nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices

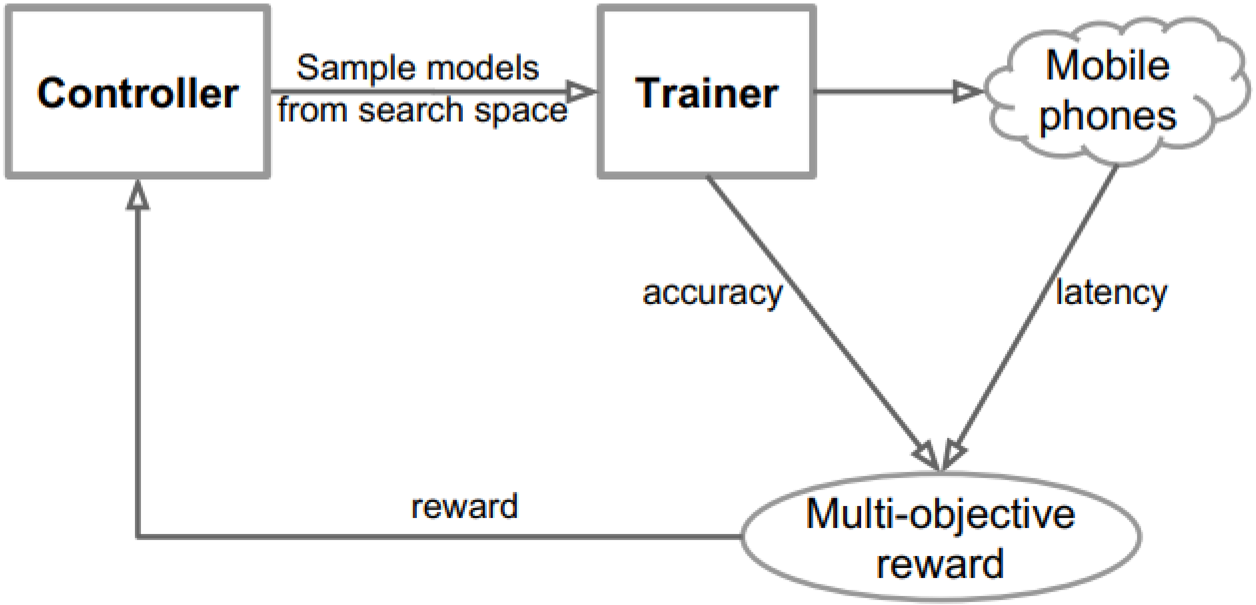
MobiSys' 22

Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang and Yunxin Liu

Microsoft Research, Rose-Hulman Institute of Technology, University of Science and Technology of China and Tsinghua University

Introduction

To design a model with both high accuracy and efficiency, **model compression** and **neural architecture search** consider the inference latency of DNN models as the hard design constraint.



MnasNet-CVPR’19

Introduction

The engineering effort is tremendous for diverse edge devices and different inference frameworks.

Edge Devices

CPU

GPU

VPU

Inference frameworks (backend)

TensorFlow Lite

TensorRT

OpenVINO

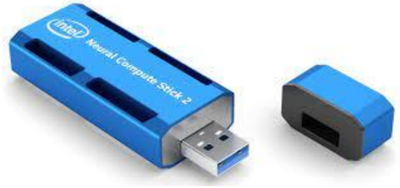
Introduction

The engineering effort is tremendous for diverse edge devices and different inference frameworks.

Edge Devices

CPU

GPU



VPU

Inference frameworks (backend)

TensorFlow Lite

TensorRT

OpenVINO™

… …

TNN

… …

Introduction

Some approaches just used theoretical metrics.

FLOPs

FLOPs+MAC

Some approaches build operator-wise lookup table.

```
Conv-input:224x224x3-output:112x112x32:
  count: 982
  mean: 4.083755600814664
  var: 0.2502372872758427
Conv_1-input:7x7x320-output:7x7x1280:
  count: 982
  mean: 3.088623217922607
  var: 0.19174675835259503
Logits-input:7x7x1280-output:1000:
  count: 982
  mean: 0.3096415478615071
  var: 0.05583766681628091
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:3-stride:2-idskip:0:
  count: 164
  mean: 6.240567073170731
  var: 0.07330791484716055
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:5-stride:2-idskip:0:
  count: 159
  mean: 7.519106918238994
  var: 0.09288016159302682
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:7-stride:2-idskip:0:
  count: 161
  mean: 9.143757763975156
  var: 0.10378775382215095
expanded_conv-input:112x112x16-output:56x56x24-expand:6-kernel:3-stride:2-idskip:0:
  count: 173
  mean: 16.766225433526014
  var: 0.5075419297470916
expanded_conv-input:112x112x16-output:56x56x24-expand:6-kernel:5-stride:2-idskip:0:
  count: 161
  mean: 18.941260869565216
  var: 0.3943272992116978
```

Introduction

Some approaches just used theoretical metrics.

