Benchmarking NVIDIA A100 vs Intel Gaudi for UNet Semantic Segmentation on CARLA

Bhavya Minesh Shah (ASU ID: 1233216428)

December 7th, 2025

Abstract

This report benchmarks a UNet semantic segmentation model on the CARLA urban driving dataset across three accelerator configurations: an NVIDIA A100 GPU (CUDA), Intel Gaudi (HPU) in eager mode, and Intel Gaudi in lazy mode. All runs share the same data pipeline, model architecture, and training hyperparameters to isolate hardware effects. We compare segmentation quality (pixel accuracy, mean IoU, precision, recall, F1), training and inference throughput, latency, and memory usage using metrics collected directly from instrumented training scripts and result JSON files. The results show that all accelerators achieve nearly identical accuracy, while Gaudi—especially in lazy mode—delivers significantly higher per-batch throughput and inference throughput, at the cost of slightly higher end-to-end training time and substantially higher device memory usage. We further characterize device memory capacity, peak usage, and available headroom on each accelerator.

1 Introduction

Modern AI workloads increasingly rely on specialized accelerators. GPUs such as the NVIDIA A100 are widely used, while AI-focused accelerators like Intel Gaudi HPUs are designed to provide higher throughput and better cost/performance for deep learning. In this work we quantify how these platforms behave for a realistic computer vision workload: multi-class semantic segmentation for autonomous driving.

We implement a UNet architecture and train it on the CARLA Vision dataset with identical software stacks and hyperparameters across:

- NVIDIA A100-SXM4-80GB with CUDA and PyTorch.
- Intel Gaudi HPU in *eager mode*.
- Intel Gaudi HPU in *lazy mode* (graph-based execution).

Our goals are:

1. Quantify segmentation quality differences across accelerators.
2. Benchmark training and inference throughput and latency.
3. Characterize memory usage, capacity, and available headroom.
4. Summarize practical recommendations for choosing an accelerator.

2 Experimental Setup

2.1 Task and Dataset

We train a semantic segmentation model on the CARLA urban driving dataset (Kaggle: *carla-vd-dataset*), using 20 semantic classes (road, vehicles, buildings, vegetation, etc.). The dataset preparation script:

- Downloads the dataset from Kaggle if not present.
- Generates random crops of size 384×512 from each original image, with `crops_per_img = 10`.
- Writes the cropped images and masks to `cropped_384x512/images` and `cropped_384x512/masks`.
- Splits images into train/validation/test with ratios 70%/15%/15% via `train_test_split`, and saves `dataset_splits.json` for consistent usage across devices.

All three training scripts load this shared `dataset_splits.json`, resulting in:

$$\text{Train samples} = 10178, \quad \text{Val samples} = 2181, \quad \text{Test samples} = 2181.$$

2.2 Model Architecture

All experiments use the same UNet implementation in PyTorch:

- Input: 3-channel RGB images, 384×512.
- Output: 20-channel per-pixel logits (one channel per class).
- Encoder: 4 down-sampling stages, each composed of two 3×3 convolutions with batch normalization and ReLU, followed by 2×2 max pooling.
- Decoder: 4 up-sampling stages with transposed convolutions, skip connections from the encoder's corresponding levels, and the same double-convolution blocks.
- Final layer: 1×1 convolution projecting to 20 classes.

The compiled model has:

$$\text{Total parameters} = 31{,}044{,}756, \quad \text{Trainable parameters} = 31{,}044{,}756,$$

with model memory $\approx 0.12\,\text{GB}$ on all devices.

2.3 Training Configuration and Hyperparameters

All three training scripts (`unet_cuda.py`, `unet_gaudi_eager.py`, `unet_gaudi_lazy.py`) share a nearly identical configuration:

- Batch size: 32.
- Epochs: 20.
- Input resolution: 384×512.
- Optimizer: Adam with learning rate 1×10^{-4}.
- Scheduler: `ReduceLROnPlateau` on validation loss (patience = 3, factor = 0.5).
- Loss function: multi-class cross-entropy on one-hot class masks.
- Workers per DataLoader: 8.
- Random seed: 42 for Python, NumPy, and device-specific RNGs.

Table 1 summarizes the core configuration.

Table 1: Shared training configuration and task setup.

