# Energy Characterization of V-JEPA2 on the SOCS Teaching Cluster

Athmane Benarous
COMP 597 – Sustainability in Systems Design

## Abstract

This report studies the energy behavior of the V-JEPA2 training workload on the SOCS teaching cluster. The required experiment matrix contains 18 runs: three measurement modes, two physically feasible batch sizes, and three repetitions per configuration, each capped at five minutes. The measurements combine a true baseline, coarse end-to-end GPU energy via CodeCarbon, and fine-grained per-phase timing with 500 ms system timelines. The main result is that V-JEPA2 is strongly GPU-centric but not fully GPU-saturating: forward and backward phases dominate step time, the GPU repeatedly operates below 95% utilization while still drawing high power, and increasing the physical batch size from 1 to 2 improves processed-sample throughput by 42.4% while reducing GPU energy per 1000 processed samples by 26.8%.

## 1 Workload Description

V-JEPA2 is a self-supervised video Transformer. Instead of predicting labels or reconstructing pixels, it learns by predicting masked target representations from visible video context. Each *step* is one parameter update and contains three required *phases*: forward pass, backward pass, and optimizer update. In this workload, the forward pass includes the trainable online encoder, the predictor, and the frozen EMA target encoder; the optimizer update is followed by a lightweight EMA refresh of the target encoder. An *epoch* is one complete pass over the dataset.

The final workload uses the reference `vit_huge` V-JEPA2 configuration with 16-frame clips, a 224×224 crop, patch size 16, tubelet size 2, predictor depth 12, predictor embedding dimension 384, and `bfloat16` activations. These settings were kept close to the benchmarked workload rather than reduced to a smaller proxy model, so that the measurements remain representative of a realistic end-to-end video training pipeline.

The data source is a FakeVideo CSV manifest stored on the course shared partition. The manifest points to actual video files consumed by the bundled JEPA video loader, so the workload still exercises video decoding, masking, and training logic rather than a synthetic tensor-only path. I use this setup (rather than random tensors) so that I/O and dataloader cost appear in the measurements where they would in a real training job, which matters when interpreting CPU utilization and step time. All runs were executed on `gpu-teach-03` with one NVIDIA RTX 5000 Ada GPU (32 GB), an AMD EPYC 9474F CPU, four allocated CPU cores, and a 4 GB Slurm RAM allocation; a single GPU keeps GPU energy attribution unambiguous (no multi-device contention in the reported numbers).

Table 1: Workload and hardware configuration used for all final measurements.

| Category | Configuration |
|---|---|
| Model | V-JEPA2 `vit_huge` video Transformer |
| Task | Self-supervised video representation learning |
| Data | FakeVideo CSV manifest on shared storage |
| Clip configuration | 16 frames, 224 crop, patch 16, tubelet 2 |
| Predictor | Depth 12, embedding dimension 384 |
| Precision | `bfloat16` |
| GPU | 1× NVIDIA RTX 5000 Ada, 32 GB |
| CPU | AMD EPYC 9474F, 4 allocated cores |
| RAM allocation | 4 GB Slurm RAM allocation |

## 2 Methodology

The main experiment matrix follows the class guidance exactly where physically possible. Each configuration runs for 5 minutes and is repeated 3 times. Three measurement modes are compared: **Mode 1 Baseline**, which measures end-to-end wall-clock time without in-loop instrumentation; **Mode 2 CodeCarbon**, which records end-to-end GPU energy; and **Mode 3 Fine-Grained**, which adds CUDA-synchronized per-phase timing and 500 ms system timelines.

Design intent: across the 18 cells, the only deliberate differences are batch size, measurement mode, and repetition index; model architecture, learning rate, clip geometry, and the five-minute wall-clock budget are shared so that comparisons isolate instrumentation and batch-size effects. Each run stops when the configured maximum training time is reached (not when a fixed epoch count is finished), which matches the assignment's fixed-duration runs and avoids epochs-based stopping from dominating wall-clock variance.

