CoDL: Efficient CPU-GPU Co-execution for Deep Learning Inference on Mobile Devices

MobiSys'22

Fucheng Jia, Shiqi Jiang, Deyu Zhang, Yunxin Liu, Yaoxue Zhang, Ting Cao, Ju Ren

Central South University, Microsoft Research,

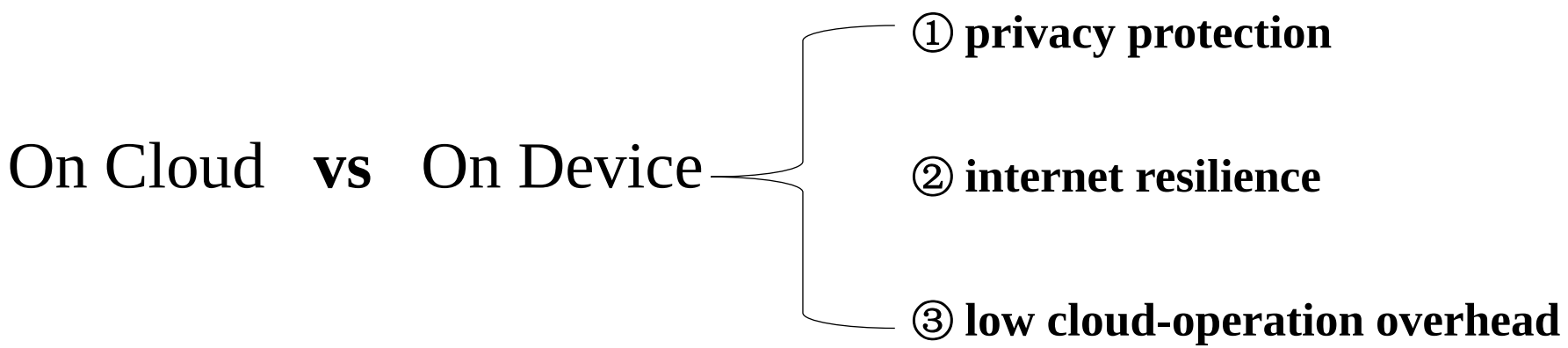
Institute for AI Industry Research (AIR), Tsinghua University

Introduction

Current on-device inference can only achieve acceptable responsiveness for some simple models

YOLO for object detection **takes over 200ms** to run on major mobile processors

To improve responsiveness, a nature thought is whether it is beneficial to **concurrently** utilize heterogeneous processors on a mobile device.

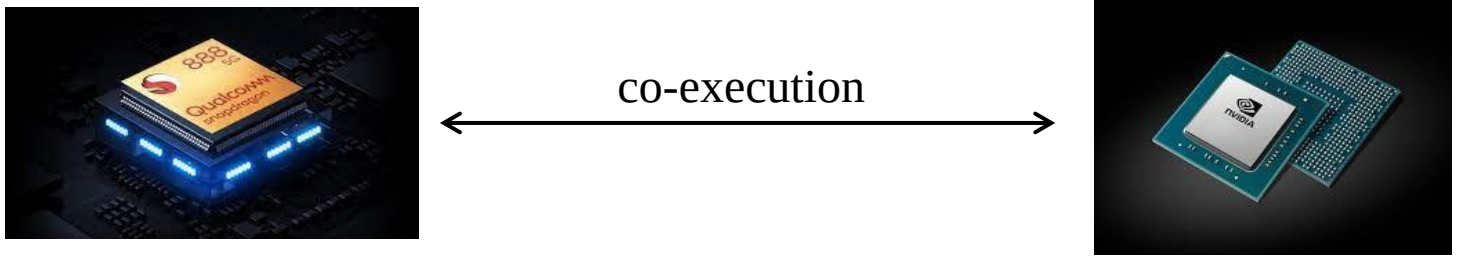


Introduction

The specific design of mobile systemon-chips (SoCs) provides this opportunity:

1）**Comparable CPU and GPU performance**. Mobile CPUs and GPUs have similar performance for DL inference

2）**A unified memory**. Avoid data copying between different memories



CPU GPU

Introduction

Challenges

1) How to reduce data sharing overhead ?

2) How to fairly partition each operator of a model between processors through latency predictor?

Introduction

Challenges

1) How to reduce data sharing overhead ?

2) How to fairly partition each operator of a model between processors through latency predictor?

μlayer & Optic enable the co-execution

Problems

1) The system is **even slower**

2) The latency predictor has **Poor accuracy(< 10%)**

3) The latency predictor can achieve high accuracy but is **too heavy**.

Introduction

Problems 1 Even slower

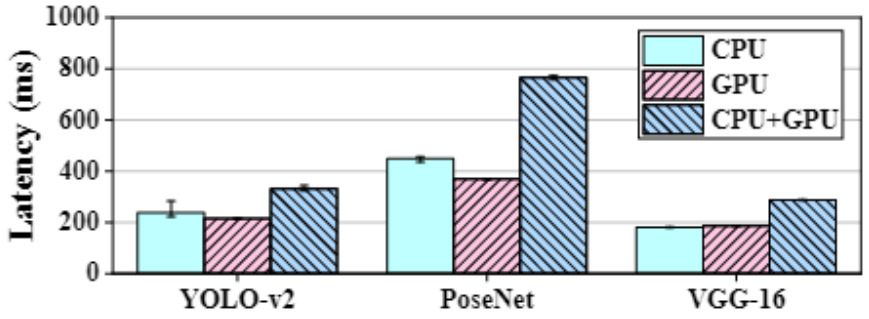


Figure 1: The latency of three DL models executed on CPU, GPU and CPU+GPU, in MACE.

Reasons:

1) the use of **unified data** type for different processors
2) the neglect of data sharing overhead
3) the unbalanced workload partitioning

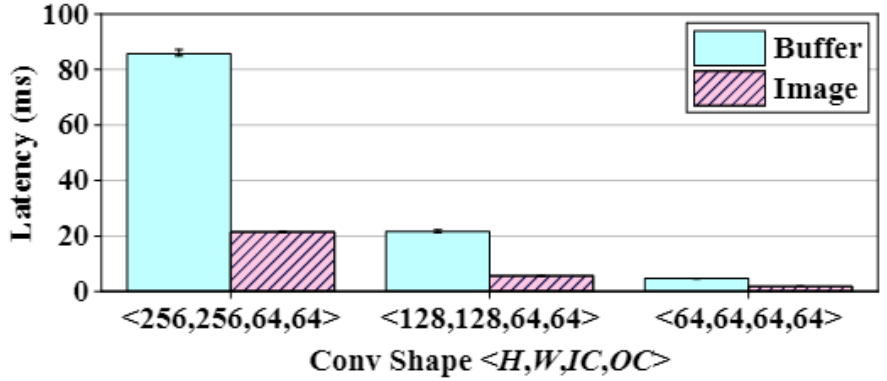


Figure 2: Latency comparison of using buffer and image data type for 3×3 convolution. The height (H) and width (W) of the input feature maps range from 64 to 256; the input channels (IC) and output channels (OC) are both 64.

