

GPU Efficiency in VLA I Model Training

Experiences and Benchmarks from Months of VLA I Vulnerability
Severity Classification Model Training

CIRCL - Computer Incident Response Center Luxembourg

2025-12-11

Contents

1	Preface	3
2	Environments used for benchmarking	3
2.1	GPU Architectures	3
2.2	Framework Versions	4
3	Dataset	4
4	Model training	5
4.1	Resulting models	5
4.2	Training Hyperparameters	5
4.2.1	A quick note on epochs and batches	5
4.3	Training results	7
4.3.1	Environment A	7
4.3.2	Environment B	7
4.3.3	Environment C	8
4.3.4	Comparisons	9
4.4	Key Observations	10
5	Benchmark Comparisons	11
5.1	Duration	11
5.2	Energy	12
5.3	Emissions	15
5.4	GPU Power	16
5.5	Energy vs. Duration	17
5.6	GPU Power vs. Duration	18
5.7	GPU Power vs. Energy	19
6	Resources	19
6.1	Related to CodeCarbon's RAM Energy Calculation	19
6.1.1	Estimation Methodology	20
6.1.2	Energy Calculation	20
6.1.3	Direct Measurement Alternative	20
6.2	Related to CodeCarbon's GPU Energy Calculation	21
6.3	Environmental Considerations	21
6.4	Litterature	21
7	Feedback	21

8 Funding

21

1 Preface

This document summarizes the benchmarking, training configuration, and performance results obtained while generating the **Vulnerability Severity Classification** model across different GPU architectures.

The **VLAI Vulnerability Severity Classification** model developed at CIRCL is regularly updated and shared on Hugging Face. It has been presented in:

Bonhomme, C., & Dulaunoy, A. (2025). *VLAI: A RoBERTa-Based Model for Automated Vulnerability Severity Classification* (Version 1.4.0) [Computer software].
<https://doi.org/10.48550/arXiv.2507.03607>

All materials used to produce this technical report—including Matplotlib scripts, datasets, and other resources—are available in the Git repository:

<https://github.com/vulnerability-lookup/gpu-vuln-bench>

2 Environments used for benchmarking

2.1 GPU Architectures

The performance benchmarks were conducted on the GPU-accelerated systems described in the table 1. Each environment varies in CPU architecture, GPU type, and memory capacity, enabling us to evaluate model training efficiency across different hardware configurations.

Table 1: GPU-accelerated systems used for benchmarking.

Environment	CPU Cores	GPU(s)	RAM
A	64 (AMD EPYC 9124 16-Core Processor)	2 × NVIDIA L40S	251.5 GB
B	224 (Intel Xeon Platinum 8480+)	2 × NVIDIA H100 NVL	2,014 GB
C	224 (Intel Xeon Platinum 8480+)	4 × NVIDIA L40S	2,014 GB

Each environment was used to execute a series of experiments designed to measure the throughput, memory utilization, and training time of the VLAI Vulnerability Severity Classification model. The following sections provide a detailed summary and analysis of these experiments.

2.2 Framework Versions

The environment used for training:

- **Python:** 3.12.3
- **Transformers:** 4.57.1
- **PyTorch:** 2.9.1+cu128
- **Datasets:** 4.4.1
- **Tokenizers:** 0.22.1

3 Dataset

The dataset used for training and evaluation is available on Hugging Face at the commit 2135755d8f42902de065d1ca30d800820b1e5cf1.

<https://huggingface.co/datasets/CIRCL/vulnerability-scores>

This is the updated version of the dataset referenced in [arXiv.2507.03607](#).

Dataset statistics:

- Number of rows: 642,080
 - Train split: 577,872
 - Test split: 64,208
 - ref: commit 2135755d8f42902de065d1ca30d800820b1e5cf1
- Downloaded size: 159 MB
- Auto-converted Parquet size: 159 MB

The test split accounts for **10%** of the dataset and can be configured in VulnTrain.

VulnTrain is developed as part of the AIPITCH project and is integrated with Vulnerability-Lookup via ML-Gateway—a FastAPI-based local server that > loads one or more pre-trained NLP models at startup and exposes them through a clean, RESTful API for inference.

For more details, see: <https://github.com/vulnerability-lookup/ML-Gateway>.

This dataset is periodically updated with data collected with Vulnerability-Lookup.

4 Model training

4.1 Resulting models

The main model is available on Hugging Face:

<https://huggingface.co/CIRCL/vulnerability-severity-classification-roberta-base>

It is a fine-tuned version of RoBERTa-base trained on the CIRCL/vulnerability-scores dataset.

Intermediate models are also available on Hugging Face and are versioned for reproducibility:

- <https://huggingface.co/CIRCL/vulnerability-severity-classification-roberta-base-expA>
- <https://huggingface.co/CIRCL/vulnerability-severity-classification-roberta-base-expB>
- <https://huggingface.co/CIRCL/vulnerability-severity-classification-roberta-base-expC>

The code of the trainer is available in the VulnTrain project.

4.2 Training Hyperparameters

The following hyperparameters were used during training:

- **Learning rate:** $3e-05$
- **Per device Batch Size:** 8
- **Seed:** 42
- **Optimizer:** ADAMW_TORCH_FUSED
- **Scheduler:** linear
- **Epochs:** 5

For a RoBERTa model, the default batch size per device we chose is **8**.

RoBERTa-base is a medium-sized Transformer model (approx. 125 million parameters). A batch size of 8 per device is a standard, conservative choice that is unlikely to cause Out-of-Memory (OOM) errors on most modern GPUs (like NVIDIA V100, A100, or even modern consumer cards like the RTX 3080/4080) for typical sequence lengths (e.g., 128 or 256 tokens).

3×10^{-5} is a standard and safe learning rate for fine-tuning RoBERTa, with the optimizer using its default settings.

4.2.1 A quick note on epochs and batches

A **batch** is a subset of the training data processed together in **one forward and backward pass**, producing gradients that update the model weights.

The batch size is the number of samples in that batch.

An **epoch** is one full pass over the entire training dataset.

Since the dataset is divided into batches, an epoch consists of multiple steps, where each step processes one batch and updates the model weights.

The **effective batch size** (batch size \times number of GPUs) influences training dynamics:

- Larger effective batches produce more stable gradients, require fewer optimization steps per epoch, and often converge faster.
- Smaller batches introduce noise in the gradients, which can help escape poor local minima and improve generalization, but each epoch takes longer.
- The impact on generalization also depends on using an appropriate learning rate.

RoBERTa often benefits from slightly larger batches. For example, using a batch of 32 samples per step can reduce gradient noise and stabilize learning, leading to quicker convergence.

