Optimizing Hindi ASR Models for NVIDIA Deployment

AI4Bharat — MLOps Engineering Assignment

Arkapravo Das
arkapravodas03@gmail.com
GitHub Repository

January 29, 2026

Contents

1 Pipeline Overview (Storyboard)

The optimization pipeline follows a clear, production-oriented progression:

1. **Profile in PyTorch** — Establish a GPU baseline and locate operator hotspots.
2. **Export to ONNX** — Deterministic graph export with shape inference.
3. **TensorRT Optimization** — Kernel fusion, Precision reduction, Engine building.
4. **Triton Packaging** — Deployment-ready inference service.

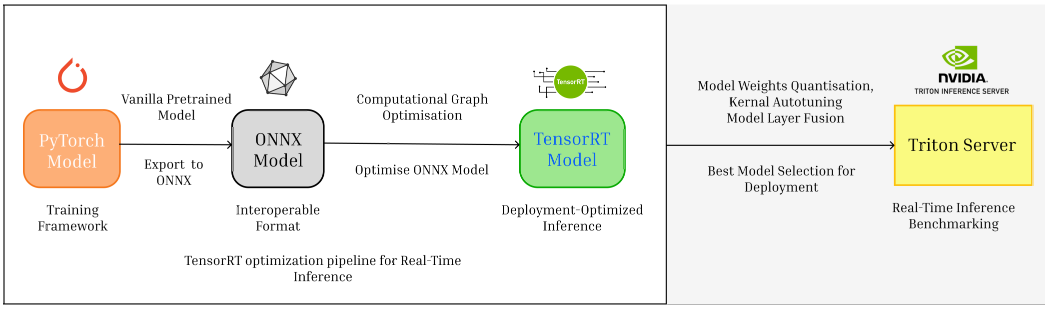


Figure 1: End-to-end optimization flow: PyTorch → ONNX → TensorRT → Triton

2 Execution Environment

- **GPU**: NVIDIA GeForce RTX 3050 (Laptop)
- **OS**: WSL2 / Ubuntu (CUDA passthrough enabled)
- **Python**: Conda environment `env-ai4bharat` (Python 3.10)
- **Model**: `ai4bharat/indicwav2vec-hindi`
- **Dataset**: Common Voice Hindi (20 samples for WER sanity check)

3 Baseline Model and PyTorch Profiling

The baseline ASR model is a Wav2Vec2-based architecture trained for Hindi speech recognition.

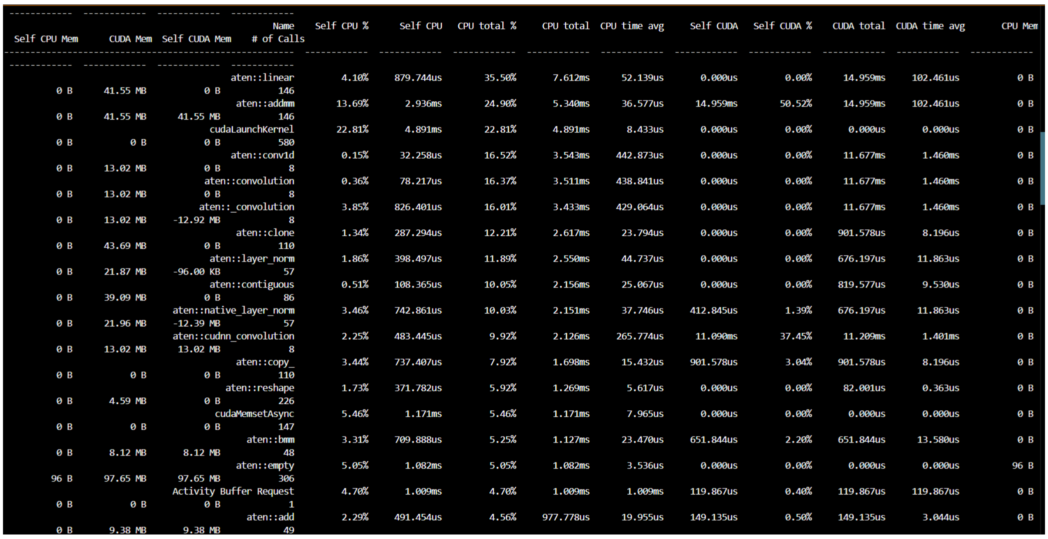
3.1 Measured PyTorch Latency

$$\text{Average PyTorch latency} = 301.88 \text{ ms}$$

3.2 Profiling Observations

PyTorch profiler revealed:

- Convolution and linear layers dominate GPU time.
- High kernel launch overhead due to unfused ops.
- Significant memory traffic during attention blocks.



Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	CPU Mem	Self CPU Mem	CUDA Mem	Self CUDA Mem	# of Calls
aten::linear	4.10%	879.744us	35.50%	7.612ms	52.139us	0.000us	0.00%	14.959ms	102.461us	0 B	0 B	41.55 MB	0 B	146
aten::addmm	13.69%	2.936ms	24.90%	5.340ms	36.577us	14.959ms	50.52%	14.959ms	102.461us	0 B	0 B	41.55 MB	41.55 MB	146
cudaLaunchKernel	22.81%	4.891ms	22.81%	4.891ms	8.433us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	580
aten::conv1d	0.15%	32.258us	16.52%	3.543ms	442.873us	0.000us	0.00%	11.677ms	1.460ms	0 B	0 B	13.02 MB	0 B	8
aten::convolution	0.36%	78.217us	16.37%	3.511ms	438.841us	0.000us	0.00%	11.677ms	1.460ms	0 B	0 B	13.02 MB	0 B	8
aten::_convolution	3.85%	826.401us	16.01%	3.433ms	429.064us	0.000us	0.00%	11.677ms	1.460ms	0 B	0 B	13.02 MB	-12.92 MB	8
aten::clone	1.34%	287.294us	12.21%	2.617ms	23.794us	0.000us	0.00%	901.578us	8.196us	0 B	0 B	43.69 MB	0 B	110
aten::layer_norm	1.86%	398.497us	11.89%	2.550ms	44.737us	0.000us	0.00%	676.197us	11.863us	0 B	0 B	21.87 MB	-96.00 KB	57
aten::contiguous	0.51%	108.365us	10.05%	2.156ms	25.067us	0.000us	0.00%	819.577us	9.530us	0 B	0 B	39.09 MB	0 B	86
aten::native_layer_norm	3.46%	742.861us	10.03%	2.151ms	37.746us	412.845us	1.39%	676.197us	11.863us	0 B	0 B	21.96 MB	-12.39 MB	57
aten::cudnn_convolution	2.25%	483.445us	9.92%	2.126ms	265.774us	11.090ms	37.45%	11.209ms	1.401ms	0 B	0 B	13.02 MB	13.02 MB	8
aten::copy_	3.44%	737.407us	7.92%	1.698ms	15.432us	901.578us	3.04%	901.578us	8.196us	0 B	0 B	0 B	0 B	110
aten::reshape	1.73%	371.782us	5.92%	1.269ms	5.617us	0.000us	0.00%	82.001us	0.363us	0 B	0 B	4.59 MB	0 B	226
cudaMemsetAsync	5.46%	1.171ms	5.46%	1.171ms	7.965us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	147
aten::bmm	3.31%	709.888us	5.25%	1.127ms	23.470us	651.844us	2.20%	651.844us	13.580us	0 B	0 B	8.12 MB	8.12 MB	48
aten::empty	5.05%	1.082ms	5.05%	1.082ms	3.536us	0.000us	0.00%	0.000us	0.000us	96 B	96 B	97.65 MB	97.65 MB	306
Activity Buffer Request	4.70%	1.009ms	4.70%	1.009ms	1.009ms	119.867us	0.40%	119.867us	119.867us	0 B	0 B	0 B	0 B	1
aten::add	2.29%	491.454us	4.56%	977.778us	19.955us	149.135us	0.50%	149.135us	3.044us	0 B	0 B	9.38 MB	9.38 MB	49

