Université Côte d'Azur

Edge Computing et IA embarquée

Optimisation de la détection d'objets embarquée pour système IOT-CPS : Approche YOLOv11

Auteurs :
Noé Florence, Nathan Amoussou, Louis Malmassary

Professeur :
Gerald Rocher

Dépôt GitHub :
`https://github.com/NathanAmoussou/3d_edge_object_detection`

January 22, 2026

Contents

1 Introduction

L'essor de l'informatique embarquée (ville intelligente, robotique, véhicules autonomes, santé connectée, etc.) s'accompagne d'un besoin croissant de déployer des modèles d'intelligence artificielle (IA) au plus proche des capteurs et des actionneurs. Ce déplacement du *cloud* vers l'*edge* est motivé à la fois par des contraintes pratiques (connectivité intermittente, coûts de transfert, confidentialité) et par des exigences techniques (faible latence, robustesse en temps réel), tout en restant limité par les ressources disponibles sur l'embarqué (mémoire, puissance de calcul, énergie). Dans ce contexte, l'optimisation de modèles—via des leviers agissant sur le modèle lui-même et sur l'exploitation du matériel—est devenue une pratique répandue et de plus en plus accessible grâce aux outils et écosystèmes actuels.

Dans cette logique, le projet présenté dans ce rapport consiste à, partant d'un modèle d'IA et d'une sélection de plateformes matérielles (décrites en section 2), appliquer plusieurs couches d'optimisations (décrites en sections 3 et 3.1). Nous mesurons ensuite l'impact de ces choix sur des métriques de performance et de coût en veillant à rendre les comparaisons aussi équitables que possible via un protocole commun (décrit en sections 5 et 6).

L'objectif final de ce projet est d'identifier, pour chaque plateforme, les compromis les plus pertinents et de conclure sur la ou les combinaisons d'optimisations offrant le meilleur équilibre entre efficacité et qualité d'inférence, par exemple (section 7).

2 Cas d'usage choisi

2.1 Architecture à optimiser

Comme architecture d'IA à optimiser, nous avons choisi un modèle de vision par ordinateur dédié à la détection d'objets. Ce type de réseau reçoit en entrée une image (ou une séquence d'images pour une vidéo) et produit en sortie un ensemble de prédictions décrivant où se trouvent les objets d'intérêt et de quel type ils sont : typiquement des boîtes englobantes (*bounding boxes*), associées à une classe et à un score de confiance. Dans la suite, l'ensemble de nos optimisations viseront donc à accélérer et/ou alléger cette chaîne d'inférence tout en contrôlant la dégradation éventuelle de la qualité de détection.

Dans ce cadre, nous choisissons **YOLO11**, une famille de détecteurs d'objets temps réel publiée par l'entreprise américaine Ultralytics en septembre 2024. Cette gamme est conçue pour offrir un bon compromis entre précision, vitesse et efficacité et est explicitement pensée pour être déployée dans des environnements variés (du GPU aux dispositifs *edge*). Ce choix est motivé à la fois par le caractère très représentatif de YOLO pour des cas d'usage *edge*/embarqué, et par l'adéquation avec nos besoins de projet centrés sur la vision.

2.2 Matériel

À la suite du choix de l'architecture IA et afin d'évaluer l'effet des optimisations dans des conditions réalistes, nous testons l'inférence sur un ensemble de plateformes hétérogènes. Notre cible principale étant l'IA *edge*/embarqué, nous décrivons d'abord trois dispositifs représentatifs de familles d'accélération différentes :

- **Raspberry Pi 4 (*Pi*)** — cible CPU-*only* généraliste. Le Pi repose sur un SoC Broadcom BCM2711 intégrant un CPU ARM Cortex-A72 (4 cœurs) et une mémoire LPDDR4 partagée. En l'absence d'accélérateur IA dédié, l'inférence est principalement limitée par le budget de calcul scalaire/SIMD du CPU, ainsi que par la bande passante mémoire et les effets thermiques. Dans la section résultats, cette plateforme sert à analyser les gains des optimisations dans un régime fortement contraint.

- **NVIDIA Jetson Orin Nano 4GB (*Orin*)** — focus GPU-*first*. L'Orin Nano combine un CPU ARM Cortex-A78AE (6 cœurs) et un GPU NVIDIA Ampere (512 cœurs CUDA, 16 Tensor Cores), avec 4 Go de LPDDR5 (64-bit, ∼34 GB/s). Un GPU (*Graphics Processing Unit*) est un processeur massivement parallèle, particulièrement efficace pour les opérations de type produits matriciels, ce qui se traduit en pratique par une accélération notable de l'inférence IA par rapport à un CPU généraliste lorsque le modèle est bien vectorisable. Dans ce projet, nous utilisons explicitement l'Orin comme une plateforme d'inférence accélérée GPU : le modèle est exécuté côté GPU via l'écosystème NVIDIA (CUDA/TensorRT), afin de tirer parti des Tensor Cores. Le CPU est principalement utilisé pour l'orchestration et une partie du pré/post-traitement, ce qui permet d'étudier séparément les goulots d'étranglement GPU et CPU + transferts. Les limites attendues sont la capacité mémoire (4 Go) et la bande passante mémoire lorsque la résolution ou les activations augmentent, ainsi que le coût CPU du pré/post-traitement si l'inférence GPU devient très rapide.

- **Luxonis OAK-D Pro (*Oak*)** — focus VPU-*first*. L'OAK-D Pro est une caméra qui exécute une grande partie du pipeline (traitements de vision, profondeur stéréo et inférence) sur un VPU embarqué (plateforme DepthAI/RVC). Un VPU (*Vision Processing Unit*) est un accélérateur spécialisé pour la vision, conçu pour exécuter efficacement des traitements d'imagerie et des réseaux de neurones à faible consommation, avec un meilleur compromis perf/W qu'un CPU. Cette approche favorise l'efficacité énergétique et réduit la charge du CPU hôte, mais impose des contraintes plus fortes de mémoire embarquée et de compatibilité opérateur (modèles à convertir/adapter). Le goulot d'étranglement peut provenir du VPU (capacité de calcul/mémoire, support d'opérateurs), mais aussi du *dataflow* hôte ↔ caméra (USB). Pour les tests avec la oak-d pro qui vont suivre, la caméra sera pilotée par notre ordinateur, via USB.

En complément (hors cible principale *edge*/embarqué), nous utilisons deux machines de référence comme points de comparaison indicatifs : (i) un PC portable *4070* équipé d'un GPU **NVIDIA GeForce RTX 4070 Laptop** (8 Go de VRAM) et de 32 Go de RAM, et (ii) un PC portable *i9* basé sur un **Intel Core i9-14900HX** (24 cœurs / 32 *threads*, 32 Go de RAM).

Le tableau 1 synthétise les plateformes décrites ci-dessus.

Table 1: Caractéristiques matérielles des plateformes et mise en relation avec les goulots d'étranglement observés.

Plateforme	**Processeur hôte**	**Accélérateur**	**Mémoire**	***Bottleneck***
Pi	ARM Cortex-A72 (4 cœurs)	*Aucun* (CPU-only)	LPDDR4 (Partagée)	SIMD saturé, latence élevée due à l'absence d'accélération dédiée.
Orin	ARM Cortex-A78AE (6 cœurs)	GPU NVIDIA Ampere (512 cœurs, 16 Tensor Cores)	4 Go LPDDR5 (Partagée)	4 Go de mémoires et coût du pré/post-traitement sur CPU.
Oak	Déporté sur Hôte (via USB)	VPU Myriad X	Embarquée (VPU)	Latence de transfert USB hôte-caméra et capacité de calcul du VPU.
i9	Intel Core i9-14900HX (24 cœurs)	*Aucun* (CPU-only)	32 Go RAM	Puissance brute élevée mais ratio perf/Watt défavorable par rapport au GPU.
4070	Intel Core i9	NVIDIA RTX 4070 (Laptop, 8 Go VRAM)	8 Go VRAM (Dédiée)	Inférence très rapide, le goulot se déplace vers les transferts PCIe et le driver.