Parameter	Value
Dataset	CARLA urban driving (20 classes)
Input resolution	384×512 random crops
Crops per original image	10
Train / Val / Test samples	10178 / 2181 / 2181
Batch size	32
Number of epochs	20
Optimizer	Adam
Initial learning rate	1×10^{-4}
Scheduler	ReduceLROnPlateau (val loss)
Loss	Cross-entropy (multi-class)
Number of workers per loader	8
Random seed	42
Model parameters	31,044,756

2.4 Hardware and Software Configuration

The three runs differ only in the accelerator and corresponding runtime stack:

- **CUDA (A100)**: NVIDIA A100-SXM4-80GB GPU (80 GB HBM2e), CUDA **12.8**, PyTorch **2.8.0+cu128**. The CUDA version is reported by both `nvidia-smi` and `torch.version.cuda` in the CUDA job output.
- **Gaudi (Eager)**: Intel Gaudi HPU (96 GB HBM) in eager mode, PyTorch built as **2.7.1+hpu_1.22.0-740.git6dbe0a4**. The jobs run inside the `gaudi-pytorch-diffusion-1.22.0.740` environment, corresponding to the Habana / HL stack **1.22.0-740**.
- **Gaudi (Lazy)**: Same Gaudi hardware and software stack as above, but with lazy (graph) mode enabled via `PT_HPU_LAZY_MODE=1`.

In all cases, host-side code is identical aside from device selection and minor HPU-specific calls (e.g., Habana memory APIs and `mark_step` for lazy mode).

2.5 DataLoader Configuration and Pinned Memory

While the dataset splits, augmentations, and batch size are shared across all runs, the DataLoader configuration differs in how host memory is handled for each accelerator.

All training and validation loops move batches to the device using:

```
images = images.to(device, non_blocking=True)
masks  = masks.to(device, non_blocking=True)
```

The key difference is how DataLoaders allocate host memory:

- **CUDA (A100):** All three DataLoaders (train/val/test) use `pin_memory=True`. Each batch is allocated in page-locked (pinned) host memory, allowing asynchronous host-to-device transfers when combined with `non_blocking=True` in `.to(device)`. This is standard practice to reduce data-transfer stalls and overlap I/O with GPU computation.

- **Gaudi (Eager/Lazy):** All Gaudi DataLoaders use `pin_memory=False`. The Habana HPU runtime implements its own optimized data-transfer path; the recommended configuration is to disable PyTorch CPU pinned memory while still using `non_blocking=True` in `.to(device)` calls.

Table 2 summarizes the DataLoader settings per device.

Table 2: DataLoader configuration: pinned memory vs. device type.

Device	`pin_memory`	Notes
CUDA (A100)	`True`	Pinned host memory, async H2D with `non_blocking`
Gaudi (Lazy mode)	`False`	HPU-specific transfer path, PyTorch pinning disabled
Gaudi (Eager mode)	`False`	Same as lazy, eager execution on HPU

Because the dataset, batch size, and number of workers are the same, this difference primarily affects how efficiently each accelerator can overlap host-to-device transfers with computation. On CUDA, pinned memory helps ensure that data-transfer overhead is not the primary bottleneck. On Gaudi, the higher device-level throughput suggests that the HPU runtime effectively hides data-transfer costs despite not using PyTorch's pinned-memory mechanism.

2.6 Metrics and Logging

A shared `PerformanceMetrics` helper maintains a confusion matrix and time/memory statistics for each run. From this, each script computes:

- Per-class IoU and mean IoU.
- Mean precision, recall, and F1 score.
- Overall pixel accuracy.
- Average, minimum, and maximum batch and iteration times.
- Throughput (samples/s), peak and average device memory usage.

After training and test evaluation, each script writes a `results_*.json` file, which is consumed by a separate `compare_results.py` analysis script to print comparison tables and generate plotting code.

3 Benchmarking Methodology

Figure 1 shows the overall benchmarking pipeline.

Key aspects of the methodology:

- All accelerators see the exact same image crops and splits.
- We train for 20 epochs in all runs; the best checkpoint is chosen by validation IoU.
- Metrics are reported on the test set using the best validation IoU checkpoint for each device.
- Training-time metrics (epoch time, average batch time, throughput, memory) are averaged over all epochs.

