Accelerating DNNs on Jetson/Desktop NVIDIA GPUs/DLA

Justin Davis

PhD Student at Colorado School of Mines

Table of Contents

Introduction

Personal and Lab Introduction

- Third year PhD student under Dr. Mehmet Belviranli
- Started undergraduate studying mining engineering
- Research interests
  - Efficient machine learning inference
  - Object detection & multi-object tracking
  - Edge/heterogenous/accelerated computing
- Experience with
  - NVIDIA Jetson platform
  - Luxonis OAK cameras
  - Raspberry Pi
- Lab Members
  - 3 PhD students
  - 2 Masters with thesis students
  - Undergraduate students
- Research interests
  - Diversely heterogeneous architectures
  - Performance & resource modeling
  - AI acceleration
  - Autonomous computing
  - Parallel programming paradigms
  - Runtime systems
  - Large-scale superconducting computers
  - Next generation transistors

Research Work

Current/Past Work

Context-aware Multi-Model Object Detection for Diversely Heterogeneous Compute Systems

- Published DATE24 – Won Outstanding Paper in ASD

HARNESS: Holistic Resource Management for Diversely Scaled Edge Cloud Systems

- Accepted ICS25

Priority-based Fast Multi-Object Tracking on Multi-Accelerator Systems

- Submitted MOBICOM25

Planning for Shared Resources in Robot Motion and Heterogeneous Computational Scheduling

- Ongoing

Context-aware Multi-Model Object Detection for Diversely Heterogeneous Compute Systems

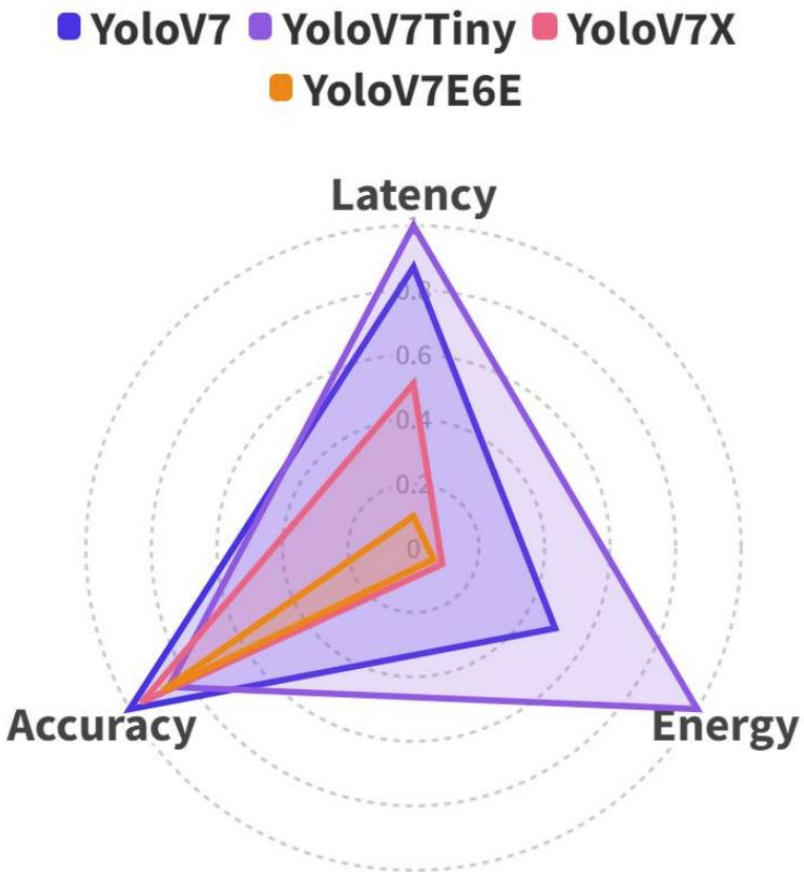
Justin Davis
Mehmet E. Belviranli

Best Paper in Autonomous System Design – DATE24

Slides adapted from DATE24 presentation

Object Detection on SoCs

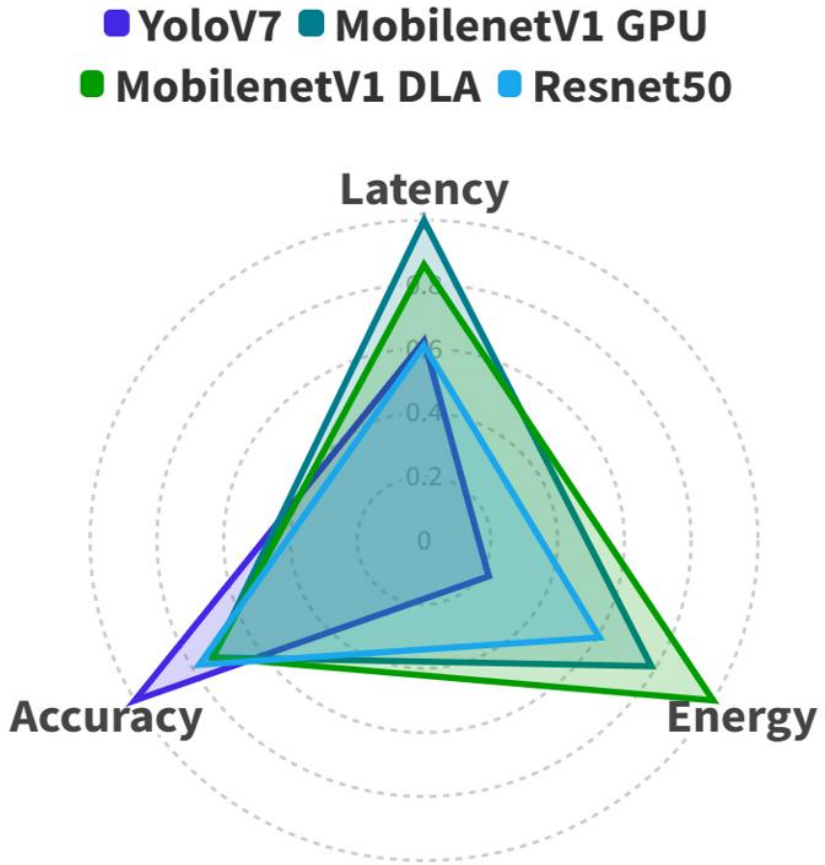
- Smaller/larger parameterizations
  - Allow accuracy/latency trade off between models
  - Larger models on edge platforms see increased latency and power draw
- Inter-Model Relationships
  - Strict monotonic relationships between energy, accuracy, and latency



Bigger is better

Object Detection using Multiple Models + Multiple Accelerators

- Running DNNs on multiple accelerators:
  - Adds scheduling complexity
  - Enables energy, accuracy, and latency tradeoffs
- Using multiple DNN architectures
  - Remove strict monotonic relationships

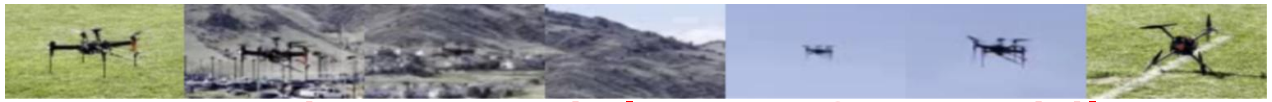


Bigger is better

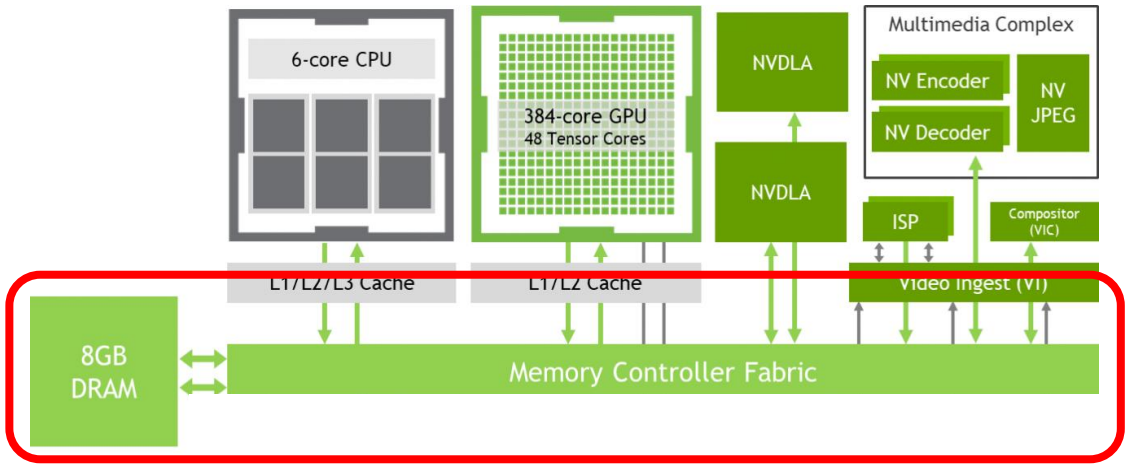
Challenges

1. We need to determine context at runtime.
2. We need to assess the current accuracy of models based purely on runtime context.
3. How many models we can load at once is restricted by the shared memory system.
4. We need to choose models without true knowledge of their prediction strength.

Where are we within our environment?



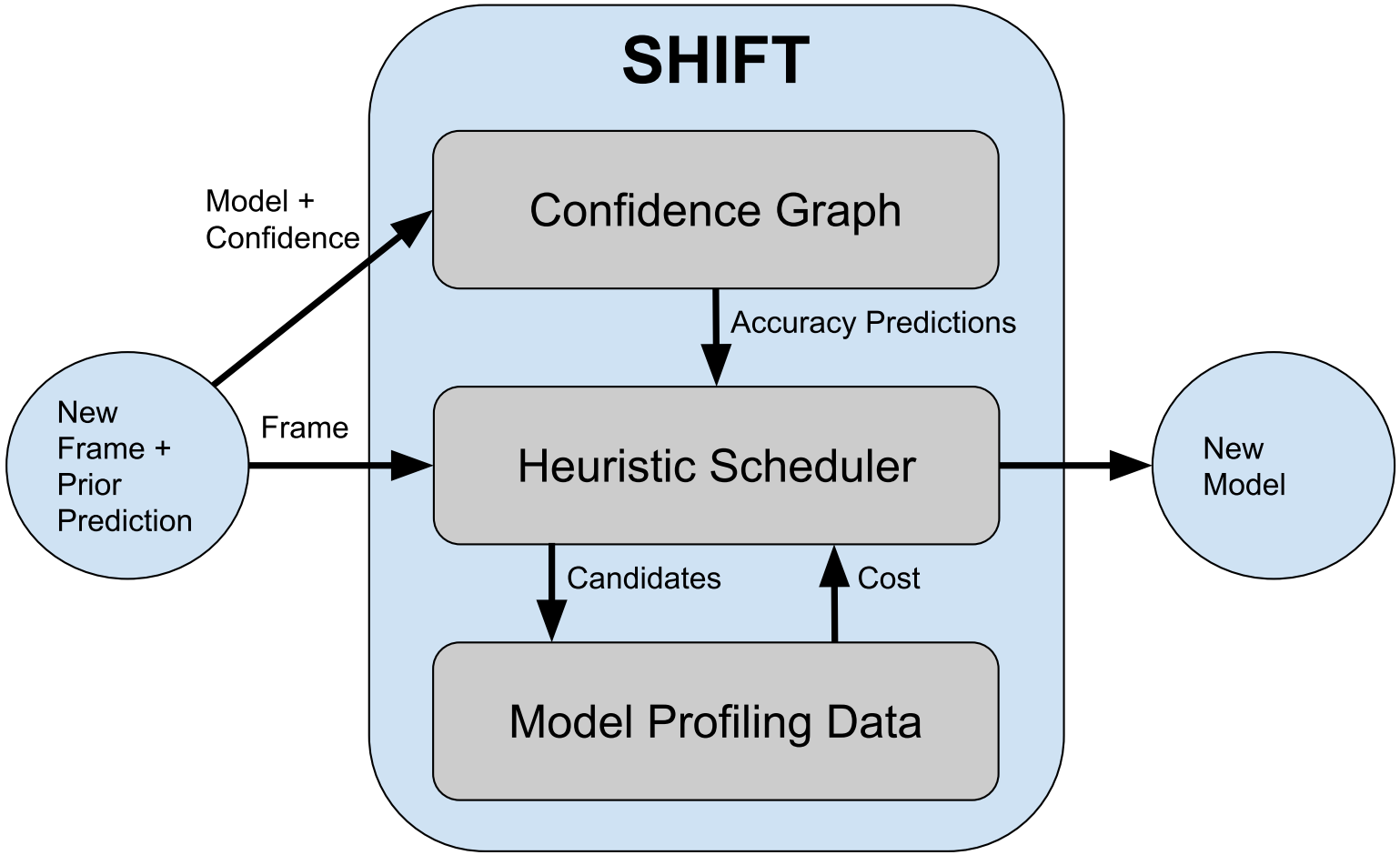
How does model X perform while the drone is here?



Shared memory limits individual capacity

Overview of SHIFT

- Model Characterization
  - Identify key traits of each model
  - Construct confidence graph
- SHIFT Scheduler
  - Context detection
  - Heuristic scheduler



Overall Methodologies

Methodology	IoU	Time (s)	Energy (J)	Success Rate	Non-GPU	Model Swaps	Pairs Used
Marlin	0.614	0.132	1.201	74.0%	0%	0	1
Marlin Tiny	0.529	0.036	0.33	64.0%	0%	0	1
SHIFT	0.598	0.047	0.262	72.2%	68.7%	42	4.3
Oracle E	0.535	0.025	0.144	76.0%	31.5%	94	6.7
Oracle A	0.657	0.108	1.423	76.0%	44.9%	409	12.3
Oracle L	0.522	0.025	0.169	76.0%	11.3%	112	6.8

Maintained good results with fewer than half of the swaps and fewer allocated models than oracle methods.

7.5x Energy usage improvement
2.8x Latency improvement
0.97x Accuracy performance
vs. YoloV7 on GPU

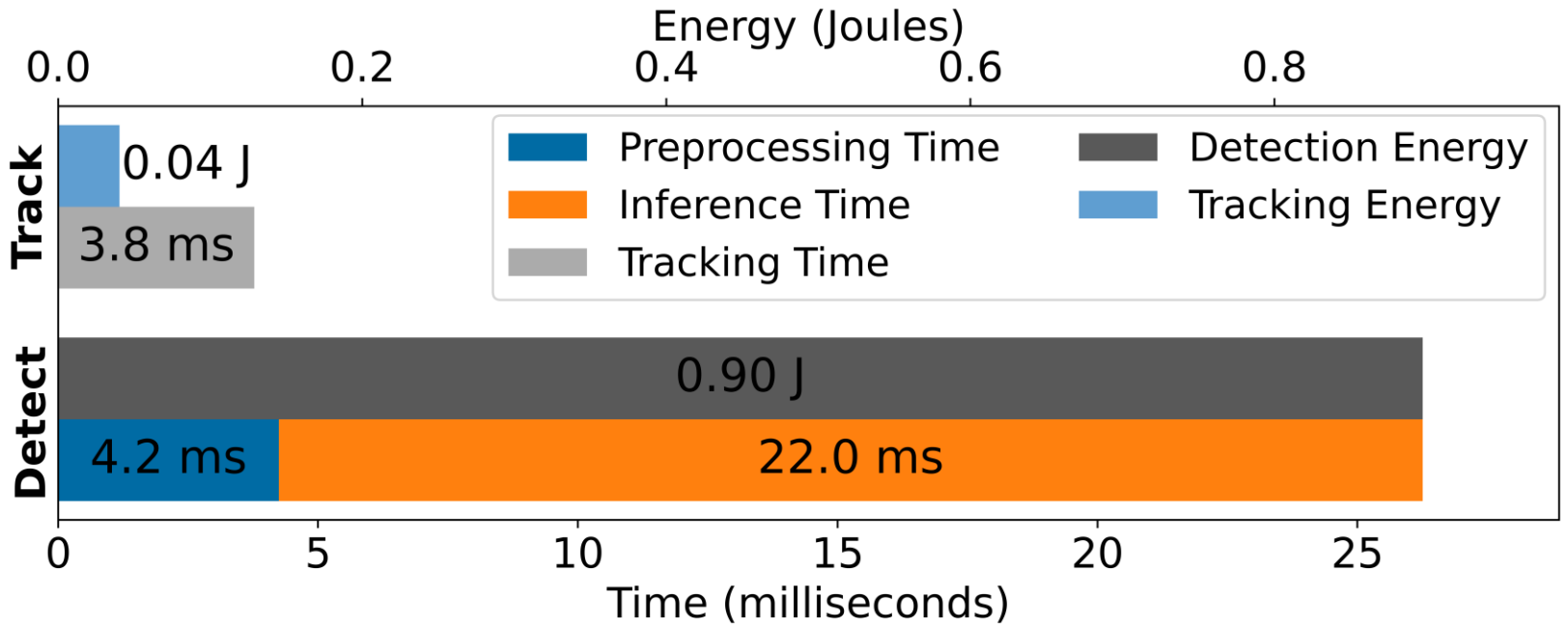
Priority-based Fast Multi-Object Tracking on Multi-Accelerator Systems

Justin Davis
Ismet Dagli
Mehmet E. Belviranli

Under Submission at MOBICOM25

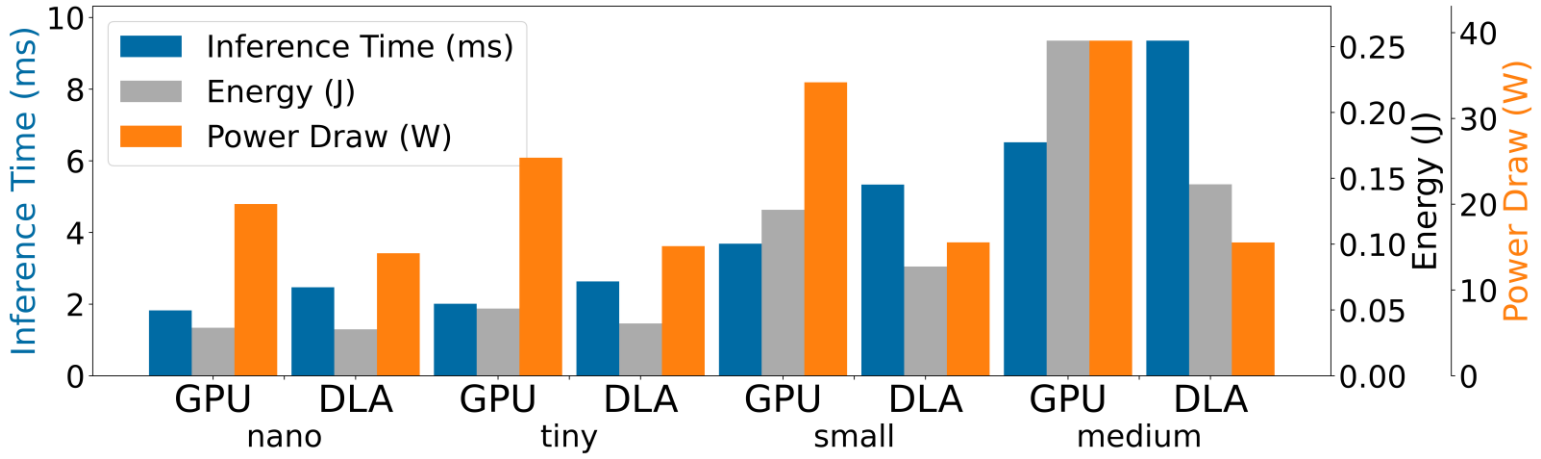
Object Detection vs. Tracking Cost Disparity

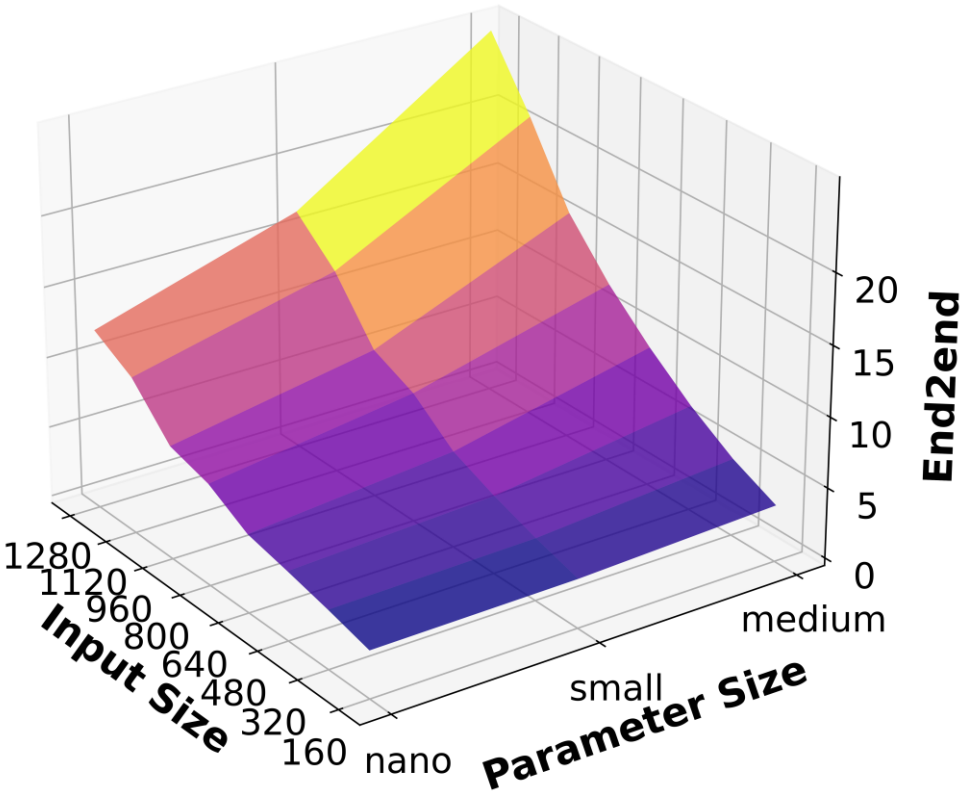
- Object detection with DNN takes significant resources
- Tracking (with no RE-ID) minimal
- Must optimize detection to make meaningful change



How Can Object Detection be Optimized?

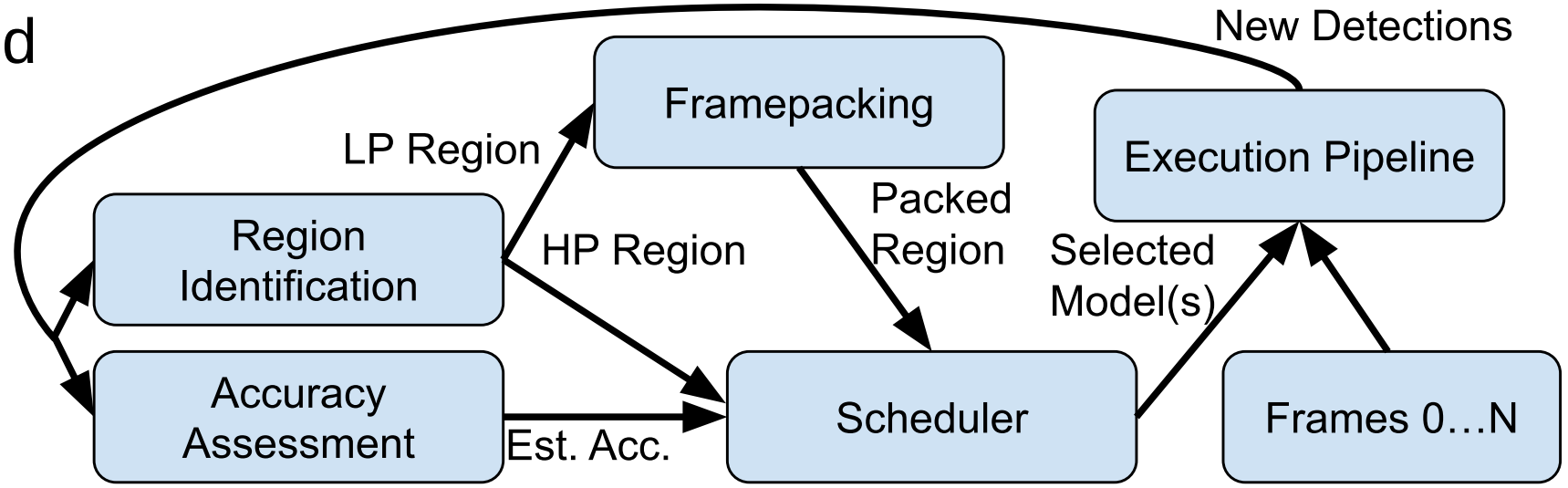
- Change parameter or input size
- Change accelerator
- How can we leverage these 3 knobs?