3 Optimisations

3.1 Contexte

Afin de structurer notre exploration, nous regroupons les optimisations en quatre familles correspondant à un découpage 2×2 : (i) optimisations à la compilation vs (ii) optimisations au *runtime*, et, dans chaque cas, (iii) optimisations agnostiques du matériel vs (iv) optimisations spécifiques à une plateforme. Intuitivement, la compilation regroupe tout ce qui transforme le modèle avant l'exécution (export, conversion de format, génération d'un moteur, choix de précision, etc.), tandis que le *runtime* correspond aux réglages appliqués au moment de l'inférence/l'utilisation (p. ex. fusions de couches activées par le moteur, choix du niveau d'optimisation du graphe). Dans notre protocole, nous nous appuyons sur trois moteurs d'exécution (de *runtime*) :

1. **ONNX Runtime (ORT)** pour exécuter des modèles au format ONNX (format standard décrivant un modèle IA sous forme d'un graphe de calcul),
2. **TensorRT (TRT)** sur matériel NVIDIA (génération d'un modèle IA sous un format *engine* optimisé pour matériel NVIDIA),
3. et le ***runtime* DepthAI (DRT)** sur Oak (exécution d'un modèle compilé au format `.blob` pour le VPU MyriadX).

Le tableau 2 récapitule, pour chaque plateforme matérielle, le moteurs d'exécution (*runtime*) retenus afin d'assurer une comparaison cohérente entre cibles.

Matériel	Moteur
i9	ORT
4070	ORT
Pi	ORT
Orin	TRT
Oak	DRT

Table 2: Correspondance entre plateformes matérielles et moteurs d'exécution considérés.

3.2 Optimisations à la compilation

3.2.1 Optimisations agnostiques matériel

Notre base d'expériences repose sur trois leviers, indépendants de la plateforme, qui modifient directement la complexité et/ou le coût numérique du modèle :

- **Taille (élagage) du modèle** (variante YOLO) : {`n`, `s`, `m`}. Une variante plus petite d'un modèle d'IA réduit typiquement le nombre de paramètres et d'opérations, donc la latence et la mémoire, au prix d'une précision potentiellement moindre. Concrètement dans notre *workflow*, nous utilisons directement les checkpoints officiels Ultralytics (`yolo11n.pt`, `yolo11s.pt`, `yolo11m.pt`) et nous exportons chaque variante séparément au format ONNX. Ce levier est particulièrement critique sur les cibles *edge* contraintes et devrait réduire aussi l'empreinte mémoire côté GPU, ce qui limite les saturations mémoire et stabilise la latence.

- **Résolution d'entrée du modèle** : {640, 512, 416, 320, 256}. Une résolution d'image en entrée du modèle plus faible diminue le coût de calcul (moins de pixels à traiter) et accélère l'inférence, mais peut dégrader la détection des petits objets. Dans notre code, la résolution est fixée à l'export via l'argument `imgsz` (export ONNX non-dynamique), ce qui fige la forme du tenseur d'entrée et, par conséquent, les dimensions des tenseurs produits par la tête de détection ; au runtime, on applique le même *letterbox* vers (`imgsz`, `imgsz`) pour rester cohérent. Sur les plateformes où la mémoire/bande passante est un facteur (LPDDR4/Pi, LPDDR5 4 Go/Orin) et/ou où le pipeline est serré (Oak), ce choix devrait réduire fortement la taille des activations et donc la pression mémoire et les transferts, souvent déterminants pour la latence *end-to-end*.

- **Quantification du modèle** : {`FP32`, `FP16`}. Passer en `FP16` réduit la bande passante mémoire et peut accélérer l'exécution sur du matériel qui l'exploite bien, avec un impact généralement limité sur la qualité. Concrètement, on pilote ce choix dès l'export Ultralytics via `half=True` (FP16) ou `half=False` (FP32), ce qui produit des modèles ONNX dont l'entrée (et les poids) sont en `float16` ou `float32` ; lors du benchmark, on détecte ce dtype et on alimente l'inférence avec des tenseurs du même type. C'est surtout pertinent sur les plateformes GPU où TensorRT/CUDA peut exploiter des chemins FP16 optimisés (notamment via les Tensor Cores) et où la réduction de bande passante/VRAM aide directement à tenir dans des budgets mémoire serrés (ex. 4 Go sur Orin).

3.2.2 Optimisations spécifiques matériel

Certaines optimisations dépendent directement de l'accélérateur ciblé :

- **Oak — nombre de SHAVEs** : {4, 5, 6, 7, 8}. Les SHAVEs sont des processeurs vectoriels internes du VPU ; en allouer davantage pour l'inférence du modèle d'IA peut améliorer le débit de certaines opérations, au prix d'une contention possible avec d'autres traitements sur l'appareil. Dans notre *workflow*, on ne change pas les SHAVEs à l'exécution : on recompile le modèle pour l'OAK en générant un `.blob` via les outils DepthAI (conversion/compilation), en fixant le nombre de SHAVEs lors de la compilation (paramètre de type `-sh`/`numShaves`). Côté matériel, cet hyperparamètre agit directement sur la part de ressources du VPU réservée au réseau : cela peut réduire la latence ou augmenter le débit sur Oak si le réseau est limité par le calcul vectoriel, mais l'effet n'est pas garanti monotone car il dépend aussi des autres blocs actifs (ISP, stéréo, transferts/queues hôte ↔ caméra).

- **Orin — heuristique de *build*** (sélection de tactiques) : {`false`, `true`}. Activer ce mode demande à TensorRT d'utiliser une sélection heuristique de certaines tactiques afin de réduire le temps de génération de l'*engine*, au prix potentiel d'un *engine* légèrement moins performant qu'avec une recherche/profilage plus exhaustive. Dans notre *workflow*, ce choix est appliqué uniquement au moment de la construction de l'*engine* TensorRT (compilation ONNX → *engine*) en basculant l'option `heuristic` ON/OFF dans la configuration du *builder* ; ensuite, l'*engine* est réutilisé (cache) pour exécuter les mesures. Côté matériel, cela n'accélère pas directement le GPU : l'objectif est surtout de raccourcir la phase de compilation et donc le *time-to-first-run*. On s'attend donc à un impact faible (souvent nul) sur les performances *runtime* une fois l'*engine* construit, même si en théorie un choix heuristique peut conduire à des tactiques un peu moins optimales.

- **Orin — *sparsity*** : {`false`, `true`}. Lorsque le modèle et le matériel le permettent, activer la sparsity vise à tirer parti d'une accélération matérielle pour des poids structurés clairsemés. Dans notre *workflow*, on active/désactive ce mode au moment de la construction TensorRT (`-sparsity=enable`), sans modifier l'architecture à l'exécution. Côté matériel, l'accélération est conditionnée à une sparsité structurée 2:4 exploitée par les Tensor Cores ; comme nos modèles YOLO ne sont pas entraînés/prunés pour respecter ce motif, activer la sparsité devrait être en grande partie ignoré (ou n'apporter qu'un gain négligeable) sur l'Orin.

3.3 Optimisations au *runtime*

3.3.1 Optimisations agnostiques matériel

Nous n'avons pas retenu d'optimisations *runtime* agnostiques des plateformes.

3.3.2 Optimisations spécifiques plateforme

Au moment de l'exécution, nous évaluons un réglage de la fusion du graphe via ORT :

- **Pi, i9, 4070 — Fusion de graphe** : {`DISABLE_ALL`, `ENABLE_BASIC`, `ENABLE_ALL`}. Juste avant l'exécution d'un modèle, ORT applique un ensemble de passes sans changement sémantique sur le graphe ONNX (simplifications, *constant folding*, élimination de nœuds redondants, fusions d'opérateurs compatibles...). Le but est de réduire le nombre d'opérations exécutées et les mouvements mémoire (moins de kernels/appels, meilleure localité), ce qui devrait diminuer la latence E2E. Le gain

est en général plus marqué sur CPU, où réduire les allocations, copies et surcoûts d'exécution améliore directement les performances et la pression mémoire. Sur GPU via ORT, les fusions peuvent aussi aider en réduisant des lancements de kernels et des passages mémoire, mais l'effet peut être moins net si le goulot principal est ailleurs (pré/post-traitement CPU, transferts).