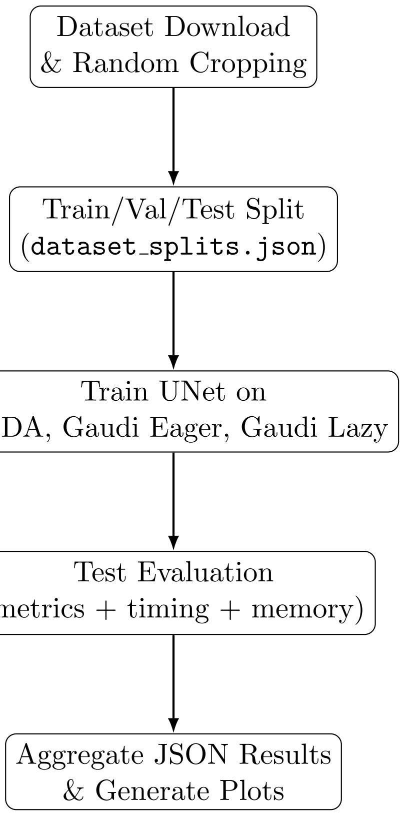

Figure 1: End-to-end benchmarking workflow shared across all accelerators.

Table 3: Test-set segmentation performance on CARLA (higher is better).

Device	**Accuracy**	**Mean IoU**	**Precision**	**Recall**	**F1**
CUDA (A100)	0.977	0.781	0.899	0.820	0.848
Gaudi (Lazy mode)	0.977	0.791	0.893	0.833	0.856
Gaudi (Eager mode)	0.977	0.730	0.797	0.770	0.782

4 Results

4.1 Segmentation Quality

Table 3 reports test-set segmentation metrics for all three hardware configurations.

Observations:

- All three devices achieve almost identical pixel accuracy ($\approx 97.7\%$).
- Mean IoU is very similar for CUDA and Gaudi lazy (0.781 vs. 0.791), with Gaudi lazy slightly ahead.
- Gaudi eager trails the other two in mean IoU and F1, but remains within a few points, indicating no drastic degradation.

4.2 Training Performance

Table 4 summarizes training-time metrics per device, including wall-clock training time, average epoch duration, and average batch latency and throughput.

Table 4: Training performance metrics (20 epochs on CARLA).

Device	**Total time**[s]	**Avg epoch**[s]	**Avg batch**[s]	
CUDA (A100)	20 182.000	849.000	0.602	53.200
Gaudi (Lazy mode)	21 867.000	910.000	0.019	1700.400
Gaudi (Eager mode)	22 388.000	941.000	0.084	378.900

Two important patterns emerge:

- **Wall-clock training time:** In this setup, the A100 achieves the fastest end-to-end training (about 2.0×10^4 seconds), with Gaudi lazy and eager about 8–11% slower in total time.
- **Per-batch throughput:** At the same time, Gaudi devices—especially in lazy mode—deliver substantially higher device-level throughput:
  - Gaudi lazy: $\approx$ 1700 samples/s ($\sim 32\times$ A100).
  - Gaudi eager: $\approx$ 379 samples/s ($\sim 7\times$ A100).

4.3 Inference / Test Performance

The test loops also collect batch latency and throughput. Table 5 summarizes the test-time metrics.

Table 5: Test-time (inference) performance on the CARLA test split.

Device	**Avg batch**[s]	
CUDA (A100)	0.191	167.600
Gaudi (Lazy mode)	0.008	4173.100
Gaudi (Eager mode)	0.063	506.200

Gaudi lazy mode clearly dominates test-time throughput:

- Gaudi lazy delivers $\sim 25\times$ higher test throughput than A100 and $\sim 8\times$ higher than Gaudi eager.
- Gaudi eager also outperforms A100 in test throughput by about $3\times$.

4.4 Memory Capacity and Usage

The result JSON files expose both model footprint and runtime memory usage on the devices. For each accelerator, we record:

- *Model size*: memory to store UNet parameters.
- *Peak allocated*: maximum memory actively allocated by the framework tensors during test evaluation.
- *Peak reserved*: maximum memory reserved by the allocator.
- *Average allocated*: mean allocated memory across test batches.
- *Device capacity*: nominal HBM capacity (80 GB for A100, $\approx$ 96 GB for Gaudi).

- *Free at peak*: capacity minus peak reserved (approximate available headroom).

Table 6 summarizes these values.

Table 6: Device memory capacity and usage during test evaluation.

Device	**Capacity**[GB]	**Peak reserved**[GB]	**Free at peak**[GB]	**Peak alloc.**[GB]	**Avg a**
CUDA (A100)	80.000	52.270	27.730	1.570	1
Gaudi (Lazy mode)	96.000	94.620	1.380	31.580	31
Gaudi (Eager mode)	96.000	94.620	1.380	19.540	18

Key takeaways:

- The UNet model itself is small ($\sim 0.12\,\text{GB}$), but the runtime footprint differs significantly.
- Gaudi lazy mode reserves almost the entire 96 GB of HBM, leaving $\approx 1.4\,\text{GB}$ free at peak; Gaudi eager exhibits similar reserved memory but with a smaller active (allocated) footprint.
- The A100 run reserves about 52 GB out of 80 GB, leaving $\approx 28\,\text{GB}$ free at peak, with only $\sim 1.6\,\text{GB}$ actually allocated for this workload.
- High reserved memory on Gaudi is consistent with graph-based execution and additional internal buffering, but it also means less absolute headroom on a single device for additional models or larger batches without re-tuning.

