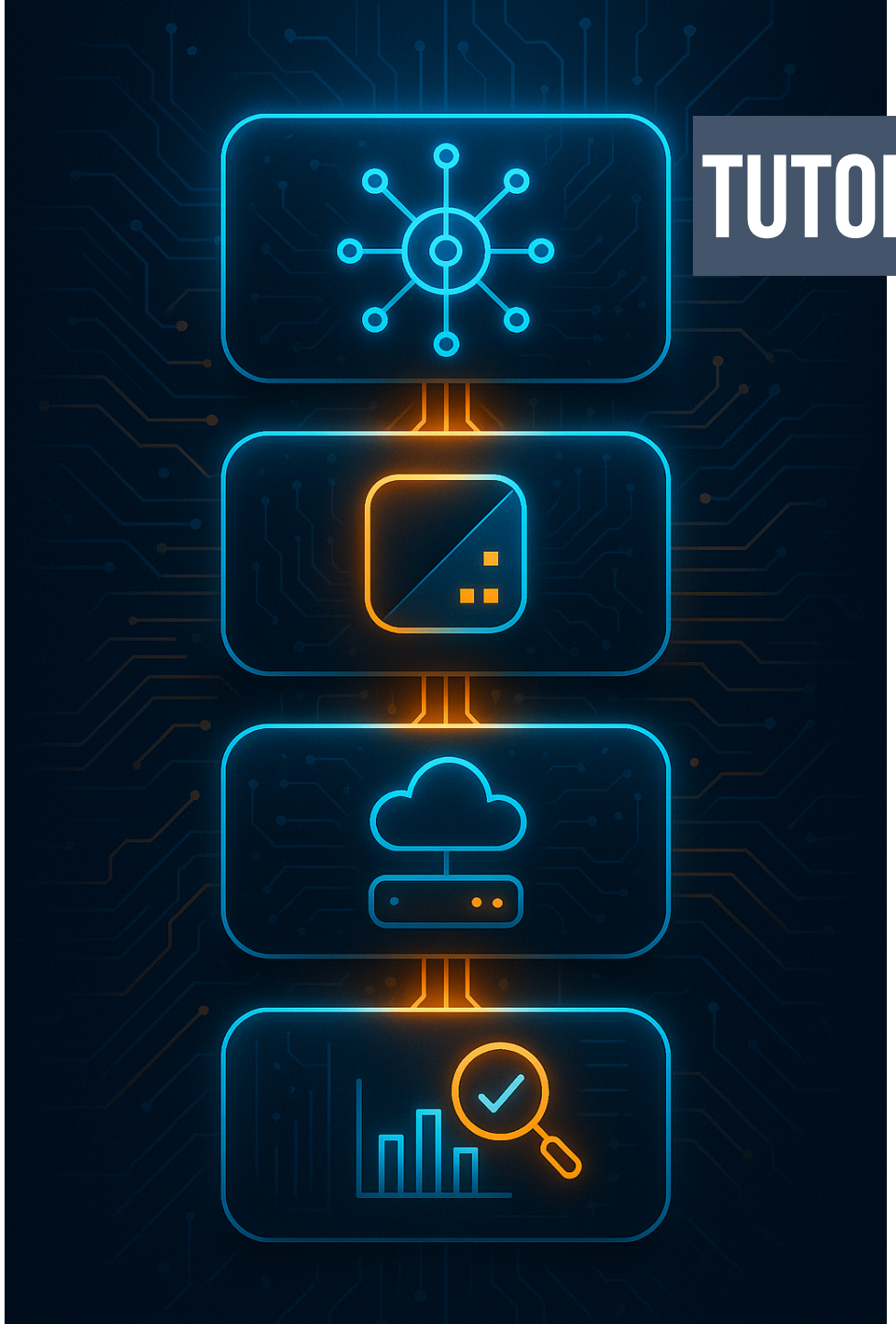
RESOURCE-CONSTRAINED VLM DEPLOYMENT ON EDGE AI

CVPR 2025 Tutorial

The IEEE/CVF Conference on Computer Vision and Pattern Recognition 2025

Nashville, TN, USA



TUTORIAL AGENDA

1 Motivation

2 Serving Engine

3 Quantization Methods

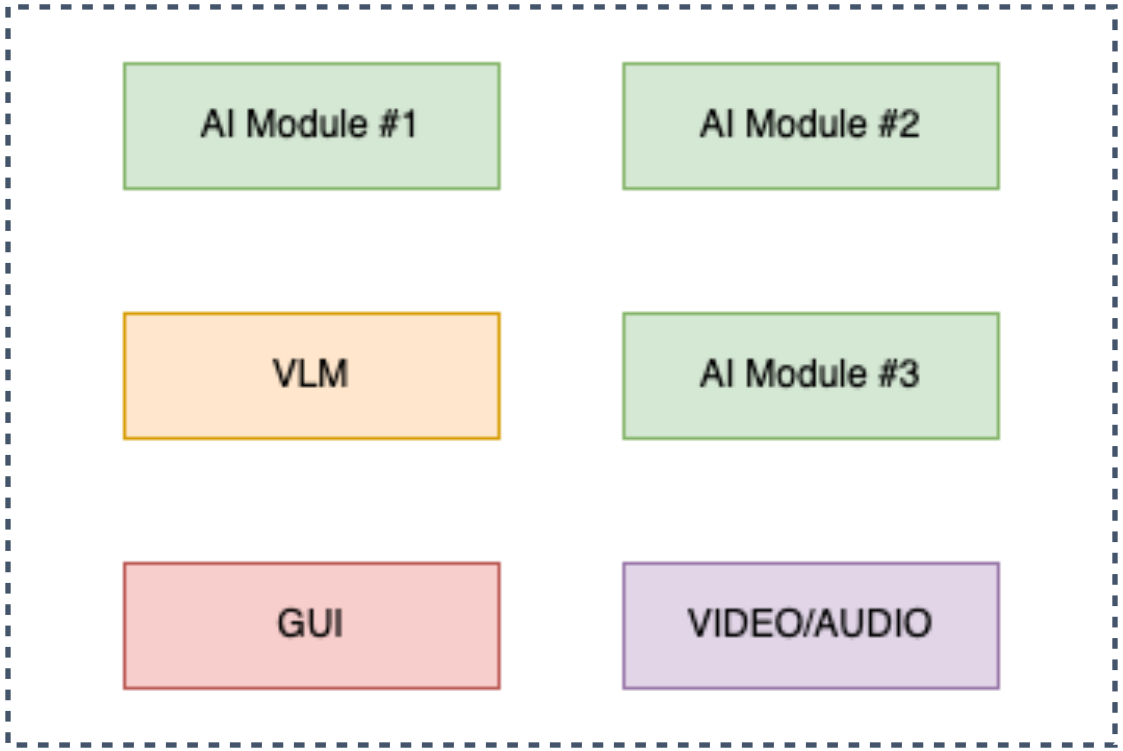
4 Evaluating Model Efficiency

5 Model Deployment

MOTIVATION

MULTI-AI SYSTEM ARCHITECTURE

Concurrent AI Modules with Lightweight VLM Deployment via API



PROPOSED SYSTEM ARCHITECTURE

A MODULAR AI SYSTEM WITH LOW-LATENCY VLM SERVICE AND OPTIMIZED MEMORY USE.

Concurrent Modules

Our system will consist of multiple AI modules operating concurrently.

Concurrent Modules

Although they are not running simultaneously, we want to keep them in memory to achieve lower latency.

