Characterizing Why TensorRT 10.7 Slows Down an NMS-Free Edge Detector on Jetson Orin Nano

Ivan Lok (UG Year 3 CS)

Abstract

We profile YOLO26n across two TensorRT stacks (10.3 and 10.7), two precisions (FP16 and INT8), and three output routes on a Jetson Orin Nano. While INT8 with compact NMS-free output gives the fastest measured pipeline (7.64 ms), TensorRT 10.7 is consistently slower than 10.3, with the largest regression (+1.44 ms) in the FP16 NMS-free compact route. Nsight Systems and Nsight Compute analysis shows this regression is not a single defective kernel. Instead, TRT 10.7 shifts to larger xmma tiles, launches the dominant tensor-core kernel $6\times$ more often across 49% more code variants, and emits a less efficient pointwise kernel with local-memory traffic absent from the TRT 10.3 build. We recommend pairing INT8 with compact output for latency-constrained deployment.

1 Introduction

End-to-end deployment latency, not just model inference time, determines whether a detector is usable in an edge application. In batch-one embedded deployment, pipeline stages such as preprocessing, output decoding, candidate filtering, memory movement, and runtime orchestration can rival the TensorRT engine execution in cost. Recent NMS-free detectors such as YOLOv10, RT-DETR, and YOLO26 remove the traditional non-maximum suppression post-processing step, suggesting that compact output heads can reduce deployment overhead [Wang et al.(2024)Wang, Chen, Liu, Chen, Lin, Han, and Ding, Zhao et al.(2023)Zhao, Lv, Xu, Wei, Wang, Dang, Liu, and Chen, Ultralytics(2026)]. However, an algorithm-level NMS-free design does not determine the deployed output contract: a TensorRT engine may still expose a dense candidate tensor (e.g. $[1, 84, 8400]$) that requires application-side filtering and formatting, or it may expose a compact output (e.g. $[1, 300, 6]$) where selection happens inside the engine.

This report studies that interaction for YOLO26n on a Jetson Orin Nano Super Developer Kit, a representative embedded GPU platform. We compare two TensorRT stacks: the JetPack-default 10.3 and the newer 10.7, since TensorRT version affects engine construction, output behavior, and runtime scheduling [NVIDIA(2025), NVIDIA(2024)]. We classify pipelines into three routes (Table 1) and evaluate FP16 and INT8 configurations across both stacks. Profiling combines end-to-end runtime measurement, Nsight Systems stage and CUDA activity analysis, and targeted Nsight Compute kernel investigation.

Table 1: Deployment route taxonomy.

Route	Engine output	Where selection happens	Representative
A	Dense $[1, 84, 8400]$	Application-side	Conventional dense output
B	Compact via engine NMS	TensorRT engine	Traditional NMS in engine
C	Compact NMS-free	YOLO26n e2e in engine	End-to-end selection

Preprint.

2 Profiling Setup And Result

2.1 Pipeline Setup

All experiments use batch-1 inference at 640×640 on a Jetson Orin Nano Super Developer Kit (MAXN power mode, fan at full speed). We compare two TensorRT stacks: the JetPack-default 10.3 (PyTorch 2.5.0a0, containerized) and 10.7 (PyTorch 2.10.0, host). All pipelines use Ultralytics 8.4.39 with COCO val2017 as the common input set. Runtime values are five-run averages over all 5,000 images; accuracy reports COCO P/R/mAP50/mAP50–95. Nsight Systems traces provide CUDA activity, memcpy, and kernel-family evidence. Nsight Compute is used only for selected P1/P3 kernels where NSYS already identified a regression.

2.2 Experimental Matrix

The full experimental matrix is listed in the Appendix (Table 14).

The matrix separates three effects: precision, output contract, and TensorRT version. P5/P4 test compact vs. dense under matched INT8; P3/P1 test the NMS-free route across TRT versions; P14/P15 and P6/P7 provide YOLOv8n dense controls. P11 and P13 are negative evidence: INT8 engine-side NMS failed to build on both stacks.

3 Profiling Observations

We analyze runtime results at the pipeline level before individual CUDA kernels. The matrix reveals two patterns: some configurations improve raw engine time but not end-to-end latency, while others trade slightly higher engine cost for substantially lower post-processing overhead.