Figure 2: Operator-level GPU time breakdown from PyTorch profiler

4 ONNX Export and Runtime Behavior

4.1 Export Strategy

The model was exported with:

- Dynamic axes for batch and time
- Constant folding
- Shape inference and validation

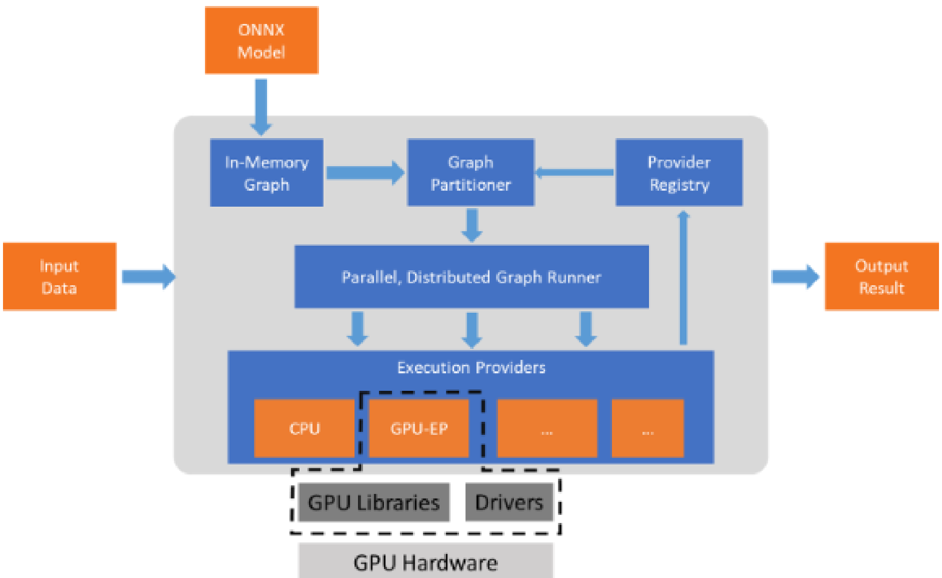


Figure 3: ONNX graph-level fusion and pruning

4.2 ONNX Latency

Average ONNX Runtime latency = 287.504 ms

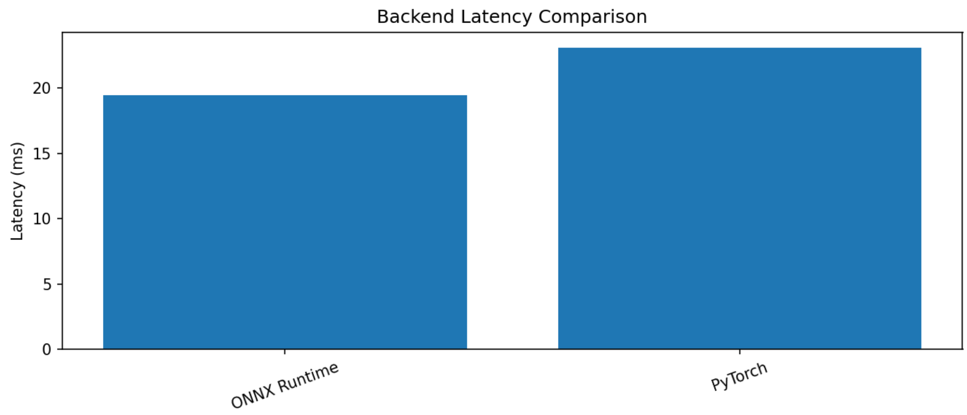


Figure 4: ONNX v/s PyTorch Model Performance

4.3 Operator-Level Graph Transformations

PyTorch profiling shows that more than 75% of total GPU time is spent in the operators

$$\{\texttt{aten::addmm}, \texttt{aten::linear}, \texttt{aten::conv1d}, \texttt{aten::cudnn\_convolution}, \texttt{aten::bmm}\},$$

which correspond to linear projections, convolutional feature extraction, and attention matrix multiplications.