Figure 1: Number of GPUs / Batch Size - Illustration 1

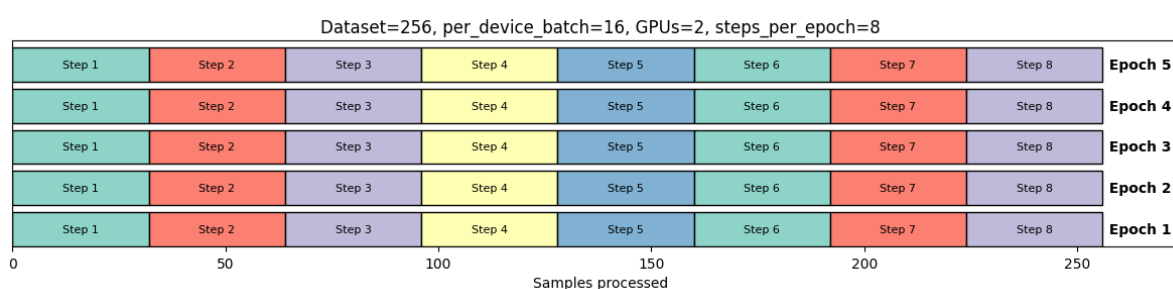
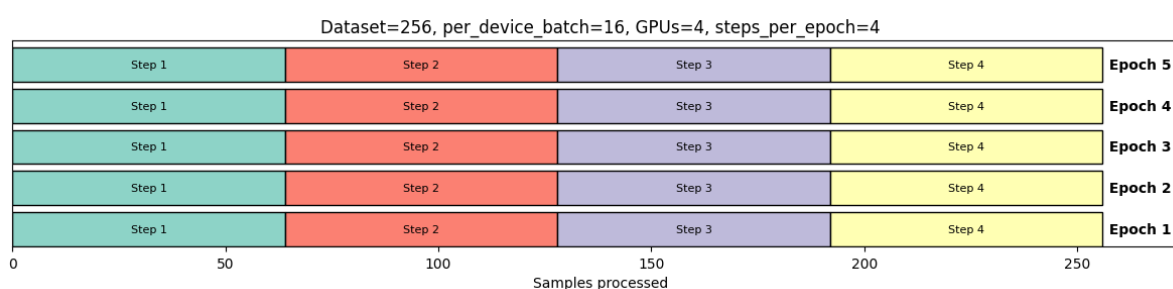


Figure 2: Number of GPUs / Batch Size - Illustration 2



Each colored rectangle represents a single training step, corresponding to one processed batch. An epoch ends once steps_per_epoch steps have been completed.

In our case, the training split contains 577,872 samples. The visualizations use a simplified view to illustrate the concepts more clearly for learning purposes. They illustrate how batch size, number of GPUs, and dataset size affect the number of training steps per epoch.

4.3 Training results

Environment	Final Loss	Final Accuracy	Epochs to Converge	Batch Size	Steps per Epoch
A	0.2537	0.8232	5	16	29470
B	0.2801	0.8230	5	16	29470
C	0.3793	0.8173	5	32	14735

Table 2: Final training results for the different environments.

Results in terms of **loss** and **accuracy** are very similar, regardless of the system used. Each experiment produced slightly different rankings, but the differences are minimal.

The samples per epoch is the same in each environments: 577872. Which corresponds to 10 per cent of the dataset

4.3.1 Environment A

Theoretically, `samples_per_epoch` should match the number of samples in the training split (577,872), but our trainer filters out entries with missing or unknown severity labels. As previously explained an **epoch** is one full pass over the entire training dataset.

Training Loss	Epoch	Step	Validation Loss	Accuracy	steps_per_epoch	samples_per_epoch
0.4999	1.0	29470	0.6657	0.7290	29470.0	471520.0
0.5279	2.0	58940	0.5911	0.7685	29470.0	471520.0
0.4775	3.0	88410	0.5392	0.7961	29470.0	471520.0
0.3753	4.0	117880	0.5125	0.8122	29470.0	471520.0
0.2537	5.0	147350	0.5169	0.8232	29470.0	471520.0

Table 3: Training results for an experiment with environment A

4.3.2 Environment B

Training Loss	Epoch	Step	Validation Loss	Accuracy	steps_per_epoch	samples_per_epoch
0.5379	1.0	29470	0.6573	0.7358	29470.0	471520.0
0.5714	2.0	58940	0.5810	0.7710	29470.0	471520.0
0.4636	3.0	88410	0.5412	0.7918	29470.0	471520.0
0.4738	4.0	117880	0.5098	0.8131	29470.0	471520.0
0.2801	5.0	147350	0.5175	0.8230	29470.0	471520.0

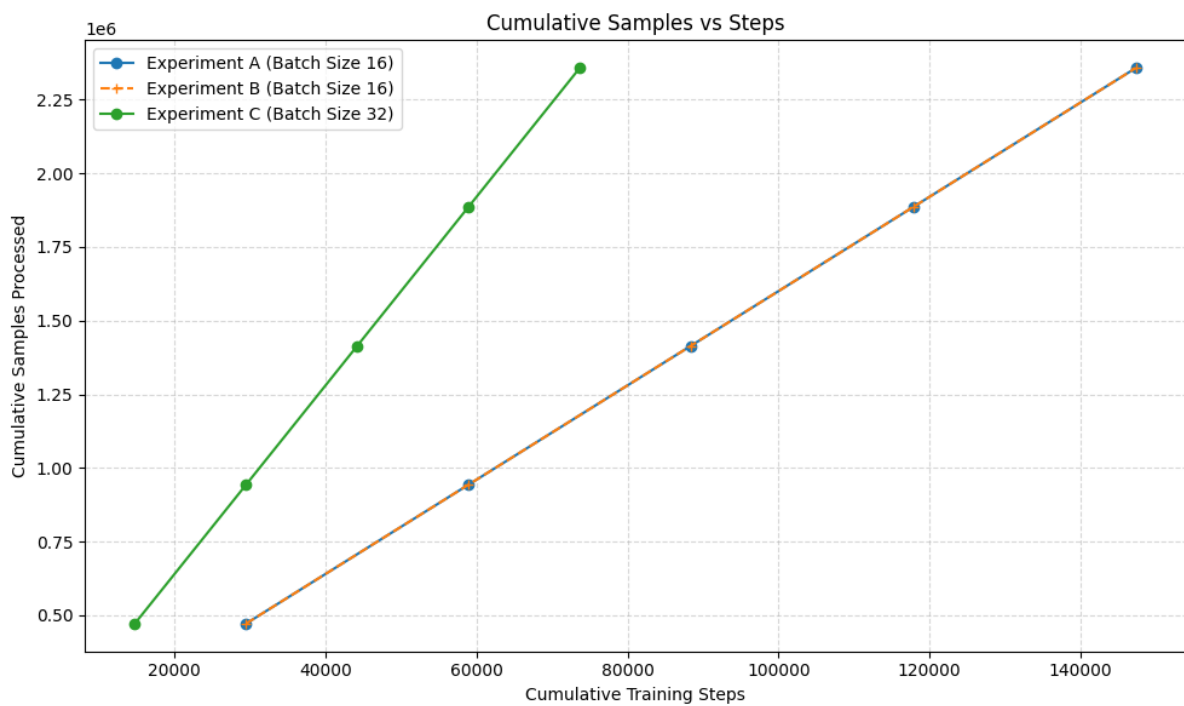
Table 4: Training results for an experiment with environment B**4.3.3 Environment C**

