Optimizarea unui algoritm de inteligenta si viziune artificiala

Structura sistemelor de calcul

Autor: Campean Bogdan
Grupa: 30237

Universitatea Tehnica
Cluj-Napoca

Facultatea de Automatica
si Calculatoare

30 decembrie 2025

Cuprins

Rezumat **2**

1 Introducere **3**
1.1 Obiectivele si contextul proiectului 3
1.2 Tehnologii utilizate 3
1.3 Solutiea propusa 4
1.4 Structura documentului 4

2 Fundamentare teoretica **5**
2.1 YOLOv8 5
2.2 ONNX-Open Neural Network Exchange 5
2.3 TensorRT 5
2.4 CUDA 6

3 Proiectare si implementare **7**
3.1 Exportul modelului YOLOv8 in format ONNX mai apoi TensorRT 7
3.2 Implementarea aplicatiei C++ 7

4 Rezultate experimentale **10**
4.1 Configuratia experimentala 10
4.2 Rezultate pe imagini cu masini 10
4.3 Rezultate pe imagini cu persoane 12
4.4 Rezultate comparate 14
4.4.1 Scenariul 1: Detectie vehicule (Masini) 14
4.4.2 Scenariul 2: Detectie persoane (Aglomeratie) 14

5 Concluzii **15**
5.1 Contributii principale 15
5.2 Observatii finale 15
5.3 Dezvoltari viitoare 15

Bibliografie **15**

Anexe **16**

Rezumat

Acest proiect prezinta optimizarea si implementarea modelului YOLOv8 (pentru detectia obiectelor in timp real), utilizand TensorRT pe doua platforme: un Latop cu placa video NVIDIA GeForce RTX 3050 si o placuta NVIDIA Jetson Orin Nano.

Obiective principale:

- Conversia modelului YOLOv8 din PyTorch in ONNX , iar mai apoi in TensorRT
- Optimizarea modelului pentru a rula in file C++
- Evaluarea performantelor dintre cele doua platforme in termeni de viteza, consum energetic si acuratete

Metodologie: Pipeline-ul de lucru include exportul modelului YOLOv8n din PyTorch in format ONNX, conversia in engine TensorRT cu precizie FP16, si rularea intr o fila C++ care integreaza pre-procesare imagine, inferenta asincrona si post-procesare NMS.

Rezultate: Jetson Orin Nano are performata mai buna ca PC-ul din cauza timpilor ridicati de incarcare a engine-ului pe PC.

1 Introducere

Tema principala a proiectului: Alegerea unui algoritm de inteligenta artificiala existent(in pytorch) si optimizarea acestuia pentru plaucta Jetson Orin Nano (pentru a rula file C++) (adica trasare ONNX sau TensorRT).

1.1 Obiectivele si contextul proiectului

Obiectivul proiectului este optimizarea modelului YOLOv8 pentru a rula eficient pe platforma embedded Jetson Orin Nano, utilizand TensorRT pentru accelerare hardware.De asemenea , se va realiza o comparatie intre performantele obtinute pe Jetson Orin Nano si cele pe un PC(laptop mai exact) echipat cu o placa video NVIDIA GeForce RTX 3050.

Jetson Orin Nano este o placuta embedded dezvoltata de NVIDIA, bazata pe arhitectura Ampere, care ofera performante ridicate pentru aplicatii de inteligenta artificiala si viziune computerizata, cu un consum de energie redus.[3]

Pentru acest proiect am ales modelul YOLOv8[1], care este un algoritm de detectie a obiectelor in timp real dezolvtat de Ultralytics.Antrenat pe datasetul COCO.

COCO (Common Objects in Context) este un dataset larg standard utilizat pentru antrenarea si evaluarea modelelor de detectie a obiectelor, continand peste 330.000 de imagini etichetate cu 80 de clase diferite de obiecte.[2]

E de asteptat ca PC-ul sa aiba performante mai bune ca Jetson vorbind strict de timpii de inferenta, insa Jetson Orin Nano ofera un raport performanta/consum energetic mult mai bun, fiind ideal pentru aplicatii embedded si vom vedea ca gestioneaza mai bine incarcarea engine-ului, avand un timp de incarcare mult mai mic decat PC-ul.

OBS: Timpul de inferenta reprezinta timpul necesar ca modelul sa proceseze o singura intrare(imagine) si sa genereze un raw output(predictie bruta) (fara pre/post procesare).

1.2 Tehnologii utilizate

Software si Framework-uri AI

- Yolov8 - Algoritm de detectie a obiectelor in timp real[1]
- PyTorch - Framework in care este antrenat modelul initial
- ONNX - Format intermediar pentru conversia modelelor
- TensorRT - SDK pentru optimizarea si rularea modelelor pe GPU-uri NVIDIA

Limbaje de porgramare

- C++ - Limbaj principal pentru implementarea aplicatiei de inferenta
- Python - Utilizat pentru exportul modelului in format ONNX
- Bash/PowerShell - Scripturi pentru conversia modelelor (mai exact din onnx in tensorrt engine)
- CMake - Fisier de build pentru compilarea aplicatiei C++ (nu e un chiar limbaj de programare, se fac linkurile si setarile de compilare)

Biblioteci

- OpenCV - Preprocesare si post-procesare imagini
- CUDA - Programare paralela pe GPU-uri NVIDIA

Hardware

- NVIDIA GeForce RTX 3050(laptop)
- NVIDIA Jetson Orin Nano[3]

1.3 Solutiea propusa

Solutia propusa implica urmatorii pasi principali:

1. Exportul modelului YOLOv8n din PyTorch in format ONNX
2. Conversia modelului ONNX in engine TensorRT optimizat pentru Jetson Orin Nano si PC
3. Implementarea unei aplicatii C++ care incarca engine-ul TensorRT
4. Rularea aplicatiei pe ambele platforme pe niste imagini de test si masurarea performantelor
5. Analiza rezultatelor obtinute

1.4 Structura documentului

Documentul este structurat in urmatoarele sectiuni:

- **Sectiunea 2 - Fundamentare teoretica**:Unde o sa prezint mai in detaliu tehnologiile utilizate(cele principale nu toate)
- **Sectiunea 3 - Proiectare si implementare**: Unde o sa detaliez procesul de optimizare si arhitectura aplicatiei C++
- **Sectiunea 4 - Rezultate experimentale**: Unde o sa prezint rezultatele obtinute pe ambele platforme si analiza acestora
- **Sectiunea 5 - Concluzii**: Unde o sa sumarizez contributiile proiectului si o sa prezint posibile dezvoltari
- **Sectiunea - Bibliografie**: Unde o sa apara aparea resursele si documentatia consultata
- **Sectiunea - Anexe**: Unde se va include codul sursa si alte resurse relevante pentru proiect(imagini de test, rezultate pe imagini,fisier cmake etc)
- **Sectiunea - Rezumat**: Unde se va regasi un rezumat al intregului proiect

2 Fundamentare teoretica

2.1 YOLOv8

YOLOv8 (You Only Look Once versiunea 8) este un algoritm de detectie a obiectelor in timp real dezvoltat de Ultralytics. Ce poate fii lansat(deployed) pe o gama larga de dispozitive, precum Jetson Orin Nano, alte placi video de la NVIDIA si pe dispozitive cu macOS.[1] Fata da versiunile anterioare, YOLOv8 aduce imbunatatiri semnificative in ceea ce priveste acuratetea si viteza. Una din imbunatartile aduse este de exemplu noul sistem anchor-free care elimina necesitatea definirii ancorelor pentru detectia obiectelor, simplificand procesul de antrenare si imbunatatind performanta pe obiecte de dimensiuni variate.