3.1 NMS-Free Compact Output Is Fastest Under A Controlled Route Comparison

The clearest route-level comparison is obtained by holding the model, precision, and TensorRT version fixed. For YOLO26n FP16 under TensorRT 10.3, the measured end-to-end order is Route C (NMS-free compact, P1), then Route B (engine-side NMS, P12), then Route A (dense, P9). We use TensorRT 10.3 for this claim because the TensorRT 10.7 FP16 compact case P3 has a separate engine regression (Section 4) that would otherwise mix route effects with compiler/runtime effects.

Table 2: Controlled route comparison for YOLO26n FP16 under TensorRT 10.3. Runtime values are five-run averages from non-profiled runs.

Route	Pipeline	Infer	Post	Total	Main interpretation
C: NMS-free compact	P1	4.487	1.100	7.800	compact without traditional NMS
B: engine-side NMS	P12	5.436	1.100	8.720	compact, but heavier engine
A: dense output	P9	4.438	2.860	9.520	fast engine, heavy postprocess

P9 has almost the same engine time as P1, but is 1.720 ms slower end-to-end because dense output leaves candidate filtering, selection, sorting, NMS, and formatting after inference. P12 removes this application-side cost by returning compact detections, but its engine time rises by 0.949 ms because traditional NMS and dynamic selection are moved into the engine. P1 keeps the compact-output benefit without paying the same engine-side NMS cost. Route C therefore avoids the expensive parts of both alternatives: it does not expose a dense candidate tensor to application-side post-processing, and it does not rely on a traditional engine-side NMS plugin.

A brief NSYS comparison confirms the mechanism. Table 3 summarizes the kernel- and memcpy-level differences normalized per image from the profiled COCO run.

3.2 INT8 Engine Gains Are Complementary To Compact Output

When the model, TensorRT version, and output route are matched, INT8 consistently reduces engine inference time. The matched pairs in Table 4 confirm this across two models and two TensorRT stacks.

Table 3: Pairwise NSYS evidence for the controlled Route C/B/A comparison. Per-image values normalized from 5,000 profiled images.

Pair	Extra work observed (NSYS)	Interpretation
P9 vs. P1	+23.8 kernel launches/img, +6.9 memcpy calls/img, CUB select/reduce, sort, NMS	Dense output leaves a candidate-compaction pipeline after inference
P12 vs. P1	`NonMaxSuppression`, `NonZero`, Myelin Top, radix sort, extra memcpy	Engine-side NMS moves dynamic selection into the engine

Table 4: Matched FP16-to-INT8 comparisons. Values are five-run runtime averages.

Matched route	FP16 infer / total	INT8 infer / total	Total gain
YOLO26n TRT10.7 Route A	P8: 4.459 / 10.060	P4: 3.819 / 9.300	0.760
YOLOv8n TRT10.3 Route A	P6: 4.194 / 9.260	P7: 3.320 / 8.340	0.920
YOLOv8n TRT10.7 Route A	P14: 4.171 / 9.620	P15: 3.408 / 8.800	0.820

However, the best deployment is not obtained by precision alone. P4 has the fastest YOLO26n INT8 dense engine in our matrix, but P5 is 1.660 ms faster end-to-end because it pairs INT8 with compact NMS-free output. INT8 and compact output are therefore complementary: INT8 reduces engine-stage cost, while the output contract controls how much post-processing work survives the engine.

3.3 TensorRT 10.7 Slowdown Is Route-Dependent

The TensorRT version comparison shows a different pattern from the route and precision results. TensorRT 10.7 is slower end-to-end in all matched pairs that we tested, but the source of the slowdown is route-dependent. In the FP16 NMS-free compact route, P3 is slower than P1 by 1.154 ms in inference and 1.440 ms in total latency. This is the only matched pair where the inference stage itself accounts for most of the regression. In contrast, the dense and engine-side NMS routes have almost unchanged engine time: P4 versus P2 differs by only 0.005 ms in inference, P8 versus P9 by 0.021 ms, and P10 versus P12 by 0.049 ms. Their total latency still increases by 0.400–0.540 ms, indicating that the overhead is mostly outside raw TensorRT compute.

Table 5: Matched TensorRT 10.3 to 10.7 comparison. Runtime values are five-run averages.