Table 2: Main experiment matrix used for the report.

| Mode | Repetitions | Metrics |
|---|---|---|
| Baseline | 3 / batch | End-to-end wall-clock |
| CodeCarbon | 3 / batch | Wall-clock + total GPU energy |
| Fine-grained | 3 / batch | Phase timing + 500 ms GPU/CPU/memory timelines |

Batch-size selection is the one unavoidable deviation from the generic three-point recipe. On this 32 GB GPU, the largest feasible *physical* power-of-two batch size is 2, and the next smaller physical power-of-two is 1. I intentionally did not fabricate a third point using gradient accumulation, because that changes the amount of work inside a class-defined step and weakens comparability. The memory timelines later show why this choice is justified: peak mean GPU memory increases from 17.7 GB at batch size 1 to 27.7 GB at batch size 2, leaving little headroom for a larger physical batch.

All reported runs use a PyTorch `DataLoader` with `num_workers=0` (single-process loading) so every configuration in the main matrix is directly comparable. I briefly considered an additional experiment varying `num_workers` to study

pipeline overlap between decoding and the GPU, but abandoned it: it would have expanded the integration surface (vendored JEPA paths, multiprocessing start methods, and cluster launch constraints) without changing the core conclusions about phase time, energy, and utilization below. No worker-count results appear in this report.

Fine-grained timing is implemented with explicit GPU synchronization before and after the phase being measured:

```
torch.cuda.synchronize()
t0 = time.perf_counter_ns()
# phase body
torch.cuda.synchronize()
t1 = time.perf_counter_ns()
```

This avoids the classic error of timing asynchronous GPU launches from the CPU. Per-phase summaries are generated from the fine-grained CSVs after excluding the first warm-up step, using `phase10_analysis.py -warmup-steps 1`. Energy and system telemetry are sampled every 500 ms, matching the Ed guidance on NVML resolution. The fine-grained path records GPU utilization, GPU memory, GPU power, PyTorch allocated and reserved memory, and system-wide CPU utilization via `psutil`. Underutilization windows are flagged whenever the mean GPU utilization of a 500 ms bin falls below 95%.

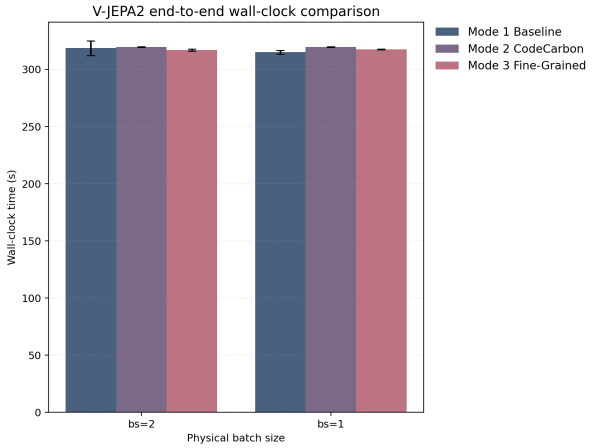
# 3 Results

## 3.1 End-to-End Time and Total GPU Energy

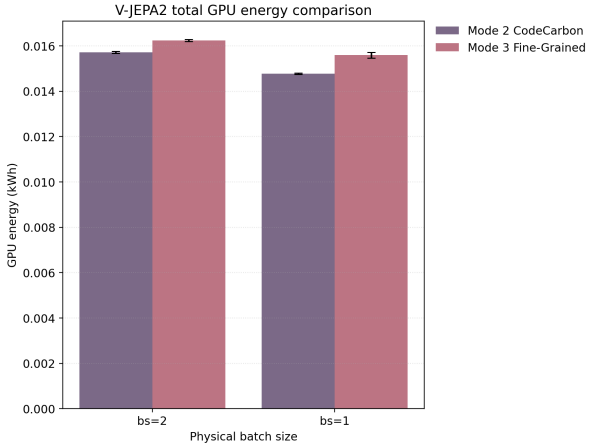
Figure 1 aggregates `run_metadata.json` wall-clock times and total GPU energy (CodeCarbon / fine-grained `run_summary`) per mode and batch size (means ± std over three runs). Mode 1 wall-clock is the external process duration; Modes 2–3 add CodeCarbon at 500 ms resolution, and fine-grained GPU energy is taken from the final CodeCarbon GPU-energy value at shutdown so it is comparable to Mode 2.

