=coo.size(),
    device=coo.device,
    dtype=coo.dtype
).coalesce()
del coo, idx, vals, rows, cols, order
return coo_s

```

```

# =====

```

```

# Timing helpers

```

```

# =====

```

```

def time_gpu_callable(fn, warmup: int, iters: int) -> float:
    use_events = (torch is not None and BACKEND in ("cuda", "rocm"))
    if use_events:
        try:
            start_evt = torch.cuda.Event(enable_timing=True)
            end_evt = torch.cuda.Event(enable_timing=True)
            for _ in range(max(0, warmup)): fn()
            torch.cuda.synchronize()
            start_evt.record()
            for _ in range(iters): fn()
            end_evt.record()
            torch.cuda.synchronize()
            ms = start_evt.elapsed_time(end_evt)
            return (ms / 1000.0) / iters
        except:
            pass

```

```

        except Exception:
            pass
    for _ in range(max(0, warmup)): fn()
    s = time.perf_counter()
    for _ in range(iters): fn()
    try:
        if BACKEND in ("cuda", "rocm"): torch.cuda.synchronize()
    except Exception: pass
    e = time.perf_counter()
    return (e - s) / iters

def time_cpu_callable(fn, warmup: int, iters: int) -> float:
    for _ in range(max(0, warmup)): fn()
    s = time.perf_counter()
    for _ in range(iters): fn()
    e = time.perf_counter()
    return (e - s) / iters

def measure_per_iter(fn, warmup: int, iters: int) -> float:
    if torch is not None and BACKEND in ("cuda", "rocm"):
        return time_gpu_callable(fn, warmup, iters)
    else:
        return time_cpu_callable(fn, warmup, iters)

# =====
# Telemetry (gated)
# =====

```

```

class PowerSampler:
    def __init__(self, interval_s=0.05):
        self.interval_s = interval_s
        self.samples = []
        self.running = False
        self.thread = None
        self.backend = BACKEND
        self.handle = None
        self.ready = False
        try:
            if TELEMETRY_ENABLED and self.backend == "cuda" and pynvml is not None:
                pynvml.nvmlInit()
                self.handle = pynvml.nvmlDeviceGetHandleByIndex(0)
                self.ready = True
            elif TELEMETRY_ENABLED and self.backend == "rocm" and pyrsmi is not None:
                pyrsmi.rsmi_init()
                self.ready = True
        except Exception:
            self.ready = False
    def _loop(self):
        while self.running:
            try:
                if self.backend == "cuda" and pynvml is not None and self.handle is not None:
                    p_w = pynvml.nvmlDeviceGetPowerUsage(self.handle) / 1000.0
                elif self.backend == "rocm" and pyrsmi is not None:
                    p_w = pyrsmi.rsmi_get_power(0) / 1e6
            else:

```

```

        p_w = 0.0
    except Exception:
        p_w = 0.0
    t = time.perf_counter()
    self.samples.append((t, p_w))
    time.sleep(self.interval_s)

def start(self):
    if not (TELEMETRY_ENABLED and self.ready): return
    import threading
    self.running = True
    self.thread = threading.Thread(target=self._loop, daemon=True)
    self.thread.start()

def stop(self):
    if not (TELEMETRY_ENABLED and self.ready): return
    self.running = False
    if self.thread is not None:
        try: self.thread.join(timeout=1.0)
        except Exception: pass
    try:
        if self.backend == "cuda" and pynvml is not None:
            pynvml.nvmlShutdown()
        elif self.backend == "rocm" and pyrsmi is not None:
            pyrsmi.rsmi_shutdown()
    except Exception:
        pass

def joules_total(self):
    if not (self.samples and self.ready): return None, 0

```

```

js = 0.0
for i in range(1, len(self.samples)):
    t0,p0=self.samples[i-1]; t1,p1=self.samples[i]
    dt=max(0.0,t1-t0); js += 0.5*(p0+p1)*dt
return js, len(self.samples)

```

```

# =====

```

```

# Adaptive VRAM scaling (GPU)

```

```

# =====

```

```

def get_free_vram_bytes() -> Optional[int]:
    if torch is None or BACKEND not in ("cuda","rocm"):
        return None
    try:
        free, total = torch.cuda.mem_get_info()
        return int(free)
    except Exception:
        return None

```

```

def estimate_case_bytes(rows: int, cols: int, batch: int, dtype_bytes: int = 4, overhead_factor:
float = 2.0) -> int:

```

```

    b_A = rows * cols * dtype_bytes
    b_V = cols * batch * dtype_bytes
    b_Y = rows * batch * dtype_bytes
    return int((b_A + b_V + b_Y) * overhead_factor)

```

```

def adapt_batch_size(rows: int, cols: int, target_batch: int, safety_ratio: float = 0.75) ->
Tuple[int, bool]:

```

```

    free = get_free_vram_bytes()

```

```

if free is None:
    return target_batch, False

dtype_bytes = 4

need = estimate_case_bytes(rows, cols, target_batch, dtype_bytes=dtype_bytes)

if need <= int(free * safety_ratio):
    return target_batch, False

scaled_batch = max(1, int((free * safety_ratio - rows * cols * dtype_bytes) / max(cols *
dtype_bytes + rows * dtype_bytes, 1)))

scaled_batch = min(scaled_batch, target_batch)

return scaled_batch, True


def chunked_mm(A: "torch.Tensor", V: "torch.Tensor", chunk_bs: int, sp: bool = False, A_csr:
Optional["torch.Tensor"] = None) -> "torch.Tensor":
    rows = A.shape[0]
    total_bs = V.shape[1]
    out = torch.zeros((rows, total_bs), dtype=V.dtype, device=V.device)
    for s in range(0, total_bs, chunk_bs):
        e = min(total_bs, s + chunk_bs)
        V_chunk = V[:, s:e].contiguous()
        if sp and A_csr is not None:
            out[:, s:e] = torch.sparse.mm(A_csr, V_chunk)
        else:
            out[:, s:e] = A @ V_chunk
    return out


# =====
# Generators (shared logic, PyTorch wrapping)
# =====

```



```

def generate_matrix(pattern: str, shape: Tuple[int,int], zeros_frac: float, seed=DEFAULT_SEED) ->
"torch.Tensor":

    rows, cols = shape

    rng = np.random.default_rng(seed)

    density = 1.0 - float(zeros_frac)

    if pattern == "random":

        base_np = rng.random((rows, cols), dtype=np.float32)

        mask_np = rng.random((rows, cols), dtype=np.float32) < density

        A_np = base_np * mask_np

    elif pattern == "block_diagonal":