Training Loss	Epoch	Step	Validation Loss	Accuracy	steps_per_epoch	samples_per_epoch
0.6270	1.0	14735	0.6594	0.7298	14735.0	471520.0
0.5675	2.0	29470	0.5780	0.7693	14735.0	471520.0
0.4690	3.0	44205	0.5363	0.7930	14735.0	471520.0
0.4373	4.0	58940	0.5069	0.8107	14735.0	471520.0
0.3793	5.0	73675	0.5071	0.8173	14735.0	471520.0

Table 5: Training results for an experiment with environment C

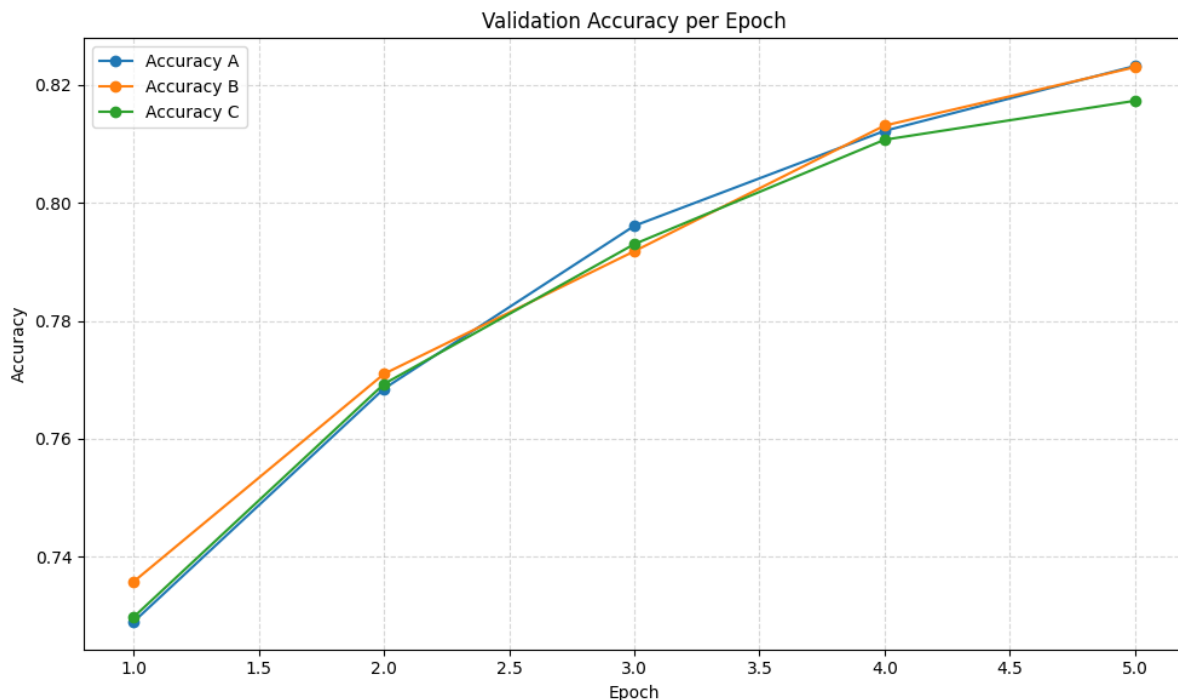
Note that $147350/2 = 73675$.

4.3.4 Comparisons



A common rule of thumb is the **linear scaling rule**: when the effective batch size is doubled, the learning rate is also doubled.

This behavior is confirmed in all of our experiments.



The chart shows the validation accuracy per epoch for the various experiments with the environments A, B, and C.

All experiments exhibit very similar accuracy trends.

Experiments in environment C reaches higher accuracy more quickly in the early epochs, reflecting faster convergence per epoch due to a larger effective batch size (more GPUs \times batch per device).

By the final epoch, all experiments achieve comparable accuracy (~ 0.82), indicating consistent model performance across the different setups.

4.4 Key Observations

- **More GPUs \rightarrow larger effective batch \rightarrow fewer steps per epoch**

- Example:

- * 4 GPUs \times 256 samples \rightarrow 1024 samples/step \rightarrow fewer steps to process the full dataset
 - * 2 GPUs \times 256 samples \rightarrow 612 samples/step \rightarrow more steps to process the same dataset

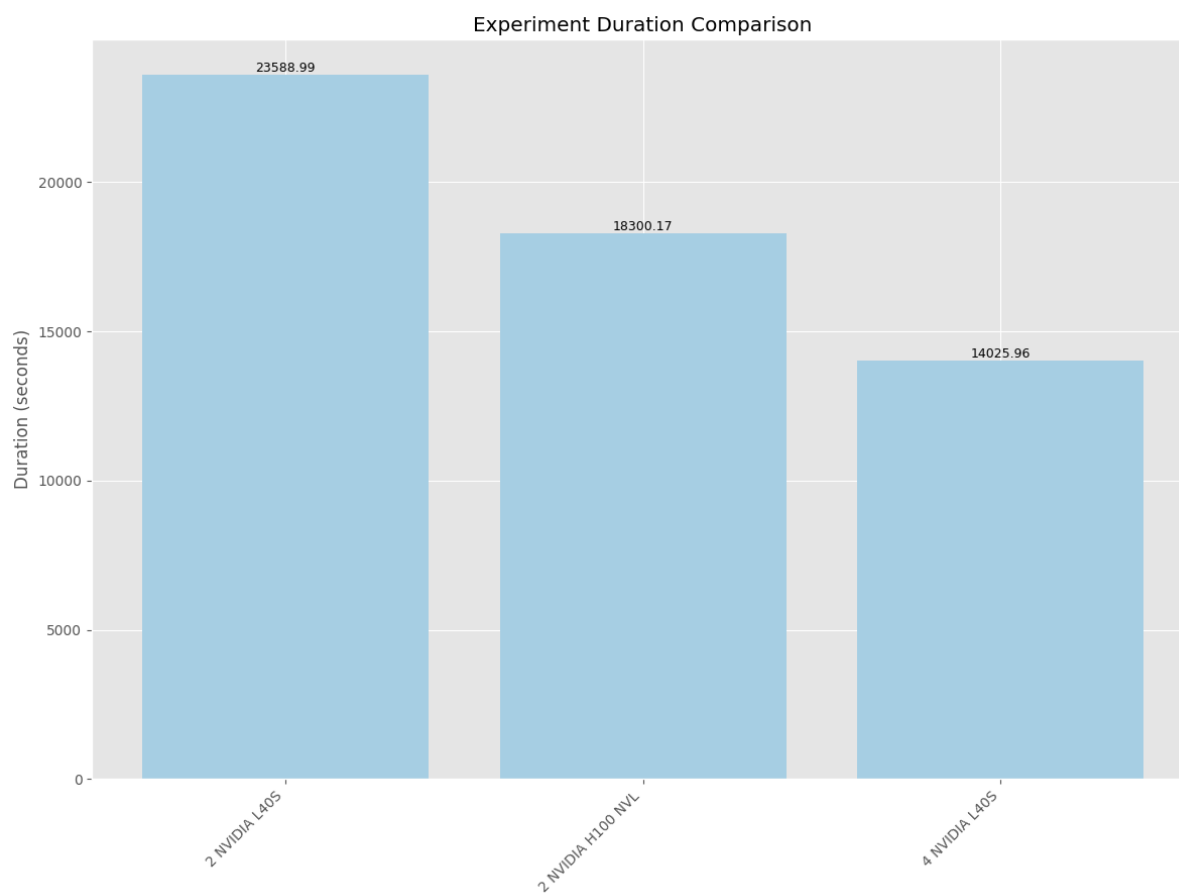
- **Larger batch size per device \rightarrow fewer steps per epoch**, but each step processes more data.
- **Epoch duration** is proportional to number of steps \times time per step, so increasing GPUs or batch size reduces total training time per epoch.