Introduction

Problems 1 Even slower

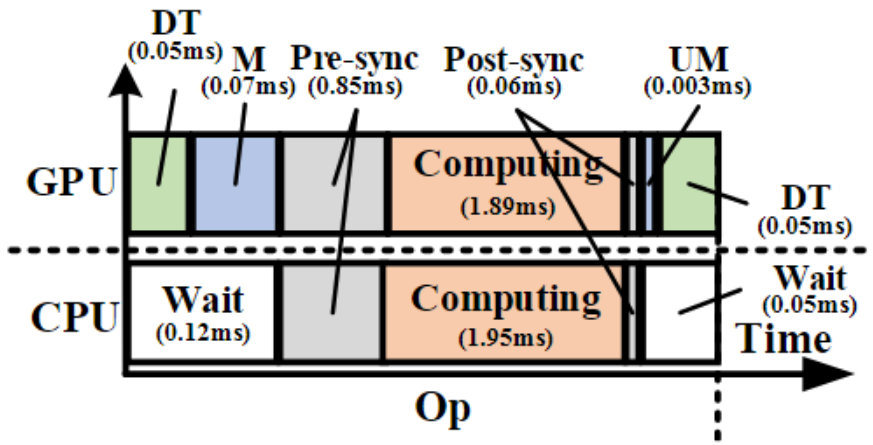


Figure 3: Latency components of co-execution of the CPU and GPU. The op is a 3×3 convolution with shape <112,112,32,64>. The partition ratio is 0.5. DT: Data Transformation, M: Mapping, UM: unmapping.

Reasons:

1) the use of unified data type for different processors
2) the neglect of **data sharing overhead**
3) the unbalanced workload partitioning

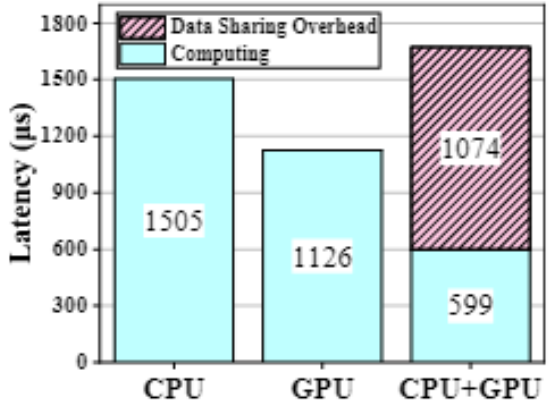


Figure 4: The latency of computing and data sharing for a 1×1 convolution with shape <52, 52, 256, 128>.

Introduction

Problems 1 Even slower

Reasons:

1) the use of unified data type for different processors
2) the neglect of data sharing overhead
3) the unbalanced **workload partitioning**

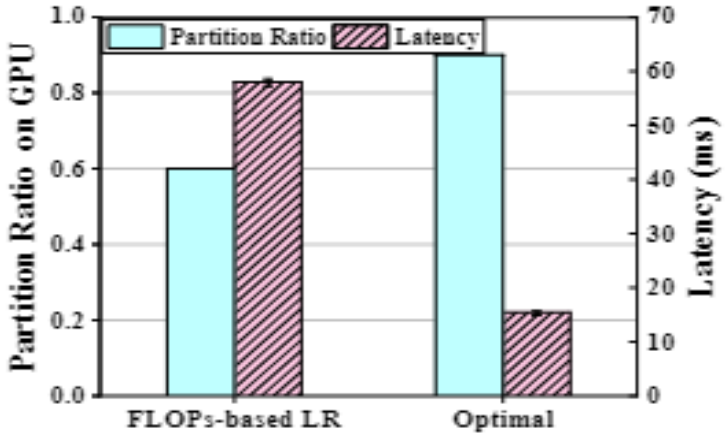


Figure 5: The partitioning ratio and latency of a 3×3 convolution with shape <240,320,64,128>.

Introduction

Problems 2 Poor accuracy

Reasons:

The latency is not simply a linear relationship with FLOPs, but greatly impacted by the platform features.

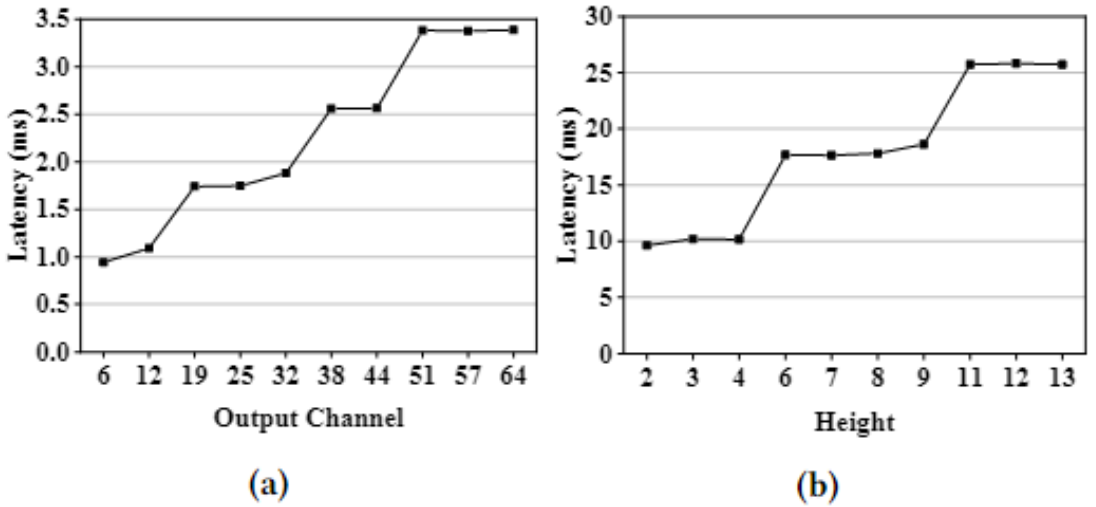


Figure 6: Non-linear latency response as FLOPs increases (by increasing channel and height) on the (a) GPU and (b) CPU.

Introduction

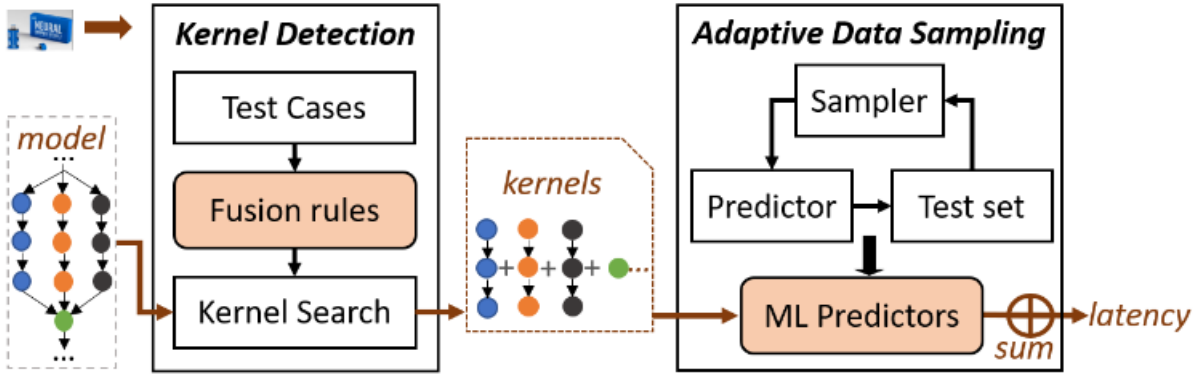
Problems 3 High accuacy but too heavy.

nn-Meter: Use black-box machine learning methods to learn the latency response based on a number of operator hyperparameters

Lacks:

1) the black-box methods obtain satisfactory accuracy at the cost of **large model size** (over 800 MB for convolution)

2) infeasible execution time, **unpractical to be deployed** on mobile devices (more than 80ms on PC)



nn-Meter

Introduction

Challenges

1) How to reduce data sharing overhead ?

2) How to fairly partition each operator of a model between processors through latency predictor?

μlayer & Optic enable the co-execution

Problems

1) The system is **even slower**

2) The latency predictor has **Poor accuracy(< 10%)**

3) The latency predictor can achieve high accuracy but is **too heavy**.

CoDL Overview

CoDL stemed from two key findings.

1) Different processors prefer different data type for optimal performance

2) To make the latency predictor both accurate and light-weight, it is imperative to incorporate platform features into the model

CoDL intergrates two techniques.