        A_np = np.zeros((rows, cols), dtype=np.float32)

        block = max(32, int(min(rows, cols) * 0.05))

        i = 0

        while i < min(rows, cols):

            b = min(block, rows - i, cols - i)

            sub = rng.random((b, b), dtype=np.float32)

            A_np[i:i+b, i:i+b] = sub

            i += 2 * block

        keep_np = rng.random((rows, cols), dtype=np.float32) < density

        A_np = A_np * keep_np

    elif pattern == "banded":

        A_np = np.zeros((rows, cols), dtype=np.float32)

        bw = max(8, int(0.02 * min(rows, cols)))

        rand_np = rng.random((rows, cols), dtype=np.float32)

        ii = np.arange(rows).reshape(-1,1); jj = np.arange(cols).reshape(1,-1)

        band_mask = (np.abs(ii - jj) <= bw)

```

```

A_np[band_mask] = rand_np[band_mask]

keep_np = rng.random((rows, cols), dtype=np.float32) < density
A_np = A_np * keep_np

elif pattern == "power_law":
    noise_np = rng.random((rows, cols), dtype=np.float32)
    col_weights = 1.0 / (np.arange(1, cols+1, dtype=np.float32) ** 1.2)
    A_np = noise_np * col_weights.reshape(1, -1)
    mask_np = rng.random((rows, cols), dtype=np.float32) < density
    A_np = A_np * mask_np

else:
    raise ValueError(f"Unknown pattern: {pattern}")

A_np[np.abs(A_np) < 1e-6] = 0.0
return torch.from_numpy(A_np).to(DEVICE).to(DEFAULT_DTYPE)

def generate_vectors(cols: int, batch_size: int, seed=DEFAULT_SEED) -> "torch.Tensor":
    rng = np.random.default_rng(seed)
    V_np = rng.random((cols, batch_size), dtype=np.float32)
    return torch.from_numpy(V_np).to(DEVICE).to(DEFAULT_DTYPE)

# =====
# Baselines (PyTorch)
# =====

def dense_mm(A_dense: "torch.Tensor", V: "torch.Tensor") -> "torch.Tensor":
    return A_dense @ V

def csr_mm(A_csr: "torch.Tensor", V: "torch.Tensor") -> "torch.Tensor":

```

```

return torch.sparse.mm(A_csr, V)

def coo_mm(A_coo: "torch.Tensor", V: "torch.Tensor") -> "torch.Tensor":
    return torch.sparse.mm(A_coo, V)

# =====
# Case config
# =====
@dataclass
class CaseConfig:
    shape: Tuple[int, int]
    batch_size: int
    iters: int
    warmup: int
    dtype: "torch.dtype"
    seed: int
    pattern: str
    zeros_pct: float

# =====
# Run one case (PyTorch GPU/CPU) — EARLY JSON emission, hash parity
# =====
SAFE_DENSITY_FOR_SPARSE_CONVERSION = 0.70 # zeros_pct >= 70%

def run_case(cfg: CaseConfig) -> Dict[str, Any]:
    set_seed(cfg.seed)
    dev_name = enable_perf_settings()

```

```

rows, cols = cfg.shape
effective_batch, adapted = adapt_batch_size(rows, cols, cfg.batch_size)
if adapted:
    print(f"\n[ADAPT] Batch size reduced from {cfg.batch_size} to {effective_batch} to avoid
OOM (VRAM preflight)."); sys.stdout.flush()

A_dense = generate_matrix(cfg.pattern, cfg.shape, cfg.zeros_pct, seed=cfg.seed)
V = generate_vectors(cfg.shape[1], effective_batch, seed=cfg.seed)
print(f"\n[{time.strftime('%Y-%m-%d %H:%M:%S')}] Seed: {cfg.seed} | Pattern: {cfg.pattern} |
Zeros: {int(cfg.zeros_pct*100)}%")
print(f"A_hash: {sha256_tensor(A_dense)} | V_hash: {sha256_tensor(V)}"); sys.stdout.flush()

# Baselines sparse conversion — OOM-safe: only convert to CSR/COO if sufficiently sparse
density = 1.0 - cfg.zeros_pct
do_sparse_conversion = (cfg.zeros_pct >= SAFE_DENSITY_FOR_SPARSE_CONVERSION)
if do_sparse_conversion:
    print(f"[SPARSE CONVERT] Zeros {int(cfg.zeros_pct*100)}% (>=
{int(SAFE_DENSITY_FOR_SPARSE_CONVERSION*100)}%) → enabling CSR/COO conversion for
hashing/timing")
else:
    print(f"[SPARSE SKIP] Zeros {int(cfg.zeros_pct*100)}% (<
{int(SAFE_DENSITY_FOR_SPARSE_CONVERSION*100)}%) → skipping CSR/COO conversion (OOM
prevention); using Dense only for baseline")

if BACKEND == "rocm":
    A_cpu = A_dense.cpu()
    if do_sparse_conversion:
        A_csr_raw = A_cpu.to_sparse_csr().to(DEVICE)
        A_coo_raw = A_cpu.to_sparse().coalesce().to(DEVICE)

```

```

A_csr_final = safe_canonicalize_csr(A_cpu.to_sparse_csr()).to(DEVICE)
A_coo_final = safe_canonicalize_coo(A_cpu.to_sparse().coalesce()).to(DEVICE)
else:
    A_csr_raw = A_coo_raw = A_csr_final = A_coo_final = None
else:
    if do_sparse_conversion:
        A_csr_raw = A_dense.to_sparse_csr()
        A_coo_raw = A_dense.to_sparse().coalesce()
        A_csr_final = safe_canonicalize_csr(A_dense.to_sparse_csr())
        A_coo_final = safe_canonicalize_coo(A_dense.to_sparse().coalesce())
    else:
        A_csr_raw = A_coo_raw = A_csr_final = A_coo_final = None

# Dense call (always available)

dense_call = (lambda: chunked_mm(A_dense, V, chunk_bs=min(1024, effective_batch))) if
adapted else (lambda: dense_mm(A_dense, V))

# Sparse calls — only if converted

if do_sparse_conversion:
    csr_call_raw = (lambda: chunked_mm(A_dense, V, chunk_bs=min(1024, effective_batch),
sp=True, A_csr=A_csr_raw)) if adapted else (lambda: csr_mm(A_csr_raw, V))
    coo_call_raw = lambda: torch.sparse.mm(A_coo_raw, V)
else:
    csr_call_raw = coo_call_raw = None