FLOPs

FLOPs+MAC

Some approaches build operator-wise lookup table.

```
Conv-input:224x224x3-output:112x112x32:
  count: 982
  mean: 4.083755600814664
  var: 0.2502372872758427
Conv_1-input:7x7x320-output:7x7x1280:
  count: 982
  mean: 3.088623217922607
  var: 0.19174675835259503
Logits-input:7x7x1280-output:1000:
  count: 982
  mean: 0.3096415478615071
  var: 0.05583766681628091
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:3-stride:2-idskip:0:
  count: 164
  mean: 6.240567073170731
  var: 0.07330791484716055
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:5-stride:2-idskip:0:
  count: 159
  mean: 7.519106918238994
  var: 0.09288016159302682
expanded_conv-input:112x112x16-output:56x56x24-expand:3-kernel:7-stride:2-idskip:0:
  count: 161
  mean: 9.143757763975156
  var: 0.10378775382215095
expanded_conv-input:112x112x16-output:56x56x24-expand:6-kernel:3-stride:2-idskip:0:
  count: 173
  mean: 16.766225433526014
  var: 0.5075419297470916
expanded_conv-input:112x112x16-output:56x56x24-expand:6-kernel:5-stride:2-idskip:0:
  count: 161
  mean: 18.941260869565216
  var: 0.3943272992116978
```

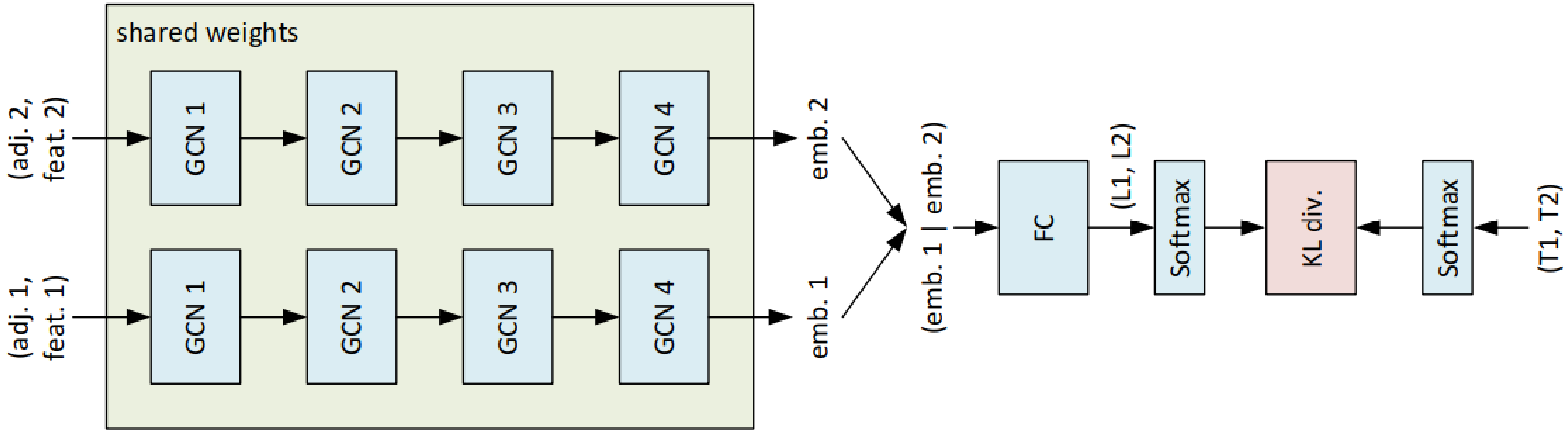
Problem:

They do not consider the model latency differences caused by **runtime optimizations** of model graphs.

