Bridging the Gap Between AI Quantization and Edge Deployment: INT4 and INT8 on the Edge

Mohammad Ibrahim Köse
Faculty of Engineering and Mathematics
Hochschule Bielefeld (HSBI)
Bielefeld, Germany
mohammad_ibrahim.koese1@hsbi.de

Qazi Arbab Ahmed
Faculty of Engineering and Mathematics
Hochschule Bielefeld (HSBI)
Bielefeld, Germany
qazi.ahmed@hsbi.de

Thorsten Jungeblut
Faculty of Engineering and Mathematics
Hochschule Bielefeld (HSBI)
Bielefeld, Germany
thorsten.jungeblut@hsbi.de

Abstract

Quantization is the key to deploying neural networks on microcontroller-class edge devices. While INT4 and mixed-precision schemes promise strong compression–accuracy trade-offs in simulation, current toolchains only support INT8 in practice. We benchmark FP32, INT8, INT4, and mixed-precision on Tiny YOLOv2, and deploy INT8 models on STM32N6, exposing this research–deployment gap. To address it, we propose a heterogeneous sub-INT8 strategy that combines INT8 acceleration with selective INT4 fallback execution, enabling practical hybrid deployment on today's edge hardware.

1 Introduction

Deploying object detectors on microcontroller-class edge devices remains challenging due to tight latency, memory, and energy budgets. Quantization is a standard lever to shrink models and accelerate inference Ding et al. [1]. Meanwhile, research-grade low-bit schemes (e.g., INT4) routinely show promising accuracy-compression trade-offs in simulation but fail to translate into real-time, on-device speedups due to the lack of native kernels and system-level support. While certain toolchains already provide preliminary support for mixed-precision settings (e.g., INT4 weights with INT8 activations) NVIDIA [2], Microsoft [3], Qualcomm Incorporated [4], such capabilities are rarely available for most microcontrollers, further limiting their practical adoption.

Contributions. The main contributions of this paper are as follows: (i) We present a controlled comparison of FP32, INT8 PTQ/QAT, INT4 QAT, and mixed precision on Tiny YOLOv2 with latency normalized to a fair FP32 baseline. (ii) We practically evaluate different implementations on STM32N6, contrasting simulation with deployable INT8 performance and memory use.

2 Background and Related Work

Quantization has emerged as a primary technique for compressing and accelerating deep neural networks on constrained hardware. While **INT8 quantization** has become the industry standard, supported by major frameworks such as TensorFlow Lite, ONNX Runtime, and vendor-specific

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MusiML@NeurIPS2025.

toolchains (e.g., STM32Cube.AI, Arm Ethos-U SDK, Google EdgeTPU compiler), research has increasingly shown the promise of **sub-INT8 schemes**.

Recent works on INT4 quantization [5–8] demonstrate competitive accuracy with significant memory and energy savings. NVIDIA's Hopper architecture, Qualcomm's HTP, and ONNX Runtime releases have also introduced preliminary support for mixed-precision execution, such as INT4 weights with INT8 activations. However, these developments largely target datacenter- or smartphone-class hardware, not microcontrollers.

On the embedded side, deployment remains limited to INT8 kernels. For example, TensorFlow Lite Micro executes only INT8 kernels efficiently, with other operators falling back to float or CPU execution. The Google Coral EdgeTPU supports only INT8 operators in TensorFlow LiteCoral AI [9], with unsupported layers falling back to the host CPU. Similarly, the Arm Cortex-M55 with Ethos-U55/U65 NPUs accelerates INT8 convolutions on the NPU while relying on the Cortex-M55 for unsupported operations Renesas Electronics Corporation [10]. On the STM32N6 platform, the Neural-ART™ accelerator executes INT8 operators, whereas unsupported operators are automatically mapped to the Cortex-M55 host.

This situation highlights a persistent *research–deployment gap*: sub-INT8 quantization works in simulation but cannot be deployed in practice on MCU-class hardware. Existing fallback mechanisms could serve as an enabler, yet they have not been systematically explored for heterogeneous quantization. Our work addresses this gap by evaluating INT4 and proposing a heterogeneous sub-INT8 deployment strategy that leverages fallback execution for selected layers.