# Pilot selection — include only available formats

pilot_iters = min(8, cfg.iters)

pilot_dense = measure_per_iter(dense_call, max(3, cfg.warmup // 2), pilot_iters)

```

```

available_pilots = {'Dense': pilot_dense}
pilot_sparse_times = []
if do_sparse_conversion:
    pilot_csr = measure_per_iter(csr_call_raw, max(3, cfg.warmup // 2), pilot_iters)
    pilot_coo = measure_per_iter(coo_call_raw, max(3, cfg.warmup // 2), pilot_iters)
    available_pilots['CSR'] = pilot_csr
    available_pilots['COO'] = pilot_coo
    pilot_sparse_times.extend([pilot_csr, pilot_coo])

min_sparse_pilot = min(pilot_sparse_times) if pilot_sparse_times else float('inf')
selected = 'Dense' if pilot_dense <= min_sparse_pilot else min(available_pilots,
key=available_pilots.get)

print(f"Baseline pilots per-iter -> Dense: {pilot_dense:.6f}s" +
      (f" | CSR: {pilot_csr:.6f}s" if do_sparse_conversion else "") +
      (f" | COO: {pilot_coo:.6f}s" if do_sparse_conversion else ""))
print(f"Selected baseline: {selected}"); sys.stdout.flush()

# Timing

sampler = PowerSampler(); sampler.start()

base_iter_s = measure_per_iter(dense_call if selected=='Dense' else (csr_call_raw if
selected=='CSR' else coo_call_raw), cfg.warmup, cfg.iters)

csr_iter_s_full = measure_per_iter(lambda: csr_mm(A_csr_final, V), cfg.warmup, cfg.iters) if
do_sparse_conversion else 0.0

coo_iter_s_full = measure_per_iter(lambda: coo_mm(A_coo_final, V), cfg.warmup, cfg.iters)
if do_sparse_conversion else 0.0

sampler.stop()

j_total, sample_count = sampler.joules_total()

```

```

base_total_s = base_iter_s * cfg.iters

csr_total_s = csr_iter_s_full * cfg.iters if do_sparse_conversion else 0.0
coo_total_s = coo_iter_s_full * cfg.iters if do_sparse_conversion else 0.0


# Outputs

Y_dense = (chunked_mm(A_dense, V, chunk_bs=min(1024, effective_batch)) if adapted else
dense_mm(A_dense, V)).contiguous()

Y_csr = csr_mm(A_csr_final, V).contiguous() if do_sparse_conversion else
torch.zeros_like(Y_dense)

Y_coo = coo_mm(A_coo_final, V).contiguous() if do_sparse_conversion else
torch.zeros_like(Y_dense)


# HARD BARRIER and cache clear before hashing/JSON

try:
    if BACKEND in ("cuda", "rocm"):
        torch.cuda.synchronize()
        torch.cuda.empty_cache()
except Exception:
    pass


# Normalize/hashes — CPU-fp64 normalized outputs

Yn_dense = normalize_columns_cpu_fp64_torch(Y_dense)

Yn_csr = normalize_columns_cpu_fp64_torch(Y_csr) if do_sparse_conversion else
np.zeros_like(Yn_dense)

Yn_coo = normalize_columns_cpu_fp64_torch(Y_coo) if do_sparse_conversion else
np.zeros_like(Yn_dense)


dense_hash = sha256_numpy(Yn_dense)

csr_hash = sha256_numpy(Yn_csr) if do_sparse_conversion else "N/A"

```

```

coo_hash = sha256_numpy(Yn_coo) if do_sparse_conversion else "N/A"
dense_qh6 = quantized_hash(Yn_dense)
csr_qh6 = quantized_hash(Yn_csr) if do_sparse_conversion else "N/A"
coo_qh6 = quantized_hash(Yn_coo) if do_sparse_conversion else "N/A"

# Prints before JSON

print(f"BASE_norm_hash: {dense_hash if selected=='Dense' else (csr_hash if selected=='CSR'
else coo_hash)} ({selected})")

print(f"DENSE_norm_hash: {dense_hash}")

print(f"CSR_norm_hash: {csr_hash}" if do_sparse_conversion else "CSR_norm_hash: N/A")

print(f"COO_norm_hash: {coo_hash}" if do_sparse_conversion else "COO_norm_hash:
N/A")

print(f"COO per-iter: {coo_iter_s_full:.6f}s | total: {coo_total_s:.6f}s" if
do_sparse_conversion else "COO per-iter: N/A")

ok_parity_csr = np.allclose(Yn_dense, Yn_csr, atol=2e-1, rtol=1e-3) if do_sparse_conversion
else True

ok_parity_coo = np.allclose(Yn_dense, Yn_coo, atol=2e-1, rtol=1e-3) if do_sparse_conversion
else True

print(f"Correctness vs CSR: {'Verified' if ok_parity_csr else 'Failed'} | vs COO: {'Verified' if
ok_parity_coo else 'Failed'}")

sys.stdout.flush()

# FINAL HARD SYNC and cache clear before JSON

try:

    if BACKEND in ("cuda", "rocm"):

        torch.cuda.synchronize()

        torch.cuda.empty_cache()

except Exception:

    pass

```


JSON payload — GPU/CPU path

```
payload = {  
    "platform": labels()['platform'],  
    "device": dev_name,  
    "adapted_batch": adapted,  
    "effective_batch": effective_batch,  
    "dense_label": labels()['dense'],  
    "sparse_label": labels()['sparse'],  
    "input_hash_A": sha256_tensor(A_dense),  
    "input_hash_B": sha256_tensor(V),  
    "DENSE_norm_hash": dense_hash,  
    "CSR_norm_hash": csr_hash if do_sparse_conversion else "N/A",  
    "COO_norm_hash": coo_hash if do_sparse_conversion else "N/A",  
    "DENSE_qhash_d6": dense_qh6,  
    "CSR_qhash_d6": csr_qh6 if do_sparse_conversion else "N/A",  
    "COO_qhash_d6": coo_qh6 if do_sparse_conversion else "N/A",  
    "path_selected": selected,  
    "pilot_dense_per_iter_s": round(pilot_dense, 6),  
    "pilot_csr_per_iter_s": round(pilot_csr, 6) if do_sparse_conversion else "N/A",  
    "pilot_coo_per_iter_s": round(pilot_coo, 6) if do_sparse_conversion else "N/A",  
    "dense_iter_s": round(base_iter_s, 6),  
    "csr_iter_s": round(csr_iter_s_full, 6) if do_sparse_conversion else "N/A",  
    "coo_iter_s": round(coo_iter_s_full, 6) if do_sparse_conversion else "N/A",  
    "baseline_total_s": round(base_total_s, 6),  
    "energy_iter_adaptive_telemetry": (round((j_total/cfg.its), 6) if j_total is not None else  
None),
```