1) Hybrid-type-friendly data sharing

2) Non-linearity-and concurrency-aware latency prediction

CoDL Overview

Three design principles

1) Fully utilizing the computing capability of each processor
2) Minimizing the extra overhead caused by the **data sharing**
3) Best partitioning and balancing the workload among heterogeneous processors

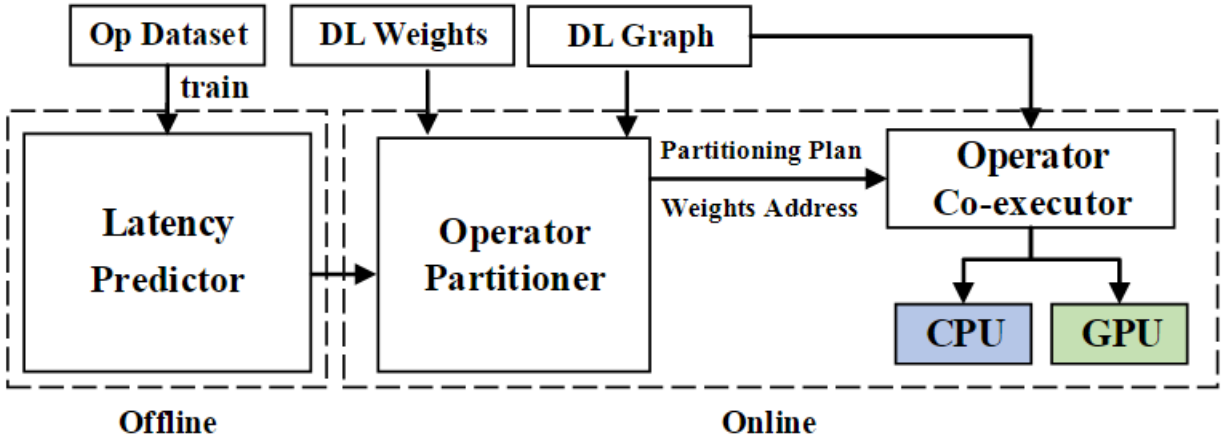


Figure 7: System architecture and workflow of CoDL.

CoDL Overview

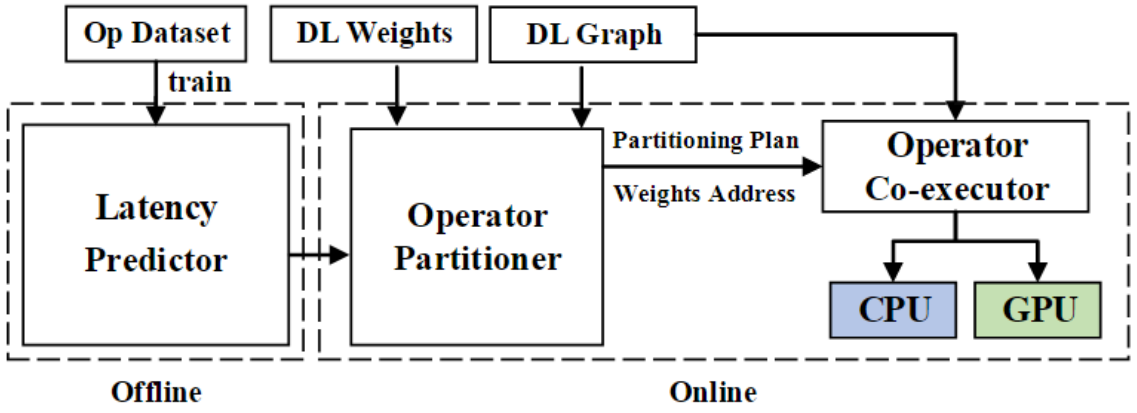


Figure 7: System architecture and workflow of CoDL.

Composition

1) A **light-weight** but effective latency predictor
   - **role:** achieve both **light-weight** and **effective**
   - consider all the data sharing
   - formulates the non-linear latency response
2) The operator partitioner
3) The operator co-exector

CoDL Overview

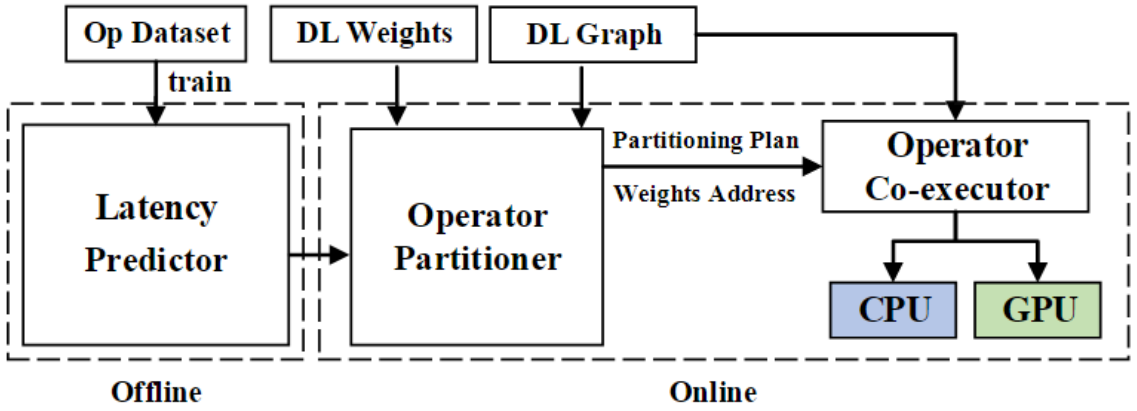


Figure 7: System architecture and workflow of CoDL.

Composition

1) A **light-weight** but effective latency predictor

2) The operator partitioner

- **role:** work out the optimal partitioning plan
- hybrid-dimension partitioning
- operator chain

3) The operator co-exector

CoDL Overview

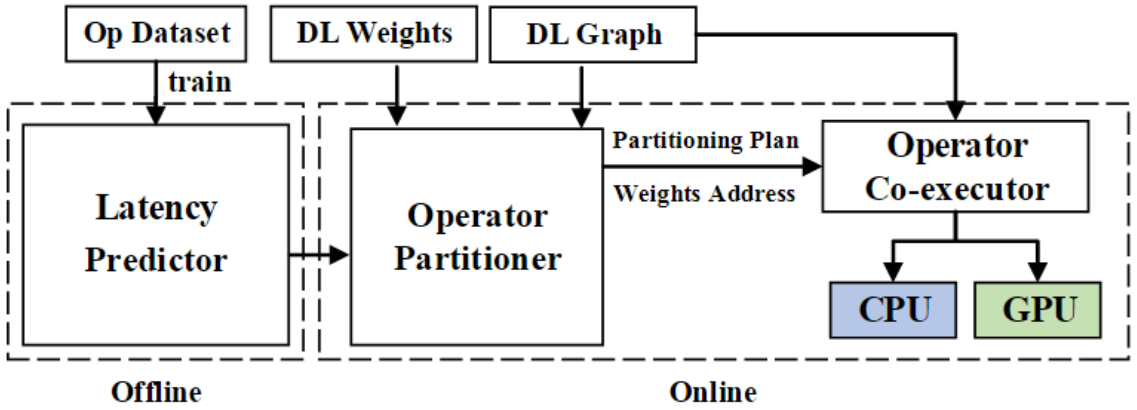


Figure 7: System architecture and workflow of CoDL.

Composition

1) A **light-weight** but effective latency predictor

2) The operator partitioner

3) The operator co-exector

- **role:** Coordinating the synchronized execution of operators according to the partitioning plan.