PMOTHS Overview

- Assess current LP and HP regions from user-defined class-priority labels
- Framepack to reduce computation on LP region
- Schedule to minimize latency



Identifying Priority Objects

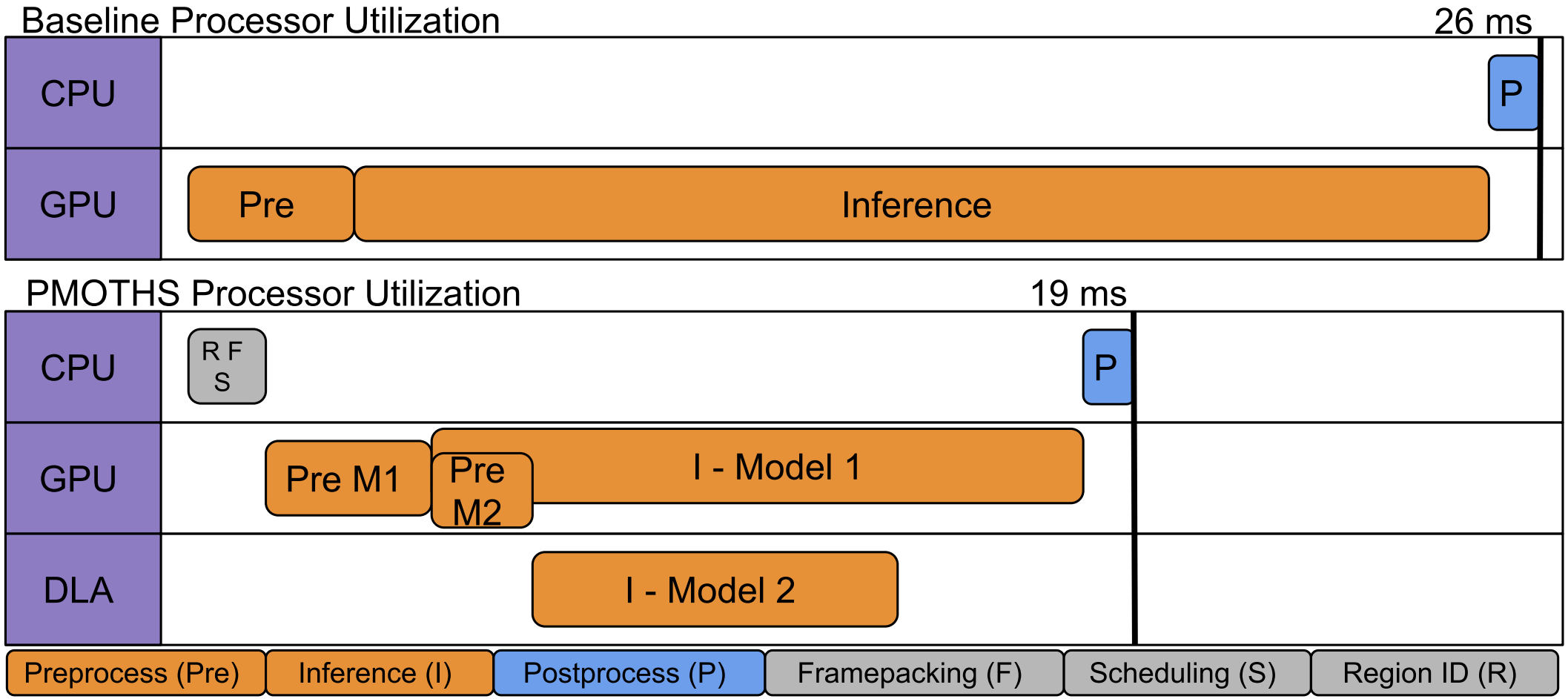
- Important objects can take up small portion
- Significant percentage of data could be ignored

MOT17-02 MOT17-09 MOT17-10 MOT17-13

Frame Packing

- Aggregate 'low-priority' cells into smaller image to reduce computational resources required
- Uses simulated annealing and 2D bin-packing

Processor Utilization



MOT17 Results

Methodology	DetA (%)	HOTA (%)	MOTA (%)	IDF1 (%)	Latency (ms)
YOLOv10 M 1280x1280	41.82	11.64	6.45	9.21	25.91
YOLOv10 S 1280x1280	39.69	11.81	7.80	9.16	16.66
YOLOv10 N 1280x1280	37.35	11.40	7.89	9.97	12.22
BigLittle [28]*	41.46	11.52	7.06	9.05	25.99
SHIFT [6]*	19.01	11.11	8.44	9.49	14.64
PMOTHS *People*	41.70	11.78	7.76	9.28	15.65

3.8x Energy usage improvement
2.2x Latency improvement
2.2x Power draw improvement
0.997x Accuracy performance
vs. Yolo10M-1280 on GPU

Software

trtutils

General purpose TensorRT engines, utilities, and CLI tools with Jetson integration.

Purpose

- Easy-to-use abstractions in Python
  - Students
- No performance sacrifice
  - SHIFT + PMOTHS
- General purpose
  - SHIFT + PMOTHS
- Concurrent
  - PMOTHS

```python
from pathlib import Path
from trtutils import TRTEngine


def main() -> None:
    engine = TRTEngine(Path("example.engine"), warmup=True)

    rand_input = engine.get_random_input()
    outputs = engine.execute(rand_input)
    print(outputs)

    for output in outputs:
        print(output.shape)


if __name__ == "__main__":
    main()
```

Design Goals

- Ease of use for those already using DNNs
  - Most using PyTorch/Tensorflow
- Fast development time
- Low or no overhead
- Thread-safe
- Can be used on Jetson with GPU/DLA or workstation with GPU

```python
from pathlib import Path
import cv2
from trtutils.impls.yolo import YOLO


def main() -> None:
    img = cv2.imread(str(Path("horse.jpg")))
    yolo = YOLO(engine, warmup=True, preprocessor="cuda")
    print(yolo.name)

    tensor, r, p = yolo.preprocess(img)
    output = yolo.run(tensor, postprocess=False)
    p_output = yolo.postprocess(output, r, p)
    bboxes = yolo.get_detections(p_output)
    # OR
    bboxes = yolo.run(img)
    # OR
    bboxes = yolo.end2end(img)

    print(bboxes)


if __name__ == "__main__":
    main()
```

Device & Programming Language Support

	JetPack 4	JetPack 5	JetPack 6
Jetson Nano	✅	❌	❌
Jetson TX2	✅	❌	❌
Jetson Xavier NX	✅	✅	❌
Jetson Xavier AGX	✅	✅	❌
Jetson Orin AGX	❌	✅	✅
Jetson Orin NX	❌	✅	✅
Jetson Orin Nano	❌	✅	✅

- JetPack Bundles
  - Linux Kernel
  - Python
  - CUDA
  - TensorRT
  - cuDNN
  - VPI
- Support majority of CUDA and TensorRT enabled systems with least effort

Python	C++
• Simple • Slow • No build system • GIL • Managed memory	• Moderate • Fast • Requires build system • Free-threaded

Device & Programming Language Support

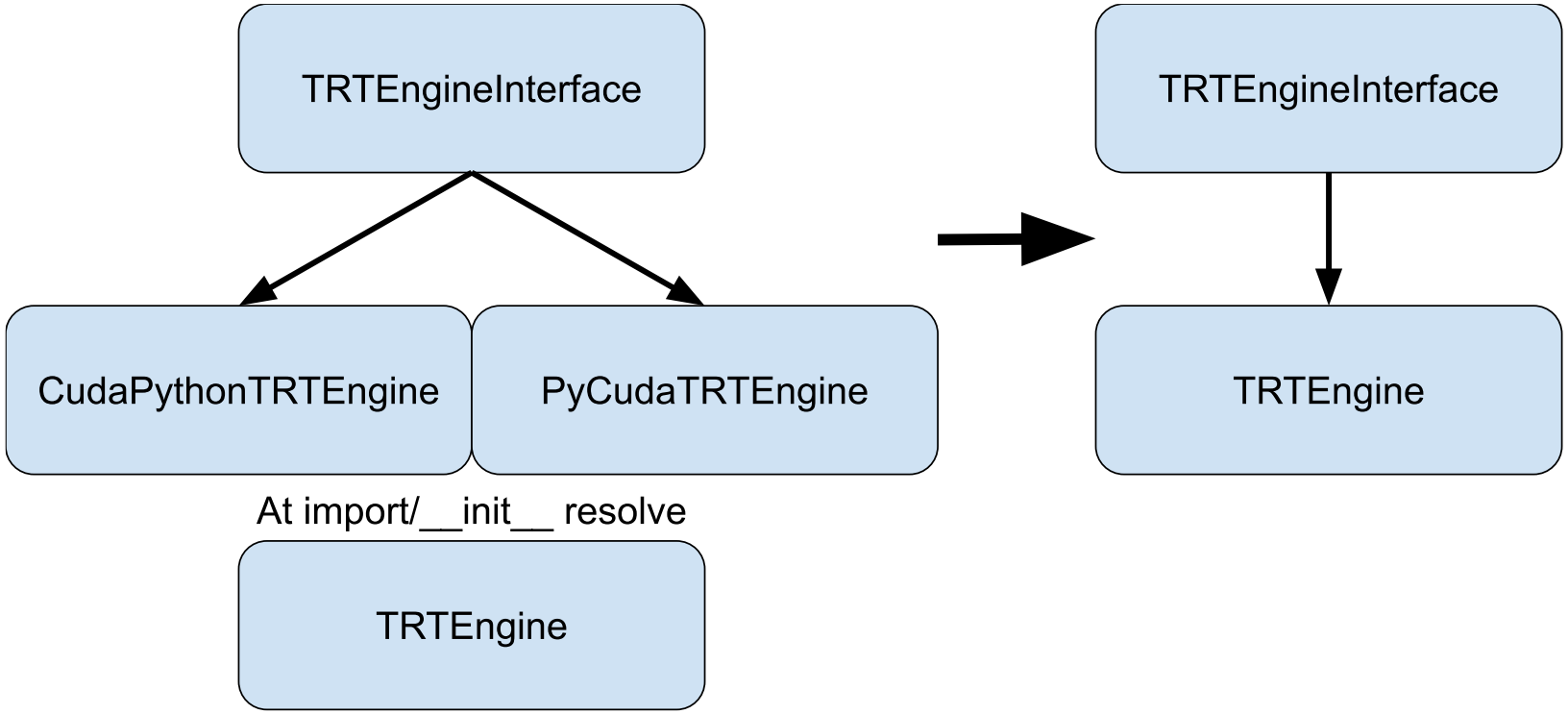
- cuda-python and tensorrt
  - 1:1 mapping to C++
  - Supports CUDA 11+
- Create Python CUDA wrappers
- Allows
  - Thread-safety
  - Debugging
  - Automated logic

```
def cuda_malloc(
    nbytes: int,
) -> int:
    """
    Perform a memory allocation using cudart.cudaMalloc.

    Parameters
    ----------
    nbytes : int
        The number of bytes to allocate.

    Returns
    -------
    int
        The pointer to the allocated memory.

    """
    with _MEM_ALLOC_LOCK:
        device_ptr: int = cuda_call(cudart.cudaMalloc(nbytes))
    LOG.debug(f"Allocated, device_ptr: {device_ptr}, size: {nbytes}")
    return device_ptr
```

trtutils Software Architecture

Originally supported multiple CUDA backends

- Support matrix too large
- Enables faster development



Handle CUDA/memory allocation separate from execution implementations

trtutils Optimizations

1. Reduce overhead
    1. No Torch function usage
    2. No expensive runtime loads to occupy memory
    3. No unnecessary abstractions
2. Use CUDA kernels directly instead of ML framework
3. No intermediate stream synchronize needed
    1. Everything occurs in single CUDA stream directly
4. Pagelocked memory

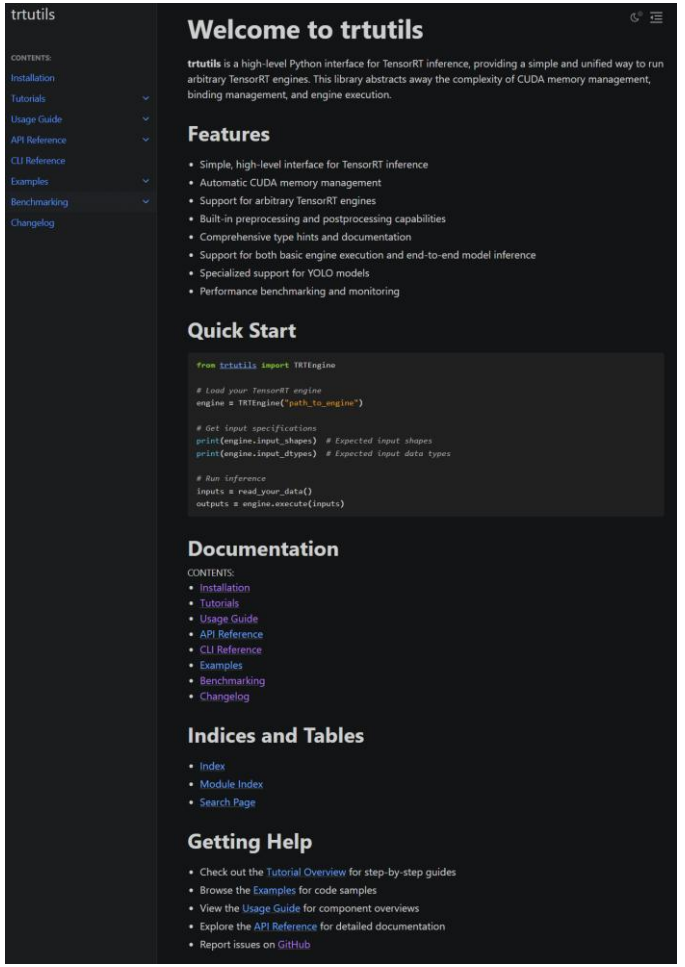
trtutils Package Overview

Submodules

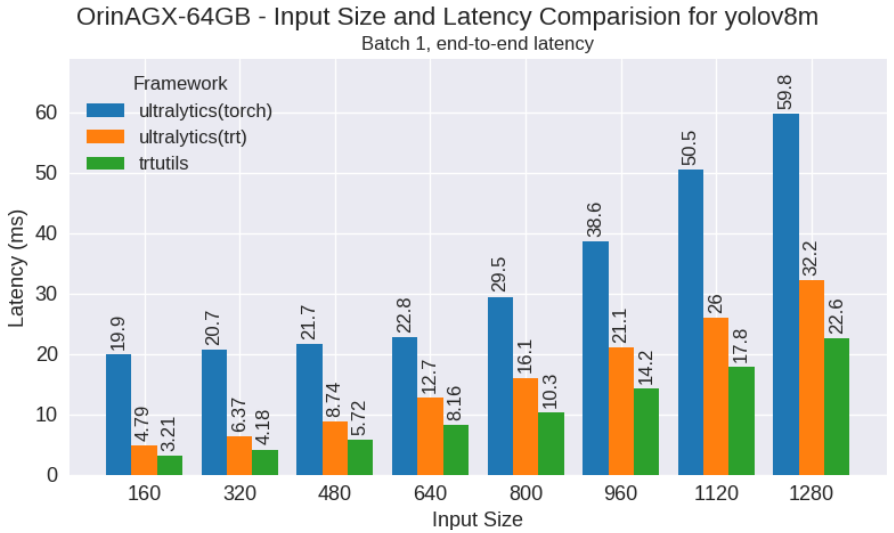
- ➔ core
- ➔ builder
- ➔ impls
- ➔ inspect
- ➔ jetson
- ➔ trtexec

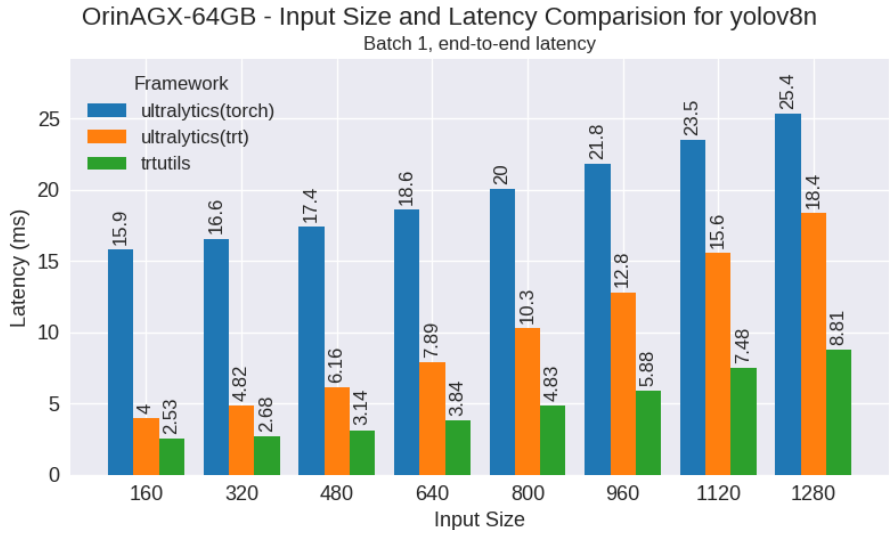
CLI

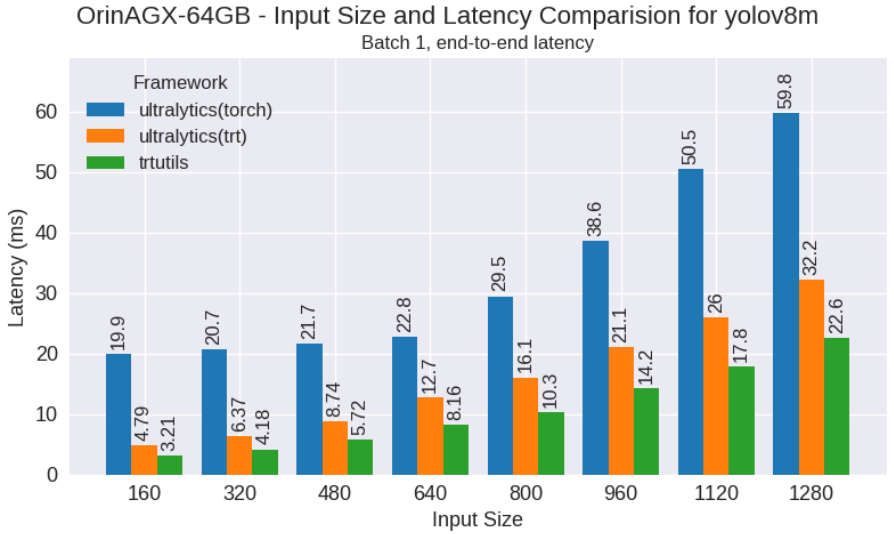
- ➔ benchmark
- ➔ build
- ➔ can_run_on_dla
- ➔ build_dla
- ➔ yolo
- ➔ inspect

