Comparative Analysis of YOLOv12 and DepthAnythingV2 Inference Performance Using PyTorch, ONNX Runtime, and NCNN

Nada Nadhira Najwa Mazaya[1], Fitri Utaminingrum[1]★, and Barlian Henryranu Prasetio[1]

Informatics Engineering Department, Faculty of Computer Science, Universitas Brawijaya, Malang, Indonesia
f3_ningrum@ub.ac.id

Abstract. CPU deployment requires inference frameworks that balance latency, throughput, memory, and accuracy. This paper presents a controlled comparison of PyTorch, ONNX Runtime, and NCNN on two representative workloads: real-time object detection with YOLOv12n and monocular depth estimation with DepthAnythingV2 (ViT-S). Unlike prior comparisons, we explicitly fix the number of CPU threads and pin execution to identical physical cores across all frameworks, removing a thread-allocation confound that otherwise inflates apparent speed differences. All latency figures are reported as mean ± standard deviation over 100 iterations after 20 warm-up runs. ONNX Runtime achieves the lowest latency on both tasks (e.g., 20.4 ms versus 31.0 ms for PyTorch and 46.6 ms for NCNN on detection at four threads). Object-detection accuracy is identical across all three frameworks, and ONNX Runtime reproduces PyTorch depth accuracy exactly; however, NCNN shows a substantial accuracy drop on the ViT-based depth model (δ_1 falling from 95.5% to 47.0%), which we trace to model conversion rather than measurement. A thread-scaling analysis further shows that unconstrained multi-threading causes CPU oversubscription with little or no latency benefit. These findings give practical guidance for CPU deployment and expose framework-specific limitations for transformer architectures.

Keywords: Inference frameworks · CPU deployment · ONNX Runtime · NCNN · YOLOv12 · DepthAnythingV2 · Monocular depth estimation · Benchmarking.

1 Introduction

Deep learning now drives computer-vision tasks such as object detection and monocular depth estimation, using convolutional and attention-based architectures [1–3] as well as object detectors [4, 5]. The convolution/attention boundary is no longer sharp: recent detectors such as YOLOv12 are convolution-dominant yet add lightweight attention [5], whereas depth models such as DepthAnythingV2 use a full ViT backbone [6]. PyTorch is common for development, but its CPU inference is often limited by runtime overhead [7], motivating optimized engines such as ONNX Runtime (graph optimization, operator fusion, and hardware-aware execution) [8] and NCNN (a lightweight, low-dependency CPU/mobile framework) [9].

★ Corresponding author.

© The Author(s) 2026
M. Murtadha et al. (eds.), *Proceedings of the 2026 11th International Conference on Sustainable Information Engineering and Technology (SIET 2026)*, Atlantis Highlights in Computer Sciences 26,
https://doi.org/10.2991/978-94-6239-828-3_7

A recurring weakness in existing comparisons is that runtime conditions are rarely controlled: frameworks are often benchmarked with different CPU thread counts, so reported speed gaps partly reflect thread allocation rather than the frameworks. This study presents a controlled deployment comparison of PyTorch, ONNX Runtime, and NCNN using YOLOv12n (convolution-dominant) and DepthAnythingV2 ViT-S (full-ViT) on a CPU platform. Our contributions are fourfold: (1) a deployment-oriented benchmark under an identical, pinned CPU-thread configuration, with results as mean ± standard deviation; (2) accuracy analysis across both architectures (detection mAP and depth metrics); (3) a thread-scaling analysis characterizing speedup and CPU oversubscription; and (4) an operator-level diagnosis localizing the NCNN depth degradation to a specific conversion stage.

2 Related Work and Background

2.1 Depth Anything V2 Model

DepthAnythingV2 is a transformer-based monocular depth model with a ViT (DINOv2) backbone that captures global context via self-attention [6]. Its default checkpoints output affine-invariant (relative) depth rather than metric depth, which affects the evaluation described in Sect. 3.5. We use the ViT-S variant for its accuracy/size/cost balance; ViT-based depth on constrained devices has been studied previously [10].