CoDL Overview

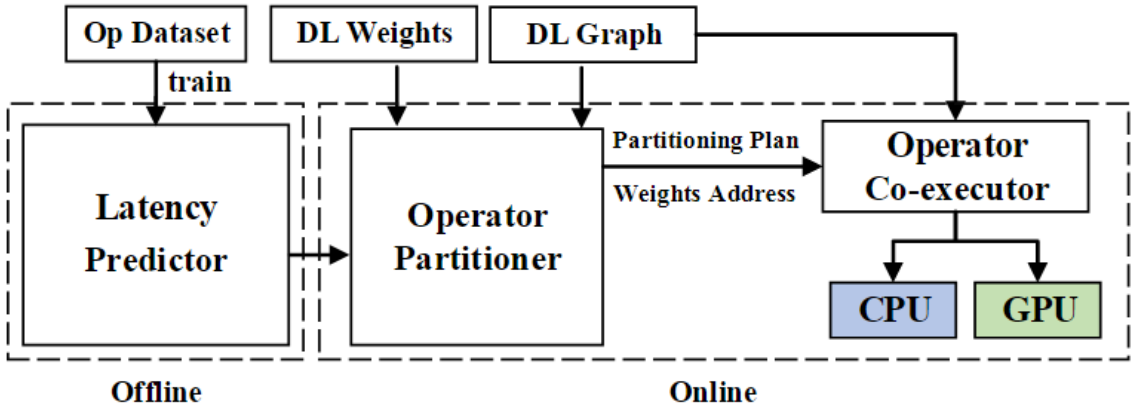


Figure 7: System architecture and workflow of CoDL.

Composition

1) A **light-weight** but effective latency predictor
2) The operator partitioner
3) The operator co-exeuctor

Hybrid-type Friendly Data Sharing

CoDL supports the use of **efficient data type** for each processor.

Challenge: It further **increase**s data sharing overhead.

Two data sharing optimization techniques of CoDL:

1) hybrid-dimension partitioning
2) operator chain

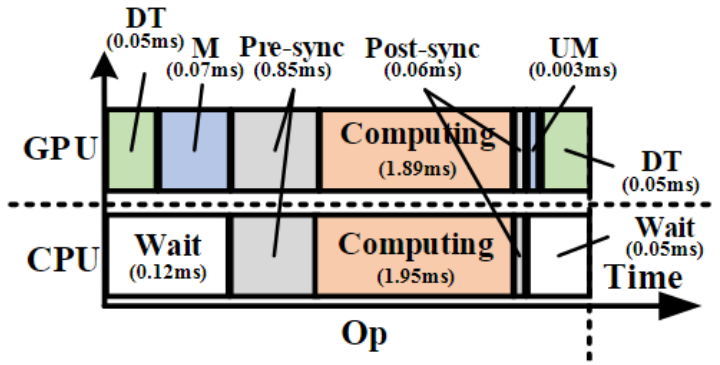


Figure 3: Latency components of co-execution of the CPU and GPU. The op is a 3×3 convolution with shape <112,112,32,64>. The partition ratio is 0.5. DT: Data Transformation, M: Mapping, UM: unmapping.

Hybrid-type Friendly Data Sharing

Hybrid-dimension partitioning

Different dimensions lead to different performance impact.

Firstly, partitioning dimension impacts data sharing overhead.

The input feature map for each operator needs to be shared dynamically.

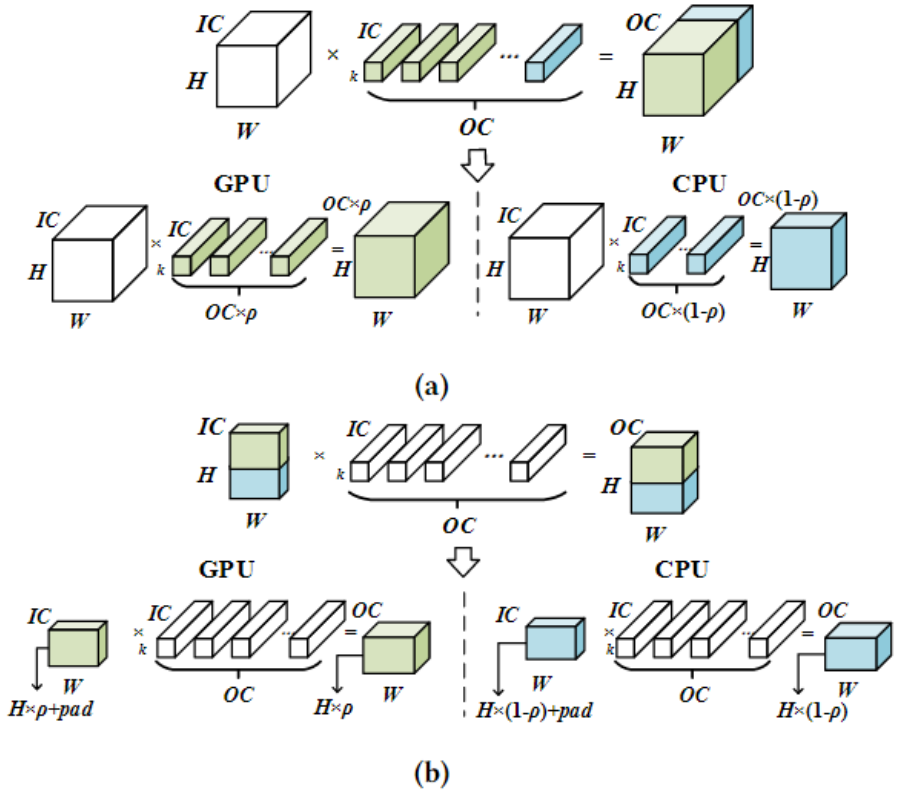


Figure 9: Data partitioning for co-execution along (a) OC and (b) H. ρ is the partitioning ratio for the GPU, and $1-\rho$ for the CPU.

Hybrid-type Friendly Data Sharing

Hybrid-dimension partitioning

Different dimensions lead to different performance impact.

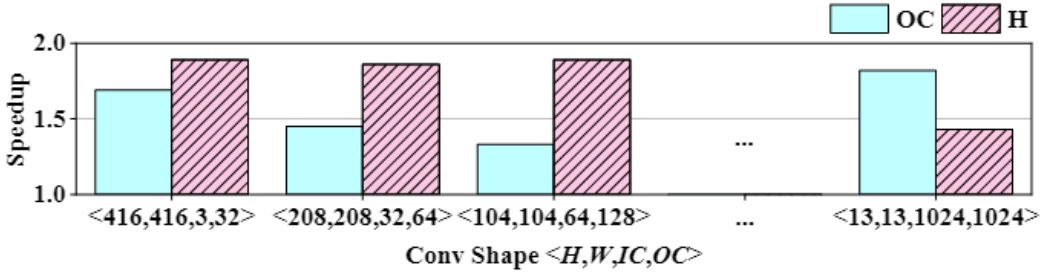


Figure 10: CPU+GPU speedup (over CPU_only) comparison between partitioning on OC and H, for convolution in YOLO model. The partitioning ratio is 0.5.

Secondly, partitioning dimension impacts processor utilization.

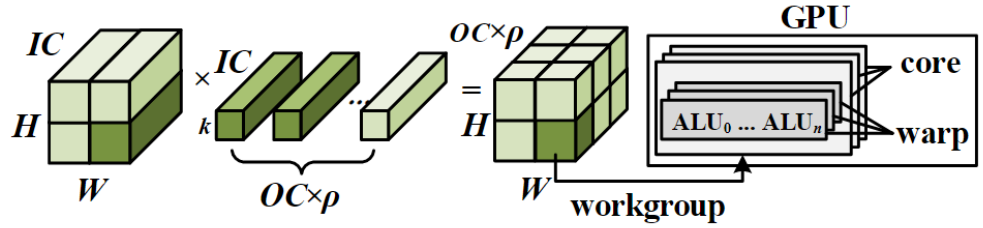


Figure 11: The output is divided into workgroups. Each work group is scheduled to run on a GPU core in the unit of warps.