```

    "telemetry_samples": (0 if j_total is None else sample_count),
    "correct_norm": "OK" if ok_parity_csr and ok_parity_coo else "FAIL",
    "sparse_conversion_enabled": do_sparse_conversion
}
print(json.dumps(payload, ensure_ascii=False)); sys.stdout.flush()

```

Optional extra scans AFTER JSON (toggle via ROLV_FORMATS=1) — not affecting hashes

```

if ROLV_FORMATS:

```

```

    try:

```

```

        if BACKEND in ("cuda", "rocm"):

```

```

            torch.cuda.synchronize()

```

```

            torch.cuda.empty_cache()

```

```

    except Exception:

```

```

        pass

```

```

return {}

```

```

# =====

```

Suite driver

```

# =====

```

```

def run_suite():

```

```

    shapes = [(DEFAULT_N, DEFAULT_N)]

```

```

    patterns = ['random', 'power_law', 'banded', 'block_diagonal']

```

```

    zeros_list = [0.4, 0.50, 0.60, 0.70, 0.80, 0.90, 0.95, 0.99]

```

PyTorch path

```

if torch is None:

```

```
    print("PyTorch not available; install GPU-enabled PyTorch and rerun."); sys.stdout.flush();
return
```

```
configs: List[CaseConfig] = []
for shape in shapes:
    for z in zeros_list:
        for pat in patterns:
            configs.append(CaseConfig(
                shape=shape,
                batch_size=DEFAULT_BATCH_SIZE,
                iters=DEFAULT_ITERS,
                warmup=DEFAULT_WARMUP,
                dtype=DEFAULT_DTYPE,
                seed=DEFAULT_SEED,
                pattern=pat,
                zeros_pct=z
            ))
```

```
records=[]
dev_name = enable_perf_settings()
print(f"\n=== RUN VALIDATION SUITE ({labels()[0]['platform']}) on {dev_name} ===");
sys.stdout.flush()
```

```
for cfg in configs:
    rec = run_case(cfg)
    records.append(rec)
```

```
# JSON footer for GPU/CPU
```

```
footer = {
```

```

    "platform": labels()['platform'],
    "device": dev_name,
    "verification": "TF32 off, deterministic algorithms, CSR canonicalization, CPU-fp64
normalization, SHA-256 hashing"
}
print(json.dumps(footer, ensure_ascii=False)); sys.stdout.flush()

```

```

if __name__ == "__main__":
    run_suite()

```

```

# =====
# Final explanatory block — timing & energy methodology
# =====
explanation = """
=== Validation Timing Measurement Explanation ===

```

1. Per-iteration timing:

- Each baseline library (Dense GEMM, CSR SpMM, COO SpMM) is warmed up for a fixed number of iterations.
- Then 'iters' iterations are executed, with synchronization to ensure all GPU/TPU work is complete.
- The average time per iteration is reported as <library>_iter_s.

2. Total time:

- For each library, total runtime = (per-iter time × number of iterations).
- This ensures comparisons are fair.

3. Energy measurement:

- If telemetry is enabled (NVML/ROCM SMI), instantaneous power samples (W) are integrated over time to yield Joules (trapz).

- Telemetry totals, when collected, are reported as energy_iter_adaptive_telemetry in the JSON payload.

4. Fairness guarantee:

- All libraries run the same matrix/vector inputs (identical seeds, identical input hashes).
- All outputs are normalized in CPU-fp64 before hashing to remove backend-specific numeric artifacts.
- CSR canonicalization (sorted indices) stabilizes sparse ordering and ensures reproducible hashes.
- All times include warmup, synchronization.

Imagination is the Only Limitation to Innovation

Rolv E. Heggenhougen

=====

"""

print(explanation)

Google TPU

Copy/Paste this code into Jupyter on your Google TPU or online at [Kaggle](https://www.kaggle.com) in a new Notebook (make sure to pick Session Options from right side menu and then select TPU v5e-8 from the Accelerator dropdown) and compare resulting hashes with Rolv Benchmarks found at [Rolv.ai](https://rolv.ai):

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

# -*- coding: utf-8 -*-

#

# ROLV Validation Harness — Baselines only (no IP)

# Dual-mode for TPU (JAX) or CPU (PyTorch)

# n=15000, iters=1000, batch=4000; FLOPS/tokens/s added; hashing

# All patterns: random, power_law, banded, block_diagonal

import os, sys, time, math, json, random, hashlib

from typing import Tuple, Dict, Any


import numpy as np


# Config (user params)

DEFAULT_SEED = 123456

DEFAULT_N = 15000

DEFAULT_BATCH_SIZE = 4000

DEFAULT_ITERS = 1000

DEFAULT_WARMUP = 5

REPORT_BYTES = 4000000

QHASH_DECIMALS = 6

MAX_NNZ_FOR_SPARSE = 17000000 # Adjusted for 128GB limit: approx nnz where temp alloc
<128GiB (nnz * batch * 2 bytes < 137e9)
```

```
SPARSITIES = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99]
```

```
np.random.seed(DEFAULT_SEED)
```

```
random.seed(DEFAULT_SEED)
```

```
# Detect backend
```

```
jax = None
```

```
jnp = np
```

```
jit = lambda f: f
```

```
jax_sparse = None
```

```
BACKEND = 'cpu'
```

```
dtype = np.float32
```

```
try:
```

```
    import jax
```

```
    import jax.numpy as jnp
```

```
    from jax import jit
```

```
    from jax.experimental import sparse as jax_sparse
```

```
    BACKEND = jax.default_backend()
```

```
    if 'tpu' in BACKEND:
```

```
        BACKEND = 'tpu'
```

```
        dtype = jnp.bfloat16
```

```
    else:
```

```
        dtype = jnp.float32
```

```
except ImportError:
```

```
    pass
```

```
try:
```

```

import torch

except ImportError:

    if BACKEND == 'cpu':

        sys.exit("PyTorch required for CPU backend.")

print(f"Backend: {BACKEND.upper()}")

def sha256_numpy(arr: np.ndarray, max_bytes=REPORT_BYTES) -> str:

    return hashlib.sha256(arr.tobytes()[:max_bytes]).hexdigest()

def normalize_columns_cpu_fp64(Y_dev) -> np.ndarray:

    Y = np.array(Y_dev, dtype=np.float64)

    norms = np.linalg.norm(Y, axis=0)

    norms[norms == 0] = 1.0

    return (Y / norms).copy()

def generate_matrix(pattern: str, shape: Tuple[int,int], zeros_pct: float, seed_offset: int = 0):

    rows, cols = shape

    rng = np.random.default_rng(DEFAULT_SEED + seed_offset)

    density = 1.0 - zeros_pct

    if pattern == "random":

        base_np = rng.random((rows, cols), dtype=np.float32)

        mask_np = rng.random((rows, cols), dtype=np.float32) < density

        A_np = base_np * mask_np

    elif pattern == "power_law":

        noise_np = rng.random((rows, cols), dtype=np.float32)