Memory Efficiency

Due to this, we need the memory consumption of VLM to be a fraction of the total system memory.

API Deployment

VLM will be deployed as a service. It will be called through REST API / OpenAI API style.

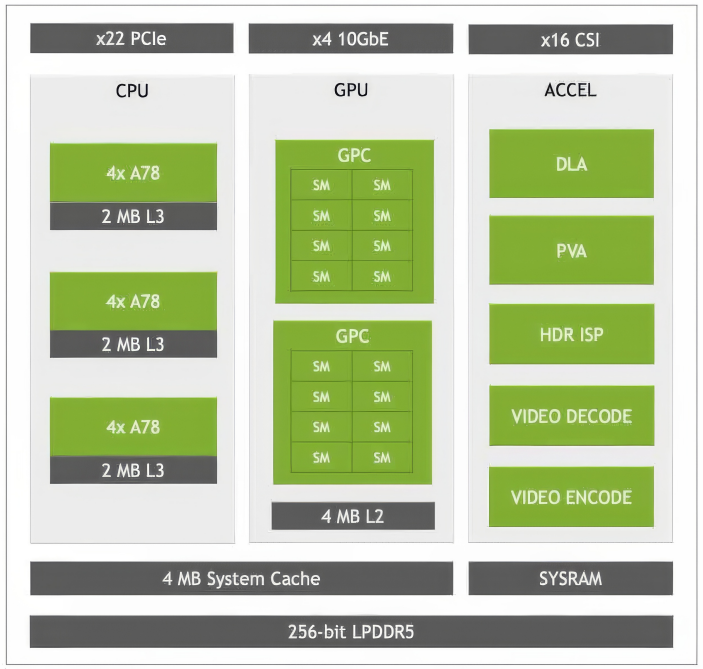
NVIDIA AGX ORIN ARCHITECTURE

Edge Device

Advanced CPU
- *12x Cortex A78 Cores ARM Arch V8.2
- *259 SPECint_rate 2006

Next-gen GPU
- *2 GPC / 8 TPC / 16 SMs
- *5.3 FP32 CUDA TFLOPs
- *10.6 FP16 CUDA TFLOPs

Higher DRAM BW
- 256-bit LPDDR5
- 205 GB/s



Rich IO Connectivity
- x4 10 GbE
- x22 PCIe, x16 CSI

DL Performance
- *275 INT8 DL Sparse TOPs
- *138 INT8 DL Dense TOPs

Enhanced PVA
- 512 INT16 GMAC/s
- 2048 INT8 GMAC/s

Shared Memory
All the modules will share 64GB of memory.

DLA Acceleration
We will be running a couple of our AI modules using DLA.

GPU Allocation
VLM, GUI, and another AI module will be utilizing a GPU.

INVESTIGATION GOALS

Overview

Serving Engine

Which serving engine can be used?

Best Quantization

Which quantization method gives us better performance?

Pipeline Optimization

Optimizing the pipeline for lower latency throughput.

SERVING VLMS ON JETSON ORIN AGX 64GB

JETSON ORIN AGX

Hardware

The NVIDIA Jetson Orin AGX 64GB is a powerful edge AI platform designed for high-performance inferencing tasks. It enables the deployment of large models such as vision-language LLMs directly on-device.

LLM Model

Capable of running Qwen-2.5 Vision-Language models (3B parameters) locally.

Storage

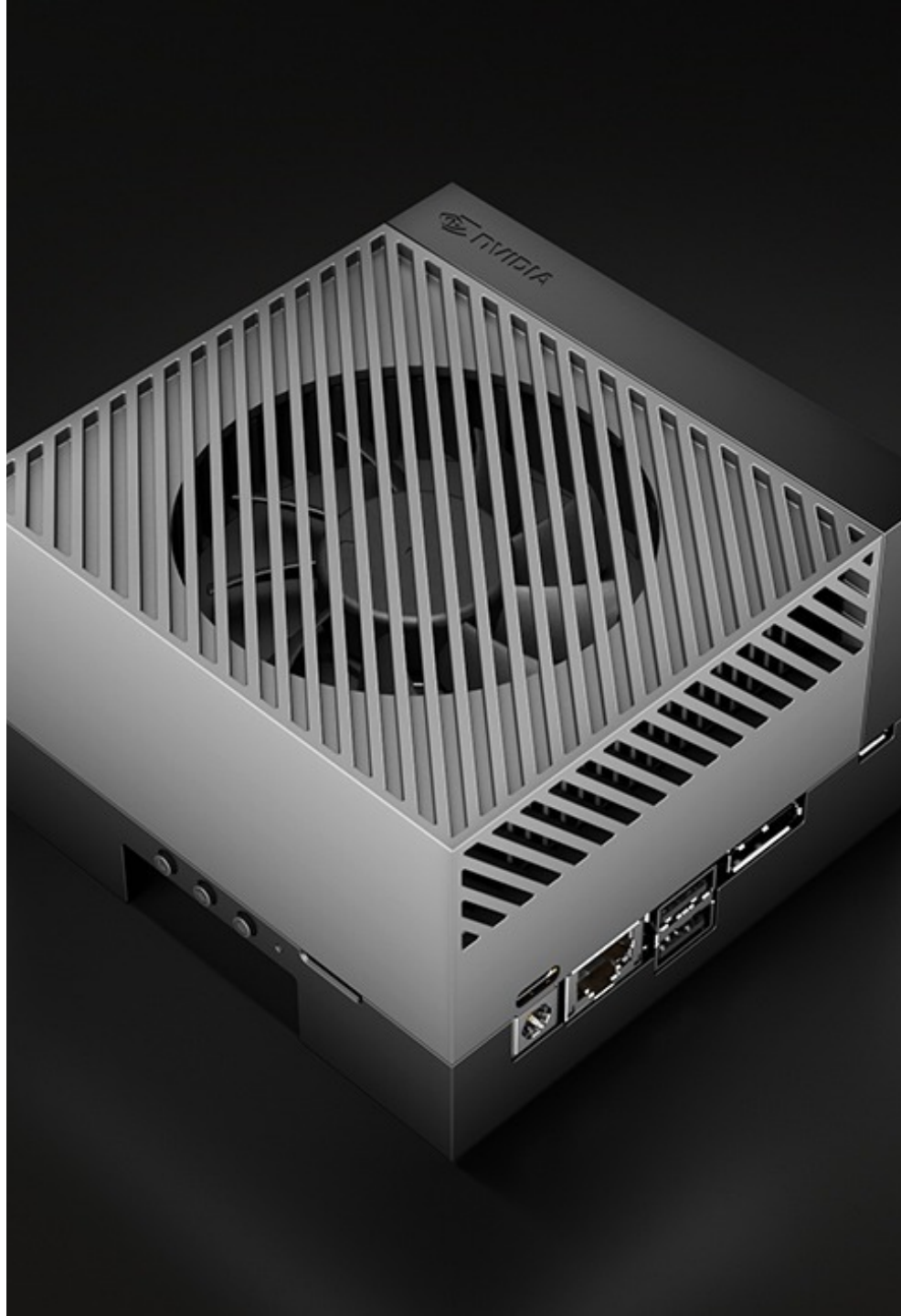
Provides ample high-speed storage for large model files, datasets, and application logs.

Operating System

Ships with the latest NVIDIA JetPack 6.2 SDK, supporting CUDA 12.6, cuDNN, TensorRT, and libraries optimized for edge deployment.