Detectia cu ancore vs fara ancore:

Ancorele sunt niste dreptunghiuri predefinite care ajuta modelul sa detecteze obiectele in imagini.Algoritmul impartea imaginea intr-o grila si pentru fiecare celula din grila punea automat niste dreptunghiuri de diferite dimensiuni (anocre sau anchor boxes) si apoi modifica (deformeaza) acele dreptuhgiuri pentru a se potrivi pe obiecte. (Predicita = cat de mult sa modifice acele dreptunghiuri predefinite pentru a se potrivi pe obiecte) [4]

Anchor-free schimba modul in care algoritmul detecteaza obiectele „Inovatia principala a detectoarelor de tip **anchor-free** (fara ancore) consta in modul in care acestea formuleaza problema detectiei. In loc sa clasifice si sa rafineze mii de candidati de tip *anchor box* (cutii de ancorare), aceste modele trateaza, de obicei, detectia ca o sarcina de predictie a punctelor sau de regresie. [5]

2.2 ONNX-Open Neural Network Exchange

ONNX este un format fisier open-source (oarecum) standard pentru reprezentarea modelelor de AI si machine learning. Acesta permite interoperabilitatea intre diferite framework-uri de deep learning, cum ar fi PyTorch, TensorFlow si altele. ONNX defineste un set de operatii standardizate si un format de stocare care faciliteaza schimbul de modele intre diferite platforme si tool-uri. Practic ONNX functioneaza ca un pod de legatura intre diferite framework-uri, permitand dezvoltatorilor sa antreneze modele intr-un framework si sa le ruleze intr-altul fara a fi nevoie sa rescrie codul sau sa reconstruiasca modelul de la zero. [6]

2.3 TensorRT

Reprezinta un motor de inferenta high-performance dezvoltat de NVIDIA, optimizat pentru rularea modelelor de deep learning pe GPU-urile NVIDIA. TensorRT ofera o serie de optimizari si tehnici avansate pentru a accelera inferenta modelelor de AI, inclusiv:

- **Layer & Tensor Fusion**: TensorRT analizeaza graful computational si fuzioneaza nodurile (operatiile) adiacente intr-un singur nucleu (kernel) masiv. Aceasta reduce overheadul lansarii kernelurilor si minimizeaza scrierile/citirile intermediare din memoria globala a GPU-ului.
- **Precision Calibration**: TensorRT poate reduce precizia matematica a calculelor de la FP32 (32-bit floating point) la FP16 sau INT8. Pe platforme embedded precum Jetson, utilizarea FP16 (asa cum s-a folosit in acest proiect) reduce consumul de memorie si creste viteza de calcul semnificativ(rezultatele ramanand asemanatoare).
- **Kernel Auto-Tuning**: Testeaza sute de implementari posibile pentru fiecare operatie matematica direct pe placa video conectata si o selecteaza pe cea mai rapida pentru acea arhitectura specifica de GPU.

- **Dynamic Tensor Memory**: Minimizeaza amprenta de memorie (footprint) prin reutilizarea eficienta a memoriei pentru tensorii temporari folositi in timpul inferentei.

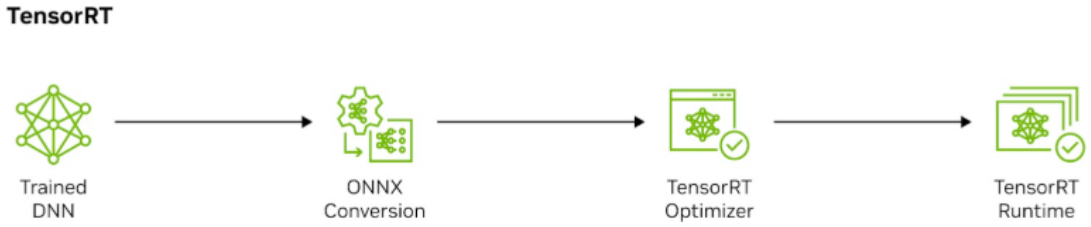

Figura 1: Fluxul de lucru: De la antrenare la TensorRT Runtime(poza de pe site-ul NVIDIA) [7]

2.4 CUDA

CUDA (Compute Unified Device Architecture) este o arhitecutra de calcul paralel dezvolat de NVIDIA care permite programatorilor sa utilizeze GPU-urile NVIDIA pentru a efectua calcule generale (nu doar grafica). In cadrul aplicatiei dezvoltate, API-ul CUDA Runtime a fost utilizat pentru gestionarea "logisticii" dintre procesorul central (CPU - Host) si placa video (GPU - Device)

3 Proiectare si implementare

Primul pas a fost normal alegerea unui alogritm de inteligenta artificiala existent si antrenat in PyTorch (YOLOv8n in cazul de fata), urmat de exportul acestuia in format ONNX. Initial am facut conversia din ONNX in format .engine TensoRT direct in programul C++, insa dura foarte mult pana se construia engine-ul si am decis sa fac conversia intr-un pas separat folosind comanda `trtexec` oferita de TensorRT. Dupa care am implementat aplicatia C++ care incarca engine-ul TensorRT si realizeaza inferenta pe imaginile de test. Am rulat aplicatia pe ambele platforme de mai multe ori si am facut o medie a timpilor obtinuti.

3.1 Exportul modelului YOLOv8 in format ONNX mai apoi TensorRT

Pentru exportul modelului YOLOv8n din PyTorch in format ONNX am folosit urmatorul cod Python:

```python
from ultralytics import YOLO
model_name = 'yolov8n.pt'

model = YOLO(model_name)

model.export(format='onnx', opset=12) #opset=operation set version 12 (compatibilitate cu multe framework-uri)
```

Dupa rularea acestui script, fisierul `yolov8n.onnx` a fost generat in directorul curent.Fisier pe care l-am folosit pentru conversia in TensorRT engine atat pe PC cat si pe Jetson Orin Nano folosind comanda `trtexec --onnx=<pathToModel.onnx> --saveEngine=<pathToOutput.trt>` (de precizat cu –fp16)

Dupa obtinerea fisierul .engine a ramas doar implementarea aplicatiei C++ care sa incarce engine-ul si sa realizeze inferenta pe imaginile de test.

3.2 Implementarea aplicatiei C++

Partile principale ale aplicatiei C++:

- **Crearea unei instante Ilogger**: Pentru a monitoriza evenimentele si erorile importante in timpul executiei.
- **Incarcarea engine-ului TensorRT din fisier**: Citirea fisierului binar .engine si utilizarea interfetei `IRuntime` pentru a deserializa modelul si a reconstrui obiectul `ICudaEngine`.
- **Gestionarea memoriei GPU**: Alocarea bufferelor in VRAM pentru intrare si iesire folosind API-ul CUDA.
- **Pre-procesarea imaginii**: Transformarea imaginii citite cu OpenCV intr-un format compatibil cu reteaua (Blob).
- **Inferenta si Sincronizarea**: Lansarea executiei asincrone si asteptarea finalizarii calculelor pe GPU.
- **Interpretarea matematica a rezultatelor**: Parsarea matricei de iesire, extragerea scorurilor de confidenta si calculul geometric al coordonatelor pentru cutiile de delimitare.
- **Post-procesarea (NMS)**: Filtrarea predictiilor si eliminarea suprapunerilor pentru a obtine rezultatul final.
- **Masurarea performantei**: Utilizarea bibliotecii standard `chrono` pentru a masura (in ms) timpul de executie al fiecarei etape (incarcare, pre-procesare, inferenta, post-procesare).