The wall-clock results show that the instrumentation stack remains lightweight. At batch size 1, the mean wall-clock time is 315.04 s for the baseline, 319.65 s with CodeCarbon, and 317.44 s with fine-grained measurement. At batch size 2, the corresponding times are 318.66 s, 319.57 s, and 316.92 s. Relative to the true baseline, the overhead stays within about 1.5%, well below the 5% ceiling discussed on Ed. The modes are therefore comparable for absolute throughput and energy conclusions.

The GPU-energy results show the same stability. At batch size 1, coarse CodeCarbon energy is 0.014775 kWh and fine-grained GPU energy is 0.015592 kWh; at batch size 2, the corresponding values are 0.015708 kWh and 0.016236 kWh. The fine-grained path is therefore only 3.4–5.5% higher than the coarse measurement. This is small enough that the fine-grained results can be interpreted as faithful workload characterizations rather than artifacts of the instrumentation itself. If wall-clock or energy were materially inflated, the phase and timeline sections would be untrustworthy; the close agreement justifies using Mode 3 below.




(a) End-to-end wall-clock.




(b) Total GPU energy.

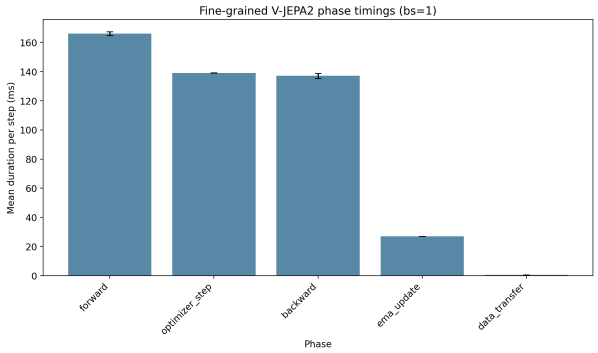
Figure 1: Required end-to-end measurements. Bars show means over the 3 report runs; error bars show standard deviation.

## 3.2 Per-Phase Time Decomposition

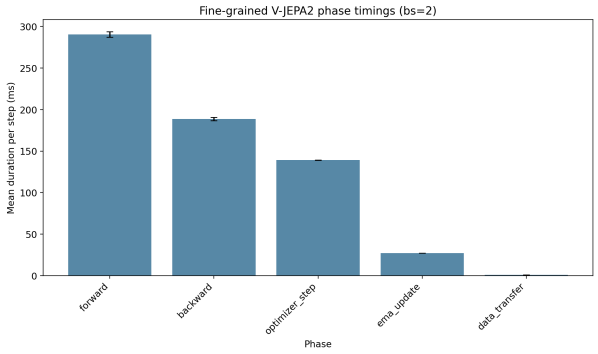
Figure 2 shows per-phase CUDA-synchronized durations (first warm-up step excluded), averaged over fine-grained runs; `phase_timing_run*.csv` supplies one row per phase instance and `phase10_analysis.py` aggregates after `-warmup-steps 1`.

At batch size 1, the mean phase times are 166.1 ms for forward, 137.1 ms for backward, 139.1 ms for optimizer step, and 27.0 ms for EMA update. At batch size 2, forward increases sharply to 290.6 ms and backward to 188.7 ms, while the optimizer step remains essentially flat at 139.2 ms and the EMA update remains near 27.1 ms.