MAXQ Mode for Power

Configured to operate in MAXQ mode, maximizing power availability to deliver peak performance for GPU-heavy workloads.



SERVING ENGINE AND MODEL COMPATIBILITY

Overview

Serving Engine	Notes
MLC-LLM (github.com/mlc-ai/mlc-llm)	Many VLMs are not available. Can run on many hardwares.
TensorRT-LLM (github.com/NVIDIA/TensorRT-LLM)	Only v0.12 is available for Jetson and does not support many new models. v0.17 >= require CUDA 12.8 but tensorRT is not available for Jetson Orin yet.
Ollama (ollama.com)	Only GGUF quantization Easy to install
vLLM (github.com/vllm-project/vllm)	Latest version available and can use Qwen2.5VL models and other many VLM models. Supports various quantizations support.
SGLang (github.com/sgl-project/sglang)	Latest version available but there is a issue with sgl-kernels for vision models.

EVALUATING SERVING ENGINES

Installation

VLLM

Type this command on your Terminal:

```
$ docker pull dustynv/
vllm:0.8.6-r36.4-cu128-24.04
```

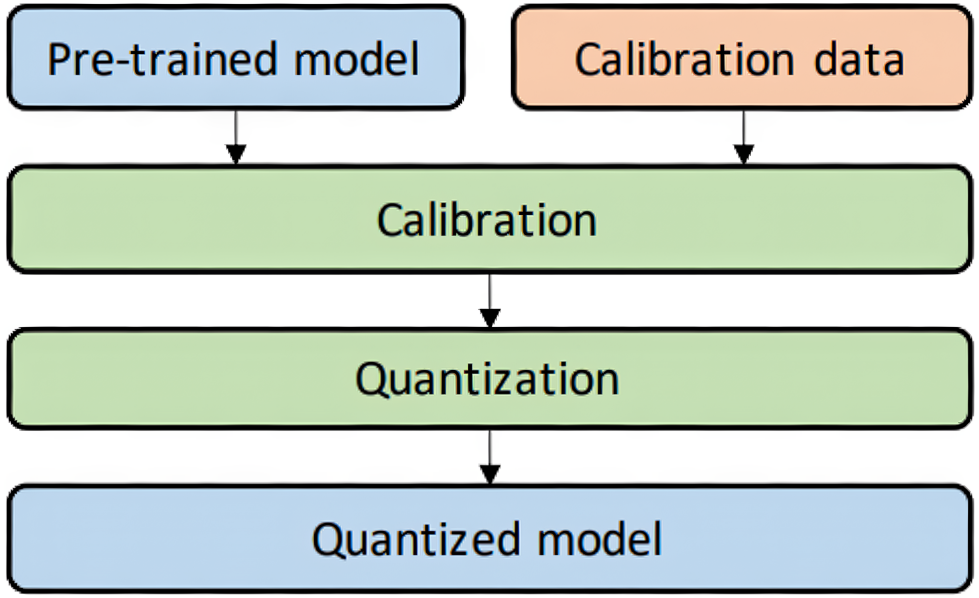
OLLAMA

Type this command on your Terminal:

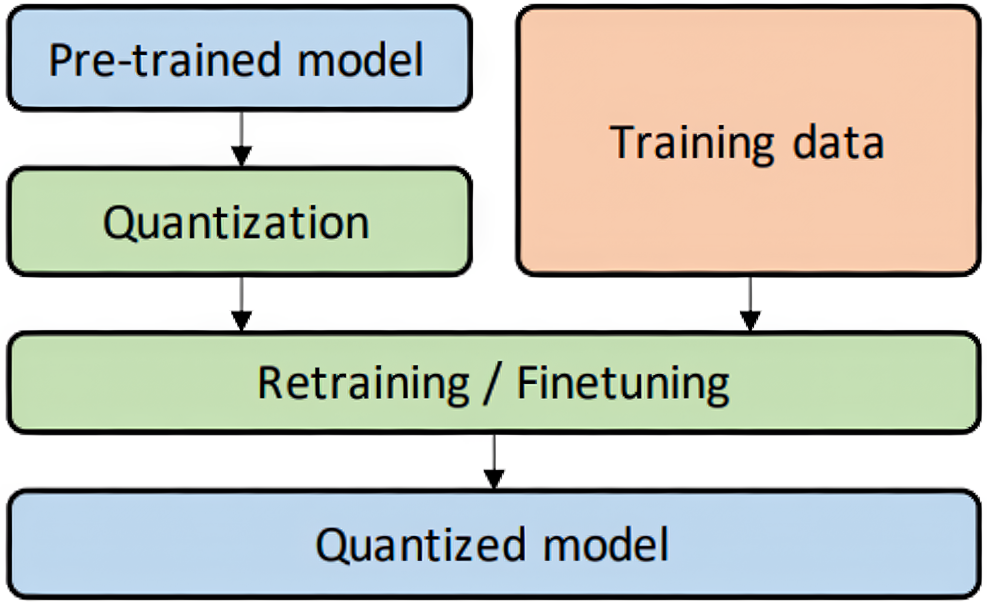
```
$ curl -fsSL https://
ollama.com/install.sh | sh
```

QUANTIZATION METHODS

A Survey of Quantization Methods for Efficient Neural Network Inference



Post Training Quantization



Quantization Aware Training

POST-TRAINING QUANTIZATION

PTQ

Static Quantization	Dynamic Quantization
Both weights and activations are pre quantized.	Only weights are pre quantized.
Require calibration dataset	Calibration dataset is optional
Can be optimized to the specific hardware for better efficiency	More portable and can be slower
TensorRT, ONNX, etc.,	ONNX, Bits and Bytes, etc.,

EVALUATED QUANTIZATION MODEL

Overview

Method	Quantization	Model
Original	BF16	Qwen/Qwen2.5-VL-3B-Instruct
AWQ	INT4	Qwen/Qwen2.5-VL-3B-Instruct-AWQ
GPTQ	INT4	RedHatAI/Qwen2.5-VL-3B-Instruct-quantized.w4a16
Bits and Bytes (Weights only)	4bit	unsloth/Qwen2.5-VL-3B-Instruct-bnb-4bit
TorchAO (Weights only)	INT8	testdummyvt/Qwen2.5-VL-3B-Instruct-int8-weightonly-torchao
GGUF	8bit	ollama run qwen2.5vl_3b-q8_0
GGUF	4bit	ollama run qwen2.5vl_3b-q4_K_M