2.2 Vision Transformers

A ViT processes an image as a patch sequence via multi-head self-attention [1], capturing long-range dependencies across vision tasks [2, 3]. Its specialized operators (layer normalization, GELU, and softmax) are less well characterized under conversion to lightweight engines than convolutions, and efficient mobile ViTs remain an active research area [11].

2.3 YOLOv12

YOLOv12 is a single-pass real-time detector [5]; unlike earlier purely convolutional YOLO models [4], it is attention-centric yet convolution-dominant, preserving real-time speed. Quantized YOLOv12 has been deployed in constrained settings [12], and efficient CNN backbones such as MobileNetV3 [13] motivate lightweight CPU deployment.

2.4 ONNX

ONNX is an interoperability standard; PyTorch models can be exported to ONNX and run with ONNX Runtime, which applies graph optimization, operator fusion, and hardware-aware execution [8, 14].

2.5 NCNN

NCNN is a lightweight, dependency-free inference framework for CPU/mobile deployment [9]; models are converted from ONNX before deployment. We evaluate NCNN for both a convolution-dominant and a transformer-based model under identical CPU conditions.

3 Proposed Work

3.1 Methodology

This study proposes a controlled, deployment-oriented benchmark of three inference frameworks: PyTorch, ONNX Runtime, and NCNN. As shown in Fig. 1, PyTorch models are exported to ONNX and then converted to NCNN; all three are benchmarked on the same models and inputs. Unlike prior comparisons, we fix the CPU thread count and pin execution to identical physical cores across all frameworks (Sect. 3.3), so differences reflect the frameworks, not thread allocation. All latency figures are mean ± standard deviation over 100 iterations after 20 warm-up runs.

3.2 Hardware and Experimental Environment

Experiments ran on one workstation: Intel Core i7-14700F (8 P-cores with Hyper-Threading up to 5.3–5.4 GHz plus 12 E-cores up to 4.2 GHz; 20 physical and 28 logical cores), 31 GiB RAM, and Ubuntu 24.04.3 LTS (kernel 6.8.0-124). The software stack comprised Python 3.12.3, PyTorch 2.11.0, ONNX Runtime 1.26.0, NCNN 1.0.20260114, and Ultralytics 8.4.46. All tests used batch size 1 and a CPU execution provider, with 100 measured iterations after 20 warm-up runs (mean ± standard deviation). CPU utilization is the sum of busy time across threads (as in `top`/`ps`), so one thread reads ≈100% and four threads ≈400%, explaining values above 100% and the approximately 1900% figure (about 19 active threads).

3.3 Threading Protocol

Each framework was constrained to an identical thread count T and pinned to the same physical P-cores. We configured the OpenMP, MKL, and OpenBLAS thread-count environment variables before start. PyTorch used `set_num_threads(T)` and

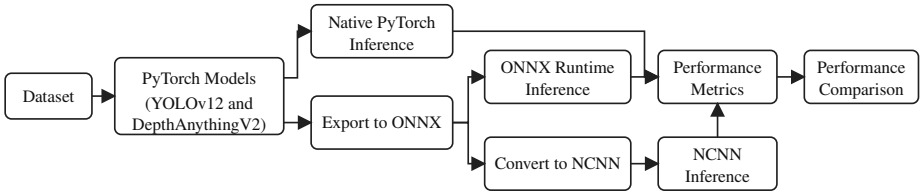


Fig. 1. Research procedure

`set_num_interop_threads(1)`. ONNX Runtime used an intra-op thread count of T, an inter-op count of 1, and sequential execution. NCNN's network option was set to T threads. Execution was pinned with `taskset` to one logical CPU per distinct P-core (avoiding Hyper-Threading siblings and E-cores), verified at runtime; each framework ran as a separate process. We report $T = 1$ (strict per-core) and $T = 4$, with a scaling sweep reported in Table 2.

3.4 Evaluation Metrics

Deployment efficiency is measured with inference latency and throughput; prediction quality is measured with depth-accuracy metrics and object-detection mAP. For N evaluated samples or valid pixels, d_i denotes the predicted depth and d_i^* the corresponding ground-truth depth.