3.4 Récapitulatif

Le tableau 3 récapitule l'ensemble des optimisations étudiées, en distinguant celles appliquées à la compilation et au *runtime*, ainsi que leur caractère agnostique ou spécifique à une plateforme.

Famille	**Optimisations**	**Effet attendu (matériel)**
Agnostique / compilation	Élagage (`n`,`s`,`m`) Résolution (640, 512, 416, 320, 256) Quantification (`FP32`, `FP16`)	↓ calcul + ↓ activations/mémoire : gains forts sur Pi (CPU-*only*) et Oak (VPU), et aide à tenir dans les budgets mémoire (ex. Orin 4 Go).
Spécifique / compilation	SHAVEs (Oak, 5 niveaux) *Heuristic* (Orin, on/off) *Sparsity* (Orin, on/off)	Oak: ajuste les ressources VPU allouées au modèle IA (débit/latence), contention possible avec le reste du pipeline. Orin: *heuristic* vise surtout ↓ temps de compilation (impact *runtime* faible). *Sparsity*: attendu faible/nul si le modèle n'est pas au motif compatible (ex. 2:4).
Spécifique / runtime	Fusions (Pi, i9, 4070, 3 niveaux)	↓ *overhead* (fusions, allocations/copies) : gains surtout sur CPU, parfois sur GPU via ORT.

Table 3: Synthèse des optimisations évaluées (portée/phase) et effet attendu selon la plateforme.

Ces optimisations sont globalement combinables : on peut, pour une plateforme donnée, composer les variantes agnostiques compilation (taille/résolution/précision) avec les options spécifiques compilation quand elles existent (p. ex. SHAVEs sur Oak, options de compilation sur TRT). En revanche, les réglages *runtime* dépendent du moteur d'exécution : la fusion ORT ne s'applique qu'avec ONNX Runtime, et les optimisations TensorRT (*heuristic*, *sparsity*) ne concernent que les modèles compilés en *engine* TRT et exécutés via TRT. De plus, selon le matériel, le point de départ n'est pas identique ; le VPU de la Oak ne prend que des modèles au format FP16, et pour la Pi, nous n'avons considérés que les modèles de taille `n`, ceux de tailles supérieurs étant trop lourds et ralentissant beaucoup trops les *benchmarks*.

À partir de cette grille d'optimisations et des contraintes propres à chaque plateforme, nous pouvons établir ci-dessous un récapitulatif du nombre de variantes effectivement évaluables par matériel.

- Pi : 1 taille × 5 résolutions × 2 quantification × 3 fusion = 30 variantes.
- Oak : 3 tailles × 5 réso. × 1 quantif. × 5 SHAVEs = 75 variantes.
- Orin : 3 tailles × 5 réso. × 2 quantif. × 2 *heuris.* × 2 *spars.* = 120 variantes,

- i9 : 3 tailles × 5 réso. × 2 quantif. × 3 fusions = 90 variantes.
- 4070 : 3 tailles × 5 réso. × 2 quantif. × 3 fusions = 90 variantes.

Soit un total de 405 variantes.

4 Métriques mesurées

Après avoir défini les différentes optimisations (agnostiques et spécifiques) ainsi que les variantes effectivement testables sur chaque plateforme, l'étape suivante consiste à mesurer systématiquement les performances des couples (matériel, combinaison d'optimisations). Pour garder une lecture claire, nous regroupons les mesures en deux familles : (i) des métriques décrivant la performance et la qualité du modèle IA (section 4.1), et (ii) des métriques décrivant l'utilisation des ressources matérielles pendant l'inférence (CPU, mémoire, accélérateurs, etc., section 4.2).

4.1 Mesures de performances du modèle d'IA

Pour garder une lecture simple et comparable entre plateformes, nous concentrons l'analyse sur deux métriques principales : (i) la latence d'inférence *end-to-end* (E2E) et (ii) la qualité de détection via le mAP@50.

- **Latence E2E (*end-to-end*) par image.** La latence E2E correspond au temps total pour traiter une image, en incluant pré-traitement → inférence → post-traitement (redimensionnement/*letterbox*, exécution du modèle, NMS/formatage des sorties). Nous avons retenu cette métrique car elle reflète la latence réellement perçue en application, et surtout parce qu'elle est mesurable de manière homogène sur tous les matériels : obtenir une décomposition fiable (pré/inférence/post) n'est pas toujours comparable d'un *backend* à l'autre, et devient particulièrement délicat sur Oak (pipeline DepthAI asynchrone, files d'attente et transferts hôte ↔ caméra).
- **Qualité de détection : mAP@50.** Le mAP@50 (*mean Average Precision* à IoU=0.50) est une métrique standard pour évaluer un détecteur d'objets. Elle mesure la qualité globale en calculant, pour chaque classe, l'aire sous la courbe précision-rappel (AP), puis en moyennant sur les classes (mAP). Une détection est considérée correcte si la boîte prédite recouvre la vérité terrain avec une IoU ≥ 0.50 ; ainsi, plus le modèle détecte les bonnes classes avec des boîtes bien localisées (et des scores permettant un bon compromis précision/rappel), plus le mAP@50 augmente.

4.2 Mesures de consommation du matériel

En complément des métriques de performance (E2E et mAP@50), nous instrumentons l'exécution afin d'observer la pression sur le système et d'interpréter les latences mesurées. Toutes les mesures sont échantillonnées périodiquement pendant le *benchmark* (*thread* de *monitoring*, typiquement toutes les 500 à 2000ms) puis résumées (moyenne, p95).

- **Utilisation CPU / GPU (et limites côté VPU)** : On mesure la charge CPU (Pi, i9) au niveau du processus de *benchmark* via `psutil` (`Process.cpu_percent`), avec une version normalisée par le nombre de cœurs logiques lorsque disponible.

Côté 4070, nous lisons l'utilisation GPU (%) et la VRAM (MB) via `nvidia-smi` (`-query-gpu=utilization.gpu,memory.used`). Sur Orin, nous n'avons pas `nvidia-smi` : nous utilisons `tegrastats` et extrayons `GR3D_FREQ` (%) et `EMC_FREQ` (%) comme proxy respectivement de l'activité GPU et de la pression mémoire/bande passante. Sur Oak (VPU), nous n'avons pas obtenu de métrique robuste et comparable d'« utilisation VPU » : nous ne reportons donc pas de % d'occupation VPU. En pratique, nous récupérons plutôt des indicateurs internes via `SystemLogger` (charge CPU embarquée *Leon CSS*) et, lorsque disponible, un *temps d'inférence VPU estimé* à partir des traces (`DEPTHAI_LEVEL=trace` + extraction par motif). Sources d'incertitude : (i) échantillonnage discret (pics possibles entre deux mesures), (ii) `cpu_percent` dépend de l'ordonnanceur et peut inclure du *jitter* Python, (iii) `GR3D_FREQ` / `EMC_FREQ` reflètent des fréquences/états (DVFS) plutôt qu'un % d'occupation « pur », (iv) côté Oak, les traces peuvent être absentes/incomplètes selon le pipeline et la verbosité activée.

- **RAM / VRAM** : Nous suivons (i) la mémoire RSS du processus de *benchmark* (empreinte mémoire du programme) et (ii) la RAM système utilisée (pression mémoire globale) à nouveau via `psutil` (`Process.memory_info().rss` et `virtual_memory`). Sur Oak, nous récupérons des compteurs mémoire internes (*DDR* et *CMX*) via `SystemLogger`. Sur 4070, la VRAM utilisée est lue via `nvidia-smi` en même temps que l'utilisation GPU. Sources d'incertitude : (i) la RSS dépend de l'allocateur Python et peut sur-estimer l'empreinte « utile » (fragmentation, caches), (ii) la RAM système utilisée inclut caches/buffers OS (donc pas directement imputable au seul benchmark), (iii) la VRAM mesurée par `nvidia-smi` correspond à de l'allocation effective (pas nécessairement à des activations « actives » à l'instant), (iv) sur mémoire partagée (Jetson), la frontière RAM/VRAM est moins nette.