- **Linear Projection Fusion:** A PyTorch linear layer computes

  $$y = xW^\top + b,$$

  which is executed as separate matrix multiplication and bias addition kernels. ONNX rewrites this pattern into a single `Gemm` node:

  $$\texttt{Gemm}(x, W, b),$$

  eliminating intermediate tensors and reducing kernel launches from $2 \to 1$.

- **Convolution and Bias Folding:** In eager execution, 1D convolution is evaluated as

  $$y = \text{Conv1d}(x, w) + b,$$

  followed by an optional activation. ONNX folds the bias term directly into the convolution operator and normalizes the layout, enabling downstream runtimes to emit a single fused convolution kernel with reduced memory traffic.

- **Batched Matrix Multiplication in Attention:** Attention blocks rely on batched matrix multiplications:

  $$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V,$$

  where the dominant cost arises from `bmm`. ONNX preserves the matmul structure while allowing layout reordering and operator grouping, which enables TensorRT to fuse multiple `bmm` operations into optimized attention kernels.

- **Net Effect:** ONNX optimization reduces the computational graph from many fine-grained operations to a smaller set of fused nodes:

$$\text{More kernels} \;\rightarrow\; \text{Fewer, wider kernels},$$

leading to lower launch overhead, fewer memory reads/writes, and improved GPU utilization. This explains the observed latency reduction when moving from PyTorch to ONNX Runtime and TensorRT.

4.4 ONNX Performance Comparison

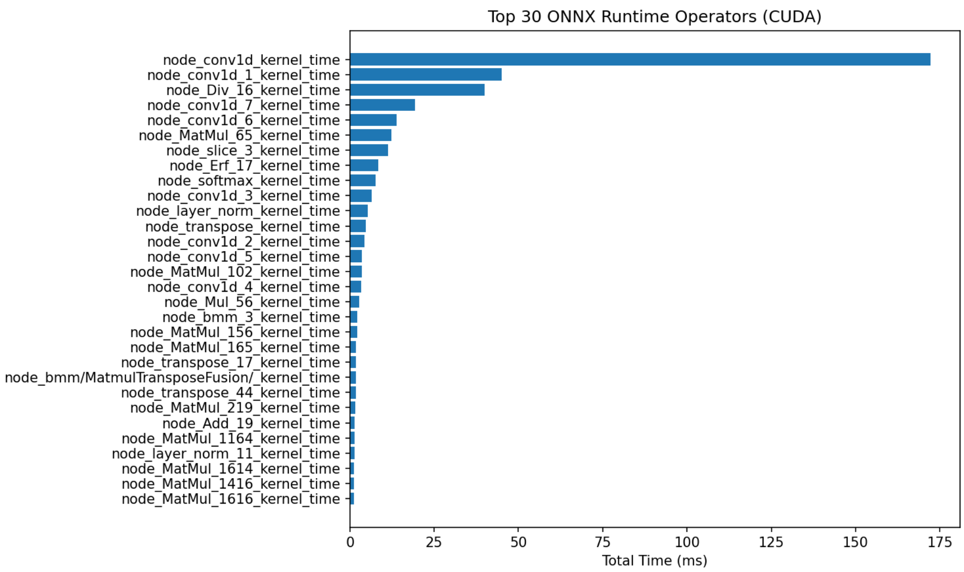


Figure 5: Top

5 TensorRT Engine Optimization

5.1 What TensorRT does (simple explanation)

TensorRT ingests the ONNX graph and:

- Maps compatible ops to highly-optimized kernels (cuDNN / cutlass),
- Fuses consecutive operations into single kernels (reducing kernel launches),
- Uses lower-precision computation (FP16/INT8) where safe for speed and memory,
- Reorders tensors to layouts preferred by hardware (NCHW/NHWC) and
- Minimizes memory traffic by allocating optimal workspace and reusing buffers.

5.2 Why FP16 / Mixed Precision help

Lower precision reduces memory bandwidth and allows faster math (tensor cores). Mixed precision keeps numerically sensitive ops in FP32 while accelerating the rest. Thus accelerating model performance.