Introduction

The state-of-the-art BRP-NAS uses graph convolutional networks to predict latency of the NASBench201 dataset on various devices. **It captures the runtime optimizations by learning the representation of model graphs and corresponding latency.**

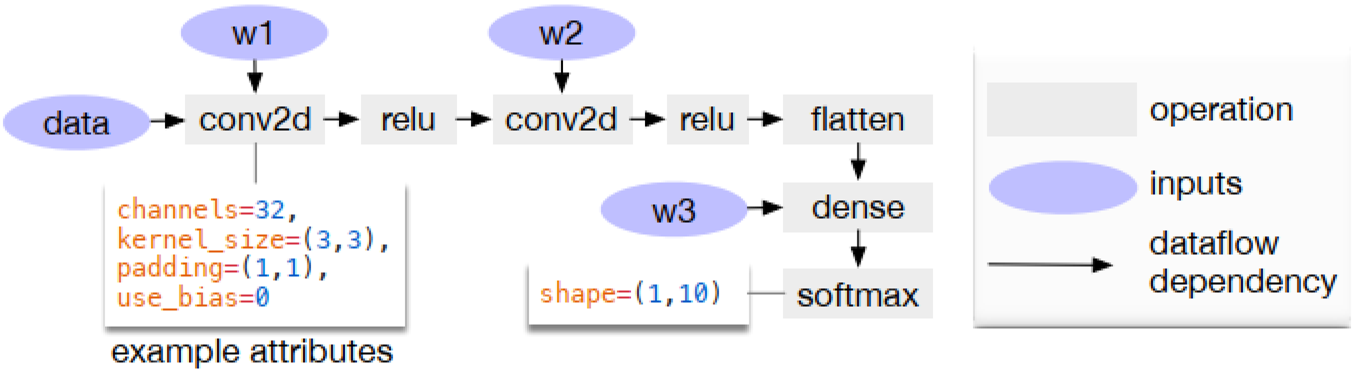
But it heavily depends on the tested model structures and may not work for many unseen model structures.



BRP-NAS-NIPS’20

Introduction

Neural network graph



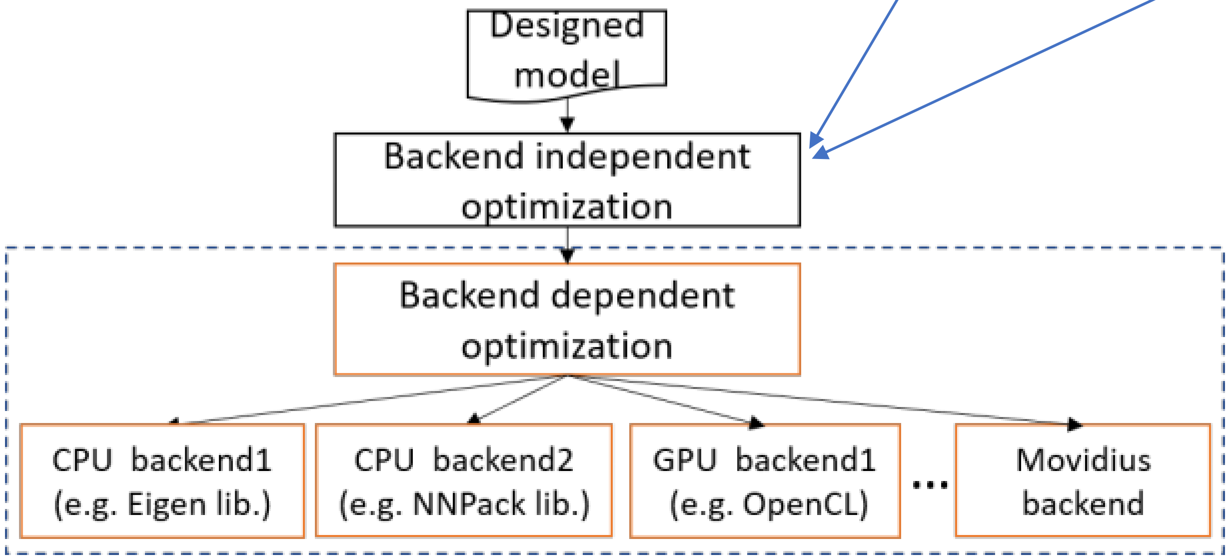
Introduction

Graph optimization

constant folding

$add(c1, add(x, c2))$ -> $add(x, c1 + c2)$

common expression elimination



operator fusion

Figure 1: Graph optimizations of framework.