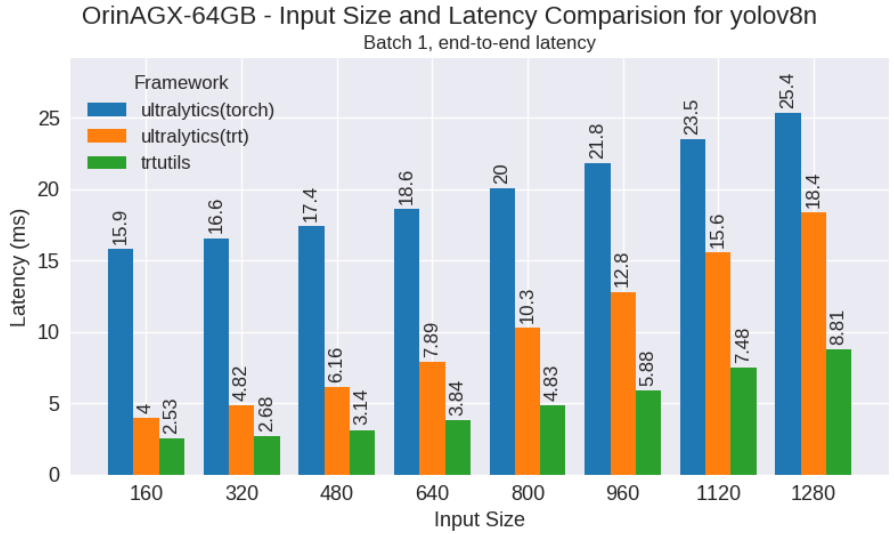


Performance – Orin AGX 64GB & Titan RTX

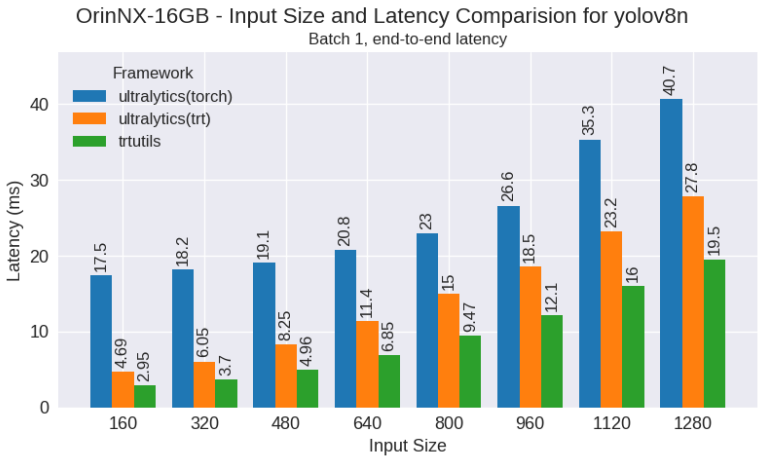


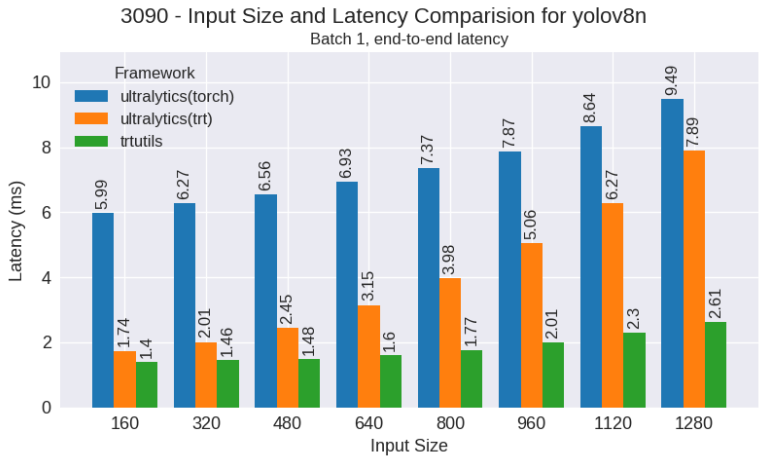


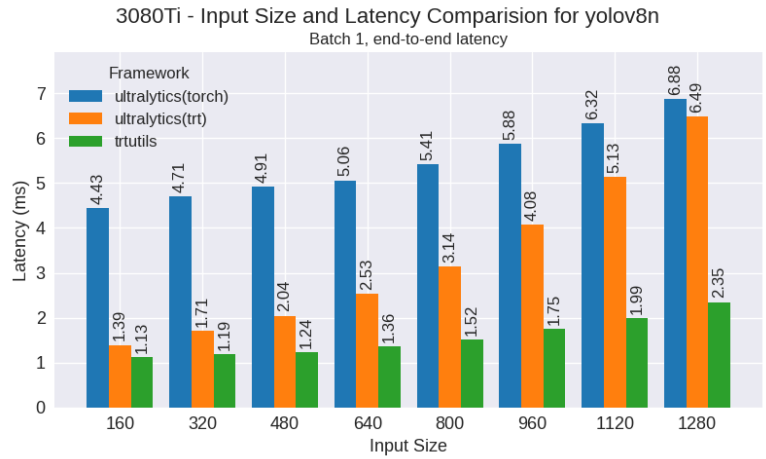


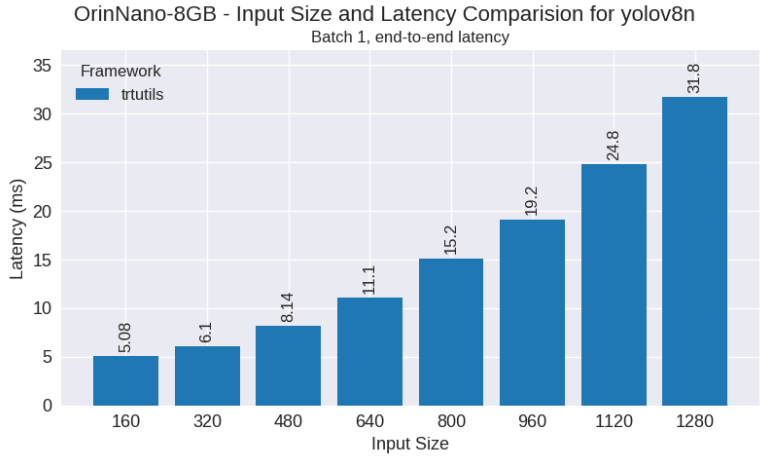


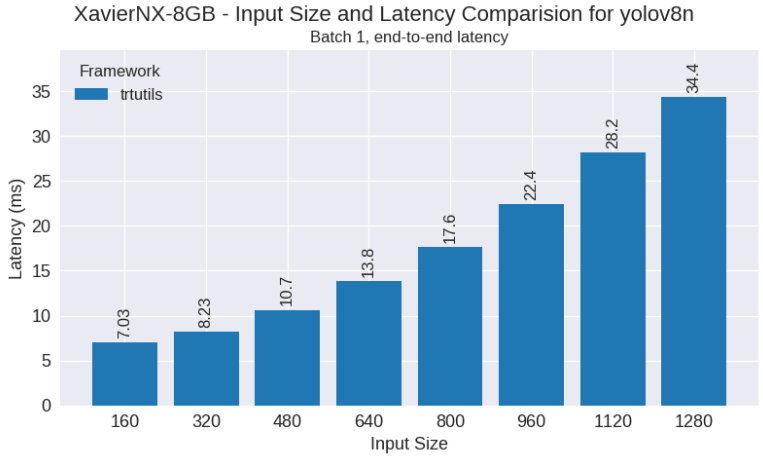
Performance – Other Systems



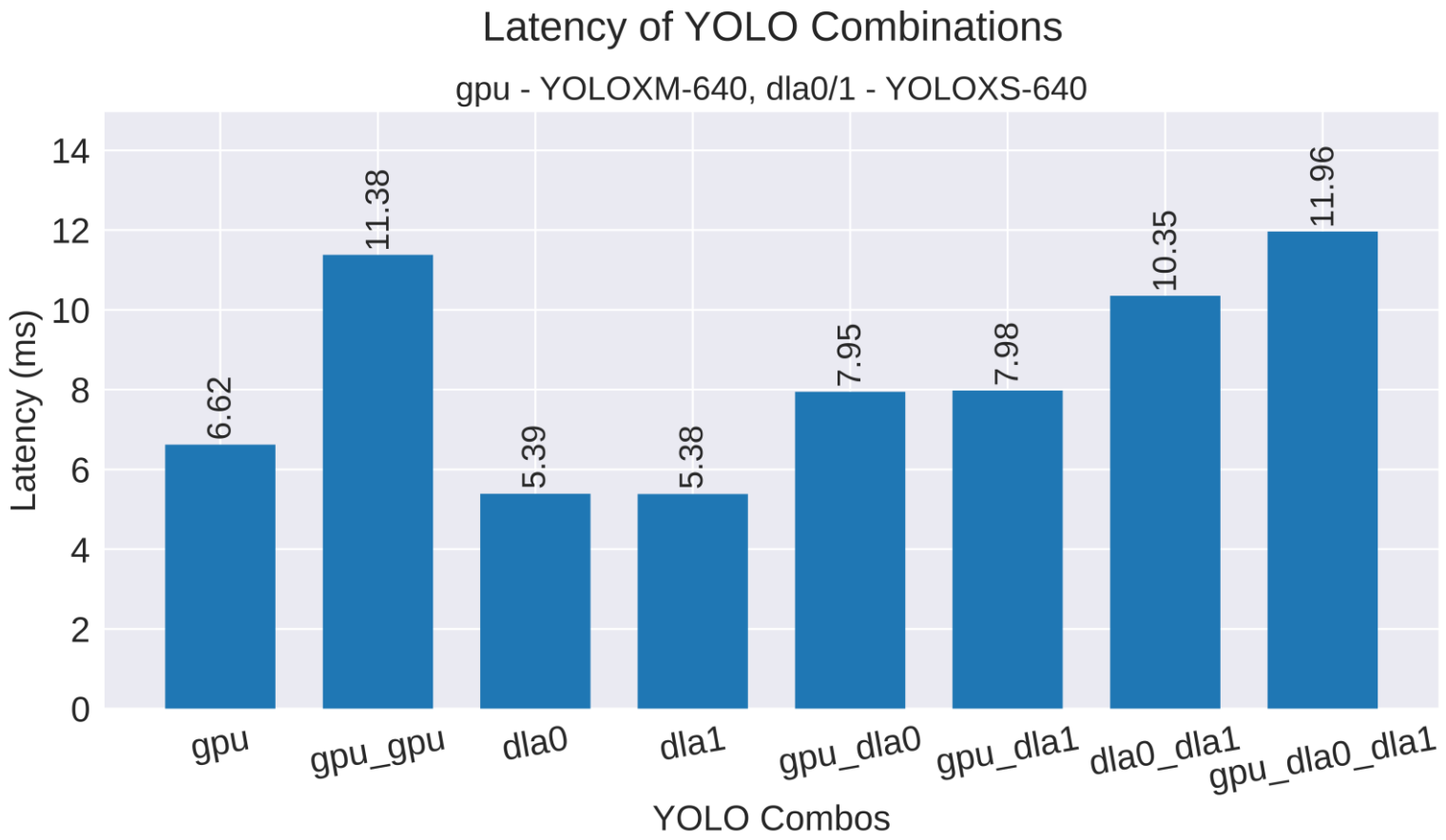








Concurrent Inference – Orin AGX 64GB

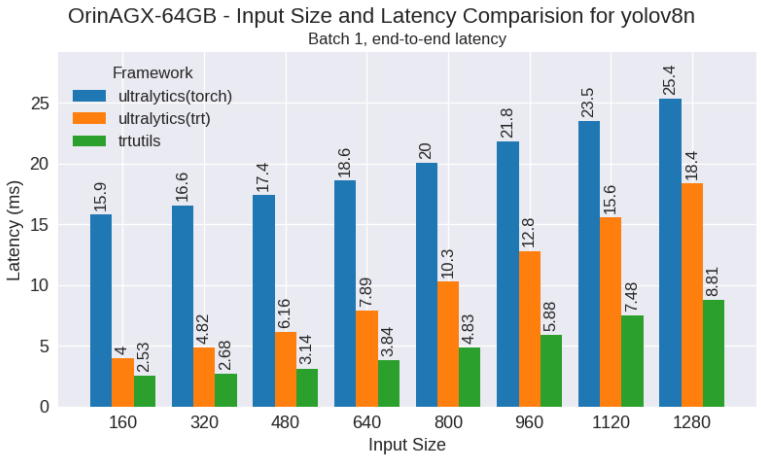


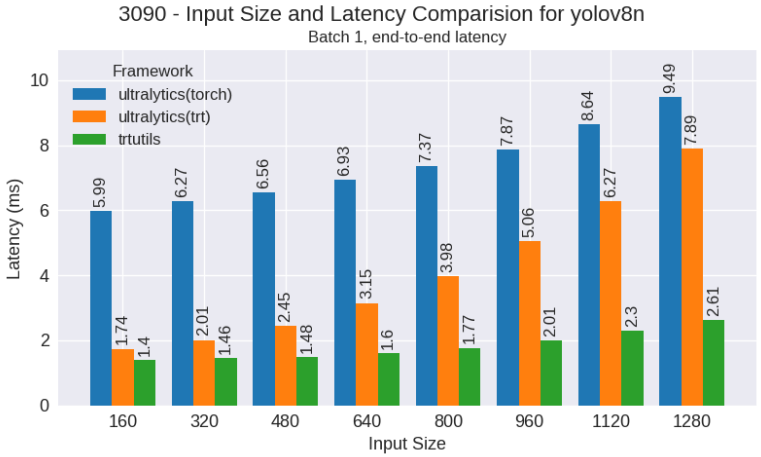
Limited in concurrency by DLA engines

- Require occasional GPU layers serializing performance
- Need end-to-end DLA compatible model

Outcomes

1. Faster than existing Python inference frameworks
    1. Pagelocked memory
    2. NVRTC to compile CUDA kernels
    3. No intermediate memory transfer
    4. No intermediate stream sync
2. ‘pip’ installable, no build requirements
3. Achieve DLA support
4. Multi-thread compatible

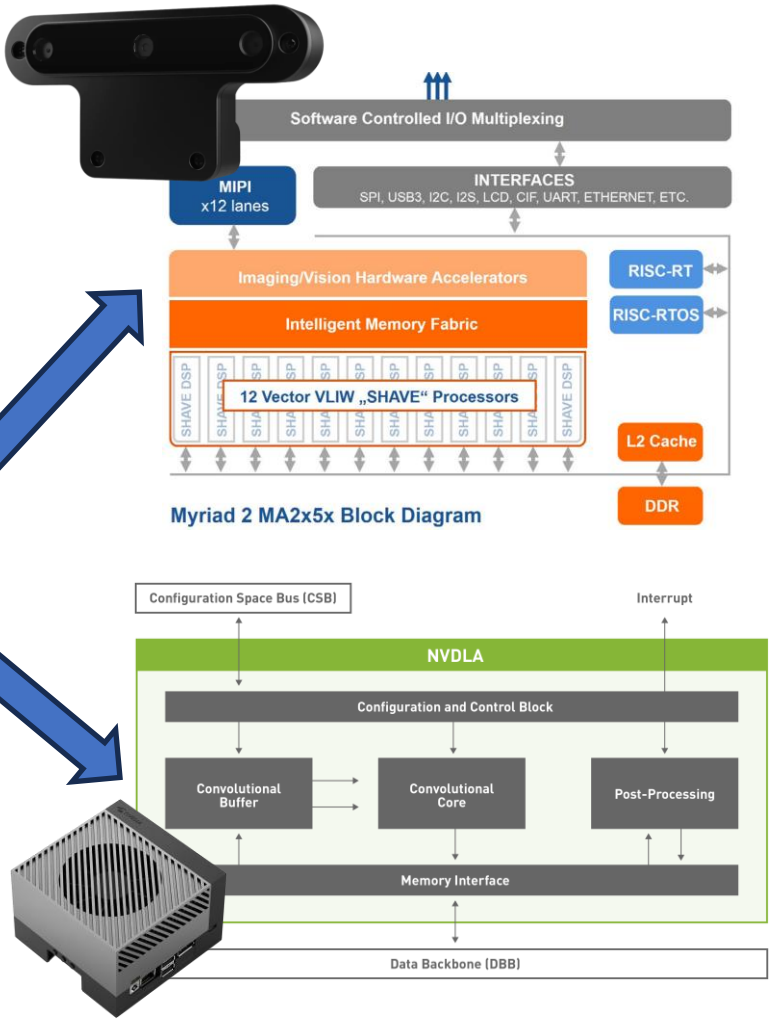




Up to 3x faster

Future Work/Direction

1. Use pagelocked memory for inputs
2. Support batching in YOLO
3. Generic detector/classifier
4. Implement research submodule and papers

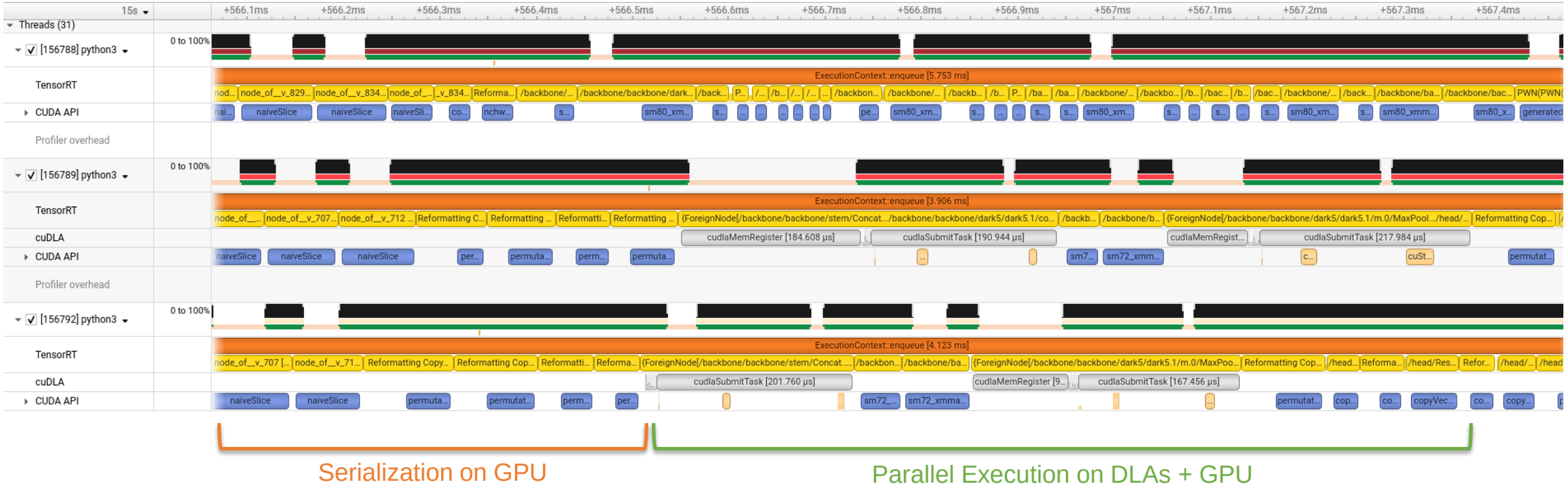


DLA as CV coprocessor

Thank you for your time!

Appendix

Concurrent Inference – Orin AGX 64GB Nsight Overview

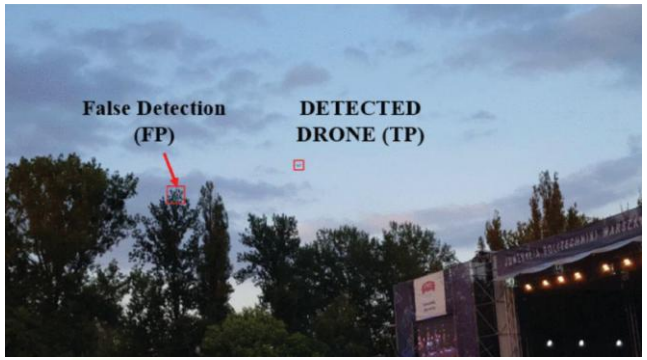


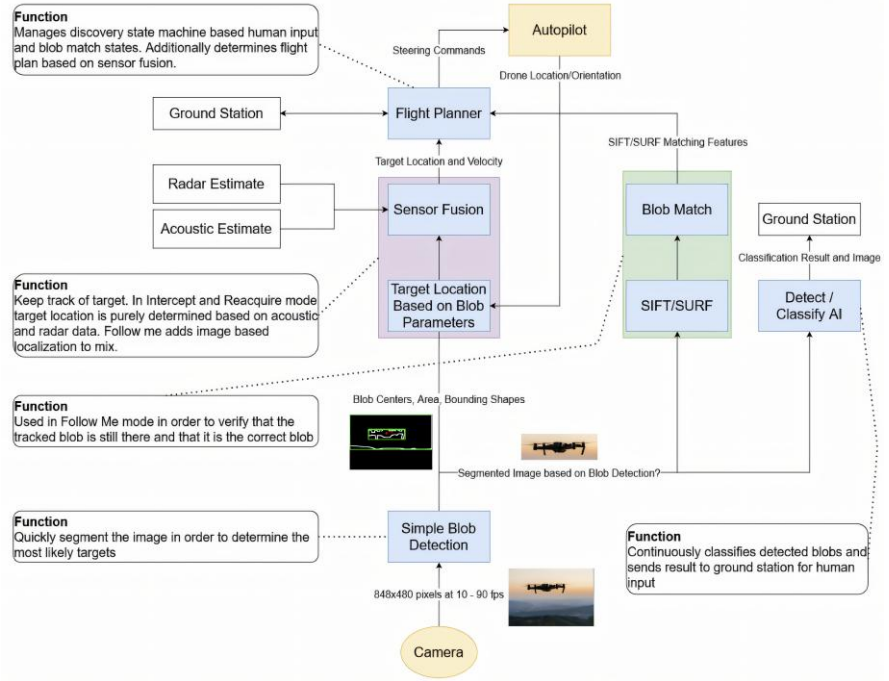
Extra

USAFA

USAFA Drone Discovery

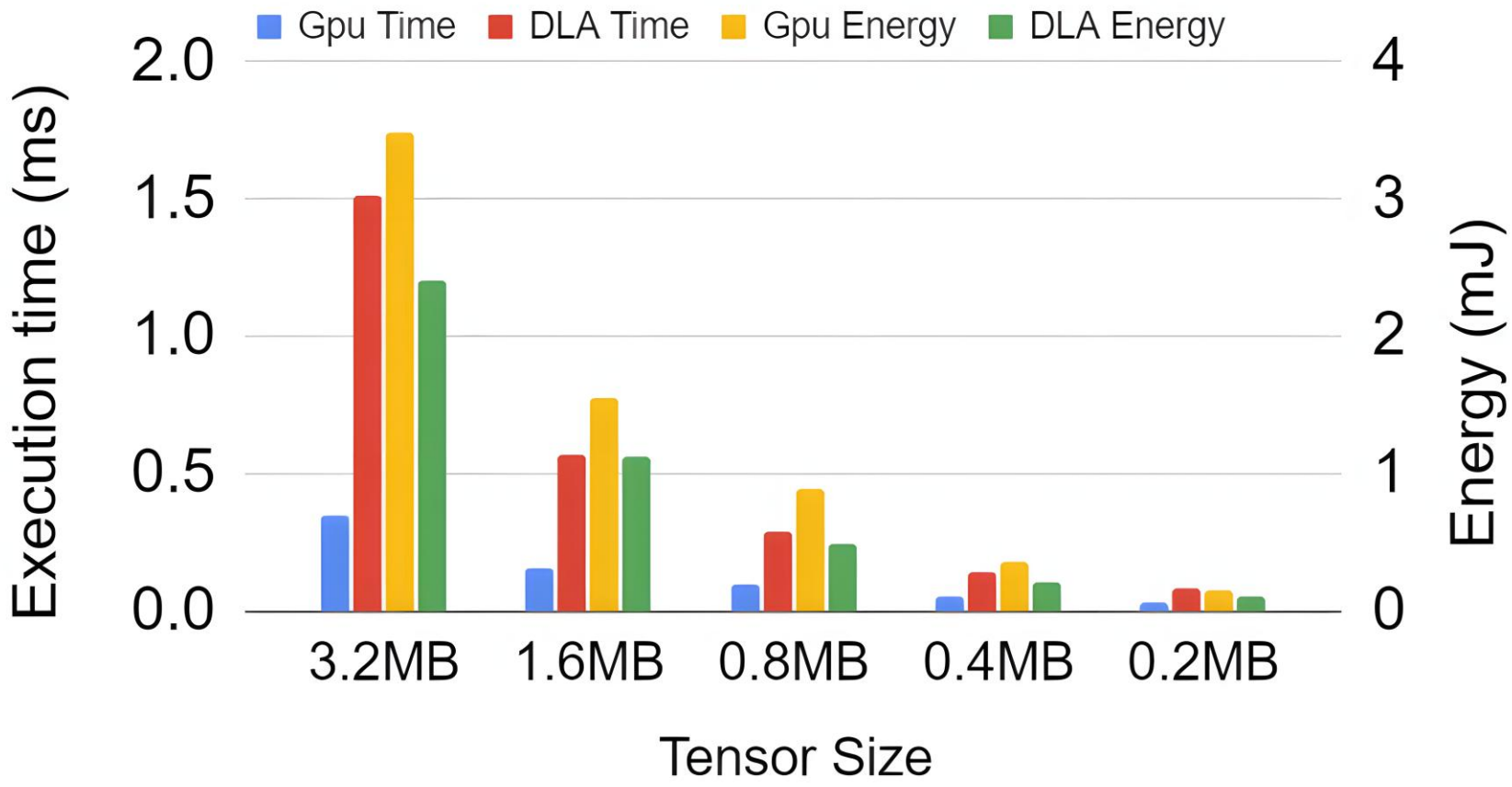
- USAFA joint project focusing on drone/UAV discovery, tracking, and pursuit.
- Responsibilities
  - Analyzed performance of DNN and traditional CV algorithms
  - Implemented autonomous control
- Takeaways
  - How can you efficiently determine accuracy of DNN at runtime?





GPU, DLA, FPGA

GPU vs. DLA - Convolution



GPU vs. DLA vs. CPU – DNN

Model	IoU	Inference (s)			Power (W)			Energy (J)		
		CPU	GPU	DLA	CPU	GPU	DLA	CPU	GPU	DLA
YoloV7	0.62	1.65	0.13	0.12	7.60	15.1	15.1	20.5	1.97	1.78
YoloV7Tiny	0.53	0.38	0.03	0.02	7.20	11.2	11.2	4.19	0.28	0.27
MobilenetV1	0.45	-	0.09	0.09	-	16.2	6.10	-	1.52	0.56

TABLE I: Average statistics for two architectures of object detection models and their performance on CPU, GPU, and GPU/DLA.

CPU vs. GPU vs. FPGA - Robotics

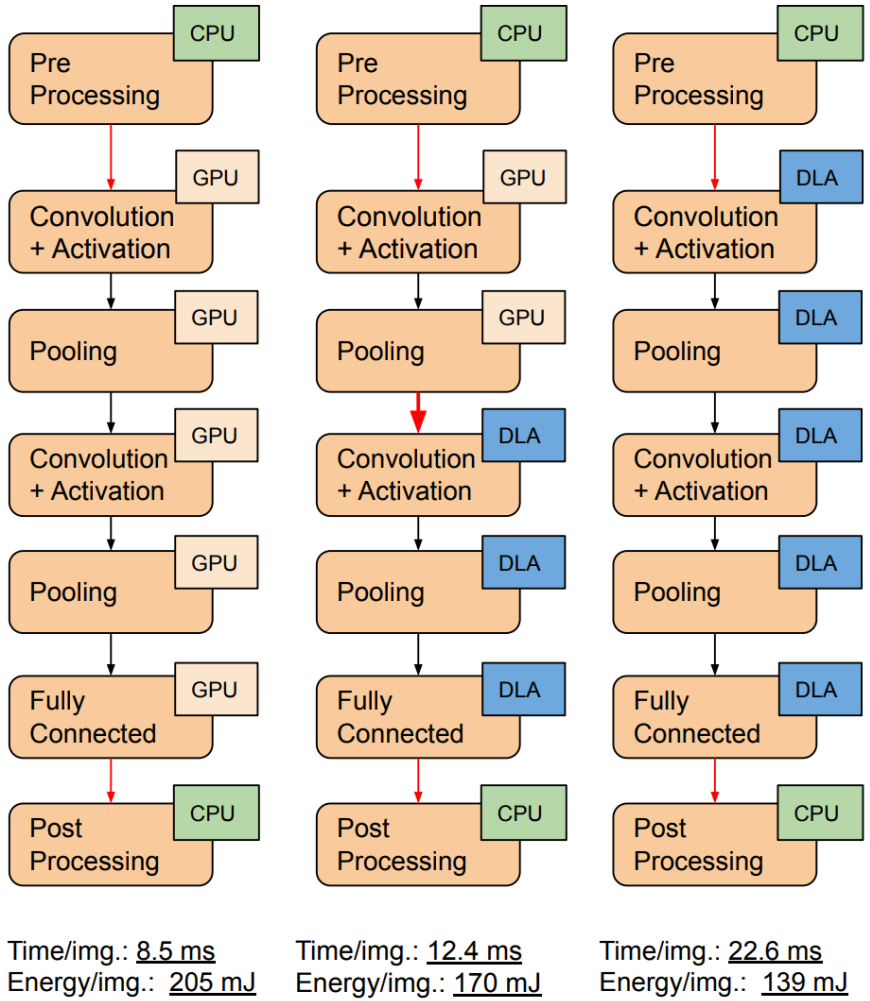
Table 2. Types of accelerators and their performance characteristics.

Computational Type/Trait	CPU	GPU	FPGA
IO	Great	Poor	Poor
Single Threaded	Great	Poor	Good
Fine-Grained Parallelism	Poor	Poor	Great
Course-Grained Parallelism	Good	Great	Good
Irregular Data & Access	Poor	Poor	Great
Single Operation Latency	Great	Good	Poor
Power Efficiency	Poor	Poor	Great
Usability	Great	Good	Poor

AXONN

AXONN – GPU/DLA

- Only GPU or only DLA may not provide the most efficient schedule
- Mixing accelerators across DNN allows more granular tradeoff
- Transition cost to swap accelerators



AXONN – GPU/DLA

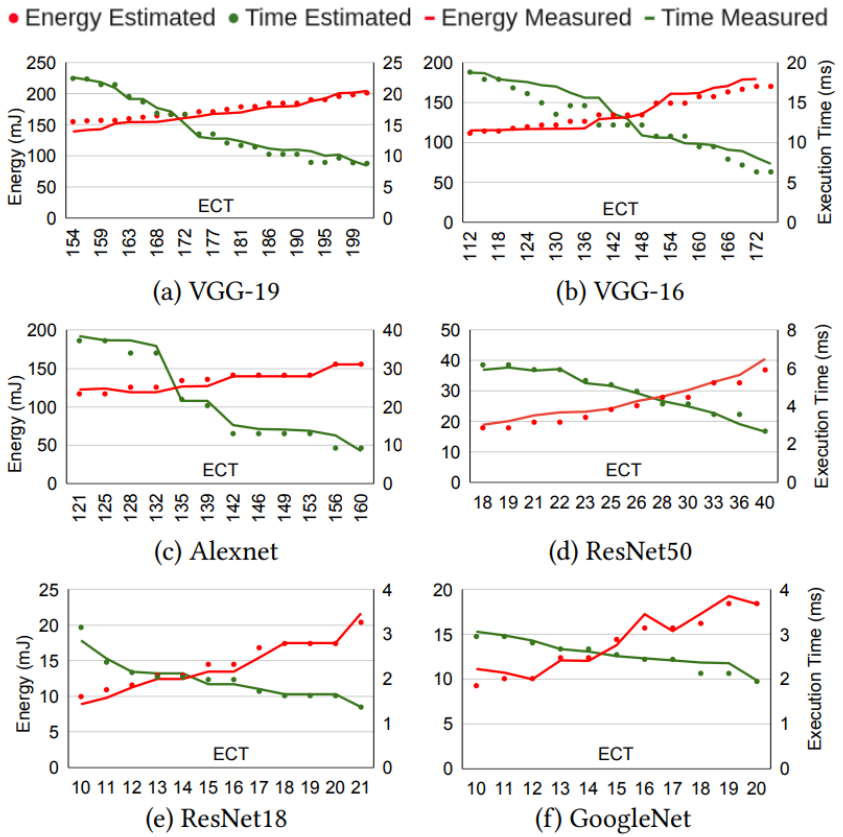
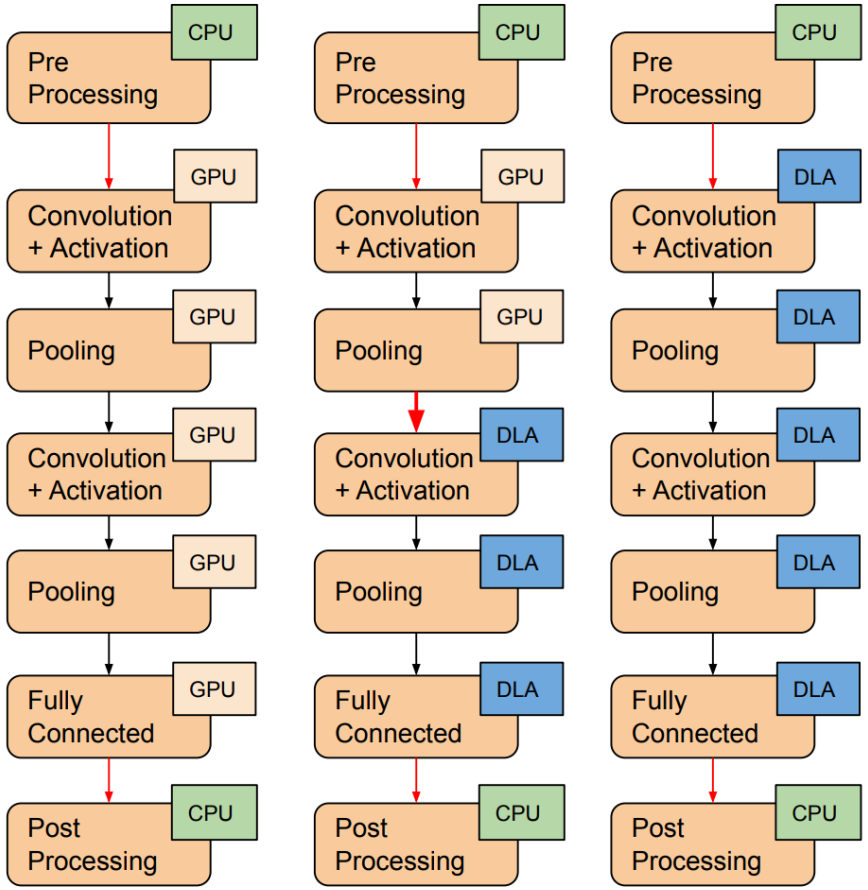


Figure 6: The comparison between the energy and execution times of the schedules estimated by *AxoNN* versus the actual energy and execution times measured for the corresponding actual runs. For each network we have targeted varying ranges of ECT that is between the energy consumption of all-DLA and all-GPU execution.