5.3 Why INT8 is trickier

INT8 quantization gives larger gains but requires calibration data and careful error monitoring. For ASR, mapping probability outputs to a small integer range can hurt WER unless calibrated correctly.

5.4 Engine Build Configuration

- Precision: FP16 & Mixed Precision
- Dynamic batch support
- Layer Folding
- Kernel Fusion for faster inference

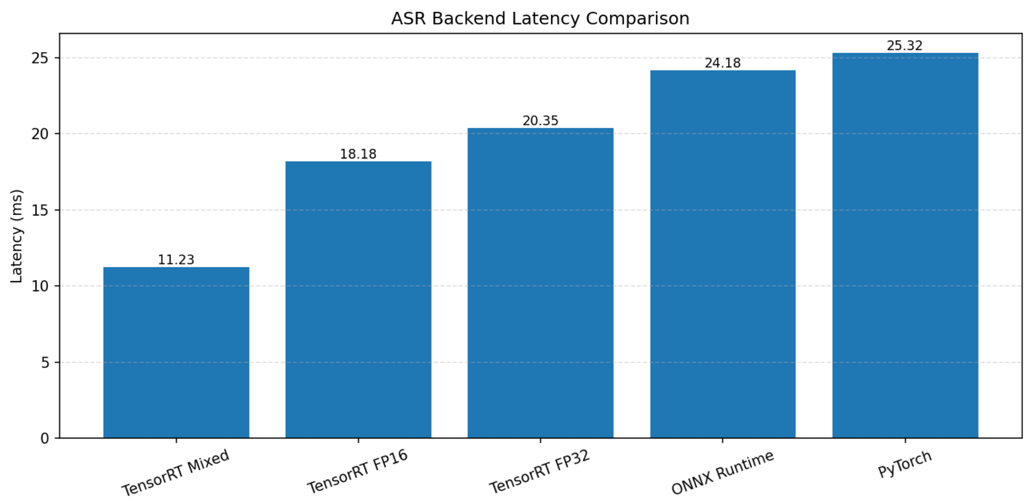


Figure 6: TensorRT Engine Latency Comparision

Key takeaway: Mixed precision delivers the best latency–accuracy trade-off for ASR workloads.

5.5 TensorRT Optimizations on ONNX Models

- **Graph-Level:** TensorRT fuses adjacent layers and eliminates redundant operators for efficiency.

$$\text{Conv} + \text{BN} + \text{ReLU} \;\Rightarrow\; \text{FusedKernel}$$

$$\text{Dropout} \;\Rightarrow\; , \quad \text{Concat} \oplus \text{Split} \oplus \text{Slice} \;\Rightarrow\; \text{SimplifiedOp}$$

- **Precision Reduction:** Models are quantized to lower precision for faster inference and smaller memory footprint.

$$\text{FP32} \to \text{FP16} \quad (\times 2 \text{ speedup}), \qquad \text{FP32} \to \text{INT8} \quad (\times 4 \text{ compression})$$

Calibration ensures accuracy is preserved:

$$\text{INT8}(x) \approx \text{scale} \cdot \text{round}\left(\frac{x}{\text{scale}}\right)$$

- **Kernel Auto-Tuning:** TensorRT selects the fastest kernel implementation for each layer.

$$\text{Layer} \mapsto \arg\min_{k \in \mathcal{K}} T(k, \text{shape}, \text{GPU})$$

where T is execution time and $\mathcal{K}$ is the set of candidate kernels.

- **Memory Management:** Dynamic allocation reduces Peak Memory usage and Bandwidth requirements.

$$M_{\text{peak}} \downarrow, \quad B \downarrow \quad \text{via fusion}$$

- **Input Shape Optimization:** TensorRT optimises inference for both fixed and dynamic input dimensions.

$$\text{Static: } (H, W) \text{ fixed}, \qquad \text{Dynamic: } (H, W) \in \Omega$$