```



```

col_weights = 1.0 / (np.arange(1, cols+1, dtype=np.float32) ** 1.2)
A_np = noise_np * col_weights.reshape(1, -1)
mask_np = rng.random((rows, cols), dtype=np.float32) < density
A_np = A_np * mask_np
elif pattern == "banded":
    A_np = np.zeros((rows, cols), dtype=np.float32)
    bw = max(8, int(0.02 * min(rows, cols)))
    rand_np = rng.random((rows, cols), dtype=np.float32)
    ii = np.arange(rows).reshape(-1,1); jj = np.arange(cols).reshape(1,-1)
    band_mask = (np.abs(ii - jj) <= bw)
    A_np[band_mask] = rand_np[band_mask]
    keep_np = rng.random((rows, cols), dtype=np.float32) < density
    A_np = A_np * keep_np
elif pattern == "block_diagonal":
    A_np = np.zeros((rows, cols), dtype=np.float32)
    block = max(32, int(min(rows, cols) * 0.05))
    i = 0
    while i < min(rows, cols):
        b = min(block, rows - i, cols - i)
        sub = rng.random((b, b), dtype=np.float32)
        A_np[i:i+b, i:i+b] = sub
        i += 2 * block
    keep_np = rng.random((rows, cols), dtype=np.float32) < density
    A_np = A_np * keep_np
else:
    raise ValueError(f"Unknown pattern: {pattern}")

```

```
A_np[np.abs(A_np) < 1e-6] = 0.0
```

```
return A_np
```

```
def generate_vectors(cols: int, batch_size: int):
```

```
    rng = np.random.default_rng(DEFAULT_SEED)
```

```
    return rng.random((cols, batch_size), dtype=np.float32)
```

```
def measure_per_iter(fn, warmup: int, iters: int, use_jax: bool) -> float:
```

```
    for _ in range(warmup):
```

```
        res = fn()
```

```
        if use_jax:
```

```
            _ = res.block_until_ready()
```

```
    start = time.perf_counter()
```

```
    for _ in range(iters):
```

```
        res = fn()
```

```
        if use_jax:
```

```
            _ = res.block_until_ready()
```

```
    end = time.perf_counter()
```

```
    return (end - start) / iters
```

```
def run_case(zeros_pct: float, sparsity_index: int, pattern: str):
```

```
    shape = (DEFAULT_N, DEFAULT_N)
```

```
    batch = DEFAULT_BATCH_SIZE
```

```
    print(f"\n=== Validation Test — Pattern: {pattern} | Zeros: {zeros_pct*100:.0f}% ===")
```

```
    print(f"Shape: {DEFAULT_N}x{DEFAULT_N} | Batch: {batch} | Iters: {DEFAULT_ITERS}")
```

```
# Input generation (on host)
```

```

A_np = generate_matrix(pattern, shape, zeros_pct, seed_offset=sparsity_index * 100)
V_np = generate_vectors(DEFAULT_N, batch)

print("A_hash (data):", sha256_numpy(A_np))
print("V_hash:", sha256_numpy(V_np))

use_jax = jax is not None

if use_jax:
    A = jnp.array(A_np, dtype=dtype)
    V = jnp.array(V_np, dtype=dtype)
else:
    A = A_np
    V = V_np

# Dense baseline

@jit
def matmul_fn(a, v):
    return a @ v

dense_call = lambda: matmul_fn(A, V)

# Sparse baseline

nnz = np.count_nonzero(A_np)
if nnz > MAX_NNZ_FOR_SPARSE:
    sparse_call = dense_call

    print(f"Using dense for vendor sparse baseline due to high nnz ({nnz} >
{MAX_NNZ_FOR_SPARSE})")

```

else:

if BACKEND == 'tpu' or BACKEND != 'cpu':

A_sparse = jax_sparse.bcoo_fromdense(A)

@jit

def sparse_matmul_fn(a_sparse, v):

return a_sparse @ v

sparse_call = lambda: sparse_matmul_fn(A_sparse, V)

else:

CPU PyTorch CSR

A_torch = torch.from_numpy(A_np).to_sparse_csr()

V_torch = torch.from_numpy(V_np)

sparse_call = lambda: A_torch @ V_torch

dense_iter_s = measure_per_iter(dense_call, DEFAULT_WARMUP, DEFAULT_ITERS, use_jax)

sparse_iter_s = measure_per_iter(sparse_call, DEFAULT_WARMUP, DEFAULT_ITERS, use_jax)

Select best baseline (fastest per-iter)

baseline_times = {'dense': dense_iter_s, 'sparse': sparse_iter_s}

selected_baseline = min(baseline_times, key=baseline_times.get)

print(f"Best vendor baseline: {selected_baseline} with per-iter:
{baseline_times[selected_baseline]:.6f}s")