AWQ AND GPTQ USING LLM-COMPRESSOR

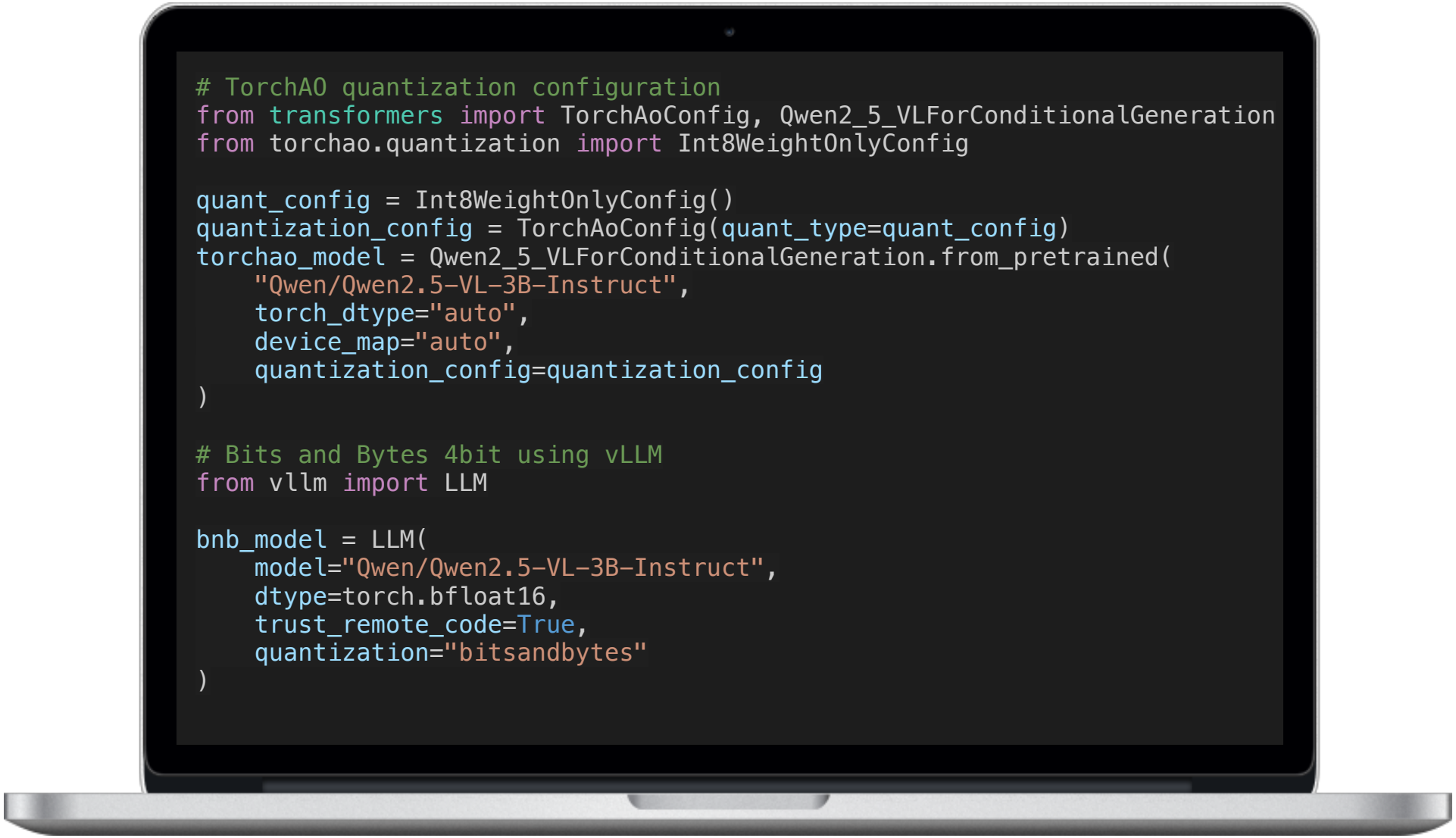
Comparison

```python
# Recipe
recipe = [
    AWQModifier(
        targets="Linear",
        scheme="W4A16",
        sequential_targets=["Qwen2_5_VLDecoderLayer"],
        ignore=["lm_head", "re:visual.*"],
    ),
]

# Perform oneshot
oneshot(
    model=model,
    tokenizer=model_id,
    dataset=ds,
    recipe=recipe,
    max_seq_length=MAX_SEQUENCE_LENGTH,
    num_calibration_samples=NUM_CALIBRATION_SAMPLES,
    trust_remote_code_model=True,
    data_collator=data_collator,
)
```

```python
# Recipe
recipe = [
    GPTQModifier(
        targets="Linear",
        scheme="W4A16",
        sequential_targets=["Qwen2_5_VLDecoderLayer"],
        ignore=["lm_head", "re:visual.*"],
    ),
]

# Perform oneshot
oneshot(
    model=model,
    tokenizer=model_id,
    dataset=ds,
    recipe=recipe,
    max_seq_length=MAX_SEQUENCE_LENGTH,
    num_calibration_samples=NUM_CALIBRATION_SAMPLES,
    trust_remote_code_model=True,
    data_collator=data_collator,
)
```

TORCHAO AND BNB USING LLM-COMPRESSOR

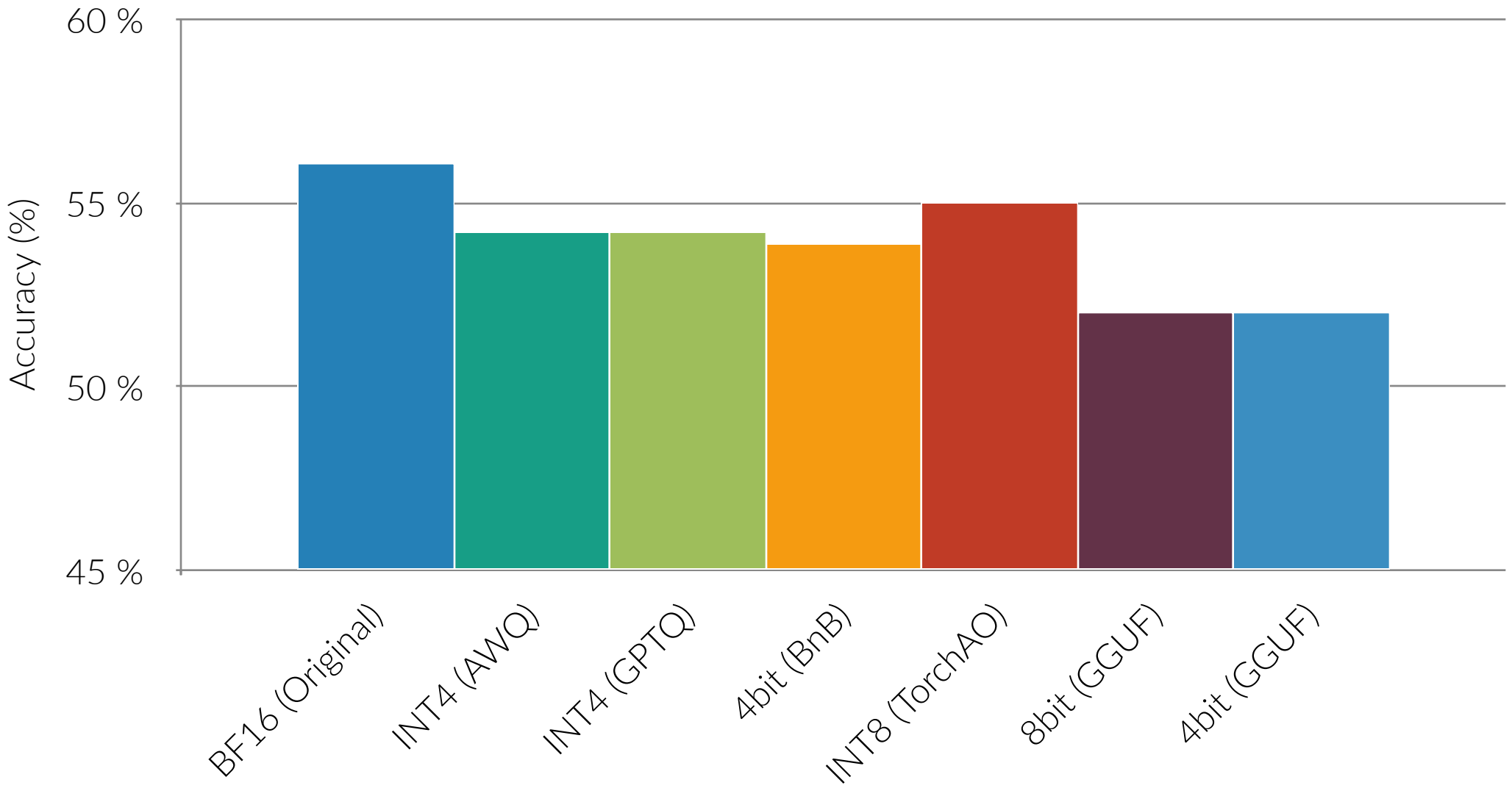
Comparison



```python
# TorchAO quantization configuration
from transformers import TorchAoConfig, Qwen2_5_VLForConditionalGeneration
from torchao.quantization import Int8WeightOnlyConfig

quant_config = Int8WeightOnlyConfig()
quantization_config = TorchAoConfig(quant_type=quant_config)
torchao_model = Qwen2_5_VLForConditionalGeneration.from_pretrained(
    "Qwen/Qwen2.5-VL-3B-Instruct",
    torch_dtype="auto",
    device_map="auto",
    quantization_config=quantization_config
)

# Bits and Bytes 4bit using vLLM
from vllm import LLM

bnb_model = LLM(
    model="Qwen/Qwen2.5-VL-3B-Instruct",
    dtype=torch.bfloat16,
    trust_remote_code=True,
    quantization="bitsandbytes"
)
```