3 Methodology

Dataset All experiments are conducted on the *VOC Person* subset derived from the PASCAL VOC benchmark. To simplify training while retaining a realistic detection setting, we only include images that contain at least one instance of the `person` class. This filtering reduces the dataset size but avoids trivial negatives and ensures that each sample contributes to the detection task. Images are resized to the input resolution required by each model and normalized to the $[0, 1]$ range. We follow standard practice in creating train, validation, and test splits, allocating 70% of the data for training, 20% for validation, and 10% for testing.

Models We use Tiny YOLOv2 Redmon and Farhadi [11] as the primary testbed for quantization experiments. This architecture is small enough to allow systematic evaluation across quantization settings but still representative of modern detection pipelines. For deployment experiments on STM32N6, we additionally evaluate two architectures provided in the STM32Cube.AI ecosystem, namely YOLOv8n and ST-YOLO-X STMicroelectronics [12], which are only available in INT8 quantized form. These larger models serve to demonstrate real-world feasibility of INT8 deployment on edge hardware.

Quantization Schemes We compare six representative schemes:

- **FP32 (Keras, optimized)**: Standard floating-point inference using BLAS-optimized kernels, serving as the accuracy and latency upper bound.
- **FP32 (QKeras baseline)**: A fake-quantized FP32 variant implemented in QKeras, where all layers use 32-bit quantizers. This ensures fair runtime comparison with quantized models, as all pass through the same operator paths.
- **INT8 PTQ (TFLite)**: Post-training quantization with TensorFlow Lite, supported by optimized kernels. This is the format deployed to STM32N6.
- **INT8 QAT (QKeras)**: Quantization-aware training with 8-bit weights and activations, representing a widely supported compromise between accuracy and efficiency.
- **INT4 QAT (QKeras)**: Aggressive 4-bit quantization for both weights and activations. As no native INT4 kernels exist in TensorFlow or STM32Cube.AI, this is evaluated only in simulation.
- **Mixed Precision (QKeras)**: A hybrid scheme with first and last layers quantized to 8 bits, and all intermediate layers using 1-bit weights and 4-bit activations. This stabilizes sensitive layers while compressing intermediate ones more aggressively.

Table 1: Simulation results on Tiny YOLOv2 (VOC person dataset). Latency measured in TensorFlow/QKeras; size reported relative to FP32 (QKeras baseline).

Scheme	mAP	Relative Latency	Latency (ms)
FP32 (QKeras baseline)	0.5731	1.00×	986
INT8 QAT	0.5110	0.92×	909
INT4 QAT	0.4224	0.89×	875
Mixed precision	0.3189	0.20×	198
FP32 (Keras, optimized baseline)	0.5882	1.00×	24
INT8 PTQ (tflite, optimized)	0.4794	0.83×	20

All QKeras-based models are trained with quantization-aware training to mitigate accuracy degradation.

Evaluation Metrics We evaluate accuracy in terms of mean Average Precision (mAP) at an Intersection-over-Union (IoU) threshold of 0.5. Inference latency is measured in two contexts: (i) within TensorFlow/QKeras for simulation, and (ii) on-device using STM32Cube.AI for deployment. Since absolute simulated latencies are inflated by the lack of optimized kernels, we report relative latency normalized to the QKeras FP32 baseline. This highlights speed-ups or slow-downs attributable to quantization under consistent conditions. For deployment on STM32N6, we additionally report throughput in frames per second (FPS), flash size, and RAM usage as provided by the toolchain. Together, these metrics capture accuracy-efficiency trade-offs across simulation and deployment.

4 Results

Simulation Results We first evaluate Tiny YOLOv2 with the different quantization schemes in TensorFlow using QKeras. The results are summarized in Table 1. To ensure fair runtime comparison, we use a QKeras-based FP32 baseline in which weights and activations are represented with 32-bit fake quantization. This baseline runs through the same computational graph as the quantized models, avoiding unfair acceleration from optimized BLAS kernels. The FP32 QKeras baseline requires approximately 986 ms per inference and achieves an mAP of 57.3%.