Figure 1 and 2 make it easier to understand why Exp C (4 GPUs, batch size 8 per device \rightarrow effective batch 32) completes fewer steps per epoch and thus runs faster per epoch than Exp A/B (2 GPUs, effective

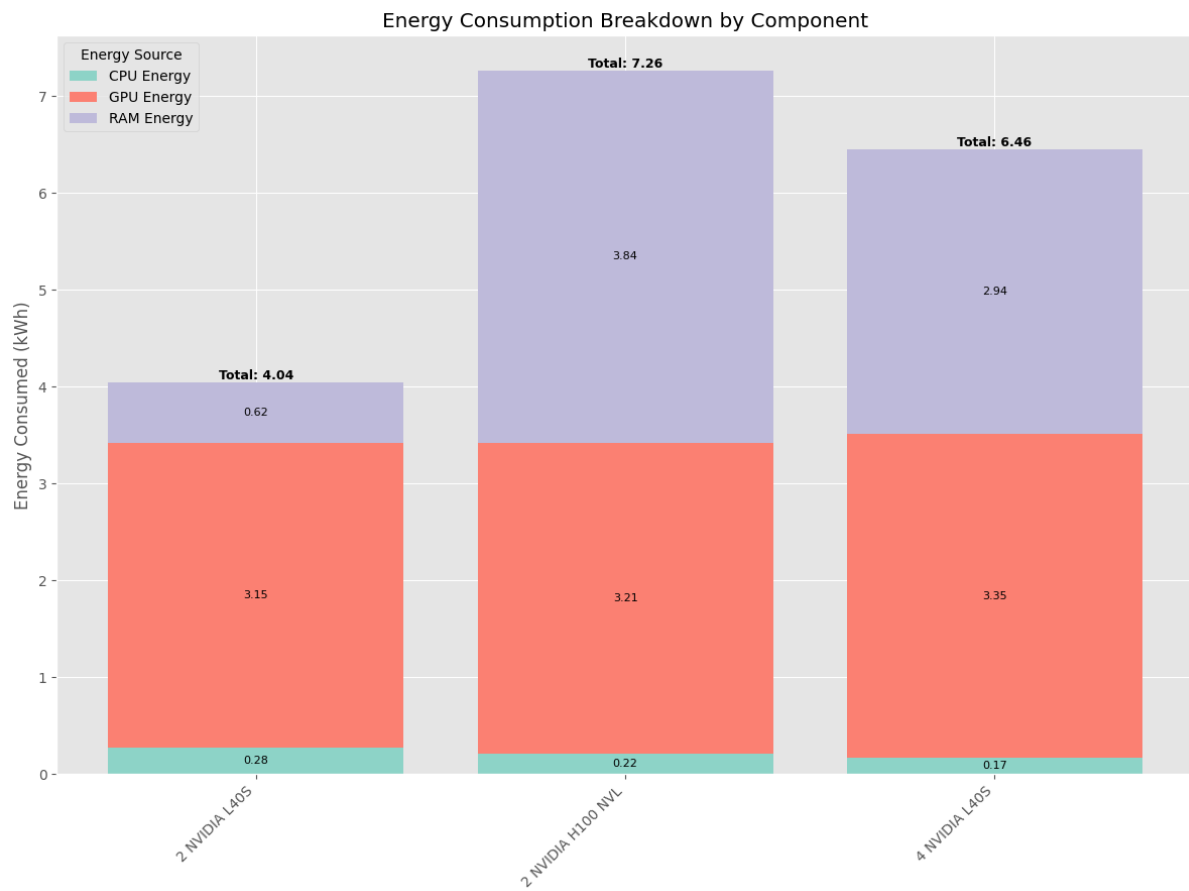
batch 16), even though the dataset and model are identical.