SHIFT

SHIFT – Related Work

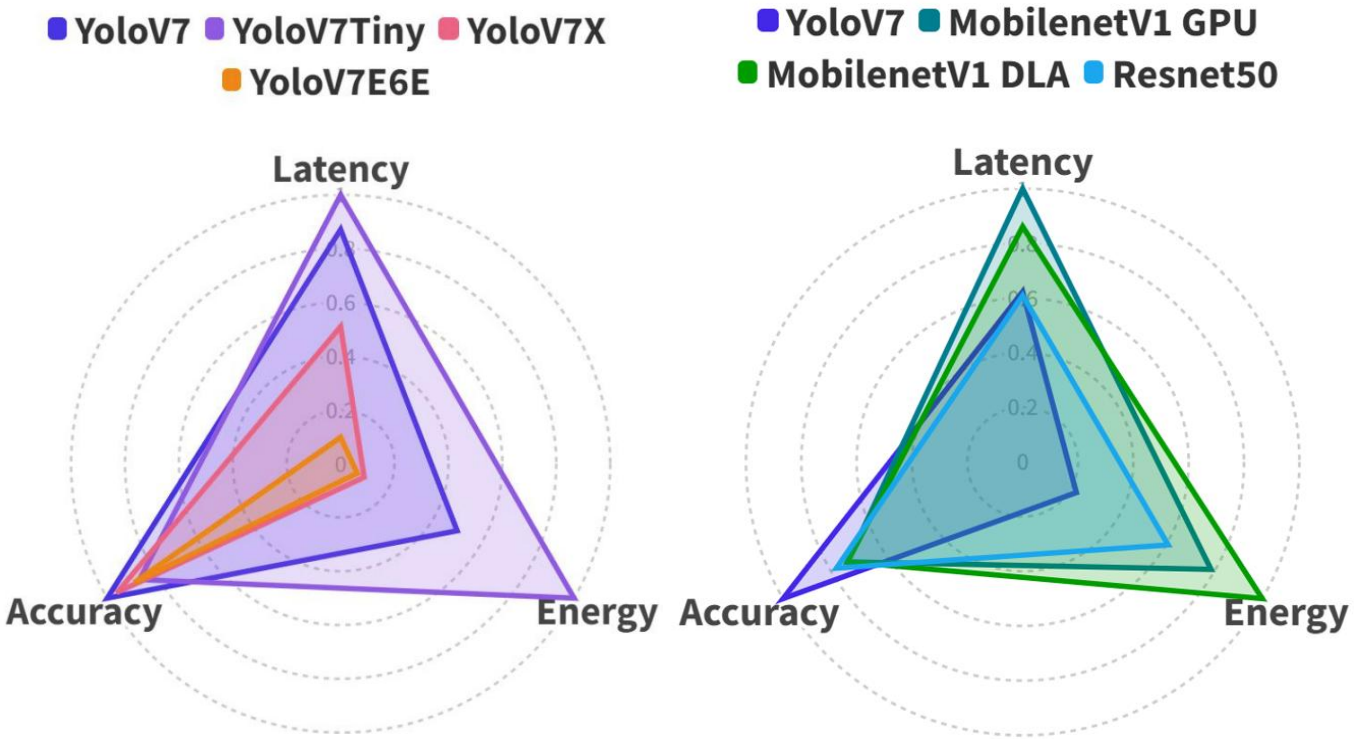
Feature \ Related Work	Glimpse [2]	MARLIN [5]	AdaVP [4]	RoaD-RuNNer [9]	Fast UQ [10]	Herald [11]	AxoNN [7]	*SHIFT*
Context Awareness	✗	✓	✓	✓	✗	✗	✗	✓
Multi-Accelerator	✗	✗	✗	✗	✗	✓	✓	✓
Multi-DNN	✗	✗	✗	✗	✓	✗	✗	✓
Energy-Aware	✗	✓	✓	✓	✗	✓	✓	✓
No-Offloading	✗	✓	✓	✗	✓	✓	✓	✓
Continuous	✓	✓	✗	✓	✗	✗	✗	✓

TABLE II: Comparison of the features offered by related works.

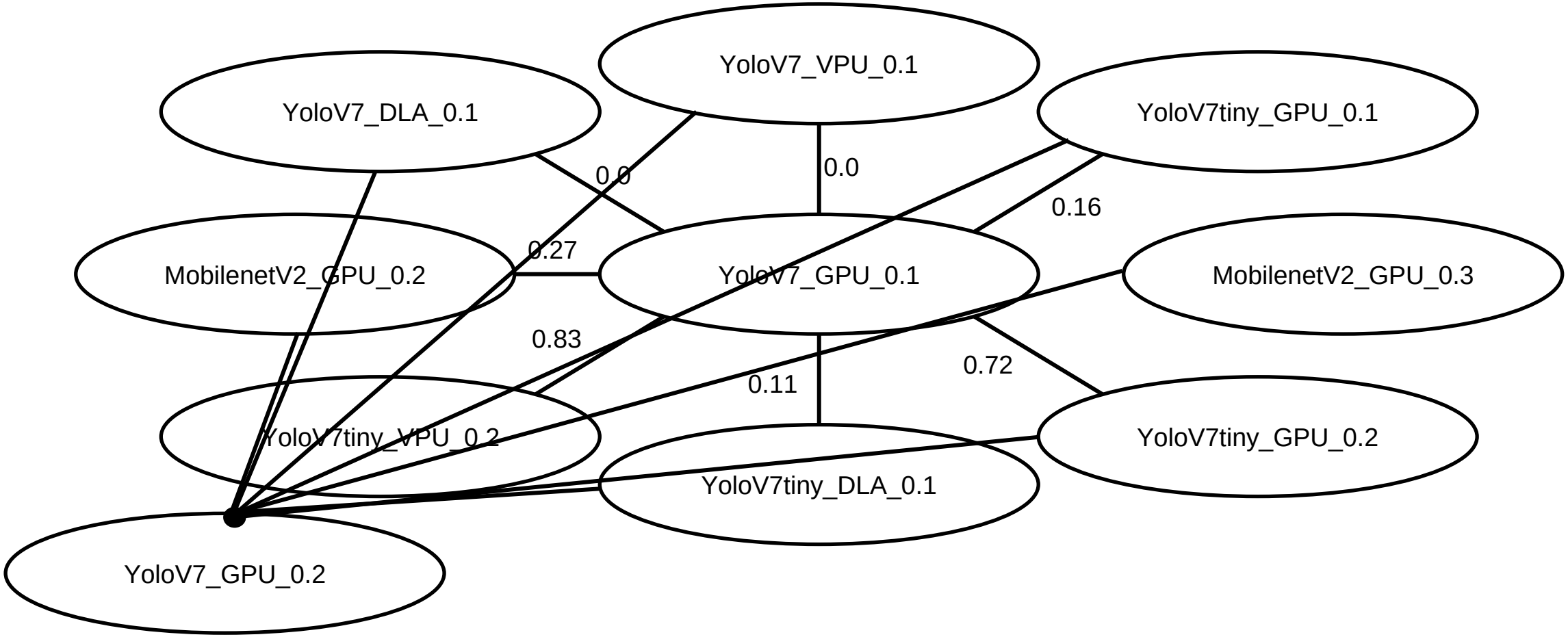
SHIFT - Problem Statement

How can we utilize multiple models and multiple accelerators while optimizing for energy/latency?

- How do we know when we have chosen correctly?
- When do we switch between models or accelerators?



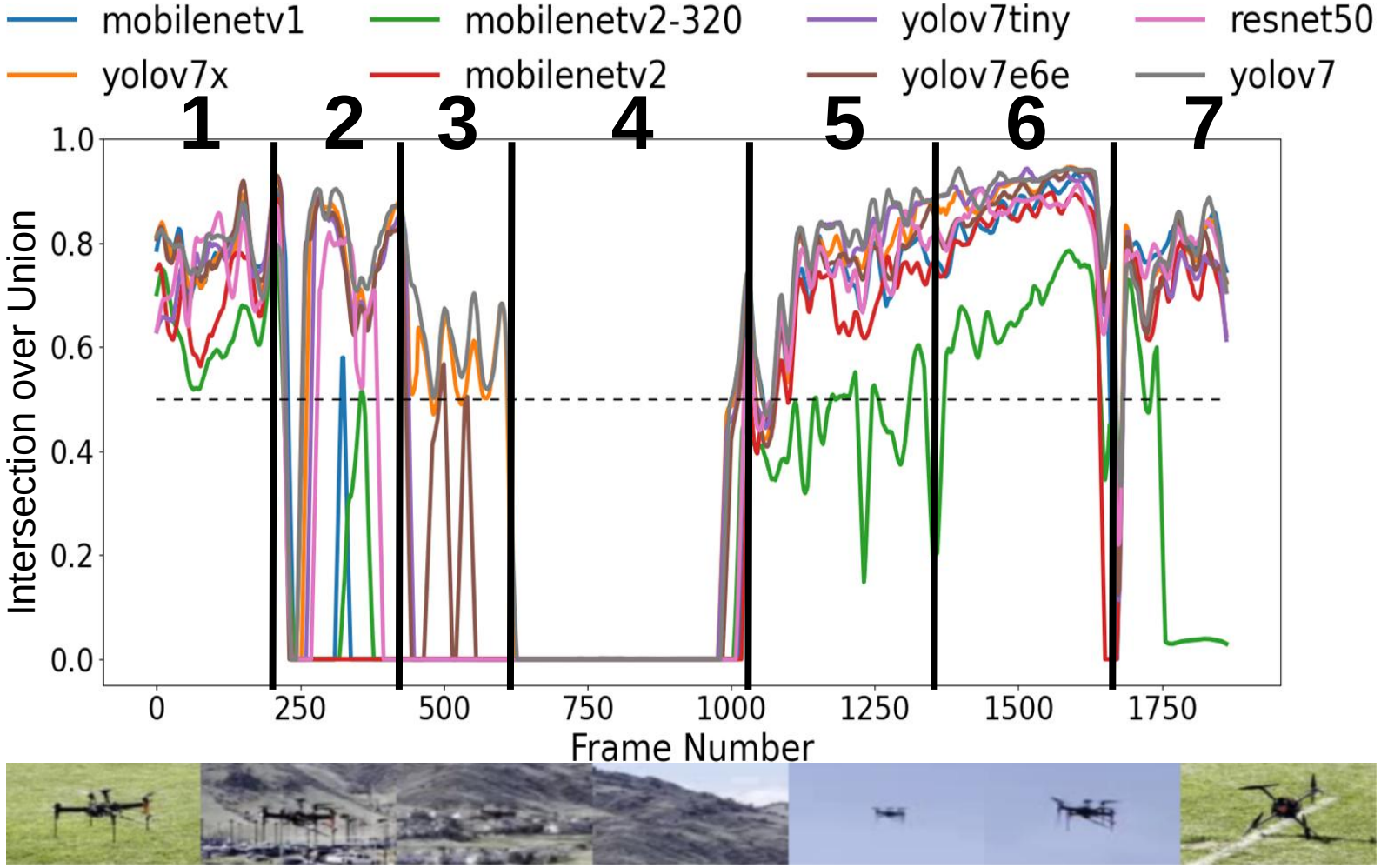
SHIFT - Confidence Graph



Multi-Model Inference

Which models achieve accuracy threshold:

1. All models
2. All YoloV7 + Resnet
3. YoloV7, YoloV7X
4. None
5. All except smallest
6. All models
7. All except smallest



SHIFT Scheduler

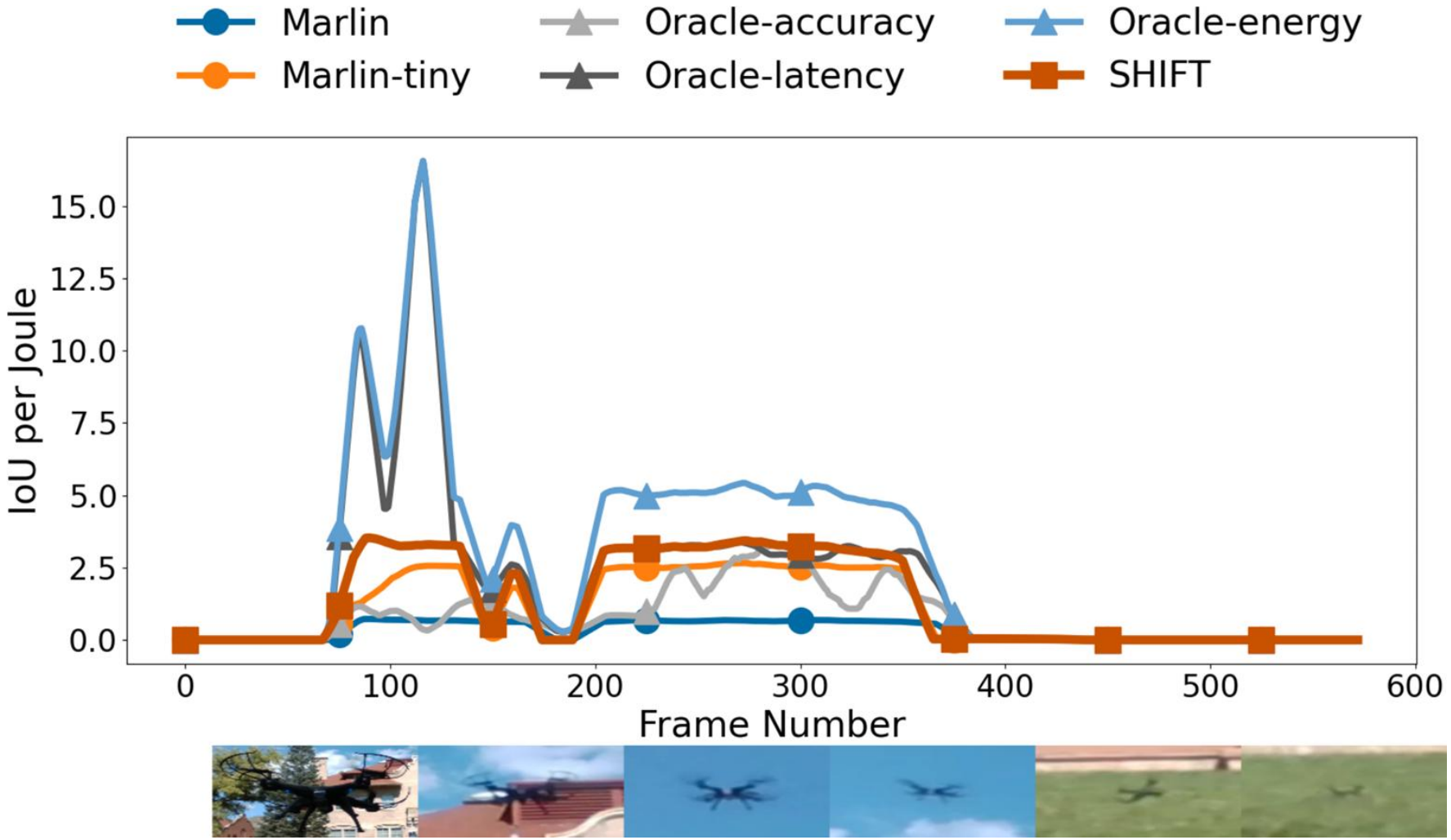
- Set of energy/accuracy/latency knobs for user adjustment
- Computes image similarity across image and detected bounding boxes to determine if context is changing
- Maximizes score

$$\mathrm{NCC}(p,c) = \frac{\sum (p - mean(p))(c - mean(c))}{(\sqrt{\sum (c - mean(c))^2} \times \sqrt{\sum (p - mean(p))^2}} \quad (1)$$

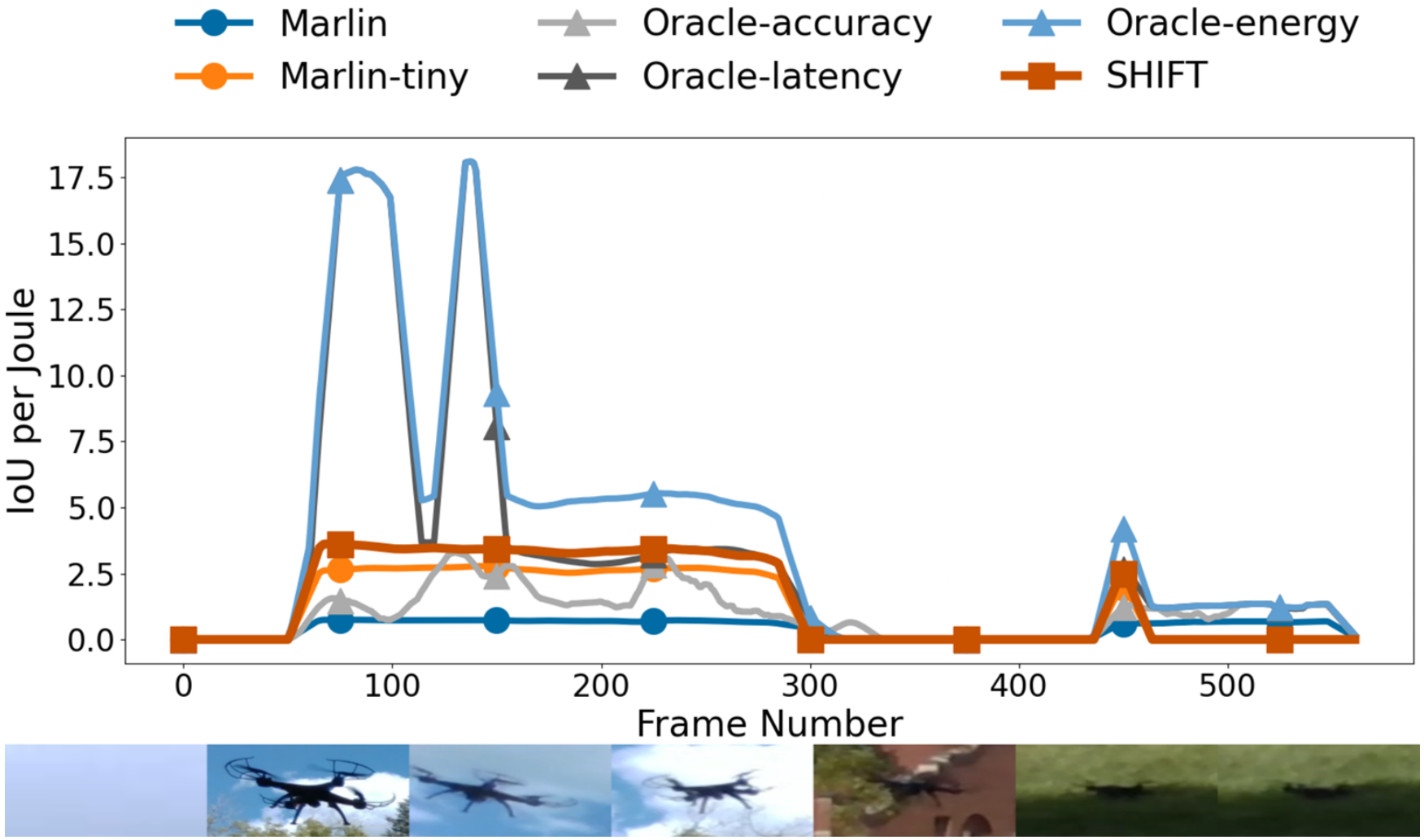
Algorithm 1 Model Scheduling

procedure *SHIFT* SCHEDULE(m, c, i, b)
  $s = \min(\mathrm{NCC}(\text{lastImage}, i), \mathrm{NCC}(\text{lastBbox}, b))$
  if $s \times c \geq accuracyThreshold$ **then**
    return m
  end if
  $E =$ scheduler.energy ▷ $0 \rightarrow 1$ model energy
  $L =$ scheduler.latency ▷ $0 \rightarrow 1$ model latency
  $W =$ scheduler.weights ▷ Tuned knobs
  $C =$ graphPredict(m, c) ▷ set of (name, acc, dist)
  $R, scores =$ map(), map()
  for $(n, a, d) \in C$ **do**
    a.Buffer.$append(a)$
    $R[n] =$ average(a.Buffer)
  end for
  $V = \{ n \mid n \in R, n \geq accuracyThreshold \}$
  if length(V) == 0 **then**
    $V = R$
  end if
  for $n \in R$.keys() **do**
    $s = R[n] * W[0] + E[n] * W[1] + L[n] * W[2]$
    $scores[n] = s$
  end for
  return max($scores$)
end procedure