Route	Pair	Infer delta	Total delta	Main reading
C FP16 compact	P3 - P1	+1.154	+1.440	engine-stage regression
A INT8 dense	P4 - P2	+0.005	+0.500	outside raw engine compute
A FP16 dense	P8 - P9	+0.021	+0.540	outside raw engine compute
B FP16 engine NMS	P10 - P12	+0.049	+0.400	mostly runtime/boundary overhead
A YOLOv8n FP16 dense	P14 - P6	-0.023	+0.360	engine not slower
A YOLOv8n INT8 dense	P15 - P7	+0.088	+0.460	small engine change

The NSYS summaries support this split. P3 has a much larger CUDA workload than P1, including higher kernel total time and about 3.9 GiB of additional device-to-device copies over the profiled run. The other matched pairs do not show a comparable kernel-total or memcpy-volume jump. For example, P4 and P2 have nearly identical memcpy volume, and P8 and P9 have similar engine compute and memcpy volume. Therefore, the TensorRT 10.7 result should be interpreted as two effects: a real engine regression for the FP16 NMS-free compact route, and smaller full-pipeline overheads for the dense and engine-NMS routes. Section 4 therefore focuses the deeper kernel analysis on P3 versus P1 rather than treating every TensorRT 10.7 slowdown as the same hardware bottleneck.

4 Further Investigation

Section 3.3 showed that the TensorRT 10.7 slowdown is not uniform across all routes. Most matched pairs have similar raw inference time and lose latency around the pipeline boundary, but P3/P1 is different: the same YOLO26n FP16 NMS-free compact route becomes 1.154 ms slower inside TensorRT inference. This section therefore treats P3/P1 as a focused engine-regression case rather than as another dense-output post-processing problem.

The investigation follows the profiling stack used throughout this report: Nsight Systems identifies which kernel families grew in P3 (Section 4.1), and Nsight Compute inspects representative launches from each family to determine whether the extra time comes from slower individual kernels or from more kernels being scheduled (Sections 4.2–4.4). The central question is whether the P3 regression is caused by one defective kernel or by a broader graph-level change in how TRT 10.7 schedules, tiles, and lays out the compact-route engine.

4.1 P1/P3 Pair: Kernel-Family Suspect Identification

P1 and P3 are a controlled pair for isolating the TensorRT-version effect on Route C. Both use YOLO26n, FP16 precision, and the NMS-free compact output contract. The difference is the deployment stack: P1 uses TensorRT 10.3, while P3 uses TensorRT 10.7. Since P3 is slower mainly in the inference stage, the first question is not whether post-processing is expensive, but what extra GPU-side work appears inside or around the TensorRT engine.

To answer this, we group CUDA kernel activities from the P1 and P3 full-COCO Nsight Systems traces into recurring kernel families. This converts the broad observation "P3 is slower" into a concrete suspect list that can be inspected with Nsight Compute. Table 6 lists the families and aggregate deltas over 5,000 COCO images.

Table 6: Kernel-family time totals from P1 and P3 NSYS traces (full COCO val2017, 5,000 images). Families are identified by CUDA kernel name patterns queried from Nsight Systems SQLite exports.

Kernel family (NSYS name pattern)	P1	P3	Δ
xmma (%xmma%)	11,431.7 ms	15,651.5 ms	+4,219.8 ms
generatedNativePointwise	1,438.5 ms	2,604.5 ms	+1,166.0 ms
CUTENSOR permutation	2,281.0 ms	2,829.6 ms	+548.6 ms
Kernel total	21,543.2 ms	27,148.2 ms	+5,605.0 ms
D2D memcpy	0 MiB / 4 calls	3,907 MiB / 5,008 calls	+3,907 MiB

The table identifies three kernel families with large NSYS deltas, plus a striking D2D memcpy signal. The largest increases are in xmma convolution/GEMM (+4,220 ms), generated pointwise helper kernels (+1,166 ms), and CUTENSOR permutation work (+549 ms). P3 also records 3,907 MiB of device-to-device memcpy traffic (5,008 calls, 140.7 ms) that is essentially absent from P1 (4 calls, $\approx$0 MiB). This D2D signal is investigated together with the CUTENSOR family in Section 4.4, where we show it is part of a broader layout-strategy shift rather than net additional overhead.