Latency Average inference latency is computed as

$$L = \frac{t_{\text{total}}}{N}. \tag{1}$$

Frames Per Second When latency L is expressed in seconds, throughput is

$$\text{FPS} = \frac{1}{L}. \tag{2}$$

Absolute Relative Error The absolute relative error is

$$\text{AbsRel} = \frac{1}{N} \sum_{i=1}^{N} \frac{|d_i - d_i^*|}{d_i^*}. \tag{3}$$

Root Mean Square Error The root mean square error is

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (d_i - d_i^*)^2}. \tag{4}$$

Delta Accuracy Delta accuracy is the proportion of valid pixels satisfying

$$\delta_k : \quad \max\left(\frac{d_i}{d_i^*}, \frac{d_i^*}{d_i}\right) < 1.25^k, \qquad k \in \{1, 2, 3\}. \tag{5}$$

Mean Average Precision For C object classes, mean average precision is

$$\text{mAP} = \frac{1}{C} \sum_{c=1}^{C} \text{AP}_c. \tag{6}$$

3.5 Evaluation Protocols

For detection, YOLOv12n was evaluated on a fixed random subset of the KITTI 2D training split (the only split with public labels), with $N = 500$ and seed 42. The COCO-pretrained YOLOv12n was mapped from COCO to Car/Pedestrian/Cyclist for zero-shot cross-domain evaluation. Identical preprocessing (letterbox 640×640 and division by 255), confidence threshold 0.001 (for mAP integration), and per-class NMS (IoU 0.45) were used across frameworks; mAP was calculated with `pycocotools`.

For depth estimation, DepthAnythingV2 ViT-S was evaluated on the NYU Depth V2 test split ($N = 654$). Because the base checkpoint outputs relative depth, we applied per-image least-squares alignment in disparity space (fit to inverse ground truth, then invert), upsampled predictions to the native ground-truth resolution, and applied the Eigen crop, a 10 m cap, and a validity mask identically for all frameworks.

4 Evaluation and Results

Under identical CPU execution (batch size 1; 20 warm-up and 100 measured iterations; mean ± standard deviation), ONNX Runtime achieves the lowest latency on both tasks. At $T = 4$, detection latency is 20.43 ms (ONNX Runtime) versus 30.98 ms (PyTorch) and 46.60 ms (NCNN); depth latency is 284.83 ms versus 409.42 ms and 688.68 ms. The ranking holds at $T = 1$ (Table 1). Because thread counts are equal, this reflects ONNX Runtime's optimization, not a larger thread budget.

ONNX Runtime is also the most memory-efficient framework for both models (177.6 and 423.1 MB); higher PyTorch resident set size mainly reflects PyTorch and Ultralytics runtime overhead.

All frameworks gain performance up to approximately four to eight threads and then plateau (Table 2). Unconstrained multi-threading does not help: unconstrained ONNX Runtime (approximately 20 threads and 1998% CPU) is slightly slower (16.55 ms) than its pinned eight-thread run (14.92 ms and approximately 800% CPU), and unconstrained PyTorch is far slower than its four-thread run. This is CPU oversubscription with no benefit, worsened by threads spilling onto E-cores, and it explains the previously reported approximately 1900% CPU utilization.

Table 1. Inference latency (mean ± standard deviation, ms), throughput, and memory at identical thread counts. CPU utilization was approximately $T \times 100\%$ for all frameworks by construction (approximately 400% at $T = 4$; NCNN depth reached approximately 364%, not fully saturating four cores).