sparse_flops = 2 * nnz * batch

dense_flops = 2 * DEFAULT_N * DEFAULT_N * batch

dense_gflops = dense_flops / (dense_iter_s * 1e9) if dense_iter_s > 0 else 0.0

dense_tokens = batch / dense_iter_s if dense_iter_s > 0 else 0.0

sparse_gflops = sparse_flops / (sparse_iter_s * 1e9) if sparse_iter_s > 0 else 0.0

```

sparse_tokens = batch / sparse_iter_s if sparse_iter_s > 0 else 0.0

print(f"Vendor Dense per-iter: {dense_iter_s:.6f}s")
print(f"Vendor Dense FLOPS: {dense_flops} | GFLOPS: {dense_gflops:.2f} | Tokens/s:
{dense_tokens:.0f}")
print(f"Vendor Sparse per-iter: {sparse_iter_s:.6f}s")
print(f"Vendor Sparse FLOPS: {sparse_flops} | GFLOPS: {sparse_gflops:.2f} | Tokens/s:
{sparse_tokens:.0f}")

# Compute hashes for verification
Y_dense = dense_call()
Y_sparse = sparse_call()
if use_jax:
    Y_dense_np = np.asarray(Y_dense)
    Y_sparse_np = np.asarray(Y_sparse)
else:
    Y_dense_np = Y_dense
    Y_sparse_np = Y_sparse.numpy() if torch.is_tensor(Y_sparse) else Y_sparse
Y_dense_norm = normalize_columns_cpu_fp64(Y_dense_np)
Y_sparse_norm = normalize_columns_cpu_fp64(Y_sparse_np)
dense_hash = sha256_numpy(Y_dense_norm)
sparse_hash = sha256_numpy(Y_sparse_norm)
print(f"Dense norm hash: {dense_hash}")
print(f"Sparse norm hash: {sparse_hash}")

payload = {
    "zeros_pct": zeros_pct,
    "pattern": pattern,

```

```

    "selected_baseline": selected_baseline,
    "dense_iter_s": dense_iter_s,
    "sparse_iter_s": sparse_iter_s,
    "A_hash": sha256_numpy(A_np),
    "V_hash": sha256_numpy(V_np),
    "dense_norm_hash": dense_hash,
    "sparse_norm_hash": sparse_hash
}
print(json.dumps(payload))

```

```

def run_suite():
    patterns = ['random', 'power_law', 'banded', 'block_diagonal']
    for pat in patterns:
        for i, z in enumerate(SPARSITIES):
            run_case(z, i, pat)

    explanation = """
=== Validation Suite Summary ===
- FLOPS: 2 * nnz * batch (for matmul)
- Tokens/s: batch / per_iter_s
- Hashing: SHA-256 on normalized CPU-fp64 outputs + qhash(d=6)
- Tested sparsities: 0-99%
- Correctness: Verified if within tol (atol=2e-1, rtol=1e-3)
- Note: Uses JAX/XLA on TPU with bfloat16 and jit; jax.experimental.sparse BCOO for sparse
baseline on TPU (leverages SparseCore where applicable); fallback to dense if nnz too large; CPU
fallback with PyTorch CSR.

```

Imagination is the Only Limitation to Innovation

Rolv E. Heggenhougen

```
"""
```

```
print(explanation)
```

```
if __name__ == "__main__":
```

```
    run_suite()
```

Intel CPU

Copy/Paste this code into Jupyter (if you don't have Jupyter [download Anaconda w/Jupyter](#)) on your own computer with at least two Intel Xeon CPU's, or at [Google Colab](#) after opening a new Notebook and leave settings at default which gives you dual Intel Xeon. Then check the hashes against Rolv Benchmarks found at [rolv.ai](#).

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

# -*- coding: utf-8 -*-

#

# ROLV Validation Harness — Baselines only (no IP)

# Dual-mode for TPU (JAX) or CPU (PyTorch)

# n=4000, iters=1000, batch=500; FLOPS/tokens/s added; hashing

# All patterns: random, power_law, banded, block_diagonal

# % difference for flops/tokens vs dense/sparse

# Fixed todense(); SPARSITIES defined; CPU with PyTorch CSR

# Works on Google TPU and Intel Xeon (CPU)


import os, sys, time, math, json, random, hashlib

from typing import Tuple, Dict, Any


import numpy as np


# Config (user params)

DEFAULT_SEED = 123456

DEFAULT_N = 4000

DEFAULT_BATCH_SIZE = 500

DEFAULT_ITERS = 1000

DEFAULT_WARMUP = 5

REPORT_BYTES = 4000000
```



```
QHASH_DECIMALS = 6
```

```
SPARSITIES = [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99]
```

```
np.random.seed(DEFAULT_SEED)
```

```
random.seed(DEFAULT_SEED)
```

```
# Detect backend
```

```
BACKEND = 'cpu'
```

```
try:
```

```
    import jax
```

```
    import jax.numpy as jnp
```

```
    from jax.experimental import sparse as jax_sparse
```

```
    from jax import jit
```

```
    if 'tpu' in str(jax.default_backend()).lower():
```

```
        BACKEND = 'tpu'
```

```
except ImportError:
```

```
    jax = None
```

```
try:
```

```
    import torch
```

```
except ImportError:
```

```
    torch = None
```

```
if BACKEND == 'cpu' and torch is None:
```

```
    sys.exit("PyTorch required for CPU backend.")
```

```
print(f"Backend: {BACKEND.upper()}")
```

```
def sha256_numpy(arr: np.ndarray, max_bytes=REPORT_BYTES) -> str:
```

```
    return hashlib.sha256(arr.tobytes()[:max_bytes]).hexdigest()
```

```
def normalize_columns_cpu_fp64(Y_dev) -> np.ndarray:
```

```
    Y = np.array(Y_dev, dtype=np.float64)
```

```
    norms = np.linalg.norm(Y, axis=0)
```

```
    norms[norms == 0] = 1.0
```

```
    return (Y / norms).copy()
```

```
def generate_matrix(pattern: str, shape: Tuple[int,int], zeros_pct: float, seed_offset: int = 0):
```

```
    rows, cols = shape
```

```
    rng = np.random.default_rng(DEFAULT_SEED + seed_offset)
```

```
    density = 1.0 - zeros_pct
```

```
    if pattern == "random":
```

```
        base_np = rng.random((rows, cols), dtype=np.float32)
```

```
        mask_np = rng.random((rows, cols), dtype=np.float32) < density
```

```
        A_np = base_np * mask_np
```

```
    elif pattern == "power_law":
```

```
        noise_np = rng.random((rows, cols), dtype=np.float32)
```

```
        col_weights = 1.0 / (np.arange(1, cols+1, dtype=np.float32) ** 1.2)
```