In continuare, sunt detaliate etapele critice ale implementarii software, evidentiind functiile specifice utilizate in codul sursa.

1. Initializarea si Alocarea Resurselor

Procesul incepe prin deserializarea fisierului `.engine` generat anterior. Acest pas transforma modelul optimizat dintr-un fisier stocat pe disk intr-un obiect utilizabil in memorie. Dupa obtinerea motorului, se creeaza un context de executie (`IExecutionContext`) care gestioneaza starea inferentei.

```
// Citirea fisierului si deserializarea
IRuntime* runtime = createInferRuntime(gLogger);
ICudaEngine* engine = runtime->deserializeCudaEngine(engineData.data(),
↪  engineData.size());

// Crearea contextului de executie
IExecutionContext* context = engine->createExecutionContext();
```

2. Pre-procesarea Datelor si Transferul Host-Device

Imaginea este citita de pe disk, dar nu poate fi trimisa direct retelei. Aceasta trebuie redimensionata la 640x640 pixeli si normalizata (valorile pixelilor impartite la 255 pentru a fi in intervalul [0,1]). Functia `blobFromImage` din OpenCV realizeaza aceste operatii si converteste formatul din BGR in RGB. Ulterior, datele sunt copiate din memoria RAM (Host) in memoria placii video (Device).

```
// Redimensionare si normalizare [0,1]
cv::dnn::blobFromImage(img, blob, 1.0 / 255.0, cv::Size(640, 640), cv::Scalar(0, 0,
↪  0), true, false);

// Alocare memorie GPU si copierea imaginii
cudaMalloc(&buffers[0], inputSize); // Input buffer
cudaMalloc(&buffers[1], outputSize); // Output buffer
cudaMemcpy(buffers[0], blob.ptr<float>(), inputSize, cudaMemcpyHostToDevice);
```

3. Executia Inferentei (Async)

TensorRT este proiectat pentru executie asincrona. Se utilizeaza un `cudaStream` pentru a pune in coada operatia de inferenta. Functia `enqueueV3` lanseaza calculele pe GPU, iar `cudaStreamSynchronize` obliga procesorul sa astepte pana cand placa video termina procesarea, garantand ca rezultatele citite ulterior sunt complete.

```
// Crearea stream-ului si lansarea executiei
cudaStream_t stream;
cudaStreamCreate(&stream);
context->enqueueV3(stream);

// Blocarea CPU pana la finalizarea GPU
cudaStreamSynchronize(stream);
```

4. Post-procesarea si Algoritmul NMS

Rezultatele brute (Raw Output) constau intr-o matrice de 8400 de posibile detectii. Pentru a extrage obiectele reale, se parcurg acesti vectori si se filtreaza detectiile cu un scor de confidenta scazut (sub 0.30). Deoarece modelul poate detecta acelasi obiect de mai multe ori (cutii

suprapuse), se aplica algoritmul **Non-Maximum Suppression (NMS)** pentru a pastra doar detectia optima.

```
// Filtrarea preliminara pe baza pragului de confidenta
if (maxClassScore > 0.30) {
    // Calcul coordonate si adaugare in liste...
    boxes.push_back(cv::Rect(left, top, width, height));
    confidences.push_back(maxClassScore);
}

// Aplicarea NMS pentru eliminarea suprapunerilor
cv::dnn::NMSBoxes(boxes, confidences, 0.30, 0.5, indices);
```

4 Rezultate experimentale

Procesul de testare consta in rularea aplicatiei C++ pe ambele platforme (PC cu RTX 3050 si Jetson Orin Nano) folosind acelasi set de imagini de test.

4.1 Configuratia experimentala

Platforma PC

- **GPU**: NVIDIA GeForce RTX 3050 4GB Laptop
- **CPU**: Intel Core i5-12450H
- **RAM**: 16GB DDR4
- **CUDA**: 12.6
- **TensorRT**: 10.13.3.9
- **OS**: Windows 10

Platforma Jetson Orin Nano

- **GPU**: NVIDIA Ampere (NVIDIA Ampere architecture with 1024 CUDA cores and 32 tensor cores)
- **CPU**: 6-core ARM Cortex-A78AE
- **RAM**: 8GB 128-bit LPDDR5
- **TDP**: 7W - 25W configurabil
- **CUDA**: -
- **TensorRT**: -

4.2 Rezultate pe imagini cu masini

Comparatie teste: Jetson Orin Nano (stanga) vs PC GeForce RTX 3050 (dreapta)

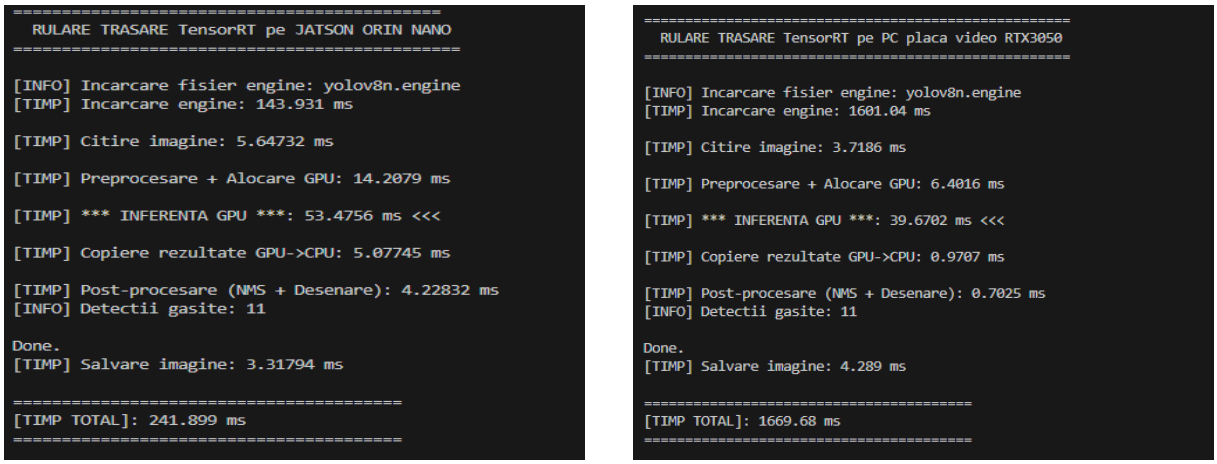

Figura 2: Detectie masini - Imagine 1: Jetson (stanga) vs PC (dreapta)