Introduction

Operator fusion

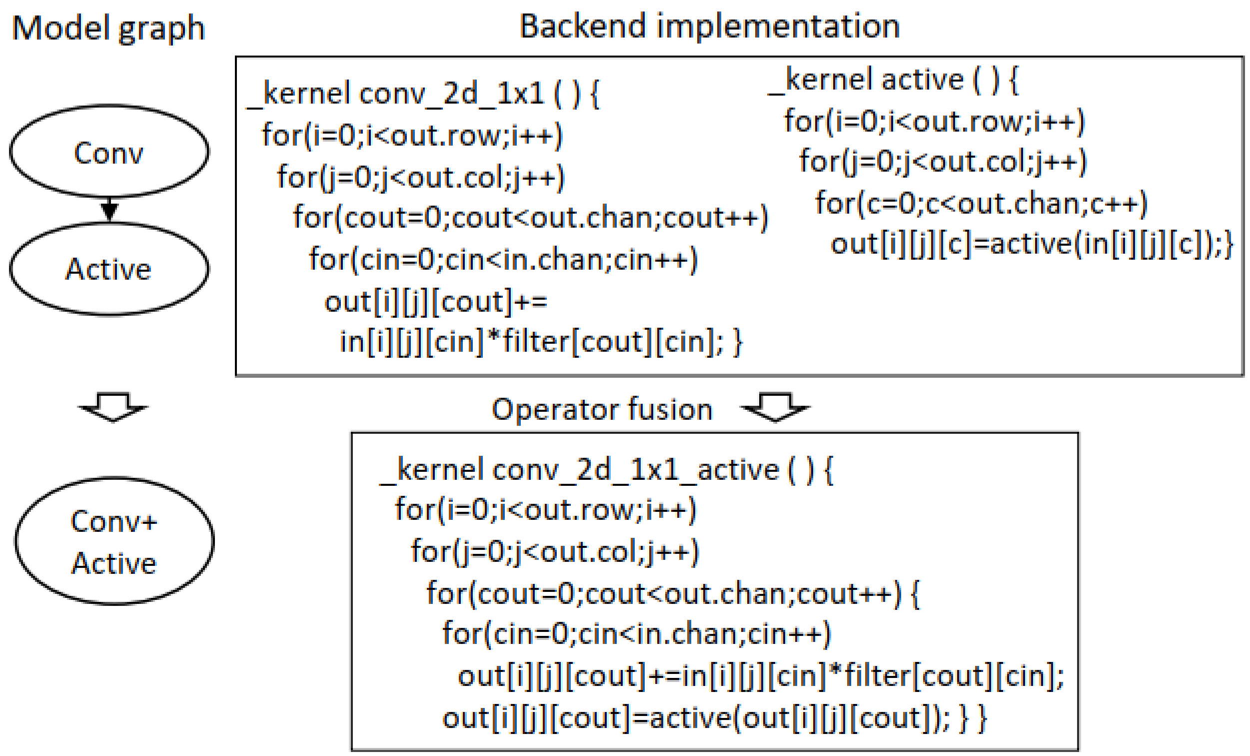
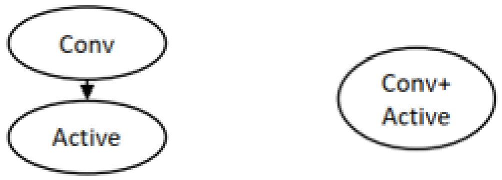


Figure 2: Kernel implementation for operator fusion("+" is used to represent fusion in this paper).

Introduction

Kernel-level prediction



	Model	Operator sum		Kernel sum	
	Latency	Latency	Error	Latency	Error
CPU	45.57ms	51.23ms	12.42%	45.41ms	0.35%
GPU	10.18ms	12.31ms	20.92%	9.91ms	2.65%
VPU	22.64ms	33.86ms	49.56%	23.18ms	2.38%

Table 1: MobileNetv2 latency.

System Overview

nn-Meter

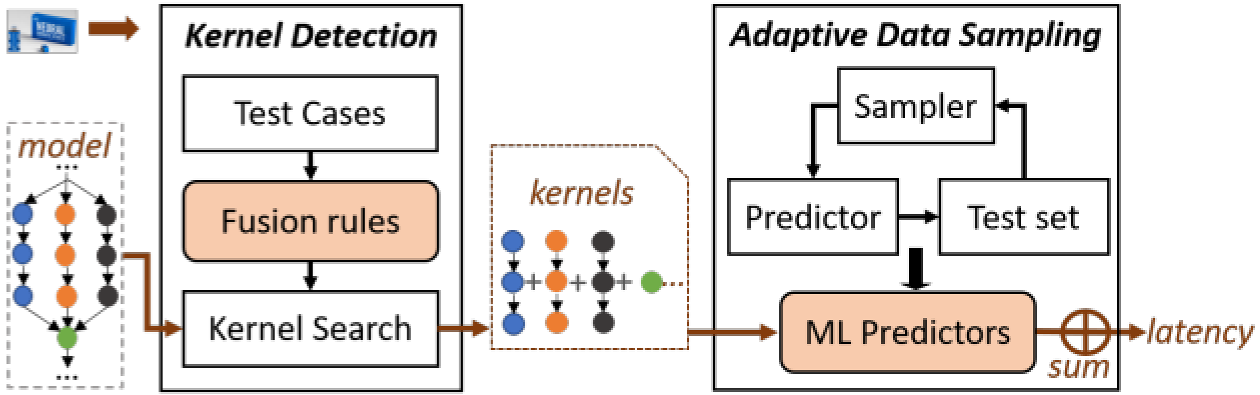


Figure 3: System architecture of nn-Meter. It offline detects fusion rules and builds ML predictors of kernels.