- **Puissance électrique** : La puissance électrique active (W) est mesurée externe au système via une prise connectée Tapo P110, insérée entre le secteur et l'alimentation du matériel testé (mesure « niveau système »). Dans le code, le module de *monitoring* se connecte à la prise via la bibliothèque `tapo` (`ApiClient`) et interroge `get_current_power()` à intervalle fixe (par défaut toutes les 2s), pour ne pas surcharger l'API Tapo qui autorise difficilement une fréquence plus élevée, avec gestion d'échecs temporaires et désactivation si les lectures échouent. Lorsque nécessaire, nous convertissons la valeur renvoyée en watts via un facteur d'échelle configurable (`-tapo-power-scale`). Sources d'incertitude : (i) mesure au niveau prise ⇒ inclut les pertes du bloc d'alimentation (rendement) et d'éventuels périphériques, (ii) résolution temporelle limitée (*polling* 2s) ⇒ pics courts potentiellement lissés, (iii) latence réseau/Wi-Fi et appels asynchrones ⇒ valeurs manquantes possibles, (iv) sensibilité à l'état du système (processus de fond), ce qui motive une mesure *idle* préalable dans le protocole.

4.3 Résumé

Le tableau 4 synthétise les différentes métriques instrumentées durant les *benchmarks*, en précisant pour chacune la méthode de mesure utilisée (outil ou bibliothèque) ainsi que les principales limites et sources d'incertitude identifiées, afin de faciliter l'interprétation des résultats et la comparaison entre plateformes.

Métrique	**Méthode de mesure**	**Incertitudes / limites**
CPU (%)	`psutil` (*process cpu_percent*)	Échantillonnage discret (pics manqués), bruit OS (tâches de fond/interruptions), *jitter* Python/ordonnanceur ; dépend du pas de mesure et de la normalisation par cœurs.
GPU (%)	PC: `nvidia-smi` ; Jetson: `tegrastats` (`GR3D_FREQ`)	Asynchronisme (kernels en rafales), DVFS (fréquence ≠ occupation), proxy Jetson (fréquence/état plutôt qu'un % exact), charge influencée par pré/post CPU et transferts.
RAM (MB)	`psutil` (RSS proc. + RAM système)	RSS peut inclure fragmentation/caches ; RAM système inclut caches/buffers OS (pas imputable au seul *benchmark*) ; comparabilité limitée sur mémoire partagée (Jetson).
VRAM (MB)	PC: `nvidia-smi` ; Oak: DepthAI `SystemLogger` (DDR/CMX)	`nvidia-smi` mesure de l'allocation (pas l'usage instantané) ; sur Oak, compteurs internes dépendants du pipeline et non directement comparables à une VRAM GPU ; valeurs peuvent varier avec buffering/queues.
Puissance (W)	Prise Tapo P110 : polling via API (toutes les 2s)	Mesure « au mur » (inclut pertes alim + périphériques), résolution temporelle faible (pics lissés), latence réseau/Wi-Fi et lectures manquantes ; dépend de l'état idle et du bruit de fond système.

Table 4: Synthèse des mesures de consommation/ressources : méthode et limites principales.

5 Protocole de benchmarking

5.1 Description

Après avoir défini l'espace des optimisations et des métriques relevées, l'enjeu devient de mesurer leurs effets de façon comparable d'une plateforme à l'autre. Nous appliquons un protocole identique sur chaque matériel, fondé sur trois précautions visant à limiter les principales sources d'incertitude identifiées précédemment :

1. **Mesure à vide (*idle baseline*).** Avant toute inférence, nous mesurons pendant quelques secondes le système *sans charge applicative*, afin d'estimer la consommation et l'utilisation des ressources dues uniquement à l'OS et aux processus en arrière-plan. Cette étape permet (i) de détecter d'éventuelles perturbations (tâches de fond, indexation, mises à jour) et (ii) de disposer d'un niveau de référence pour interpréter les mesures « en charge ». Elle est particulièrement utile pour la puissance mesurée au niveau prise (qui inclut le bruit de fond et les pertes d'alimentation) ainsi que pour les métriques CPU/RAM (sensibles aux caches/buffers OS).

2. **Phase de *warmup* (inférences non comptabilisées).** Nous exécutons un nombre fixe d'inférences initiales sans les comptabiliser (par défaut 10), afin de stabiliser le pipeline d'exécution. Cette phase vise à atténuer les effets de *cold*

start (allocations mémoire, initialisation des *backends*, remplissage des caches, premières compilations/optimisations internes) et la montée en fréquence progressive des composants (DVFS), qui peuvent biaiser les premières itérations. L'effet est marqué sur les exécutions accélérées (GPU/TensorRT) où les lancements asynchrones et les buffers peuvent s'amortir après quelques itérations, et sur Oak où les files d'attente/pipelines *device* ↔ *host* peuvent nécessiter un régime établi pour être représentatifs.

3. **Répétitions (3 exécutions) et agrégation.** Chaque expérience (même couple *hardware*/*runtime* et même variante de modèle) est répétée trois fois, puis nous agrégeons les résultats lors de l'analyse. Cette pratique permet de lisser la variabilité *run-to-run* due au bruit OS, au *jitter* d'échantillonnage des capteurs (CPU/GPU) et, pour la puissance, au *polling* relativement lent de la prise (toutes les 2 secondes, pouvant lisser des pics courts). Les répétitions sont aussi un moyen simple de mitiger les mesures aberrantes (transitoires réseau pour la prise, *throttling* ponctuel, contention inattendue), qui pourraient autrement être sur-interprétées.

Une fois ces conditions réunies, chaque couple (*hardware*, *runtime*) exécute les mêmes tests d'inférence sur un jeu de données commun. Nous utilisons `coco128`, qui contient 128 images, suffisamment petit pour rester exécutable rapidement sur l'ensemble des plateformes, tout en étant assez grand pour obtenir des métriques exploitables. Pour chaque variante, l'inférence est effectuée sur l'ensemble des images et les métriques sont ensuite calculées et enregistrées au format CSV.

6 Résultats et analyse

Cette section analyse les résultats obtenus avec le protocole de section 5, en lien direct avec les leviers d'optimisation introduits en section 3 et les contraintes matérielles décrites en section 2.

La figure 1 est un histogramme comparant la latence E2E p95 (ms) en fonction de la résolution d'entrée (px), pour trois plateformes *edge* (Orin, Oak, Pi). L'axe des abscisses correspond aux résolutions testées (256, 320, 416, 512, 640) et l'axe des ordonnées à la latence p95.

On observe une augmentation nette de la latence avec la résolution sur toutes les plateformes. L'Orin reste dans un régime de latence faible et croît modérément, tandis que Pi et Oak voient leur latence augmenter fortement, atteignant des ordres de grandeur similaires à haute résolution (à 640 px, Oak $\approx$ 636 ms et Pi $\approx$ 638 ms). Ce résultat confirme l'hypothèse que la résolution agit comme levier principal sur les cibles contraintes, car elle augmente simultanément la charge de calcul et la pression mémoire via la taille des activations.