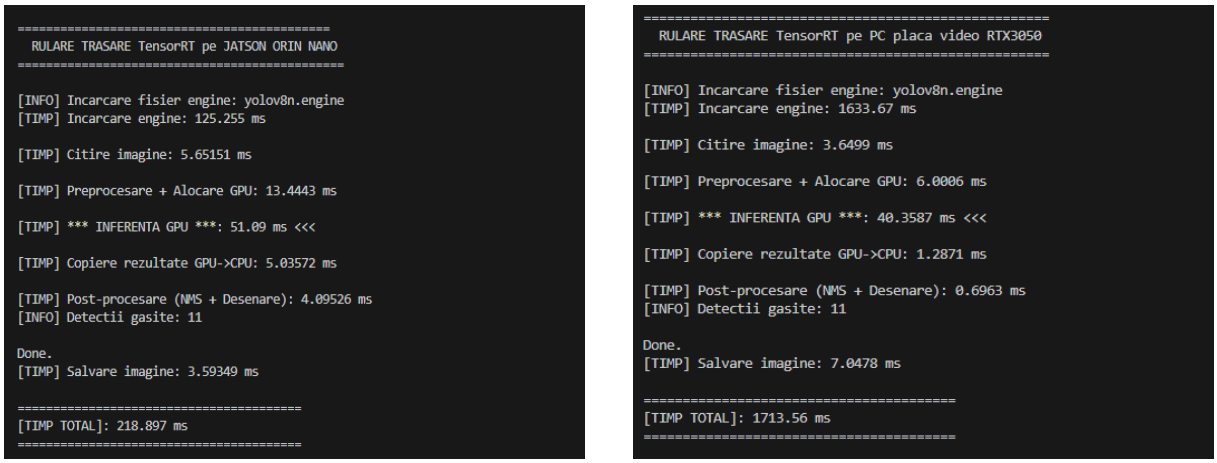

Figura 3: Detectie masini - Imagine 2: Jetson (stanga) vs PC (dreapta)

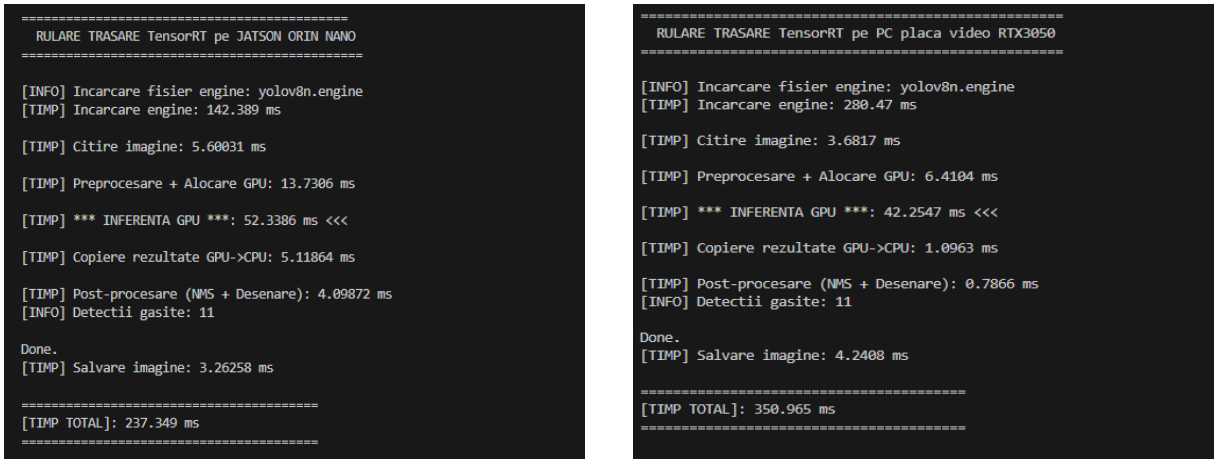

Figura 4: Detectie masini - Imagine 3: Jetson (stanga) vs PC (dreapta)

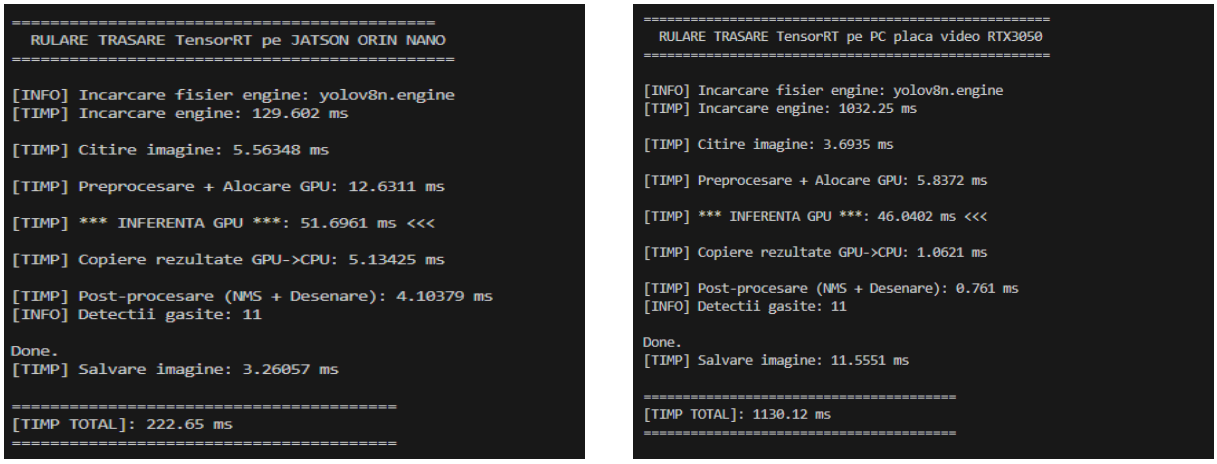

Figura 5: Detectie masini - Imagine 4: Jetson (stanga) vs PC (dreapta)

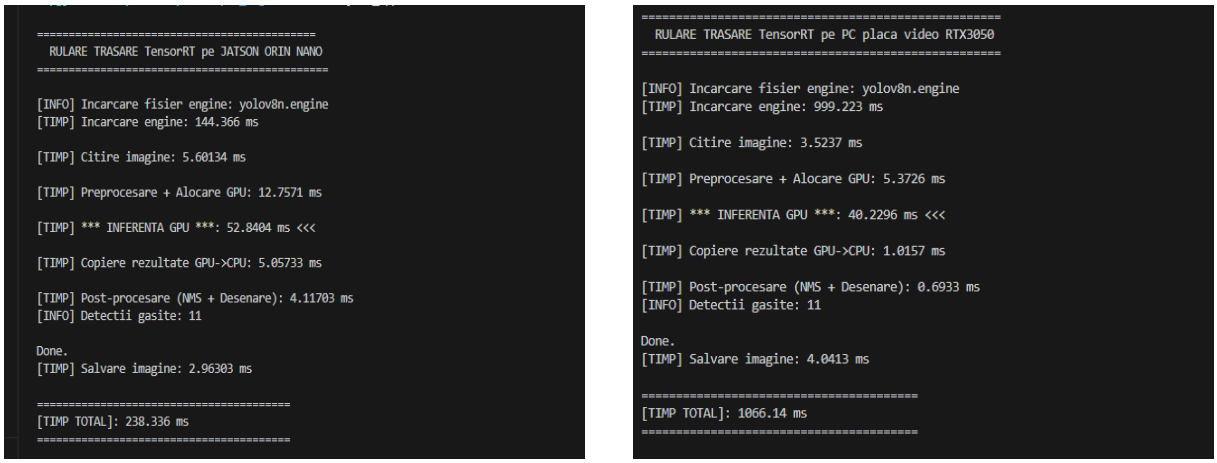

Figura 6: Detectie masini - Imagine 5: Jetson (stanga) vs PC (dreapta)