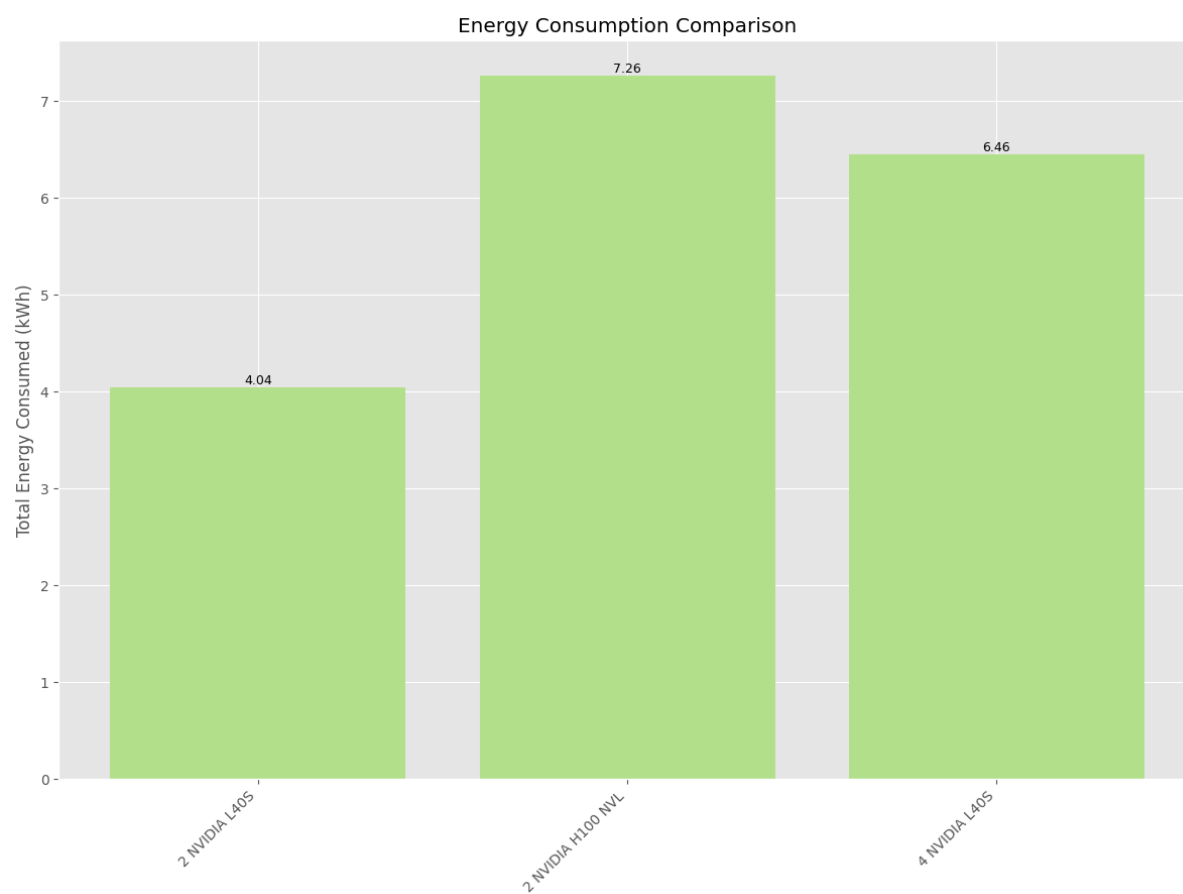
5 Benchmark Comparisons

5.1 Duration

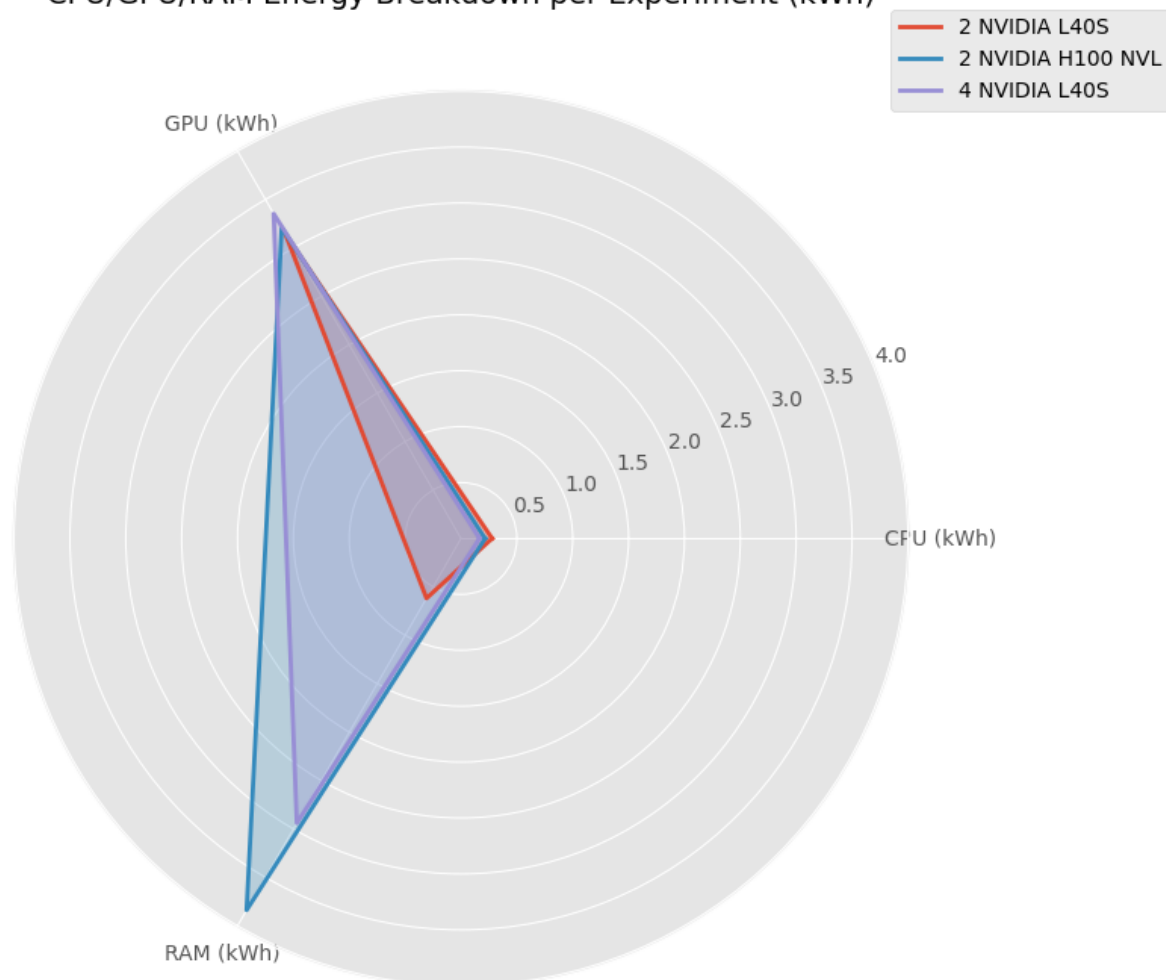


5.2 Energy