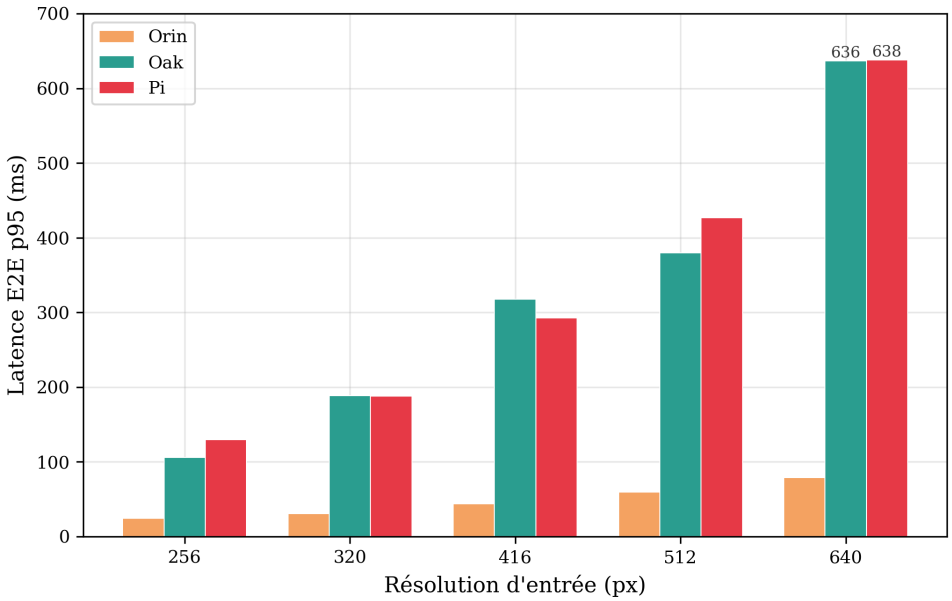


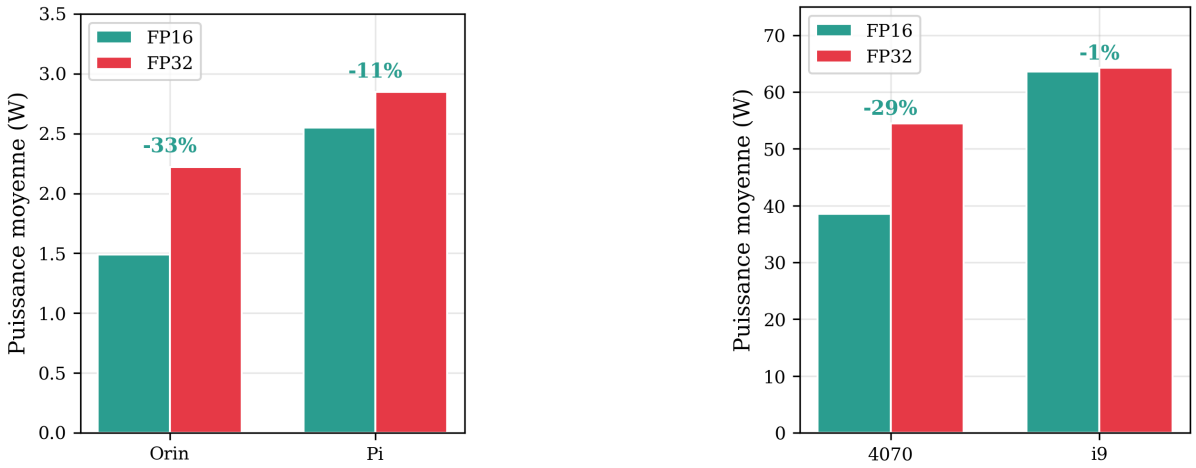
Figure 1: Latence en fonction de la résolution d'entrée, par plateformes *edges*

La figure 2 regroupe deux histogrammes : (i) plateformes *edge* et (ii) plateformes 4070 et i9. L'axe des abscisses correspond à la plateforme, l'axe des ordonnées à la puissance moyenne (W), et chaque paire de barres compare FP16 à FP32.

Sur Orin et 4070, FP16 réduit fortement la puissance moyenne, ce qui est cohérent avec une exécution accélérée exploitant efficacement la réduction de précision. À l'inverse, sur i9 l'effet est quasi nul, et sur Pi l'effet est plus modéré : cela met en évidence une tendance négative importante, à savoir que FP16 n'est pas un levier énergétique déterminant en exécution CPU-only.

Plateforme	Variation de puissance (FP16 vs FP32)	Observation
Orin	-33%	baisse marquée
Pi	-11%	baisse modérée
4070	-29%	baisse marquée
i9	-1%	effet négligeable

Table 5: Effet de FP16 sur la puissance moyenne (valeurs indiquées sur figure 2).



(a) Plateformes *edge* (b) Plateformes 4070 et i9

Figure 2: Puissance électrique en fonction du niveau de quantification, par plateformes

La figure 3 est un histogramme comparant la latence E2E p95 (ms) en fonction du niveau de fusion ORT (`DISABLE_ALL`, `ENABLE_BASIC`, `ENABLE_ALL`) sur 4070, Pi et i9. L'axe des abscisses est la plateforme, l'axe des ordonnées la latence, et les couleurs correspondent aux niveaux de fusion.

On observe que `ENABLE_ALL` a un impact très fort sur i9, un impact faible sur Pi, et un impact négligeable sur 4070. Cela confirme l'hypothèse que les fusions ORT sont surtout pertinentes quand l'exécution est dominée par le CPU, alors qu'en régime accéléré (GPU) la latence est davantage contrainte par d'autres éléments (orchestration, transferts, kernels).

Plateforme	Gain relatif avec `ENABLE_ALL`	Lecture
i9	-31%	réduction importante de la latence
Pi	-4%	effet limité
4070	$\approx 0\%$	effet négligeable

Table 6: Impact du niveau de fusion ORT sur la latence (valeurs indiquées sur figure 3).

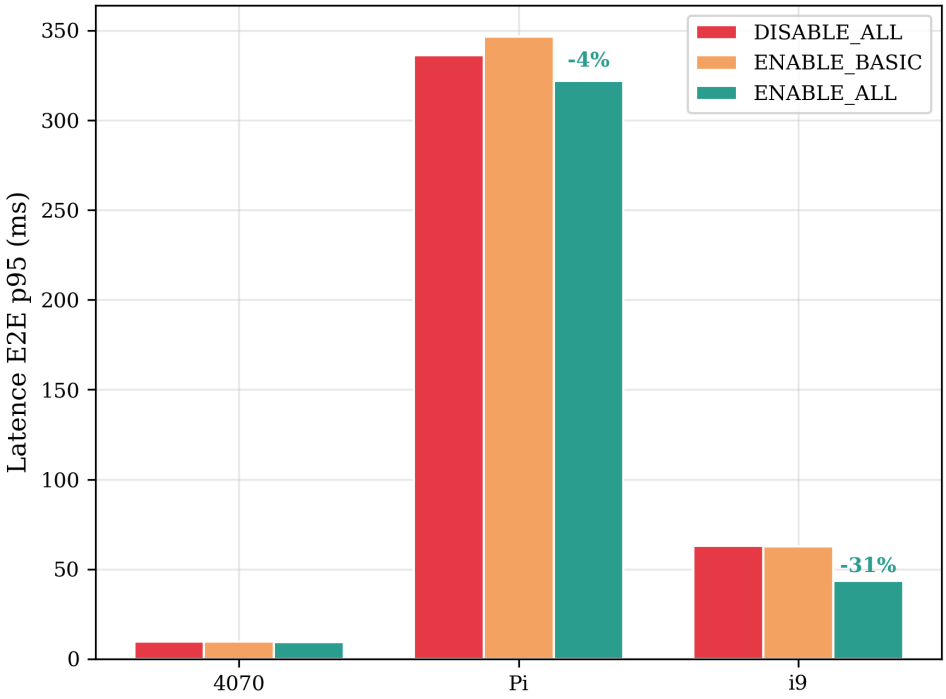


Figure 3: Latence en fonction du niveau de fusion, par plateformes

La figure 4 est un nuage de points (échelles log) reliant puissance (W, axe x) et latence E2E p95 (ms, axe y) sur Oak. La figure encode la résolution (forme), la taille du modèle (taille du marqueur) et le nombre de SHAVEs (couleur). La tendance principale est structurée par la résolution et la taille du modèle : à résolution plus élevée et modèle plus grand, la latence augmente fortement. Le nombre de SHAVEs agit comme réglage plus fin : l'effet n'est pas strictement monotone, et reste secondaire face à la résolution et au niveau d'élagage.