These three families define the scope for the Nsight Compute investigation in Sections 4.2–4.4.

4.2 Tensor Core Convolution and GEMM Operations

The xmma family of Tensor Core convolution and GEMM kernels contributes the largest absolute increase in NSYS: +4,220 ms over the full COCO run (Table 6). We first decomposed this family by kernel name and grid shape to understand whether the extra time comes from slower individual launches, from more launches, or from a different tiling strategy. P1 uses 37 distinct xmma kernel name variants across 500,100 launches. P3 uses 55 variants across 485,097 launches—49% more code-path diversity at a similar total launch count. Fourteen kernel names appear in both pipelines, providing an opportunity for a matched per-launch comparison.

We identified one kernel present with the same grid and block shape in both engines: `tilesize256×128×64_stage3` with grid $(1, 7, 1)$, block $(256, 1, 1)$, and 208 registers per thread. Table 7 reports the Nsight Compute comparison for this matched launch.

The per-launch comparison gives a clear answer: P3's matched kernel is 34.5% faster per launch, driven by a 21.6 percentage-point higher L2 hit rate (89.2% vs. 67.6%). Individual xmma kernel efficiency has improved, not degraded. However, P3 launches this same kernel 30,006 times over the COCO run versus 5,001 in P1—a factor of six—so the per-launch speed advantage is overwhelmed by launch-count multiplication at the aggregate level.

The launch-count increase is part of a broader tiling-strategy divergence between the two TensorRT versions. Table 8 compares the dominant kernel configurations.

Table 7: Matched xmma kernel NCU comparison. Same kernel name, same grid $(1, 7, 1)$, same block $(256, 1, 1)$, same 208 registers per thread. Values are means of 2 (P1) and 6 (P3) captured launches.

Metric	P1 (TRT 10.3)	P3 (TRT 10.7)	Delta
Duration	62.8 us	41.1 us	P3 34.5% faster
Elapsed cycles	62,942	41,312	−34.4%
L2 hit rate	67.6%	89.2%	+21.6 pp
Compute (SM) throughput	53.9%	40.9%	−13.0 pp
Memory throughput	44.7%	34.5%	−10.2 pp
Achieved occupancy	17.4%	17.3%	same
NSYS launch count	5,001	30,006	6× more
NSYS total time	308.7 ms	1,099.5 ms	+790.8 ms

Table 8: Tiling strategy comparison between P1 and P3 xmma families.

Property	P1 (TRT 10.3)	P3 (TRT 10.7)
Dominant tile size	$128 \times 32 \times 32$	$256 \times 128 \times 64$
Block shape	$(128, 1, 1)$	$(256, 1, 1)$
Registers per thread	110	208
Theoretical occupancy	33.3%	16.7%
Kernel name variants	37	55

TRT 10.7 shifts toward larger tiles with double the register footprint. The regs=208 kernel is register-limited at 16.7% theoretical occupancy, whereas the regs=110 kernel used by P1 can reach 33.3%. P3 trades occupancy for bigger per-block tiles. The NCU data confirm that the per-launch consequence of this tradeoff is positive—better L2 cache behavior and shorter duration—but the aggregate consequence is 49% more kernel variants and a 6× launch-count increase on the single most time-consuming variant. The xmma growth is therefore a scheduling and tactic-selection problem: individual launches are efficient, but TRT 10.7 launches more of them from a more diverse set of code paths.

4.3 Generated Pointwise Helper Kernels

The generatedNativePointwise family grows from 1,438.5 ms in P1 to 2,604.5 ms in P3 (+1,166.0 ms). Unlike the xmma family, where the per-launch efficiency actually improved, pointwise helper kernels represent a diffuse class of elementwise operations that TensorRT's code generator emits as graph glue around fused regions. A raw family total can be misleading: two pointwise kernels with the same CUDA name may map to different TensorRT layers in different engines. To obtain a clean per-launch comparison, we traced the CUPTI `correlationId` path from each CUDA kernel launch, through its `cuLaunchKernel` call, to the owning TensorRT NVTX range. This mapping identifies the engine layer that issued the kernel.