System Overview

Benchmark dataset collection

Model variants	avg FLOPs (M)	Latency(ms) Mobile CPU min - max	Mobile GPU min - max	Intel VPU min - max
AlexNets	973	7.1 - 494.4	0.4 - 81.7	2.1 - 47.3
VGGs	28422	178.4 - 10289	20.1 - 1278	25.6 - 1467
DenseNets	1794	109.6 - 431.6	26.7 - 69.5	26.4 - 70.7
ResNets	4151	35.9 - 1921.7	7.3 - 329.5	10.7 - 145.5
SqueezNets	1597	42.7 - 524.9	7.5 - 72.2	6.9 - 57.3
GoogleNets	1475	115.5 - 274.6	23.0 - 49.0	12.2 - 24.4
MobileNetv1s	547	27.5 - 140.0	5.5 - 28.8	8.9 - 37.0
MobileNetv2s	392	15.6 - 211.0	3.5 - 37.0	11.3 - 86.1
MobileNetv3s	176	10.4 - 78.4	4.3 - 18.6	17.4 - 70.8
ShuffleNetv2s	307	22.2 - 84.3	-	20.9 - 44.2
MnasNets	327	25.6 - 99.3	5.8 - 24.1	19.8 - 60.9
ProxylessNass	532	34.5 - 195.9	7.9 - 72.2	18.0 - 77.8
NASBench201	97.5	5.6 - 27.9	1.8 - 8.3	2.3 - 6.4

Table 2: The FLOPs and latency of each model variants in our proposed dataset. It covers a wide spectrum.

Key1-Kernel Detection

Test case design

operator type and **operator connection**

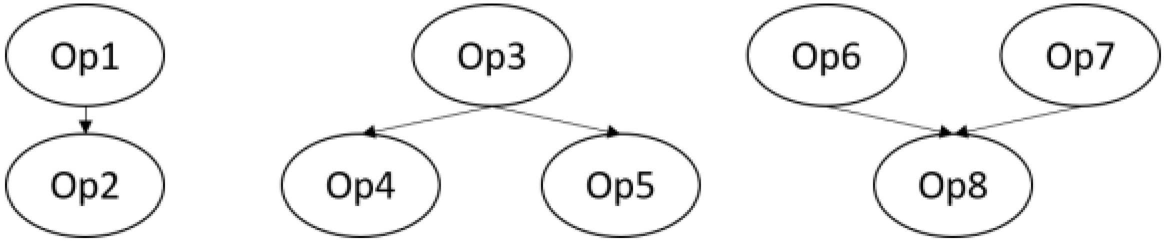
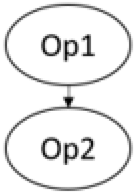


Figure 4: Operator connections: (a) single inbound and outbound; (b) multiple outbounds; (c) multiple inbounds.

Key1-Kernel Detection

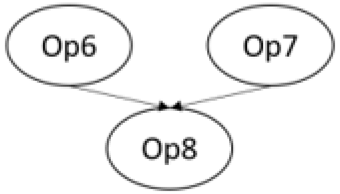
Test case design

How to find fusion rules in black-box backend?



$$T_{Op1} + T_{Op2} - T_{(Op1,Op2)} > \alpha * \min(T_{Op1}, T_{Op2})$$

Pick the closest time cost as the real time among:



$$T_{Op6} + T_{Op7} + T_{Op8}, T_{Op6+Op8} + T_{Op7}, T_{Op6} + T_{Op7+Op8}$$

Key1-Kernel Detection

Example-pool_relu

Backend	T_{pool} (μs)	T_{relu} (μs)	$T_{(pool,relu)}$ $(T_{pool} + T_{relu})$	Rule
VPU	13	26	16 (39)	“pool_relu":True
GPU	5.08	3.50	6.00 (8.60)	“pool_relu":True
CPU	23.60	0.81	24.48 (24.42)	“pool_relu":False

Table 3: A fusion detection example (pool, relu).