Hybrid-type Friendly Data Sharing

Hybrid-dimension partitioning

Different dimensions lead to different performance impact.

Thirdly, partitioning dimension impacts data access overhead.

The partitioning dimension should be consistent with the tensor layout and make sure that the shared data is continuously stored in memory.

Hybrid-type Friendly Data Sharing

Hybrid-dimension partitioning

Based on the predicted latency, CoDL can rapidly evaluate different partitioning plans online, and find the optimal partitioning dimension and ratio for each operator of a DL model.

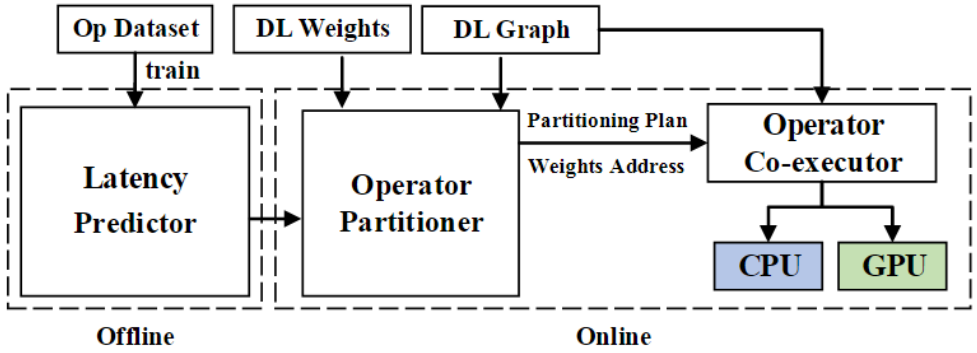


Figure 7: System architecture and workflow of CoDL.

Hybrid-type Friendly Data Sharing

Operator chain

Target: Reduce the number of operators requiring shared data.

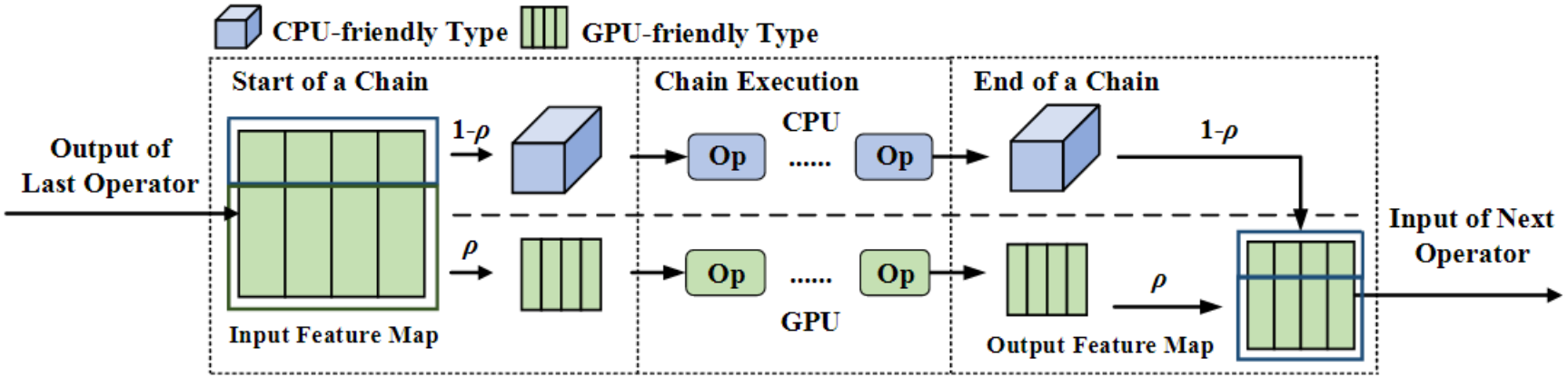


Figure 8: Co-execution of an operator chain.

Hybrid-type Friendly Data Sharing

Operator chain

Target: Reduce the number of operators requiring shared data.

Challenge: How to rapidly decide which operators to chain up in a DL model.

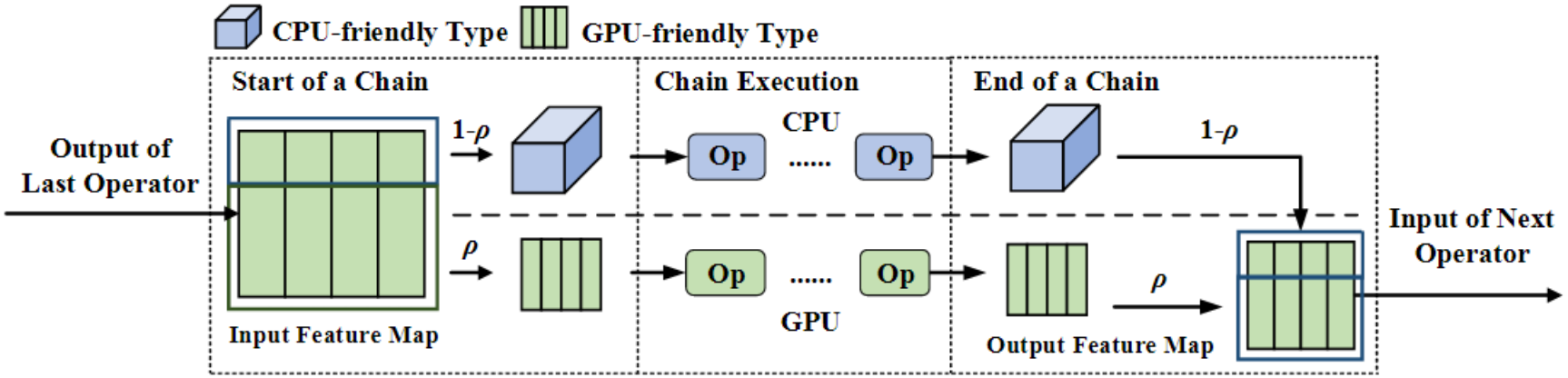


Figure 8: Co-execution of an operator chain.

Hybrid-type Friendly Data Sharing

Operator chain

Target: Reduce the number of operators requiring shared data.

Challenge: How to rapidly decide which operators to chain up in a DL model.

- The partitioning ratio of a chain may not be ideal for each of its operator's performance.
- The longer the chain is, the more padding and thus more additional computations there are.

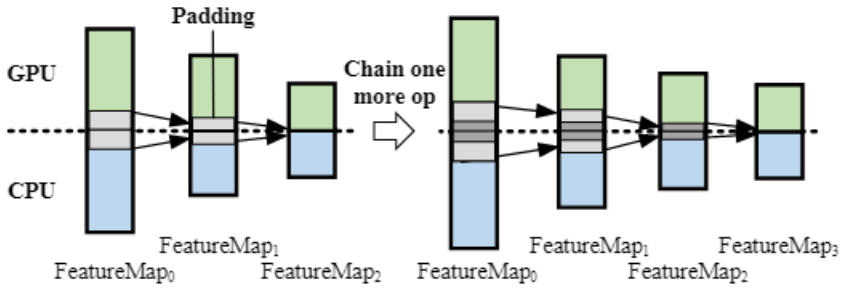


Figure 12: To chain one more operator requires more padding for each feature map in a chain.