Forward and backward scale with tensor traffic and autograd work as the physical batch grows; the optimizer and EMA updates are comparatively stable *per-step* costs because they iterate parameters once per step regardless of batch size. That asymmetry is exactly what later drives better energy-per-sample at batch size 2 (Figure 6). Data transfer stays sub-millisecond on average, so it is too short for hardware energy attribution at 500 ms resolution; its share of step time appears instead in Figure 3a.




(a) Phase timing at batch size 1.




(b) Phase timing at batch size 2.

Figure 2: Average phase duration with standard deviation from the fine-grained runs. The first warm-up step is excluded from these steady-state summaries.

## 3.3 Phase-Time Shares and Proxy Energy Attribution

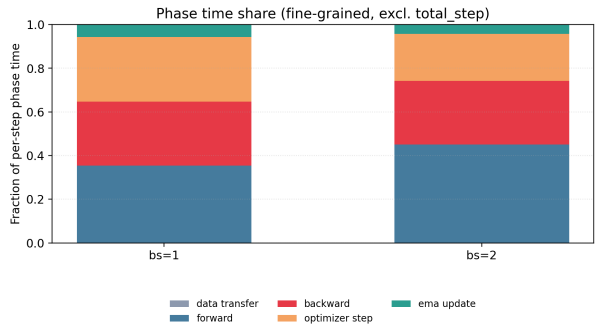
Figure 3a normalizes each phase's mean duration by the sum of all non-`total_step` phases so the bars sum to 100% of *phase time*. Figure 3b splits each run's total CodeCarbon GPU energy across phases in proportion to those mean durations, then averages across the three fine-grained runs (mean $\pm$ std). The proxy assumes roughly uniform effective GPU power across phases within a step so energy $\propto$ time share; hardware does not give true per-phase GPU energy at 500 ms, and sub-millisecond phases cannot align with CodeCarbon intervals. Numeric factors appear in `phase_energy_proxy_by_run.csv`.

At batch size 1, forward and backward together account for most of the phase time; at batch size 2, forward's *share* rises because its absolute time grows faster than the fixed optimizer cost. The proxy energy bars mirror that split: forward and backward dominate GPU energy in the same proportions as time because of the proportional allocation rule.

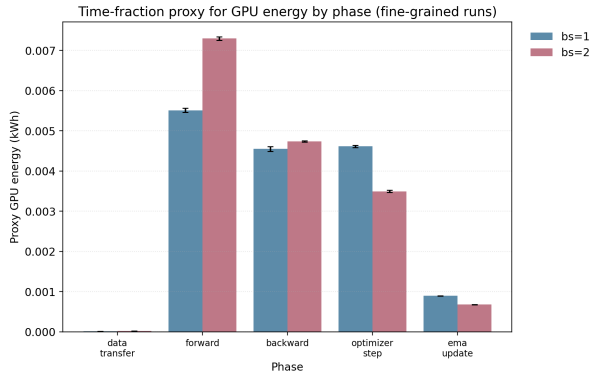
These plots make the batch-size trade-off explicit: larger batches shift time (and proxy energy) into compute-heavy phases but amortize fixed optimizer/EMA work, consistent with Figure 6.

## 3.4 Warmup and Steady-State Step Times

Figure 4 plots mean total step time (ms) versus step index for the first 50 steps (mean $\pm$ std across fine-grained runs at each batch size), from `step_summary_run*.csv` (`total_step_ms` per step), including step 1 to expose startup transients.




(a) Normalized phase-time shares.




(b) Proxy GPU energy by phase.

Figure 3: Phase composition of a step and proportional GPU-energy attribution for fine-grained runs. See `phase_energy_proxy_kwh.csv` for numeric values.