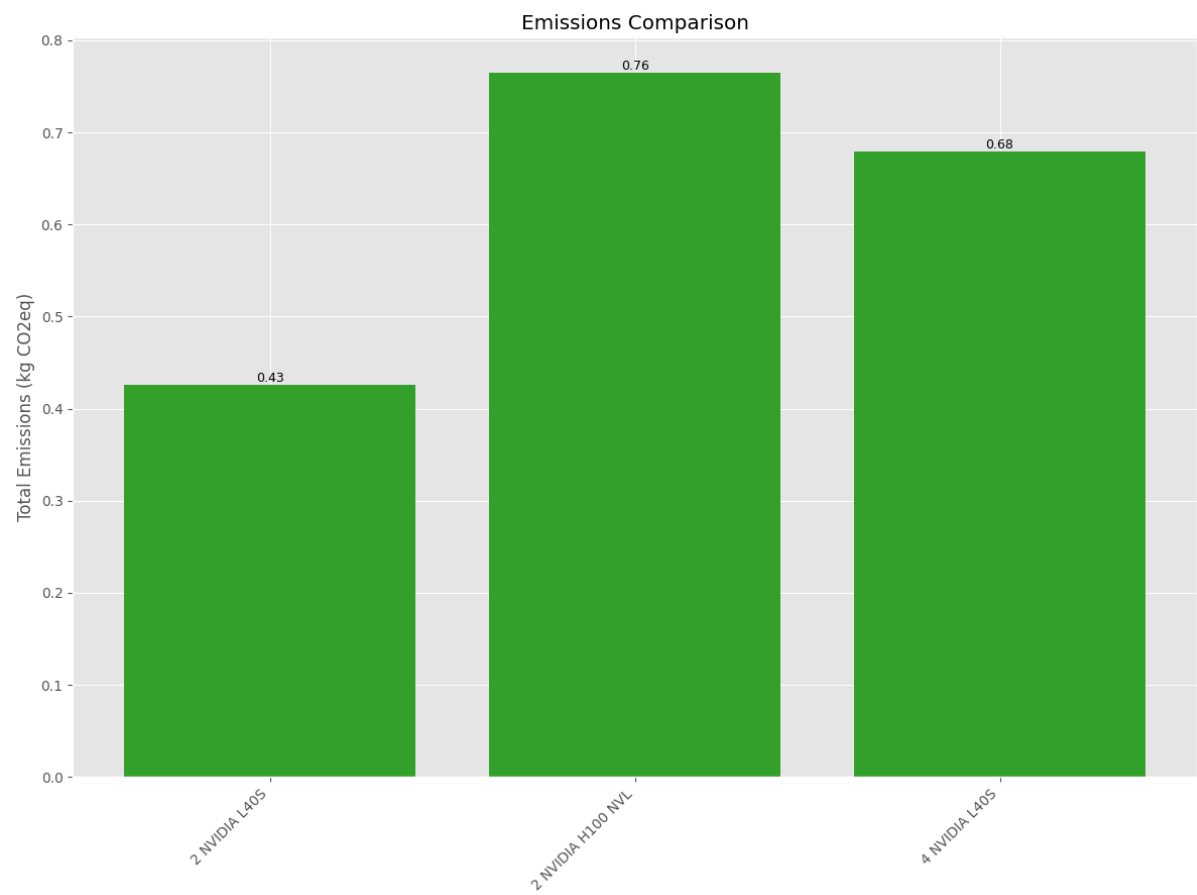




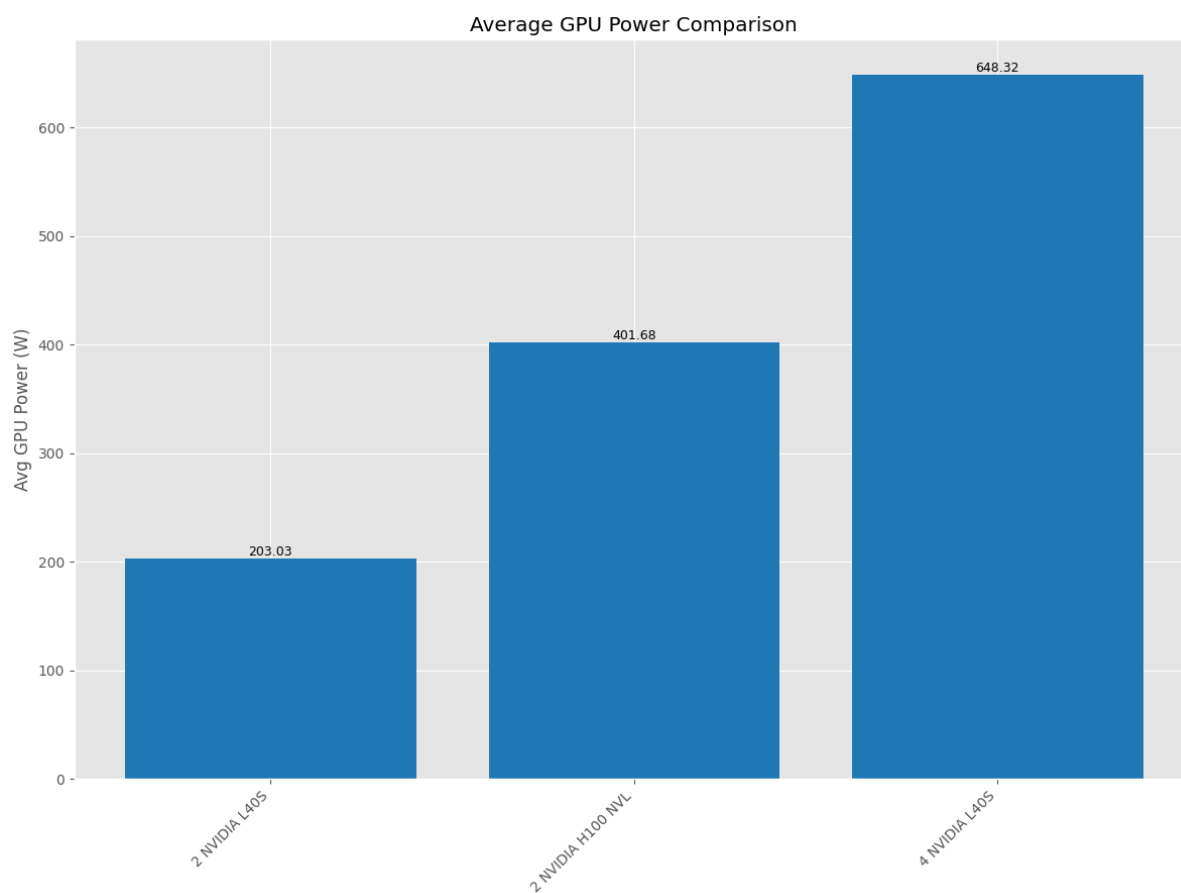
CPU/GPU/RAM Energy Breakdown per Experiment (kWh)