4.5 Training Dynamics and Learning Curves

The training histories show smooth and comparable convergence across devices:

- Training loss decreases from ≈ 1.15 to ≈ 0.06 on all devices.
- Validation loss follows a similar trajectory, converging to ≈ 0.06.
- Train and validation IoU, accuracy, and F1 curves track each other closely across accelerators, indicating stable optimization.

To better visualize the optimization process on each accelerator, Figures 2–5 show the training and validation metrics versus epoch for all three devices (CUDA on A100, Gaudi Eager, Gaudi Lazy). These figures are generated by the provided `compare_results.py` script.

5 Discussion

5.1 Accuracy and Numerical Behavior

The main takeaway is that *model quality is effectively hardware-independent* in this setup:

- All three accelerators converge to nearly identical pixel accuracy (≈ 0.977) and very similar mean IoU values.
- Gaudi lazy slightly outperforms A100 in mean IoU and F1 by a small margin, but the differences are within typical run-to-run variation.
- Gaudi eager shows slightly lower IoU/F1, which may be due to small differences in training dynamics or the scheduler's interaction with its validation loss curve; no systematic degradation was observed.

Per-class IoUs also exhibit similar patterns across devices, with some classes (e.g., rare or small objects) having near-zero IoU on all accelerators, reflecting dataset difficulty rather than hardware limitations.

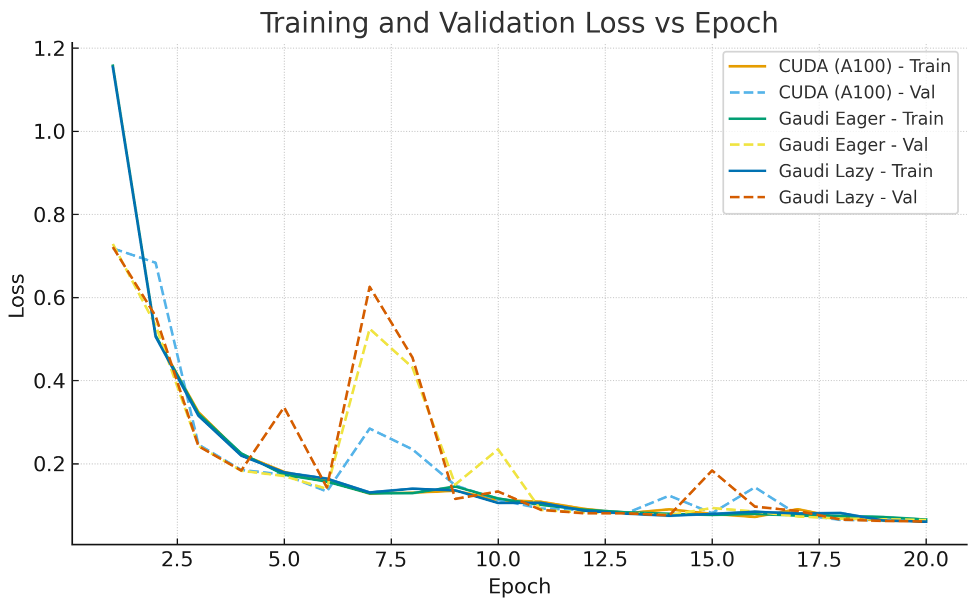

Figure 2: Training and validation loss versus epoch for CUDA (A100), Gaudi Eager, and Gaudi Lazy. Solid lines denote training loss, dashed lines denote validation loss.

5.2 Performance, Latency, and DataLoader Effects

The contrast between device-level throughput and end-to-end time is striking:

- Gaudi lazy mode achieves $\sim 32\times$ higher *training throughput* than A100 and $\sim 4.5\times$ higher than Gaudi eager for the same batch size, while Gaudi eager is itself $\sim 7\times$ faster than A100 in per-batch throughput.
- Test-time throughput shows similar trends, with Gaudi lazy up to $\sim 25\times$ faster than A100.
- Despite this, the *total training time* is slightly shorter on A100 than on Gaudi in this experiment.