The selected P1 and P3 pointwise launches both map to `PWN(PWN(/model.0/act/Sigmoid), PWN(/model.0/act/Mul))`, the `Sigmoid+Mul` activation immediately after the first convolution. This is an early-backbone layer, not a detection-head, NMS, or Python post-processing artifact. The launch shapes alone reveal that TRT 10.7 emits a structurally different kernel for the same semantic position (Table 9). P1 uses a large grid of 1,600 blocks with only 27 registers per thread, exposing substantial parallelism. P3 uses a grid of only 200 blocks with 80 registers per thread, trading parallel work for higher per-thread resource usage.

Table 9: Layer-aligned generatedNativePointwise launch shape for `/model.0/act`.

Pipeline	Grid / block	Regs/thread	Calls	Avg duration
P1	g=$(1600, 1, 1)$, b=$(128, 1, 1)$	27	5,001	74.5 us
P3	g=$(200, 1, 1)$, b=$(128, 1, 1)$	80	5,001	118.5 us

Nsight Compute metric-probe data confirm this structural difference (Table 10). P3's launch is 1.49× slower (113.8 us vs. 76.4 us) despite using 8× fewer blocks and threads. The global input and output byte counts are nearly identical (∼3.27 MiB each), ruling out increased global data movement as the cause. The slowdown instead appears in resource utilization: P3 has 3× more registers per thread (80 vs. 27), less than half the active-warp coverage (42.4% vs. 94.1%), substantially lower SM throughput (17.3% vs. 55.8%), and a much lower instructions-per-cycle rate (0.74 vs. 2.29).

Table 10: Layer-aligned pointwise NCU metric-probe comparison for `/model.0/act`.

Metric	P1	P3	Reading
Duration	76.4 us	113.8 us	1.49× slower
Grid size	1,600	200	less parallel work
Threads	204,800	25,600	8× fewer
Registers/thread	27	80	3× higher
Active warps	94.1%	42.4%	lower coverage
SM throughput	55.8%	17.3%	weaker utilization
IPC active	2.289	0.737	lower rate
L1 global load	3.277	3.277 MiB	same
L1 global store	3.266	3.277 MiB	same
Local load reqs	0	44,800	P3 only
Local store reqs	0	25,600	P3 only
L2 local load	0	350.8 KiB	P3 only
L2 local store	0	1.878 MiB	P3 only

The most diagnostic new signal is local memory. P3's launch has 44,800 local load requests and 25,600 local store requests with corresponding L2-local traffic, whereas P1's launch has none. Local memory in CUDA is backed by the same on-chip storage as the L1 cache; its appearance is consistent with register spilling or a compiler-generated indexing path that requires scratchpad storage. Combined with the 3× higher register count and halved occupancy, the evidence points to a TRT 10.7 code-generation decision that produces a less efficient pointwise kernel for this layer.

The generatedNativePointwise family is the only suspect among the three where Nsight Compute shows a clear per-launch efficiency loss. It does not explain the entire P3 regression—xmma scheduling and the layout-strategy shift also contribute—but it provides a concrete, layer-aligned example of a TRT 10.7 kernel that is individually worse than its TRT 10.3 counterpart.

4.4 Layout Movement: D2D Copies and CUTENSOR Permutation

P3's NSYS trace contains 5,008 device-to-device memcpy calls totaling 3,907 MiB, whereas P1 has essentially none (4 calls, ≈0 MiB). Taken at face value, this is the single largest anomaly in the memory subsystem. The CUTENSOR permutation family also grows (+548.6 ms). Together, they suggest that P3 may be paying for additional tensor layout conversion—but the D2D and CUTENSOR increases must be read alongside `genericReformat`, the third component of the layout-management subsystem. Table 11 reports the full composition.

Table 11: P1 vs. P3 layout-management composition (NSYS, full COCO val2017).

Component	P1	P3	Delta
D2D memcpy	0.0 ms / 0 MiB / 4 calls	140.7 ms / 3,907 MiB / 5,008 calls	+3,907 MiB
CUTENSOR permutation	2,281.0 ms	2,829.6 ms	+548.6 ms
`genericReformat`	1,807.1 ms	789.0 ms	−1,018.1 ms
Layout total	**4,088.1 ms**	**3,759.3 ms**	**−328.8 ms**

The total layout-management cost (CUTENSOR + genericReformat + D2D) is *lower* in P3 (3,759 ms) than in P1 (4,088 ms). TRT 10.7 substitutes reformat packed-copy kernels (−1,018 ms) with D2D alignment copies (+141 ms) and additional CUTENSOR permutation work (+549 ms). The +3.9 GiB D2D signal is therefore a strategy shift, not net additional overhead.