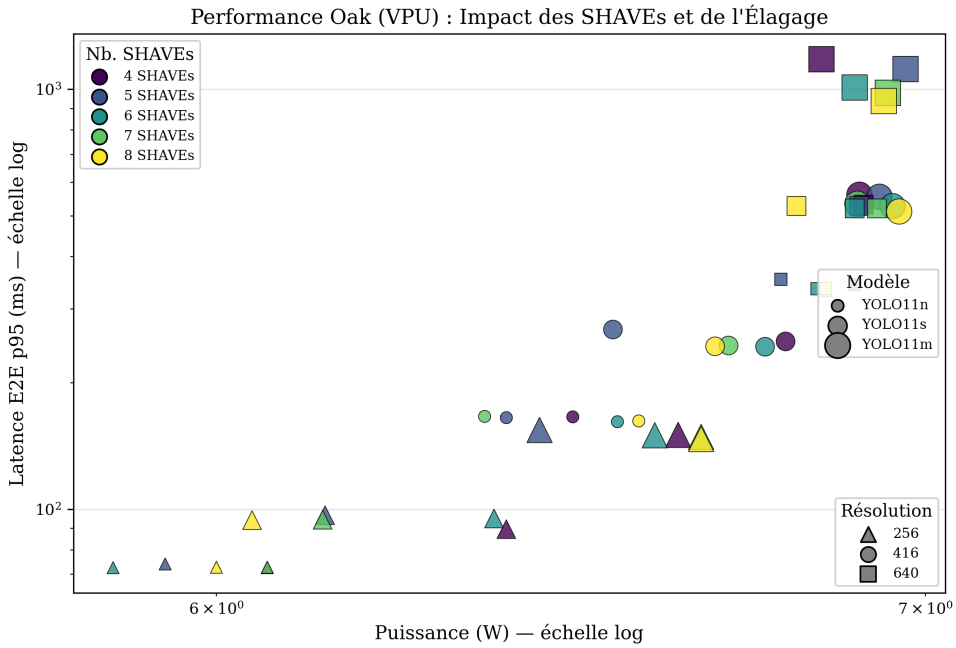


Figure 4: Compromis latence/puissance électrique en fonction de du niveau d'élagage et de résolution, et du nombre de SHAVEs, sur Oak

Pour Orin, la figure 5 confirme que les points se structurent d'abord par résolution et taille de modèle, et que FP16 tend à améliorer le compromis énergétique. En revanche, nos options spécifiques TensorRT (mode *heuristic* et *sparsity*, tableau 7) ne montrent pas de tendance robuste sur les métriques *runtime* (latence E2E p95 et puissance moyenne) au-delà de la variabilité : c'est une tendance négative importante à rapporter, cohérente avec le fait que (i) *heuristic* vise principalement le temps de construction de l'engine et (ii) *sparsity* requiert une sparsité structurée absente de nos variantes.

	Heuristic		*Sparsity*	
Métrique	H0	H1	SP0	SP1
E2E p95 (ms)	**47.08**	47.42	47.31	**47.19**
mAP@50 (%)	**66.32**	66.3	**66.31**	66.3
GPU util (%)	**45.88**	46.23	46.67	**45.44**
Puissance moyenne (W)	1.87	**1.84**	1.87	**1.84**

Table 7: Comparaison des mesures en fonction des niveaux d'*heuristic* et de *sparsity*

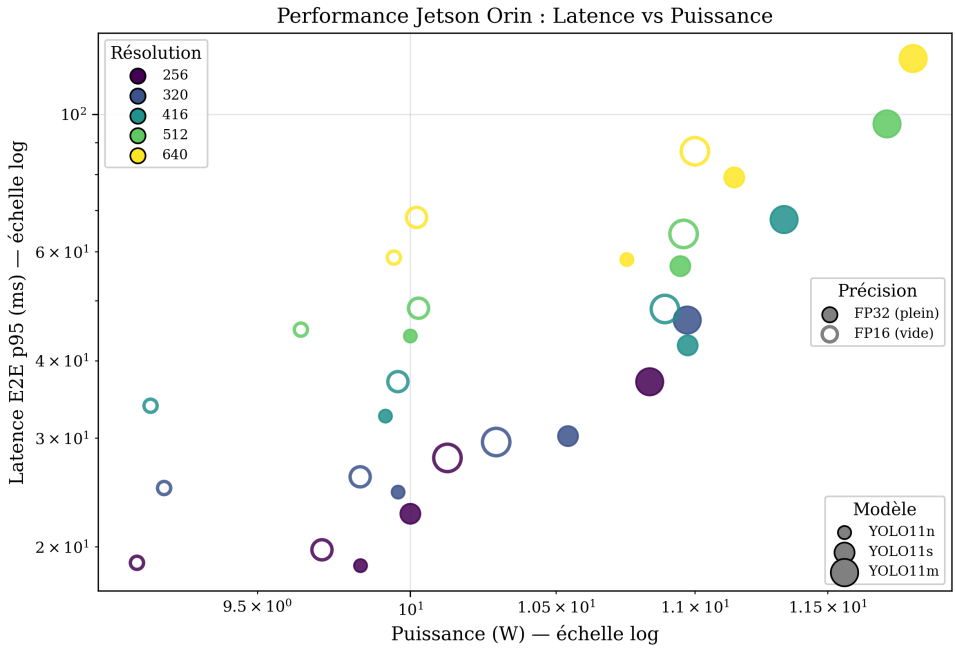


Figure 5: Compromis latence/puissance en fonction du niveau d'élagage, de résolution et de quantification, sur Orin

La figure 6 représente la précision mAP@50 (axe y) en fonction de la latence E2E p95 (ms, axe x en log), en distinguant plateformes, tailles de modèles et résolutions. Cette figure synthétise le compromis global : la précision augmente globalement avec la résolution et la taille du modèle, au prix d'une latence plus élevée.

On observe une séparation claire des régimes : 4070 se situe dans la zone faible latence avec précision élevée, Orin et i9 forment un compromis intermédiaire, tandis que Pi et Oak sont décalés vers des latences nettement plus élevées pour des niveaux de précision comparables. Ce résultat met en évidence que, sur les cibles *edge*, la contrainte de latence impose plus souvent de réduire résolution et taille du modèle, ce qui limite la précision atteignable à latence acceptable.

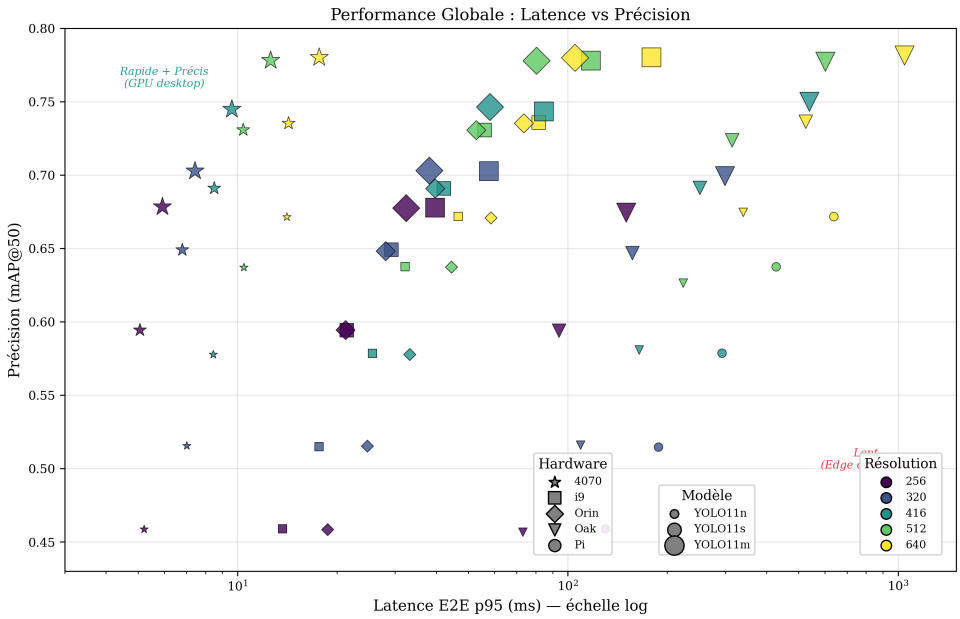


Figure 6: Compromis précision/latence en fonction de la plateforme, du niveau d'élagage et de résolution

Les figures 7 à 9 montrent, pour Pi, Orin et Oak, un nuage de points puissance moyenne (W, axe y) vs latence E2E p95 (ms, axe x), et mettent en évidence les variantes non dominées (front de Pareto). Les points annotés correspondent aux configurations candidates selon l'objectif (minimiser la latence à puissance donnée, ou minimiser la puissance à latence donnée).