This suggests that for this particular implementation:

- Host-side overheads (Python, DataLoader workers, I/O, synchronization) limit how much of Gaudi's higher device throughput translates into wall-clock speedups.
- On A100, `pin_memory=True` plus `non_blocking=True` already optimizes host-to-device transfers, reducing the chance that H2D copies dominate batch latency.
- On Gaudi, PyTorch's `pin_memory` flag is disabled but the HPU runtime still delivers very small batch times, indicating that device-side execution and its transfer path are not the bottleneck; the remaining gap in wall-clock training time is largely due to shared host-side overhead.

5.3 Memory Usage and Scalability

Gaudi lazy mode consumes the most device memory, followed by Gaudi eager and then A100. This has several implications:

- For small models like this UNet, high reserved memory on Gaudi is acceptable, but for larger models it may limit batch size or require model or pipeline parallelism.

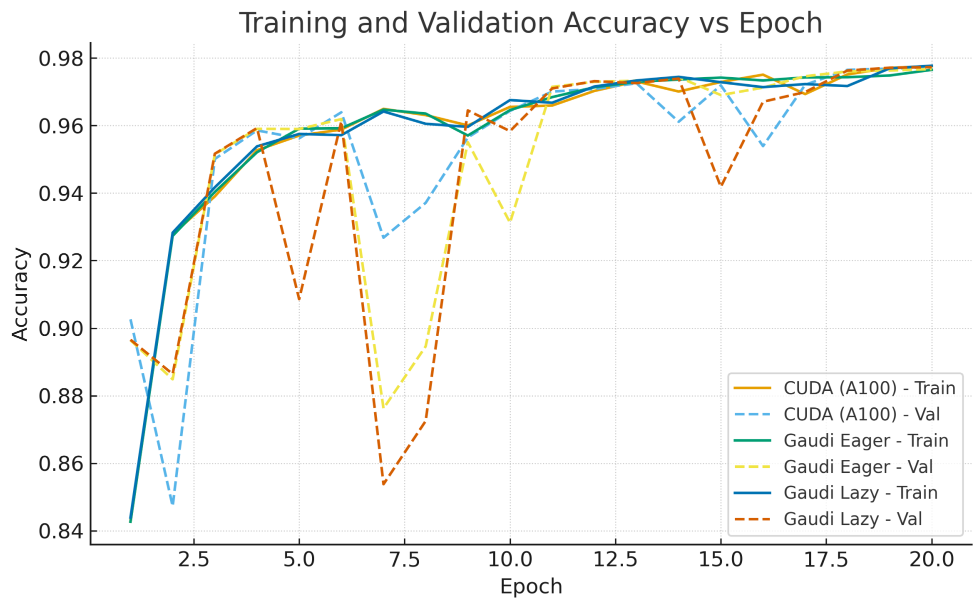

Figure 3: Training and validation accuracy versus epoch for all accelerators. Solid lines denote training accuracy, dashed lines denote validation accuracy.

- The large *capacity* on Gaudi (96 GB) makes it a strong candidate for scaling up to:
  - Higher-resolution input images.
  - Larger batch sizes.
  - Deeper or wider segmentation architectures.
- On the A100 run, only a small fraction of the 80 GB HBM is actually allocated for this workload, leaving substantial headroom for either larger models or additional concurrent jobs.

6 Conclusions and Recommendations

6.1 Summary of Findings

- **Accuracy:** All accelerators achieve similar segmentation quality on CARLA. Hardware choice does not meaningfully affect final accuracy in this setup.
- **Inference performance:** Intel Gaudi in lazy mode offers the highest test-time throughput (up to $\sim 25\times$ faster than A100), making it very attractive for deployment scenarios with heavy inference load.
- **Training performance:** While Gaudi has much higher per-batch throughput, overall training time in this particular implementation is slightly lower on A100 due to host-side bottlenecks and input-pipeline overheads.
- **Memory:** Gaudi (especially lazy mode) uses substantially more device memory, reserving almost the entire 96 GB HBM, but this can be leveraged for larger models or batches when the workload is tuned for it.