QUANTIZATION VS ACCURACY

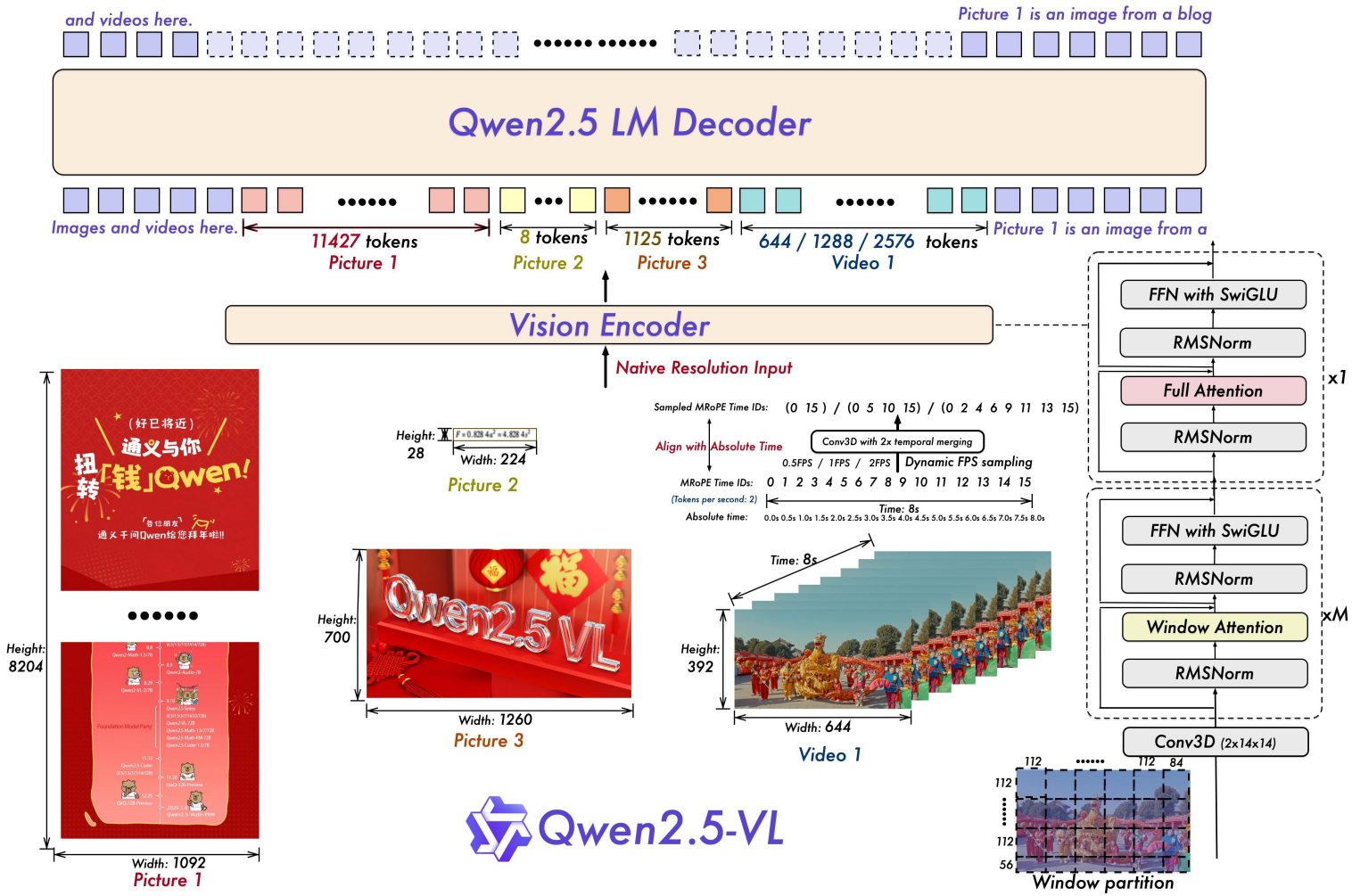
MMStar



IMAGE/VIDEO TOKENIZATION

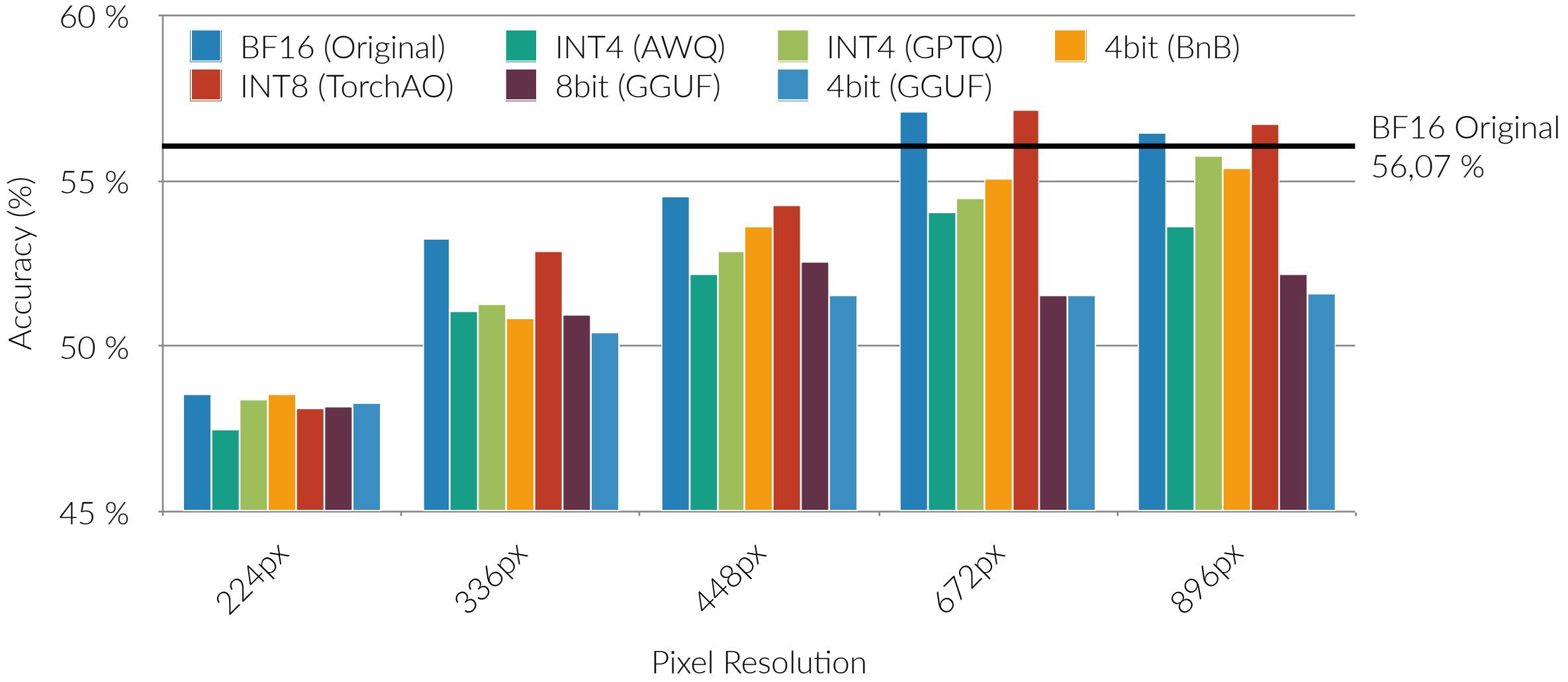
Qwen2.5VL Models

```
min_pixels = 256 * 28 * 28
max_pixels = 1280 * 28 * 28
processor = AutoProcessor.from_pretrained(
    "Qwen/Qwen2.5-VL-3B-Instruct",
    min_pixels=min_pixels,
    max_pixels=max_pixels
)
```