Sur Pi, le front met en avant des variantes `n/256` comme meilleures en latence, avec quelques compromis à plus haute résolution. Sur Orin, le front est réduit et souligne surtout l'intérêt de FP16 à latence comparable. Sur Oak, un candidat `n/256/FP16` avec un nombre intermédiaire de SHAVEs ressort comme compromis favorable, ce qui est cohérent avec l'analyse précédente (SHAVEs agit surtout en réglage fin).

7 Conclusion

Ce projet avait pour objectif d'étudier, dans un cadre reproductible, comment différentes couches d'optimisation d'un détecteur d'objets (YOLOv11) interagissent avec des plateformes hétérogènes, en particulier *edge*. Pour cela, nous avons construit un *workflow* unifié (export, compilation, exécution) et un protocole de *benchmark* commun, mesurant principalement la latence *end-to-end* (p95), la puissance moyenne au niveau système, et la précision (mAP@50), tout en instrumentant l'exécution pour interpréter les résultats.

Les résultats montrent que les leviers les plus structurants sont ceux qui modifient directement la complexité du problème : la résolution d'entrée et la taille du modèle. Sur les cibles contraintes (Pi et Oak), la résolution domine largement la latence, avec une dégradation forte lorsque l'on monte en 512–640 px, tandis que l'Orin maintient un régime nettement plus rapide grâce à l'accélération GPU. La quantification FP16 améliore nettement le compromis énergétique sur les plateformes accélérées (Orin, 4070), alors que l'effet est faible ou négligeable sur CPU (i9 et, dans une moindre mesure, Pi). Côté *runtime*, les optimisations de fusion ORT apportent un gain important sur i9, mais restent limitées sur Pi et quasi nulles sur 4070, ce qui confirme que leur intérêt dépend fortement

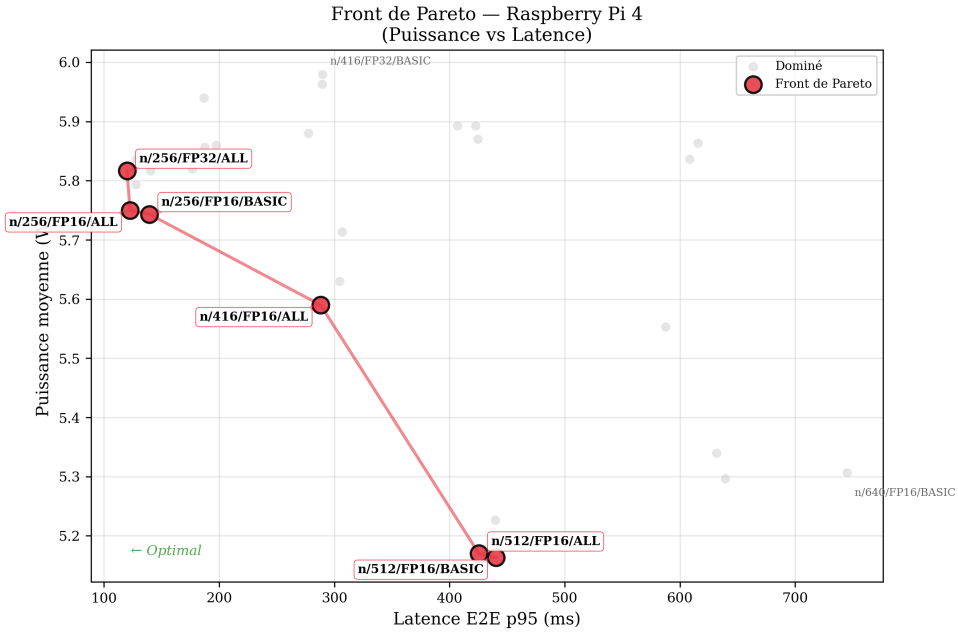


Figure 7: Meilleures variantes évaluées sur Pi par rapport au compromis puissance électrique/latence

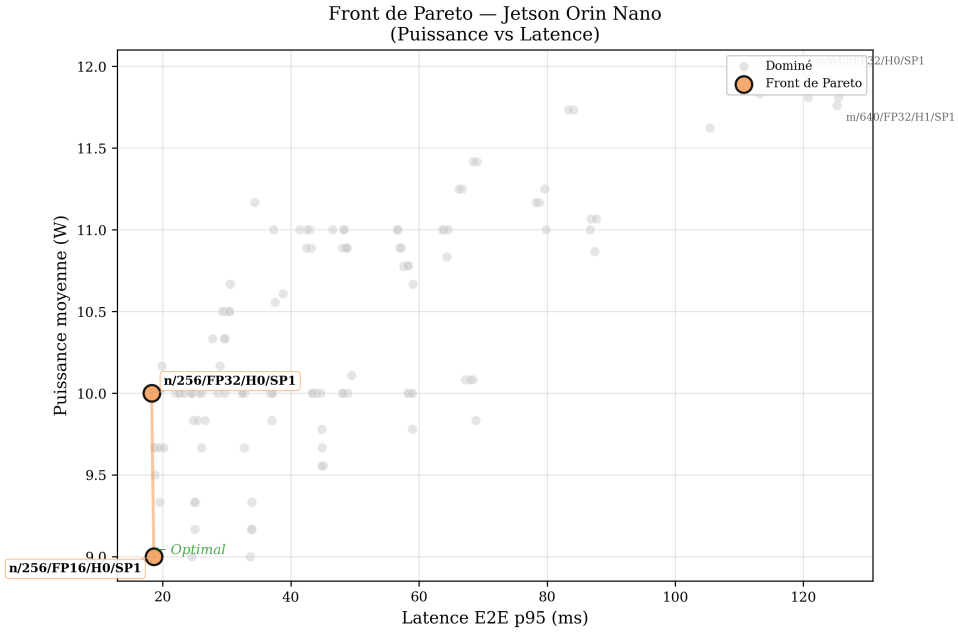


Figure 8: Meilleures variantes évaluées sur Orin par rapport au compromis puissance électrique/latence