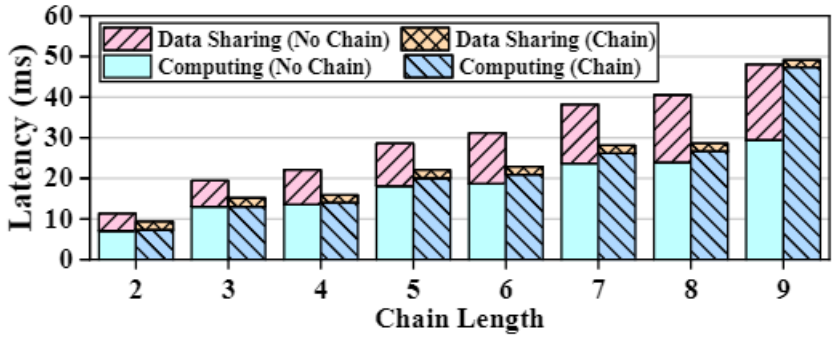


Figure 13: Latency comparison of chain vs no-chain on data sharing and computation, for operators in YOLO model. The partitioning ratio is 0.5.

Hybrid-type Friendly Data Sharing

Operator chain - Chain searching algorithm.

算法思路

① 从一个不在任何链上的算子开始，对每一个可能的分区比例 ρ，该算法不断链接更多的算子来计算与无链状况下的增益。

② 当再添加一个算子没有任何增益时停止

③ 此过程中， ρ 与 此链相关的最大链增益都会被记录下来，遍历完所有可能的 ρ 之后，最终记录到链的参数中。

④ 然后算法开始找下一个链

Algorithm 1: Chain searching algorithm

input : $ParPlan$ the partitioning plan of operators in the model without chain.
output: $Chains$ the operator chain settings.

```
1  next ←0;
2  (op_head, dim_head, ρ_head) ← ParPlan[next++];
3  while op_head is not NULL do
4      T_gain, T_maxGain ← 0, i←next;
5      foreach ρ ∈ [ρ_head − δ, ρ_head + δ] do
6          Chain_cur .initialize(op_head, dim_head, ρ);
7          T_chain ← Predictor(op_head, dim_head, ρ, false);
8          while T_gain ≥ 0 and i < N_op do
9              (op_next, dim_next, ρ_next) ← ParPlan[i++];
10             T_noChain ← T_chain +
                Predictor(op_next, dim_next, ρ_next, false);
11             Chain_cur.append(op_next, dim_head, ρ);
12             AdjustPadding(Chain_cur);
13             T_chain ←
                ∑_{op∈Chain_cur} Predictor(op, dim, ρ, true);
14             T_gain ← T_noChain − T_chain;
15             if T_gain > T_maxGain then
16                 Chain_maxGain, T_maxGain, next ←
                    Chain_cur, T_gain, i;
17     Chains.append(Chain_maxGain);
18     (op_head, dim_head, ρ_head) ← ParPlan[next++];
```

Search Space

$$2 \times N_{op} \times (\delta / 0.1)$$

CoDL Overview

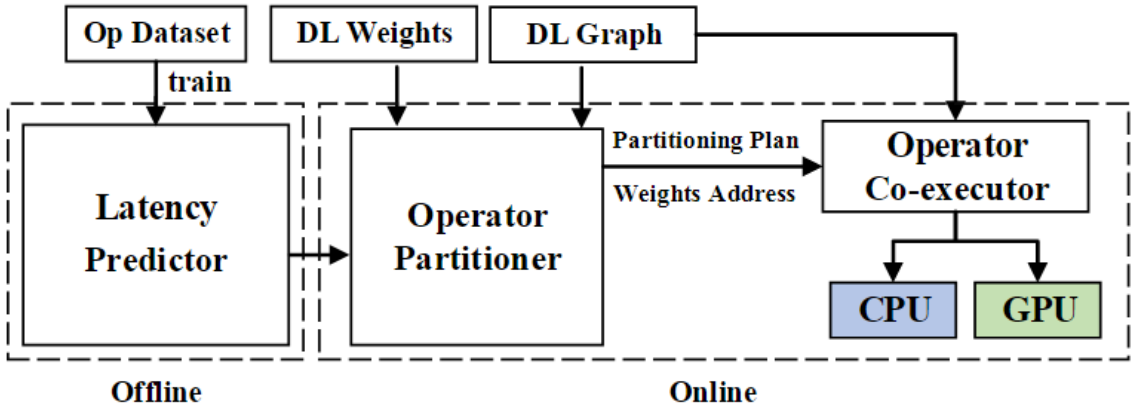


Figure 7: System architecture and workflow of CoDL.

Composition

1) A **light-weight** but effective latency predictor
2) The operator partitioner
3) The operator co-exector

Non-linerity and Concurency-aware Latency Prediction

The partitioning and operator chain techniques **rely on** latency prediction of operator co-execution.

Challenge: How to achieve both accurate and light-weight

Problems of the available latency predictors:

- did not consider the **data sharing overhead**
- cannot be **accurate and light-weight** at the same time

The design of CoDL

- including all the **data-sharing overhead** for co-execution
- analytically formulating the **non-linear latency** response caused by platform features, which lowers the difficulty of learning.

Non-linerity and Concurency-aware Latency Prediction

The design of CoDL

- including all the **data-sharing overhead** for co-execution
- analytically formulating the **non-linear latency** response caused by platform features, which lowers the difficulty of learning.

Input: operator hyperparameter, dim, ρ, a flag for chain or not

Output: the predicted latency of the operator co-execution on a given platform

Non-linerity and Concurency-aware Latency Prediction

Latency composition of concurrency

$$T = T_{trans} + T_{map} + T_{psync} + \max(T_{comp}^{cpu}, T_{comp}^{gpu})$$

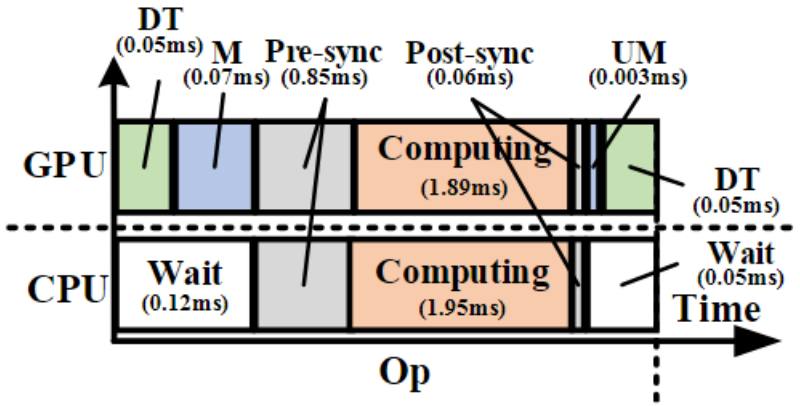


Figure 3: Latency components of co-execution of the CPU and GPU. The op is a 3×3 convolution with shape <112,112,32,64>. The partition ratio is 0.5. DT: Data Transformation, M: Mapping, UM: unmapping.

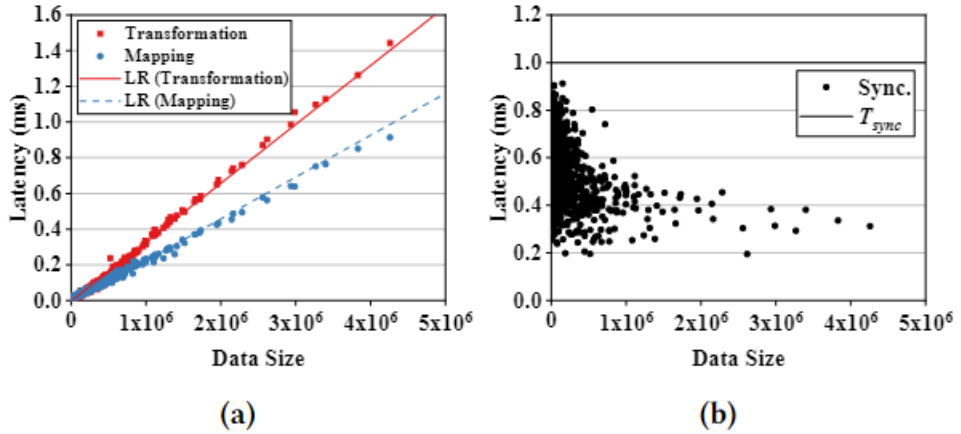


Figure 14: The latency distribution of various sizes of shared data for (a) data transformation and mapping; (b) pre-sync.