Key1-Kernel Detection

Algorithm 1 Kernel searching

Input: G a CNN model graph; R a set of fusion rules for a backend;
Output: Updated G with fused operators

function FUSE(O_{pred}, O_{succ})
  $O \leftarrow$ add a new operator in G as $O_{pred} + O_{succ}$
  $O.in \leftarrow O_{pred}.in \cup O_{succ}.in - O_{pred}$
  $O.out \leftarrow O_{pred}.out \cup O_{succ}.out - O_{succ}$
  $O.type \leftarrow O_{pred}.type$
  remove O_{pred}, O_{succ} from G
  return O
end function
function DFSTRAVERSE(O_{pred})
  for $O_{succ} \in O_{pred}.out$ **do**
    if $Rule_{fuse}(O_{pred}.type, O_{succ}.type)$
      and (len($O_{pred}.out$)==1 or $Rule_{multiout}()$)
      and (len($O_{succ}.in$)==1 or $Rule_{multiin}()$) **then**
        $O_{next} \leftarrow$ FUSE(O_{pred}, O_{succ})
    else
        $O_{next} \leftarrow O_{succ}$
    end if
    DFSTRAVERSE(O_{next})
  end for
end function
▷ Initial traverse input is the root of G
DFSTRAVERSE(O_{root})

Key1-Kernel Detection

Example-ResNet18

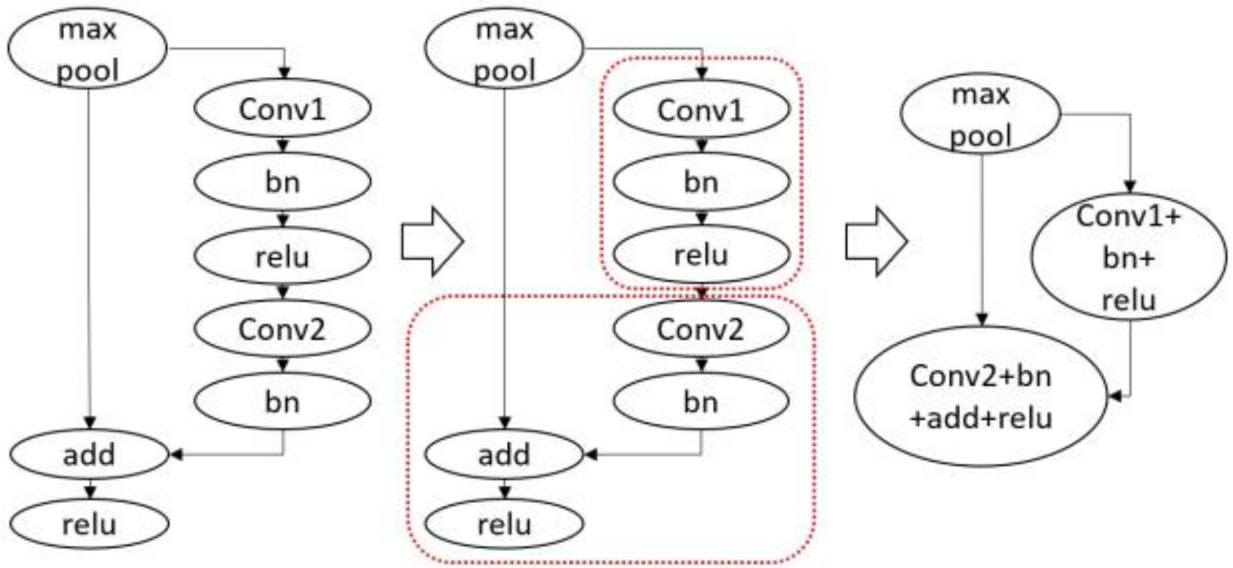


Figure 5: A kernel search example on a subgraph of ResNet18 model. The found kernels are {maxpool, Conv+bn+relu, Conv+bn+add+relu}.

Key1-Kernel Detection

Example-ResNet18

VPU		GPU		CPU	
kernel	#	kernel	#	kernel	#
Conv+bn+relu	9	Conv+bn+relu	9	Conv+bn+relu	9
maxpool	1	maxpool	1	maxpool	1
Conv+bn	11	Conv+bn+add+relu	8	Conv+bn	11
add+relu	8	Conv+bn	3	add+relu	8
avgpool	1	avgpool	1	avgpool	1
FC	1	FC	1	FC	1

Table 4: Found kernels for ResNet18.