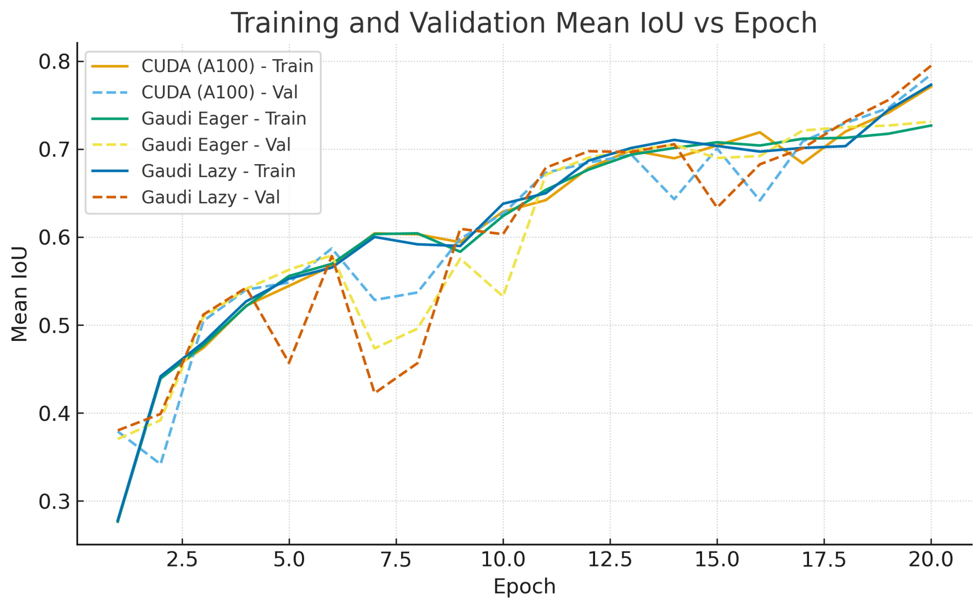

Figure 4: Training and validation mean IoU versus epoch.

6.2 Practical Recommendations

- For **research and prototyping** where wall-clock training time is critical and the model size is moderate, an A100 GPU is a safe choice, especially if existing pipelines are already optimized for CUDA and pinned-memory DataLoaders.
- For **high-throughput inference** and for **training at larger scale** (bigger models, higher resolution, larger batches), Intel Gaudi—particularly in lazy mode—is very promising, provided that the input pipeline and the rest of the stack are tuned to reduce host overheads.
- Future work should:
  - Explore larger batch sizes and mixed precision on Gaudi, where its throughput advantage is likely to translate more directly into wall-clock gains.
  - Profile and optimize the data pipeline (e.g., prefetching, caching, and distributed loading) to better saturate each accelerator.
  - Extend the benchmark to more complex segmentation backbones (e.g., UNet++ or transformer-based encoders).

7 Code

Link to GitHub repository for reproducibility: Link

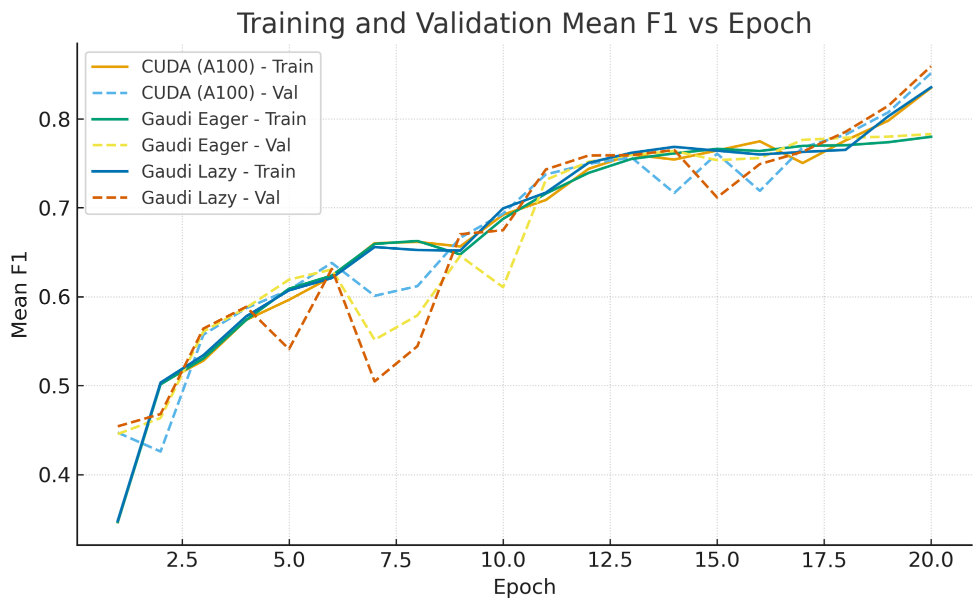

Figure 5: Training and validation mean F1 score versus epoch.