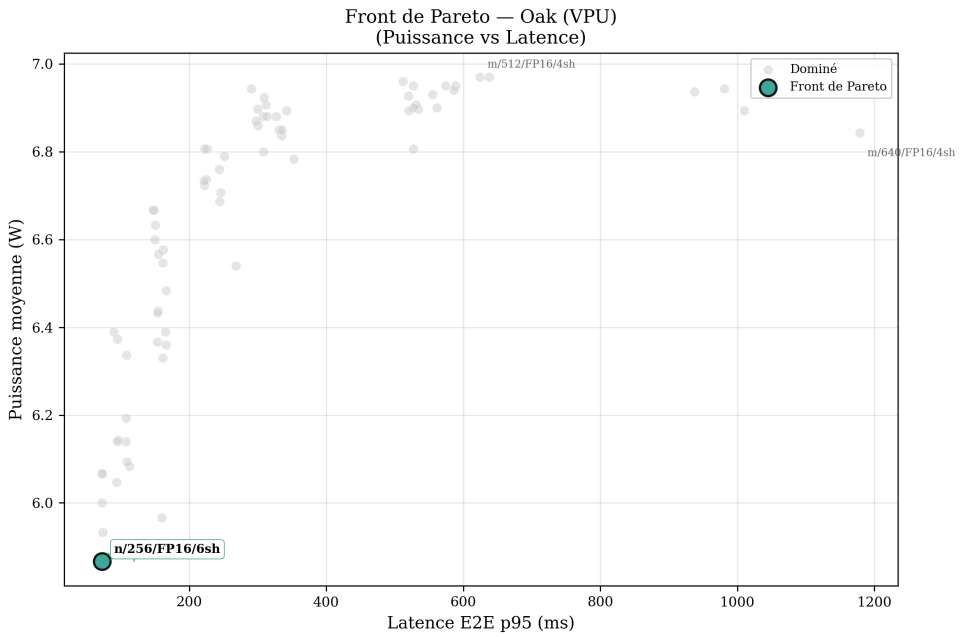


Figure 9: Meilleures variantes évaluées sur Oak par rapport au compromis puissance électrique/latence

du goulot (overhead d'exécution vs calcul accéléré). Enfin, les optimisations spécifiques plateforme se comportent comme attendu : sur Oak, le nombre de SHAVEs agit surtout comme réglage fin derrière résolution/élagage, et sur Orin les options *heuristic* et *sparsity* n'ont pas montré d'impact *runtime* robuste dans nos conditions, ce qui suggère que leur effet principal est ailleurs (notamment sur le *build time*) ou conditionné à des prérequis (sparsité structurée).

Ces observations permettent de proposer, par plateforme, des configurations candidates via une lecture multi-objectif (fronts de Pareto puissance/latence) : (i) sur Pi et Oak, des variantes petites et basse résolution constituent la base la plus réaliste pour respecter des contraintes temps réel, (ii) sur Orin, FP16 domine généralement FP32 à latence comparable, et (iii) sur i9/4070, les choix de *runtime* (fusions) et de précision peuvent surtout être guidés par des objectifs énergétiques plutôt que par la latence brute.

Plusieurs limites doivent toutefois être soulignées : la comparabilité des métriques d'occupation accélérateur n'est pas parfaite (en particulier sur Oak), et le jeu de données coco128 réduit le temps d'évaluation mais peut limiter la stabilité statistique de certaines métriques. En ouverture, un prolongement naturel consisterait à (i) ajouter explicitement les métriques manquantes liées au déploiement (énergie par image, débit, mémoire, *time-to-first-run* et temps de compilation des moteurs), (ii) étendre l'étude à INT8 (calibration) et à des optimisations structurelles (pruning/sparsité structurée) pour rendre effectives certaines options matérielles, et (iii) mieux caractériser les effets système (température, *throttling*, multi-flux, contention CPU pré/post-traitement, transfert USB sur Oak) afin de rapprocher encore les mesures des contraintes d'une intégration *IoT-CPS* réelle.

Annexe

Metric	Hardware	n	s	m
E2E p95 (ms)	4070	9.05	**9.0**	10.64
	Oak	**181.82**	268.12	526.87
	Orin	**35.9**	43.06	62.78
	i9	**27.15**	46.11	95.75
mAP@50 (%)	4070	57.23	68.02	**73.7**
	Oak	57.1	67.86	**73.67**
	Orin	57.21	68.0	**73.72**
	i9	57.24	68.03	**73.65**
CPU util (%)	i9	73.02	71.38	**70.1**
GPU util (%)	4070	**13.79**	16.53	24.67
	Orin	**32.23**	45.24	60.69
RAM (MB)	4070	1053.94	1053.69	**1006.45**
	i9	**106.03**	153.3	265.17
VRAM (MB)	4070	**-57.15**	12.73	157.13
Power mean (W)	4070	**38.76**	45.18	55.64
	Oak	**1.45**	1.6	1.86
	Orin	**1.25**	1.82	2.5

Table 8: Mesures en fonction des niveaux d'élagage

Metric	Hardware	ALL	BASIC	DISABLE
E2E p95 (ms)	4070	**9.36**	9.62	9.71
	Pi	**321.89**	346.64	336.4
	i9	**43.43**	62.61	62.98
mAP@50 (%)	4070	**66.32**	66.31	66.31
	Pi	**57.24**	**57.24**	**57.24**
	i9	**66.31**	**66.31**	**66.31**
CPU util (%)	Pi	97.56	97.04	**96.91**
	i9	**70.15**	73.1	71.25
GPU util (%)	4070	**15.15**	19.15	20.69
RAM (MB)	4070	**1037.52**	1038.63	1037.93
	Pi	116.66	119.08	**110.9**
	i9	187.05	**168.63**	168.82
VRAM (MB)	4070	**35.87**	37.4	39.44
Power mean (W)	4070	**43.21**	47.68	48.69
	Pi	**2.66**	2.72	2.72
	i9	64.71	**63.53**	**63.53**

Table 9: Mesures en fonction des niveaux de fusion

Metric	Hardware	fp16	fp32
E2E p95 (ms)	4070	10.08	**9.05**
	Orin	**42.51**	51.99
	Pi	346.74	**323.2**
	i9	58.44	**54.24**
mAP@50 (%)	4070	**66.34**	66.29
	Orin	**66.32**	66.3
	Pi	**57.28**	57.21
	i9	**66.32**	66.29
CPU util (%)	Pi	**95.4**	98.94
	i9	**68.46**	74.54
GPU util (%)	4070	**13.1**	23.56
	Orin	**34.77**	57.33
RAM (MB)	4070	1048.76	**1027.3**
	Pi	**114.72**	116.38
	i9	191.7	**157.97**
VRAM (MB)	4070	**-7.57**	82.71
Power mean (W)	4070	**38.55**	54.5
	Orin	**1.49**	2.22
	Pi	**2.55**	2.85
	i9	**63.56**	64.29

Table 10: Mesures en fonction des niveaux de quantisation

Metric	Hardware	256	320	416	512	640
E2E p95 (ms)	4070	**5.4**	7.08	8.85	11.15	15.34
	Oak	**105.71**	188.24	317.69	379.91	636.45
	Orin	**24.12**	30.29	43.63	59.13	79.08
	Pi	**129.68**	188.09	292.61	426.58	637.9
	i9	**24.94**	34.86	50.85	68.54	102.5
mAP@50 (%)	4070	57.73	62.26	67.14	71.55	**72.91**
	Oak	57.51	62.09	67.42	70.92	**73.1**
	Orin	57.7	62.24	67.18	71.55	**72.89**
	Pi	45.9	51.49	57.88	63.77	**67.18**
	i9	57.71	62.24	67.09	71.55	**72.94**
CPU util (%)	Pi	100.69	98.61	96.94	**94.36**	95.26
	i9	77.42	69.67	**67.81**	73.31	69.29
GPU util (%)	4070	16.16	16.65	**14.37**	17.31	27.15
	Orin	45.98	44.28	45.18	**44.53**	50.3
RAM (MB)	4070	1051.09	1057.88	1051.42	1047.68	**982.06**
	Pi	**82.64**	**82.64**	89.71	106.17	216.58
	i9	**122.49**	132.42	162.55	192.98	263.74
VRAM (MB)	4070	**-9.11**	0.52	3.7	78.38	114.37
Power mean (W)	4070	**43.46**	**43.46**	46.04	48.67	51.0
	Oak	**1.27**	1.46	1.7	1.87	1.88
	Orin	**1.44**	1.63	1.87	2.05	2.28
	Pi	2.79	2.84	2.79	2.54	**2.53**
	i9	**63.32**	63.53	63.62	63.81	65.34

Table 11: Mesures en fonction des niveaux de résolution

Metric	4	5	6	7	8
E2E p95 (ms)	337.97	338.85	319.66	317.47	**314.05**
mAP@50 (%)	**66.21**	**66.21**	**66.21**	**66.21**	**66.21**
Power mean (W)	1.69	**1.59**	1.66	1.63	1.6

Table 12: Mesures en fonction du nombre de SHAVEs