Among the quantized variants, INT8 QAT preserves most of the FP32 accuracy with a moderate latency reduction (909 ms, 92% relative latency). INT4 QAT achieves further compression but at the cost of reduced accuracy (42.2% mAP) and only slightly improved runtime (875 ms, 89% relative latency). The mixed-precision configuration, where the first and last layers are quantized to 8 bits while intermediate layers use 1-bit weights and 4-bit activations, demonstrates a more drastic speed-up (198 ms, 20% of the FP32 baseline). However, this comes at a considerable drop in accuracy (31.9% mAP). Overall, the QKeras experiments highlight the trade-off: lower precision can reduce runtime within the simulation environment, but accuracy degrades as the bit-width decreases, especially for aggressive mixed-precision settings.

For reference, the lower block of Table 1 reports results from optimized kernels in TensorFlow and TFLite. Here, FP32 inference with optimized BLAS kernels achieves only 24 ms latency while maintaining 58.8% mAP, and INT8 PTQ in TFLite achieves 20 ms latency at 47.9% mAP. This stark contrast illustrates the limitation of simulation-based latency measurements: while QKeras provides a fair platform to compare quantization schemes, its latencies are inflated due to the lack of dedicated low-bit kernels. In practice, hardware and optimized runtimes exploit vectorized instructions and kernel-level optimizations, which explains why absolute latencies differ by more than an order of magnitude compared to simulation.

Taken together, these findings emphasize two points: (i) quantization-aware training with INT8 achieves the best accuracy–efficiency balance in simulation, and (ii) INT4 and mixed-precision schemes remain promising from a model compression perspective, but their practical benefit cannot be realized without proper runtime support.

Deployment Results on STM32N6 We next deployed quantized models on STM32N6570-DK STMicroelectronics [13] using STM32Cube.AI. Unlike the simulation experiments, only INT8

Table 2: Deployment results on STM32N6.

Model	mAP@50	Throughput (FPS)	Weights (MB)	Act. (MB)
Tiny YOLOv2 INT8 PTQ	0.4794	33.3	10.775	0.016
YOLOv8n INT8 PTQ	0.5893	18.9	2.972	1.353
ST-YOLO-X INT8 PTQ	0.5231	22.3	0.891	0.016

quantization is supported by the toolchain. Table 2 reports the performance of Tiny YOLOv2, YOLOv8n, and ST-YOLO-X after INT8 PTQ. All three models achieve real-time throughput on the STM32N6, with YOLOv8n providing the best accuracy–speed balance. The memory footprint of INT8 models is well within the constraints of the device, confirming the efficiency of INT8 deployment.

Discussion The comparison between simulation and deployment highlights a significant gap. INT4 and mixed-precision quantization are promising in simulation, offering substantial size reductions with only minor accuracy loss, but are practically unusable due to high latency and lack of runtime support. On STM32N6, INT8 models run efficiently in real time, yet INT4 and mixed-precision are unsupported.

Limitations. This study has several limitations. First, all experiments were conducted on a single dataset (VOC person), which restricts the generalizability of the findings to broader detection tasks. Second, deployment results are limited to one hardware platform (STM32N6), so conclusions may not directly transfer to other MCU- or NPU-based systems. Finally, the INT4 results are obtained only in simulation, since no native INT4 kernels are currently available for deployment.

5 Heterogeneous Sub-INT8 Deployment

We propose a *heterogeneous sub-INT8 deployment* strategy to bridge the gap between promising simulation results for sub-INT8 quantization and the lack of native runtime support on current MCU-class hardware:

- **Compute-intensive layers** (e.g., large Conv2D blocks in the backbone) are quantized to INT8 and executed on the NPU/TPU, leveraging hardware acceleration.
- **Memory-dominant but latency-insensitive layers** (e.g., 1×1 convolutions, bottleneck transformations, or deeper feature processing) are quantized to INT4 and executed in software on the host CPU or DSP. This reduces flash and RAM usage while minimizing impact on overall throughput.

This principle is broadly applicable across embedded AI systems. For instance, Arm Cortex-M55 with Ethos-U55/U65 NPUs or the Google Coral EdgeTPU. By deliberately quantizing selected Conv2D layers to INT4, developers can exploit this fallback mechanism to deploy hybrid INT4/INT8 models.