The rubric asks whether phase durations are "too short"; here the issue is *step-level* transients: allocator growth and cache behavior can stretch early steps. Excluding exactly one warm-up step for phase averages is therefore a conservative steady-state choice justified by the faster convergence visible in Figure 4.

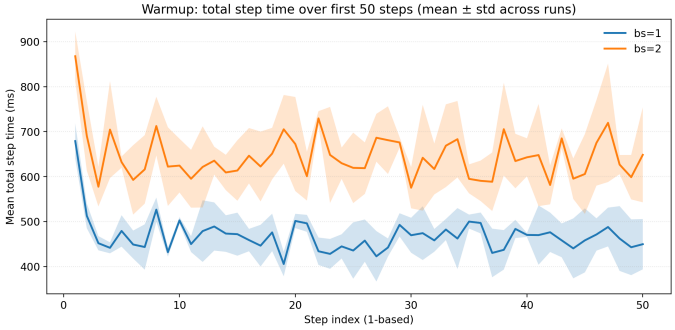
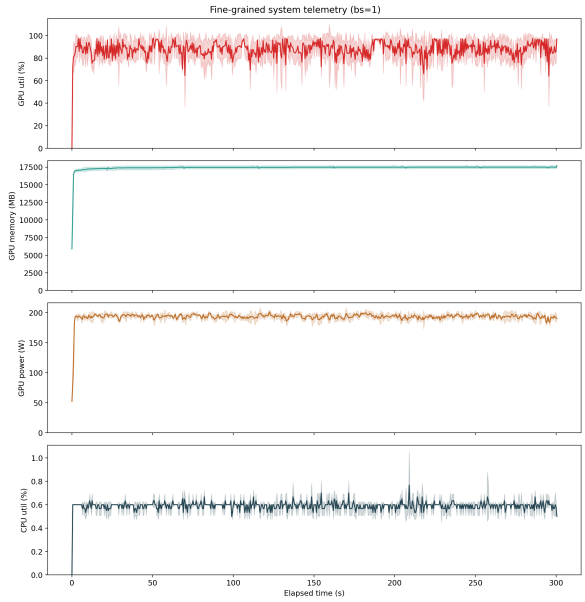



Figure 4: Warmup behavior of total step time before applying the steady-state cut used for phase aggregates.

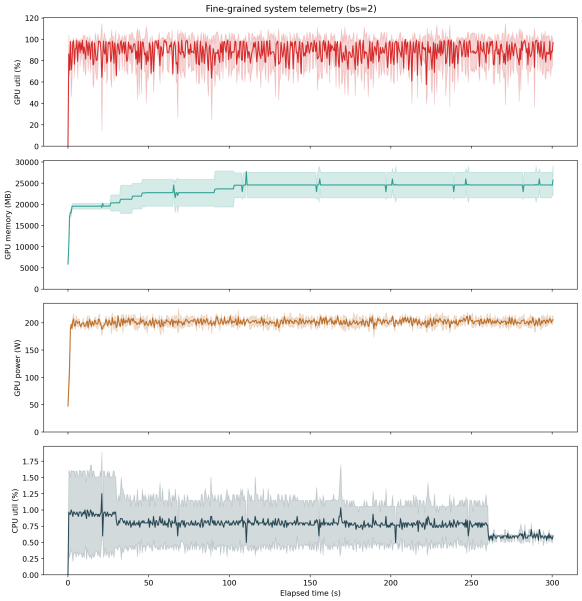
## 3.5 System Utilization, Power, and Underutilization Windows

Figure 5 shows four 500 ms panels (GPU utilization, GPU memory with NVML/PyTorch fallback, GPU power, system-wide CPU utilization), averaged across runs per batch size; `system_timeline_run*.csv` is binned in `phase10_analysis.py` with shaded $\pm 1$ std bands across repetitions.

The mean GPU utilization is 87.75% at batch size 1 and 88.49% at batch size 2. Even with the larger batch, 69.9% of the




(a) Telemetry at batch size 1 (four panels).