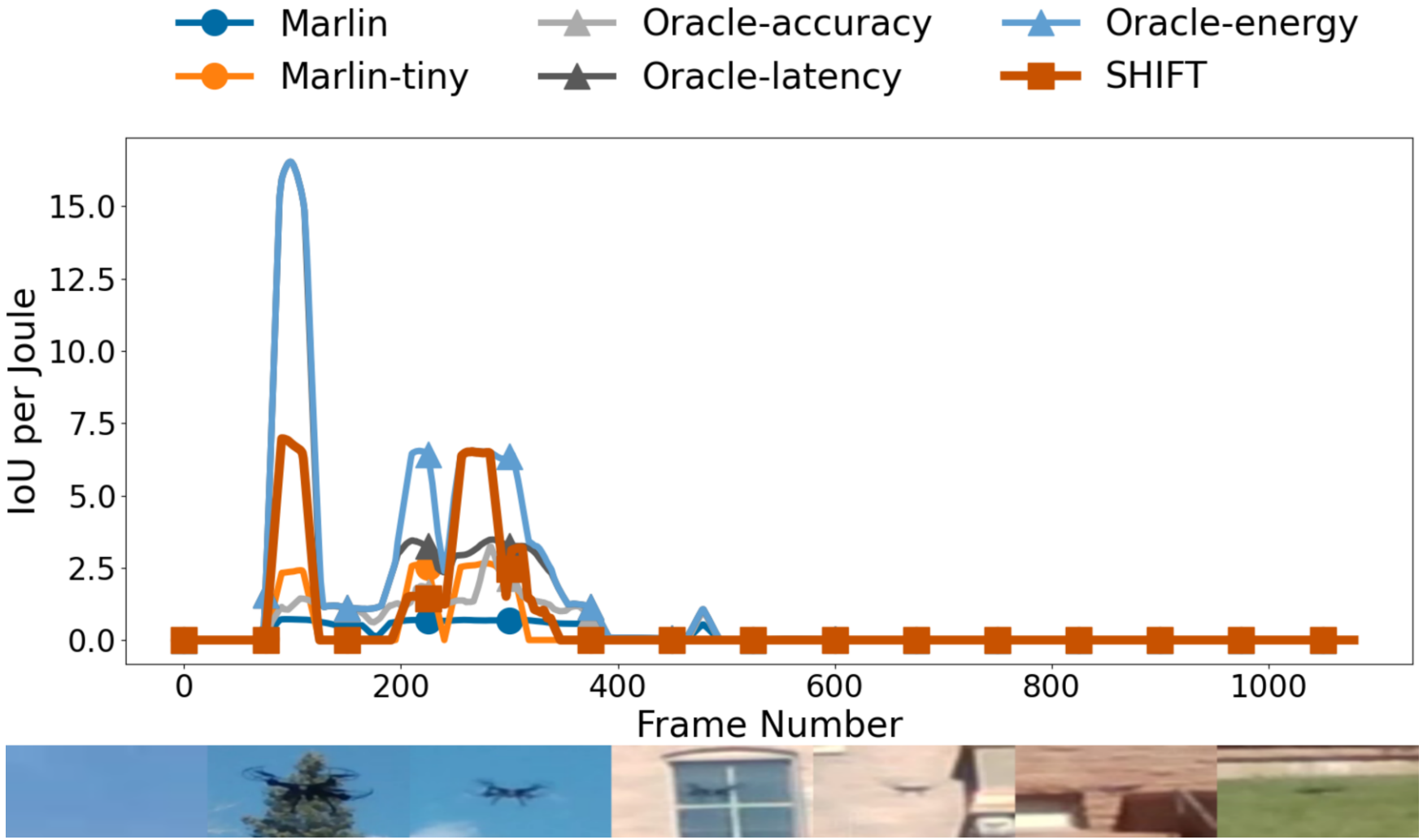
SHIFT - Scenario 1



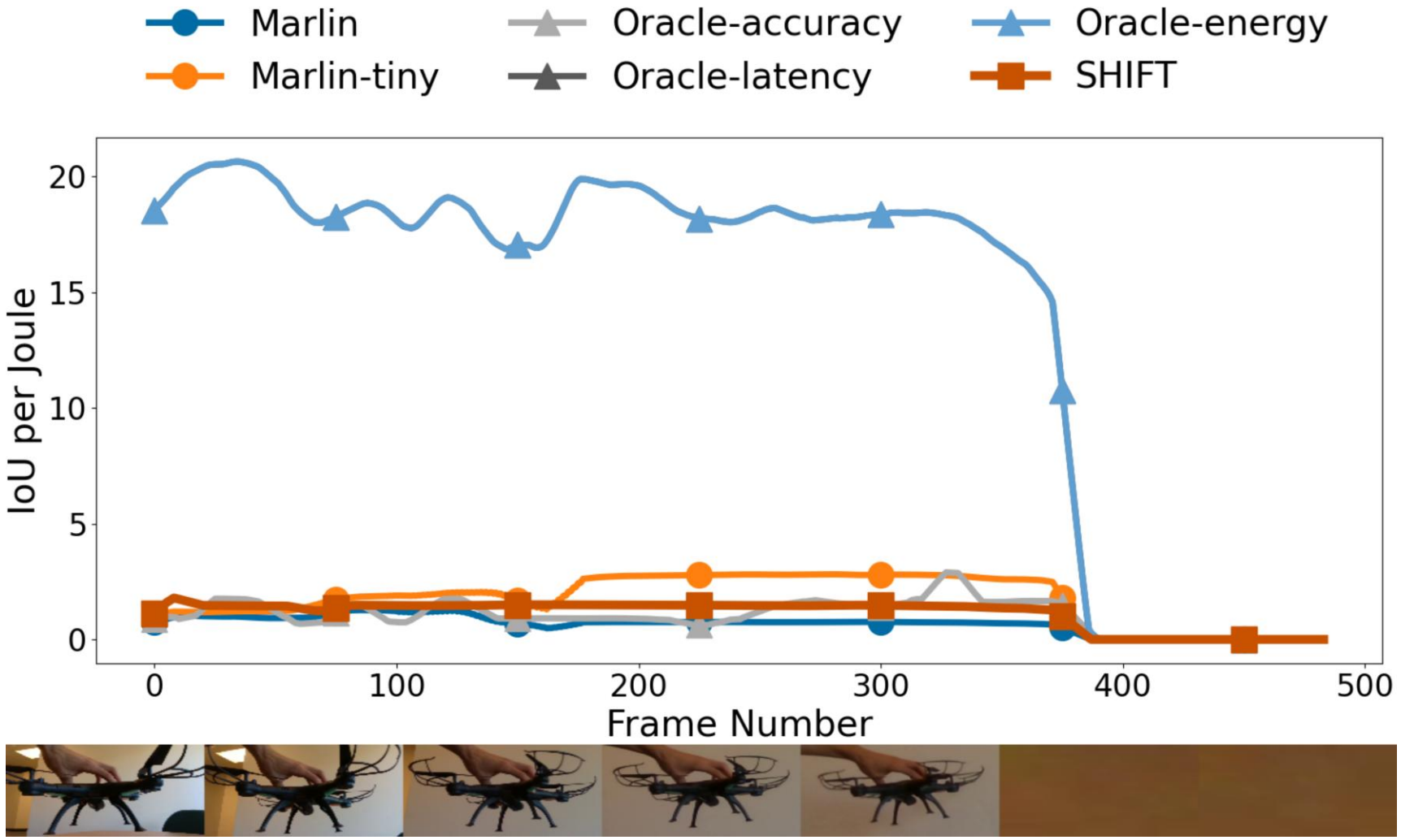
SHIFT - Scenario 2



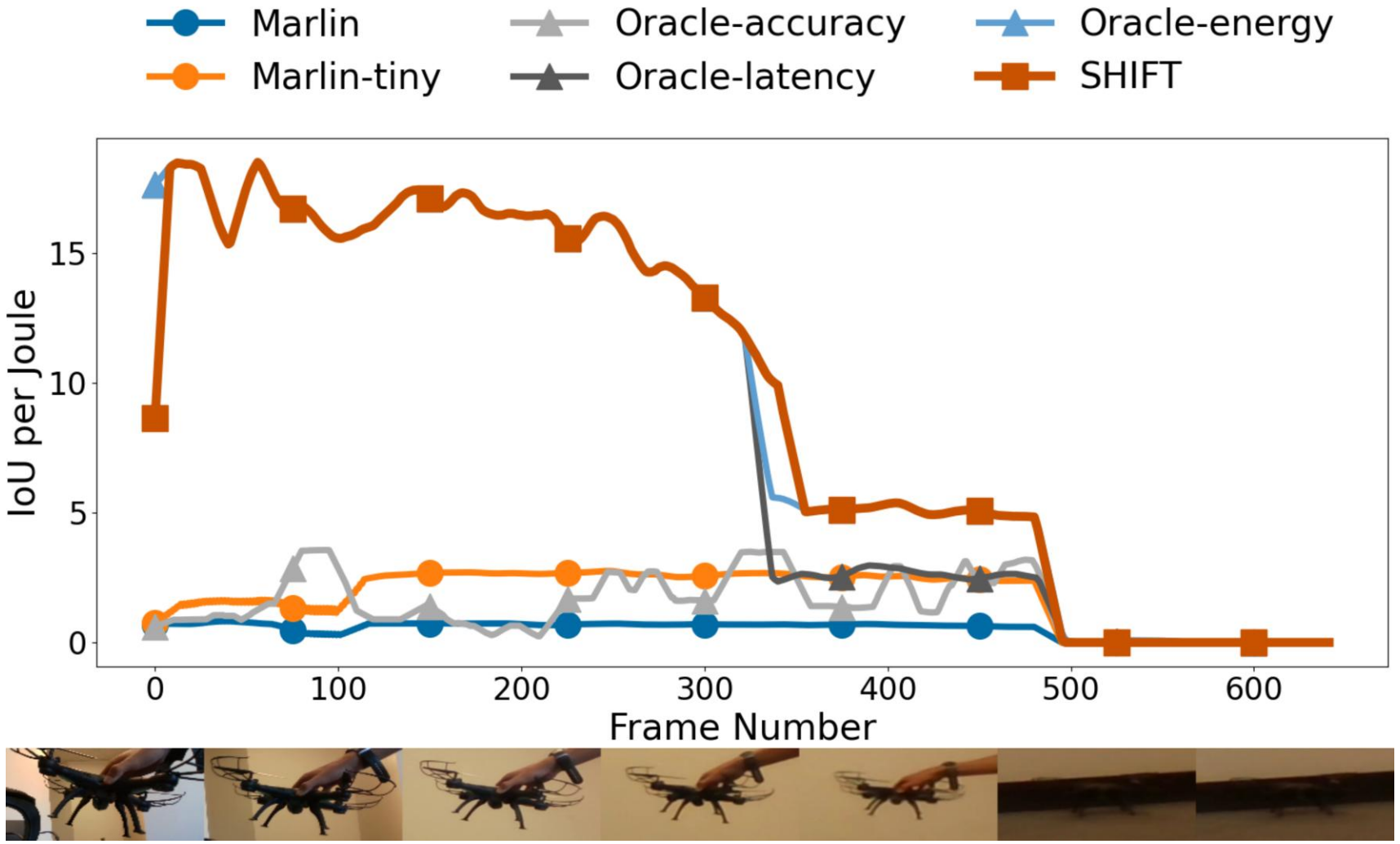
SHIFT - Scenario 3



SHIFT - Scenario 4



SHIFT - Scenario 5

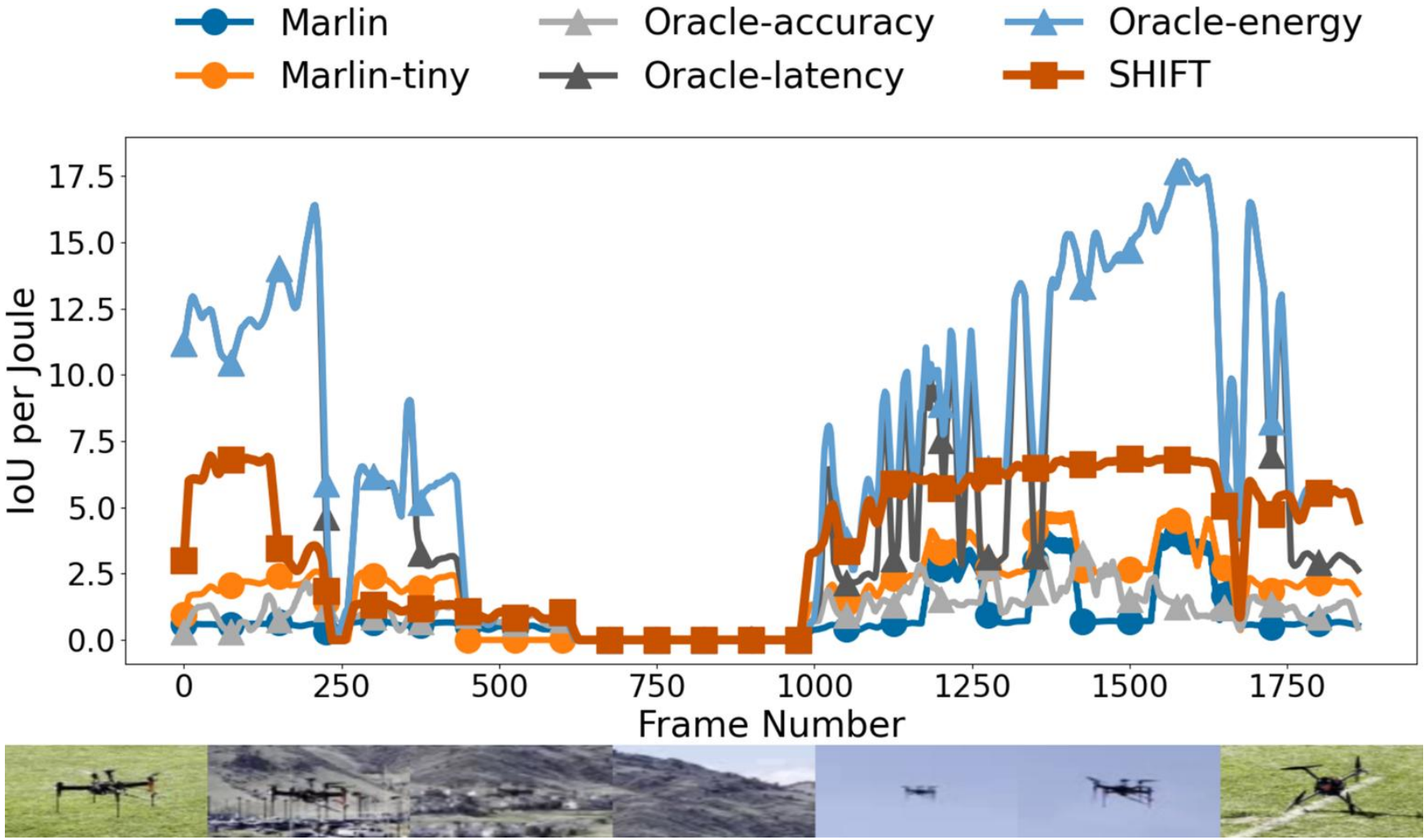


Results - All Models

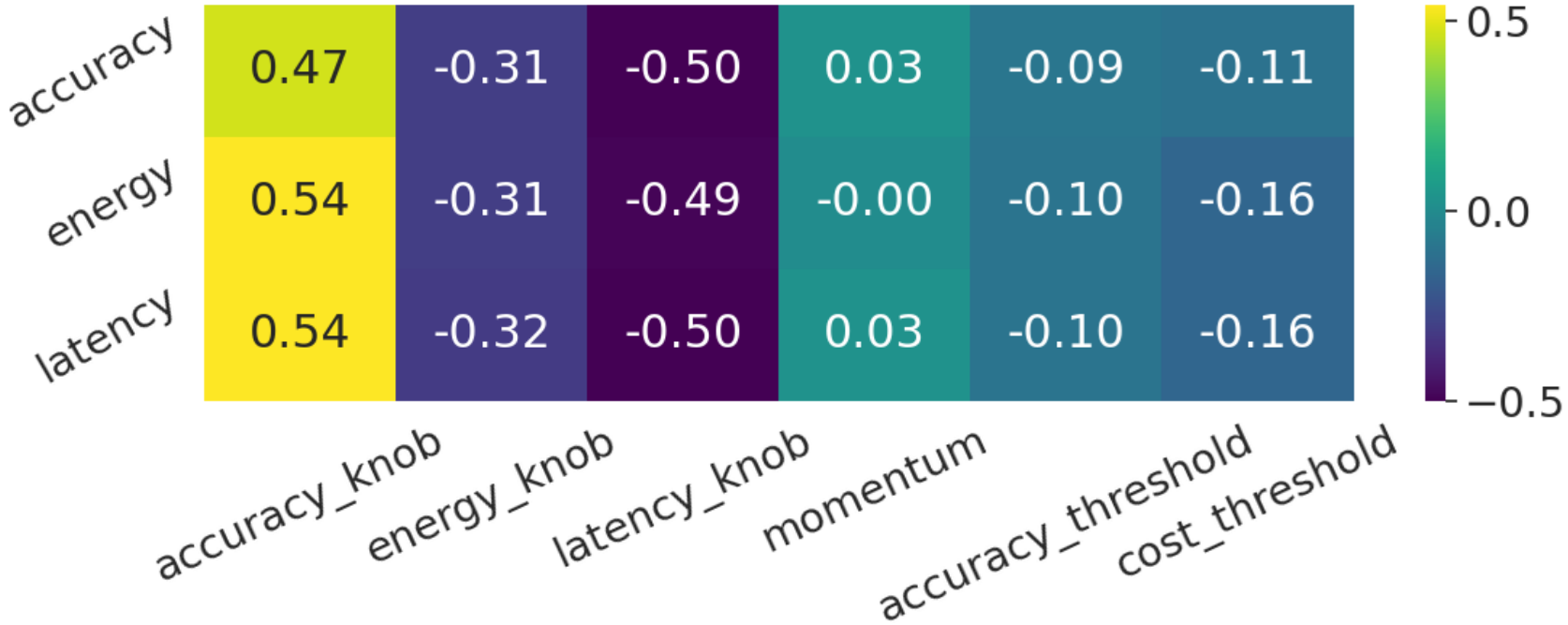
Model Name	Accuracy		Avg. Time (s)			Avg. Energy (Joules)			Avg. Power Draw (W)		
	Avg. IoU	Success Rate	GPU	GPU/DLA	OAK-D	GPU	GPU/DLA	OAK-D	GPU	GPU/DLA	OAK-D
YoloV7-E6E	0.564	65.8%	0.255	0.221	-	3.947	1.228	-	15.48	5.56	-
YoloV7-X	0.593	71.1%	0.222	0.195	-	3.586	1.088	-	16.15	5.57	-
YoloV7	0.618	74.1%	0.130	0.118	0.894	1.968	0.656	1.391	15.14	5.56	1.56
YoloV7-Tiny	0.533	64.0%	0.025	0.024	0.107	0.280	0.134	0.206	11.2	5.58	1.93
SSD Resnet50	0.480	58.9%	0.151	0.138	-	2.504	0.816	-	16.58	5.91	-
SSD MobilenetV1	0.452	55.4%	0.094	0.092	-	1.519	0.561	-	16.16	6.10	-
SSD MobilenetV2	0.401	51.3%	0.023	0.058	-	0.248	0.307	-	10.78	5.29	-
SSD MobilenetV2 320x320	0.304	36.2%	0.009	0.023	-	0.046	0.100	-	5.11	4.35	-

YoloV7 on GPU achieves highest success rate across our tests. This model serves as our baseline

Scenario


Marlin
Marlin-tiny
Oracle-accuracy
Oracle-latency
Oracle-energy
SHIFT
IoU per Joule
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
250
500
750
1000
1250
1500
1750
Frame Number

SHIFT - Sensitivity Analysis



1. Knobs have correct relationship relative to each metric
2. Momentum (filtering results) does not have significant impact
3. Reducing cost threshold (using closer nodes only) has positive impacts

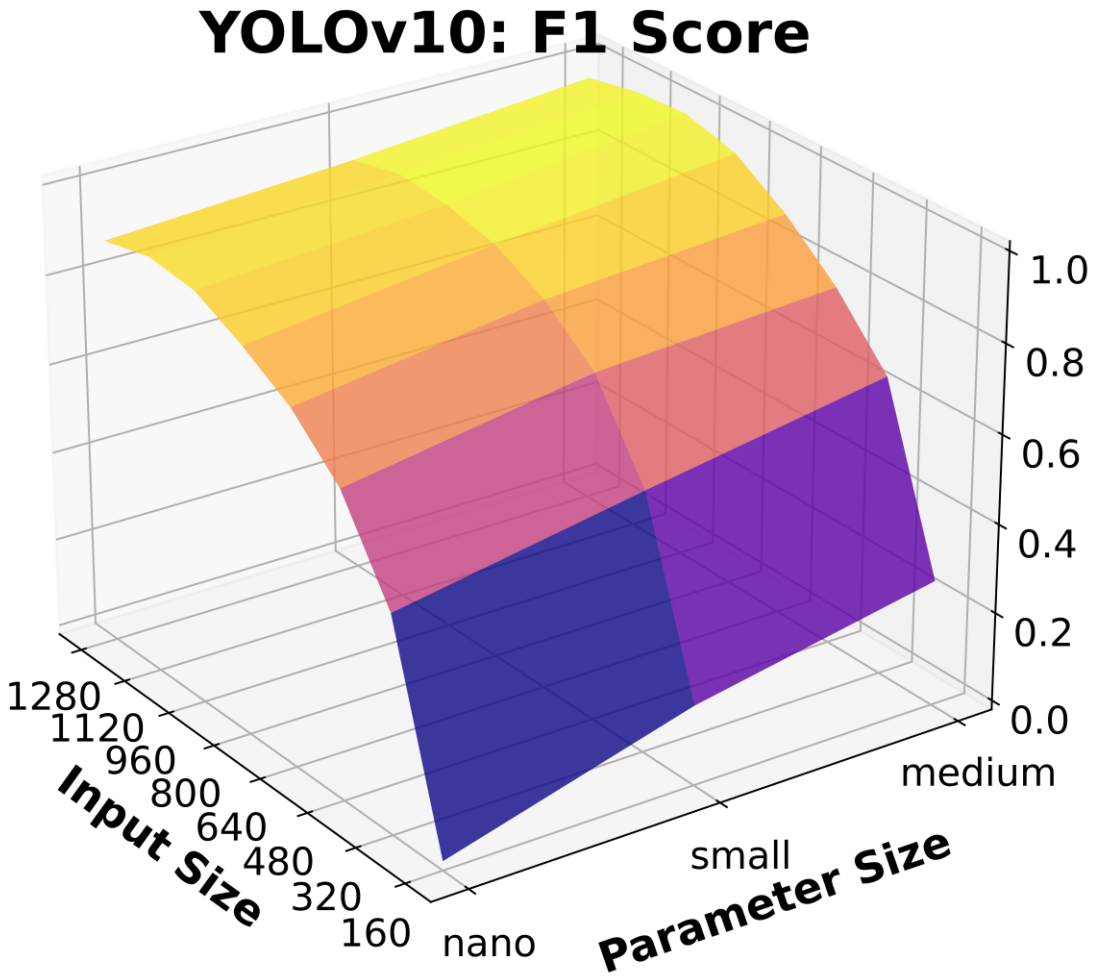
PMOTHS

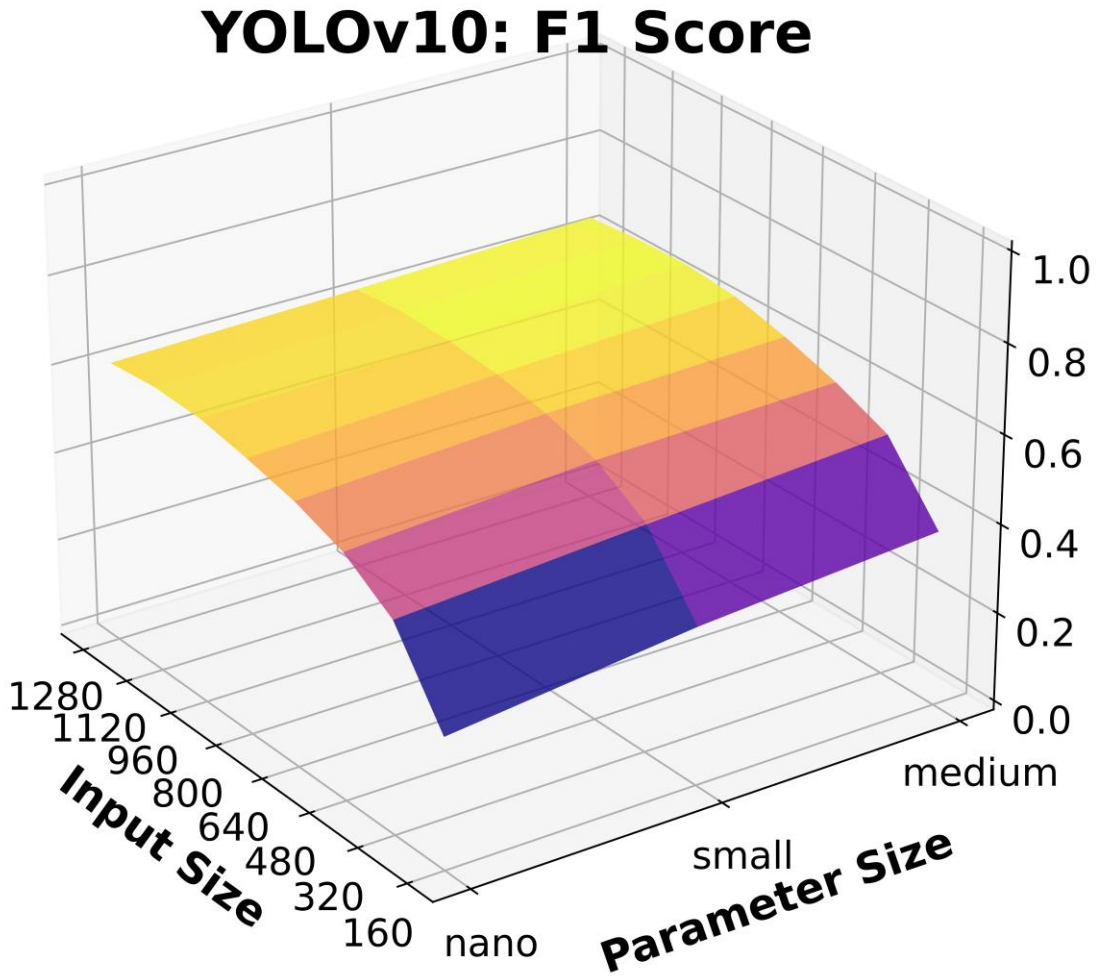
PMOTHS – Related Work

Feature \ Related Work	HaxCoNN [4]	CARin [27]	BigLittle [28]	NestDNN [10]	CACTUS [29]	SHIFT [6]	FlexPatch [37]	*PMOTHS*
Context Aware	✗	✗	✓	✗	✓	✓	✓	✓
Accuracy Predictions	✗	✗	✗	✗	✗	✓	✗	✓
Concurrent DNNs	✓	✓	✗	✗	✗	✗	✗	✓
Non-GPU Accelerators	✓	✓	✗	✗	✗	✓	✗	✓
Object Detection	✗	✗	✗	✗	✗	✓	✓	✓
No Additional Training	✓	✓	✓	✗	✗	✓	✓	✓
ROI Focusing	✗	✗	✗	✗	✗	✗	✓	✓
Low Overhead	✓	✗	✗	✓	✗	✓	✗	✓

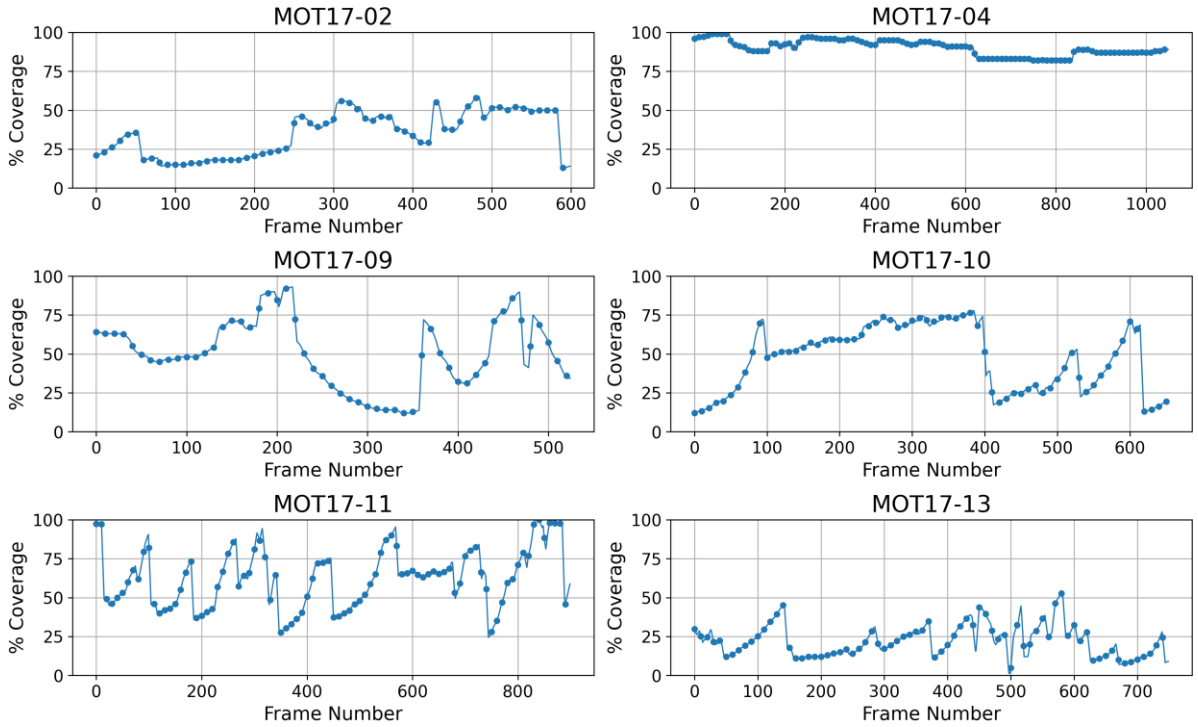
Table 1: Comparison of the features offered by related works.