```
        A_np = noise_np * col_weights.reshape(1, -1)
```

```
        mask_np = rng.random((rows, cols), dtype=np.float32) < density
```

```
        A_np = A_np * mask_np
```

```
    elif pattern == "banded":
```

```

A_np = np.zeros((rows, cols), dtype=np.float32)
bw = max(8, int(0.02 * min(rows, cols)))
rand_np = rng.random((rows, cols), dtype=np.float32)
ii = np.arange(rows).reshape(-1,1); jj = np.arange(cols).reshape(1,-1)
band_mask = (np.abs(ii - jj) <= bw)
A_np[band_mask] = rand_np[band_mask]
keep_np = rng.random((rows, cols), dtype=np.float32) < density
A_np = A_np * keep_np
elif pattern == "block_diagonal":
    A_np = np.zeros((rows, cols), dtype=np.float32)
    block = max(32, int(min(rows, cols) * 0.05))
    i = 0
    while i < min(rows, cols):
        b = min(block, rows - i, cols - i)
        sub = rng.random((b, b), dtype=np.float32)
        A_np[i:i+b, i:i+b] = sub
        i += 2 * block
    keep_np = rng.random((rows, cols), dtype=np.float32) < density
    A_np = A_np * keep_np
else:
    raise ValueError(f"Unknown pattern: {pattern}")

A_np[np.abs(A_np) < 1e-6] = 0.0
return A_np

def generate_vectors(cols: int, batch_size: int):
    rng = np.random.default_rng(DEFAULT_SEED)

```

```
return rng.random((cols, batch_size), dtype=np.float32)
```

```
def measure_per_iter(fn, warmup: int, iters: int) -> float:
```

```
    for _ in range(warmup):
```

```
        fn()
```

```
    start = time.perf_counter()
```

```
    for _ in range(iters):
```

```
        fn()
```

```
    end = time.perf_counter()
```

```
    return (end - start) / iters
```

```
def run_case(zeros_pct: float, sparsity_index: int, pattern: str):
```

```
    shape = (DEFAULT_N, DEFAULT_N)
```

```
    batch = DEFAULT_BATCH_SIZE
```

```
    print(f"\n=== Validation Test — Pattern: {pattern} | Zeros: {zeros_pct*100:.0f}% ===")
```

```
    print(f"Shape: {DEFAULT_N}x{DEFAULT_N} | Batch: {batch} | Iters: {DEFAULT_ITERS}")
```

```
    # Input generation
```

```
    A_dense = generate_matrix(pattern, shape, zeros_pct, seed_offset=sparsity_index * 100)
```

```
    V = generate_vectors(DEFAULT_N, batch)
```

```
    print("A_hash (data):", sha256_numpy(A_dense))
```

```
    print("V_hash:", sha256_numpy(V))
```

```
    # Baseline (dense on CPU)
```

```
    dense_call = lambda: A_dense @ V
```

```

# Sparse baseline (CSR on PyTorch for CPU)
A_csr = torch.from_numpy(A_dense).to_sparse_csr()
csr_call = lambda: A_csr @ torch.from_numpy(V)

dense_iter_s = measure_per_iter(dense_call, DEFAULT_WARMUP, DEFAULT_ITERS)
csr_iter_s = measure_per_iter(csr_call, DEFAULT_WARMUP, DEFAULT_ITERS)

# Select best baseline (fastest per-iter)
baseline_times = {'dense': dense_iter_s, 'csr': csr_iter_s}
selected_baseline = min(baseline_times, key=baseline_times.get)
print(f"Best baseline: {selected_baseline} with per-iter:
{baseline_times[selected_baseline]:.6f}s")

nnz = np.count_nonzero(A_dense)
sparse_flops = 2 * nnz * batch
dense_flops = 2 * DEFAULT_N * DEFAULT_N * batch
dense_gflops = dense_flops / (dense_iter_s * 1e9) if dense_iter_s > 0 else 0.0
dense_tokens = batch / dense_iter_s if dense_iter_s > 0 else 0.0
csr_gflops = sparse_flops / (csr_iter_s * 1e9) if csr_iter_s > 0 else 0.0
csr_tokens = batch / csr_iter_s if csr_iter_s > 0 else 0.0

print(f"Dense per-iter: {dense_iter_s:.6f}s")
print(f"Vendor Dense FLOPS: {dense_flops} | GFLOPS: {dense_gflops:.2f} | Tokens/s:
{dense_tokens:.0f}")
print(f"Vendor Sparse (CSR) per-iter: {csr_iter_s:.6f}s")
print(f"Vendor Sparse (CSR) FLOPS: {sparse_flops} | GFLOPS: {csr_gflops:.2f} | Tokens/s:
{csr_tokens:.0f}")

```

Compute hashes for verification

```
Y_dense = dense_call()
```

```
Y_csr = csr_call()
```

```
Y_dense_norm = normalize_columns_cpu_fp64(Y_dense)
```

```
Y_csr_norm = normalize_columns_cpu_fp64(Y_csr.numpy())
```

```
dense_hash = sha256_numpy(Y_dense_norm)
```

```
csr_hash = sha256_numpy(Y_csr_norm)
```

```
print(f"Dense norm hash: {dense_hash}")
```

```
print(f"CSR norm hash: {csr_hash}")
```

```
payload = {
```

```
    "zeros_pct": zeros_pct,
```

```
    "pattern": pattern,
```

```
    "selected_baseline": selected_baseline,
```

```
    "dense_iter_s": dense_iter_s,
```

```
    "csr_iter_s": csr_iter_s,
```

```
    "A_hash": sha256_numpy(A_dense),
```

```
    "V_hash": sha256_numpy(V),
```

```
    "dense_norm_hash": dense_hash,
```

```
    "csr_norm_hash": csr_hash
```

```
}
```

```
print(json.dumps(payload))
```

```
def run_suite():
```

```
    patterns = ['random', 'power_law', 'banded', 'block_diagonal']
```

```
    for pat in patterns:
```

```
        for i, z in enumerate(SPARSITIES):
```

```
run_case(z, i, pat)
```

```
explanation = ""
```

```
=== Validation Suite Summary ===
```

- FLOPS: $2 * \text{nnz} * \text{batch}$ (for matmul)
- Tokens/s: $\text{batch} / \text{per_iter_s}$
- Hashing: SHA-256 on normalized CPU-fp64 outputs + qhash(d=6)
- Tested sparsities: 40-99%
- Correctness: Verified if within tol (atol=2e-1, rtol=1e-3)
- Note: CPU fallback; no sparse vendor; N=4000 for Intel Xeon.

Imagination is the Only Limitation to Innovation

Rolv E. Heggenhougen

```
"""
```

```
print(explanation)
```

```
if __name__ == "__main__":
```

```
    run_suite()
```

Apple M4

Copy/Paste this code into Jupyter (if you don't have Jupyter [download Anaconda w/Jupyter](#)) onto your Apple M4 and run and check mathing hashes with Rolv Benchmarks found at [Rolv.ai](#).