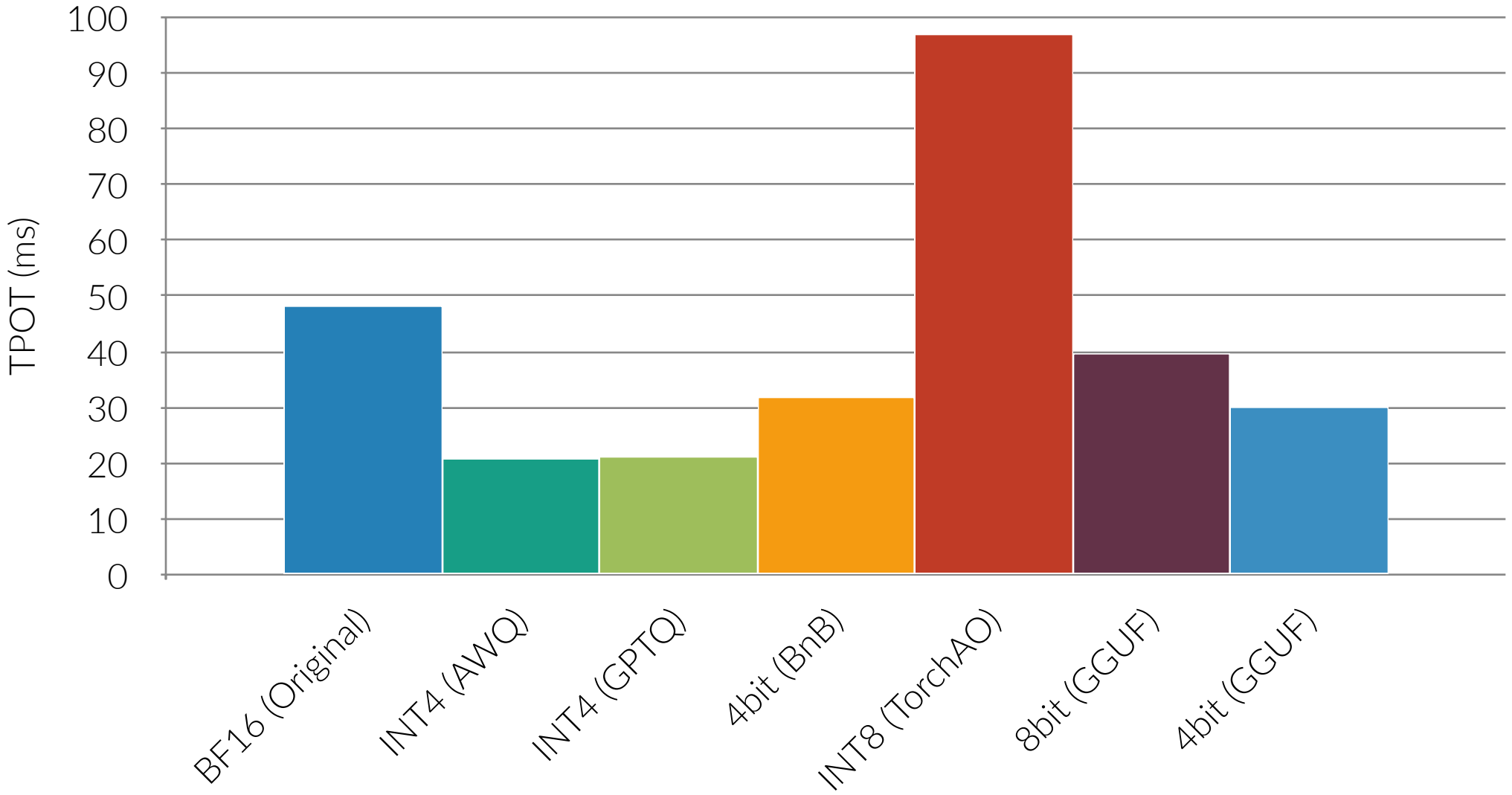
QUANTIZATION VS ACCURACY

At different image resolutions



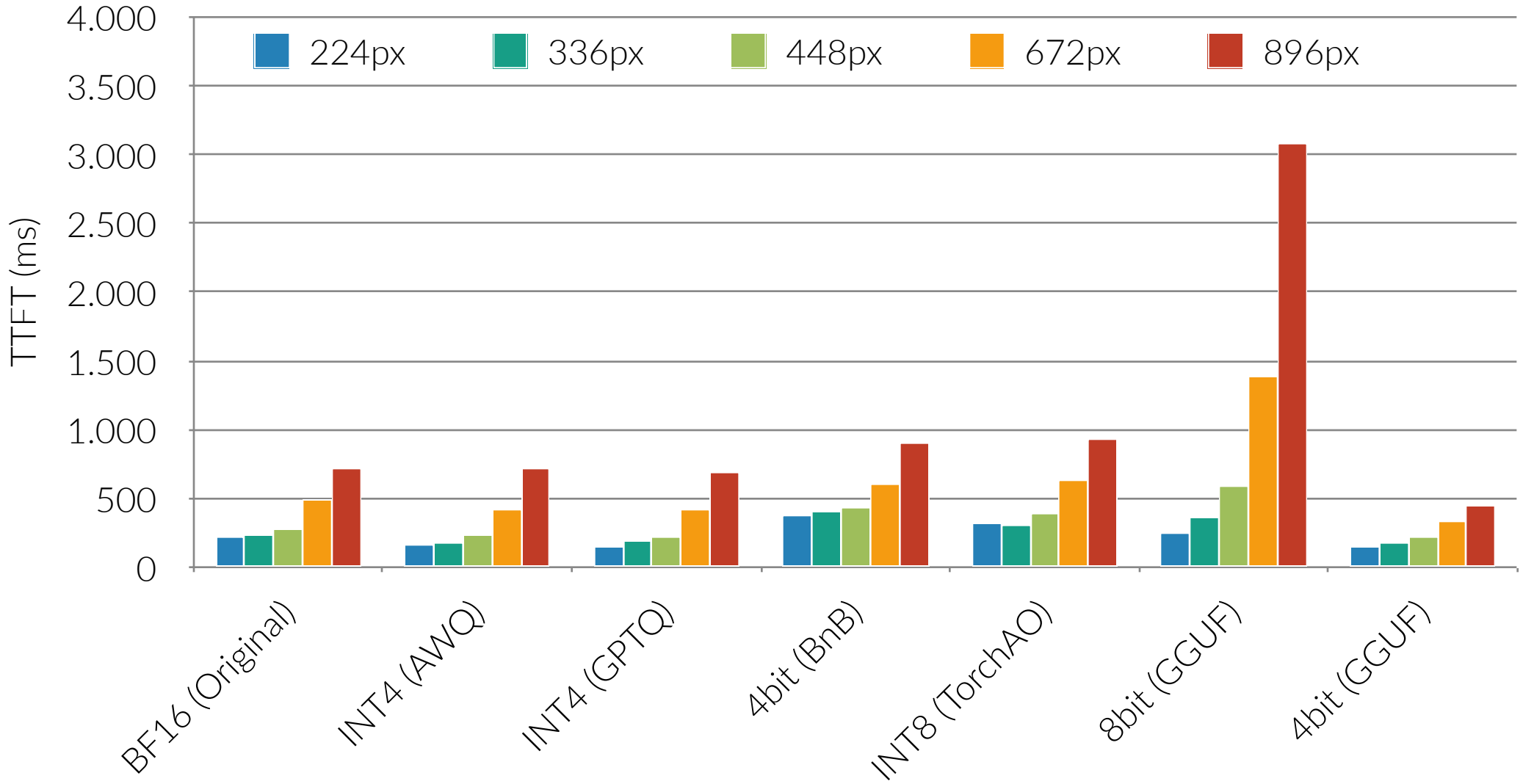
QUANTIZATION VS TPOT

Time per Output Token



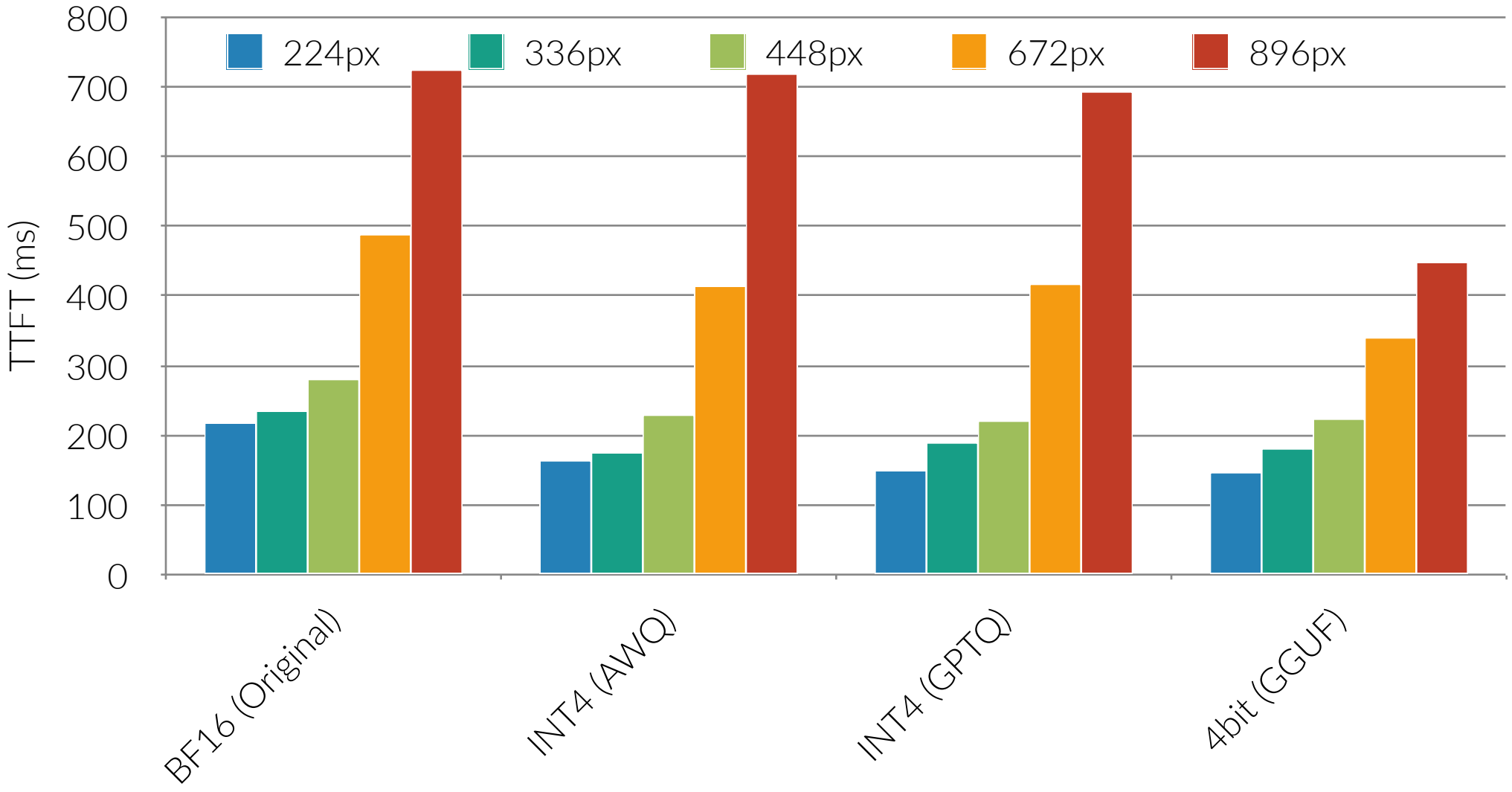
QUANTIZATION VS TTFT

Time to First Token