Non-linerity and Concurency-aware Latency Prediction

Non-linearity-extracted computing latency prediction

The non-linear latency response is mainly due to two reasons.

1) different algorithms have different latency response to **hyperparameter scaling**.
2) **data blocking** on different levels leads to stepped latency response

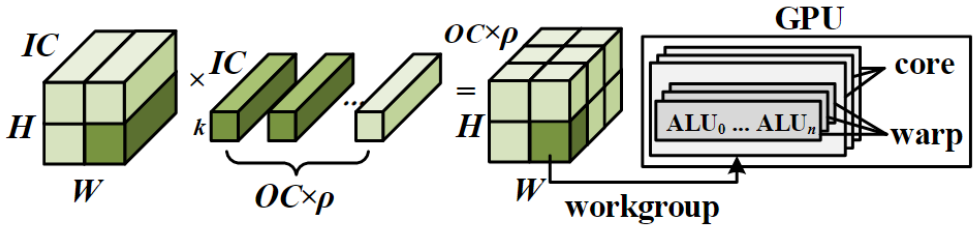


Figure 11: The output is divided into workgroups. Each work group is scheduled to run on a GPU core in the unit of warps.

Non-linerity and Concurency-aware Latency Prediction

Non-linearity-extracted computing latency prediction

Formulating non-linearity.

$$T_{kernel} = \lceil \frac{Size_{output}}{Size_{block} \cdot Core\#} \rceil \cdot \lceil \frac{Size_{block}}{Size_{basicUnit}} \rceil \cdot t_{basicUnit} \quad (3)$$

$$T = T_{trans} + T_{map} + T_{psync} + \max(T_{comp}^{cpu}, T_{comp}^{gpu})$$

An extreme light-weight linear regression model can be used to learn *t_basicUnit* and achieve high accuracy.

Table 1: Latency prediction of convolution algorithms for a partitioned operator on a given processor.

Params.	H, W: height and width, IC, OC: the number of input and output channels, F: filter size, Ω: the number of cores of the processor, Φ_c: the blocking size for a core c, ϕ_c: the size of a basic execution unit in c, Φ_{wt}: the size of a Winograd tile, Ψ: the number of Winograd tiles, t_c: time of a basic unit for direct convolution, t_p: time of a basic unit for data packing, t_{mm}: time of a basic unit for matrix multiplication, t_{it}, t_{ot}: time of a basic unit for transforming in-&out-feature map.
Direct	$T_{comp} = \lceil \frac{H \cdot W \cdot OC}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_c$
GEMM	$T_{comp} = T_{packing} + T_{MM}$ $T_{packing} = \lceil \frac{H \cdot W \cdot IC \cdot F^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_p$ $T_{MM} = \lceil \frac{H \cdot W \cdot IC \cdot OC \cdot F^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{mm}$
Winograd	$T_{comp} = T_{inputTrans} + (\Phi_{wt} + 2)^2 T_{GEMM} + T_{outputTrans}$ $\Psi = \lceil \frac{H}{\Phi_{wt}} \rceil \lceil \frac{W}{\Phi_{wt}} \rceil$ $T_{inputTrans} = \lceil \frac{IC \cdot \Psi \cdot (\Phi_{wt}+2)^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{it}$ $T_{outputTrans} = \lceil \frac{OC \cdot \Psi \cdot \Phi_{wt}^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{ot}$

Evaluation

Setup

Table 2: Hardware used in the evaluation.

Device	SoC	CPU	GPU
Xiaomi 9	Snapdragon 855	Kyro 485	Adreno 640
Redmi K30 Pro	Snapdragon 865	Kyro 585	Adreno 650
Xiaomi 11 Pro	Snapdragon 888	Kyro 680	Adreno 660
Honor V30 Pro	Kirin 990	Cortex-A76	Mali-G76

models: RetinaFace (RF), YOLOv2 (YOLO) , VGG-16 (VGG) , PoseNet (PN) and Fast Style Transfer (FST)

Evaluation

Overall performance

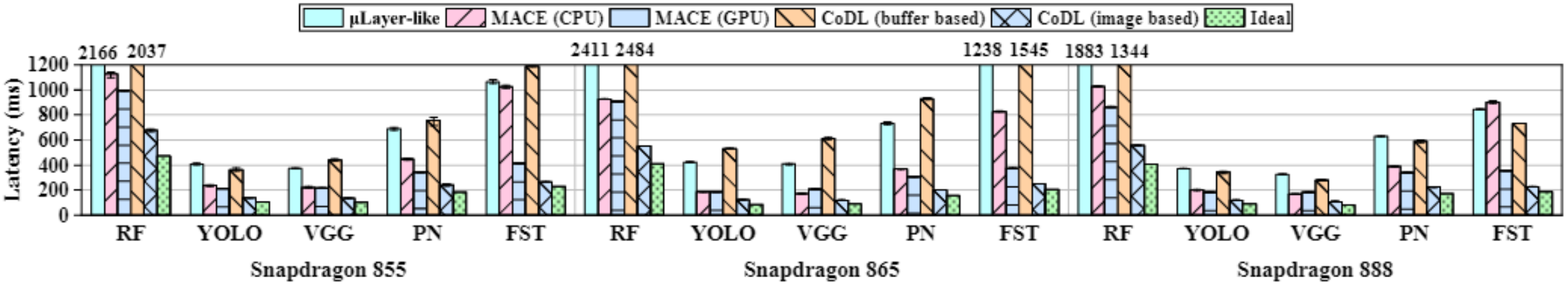


Figure 15: Overall performance of CoDL and baselines on Snapdragon platforms.

Evaluation

Overall performance

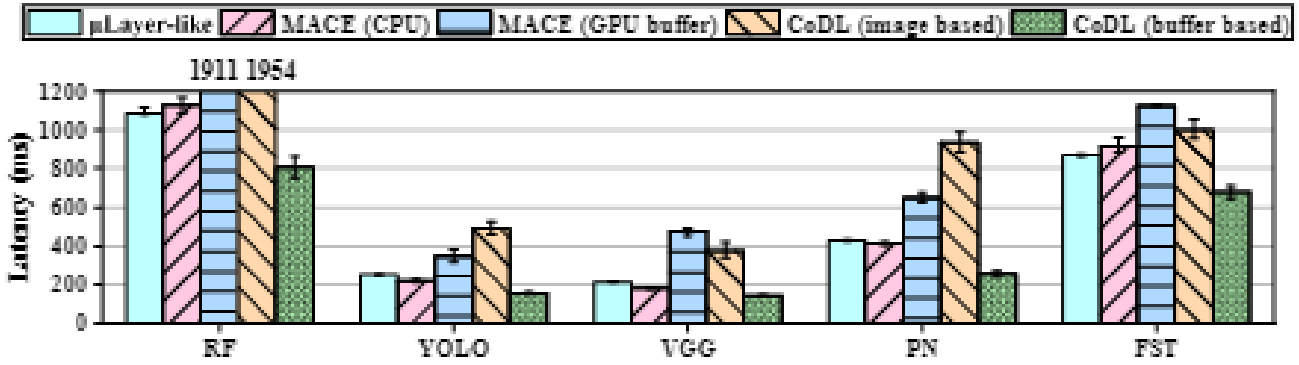


Figure 16: Performance of CoDL and baselines on Kirin 990.

Evaluation

Performance of the hybrid-dimension partitioning

Figure 17: Distribution of partitioning dimension on each operator of selected models.