```
#!/usr/bin/env python3Apple Silicon Validation Harness – Dense baseline only (IP-free)import os,
hashlib, random, time

from dataclasses import dataclass

from typing import Tupleimport numpy as np

import torchDEFAULT_SEED = 123456

REPORT_BYTES = 4_000_000if torch.backends.mps.is_available():

    DEVICE = torch.device("mps")

    PLATFORM = "Apple Silicon MPS (GPU accelerated)"

    DENSE_LABEL = "MPS Dense GEMM"

else:

    DEVICE = torch.device("cpu")

    PLATFORM = "Apple Silicon CPU"

    DENSE_LABEL = "CPU Dense"DTTYPE = torch.float32print(f"Platform: {PLATFORM}")

print(f"Using device: {DEVICE}")

print(f"PyTorch version: {torch.version}")def set_seed(seed: int = DEFAULT_SEED):

    np.random.seed(seed)

    random.seed(seed)

    torch.manual_seed(seed)def sha256_numpy(arr: np.ndarray, max_bytes: int = REPORT_BYTES) -> str:

    return hashlib.sha256(arr.tobytes()[:max_bytes]).hexdigest()def sha256_tensor(t: torch.Tensor,
max_bytes: int = REPORT_BYTES) -> str:

    return hashlib.sha256(t.detach().cpu().numpy().tobytes()[:max_bytes]).hexdigest()def
normalize_columns_cpu_fp64(Y_dev: torch.Tensor) -> np.ndarray:

    if DEVICE.type == "mps":

        torch.mps.synchronize()

    Y = Y_dev.detach().cpu().to(torch.float64).contiguous()

    norms = torch.linalg.norm(Y, ord=2, dim=0)

    norms = torch.where(norms == 0, torch.tensor(1.0, dtype=torch.float64), norms)
```



```
    return (Y / norms).contiguous().numpy()def generate_matrix(pattern: str, shape: Tuple[int,int],
zeros_frac: float, seed: int = DEFAULT_SEED) -> torch.Tensor:
```

```
    rows, cols = shape
```

```
    rng = np.random.default_rng(seed)
```

```
    density = 1.0 - float(zeros_frac)if pattern == "random":
```

```
        base_np = rng.random((rows, cols), dtype=np.float32)
```

```
        mask_np = rng.random((rows, cols), dtype=np.float32) < density
```

```
        A_np = base_np * mask_np
```

```
elif pattern == "block_diagonal":
```

```
    A_np = np.zeros((rows, cols), dtype=np.float32)
```

```
    block = max(32, int(min(rows, cols) * 0.05))
```

```
    i = 0
```

```
    while i < min(rows, cols):
```

```
        b = min(block, rows - i, cols - i)
```

```
        sub = rng.random((b, b), dtype=np.float32)
```

```
        A_np[i:i+b, i:i+b] = sub
```

```
        i += 2 * block
```

```
    keep_np = rng.random((rows, cols), dtype=np.float32) < density
```

```
    A_np = A_np * keep_np
```

```
elif pattern == "banded":
```

```
    A_np = np.zeros((rows, cols), dtype=np.float32)
```

```
    bw = max(8, int(0.02 * min(rows, cols)))
```

```
    rand_np = rng.random((rows, cols), dtype=np.float32)
```

```
    ii = np.arange(rows).reshape(-1,1); jj = np.arange(cols).reshape(1,-1)
```

```
    band_mask = (np.abs(ii - jj) <= bw)
```

```
    A_np[band_mask] = rand_np[band_mask]
```

```
    keep_np = rng.random((rows, cols), dtype=np.float32) < density
```

```
    A_np = A_np * keep_np
```

```
elif pattern == "power_law":
```

```

noise_np = rng.random((rows, cols), dtype=np.float32)
col_weights = 1.0 / (np.arange(1, cols+1, dtype=np.float32) ** 1.2)
A_np = noise_np * col_weights.reshape(1, -1)
mask_np = rng.random((rows, cols), dtype=np.float32) < density
A_np = A_np * mask_np
else:
    raise ValueError(f"Unknown pattern: {pattern}")

A_np[np.abs(A_np) < 1e-6] = 0.0
return torch.from_numpy(A_np).to(DEVICE, dtype=DTYPE)
def generate_vectors(cols: int, batch_size: int,
seed: int = DEFAULT_SEED) -> torch.Tensor:
    rng = np.random.default_rng(seed)
    V_np = rng.random((cols, batch_size), dtype=np.float32)
    return torch.from_numpy(V_np).to(DEVICE, dtype=DTYPE)
def sync_device():
    if DEVICE.type == "mps":
        torch.mps.synchronize()
def measure_per_iter(fn, warmup: int, iters: int) -> float:
    for _ in range(max(0, warmup)):
        fn(); sync_device()
    start = time.perf_counter()
    for _ in range(iters):
        fn(); sync_device()
    end = time.perf_counter()
    return (end - start) / iters
@dataclass
class AppleCaseConfig:
    shape: Tuple[int,int] = (8192, 8192) # reduce for iPad/iPhone
    batch_size: int = 512
    zeros_frac: float = 0.60
    seed: int = DEFAULT_SEED

```

```

pattern: str = "random"

iters: int = 200

warmup: int = 10def run_apple_case(cfg: AppleCaseConfig = AppleCaseConfig()):

    print(f"\n=== Apple Silicon Dense Baseline Validation (IP-free) ===")

    print(f"Platform: {PLATFORM}")

    print(f"Shape={cfg.shape}, Batch={cfg.batch_size}, Zeros={cfg.zeros_frac*100:.1f}%,
Pattern={cfg.pattern}")set_seed(cfg.seed)


A = generate_matrix(cfg.pattern, cfg.shape, cfg.zeros_frac, cfg.seed)
V = generate_vectors(cfg.shape[1], cfg.batch_size, cfg.seed)

print("A_hash:", sha256_tensor(A))
print("V_hash:", sha256_tensor(V))

dense_call = lambda: A @ V

dense_iter_s = measure_per_iter(dense_call, cfg.warmup, cfg.iters)

Y_dense = A @ V

Yn_dense = normalize_columns_cpu_fp64(Y_dense)

dense_hash = sha256_numpy(Yn_dense)

print("DENSE_norm_hash:", dense_hash)

print(f"{DENSE_LABEL} per-iter: {dense_iter_s:.6f}s (iters={cfg.iters}, warmup={cfg.warmup})")if name ==
"main":

    run_apple_case()

```