Timestamp correlation places all 5,001 D2D copies (0.78 MiB each, on stream 7) outside `myelinGraphExecute` NVTX ranges, alongside CUTENSOR permutation kernels, xmma launches, and `genericReformat::copyVectorizedKernel` calls. They function as alignment buffers between engine subgraphs rather than as engine-internal data movement. P1 relies on `copyPackedKernel` (1,526 ms) for the same class of operations; P3 uses `copyVectorizedKernel` with explicit D2D alignment, which is more efficient per byte but requires aligned buffers.

A representative CUTENSOR permutation launch was captured in both pipelines (grid=(1,1600,1), block=(128,1,1), 28 registers). Table 12 confirms that the per-launch efficiency is essentially identical.

The +548 ms CUTENSOR family growth comes from P3 scheduling permutation launches in grid shapes absent from P1 (e.g., grid=(2,50,1) at 318 ms, grid=(4,13,1) at 218 ms), not from individual kernels running slower.

Table 12: Matched CUTENSOR permutation launch (NCU).

Metric	P1	P3
Duration	102.8 us	104.5 us
Memory Throughput	66.8%	65.6%
Compute Throughput	60.7%	62.3%
L2 Hit Rate	40.0%	40.0%
Achieved Occupancy	100.5%	98.3%

The layout-movement suspect therefore follows the same pattern as xmma—more aggregate work, unchanged per-launch efficiency—but with an additional structural finding: the composition of layout operations shifted from packed-copy reformatting toward vectorized reformatting with explicit D2D alignment. The D2D signal should not be treated as a regression source; it reflects a tradeoff within a layout-management subsystem whose total cost is comparable between the two TensorRT versions (3,759 ms vs. 4,088 ms).

4.5 Synthesis

Table 13 consolidates the NCU evidence across the three investigated kernel families.

Table 13: Synthesis of the three-suspect NCU investigation.

Suspect	Per-launch slower?	More launches?	Main mechanism
xmma (Sec. 4.2)	No – P3 34.5% faster	Yes – 6×, 55 variants	Scheduling & tiling divergence
pointwise (Sec. 4.3)	Yes – lower occ., local-memory	TBD	Only confirmed per-kernel regression
CUTENSOR/D2D (Sec. 4.4)	No – matched launch identical	Yes – strategy shift	Net layout cost comparable; D2D replaces reformat

Of the three families that showed large NSYS deltas, only one—generated pointwise helper kernels—exhibits a genuine per-launch efficiency loss. The matched xmma kernel is 34.5% faster in P3, with a substantially higher L2 hit rate, and the matched CUTENSOR permutation launch is identical at the microarchitectural level. The xmma family's aggregate growth is driven by launch-count multiplication (6× on the dominant kernel) and by the use of 49% more kernel name variants, representing a more fragmented tiling strategy. The CUTENSOR/D2D increases reflect a layout-strategy tradeoff: TRT 10.7 uses vectorized reformat kernels with explicit D2D alignment copies instead of packed-copy reformatting, and the net layout-management cost is comparable (3,759 ms vs. 4,088 ms).

The single clear per-kernel regression is the generatedNativePointwise family. A layer-aligned comparison at `/model.0/act` shows that TRT 10.7 emits a higher-register (80 vs. 27), lower-occupancy (42.4% vs. 94.1%) pointwise kernel with local-memory traffic that is entirely absent from the P1 kernel at the same layer. This represents a concrete code-generation difference rather than a scheduling or launch-count artifact.

Taken together, the evidence supports interpreting the P3 regression as a graph-level TRT 10.7 engine schedule change, not a single defective kernel. Two mechanisms account for the majority of the extra inference time: (1) TRT 10.7 schedules more xmma launches—6× more on the dominant kernel—across a wider set of tile-size variants (55 vs. 37), producing additional tensor-core work despite efficient per-launch execution; and (2) TRT 10.7 emits higher-register, lower-occupancy pointwise helper kernels with local-memory activity for at least one early-backbone layer. The layout-management subsystem is not a regression contributor. The strongest remaining open question is which specific TRT 10.7 tactic or layout decision introduces the 6× launch-count multiplication on the dominant xmma kernel.