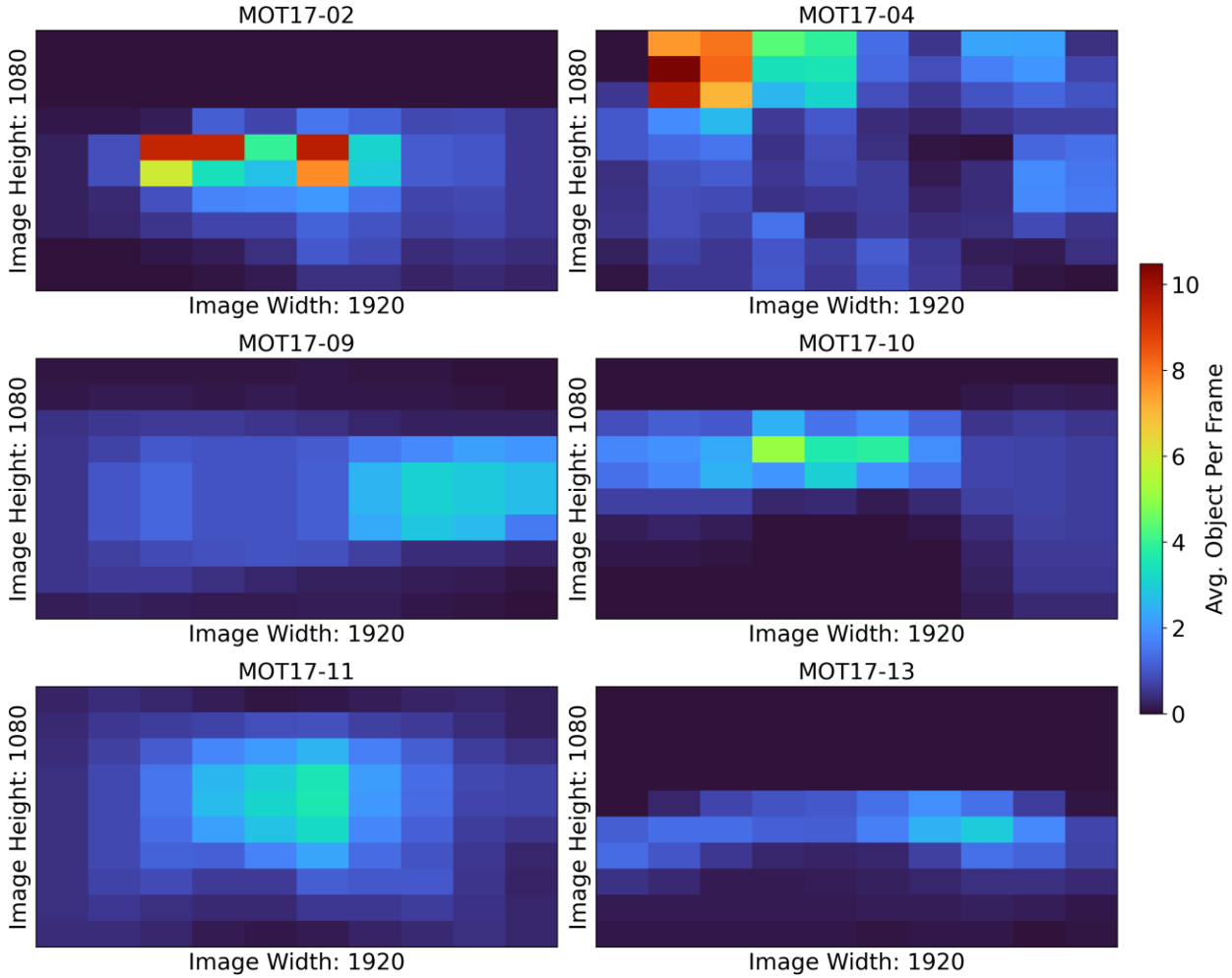
PMOTHS – F1 vs. Input/Parameter Size



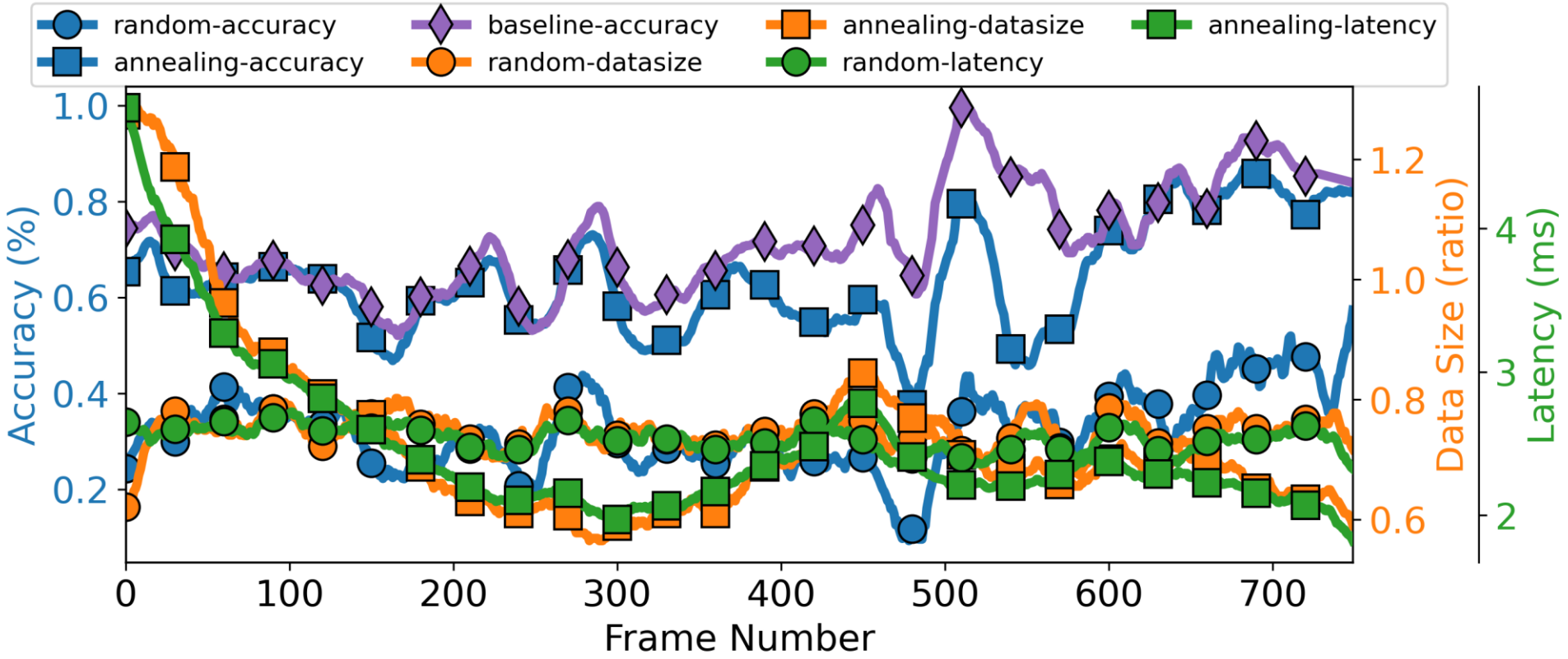


Spatial Analysis of Objects in MOT17





PMOTHS – Framepacking Analysis



PMOTHS – Algorithms

Algorithm 1 Frame Packing via Shelf Bin Packing

Input: Image I, Groups G
Output: Packed Image P

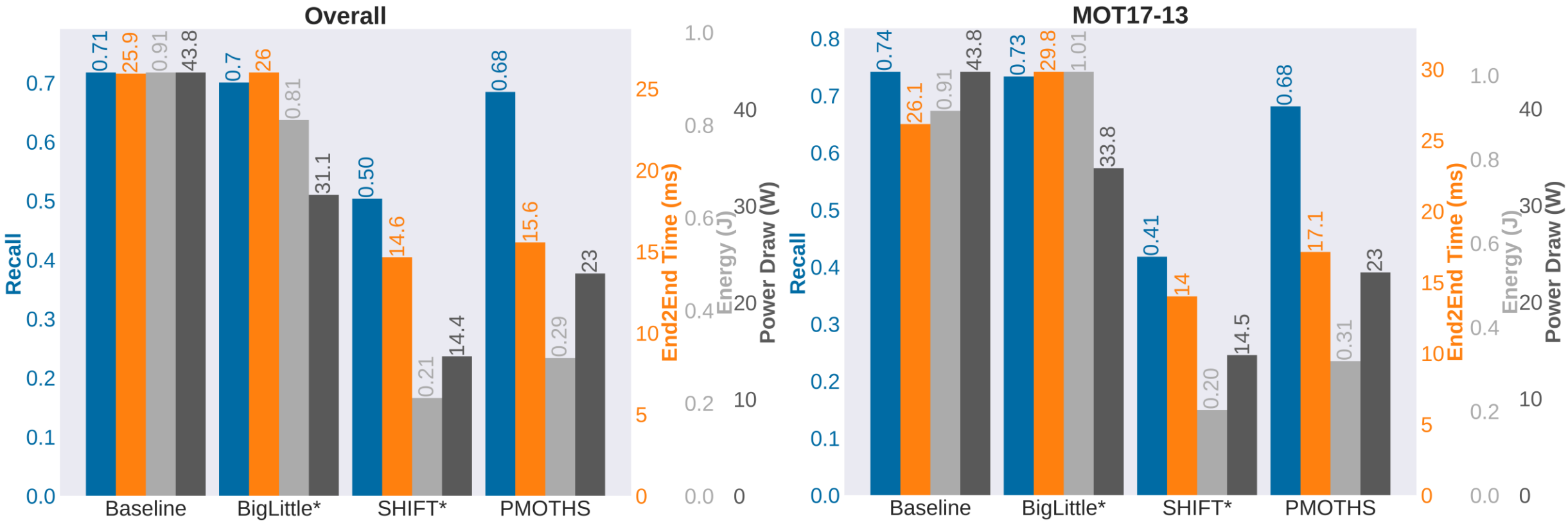
if $G = \emptyset$ **then**
  return Empty Image
end if
$G \leftarrow \text{sort}(G)$ ▷ Sort descending by size
$ts \leftarrow \text{ceil}(\text{sqrt}(\text{sum}(\text{size}(g) \forall g \in G)))$ ▷ Target size
$S \leftarrow \emptyset$ ▷ Set of shelves
for all $g \in G$ **do**
  for all $s \in S$ **do**
    if $g_{\text{width}} \neq s_{\text{width}} \vee g_{\text{height}} + s_{\text{height}} > ts$ **then**
      continue
    end if
    $s \leftarrow s \cap \{g\}$
    break
  end for
  $S \leftarrow S \cup \{g\}$
end for
$h \leftarrow \max(s_{\text{height}} \forall s \in S)$
$w \leftarrow \text{sum}(s_{\text{width}} \forall s \in S)$
$P = image[h, w]$
$frontier \leftarrow 0$
for all $s, s_w, s_h \in S$ **do**
  for all $i \in s_h$ **do**
    for all $j \in s_w$ **do**
      $\text{bbox} \leftarrow s[i \times s_w + j]_{\text{bbox}}$ ▷ Get bbox from cell
      $\text{bbox}_{\text{new}} \leftarrow \text{getBbox}(i, j)$ ▷ Get new bbox
      $P[\text{bbox}_{\text{new}}] \leftarrow I[\text{bbox}]$ ▷ Update image
    end for
  end for
  $frontier \leftarrow frontier + s_w$
end for
return P

Algorithm 2 Runtime Scheduler

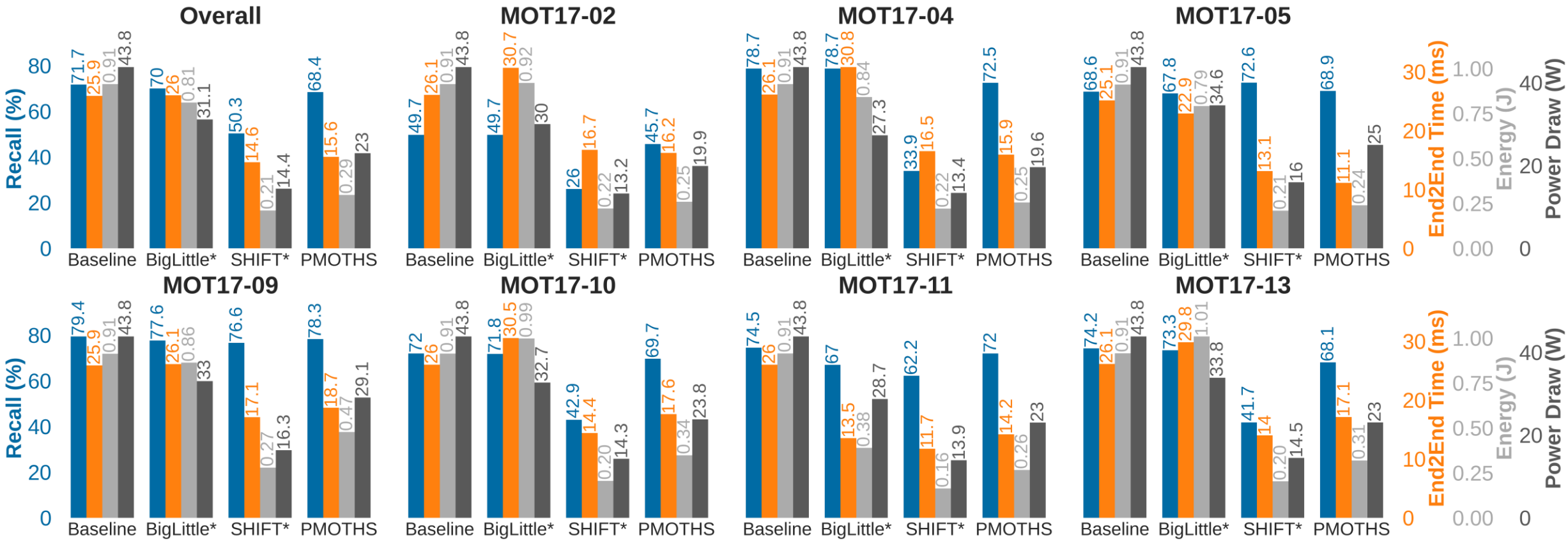
Input: Previous Detections D, LP accuracy goal γ_{lp}, Possible Schedules S, HP Bounding box B
Output: New schedule configuration $C \in S$

if $\mathcal{D} = \emptyset$ **or** B is undefined **then**
  return C_{baseline} ▷ Exit without changes
end if
Compute HP bounding box area $A_{\text{hp}} \leftarrow \text{area}(B)$
Compute HP Confidence $c_{\text{hp}} \leftarrow$ mean confidence in D_{hp}
Predict HP F1 $P_{hp} \leftarrow \text{predict}(D, c_{hp})$
Get baseline scaling ratio $s_b \leftarrow b_m / A_{hp}$
Get median bounding box size $s_{\text{bbox}} \leftarrow \text{medianSize}(D)$
Compute scale $b_s \leftarrow 1.0 - (s_{\text{bbox}} / \max(\text{img}_{\text{height}}, \text{img}_{\text{width}}))$
Initialize $\mathcal{M}_{\text{hp}}^{\text{valid}} \leftarrow \emptyset$
for all $m \in \mathcal{M}_{\text{hp}}$ **do**
  Retrieve model area $A_m \leftarrow$ input area of m
  Compute scaling ratio $s_m \leftarrow A_m / A_{\text{hp}}$
  if $s_m \leq s_b * b_s$ & $P_{\text{hp}}[m] \geq P_{\text{hp}}[\text{baseline}]$ **then**
    $\mathcal{M}_{\text{hp}}^{\text{valid}} \leftarrow \mathcal{M}_{\text{hp}}^{\text{valid}} \cup \{m\}$
  end if
end for
if $\mathcal{M}_{\text{hp}}^{\text{valid}} \neq \emptyset$ **then**
  $m_{\text{hp}}^{\text{new}} \leftarrow \min_{m \in \mathcal{M}_{\text{hp}}^{\text{valid}}} \text{latency}(m)$
else
  return $C_{baseline}$
end if
$\mathcal{S}_{\text{filtered}} \leftarrow \{s \in S : \text{HP model in } s = m_{\text{hp}}^{\text{new}}\}$
Compute LP Confidence $c_{\text{lp}} \leftarrow$ mean confidence in D_{lp}
Predict LP F1 $P_{\text{lp}} \leftarrow \text{predict}(D, c_{\text{lp}})$
Initialize $\mathcal{S}_{\text{final}} \leftarrow \emptyset$
for all $s \in \mathcal{S}_{\text{filtered}}$ **do**
  Retrieve LP model $m_{\text{lp}} \leftarrow$ LP model in s
  if $P_{\text{lp}}[m_{\text{lp}}] \geq \gamma_{lp}$ & $\text{latency}(m_{\text{lp}}) + \text{latency}(m_{\text{hp}}) < \text{latency}(m_{\text{baseline}})$ **then**
    $\mathcal{S}_{\text{final}} \leftarrow \mathcal{S}_{\text{final}} \cup \{s\}$
  end if
end for
if $\mathcal{S}_{\text{final}} = \emptyset$ **then**
  return C_{baseline}
else
  $s_{\text{new}} \leftarrow \min_{s \in \mathcal{S}_{\text{final}}} \text{latency}(s)$
end if
return s_{new}

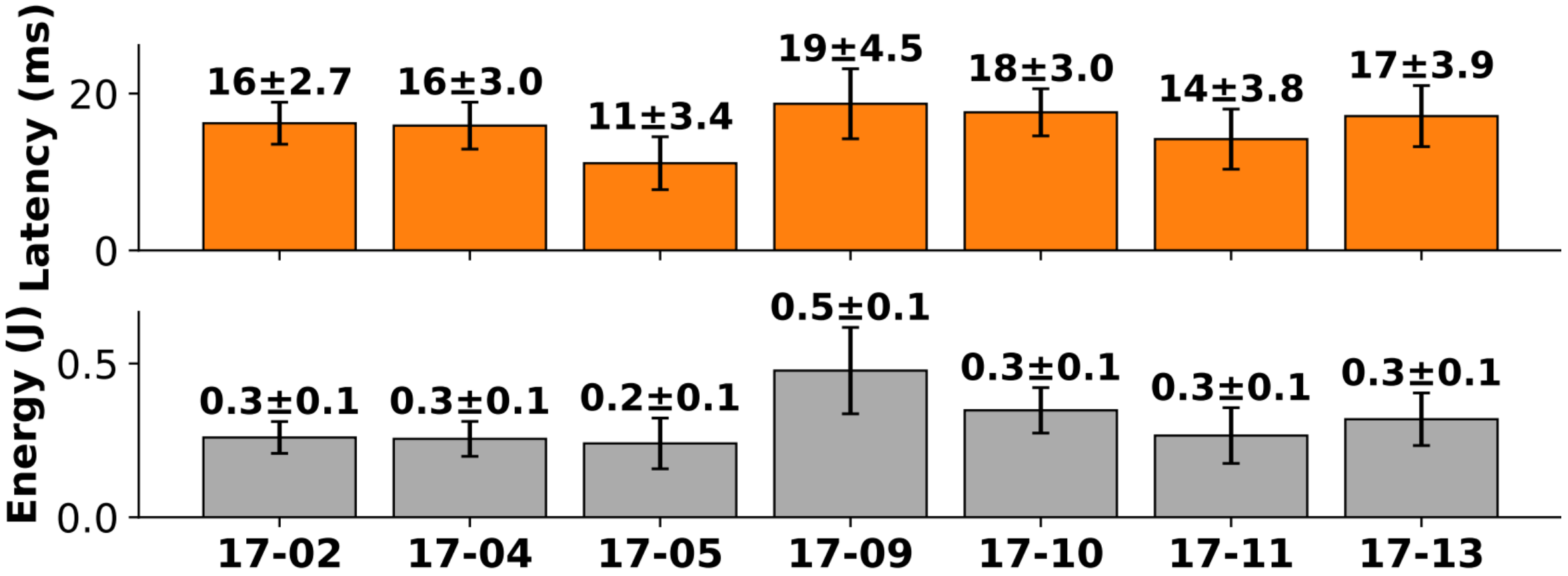
MOT17 Results



PMOTHS – Result Matrix



PMOTHS – Variance per Sequence



PMOTHS – Schedules

Schedule	17-02	17-04	17-05	17-09	17-10	17-11	17-13
Baseline	0.17%	0.10%	0.12%	20.9%	0.15%	3.67%	0.13%
GPU+GPU	63.3%	74.4%	72.0%	45.1%	62.8%	58.4%	60.1%
GPU+DLA	36.5%	25.5%	27.8%	33.9%	37.0%	37.9%	39.7%

Table 3: Scheduling decision distribution across GPU+GPU, GPU+DLA executions, and baseline single model fallback on each MOT17 scenario.

Metric	0.4	0.5	0.6	0.7	0.8	0.9
Time (ms)	15.2	15.6	16.8	17.8	19.3	20.3
Energy (J)	0.29	0.30	0.36	0.41	0.48	0.54
Powerdraw (W)	23.3	23.0	24.9	26.4	28.4	29.8
DLA Usage	66.3%	33.5%	13.4%	9.5%	7.5%	5.5%
Baseline Usage	2.8%	2.8%	13.6%	23.9%	41.5%	53.7%

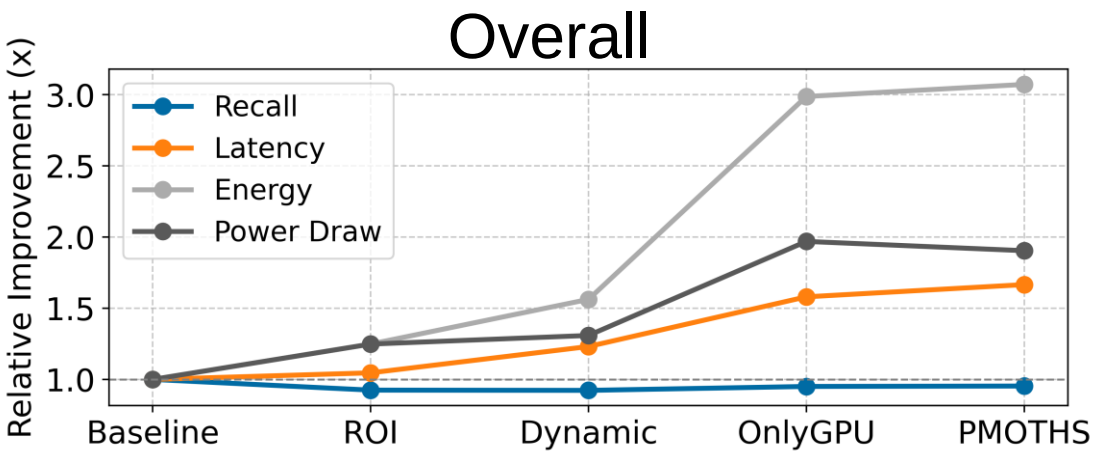
Table 4: Varying LP accuracy goal values and the effect on runtime metrics and schedules.