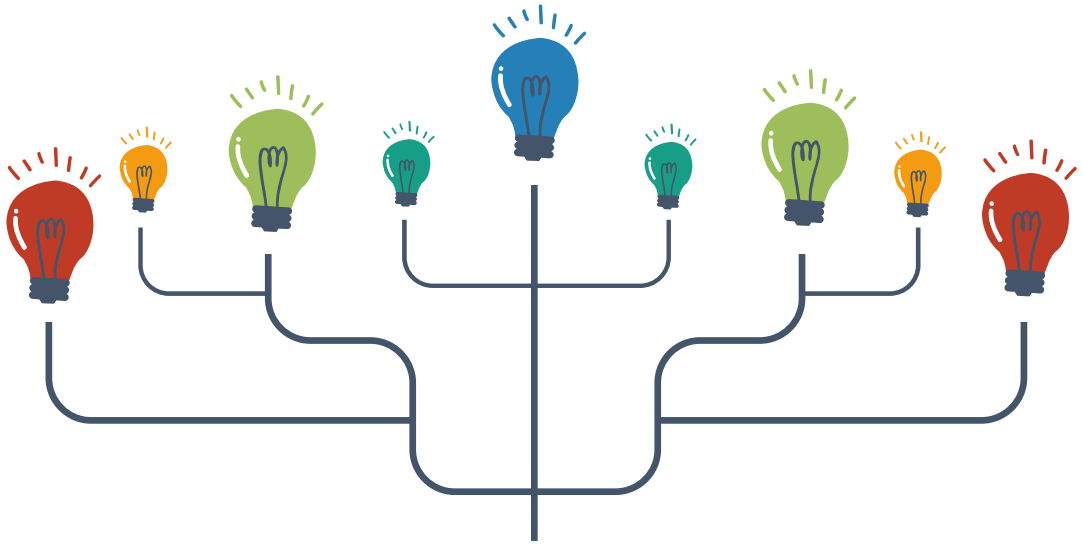
QUANTIZATION VS TTFT

TTFT for Selected Quantization Methods and Resolutions



OUR FINDINGS

Data Analysis



THAT'S SO GREAT IDEA

1. **vLLM + GPTQ**
   We consider vLLM with GPTQ W4A16 as the best balance of speed and accuracy.