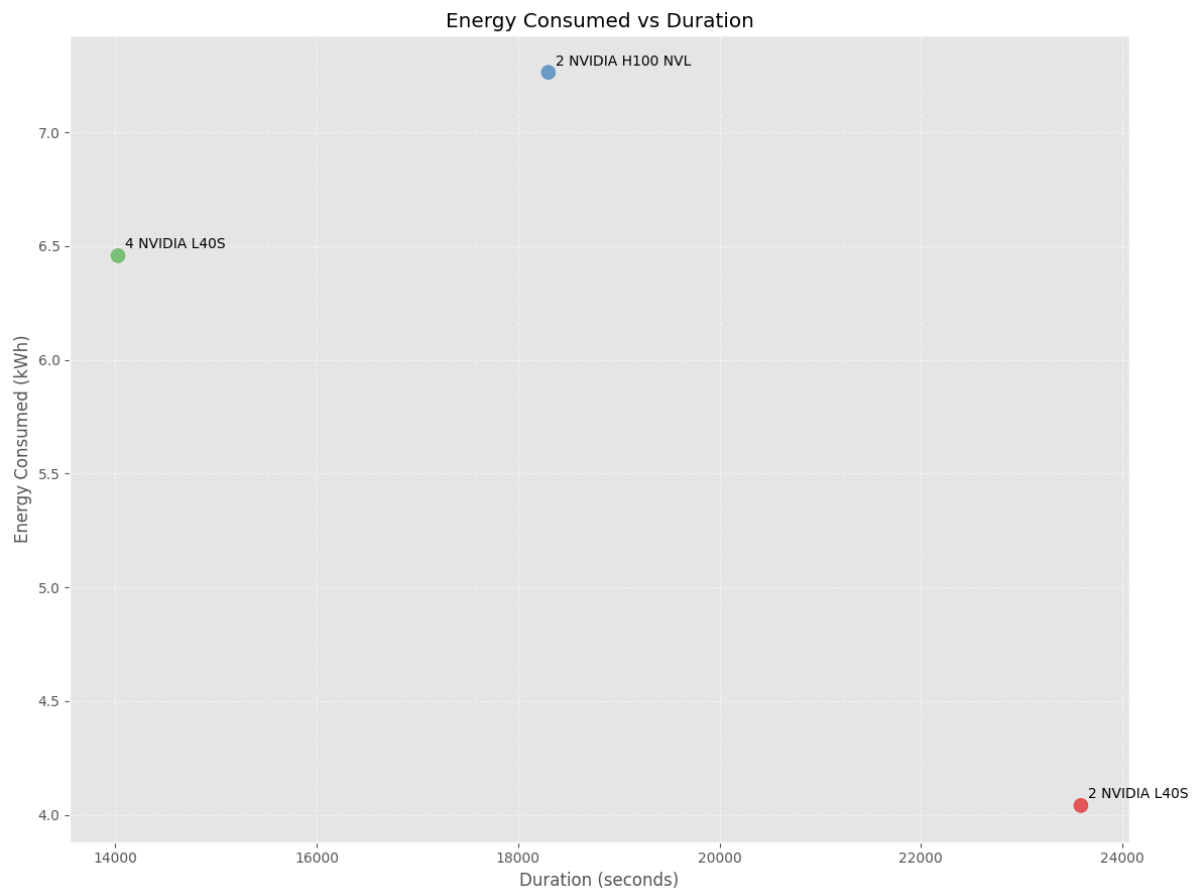
5.3 Emissions



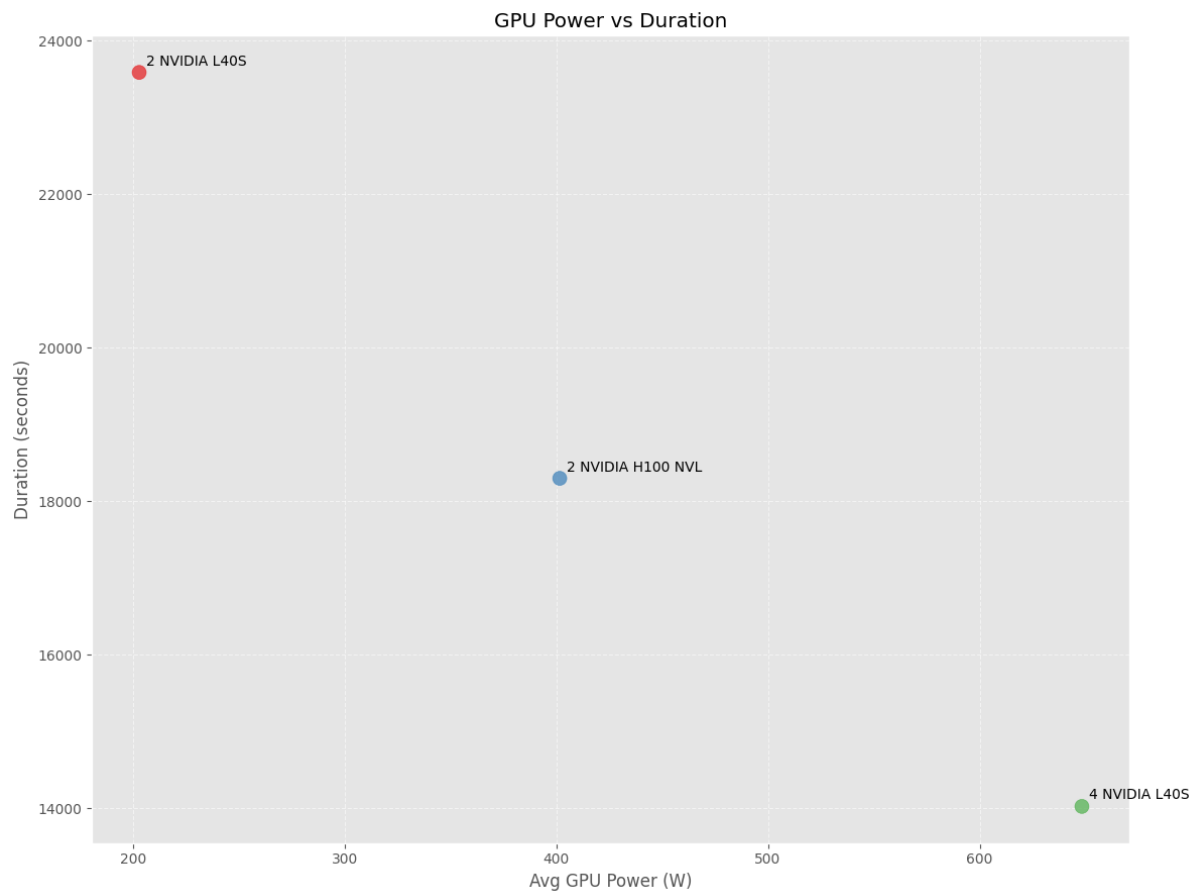
5.4 GPU Power



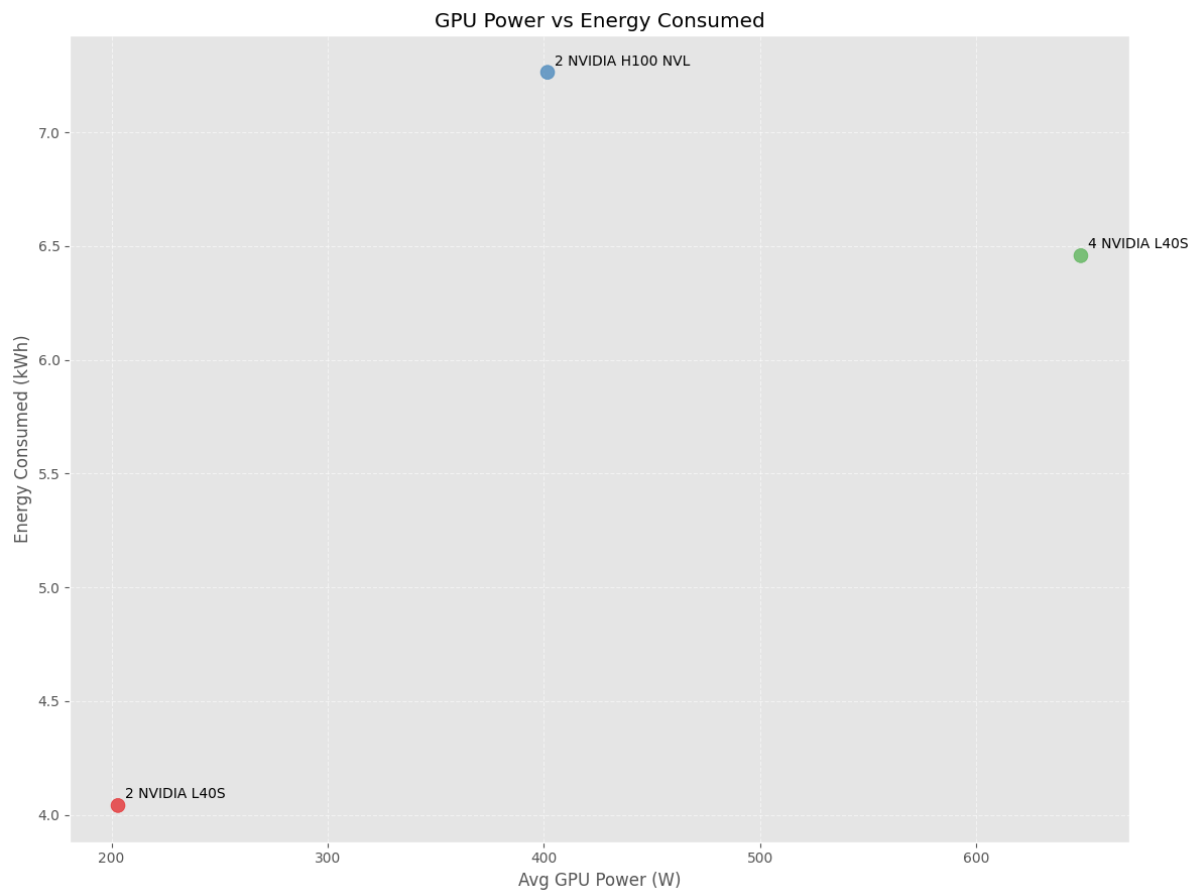
5.5 Energy vs. Duration



5.6 GPU Power vs. Duration



5.7 GPU Power vs. Energy



6 Resources

6.1 Related to CodeCarbon's RAM Energy Calculation

CodeCarbon primarily calculates the energy used by **RAM** through a **power consumption model** based on estimations, rather than direct hardware measurement, unless specific system features are available.

The power estimation for a “large server” is approximately 40W (using 8x128GB DIMMs with high efficiency scaling).

Reference: <https://mlco2.github.io/codecarbon/methodology.html#ram>

6.1.1 Estimation Methodology

The default method relies on a fixed power consumption value per installed RAM module (DIMM):

1. **Fixed Power per DIMM:** A standardized, average power consumption value is assigned to each RAM module.
 - For **x86 Systems** (most standard laptops/desktops), this is typically set at **5 Watts** per DIMM.
 - For **ARM Systems** (e.g., Raspberry Pi), a lower base power, like **1.5W** per DIMM, or a constant of **3W**, is used.
2. **Counting RAM Modules:** CodeCarbon attempts to determine the **number of installed RAM modules (DIMMs)** on the system by querying the operating system.
3. **Total Power Calculation:** The estimated total RAM power is calculated by multiplying these two values:

$$\text{RAM Power (Watts)} = \text{Fixed Power per DIMM} \times \text{Number of RAM Slots Used}$$

4. **Scaling (for Servers):** For systems with many DIMMs (e.g., servers with 8+ slots), a scaling factor is applied to reduce the power assigned to each additional DIMM, acknowledging that power consumption doesn't increase strictly linearly in large configurations.

6.1.2 Energy Calculation

Once the estimated **RAM Power** (in Watts) is determined, the **Energy Consumed** (in kilowatt-hours, or kWh) is calculated based on the duration of the code execution:

$$\text{Energy (kWh)} = \frac{\text{Power (Watts)} \times \text{Time (hours)}}{1000}$$

6.1.3 Direct Measurement Alternative