Task	Framework	Latency $T = 1$	Latency $T = 4$	FPS ($T = 4$)	RAM (MB)
Detection (YOLOv12n)	PyTorch	76.58 ± 1.71	30.98 ± 1.80	32.28	732.8
	ONNX Runtime	62.04 ± 0.36	20.43 ± 0.29	48.95	177.6
	NCNN	88.34 ± 2.90	46.60 ± 0.73	21.46	218.2
Depth (DA-V2 ViT-S)	PyTorch	1151.91 ± 5.33	409.42 ± 19.10	2.44	849.0
	ONNX Runtime	865.33 ± 0.80	284.83 ± 1.57	3.51	423.1
	NCNN	1164.89 ± 76.88	688.68 ± 4.38	1.45	785.4

Table 2. YOLOv12n detection latency (ms) versus thread count (mean ± standard deviation). Unconstrained runs use approximately 20 threads (approximately 1998% CPU) yet do not outperform the pinned eight-thread runs, illustrating oversubscription with no latency benefit.

Framework	$T = 1$	$T = 2$	$T = 4$	$T = 8$	Unconstrained
PyTorch	76.58	52.58	30.98	21.75	39.68 (1999% CPU)
ONNX Runtime	62.04	34.13	20.43	14.92	16.55 (1998% CPU)
NCNN	88.34	61.10	46.60	43.36	42.77 (799% CPU)

Table 3. Object-detection accuracy for YOLOv12n on KITTI under zero-shot evaluation ($N = 500$)

Framework	mAP@0.5	mAP@0.5:0.95	AP Car	AP Pedestrian	AP Cyclist
PyTorch	38.40%	19.55%	38.07%	20.43%	0.15%
ONNX Runtime	38.40%	19.55%	38.07%	20.43%	0.15%
NCNN	38.40%	19.55%	38.07%	20.43%	0.15%

On KITTI ($N = 500$, zero-shot COCO-to-KITTI, so absolute mAP is modest by construction), all three frameworks are identical down to detection counts (Table 3 and Fig. 2), confirming detection fidelity for the convolution-dominant model. Low AP for Cyclist (0.15%) reflects the class mapping, identically across frameworks, rather than a framework effect.

On NYU Depth V2 ($N = 654$), with disparity-space alignment, Eigen crop, and a 10 m cap, PyTorch and ONNX Runtime are numerically identical (AbsRel 0.0619, δ_1 95.47%), confirming clean ONNX conversion (Figs. 3 and 4). NCNN degrades substantially (AbsRel 0.2901, δ_1 46.95%; Table 4). Comparing intermediate ONNX and NCNN activations along the backbone (with inputs verified identical to float32 precision), the input projection and first layer normalization match ONNX exactly (range ratio approximately 1.00); divergence appears abruptly inside the first block's self-attention sub-layer (post-attention range approximately 11 times that of ONNX, before any GELU/MLP) and accumulates through the residual stream. We therefore attribute the loss to ONNX-to-NCNN conversion of the attention sub-layer (candidates include QKV projection, per-head reshape/permute, and softmax) rather than to measurement; isolating the exact operator and testing alternative converters (e.g., PNNX) are left for future work.

5 Discussion

The choice of inference framework has a clear impact on CPU deployment performance. Under a controlled and identical thread configuration, ONNX Runtime achieves the lowest latency and highest throughput for both YOLOv12n and DepthAnythingV2 ViT-S, and it is also the most memory-efficient framework for both. Because thread counts are held equal, these gains are attributable to ONNX Runtime's graph optimization and operator fusion rather than to a larger thread budget—an important distinction,

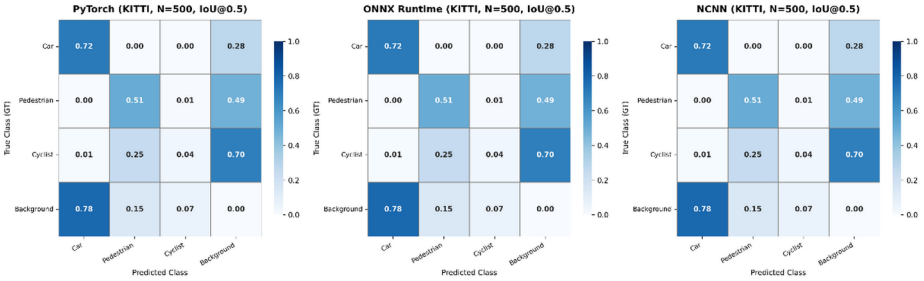


Fig. 2. Normalized confusion matrices for YOLOv12n on KITTI ($N = 500$, IoU@0.5, confidence ≥ 0.25) for PyTorch, ONNX Runtime, and NCNN. The three matrices are identical, confirming that ONNX export and NCNN conversion preserve detection behavior.