(b) Telemetry at batch size 2 (four panels).

Figure 5: Fine-grained 500 ms system telemetry averaged across the 3 report runs. Shaded bands show one standard deviation across runs.

timeline samples remain below the 95% threshold. The longest underutilization window at batch size 1 lasts 23.5 s (162.5–186.0 s) with mean GPU utilization 85.2% and mean GPU power 194.6 W. At batch size 2, the longest window lasts 8.5 s (169.5–178.0 s) with mean GPU utilization 83.9% and mean GPU power 199.4 W. The new power panel shows that "low" utilization is *not* idle: power stays high whenever the GPU is active, so the dominant sustainability issue is incomplete compute occupancy, not long idle gaps.

V-JEPA2 uses random 3-D multiblock masks, so per-step work varies. Near the memory ceiling (batch size 2), allocator noise widens the NVML memory band even when PyTorch-allocated tensors are stable. CPU utilization stays near zero because decoding and collation use few cores relative to the GPU in this integration (`num_workers=0` in the main matrix).

The CPU trace is system-wide `psutil` sampling on a shared node, so it is used to rule out a CPU bottleneck rather than to profile fine-grained CPU activity.

### 3.6 Batch-Size Effects and Instrumentation Overhead

Figure 6 (left) plots processed samples per second versus physical batch size per mode; (right) shows throughput of CodeCarbon and fine-grained modes relative to the true baseline at the same batch size. Throughput is `global_steps` × batch size divided by external wall-clock from metadata; overhead ratios come from `instrumentation_overhead_summary.csv`.

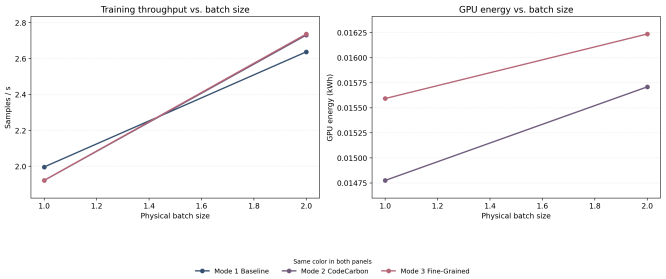
The correct cross-batch throughput metric is *processed samples per second*, not steps per second. In the fine-grained runs, throughput rises from 1.922 to 2.737 samples/s (+42.4%); CodeCarbon matches (+42.2%). GPU energy per 1000 processed samples drops from 0.025561 kWh to 0.018719 kWh (26.8%) in fine-grained data and similarly in CodeCarbon (25.2%).

Larger physical batches improve arithmetic intensity and reuse optimizer work across more samples. The overhead subplot confirms that these trends reflect the workload, not measurement bias.