In the short term, heterogeneous sub-INT8 deployment provides a practical mechanism to exploit INT4 quantization on hardware that officially supports only INT8. In the long term, we envision toolchains exposing explicit operator placement controls (e.g., `force_cpu`, `force_npu`) and NPUs adding native INT4 kernels. This would enable full exploitation of sub-INT8 quantization while retaining the performance benefits of hardware acceleration.

6 Conclusion and Future Work

We benchmarked FP32, INT8, INT4, and mixed-precision quantization on Tiny YOLOv2 and deployed INT8 models on STM32N6, exposing a persistent research–deployment gap: while sub-INT8 quantization demonstrates promising compression–accuracy trade-offs in simulation, current toolchains and NPUs only support INT8 execution.

To bridge this gap, we proposed a heterogeneous sub-INT8 deployment strategy: compute-intensive layers remain in INT8 and execute on the NPU, while memory-heavy but latency-insensitive convolutions are quantized to INT4 and executed on the CPU. This approach leverages existing fallback

mechanisms, provides a first pathway for practical INT4 deployment on current MCUs, and generalizes to other platforms that combine NPUs/TPUs with a host CPU.

Future work will focus on systematically identifying which layers benefit most from INT4 placement, developing compiler-level support for explicit operator placement (e.g., `force_cpu`, `force_npu`), and preparing for upcoming hardware generations with native sub-INT8 kernel support.

References

[1] Caiwen Ding, Shuo Wang, Ning Liu, Kaidi Xu, Yanzhi Wang, and Yun Liang. REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs, 2019. arXiv preprint.

[2] NVIDIA. Working with quantized types — NVIDIA tensorrt documentation (v10.9.0). `https://docs.nvidia.com/deeplearning/tensorrt/10.9.0/inference-library/work-quantized-types.html`, 2025. Accessed: 2025-09-19.

[3] Microsoft. ONNX runtime releases. `https://github.com/microsoft/onnxruntime/releases`, 2025. Accessed: 2025-09-19.

[4] Qualcomm Incorporated. HTP guidelines INT4 weights. `https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-50/htp_guidelines_int4_weights.html`, 2025. Accessed: 2025-09-19.

[5] Long Huang, Zhiwei Dong, Song-Lu Chen, Ruiyao Zhang, Shutong Ti, Feng Chen, and Xu-Cheng Yin. HQOD: Harmonious quantization for object detection, 2024. URL `https://arxiv.org/abs/2408.02561`.

[6] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. Understanding INT4 quantization for language models: Latency speedup, composability, and failure cases. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*. PMLR, 2023.

[7] Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2805–2814, 2019. doi: 10.1109/CVPR.2019.00292.

[8] Bestami Gunay, Sefa Burak Okcu, and Hasan Sakir Bilge. LP-YOLO: Low precision YOLO for face detection on FPGA. In *Proceedings of the 8th World Congress on Electrical Engineering and Computer Systems and Sciences*, 2022. doi: 10.11159/mvml22.108.

[9] Coral AI. Tensorflow models on the edge TPU. `https://coral.ai/docs/edgetpu/models-intro/`, 2025. Accessed: 2025-09-22.

[10] Renesas Electronics Corporation. Using the ethos-u npu with RA8 MCUs. Application Note R01AN7712EU0100 Rev.1.00, Renesas Electronics, July 18 2025. URL `https://www.renesas.com/en/document/apn/using-ethos-u-npu-ra8-mcus`.

[11] Joseph Redmon and Ali Farhadi. YOLO9000: Better, faster, stronger, 2016. URL `https://arxiv.org/abs/1612.08242`.

[12] STMicroelectronics. STM32 AI Model Zoo: Object detection use case. `https://github.com/STMicroelectronics/stm32ai-modelzoo/tree/main/object_detection`, 2025. Accessed: 2025-09-21.

[13] STMicroelectronics. Stm32n6570-dk discovery kit documentation. `https://www.st.com/en/evaluation-tools/stm32n6570-dk.html#documentation`, 2025. Accessed: 2025-09-21.