Table 4. Monocular depth accuracy on NYU Depth V2 ($N = 654$)

Framework	AbsRel	RMSE	δ_1	δ_2	δ_3
PyTorch	0.0619	0.2990	95.47%	98.79%	99.59%
ONNX Runtime	0.0619	0.2990	95.47%	98.79%	99.59%
NCNN	0.2901	1.0101	46.95%	77.97%	92.42%

since uncontrolled comparisons can otherwise conflate framework quality with thread allocation.

The thread-scaling analysis adds a practical caveat: beyond approximately four to eight threads, returns diminish, and unconstrained multi-threading can degrade latency on a hybrid P/E-core CPU. Practitioners deploying on such CPUs should cap thread counts and pin execution to performance cores rather than allow unrestricted threading.

Architecture also shapes deployment behavior. YOLOv12n, a convolution- dominant model, achieves far lower latency than the full-ViT DepthAnythingV2 across all frameworks, consistent with the higher matrix and memory-bandwidth demands of self-attention. More importantly, the two architectures differ in conversion robustness. Detection accuracy is preserved perfectly across all three frameworks, but NCNN loses substantial depth accuracy on the ViT model. Our operator-level analysis attributes this to conversion of the attention sub-layer rather than to a measurement artifact or to attention in general—the lightweight attention in the detector converts without issue. We therefore frame this as a conversion-stage fidelity limitation localized to full-ViT attention, not as evidence that NCNN is inherently unable to execute transformer models; establishing whether alternative conversion pipelines resolve it is future work.

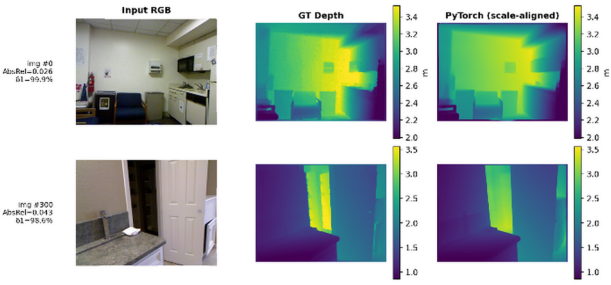


Fig. 3. Qualitative monocular depth results of DepthAnythingV2 ViT-S (PyTorch) on NYU Depth V2, with per-image scale-shift alignment. Predicted maps closely match ground truth across representative indoor scenes.

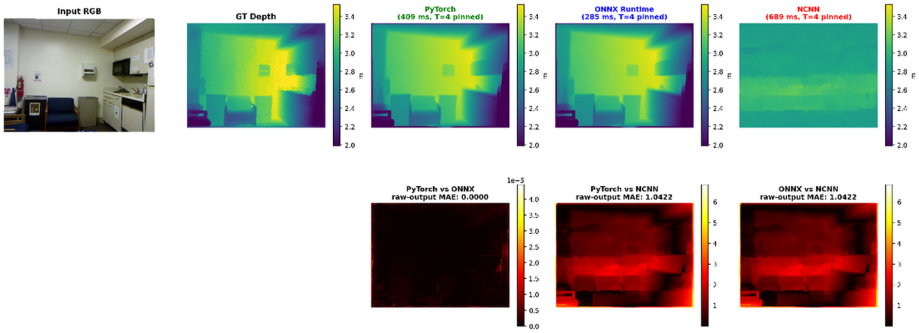


Fig. 4. Cross-framework depth comparison on a representative NYU scene (latencies at $T = 4$, pinned). PyTorch and ONNX Runtime are numerically identical (raw-output MAE approximately zero), whereas NCNN shows the degradation quantified in Table 4.