4.3 Rezultate pe imagini cu persoane

Comparatie teste: Jetson Orin Nano (stanga) vs PC GeForce RTX 3050 (dreapta)

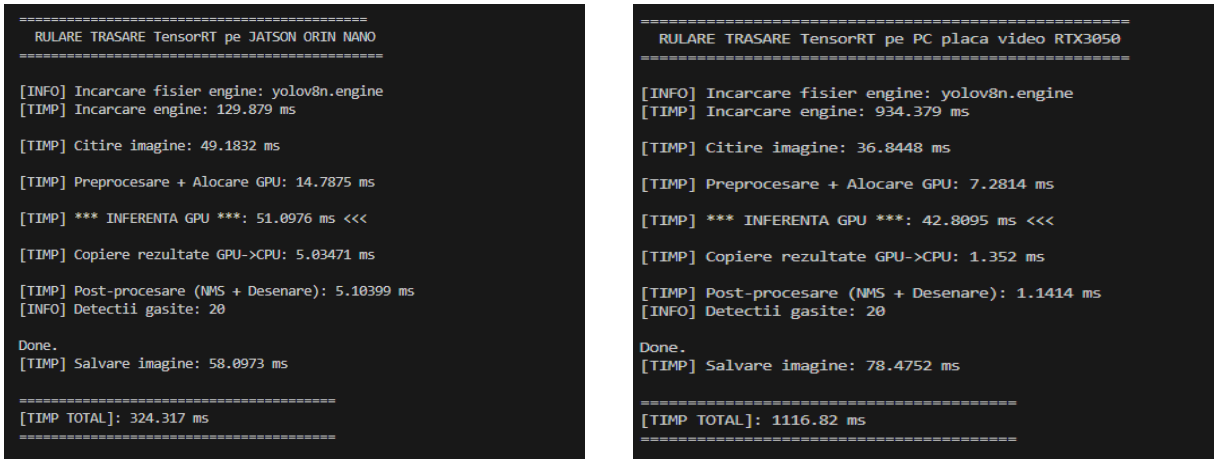

Figura 7: Detectie persoane - Imagine 1: Jetson (stanga) vs PC (dreapta)

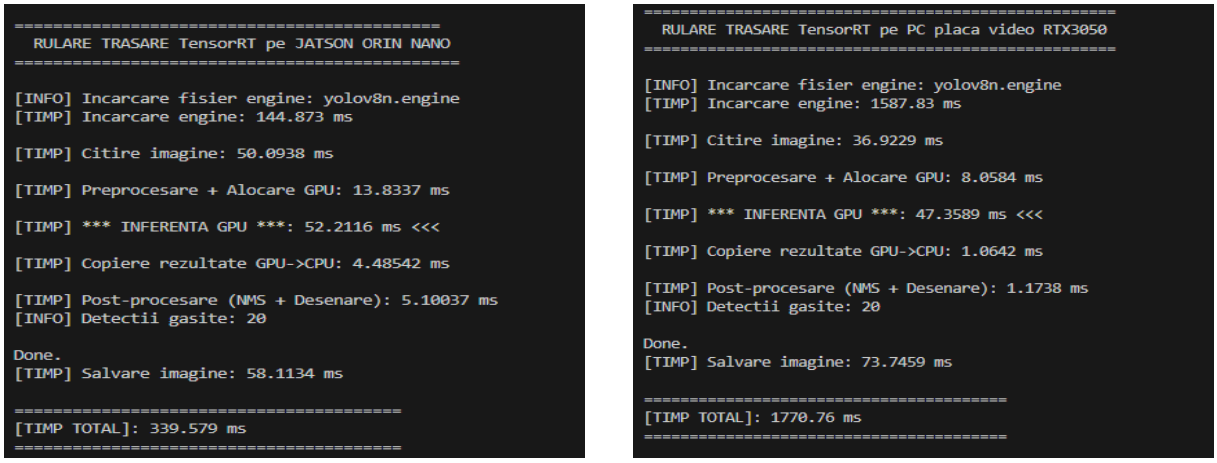

Figura 8: Detectie persoane - Imagine 2: Jetson (stanga) vs PC (dreapta)

```
========================================
  RULARE TRASARE TensorRT pe JATSON ORIN NANO
========================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 142.116 ms

[TIMP] Citire imagine: 49.6744 ms

[TIMP] Preprocesare + Alocare GPU: 15.045 ms

[TIMP] *** INFERENTA GPU ***: 51.7535 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 5.01884 ms

[TIMP] Post-procesare (NMS + Desenare): 5.1478 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 58.9441 ms

====================================
[TIMP TOTAL]: 338.65 ms
====================================
```

```
================================================
  RULARE TRASARE TensorRT pe PC placa video RTX3050
================================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 1591.54 ms

[TIMP] Citire imagine: 38.2573 ms

[TIMP] Preprocesare + Alocare GPU: 7.3927 ms

[TIMP] *** INFERENTA GPU ***: 45.2682 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 0.9906 ms

[TIMP] Post-procesare (NMS + Desenare): 1.0675 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 71.3868 ms

====================================
[TIMP TOTAL]: 1769.88 ms
====================================
```

Figura 9: Detectie persoane - Imagine 3: Jetson (stanga) vs PC (dreapta)

```
========================================
  RULARE TRASARE TensorRT pe JATSON ORIN NANO
========================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 136.018 ms

[TIMP] Citire imagine: 49.3148 ms

[TIMP] Preprocesare + Alocare GPU: 14.0966 ms

[TIMP] *** INFERENTA GPU ***: 53.4114 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 5.17804 ms

[TIMP] Post-procesare (NMS + Desenare): 5.26732 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 58.2006 ms

====================================
[TIMP TOTAL]: 333.577 ms
====================================
```

```
================================================
  RULARE TRASARE TensorRT pe PC placa video RTX3050
================================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 333.283 ms

[TIMP] Citire imagine: 38.7673 ms

[TIMP] Preprocesare + Alocare GPU: 8.0107 ms

[TIMP] *** INFERENTA GPU ***: 48.432 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 1.3772 ms

[TIMP] Post-procesare (NMS + Desenare): 1.1086 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 73.9081 ms

====================================
[TIMP TOTAL]: 518.216 ms
====================================
```

Figura 10: Detectie persoane - Imagine 4: Jetson (stanga) vs PC (dreapta)

```
========================================
  RULARE TRASARE TensorRT pe JATSON ORIN NANO
================================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 146.671 ms

[TIMP] Citire imagine: 49.1565 ms

[TIMP] Preprocesare + Alocare GPU: 13.7081 ms

[TIMP] *** INFERENTA GPU ***: 53.3743 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 5.09285 ms

[TIMP] Post-procesare (NMS + Desenare): 5.07007 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 58.334 ms

====================================
[TIMP TOTAL]: 342.295 ms
====================================
```

```
================================================
  RULARE TRASARE TensorRT pe PC placa video RTX3050
================================================

[INFO] Incarcare fisier engine: yolov8n.engine
[TIMP] Incarcare engine: 239.712 ms

[TIMP] Citire imagine: 35.7418 ms

[TIMP] Preprocesare + Alocare GPU: 7.4929 ms

[TIMP] *** INFERENTA GPU ***: 54.7607 ms <<<

[TIMP] Copiere rezultate GPU->CPU: 1.1782 ms

[TIMP] Post-procesare (NMS + Desenare): 1.228 ms
[INFO] Detectii gasite: 20

Done.
[TIMP] Salvare imagine: 71.5528 ms

====================================
[TIMP TOTAL]: 425.04 ms
====================================
```

Figura 11: Detectie persoane - Imagine 5: Jetson (stanga) vs PC (dreapta)

4.4 Rezultate comparate

In aceasta sectiune sunt prezentate rezultatele medii obtinute in urma a 5 rulari consecutive pentru fiecare imagine. Timpii sunt exprimati in milisecunde (ms).