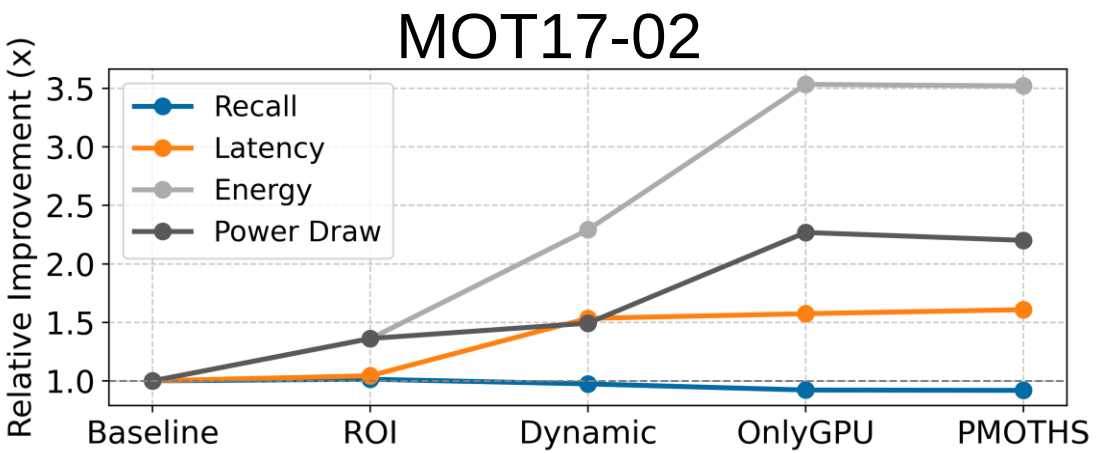
PMOTHS – All Models

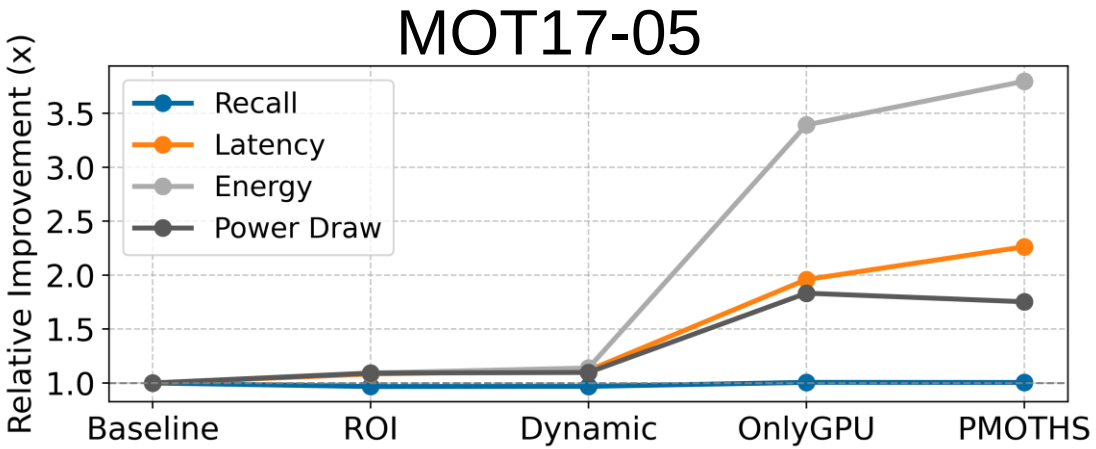
Model Info				Metrics	Input Size							
Model	**Param**	**Device**	**Precision**		**160**	**320**	**480**	**640**	**800**	**960**	**1120**	**1280**
YOLOv10[33]	M	GPU	FP16	Recall	0.4x	0.64x	0.75x	0.83x	0.91x	0.96x	0.99x	**1.0x**
				Latency	3.48ms	4.97ms	6.98ms	9.45ms	12.31ms	15.84ms	21.21ms	**25.91ms**
				Energy	0.04J	0.07J	0.14J	0.24J	0.37J	0.54J	0.73J	**0.91J**
				Power	17.65W	24.52W	31.29W	35.57W	40.2W	42.0W	43.73W	**43.8W**
	S	GPU	FP16	Recall	0.36x	0.59x	0.72x	0.8x	0.87x	0.92x	0.96x	0.98x
				Latency	2.94ms	3.88ms	4.99ms	6.31ms	8.69ms	10.04ms	14.01ms	16.66ms
				Energy	0.02J	0.04J	0.07J	0.11J	0.18J	0.25J	0.34J	0.44J
				Power	15.13W	20.26W	25.27W	29.23W	32.11W	35.51W	36.08W	37.91W
	N	GPU	FP16	Recall	0.28x	0.53x	0.66x	0.75x	0.82x	0.87x	0.92x	0.93x
				Latency	2.82ms	3.63ms	4.35ms	5.02ms	6.68ms	7.51ms	10.59ms	12.22ms
				Energy	0.02J	0.03J	0.04J	0.06J	0.09J	0.13J	0.18J	0.23J
				Power	13.08W	16.41W	20.44W	23.1W	26.24W	28.45W	29.87W	31.66W
YOLOX [11]	M	DLA	INT8	Recall	0.54x	0.97x	0.99x	1.0x	0.95x	0.96x	0.86x	0.82x
				Latency	3.18ms	4.51ms	7.08ms	11.21ms	16.42ms	22.87ms	30.46ms	40.05ms
				Energy	0.03J	0.04J	0.08J	0.13J	0.21J	0.29J	0.4J	0.55J
				Power	12.72W	13.4W	14.67W	14.79W	14.74W	14.69W	14.75W	15.24W
	S	DLA	INT8	Recall	0.53x	0.95x	0.92x	0.94x	0.95x	0.43x	0.34x	0.24x
				Latency	2.69ms	3.42ms	4.91ms	7.41ms	10.52ms	14.23ms	18.9ms	23.76ms
				Energy	0.02J	0.03J	0.05J	0.08J	0.12J	0.17J	0.23J	0.3J
				Power	11.96W	12.71W	14.66W	15.18W	15.08W	15.15W	15.3W	15.32W

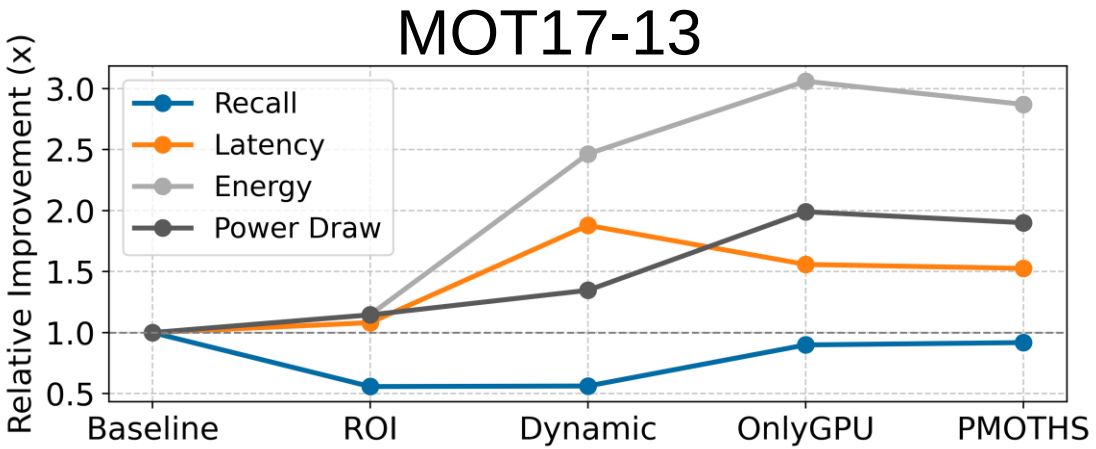
Table 5: Performance comparision of YOLOv10 and YOLOX models across all MOT17 sequence. All metrics are reported as the average for each frame across all sequences. Power draw is reported as the total system draw to run the model.

PMOTHS – Ablation Study



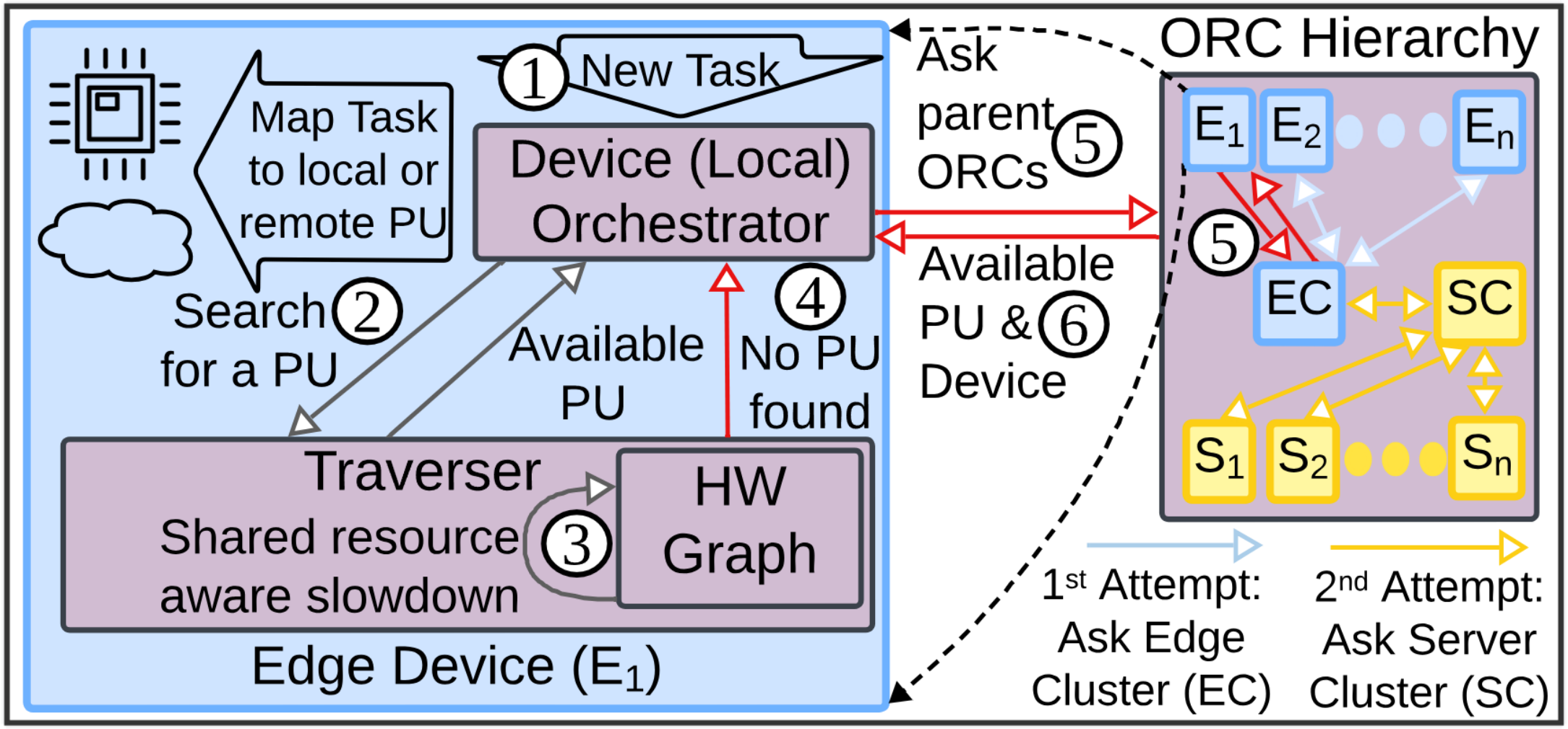




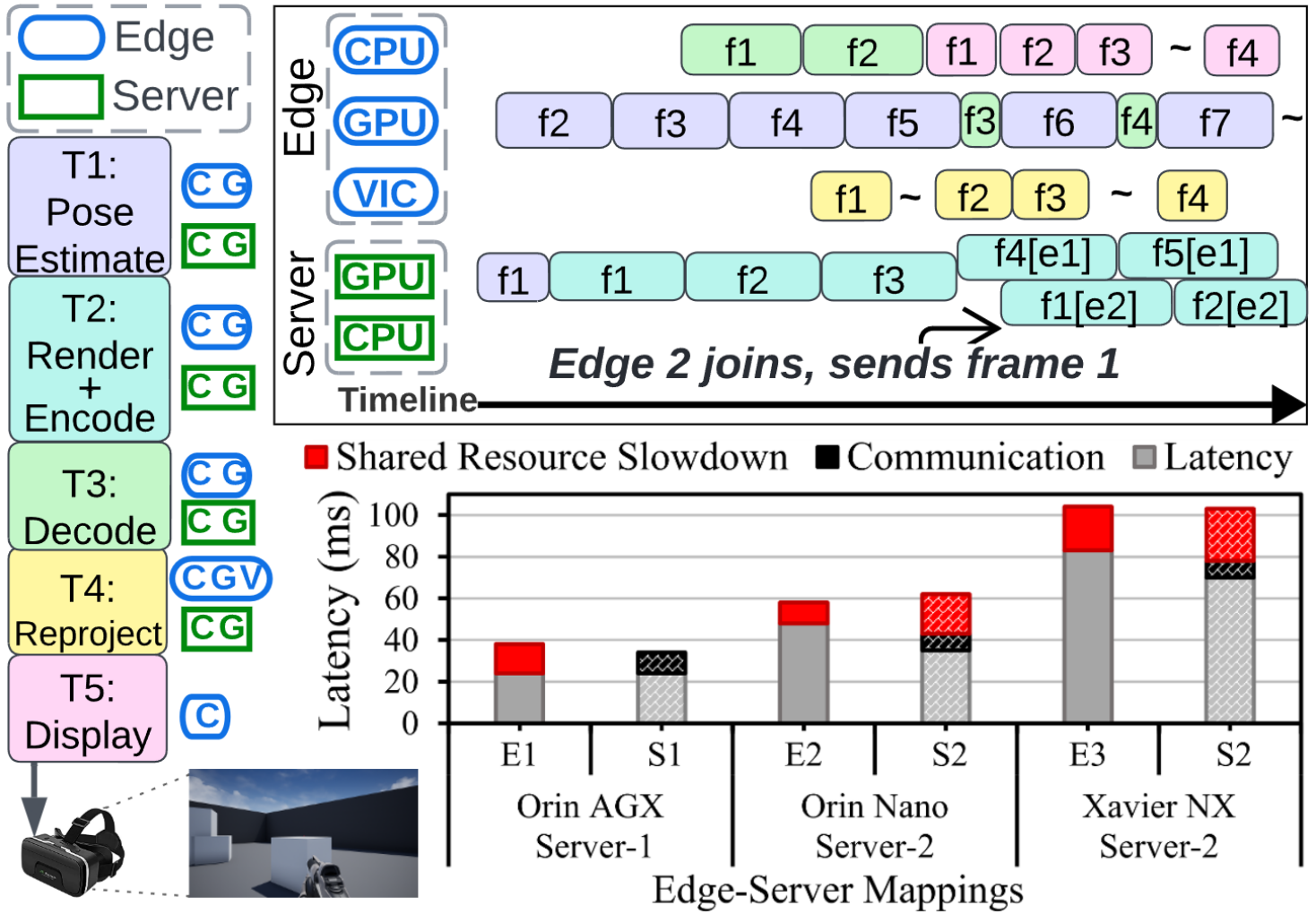


HARNESS

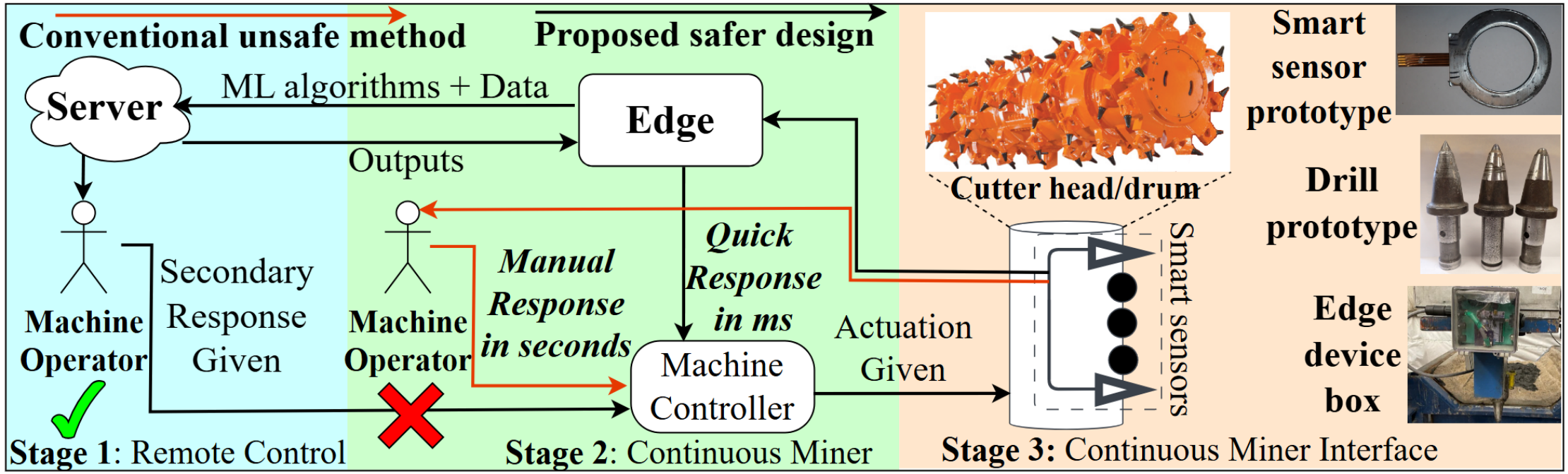
HARNESS – Overview

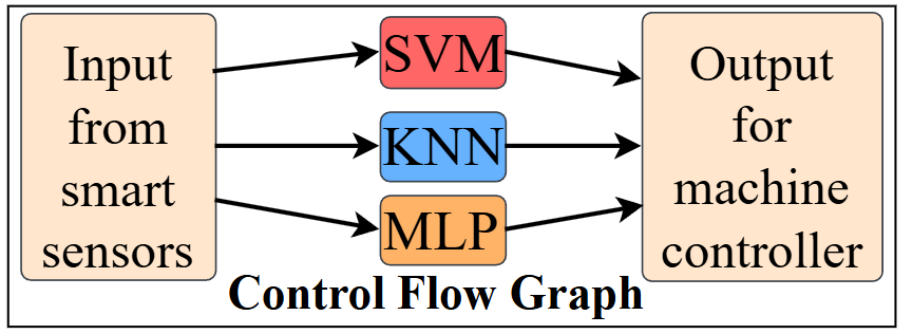


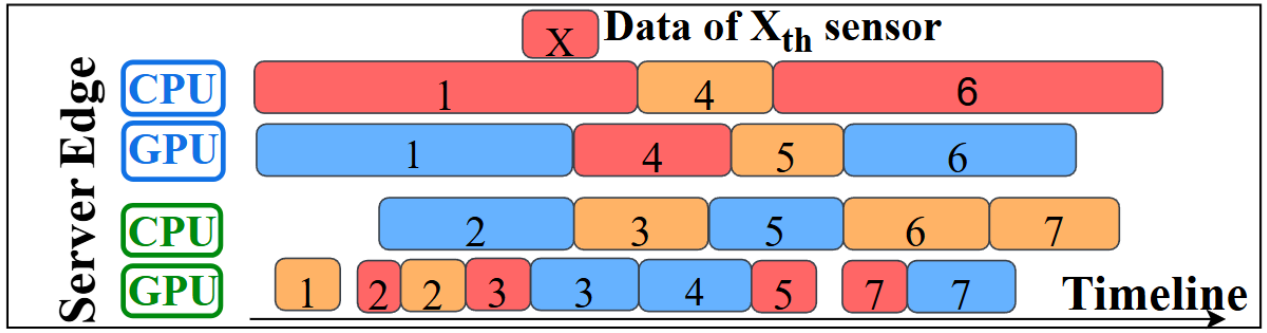
HARNESS – Example: CloudVR



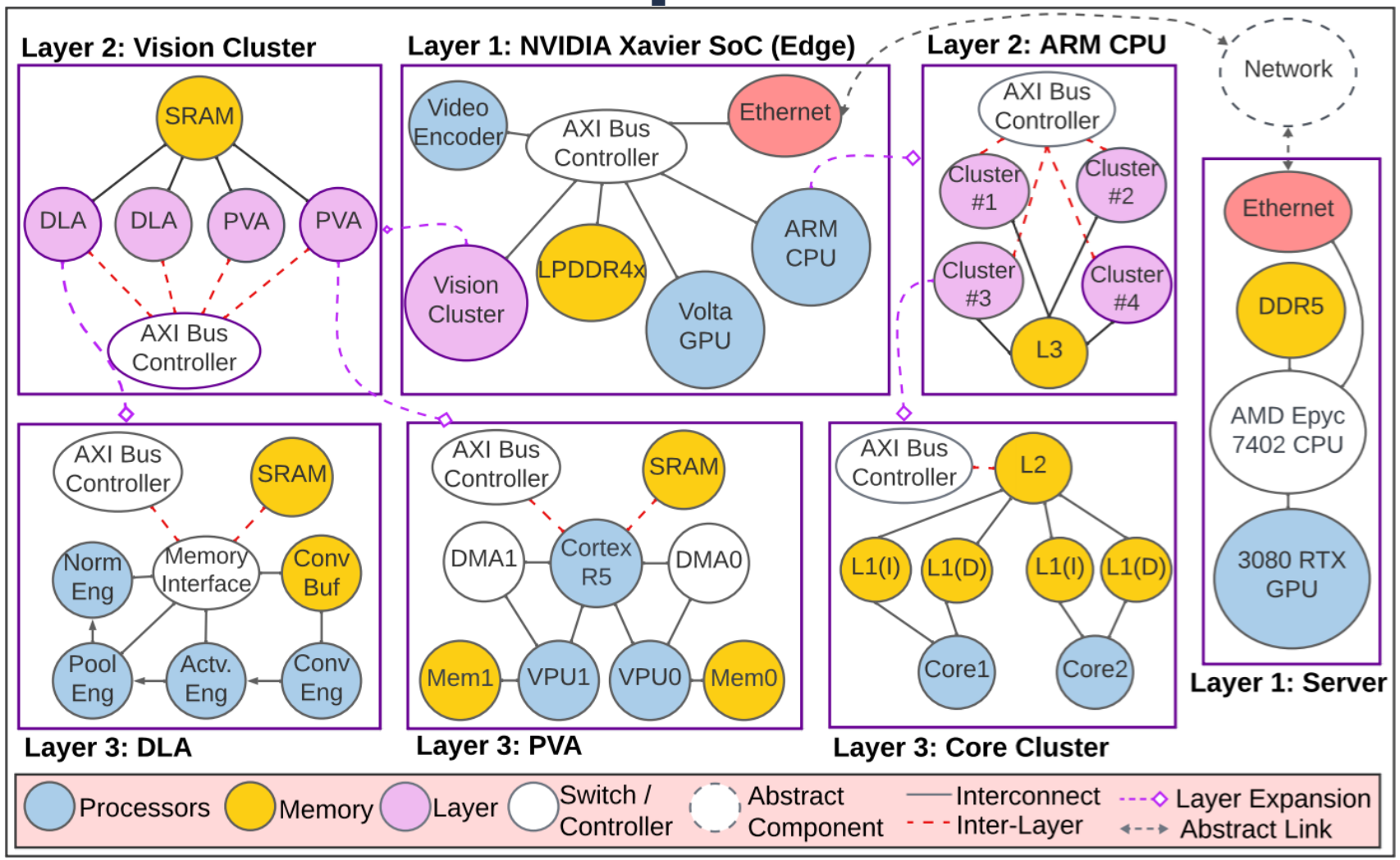
HARNESS – Example: Mining



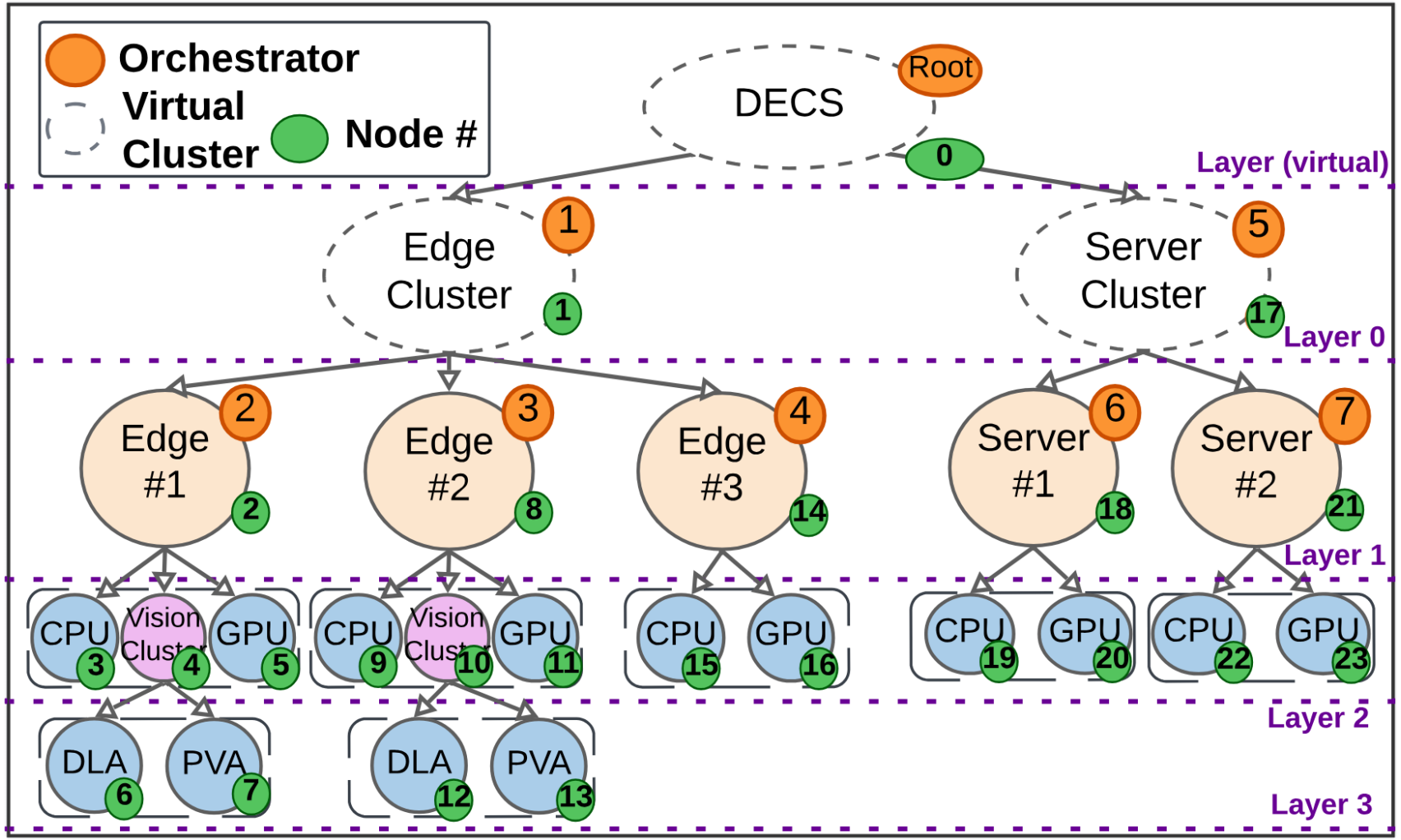




HARNESS – Example HW-GRAPH – P1



HARNESS – Example HW-GRAPH – P2



HARNESS – Latency Comparison

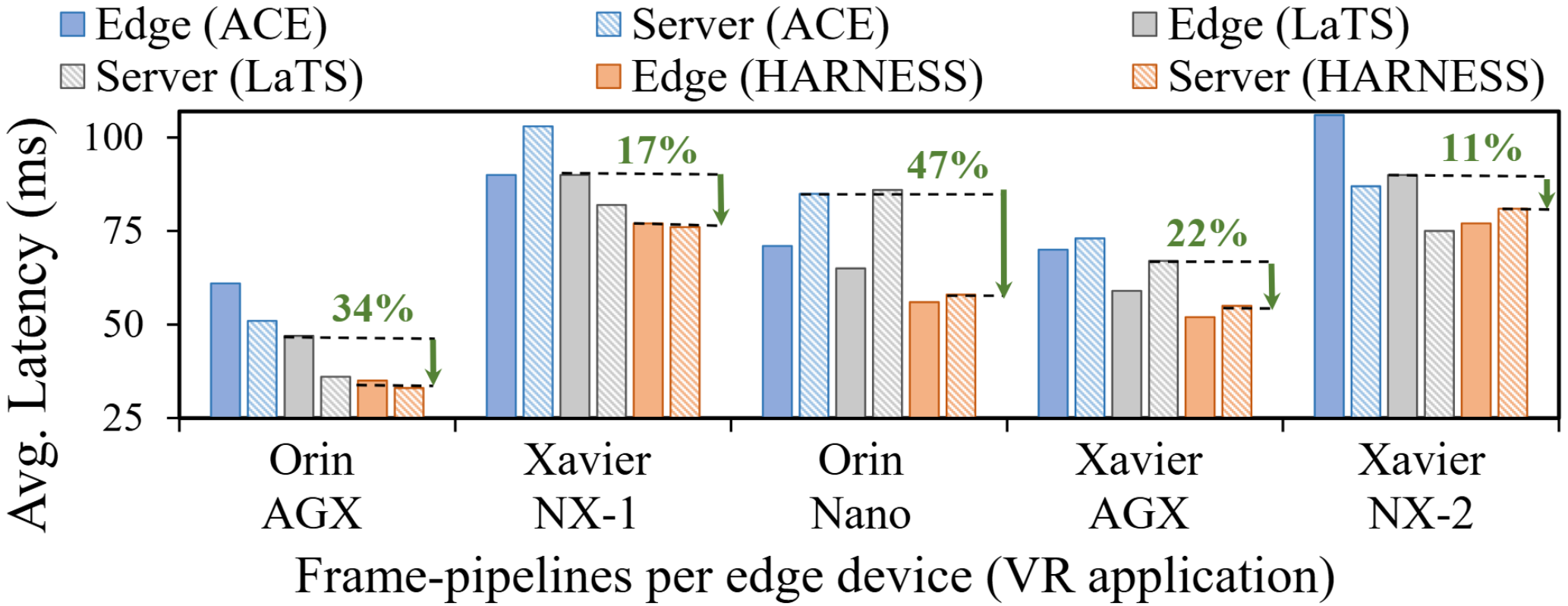


Figure 9: Average overall latency of *HARNESS* and others.