6 Conclusion

This study presented a controlled, deployment-oriented benchmarking analysis of PyTorch, ONNX Runtime, and NCNN using YOLOv12n and DepthAnythingV2 ViT-S on a CPU platform. By fixing the CPU thread count and core affinity across frameworks and reporting results as mean ± standard deviation, the comparison isolates framework behavior from thread-allocation effects that confounded prior work. ONNX Runtime consistently achieved the lowest latency and highest throughput for both models and was also the most memory-efficient, while a thread-scaling analysis showed that unconstrained multi-threading yields no latency benefit and can cause CPU oversubscription. The results further show that conversion robustness depends on architecture: object-detection accuracy was preserved identically across all three frameworks, whereas NCNN exhibited a substantial depth-accuracy drop on the ViT model, which an operator-level analysis localized to conversion of the first attention sub-layer rather than to measurement. ONNX Runtime is thus the most suitable framework for CPU-

based computer-vision deployment requiring high inference performance, while NCNN remains attractive for lightweight convolutional deployment. Future work will extend the controlled protocol to mobile and edge hardware, quantized and GPU execution, alternative ONNX-to-NCNN conversion tooling, and additional architectures.

Acknowledgments. This research was supported by the Indonesia Endowment Fund for Education Agency (LPDP), Ministry of Finance of the Republic of Indonesia (0029933/FOR/M/PARS-2024). Additional support was provided by the Faculty of Computer Science, Universitas Brawijaya, Malang, Indonesia, through the organization and provision of this conference. The authors gratefully acknowledge the support and opportunity to present and disseminate this research.

References

1. Dosovitskiy, A., et al.: An image is worth 16x16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (2021), https://arxiv.org/abs/2010.11929
2. Liu, Z., et al.: Swin transformer: Hierarchical vision transformer using shifted windows. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (2021), https://arxiv.org/abs/2103.14030
3. Mehta, S., Rastegari, M.: MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer. In: International Conference on Learning Representations (2022), https://arxiv.org/abs/2110.02178
4. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 779–788 (2016), https://arxiv.org/abs/1506.02640
5. Tian, Y., Ye, Q., Doermann, D.: YOLOv12: Attention-centric real-time object detectors. arXiv preprint arXiv:2502.12524 (2025), https://arxiv.org/abs/2502.12524
6. Yang, L., et al.: Depth anything v2. arXiv preprint arXiv:2406.09414 (2024), https://arxiv.org/abs/2406.09414
7. Zhai, H., Du, J., Ai, Y., Hu, T.: Edge deployment of deep networks for visual detection: A review. IEEE Sensors Journal **25**(2), 1–18 (2024), https://ieeexplore.ieee.org/document/10786287
8. Microsoft: ONNX Runtime performance tuning. Online documentation (2024), https://onnxruntime.ai/docs/performance/, accessed 22 September 2026
9. Tencent: NCNN: A high-performance neural network inference framework optimized for mobile, embedded, and desktop platforms. GitHub repository (2026), https://github.com/Tencent/ncnn, accessed 22 September 2026
10. Liu, X., Wei, W., Liu, C., Peng, Y.: Real-time monocular depth estimation merging vision transformers on edge devices for AIoT. IEEE Transactions on Consumer Electronics **69**(4), 1–10 (2023), https://ieeexplore.ieee.org/document/10091191/
11. Pan, J., et al.: EdgeViTs: Competing light-weight CNNs on mobile devices with vision transformers. In: European Conference on Computer Vision (2022), https://arxiv.org/abs/2205.03436
12. Lamaakal, I., Elgarrab, I., Alouach, A., Maleh, Y.: A vehicular-edge federated, quantized YOLOv12 system for real-time 3d hand-gestures-based AAV control. IEEE Access **13**, 1–15 (2025), https://ieeexplore.ieee.org/document/11313047

13. Howard, A., et al.: Searching for MobileNetV3. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (2019), https://arxiv.org/abs/1905.02244
14. Banbury, C., et al.: MLPerf Tiny Benchmark. In: Advances in Neural Information Processing Systems. vol. 34 (2021), https://proceedings.mlr.press/v139/banbury21a.html

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