2. **Resolution Depends**
   When it comes to resolution, it entirely depends on the task or objective of the VLM.

3. **Low-Res Tasks**
   Lower resolution works for large objects, grounding, and scene description.

4. **High-Res Needs**
   Higher resolution is required for granularity.

5. **Batching Optimized**
   If one needed to use batched input, GPTQ and AWS are still better options with vLLM as they are optimized kernels.

EVALUATED QUANTIZATION MODEL

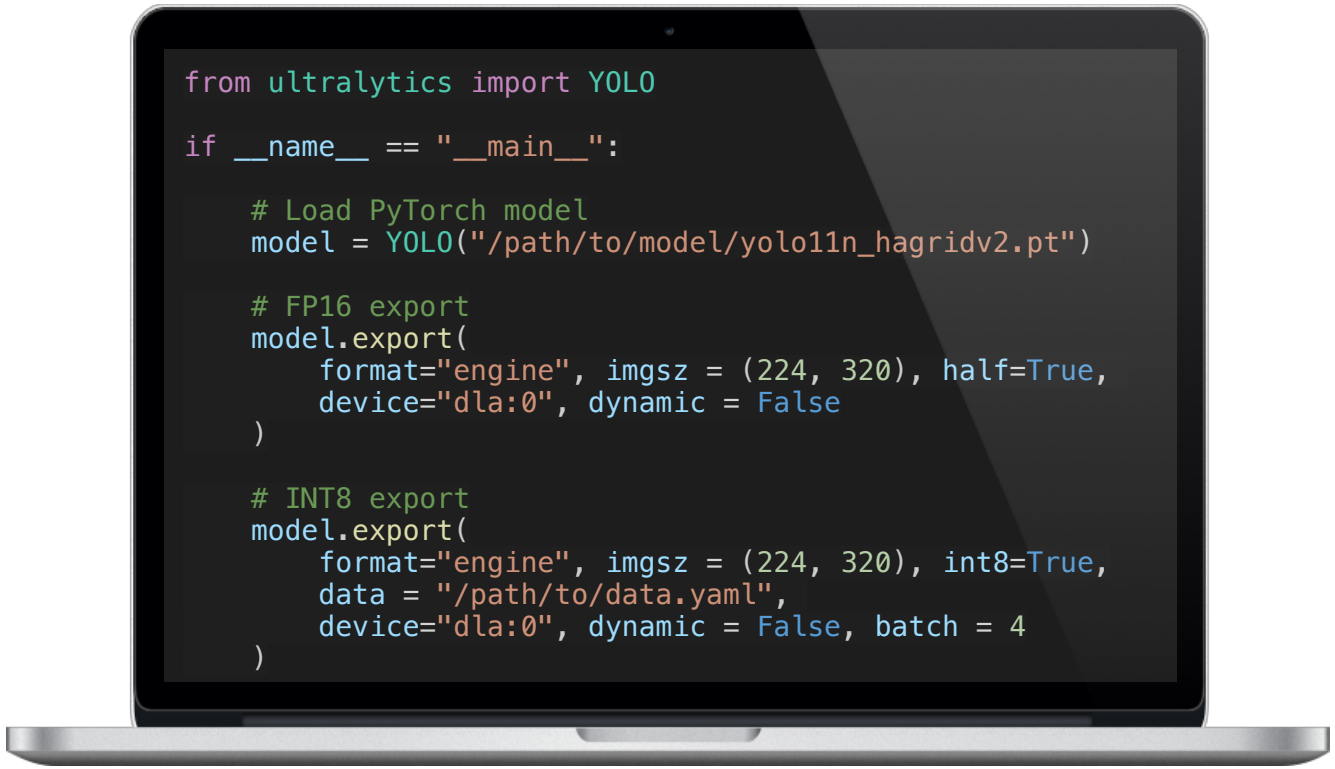
Overview

Quantization	Method	Tok/sec	req/sec
BF16	Original	152.67	1.19
INT4	AWQ	169.2	1.32
INT4	GPTQ	173.23	1.35
4bit	Bits and Bytes	119.33	0.93
INT8	TorchAO	127.18	0.99

DEPLOYING IN MULTI-AI SYSTEM ARCHITECTURE

YOLO11N ON DLA

Overview



```python
from ultralytics import YOLO

if __name__ == "__main__":

    # Load PyTorch model
    model = YOLO("/path/to/model/yolo11n_hagridv2.pt")

    # FP16 export
    model.export(
        format="engine", imgsz = (224, 320), half=True,
        device="dla:0", dynamic = False
    )

    # INT8 export
    model.export(
        format="engine", imgsz = (224, 320), int8=True,
        data = "/path/to/data.yaml",
        device="dla:0", dynamic = False, batch = 4
    )
```

- We used Ultralytics export to convert the PyTorch model to TensorRT.
- DLA supports FP16 and INT8.
- The issue with INT8 on DLA is that we need to calibrate the model.
- For DLA, the input and output shapes must be static.
- A batch size of at least 4 is required to achieve better results from quantization.

YOLO11N ON DLA

Results

Quantization	Speed	mAP
FP32 (PyTorch)	35FPS (GPU)	0.98
FP16	56FPS (DLA)	0.98
INT8 (batch=1)	70FPS (DLA)	0.88
INT8 (batch=4)	22 BPS (DLA)	0.93

*Note: We considered preprocessing + inference + post processing for FPS

MULTI-AI SYSTEM ARCHITECTURE

Prototype

MULTI-AI SYSTEM ARCHITECTURE

Notes

GPTQ W8A8

Got *No compiled cutlass_scaled_mm* issue with pytorch.

DYNAMIC WEIGHTS

Stuck at torch.complie during serving and Jetson randomly restarted.

BNB (HUGGING FACE)

Tried on the fly INT8 version of the model in vLLM itself. It runs, but Jetson turns off after overheating.

ONNX OR TENSORRT

Did not have enough time to deep dive into them.

STATIC SHAPE

Currently, static shape INT8 does not work for DLA TensorRT export. Open Issue: https://github.com/ultralytics/ultralytics/issues/20984

ACKNOWLEDGEMENTS

GitHub Repositories

Jetson Containers

[@dusty-nv, @johnnynunez] + [@shahizat]

Ultralytics

[@Y-T-G]

vLLM & llm-compressor

[@kylesayrs]

QUESTIONS & ANSWERS

THANK YOU!

CLOSING REMARKS AND JOINT Q&A