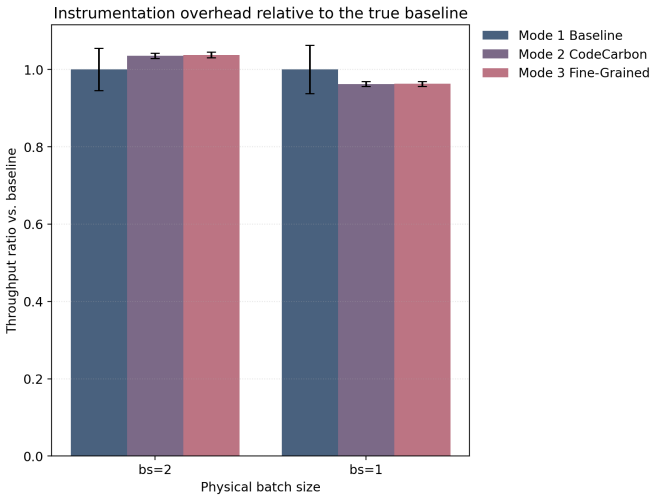
## 4 Discussion / Reflection

**Where energy is spent.** The phase decomposition (Figures 2, 3a, 3b) shows that forward and backward passes together consume the majority of every training step. At batch size 1, forward accounts for roughly 35% and backward for 29% of phase time; at batch size 2, forward's share rises to about 45% because its absolute duration nearly doubles while the optimizer and EMA phases stay flat. The proxy energy attribution mirrors these fractions, confirming that compute-bound phases dominate the GPU energy budget. The optimizer step, despite touching every parameter, is a fixed per-step cost that does not scale with batch size, and the EMA update is lighter still. This means that any strategy to reduce total energy should focus on the forward and backward passes, for example through mixed-precision optimizations, activation checkpointing, or architectural pruning of the predictor.

**Sources of inefficiency.** The timeline data (Figure 5) reveals that the GPU rarely reaches full occupancy: mean utilization is 87.75% at batch size 1 and 88.49% at batch size 2, with roughly 70% of 500 ms bins falling below the 95% threshold. Crucially, the power panel shows that GPU power draw remains high (190–200 W) even during these under-utilized intervals, so the GPU is not idling but rather executing kernels that do not saturate all streaming multiprocessors. This pattern is consistent with the V-JEPA2 architecture: the random 3-D multiblock masking produces variable-length token sequences per step, and the predictor's cross-attention between visible and masked tokens creates irregular workloads that the hardware cannot always pack efficiently. The near-zero CPU utilization confirms that the bottleneck is not data loading or host-side preprocessing but rather




(a) Cross-batch trends in throughput and total GPU energy.




(b) Instrumentation overhead relative to the baseline.

Figure 6: Batch-size effects and instrumentation costs derived from the aggregated report runs.

GPU compute scheduling.

**Batch-size trade-offs.** Doubling the physical batch from 1 to 2 yields a 42.4% throughput improvement (1.922 to 2.737 samples/s) while reducing GPU energy per 1000 samples by 26.8% (Figure 6). The gain comes from two sources: first, the compute-bound phases (forward and backward) scale sub-linearly with batch size because matrix operations benefit from larger tiles; second, the fixed per-step costs (optimizer, EMA) are amortized over twice as many samples. However, peak GPU memory jumps from 17.7 GB to 27.7 GB, leaving only about 4 GB of headroom on the 32 GB card. A physical batch size of 4 would exceed device memory, so further throughput scaling on this hardware would require gradient accumulation, which changes the definition of a step and weakens direct comparability. The diminishing returns are also visible in utilization: batch size 2 only raises mean GPU utilization by 0.74 percentage points, suggesting that the occupancy bottleneck is architectural rather than purely a matter of batch volume.

**Measurement overhead.** End-to-end overhead from the instrumentation stack stays within 1.5% of the true baseline across both batch sizes (Figure 1), and coarse CodeCarbon energy agrees with fine-grained GPU energy to within 3.4–5.5%. The supplementary validation sweep (Appendix A) reproduces the same wall-clock and energy means to within 0.24 s and 0.0002 kWh respectively, confirming that the conclusions are not artifacts of a single favorable repetition. The phase energy proxy should be interpreted as a time-proportional allocation of total run energy rather than true per-kernel measurements, but the close agreement between coarse and fine-grained totals gives confidence that the allocation is reasonable.

**Limitations and anomalies.** Only two physical power-of-two batch sizes fit this GPU, limiting the granularity of batch-size trend analysis. CPU telemetry is system-wide `psutil` on a shared cluster node, so it reflects co-tenant activity rather than isolated per-job CPU usage. The 500 ms sampling resolution of NVML and CodeCarbon means that sub-millisecond phases (e.g., data transfer) cannot be attributed individual energy costs; their contribution is folded into the time-proportional proxy. Finally, the FakeVideo manifest exercises the full video decoding pipeline but produces synthetic content, so data-dependent effects such as variable compression ratios or I/O contention from real storage are not captured in these measurements.