On Linux systems, CodeCarbon offers a more accurate method with the **Intel Running Average Power Limit (RAPL)** interface.

- If the `rapl_include_dram` parameter is set to `True`, CodeCarbon will attempt to use the **direct power measurement** for the DRAM (memory subsystem) provided by RAPL, overriding the fixed power estimation model. This method offers the most precise consumption data when available.

Reference: <https://mlco2.github.io/codecarbon/parameters.html>

6.2 Related to CodeCarbon's GPU Energy Calculation

The energy consumption is tracked using `nvidia-ml-pylibrary`.

Reference: <https://mlco2.github.io/codecarbon/methodology.html#gpu>

6.3 Environmental Considerations

Our server room is hosted in LuxConnect's data centers, which are powered entirely by renewable energy (<https://www.luxconnect.lu/infrastructure>).

6.4 Litterature

- "Natural Language Processing with Transformers"
<https://www.librarything.com/work/27807959/281493045>
- "How AI Works: From Sorcery to Science"
<https://www.librarything.com/work/31127745/287620374>

7 Feedback

Feel free to share your feedback at info@circl.lu or publicly:

<https://github.com/vulnerability-lookup/gpu-vuln-bench/issues>

8 Funding



**Co-funded by
the European Union**

AIPITCH aims to create advanced artificial intelligence-based tools supporting key operational services in cyber defense. These include technologies for early threat detection, automatic malware classification, and improvement of analytical processes through the integration of Large Language Models (LLM). The project has the potential to set new standards in the cybersecurity industry.

The project leader is NASK National Research Institute. The international consortium includes:

- CIRCL (Computer Incident Response Center Luxembourg), Luxembourg
- The Shadowserver Foundation, Netherlands
- NCBJ (National Centre for Nuclear Research), Poland
- ABI LAB (Centre of Research and Innovation for Banks), Italy

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the European Cybersecurity Competence Centre can be held responsible for them.