4.4.1 Scenariul 1: Detectie vehicule (Masini)

In tabelul de mai jos sunt detaliati timpii de procesare pentru imaginile in care predomina vehiculele (11 detectii per imagine).

Etapa de procesare	**Jetson Orin Nano**	**PC (RTX 3050)**
Incarcare engine	137.11 ms	1109.33 ms
Citire imagine	5.61 ms	3.65 ms
Preprocesare + Alocare	13.35 ms	6.00 ms
Inferenta GPU (Pura)	**52.29 ms**	**41.71 ms**
Copiere GPU $\rightarrow$ CPU	5.08 ms	1.09 ms
Post-procesare (NMS)	4.13 ms	0.73 ms
Salvare imagine	3.28 ms	6.23 ms
TIMP TOTAL	**231.83 ms**	**1186.09 ms**

Tabela 1: Comparatie timpi de executie - Scenariul Masini

4.4.2 Scenariul 2: Detectie persoane (Aglomeratie)

Pentru acest scenariu s-au folosit imagini complexe cu grupuri de persoane (20 detectii per imagine).

Etapa de procesare	**Jetson Orin Nano**	**PC (RTX 3050)**
Incarcare engine	139.91 ms	937.35 ms
Citire imagine	49.48 ms	37.31 ms
Preprocesare + Alocare	14.30 ms	7.65 ms
Inferenta GPU (Pura)	**52.37 ms**	**47.73 ms**
Copiere GPU $\rightarrow$ CPU	4.96 ms	1.19 ms
Post-procesare (NMS)	5.14 ms	1.14 ms
Salvare imagine	58.34 ms	73.82 ms
TIMP TOTAL	**335.69 ms**	**1120.14 ms**

Tabela 2: Comparatie timpi de executie - Scenariul Persoane

Observatii: Se remarca faptul ca timpul de incarcare al engine-ului pe PC este semnificativ mai mare din cauza overhead-ului introdus de sistemul de operare si drivere, in timp ce pe Jetson incarcarea este optimizata. La nivel de inferenta pura, PC-ul este cu aproximativ 10 ms mai rapid.

5 Concluzii

5.1 Contributii principale

Acest proiect a demonstrat cu succes:

- Implementarea completa a unui pipeline de optimizare YOLOv8 cu TensorRT
- Compararea performantelor pe platforme desktop vs. embedded
- Analiza compromisurilor intre viteza si eficienta energetica
- Dezvoltarea unei aplicatii C++ de inferenta in timp real

5.2 Observatii finale

Alegerea platformei depinde de context:

- **Jetson Orin Nano**: Ideal pentru aplicatii embedded cu constrangeri energetice (drone, robotica, IoT)
- **PC RTX 3050**: Optim pentru aplicatii desktop cu cerinte de throughput ridicat

Beneficiile TensorRT:

- Accelerare semnificativa fata de PyTorch
- Optimizari FP16 fara pierdere notabila de acuratete
- Integrare eficienta in aplicatii C++ de productie

5.3 Dezvoltari viitoare

- Testare cu precizie INT8 pentru eficienta suplimentara
- Integrare cu streaming video in timp real (in loc de imagini statice)
- Optimizare pentru modele YOLO mai noi (YOLOv9, YOLOv10)

Bibliografie

Bibliografie

[1] **YOLOv8 Documentation:** `https://docs.ultralytics.com/`

[2] **COCO Dataset Documentation:** `https://docs.ultralytics.com/datasets/detect/coco/`

[3] **NVIDIA Jetson Orin Nano specs:**`https://nvdam.widen.net/s/zkfqjmtds2/jetson-orin-datasheet-nano-developer-kit-3575392-r2`

[4] **Explicatie sistem anchore:** `https://www.ultralytics.com/glossary/anchor-based-detectors`

[5] **Anchor-free detectors explanation:** `https://www.ultralytics.com/glossary/anchor-free-detectors`

[6] **ONNX documentaite:** `https://github.com/onnx/onnx`

[7] **NVIDIA TensorRT:** `https://developer.nvidia.com/tensorrt`

[8] **NVIDIA TensorRT C++ API Documentation:** `https://docs.nvidia.com/deeplearning/tensorrt/latest/inference-library/c-api-docs.html`

[9] **NVIDIA TensorRT Library :** https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-1040/api/c_api/index.html

Anexe

Cod Sursă - Implementare TensorRT YOLOv8

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <opencv2/opencv.hpp>
#include <NvInfer.h>
#include <cuda_runtime_api.h>
#include <chrono>

using namespace nvinfer1;

// Logger simplu obligatoriu pentru API
class Logger : public ILogger {
    void log(Severity severity, const char* msg) noexcept override {
        if (severity <= Severity::kWARNING) { // Afisam doar erorile si avertismentele importante
            std::cout << "[TRT] " << msg << std::endl;
        }
    }
};

Logger gLogger;

// Functie citire fisier binar
std::vector<char> loadEngineFile(const std::string& fileName) {
    std::ifstream file(fileName, std::ios::binary | std::ios::ate); // deschide fisierul la sfarsit ate=at end
    if (!file.good()) {// verifica daca fisierul s-a deschis corect
        std::cerr << "Eroare citire fisier: " << fileName << std::endl; // mesaj eroare
        return {};
    }
    size_t size = file.tellg();// obtine dimensiunea fisierului (cursor la final fisier aici)
    file.seekg(0, std::ios::beg);// muta cursor la inceput
    std::vector<char> buffer(size); // aloca buffer de dimensiunea fisierului
    file.read(buffer.data(), size); // citeste tot continutul in buffer
    return buffer;
}

// Constante YOLOv8
const int INPUT_W = 640;
const int INPUT_H = 640;
const int NUM_CLASSES = 80; //tipul de obiecte din COCO pe care a fost antrenat modelul
//0= persoana, 1= bicicleta, 2= masina, etc.
//COCO=Common Objects in Context

```