5 Optimization Opportunities

The profiling evidence from Sections 3 and 4 supports two categories of recommendations: practical deployment choices (which pipeline configuration to deploy and how to structure post-processing) and forward-looking considerations for TensorRT version upgrades and quantization strategy.

5.1 Practical Deployment Recommendations

The fastest measured pipeline in our matrix is P5 (YOLO26n INT8 Route C, 7.64 ms). Its advantage does not come from having the lowest raw engine time—P4 has a faster engine (3.82 ms vs. 4.11 ms). It comes from avoiding the dense post-processing path that costs P4 an additional 1.96 ms in the post-processing stage (3.10 ms vs. 1.14 ms). Section 3.1 established the route ranking under controlled FP16 conditions (Route C $<$ B $<$ A), and Section 3.2 showed that INT8 engine gains are complementary to compact output rather than a substitute for it. The practical deployment recommendation follows directly: when latency is the primary constraint, deploy INT8 with compact NMS-free output (Route C). When accuracy is the primary constraint, use FP16 Route C (P1, 7.80 ms, mAP50–95 0.397), since the INT8 Route C accuracy tradeoff (P5, mAP50–95 0.362) represents a moderate 0.035 mAP50–95 drop.

The failed INT8 engine-NMS builds (P11, P13) provide negative evidence that narrows the practical choice. The apparently attractive Route B + INT8 combination could not be built on either TensorRT 10.3 or 10.7. Engine-side NMS with INT8 is therefore not a currently viable deployment path on this platform with the tested tooling, leaving Route A (dense) or Route C (compact) as the practical options under INT8 quantization.

For deployments where compact output is unavailable (Route A), the NSYS evidence provides a concrete breakdown of where post-processing time goes. P9's full-pipeline trace (Section 3.1) shows $+23.8$ kernel launches per image, $+6.9$ memcpy calls per image, and $+0.257$ MiB D2D per image relative to P1. The post-processing pipeline includes CUB DeviceSelect and DeviceReduce, radix sort for confidence ranking, NMS, batched gather/copy, and result formatting—a chain of small, fragmented GPU operations rather than a single expensive kernel. The optimization direction follows from this breakdown: split post-processing into explicit stages (decode $\rightarrow$ score filtering $\rightarrow$ NMS-or-selection $\rightarrow$ formatting) so each stage can be profiled individually; reduce the candidate count before expensive sort and NMS operations; avoid unnecessary tensor layout conversions between stages; and batch or fuse small filtering kernels where possible.

5.2 TensorRT Version and Quantization Considerations

The TRT 10.7 vs. 10.3 comparison carries two forward-looking lessons for deployment engineers who need to evaluate TensorRT version upgrades.

First, the xmma tiling-strategy divergence (Section 4.2) demonstrates that a TensorRT version upgrade can change the compiler's tile-size selection without warning. TRT 10.7 shifted from small tiles ($128\times32\times32$, 110 registers, 33% theoretical occupancy) to large tiles ($256\times128\times64$, 208 registers, 17% theoretical occupancy). Per-launch efficiency improved—the matched kernel was 34.5% faster with an 89.2% L2 hit rate—but the scheduler launched $6\times$ more instances of the dominant kernel across 49% more code variants. A practical takeaway is that per-kernel NCU efficiency is not sufficient to predict end-to-end latency: the scheduler's launch-count and variant-count decisions matter at least as much as individual kernel quality. Validating a TRT upgrade with end-to-end profiling, not only `trtexec` engine benchmarking, is essential.

Second, the pointwise code-generation regression at `/model.0/act` (Section 4.3) is a compiler-level issue: TRT 10.7 emitted a higher-register (80 vs. 27), lower-occupancy (42.4% vs. 94.1%) pointwise kernel with local-memory traffic that was entirely absent from the TRT 10.3 kernel at the same layer. This is not a hardware limitation—the same GPU runs both kernels—but a generated-code difference. When evaluating a TRT version upgrade for an edge deployment, spot-checking a few representative pointwise or elementwise kernels with NCU before committing to the new version can catch this class of regression early, without requiring a full COCO-scale profiling run.