HARNESS – Accuracy Comparison

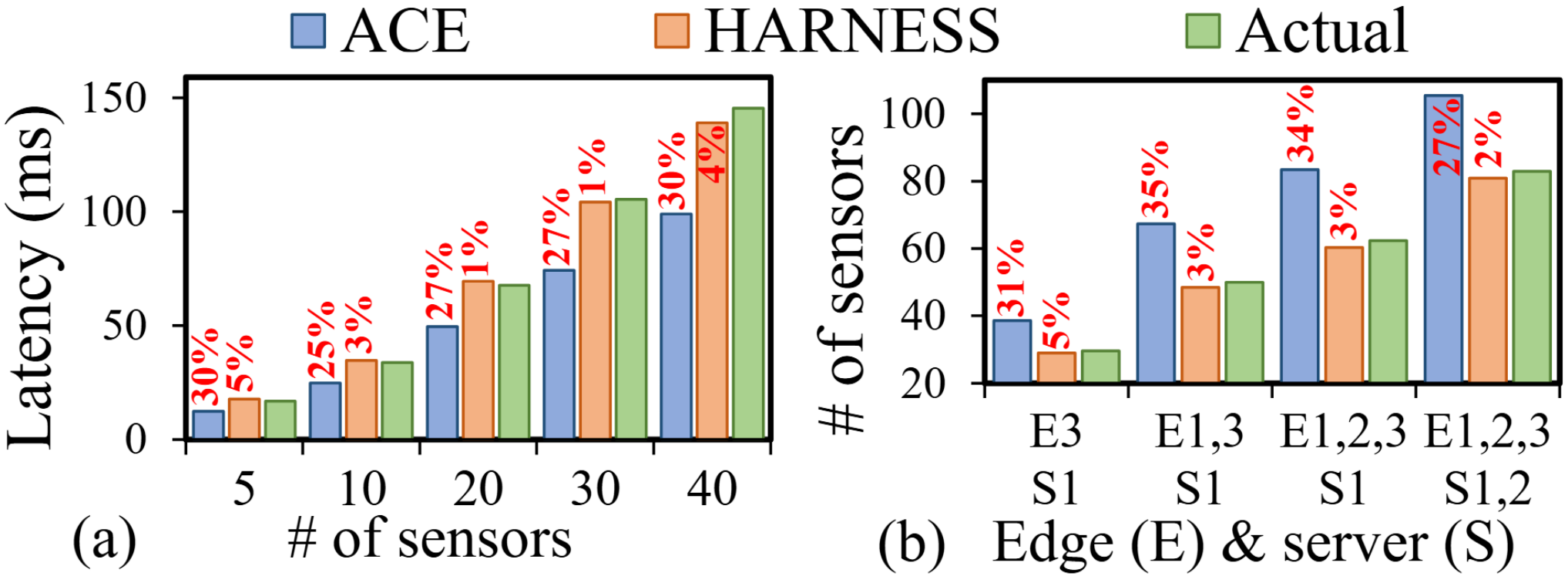
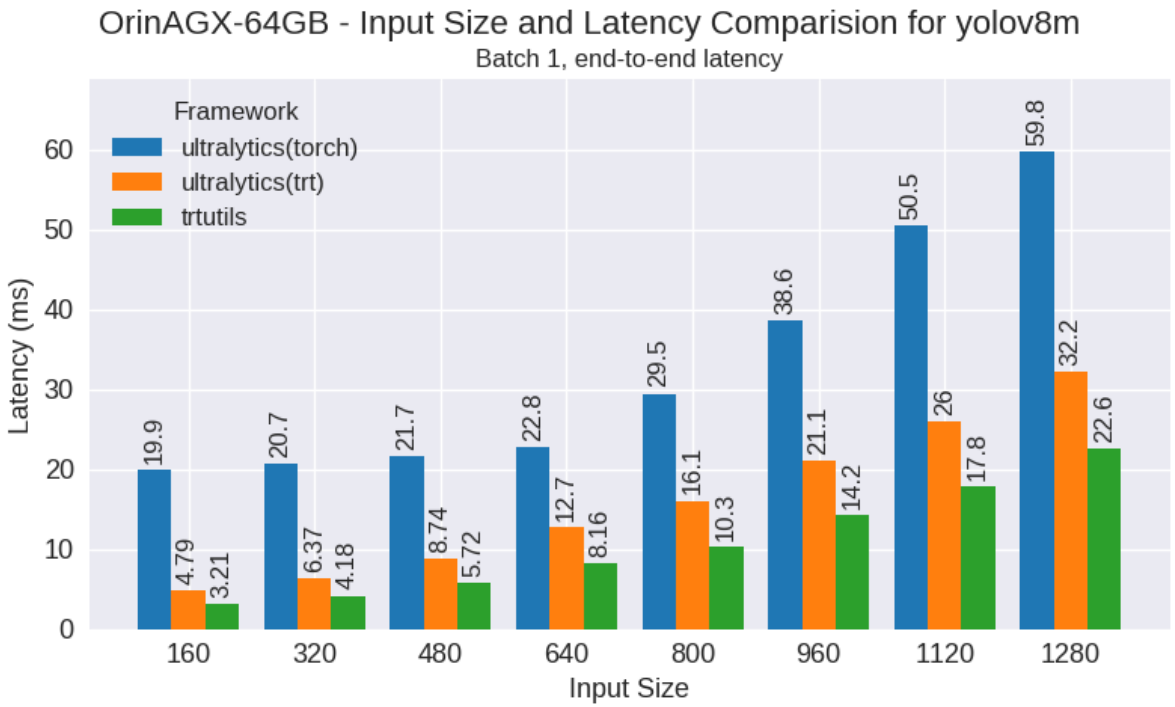
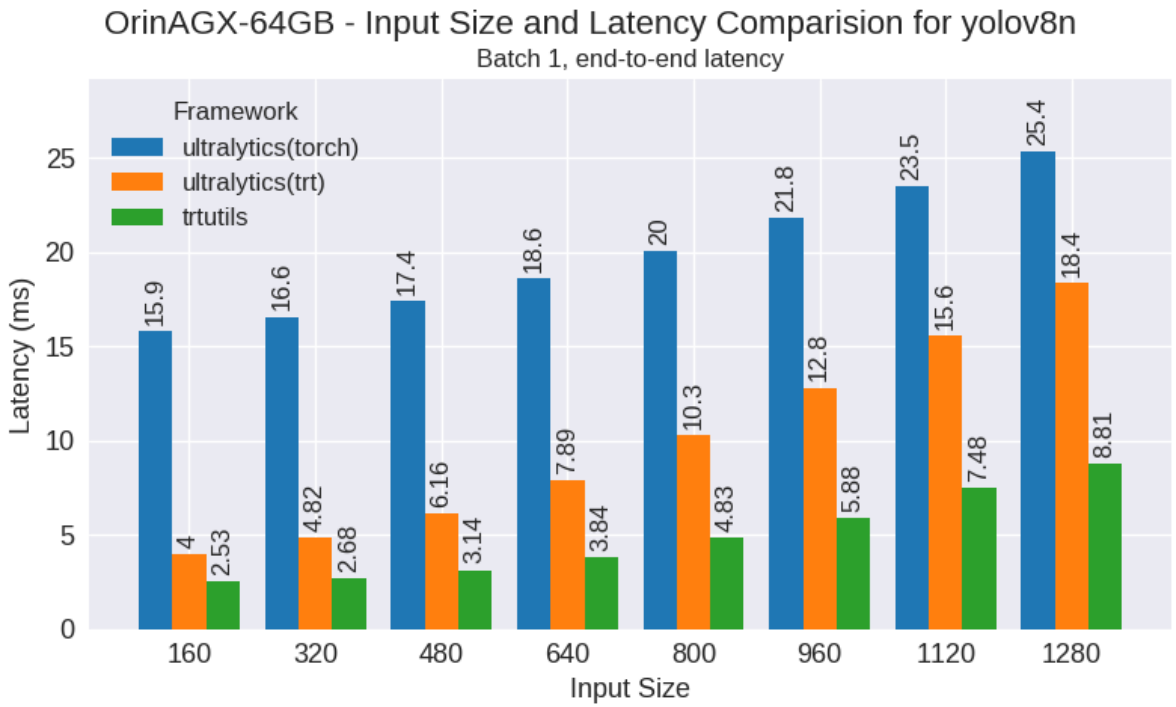


Figure 10: Prediction error rates of ACE & *HARNESS* for a system with (a) 1 edge device + server, (b) increased # devices.

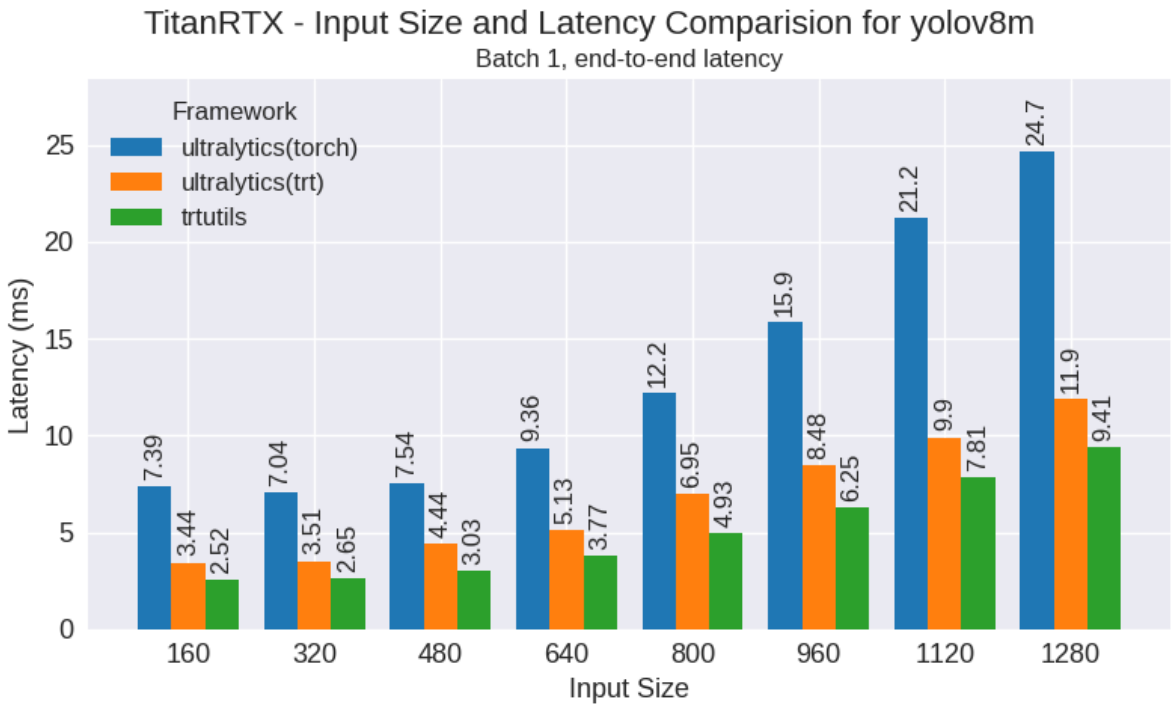
TRTUTILS

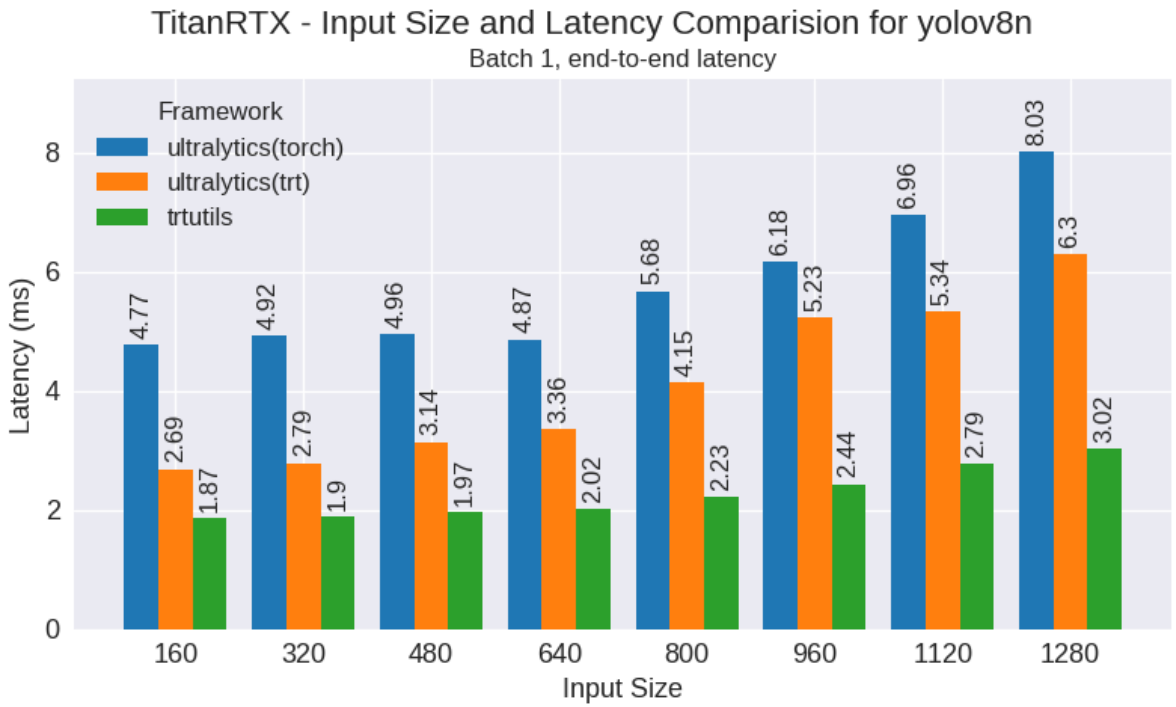
Performance – Orin AGX 64GB





Performance – TitanRTX





Programming Language

Python	C++
• Simple • Slow • No build system • GIL • Managed memory	• Moderate • Fast • Requires build system • Free-threaded

Device Support

	JetPack 4 CUDA 10	JetPack 5 CUDA 11	JetPack 6 CUDA 12
Jetson Nano	✅	❌	❌
Jetson TX2	✅	❌	❌
Jetson Xavier NX	✅	✅	❌
Jetson Xavier AGX	✅	✅	❌
Jetson Orin AGX	❌	✅	✅
Jetson Orin NX	❌	✅	✅
Jetson Orin Nano	❌	✅	✅

- JetPack Bundles
  - Linux Kernel
  - Python
  - CUDA
  - TensorRT
  - cuDNN
  - VPI
- Support majority of CUDA and TensorRT enabled systems with least effort

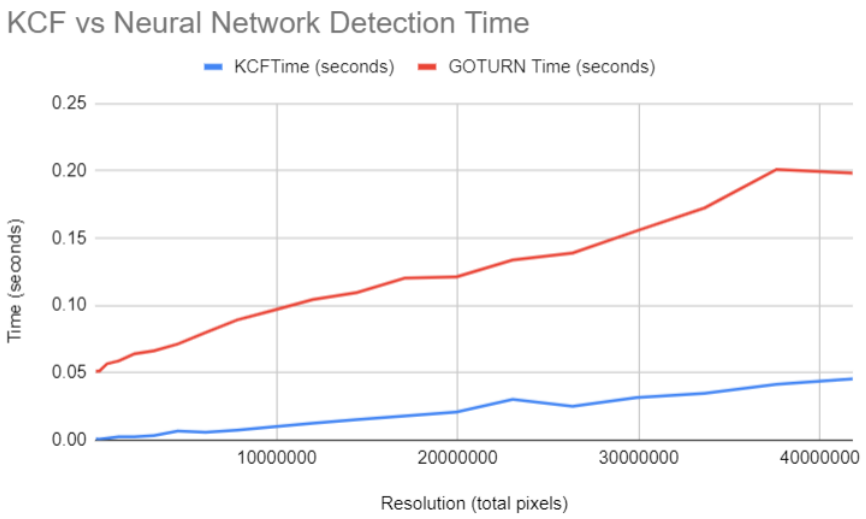
Examples

```
kernel_bytes = kernel.encode()
kernel_name_bytes = f"{name}.cu".encode()

if verbose:
    LOG.debug(f"Compiling kernel: {name}")

# compile the kernel
with _NVRTC_LOCK, _MEM_ALLOC_LOCK:
    try:
        prog = nvrtc_call(
            nvrtc.nvrtcCreateProgram(kernel_bytes, kernel_name_bytes, 0, [], []),
        )
    except RuntimeError as err:
        if "Failed to dlopen libnvrtc" in str(err):
            err_msg = str(err)
            err_msg += " Ensure the version of cuda-python matches CUDA."
            raise RuntimeError(err_msg) from err
        raise
    opts = [] if opts is None else opts
    nvrtc_call(nvrtc.nvrtcCompileProgram(prog, len(opts), opts))

    # generate the actual kernel ptx
    ptx_size = nvrtc_call(nvrtc.nvrtcGetPTXSize(prog))
    ptx_buffer = b"\0" * ptx_size
    nvrtc_call(nvrtc.nvrtcGetPTX(prog, ptx_buffer))

return np.char.array(ptx_buffer)
```

```
stream = create_stream()

# block and thread info
num_threads: tuple[int, int, int] = (32, 32, 1)
num_blocks: tuple[int, int, int] = (
    math.ceil(o_width / num_threads[1]),  # X-axis (width)
    math.ceil(o_height / num_threads[0]),  # Y-axis (height)
    1,
)

# set is_input since we do not use the host_allocation here
input_binding = create_binding(
    img,
    is_input=True,
)
dummy_output: np.ndarray = np.zeros(
    (o_height, o_width, 3),
    dtype=np.uint8,
)
# set pagelocked memory since we read from the host allocation
output_binding = create_binding(
    dummy_output,
    pagelocked_mem=True,
)

# load the kernel
kernel = Kernel(
    *kernels.LINEAR_RESIZE,
)
```

KCF GPU

Profiling




```
                                  658         /* Convert BGR to ColorNames
                                  659          */
                                  660         void FastTrackerMP::extractCN(Mat patch_data, Mat & cnFeatures) const {
                                  661           ZoneScopedN("ftmp extractCn");
                                  662           Vec3b & pixel = patch_data.at<Vec3b>(0,0);
                                  663           unsigned index;
                                  664
                                  665           if(cnFeatures.type() != CV_32FC(10))
                                  666             cnFeatures = Mat::zeros(patch_data.rows,patch_data.cols,CV_32FC(10));
                                  667
                                  668           // perform the loops in parallel using OpenMP
                                  669           // and collapsed the loops to avoid overhead
                                  670
                                  671           // for loop variables
                                  672           const int batch_size = 2;
                                  673           const int patch_area = patch_data.rows*patch_data.cols;
 0.01%                     0.5%   674  @22      #pragma omp parallel for
                                  675  @30      for(int idx=0;idx<patch_area;idx+=batch_size){
 1.48%  1.68  100%  0.5%   676  @9         for(int offset=0; offset<batch_size; offset++){
 0.72%  1.03 14.7%  0.2%   677  @10          if(idx+offset >= patch_area) break;
 0.54%  0.27              678  @4           int h = idx+offset;
10.81%  0.80       10.0%   679  @7           int i = h/patch_data.cols;
 3.34%  0.12        4.8%   680  @7           int j = h%patch_data.cols;
 1.85%  0.62        3.2%   681  @12          Vec3b pixel = patch_data.at<Vec3b>(i,j);
36.18%  0.80        100%   682  @43          unsigned index=(unsigned)(floor((float)pixel[2]/8)+32*floor((float)pixel[1]/8)+32*32*floor((float)pixel[0]/8));
                                  683          // for(int k=0;k<10;k++)
                                  684          //   cnFeatures.at<Vec<float,10> >(i,j)[k] = ColorNames[index][k];
                                  685          //auto t = cnFeatures.at<Vec<float, 10> >(i, j);
 1.03%  0.00        0.6%   686  @3           Vec<float, 10> t;
 4.01%  1.00       16.7%   687  @15          t[0] = ColorNames[index][0];
 3.03%  1.58        8.1%   688  @15          t[1] = ColorNames[index][1];
 3.22%  1.87        7.7%   689  @15          t[2] = ColorNames[index][2];
 3.84%  1.57        7.7%   690  @15          t[3] = ColorNames[index][3];
 4.10%  1.38        7.9%   691  @15          t[4] = ColorNames[index][4];
 3.91%  1.36        5.1%   692  @15          t[5] = ColorNames[index][5];
 3.51%  1.64        3.4%   693  @15          t[6] = ColorNames[index][6];
 3.81%  1.83        2.3%   694  @15          t[7] = ColorNames[index][7];
 3.73%  1.63        1.6%   695  @15          t[8] = ColorNames[index][8];
 4.25%  1.47        2.0%   696  @15          t[9] = ColorNames[index][9];
 6.64%  0.98        3.6%   697  @17          cnFeatures.at<Vec<float,10> >(i,j) = t;
                                  698        }
```

```
                                  555         /* Convert BGR to ColorNames
                                  556          */
 0.02%  0.00                      557  @11     void FastTracker::extractCN(Mat patch_data, Mat & cnFeatures) const {
                                  558  @6        Vec3b & pixel = patch_data.at<Vec3b>(0,0);
                                  559            unsigned index;
                                  560
                                  561  @7        if(cnFeatures.type() != CV_32FC(10))
                                  562  @22         cnFeatures = Mat::zeros(patch_data.rows,patch_data.cols,CV_32FC(10));
                                  563
 0.04%  0.00                      564  @9        for(int i=0;i<patch_data.rows;i++){
 1.58%  1.58  7.6%  3.9%   565  @7           for(int j=0;j<patch_data.cols;j++){
 3.09%  1.07        7.2%   566  @11            pixel=patch_data.at<Vec3b>(i,j);
41.42%  0.44        100%   567  @43            index=(unsigned)(floor((float)pixel[2]/8)+32*floor((float)pixel[1]/8)+32*32*floor((float)pixel[0]/8));
                                  568
                                  569            //copy the values
15.81%  1.44  100% 41.2%   570  @5             for(int _k=0;_k<10;_k++){
38.05%  1.33       92.8%   571  @25              cnFeatures.at<Vec<float,10> >(i,j)[_k]=ColorNames[index][_k];
                                  572            }
                                  573           }
                                  574         }
                                  575
       0.00                       576  @9     }
                                  577
```

CSK GPU

SIMD vs. Multi-Thread vs. CUDA

- Implemented CSK in NumPy and PyTorch
  - NumPy uses SIMD instructions for FFT
  - PyTorch uses multi-threaded or CUDA
- Swap between versions depending on size of object to track