```
44  int main() {
45      std::cout << "\n=====================================================" <<
        ↪  std::endl;
46      std::cout << "  RULARE TRASARE TensorRT pe PC placa video RTX3050" << std::endl;
47      std::cout << "=====================================================\n" <<
        ↪  std::endl;
48
49      auto total_start = std::chrono::high_resolution_clock::now();
50
51      // Pas 1: Citim fisierul
52      auto t1 = std::chrono::high_resolution_clock::now();
53      std::string engineFile = "yolov8n.engine";
54      std::cout << "[INFO] Incarcare fisier engine: " << engineFile << std::endl;
55      std::vector<char> engineData = loadEngineFile(engineFile); //apel functie citire
        ↪  fisier
56      if (engineData.empty()) return -1;
57
58      // Pas 2: Cream Runtime-ul
59      // "createInferRuntime" returneaza un raw pointer catre IRuntime
60      IRuntime* runtime = createInferRuntime(gLogger); //parametru clasa logger pt
        ↪  gestionare erori
61      if (!runtime) {
62          std::cerr << "Eroare la crearea Runtime!" << std::endl;
63          return -1;
64      } // rol de deserializare a engine-ului ( este o instanta a motorului de
        ↪  inferenta)
65
66      // Pas 3: Deserializam Engine-ul
67      ICudaEngine* engine = runtime->deserializeCudaEngine(engineData.data(),
        ↪  engineData.size()); //metoda de deserializare din runtime
68      if (!engine) {
69          std::cerr << "Eroare la deserializarea Engine!" << std::endl;
70          // cleanup pointeri
71          delete runtime;  //delete cheama destructorul si elibereaza memoria
72          return -1;
73      }
74      // Pas 4: Cream Contextul de Executie
75      // se creaza un context de executie din engine ca un proces separat de inferenta
76      // aloca resursele necesare pentru executie
77      // contextul e ca un thread de executie pentru engine
78      // se aloca memorie pentru tensori, se seteaza pointeri, etc
79      IExecutionContext* context = engine->createExecutionContext();
80      if (!context) {
81          std::cerr << "Eroare la crearea Contextului!" << std::endl;
82          delete engine;
83          delete runtime;
84          return -1;
85      }
86      auto t2 = std::chrono::high_resolution_clock::now();
87      std::cout << "[TIMP] Incarcare engine: " << std::chrono::duration<double,
        ↪  std::milli>(t2 - t1).count() << " ms\n" << std::endl;
88
89
90
91
92      // === 2. PREGATIRE DATE (OpenCV & CUDA) ===
```

```
93      auto t3 = std::chrono::high_resolution_clock::now();
94      //cv::Mat img = cv::imread("cars.jpg");
95      cv::Mat img = cv::imread("test.jpg");
96      if (img.empty()) {
97          std::cerr << "Imagine lipsa!" << std::endl; //cerr print mesaj eroare
98          // cleanup pointeri
99          delete context;
100         delete engine;
101         delete runtime;
102         return -1;
103     }
104     auto t4 = std::chrono::high_resolution_clock::now();
105     std::cout << "[TIMP] Citire imagine: " << std::chrono::duration<double,
        ↪  std::milli>(t4 - t3).count() << " ms\n" << std::endl;
106
107
108
109     auto t5 = std::chrono::high_resolution_clock::now();
110     cv::Mat blob; //blob=Binary Large Object
111     //o matrice 4d care contine imaginea preprocesata
112     //N(Number/Batch Size), C(Channels), H(Height), W(Width)\
113     //1x3x640x640
114     cv::dnn::blobFromImage(img, blob, 1.0 / 255.0, cv::Size(INPUT_W, INPUT_H),
        ↪  cv::Scalar(0, 0, 0), true, false);
115     //1.0/255.0  factor de scalare pentru normalizare la [0,1]
116     //cv::size(INPUT_W, INPUT_H)=dimensiunea la care se redimensioneaza imaginea
        ↪  primita
117     //true=converteste BGR in RGB //bgr=blue green red spatiul de culoare default in
        ↪  OpenCV
118     //false=nu face crop imagine //nu taie din imagine
119     // cv::Scalar(0,0,0)=valoarea medie scazuta din fiecare canal (aici 0 deci nu se
        ↪  scade nimic)
120
121     void* buffers[2];
122     // Calcul dimensiuni
123     int inputSize = 1 * 3 * INPUT_H * INPUT_W * sizeof(float);//dimensiune input
124     // Output: [1, 84, 8400] standard pentru yolov8
125     int outputElements = 1 * (4 + NUM_CLASSES) * 8400;
126     int outputSize = outputElements * sizeof(float);
127
128     // Alocare GPU pt input si output
129     // mutare date din memoria CPU in memoria GPU
130     cudaMalloc(&buffers[0], inputSize);
131     cudaMalloc(&buffers[1], outputSize);
132
133     // Copiere imagine(blob) pe GPU
134     cudaMemcpy(buffers[0], blob.ptr<float>(), inputSize, cudaMemcpyHostToDevice);
135     //buffers[0]=adresa buffer input pe GPU
136     //blob.ptr<float>()=pointer la datele din blob (imaginea preprocesata)
137     //inputSize=dimensiunea datelor de copiat
138     //cudaMemcpyHostToDevice=directia copierei (CPU->GPU)
139     auto t6 = std::chrono::high_resolution_clock::now();
140     std::cout << "[TIMP] Preprocesare + Alocare GPU: " <<
        ↪  std::chrono::duration<double, std::milli>(t6 - t5).count() << " ms\n" <<
        ↪  std::endl;
141
```

```
142     // === 3. INFERENTA ===
143
144     // Setam pointerii catre bufferele GPU in Context
145     auto t7 = std::chrono::high_resolution_clock::now();
146     context->setInputTensorAddress("images", buffers[0]);
147     context->setTensorAddress("output0", buffers[1]);
148     // "images" si "output0" sunt numele tensorilor definite in modelul YOLOv8
149     // buffers[0] este inputul, buffers[1] este outputul
150     // mai ok cu SetOutputTensorAddress in loc de SetTensorAddress, dar ambele
        ↪ functioneaza
151     // depinde de versiunea TensorRT
152
153     // Cream stream CUDA pentru executie asincrona
154     // cudaStream_t = o coada de comenzi care ruleaza asincron pe GPU
155     // un to do list pentru GPU de la CPU
156     cudaStream_t stream;
157     cudaStreamCreate(&stream);
158
159     // Lansam executia
160     context->enqueueV3(stream); //lansare stream asincrona
161
162     // Asteptam sa termine GPU-ul
163     cudaStreamSynchronize(stream); //blocheaza CPU pana cand toate operatiile din
        ↪ stream sunt terminate
164     //altfel am avea junk in output (pentru ca e asincron)
165
166     auto t8 = std::chrono::high_resolution_clock::now();
167     std::cout << "[TIMP] *** INFERENTA GPU ***: " << std::chrono::duration<double,
        ↪ std::milli>(t8 - t7).count() << " ms <<<\n" << std::endl;
168
169     // === 4. RECUPERARE REZULTATE ===
170     auto t9 = std::chrono::high_resolution_clock::now();
171     std::vector<float> cpuOutput(outputElements);
172     //outputElements=dimensiunea output-ului in float-uri
173     cudaMemcpy(cpuOutput.data(), buffers[1], outputSize, cudaMemcpyDeviceToHost);
174     //cpuOutput.data()=pointer la datele din vectorul cpuOutput unde se vor copia
        ↪ rezultatele
175     //buffers[1]=adresa buffer output pe GPU
176     //outputSize=dimensiunea datelor de copiat
177     //cudaMemcpyDeviceToHost=directia copierei (GPU->CPU)
178     auto t10 = std::chrono::high_resolution_clock::now();
179     std::cout << "[TIMP] Copiere rezultate GPU->CPU: " <<
        ↪ std::chrono::duration<double, std::milli>(t10 - t9).count() << " ms\n" <<
        ↪ std::endl;
180
181     // === 5. POST-PROCESARE (Matematica YOLO) ===
182     //prelucrarea rezultatelor brute pentru a obtine detectiile finale
183     auto t11 = std::chrono::high_resolution_clock::now();
184     std::vector<int> classIds; //vector de id-uri de clase detectate
185     std::vector<float> confidences; //vector de scoruri de incredere
186     std::vector<cv::Rect> boxes; //vector de dreptunghiuri pt boxele de delimitare
187
188     //matricea output este [1, 84, 8400]
189     //84-4 coordonate + 80 clase (x_center, y_center, width, height, class0,
        ↪ class1,...class79)
190     //8400- numar predictii
```