6 Triton Inference Server Evaluation

6.1 Deployment Setup

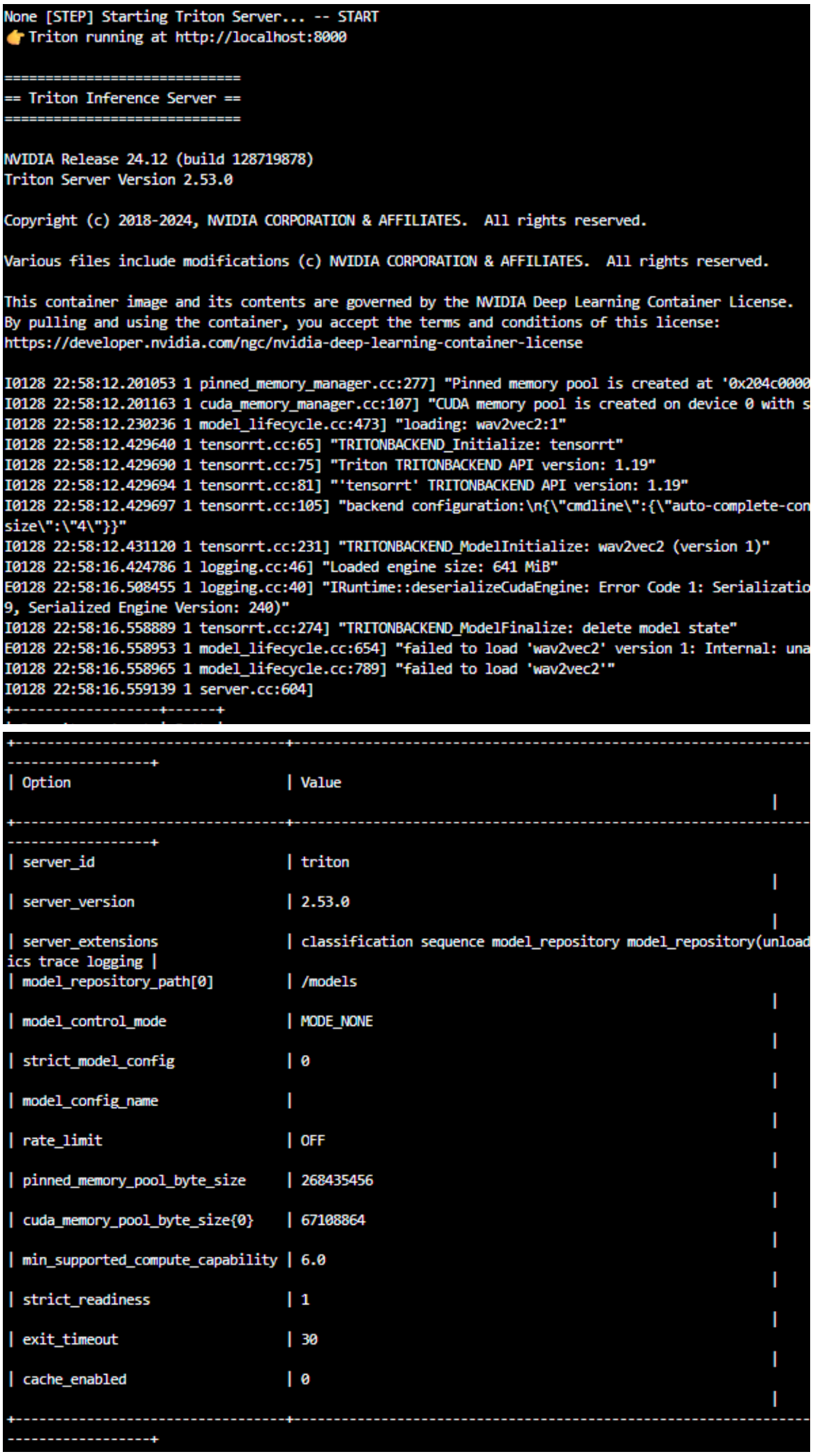
- Backend: TensorRT Mixed Precision Model (Best Performance)
- Dynamic batching enabled
- Multiple instance groups

6.2 Stress Testing

Triton was evaluated under repeated inference requests to analyze:

- Latency stability under load
- Throughput–latency trade-offs
- Output numerical consistency

6.3 Triton Inference Server Results


```
None [STEP] Starting Triton Server... -- START
👉 Triton running at http://localhost:8000

=============================
== Triton Inference Server ==
=============================

NVIDIA Release 24.12 (build 128719878)
Triton Server Version 2.53.0

Copyright (c) 2018-2024, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION & AFFILIATES.  All rights reserved.

This container image and its contents are governed by the NVIDIA Deep Learning Container License.
By pulling and using the container, you accept the terms and conditions of this license:
https://developer.nvidia.com/ngc/nvidia-deep-learning-container-license

I0128 22:58:12.201053 1 pinned_memory_manager.cc:277] "Pinned memory pool is created at '0x204c0000
I0128 22:58:12.201163 1 cuda_memory_manager.cc:107] "CUDA memory pool is created on device 0 with s
I0128 22:58:12.230236 1 model_lifecycle.cc:473] "loading: wav2vec2:1"
I0128 22:58:12.429640 1 tensorrt.cc:65] "TRITONBACKEND_Initialize: tensorrt"
I0128 22:58:12.429690 1 tensorrt.cc:75] "Triton TRITONBACKEND API version: 1.19"
I0128 22:58:12.429694 1 tensorrt.cc:81] "'tensorrt' TRITONBACKEND API version: 1.19"
I0128 22:58:12.429697 1 tensorrt.cc:105] "backend configuration:\n{\"cmdline\":{\"auto-complete-con
size\":\"4\"}}"
I0128 22:58:12.431120 1 tensorrt.cc:231] "TRITONBACKEND_ModelInitialize: wav2vec2 (version 1)"
I0128 22:58:16.424786 1 logging.cc:46] "Loaded engine size: 641 MiB"
E0128 22:58:16.508455 1 logging.cc:40] "IRuntime::deserializeCudaEngine: Error Code 1: Serializatio
9, Serialized Engine Version: 240)"
I0128 22:58:16.558889 1 tensorrt.cc:274] "TRITONBACKEND_ModelFinalize: delete model state"
E0128 22:58:16.558953 1 model_lifecycle.cc:654] "failed to load 'wav2vec2' version 1: Internal: una
I0128 22:58:16.558965 1 model_lifecycle.cc:789] "failed to load 'wav2vec2'"
I0128 22:58:16.559139 1 server.cc:604]
+------------------+------+
```

Option	Value
server_id	triton
server_version	2.53.0
server_extensions	classification sequence model_repository model_repository(unload ... ics trace logging
model_repository_path[0]	/models
model_control_mode	MODE_NONE
strict_model_config	0
model_config_name	
rate_limit	OFF
pinned_memory_pool_byte_size	268435456
cuda_memory_pool_byte_size{0}	67108864
min_supported_compute_capability	6.0
strict_readiness	1
exit_timeout	30
cache_enabled	0

Figure 7: Triton stress-testing behavior (Terminal Logs)

7 Consolidated Results

Table 1: Backend latency and accuracy summary

Backend	Latency (ms)	Speedup	WER	Notes
PyTorch (GPU)	25.32	1.00×	0.3683	Baseline execution
ONNX Runtime	24.18	1.04×	0.368	Graph Optimised
TensorRT FP32	20.352	1.24×	0.3686	Engine specialization
TensorRT FP16	18.18	1.39×	0.3691	Tensor Core acceleration
TensorRT Mixed	11.230	2.25×	0.3687	Best Latency-accuracy trade-off

8 Conclusion

This work demonstrates a complete and reproducible pathway from a research-grade PyTorch ASR model to a production-ready NVIDIA inference stack.

Key outcomes:

- End-to-end latency reduced by ∼2.25× using TensorRT mixed precision
- No significant degradation in WER
- Deployment achieved using industry-standard Triton Inference Server

The pipeline demonstrates the performance and latency optimizations achievable when deploying models beyond vanilla **PyTorch**. For real-time inference, achieving an approximate 2–3× boost in performance can be a significant game changer.