References

[NVIDIA(2024)] NVIDIA. Tensorrt 10.7.0 release notes. `https://docs.nvidia.com/deeplearning/tensorrt/latest/getting-started/release-notes-10/10.7.0.html`, 2024. Accessed: 2026-05-08.

[NVIDIA(2025)] NVIDIA. Nvidia jetpack sdk 6.2. `https://developer.nvidia.com/embedded/jetpack-sdk-62`, 2025. Accessed: 2026-05-08.

[Ultralytics(2026)] Ultralytics. Ultralytics yolo26 documentation. https://docs.ultralytics.com/models/yolo26/, 2026. Accessed: 2026-05-08.

[Wang et al.(2024)Wang, Chen, Liu, Chen, Lin, Han, and Ding] Ao Wang, Hui Chen, Lihao Liu, Kai Chen, Zijia Lin, Jungong Han, and Guiguang Ding. Yolov10: Real-time end-to-end object detection. *arXiv preprint arXiv:2405.14458*, 2024.

[Zhao et al.(2023)Zhao, Lv, Xu, Wei, Wang, Dang, Liu, and Chen] Yian Zhao, Wenyu Lv, Shangliang Xu, Jinman Wei, Guanzhong Wang, Qingqing Dang, Yi Liu, and Jie Chen. Detrs beat yolos on real-time object detection. *arXiv preprint arXiv:2304.08069*, 2023.

Appendix

Table 14: Summary of measured YOLO inference pipelines. Runtime columns are five-run averages over 5,000 COCO images. mAP50–95 is the COCO metric averaged over IoU thresholds 0.50–0.95. P11 and P13 are negative evidence (INT8 engine-side NMS failed on both TRT stacks).

ID	Model	TRT	Precision	Route	Output	Pre ms	Infer ms	Post ms	Total ms	mAP50	mAP50–95	Role
P1	YOLO26n	10.3	FP16	C	compact	2.200	4.487	1.100	7.800	0.551	0.397	FP16 compact baseline
P2	YOLO26n	10.3	INT8	A	dense	2.200	3.814	2.800	8.800	0.538	0.373	TRT10.3 dense INT8 control
P3	YOLO26n	10.7	FP16	C	compact	2.420	5.641	1.160	9.240	0.551	0.397	FP16 compact regression case
P4	YOLO26n	10.7	INT8	A	dense	2.400	3.819	3.100	9.300	0.538	0.373	Fast dense engine, high postprocess
P5	YOLO26n	10.7	INT8	C	compact	2.400	4.113	1.140	7.640	0.517	0.362	Fastest measured total latency
P6	YOLOv8n	10.3	FP16	A	dense	2.200	4.194	2.860	9.260	0.518	0.368	Conventional FP16 dense control
P7	YOLOv8n	10.3	INT8	A	dense	2.200	3.320	2.840	8.340	0.503	0.354	Conventional INT8 dense control
P8	YOLO26n	10.7	FP16	A	dense	2.400	4.459	3.180	10.060	0.562	0.403	TRT10.7 YOLO26n dense FP16
P9	YOLO26n	10.3	FP16	A	dense	2.200	4.438	2.860	9.520	0.562	0.403	TRT10.3 YOLO26n dense FP16
P10	YOLO26n	10.7	FP16	B	compact	2.420	5.485	1.200	9.120	0.439	0.333	TRT10.7 engine-side NMS
P12	YOLO26n	10.3	FP16	B	compact	2.200	5.436	1.100	8.720	0.439	0.332	TRT10.3 engine-side NMS
P14	YOLOv8n	10.7	FP16	A	dense	2.400	4.171	3.020	9.620	0.518	0.368	TRT10.7 YOLOv8n FP16 control
P15	YOLOv8n	10.7	INT8	A	dense	2.400	3.408	3.000	8.800	0.497	0.350	TRT10.7 YOLOv8n INT8 control
P11	YOLO26n	10.7	INT8	B attempt	compact	–	–	–	–	–	–	Build failed
P13	YOLO26n	10.3	INT8	B attempt	compact	–	–	–	–	–	–	Build failed