```
191     int rows = 8400;//numar predictii
192     float x_factor = (float)img.cols / INPUT_W; //factor scalare latime
193     float y_factor = (float)img.rows / INPUT_H; //factor scalare inaltime
194     //poza originala 1920x1080 etc. , input model 640x640 se scalaeaza inapoi la dim
        ↪  originala
195     float* data = cpuOutput.data(); //pointer la datele output-ului
196     //anchor free pt YOLOv8
197     for (int i = 0; i < rows; ++i) { //parcurgem fiecare ancorare
198         float* classes_scores = data + 4 * rows + i; //pointer la scorurile claselor
            ↪  pentru ancorarea i
199         float maxClassScore = 0.0; //initializam scor maxim clasa
200         int maxClassId = -1; //initializam id clasa maxima
201
202
203         // Cautam clasa cu scor maxim pentru ancora curenta
204         // reprezinta obiectul detectat cu cea mai mare probabilitate
205         // adica din cele 80 de clase posibile daca clasa 2 de ex are cel mai mare
            ↪  scor probabil obiectul apartine clasei 2
206         for (int c = 0; c < NUM_CLASSES; ++c) { //parcurgem fiecare clasa
207             float score = data[(4 + c) * rows + i]; //accesam scorul clasei c pentru
                ↪  ancorarea i
208             if (score > maxClassScore) {//daca scorul curent e mai mare decat maximul
                ↪  gasit
209                 maxClassScore = score;//actualizam scorul maxim
210                 maxClassId = c; //actualizam id clasa maxima
211             }
212         }
213         // ia doar clasele cu scor peste un prag mai mare decat cel setat
214         if (maxClassScore > 0.30) { // Prag de confidenta acceptabil
215             // Extragem coordonatele (cx, cy, w, h)
216             float cx = data[0 * rows + i];
217             float cy = data[1 * rows + i];
218             float w = data[2 * rows + i];
219             float h = data[3 * rows + i];
220
221             // calculam coordonatele casetei de delimitare in dimensiunile originale
                ↪  ale imaginii (boxul de pe imagine)
222             int left = int((cx - 0.5 * w) * x_factor);
223             int top = int((cy - 0.5 * h) * y_factor);
224             int width = int(w * x_factor);
225             int height = int(h * y_factor);
226
227             // stocam rezultatele
228             boxes.push_back(cv::Rect(left, top, width, height)); //cv::Rect=clasa
                ↪  OpenCV pentru dreptunghiuri
229             confidences.push_back(maxClassScore);
230             classIds.push_back(maxClassId);
231         }
232     }
233     //cand parcurg predictiile, pentru fiecare predictie gasesc clasa cu scorul maxim
234     //cand fac boxul(cu cv::Rect) pt predictiile apropiate pot avea boxuri suprapuse
235     //de aceea aplic NMS (Non-Maximum Suppression) pentru a elimina suprapunerile
236     std::vector<int> indices;
237     cv::dnn::NMSBoxes(boxes, confidences, 0.30, 0.5, indices); //deseneaza doar
        ↪  dreptunghiuile dupa NMS adica cele cu scorul cel mai mare si elimina
        ↪  suprapunerile
```

```
238     //NMSBoxes(vector de boxe, vector de scoruri, prag scor, prag NMS, vector de
        ↪  indici rezultati)
239      for (int idx : indices) {
240         cv::Rect box = boxes[idx];
241         cv::rectangle(img, box, cv::Scalar(0, 255, 0), 2);//deseneaza dreptunghi pe
            ↪  imagine (verde, grosime 2)
242 
243         // Cream label-ul cu ID clasa si scor de confidenta
244         // ex: "2: 0.87" inseamna clasa 2 (masina) cu 87% confidenta
245         std::string label = std::to_string(classIds[idx]) + ": " +
            ↪  std::to_string(confidences[idx]).substr(0, 4);
246         // std::to_string(confidences[idx]).substr(0, 4) = ia primele 4 caractere din
            ↪  scor (ex: 0.87 in loc de 0.876543)
247 
248         // Punem textul deasupra boxului (y - 5 pixeli deasupra coltului stanga-sus)
249         cv::putText(img, label, cv::Point(box.x, box.y - 5), cv::FONT_HERSHEY_SIMPLEX,
            ↪  0.5, cv::Scalar(0, 255, 0), 2);
250         // cv::Point(box.x, box.y - 5) = pozitie text (cu 5 pixeli mai sus decat
            ↪  boxul)
251         // cv::FONT_HERSHEY_SIMPLEX = font standard
252         // 0.5 = marime font
253         // cv::Scalar(0, 255, 0) = culoare verde (BGR format)
254         // 2 = grosime text
255     }
256     auto t12 = std::chrono::high_resolution_clock::now();
257     std::cout << "[TIMP] Post-procesare (NMS + Desenare): " <<
        ↪  std::chrono::duration<double, std::milli>(t12 - t11).count() << " ms" <<
        ↪  std::endl;
258     std::cout << "[INFO] Detectii gasite: " << indices.size() << "\n" << std::endl;
259 
260     auto t13 = std::chrono::high_resolution_clock::now();
261     cv::imwrite("result_test.jpg", img); //salveaza imaginea rezultata
262     std::cout << "Done." << std::endl;
263     auto t14 = std::chrono::high_resolution_clock::now();
264     std::cout << "[TIMP] Salvare imagine: " << std::chrono::duration<double,
        ↪  std::milli>(t14 - t13).count() << " ms\n" << std::endl;
265 
266     // === 6. CURATARE MANUALA ===
267 
268     // Eliberam resursele CUDA
269     cudaStreamDestroy(stream);
270     cudaFree(buffers[0]);
271     cudaFree(buffers[1]);
272 
273     // Eliberam obiectele TensorRT manual in ordine inversa crearii
274     // In versiunile moderne C++ API, se foloseste delete
275     delete context;
276     delete engine;
277     delete runtime;
278 
279     auto total_end = std::chrono::high_resolution_clock::now();
280     std::cout << "========================================" << std::endl;
281     std::cout << "[TIMP TOTAL]: " << std::chrono::duration<double,
        ↪  std::milli>(total_end - total_start).count() << " ms" << std::endl;
282     std::cout << "========================================\n" << std::endl;
283 
```

```
284         return 0;
285     }
```