Key2-Latency Prediction

Kernel characterization

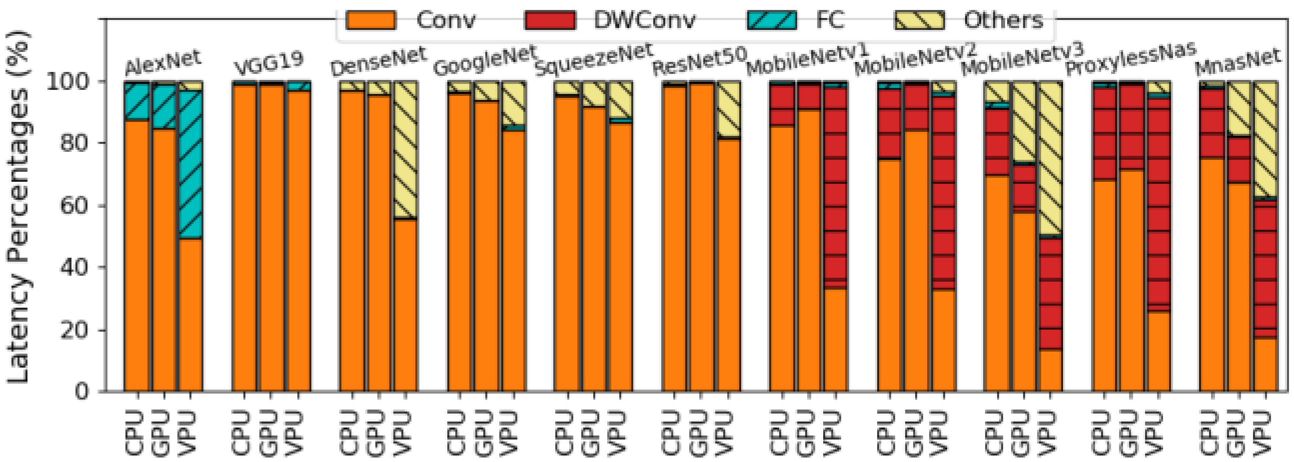


Figure 6: Model latency percentage breakdown. Conv and DWConv are the latency-dominating kernels.

Conv and DWConv dominate the latency

Key2-Latency Prediction

Kernel characterization

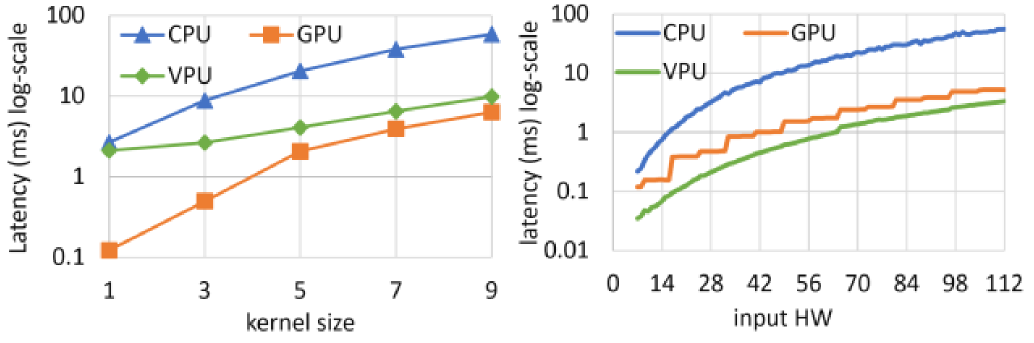
Dimension	Sample space
input HW	224, 112, 56, 32, 28, 27, 14, 13, 8, 7, 1
kernel size K	1, 3, 5, 7, 9
stride S	1, 2, 4
C_{cin}	range(3, 2160)
C_{out}	range(16, 2048)

Table 5: Sample space of Conv+bn+relu. It contains $\approx$ 0.7 billion configurations.

Large sample space of Conv

Key2-Latency Prediction

Kernel characterization



(a) Kernel Size (b) Input HW

Figure 7: Conv+bn+relu with (a): different kernel sizes (HW=224, C_{in}=3, C_{out}=32, S=1); (b): different input heights/widths. (C_{in}=C_{out}=64, K=3, S=1)