# Reproducibility and Raw Data

All raw CSV files used to generate the report figures are included in the structured run directories under `analysis_inputs/vjepa2/` and the supplementary validation sweep under `analysis_inputs/vjepa2_validation_extra/`. Aggregated tables written by the plotting scripts include `phase_energy_proxy_kwh.csv`, `phase_energy_proxy_by_run.csv`, and `step_timing_samples.csv`. Figures and CSVs are emitted under `analysis_outputs/vjepa2/` and `analysis_outputs/vjepa2_extended/` (validation figures). All source code is available at `https://github.com/Otm02/vjepa2-energy-char`.

# Use of LLM Tools

I used VSCode's GitHub Copilot for code suggestions and debugging during development, as well as for rephrasing and adjusting the LaTeX formatting of this report for correctness and readability. All ideas, experiment designs, methodology decisions, and interpretations of results are entirely my own.

# A Supplementary Validation Sweep

To strengthen confidence in the main 18-run matrix without changing the primary averages used in the report, I ran an additional 8-run validation sweep: 2 extra CodeCarbon runs and 2 extra fine-grained runs for each of the two batch sizes. Table 3 shows that the extra runs reproduce the same behavior closely. The coarse wall-clock means differ from the main matrix by at most 0.24 s, and the fine-grained GPU-energy means differ by at most 0.0002 kWh. Mean GPU utilization and peak GPU memory also stay within the same qualitative regime, reinforcing that the conclusions are stable rather than driven by one favorable repetition.

Table 3: Main report means versus supplementary validation means.

| BS | Metric | Main matrix | Validation sweep |
|---|---|---|---|
| 1 | CodeCarbon wall-clock (s) | 319.650 | 319.634 |
| 2 | CodeCarbon wall-clock (s) | 319.571 | 319.812 |
| 1 | Fine-grained wall-clock (s) | 317.438 | 316.646 |
| 2 | Fine-grained wall-clock (s) | 316.925 | 317.556 |
| 1 | Fine-grained GPU energy (kWh) | 0.0156 | 0.0155 |
| 2 | Fine-grained GPU energy (kWh) | 0.0162 | 0.0162 |
| 1 | Mean GPU utilization (%) | 87.74 | 88.39 |
| 2 | Mean GPU utilization (%) | 88.48 | 88.68 |

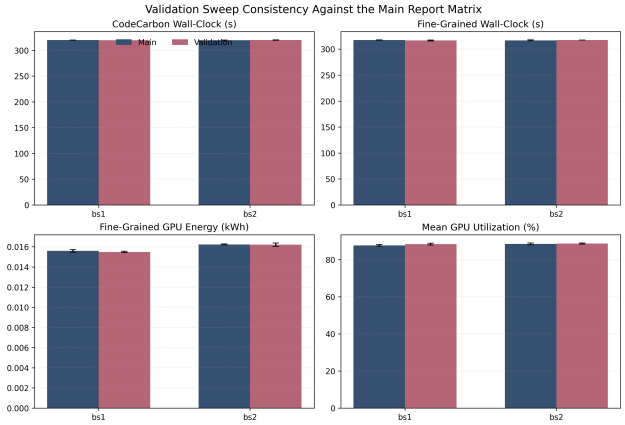



Figure 7: Validation sweep consistency figure comparing the main report matrix against the supplementary 8-run sweep.

# B Implementation Notes

The workload integration and measurement logic are part of the submitted code artifact. The key modifications are:

- a V-JEPA2 model wrapper that integrates the benchmark into the course infrastructure;
- a FakeVideo dataset adapter that forwards the CSV manifest into the JEPA video loader;
- a custom trainer with explicit phase boundaries for data transfer, forward, backward, optimizer step, and EMA update;
- a fine-grained statistics collector that records per-phase timings and 500 ms system timelines;
- a deterministic experiment runner and aggregation script that produce the CSV files and figures used in this report.