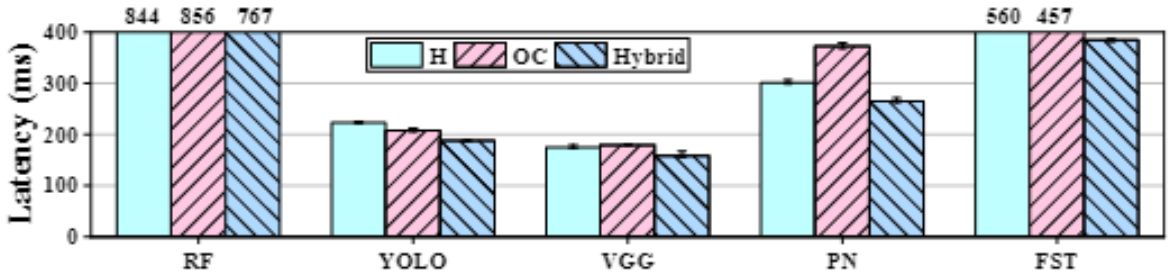


Figure 18: Inference latency with and without hybrid-partitioning dimension.

Evaluation

Performance of the operator chain

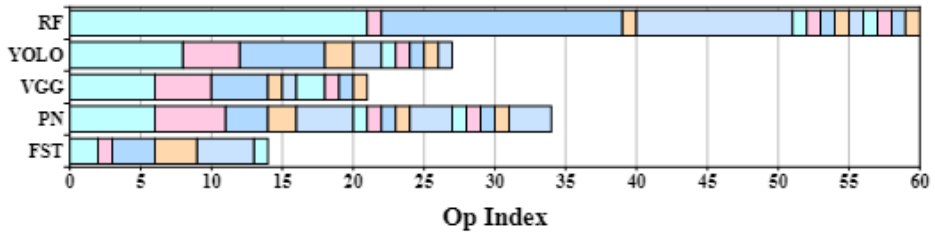


Figure 19: Generated operator chains on the selected models. The operators with same color are in the same chain.

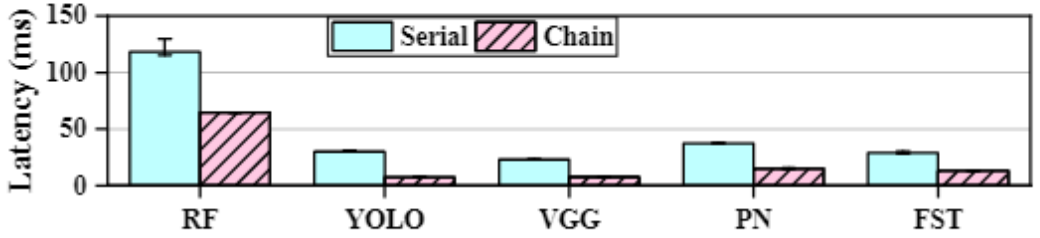


Figure 20: Data sharing overhead with and without the operator chain.

Evaluation

Performance of the latency predictor

Table 3: ±10% accuracy and model size of the latency predictors.

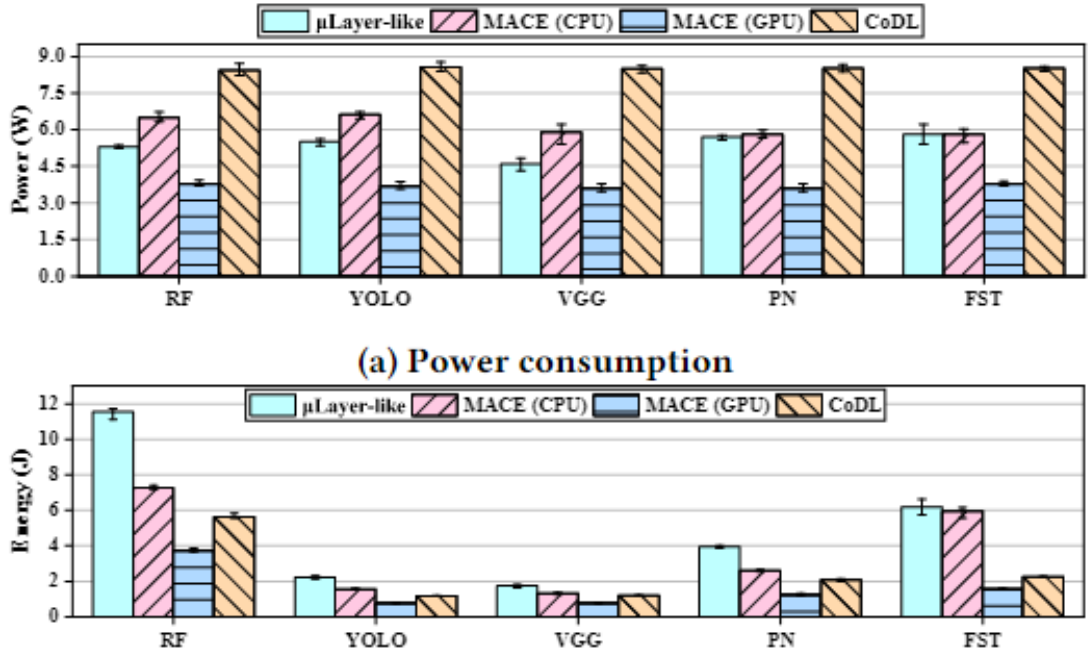
Predictor	Device	±10% Accuracy	Model Size
FLOPs-based	Xiaomi 9	10.46%	8B
	Redmi K30	6.99%	
	Xiaomi 11	9.40%	
nn-Meter	-	~90%	~800MB
Ours	Xiaomi 9	84.03%	500B
	Redmi K30	85.17%	
	Xiaomi 11	82.96%	

Table 4: ±10% prediction accuracy on typical convolution implementations.

Device	CPU-Direct	GPU-Direct	GEMM	Winograd
Xiaomi 9	77.77%	84.64%	91.14%	93.27%
Redmi K30	94.80%	78.39%	90.06%	93.72%
Xiaomi 11	84.56%	80.37%	85.71%	87.10%

Evaluation

System overhead



(a) Power consumption

(b) Energy consumption per inference

Figure 21: Power and energy consumption of CoDL and baselines on Snapdragon 855.

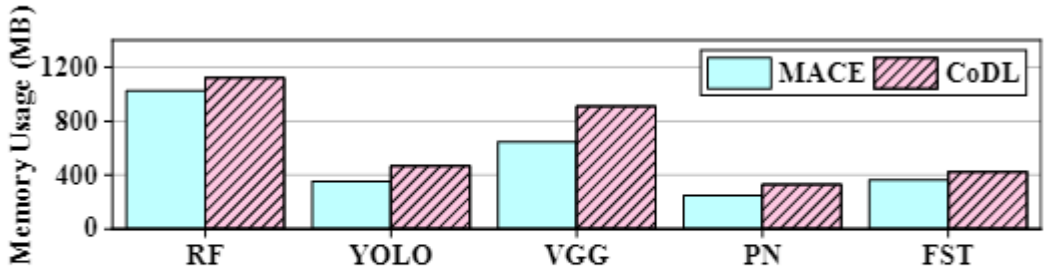


Figure 22: Memory usage of CoDL and MACE on Snapdragon 855.

Conclusion

Advantages:

1) balances the overhead of data sharing and computation by adaptively partitioning the operators from hybrid-dimensions

2) chaining operators up to reduce the data sharing times

3) fairly distributes the workload onto the heterogeneous processors by designing a light-weight but accurate latency predictor

Disadvantages:

1) consumes more power than single processor solution

2) hardly achieves speedup for lightweight DL models

3) only implement the co-execution of the inference process, it did not take the pre-processing and post-processing into consideration, which can not apply to the whole process of the computer vision task.

Thank You

2022-11-21

Presented by Ye Wan