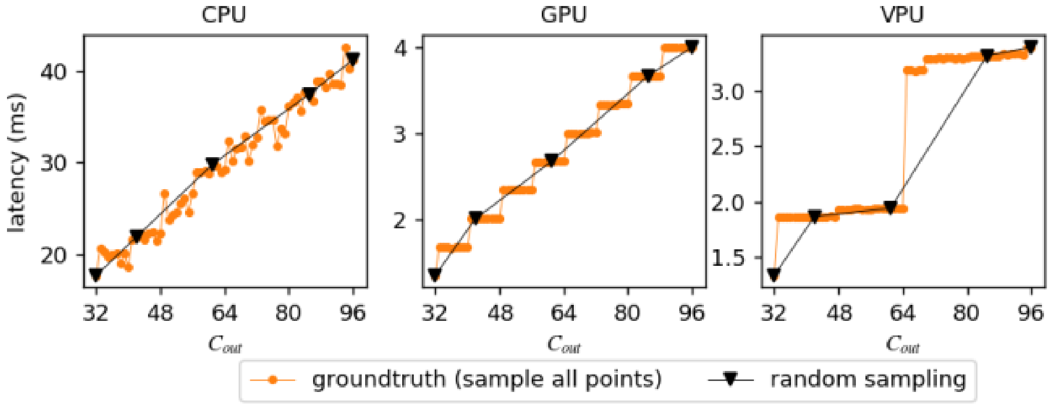


Figure 8: Latency of Conv+bn+relu with different output channel numbers. The groundtruth with sampling all channel numbers shows a staircase pattern on VPU and GPU. (HW=112, C_{in}=32, K=3, S=1)

Non-linear latency pattern

Random sampling misses hardware-crucial data

Key2-Latency Prediction

Algorithm 2 Adaptive Data Sampling Algorithm

Input: P prior possibility distribution from existing model zoo;
N initial data to sample from P; TD initial Test set;
M, number of data to sample for fine-grained sampling
e, the error threshold for regression model performance;
Output: all the sampled data (X,Y)

function FINEGRAINEDSAMPLE(X, M)
    for $x \in X$ **do**
        $D \leftarrow$ sample M data, we fix the (HW, K, S), channel numbers are randomly sampled from range $(0.4C_o, 1.2C_o)$
        $X_{new} \leftarrow X_{new}+D$
    end for
    $Y_{new} \leftarrow$ MEASURELATENCYONDEVICE(X_{new})
    return (X_{new}, Y_{new})
end function

▷ initialize N data from prior distribution to measure
$(X,Y) \leftarrow$ sample N data from distribution P
$f \leftarrow$ Construct regression model with (X_{train}, Y_{train})
$TD \leftarrow TD + (X_{test}, Y_{test})$
$e(f) \leftarrow$ test f on TD
▷ perform fine-grained sampling for inaccurate data
while $e(f) > e$ **do**
    $X^* \leftarrow$ select data with large predict error from TD
    $(X_i, Y_i) \leftarrow$ FINEGRAINEDSAMPLE(X^*, M)
    $(X, Y) \leftarrow (X, Y) + (X_i, Y_i)$
    update regression model f with (X_{train}, Y_{train})
    $TD \leftarrow TD + (X_{test}, Y_{test})$
    $e(f) \leftarrow$ test f on TD
end while

Key2-Latency Prediction

Collected data

Kernel	Features	# Collected Data CPU	GPU	VPU
Conv+bn+relu	HW, C_{in}, C_{out}, K S, FLOPs, params	15824	14040	39968
DWConv+bn+relu	$HW, C_{in}, K, S,$ FLOPs, params	4255	5054	7509
FC	C_{in}, C_{out} FLOPs, params	2000	3700	7065
maxpool	HW, C_{in}, K, S	1200	1366	1264
avgpool	HW, C_{in}, K, S	2575	1523	2179
SE	HW, C_{in}, ratio	2000	2000	2000
hswish	HW, C_{in}	1567	1567	1533
channelshuffle	HW, C_{in}	1000	-	1000
bn+relu	HW, C_{in}	2307	2000	-
add+relu	HW, C_{in}	2000	2000	2262
concat	$HW, C_{in1}, C_{in2},$ C_{in3}, C_{in4}	7674	8513	-

Table 6: Main kernels, features and valid data.

train:val:test 7:2:1

Key2-Latency Prediction

Predictor

Random Forests Regression

Use **NNI** to auto-tune hyper-parameters.

Measure latency:

- TFLite backend CPU: Using TFLite benchmark tool.
- TFLite backend GPU: A self-implemented profiler.
- OpenVINO backend VPU: OpenVINO toolkit.

Experiments

Setup

	Device	Processor	Framework
CPU	Pixel4	CortexA76 CPU	TFLite v2.1
GPU	Xiaomi Mi9	Adreno 640 GPU	TFLite v2.1
VPU	Intel NCS2	MyriadX VPU	OpenVINO2019R2[17]

Table 7: Evaluated edge devices.

Experiments

End-to-end prediction evaluation

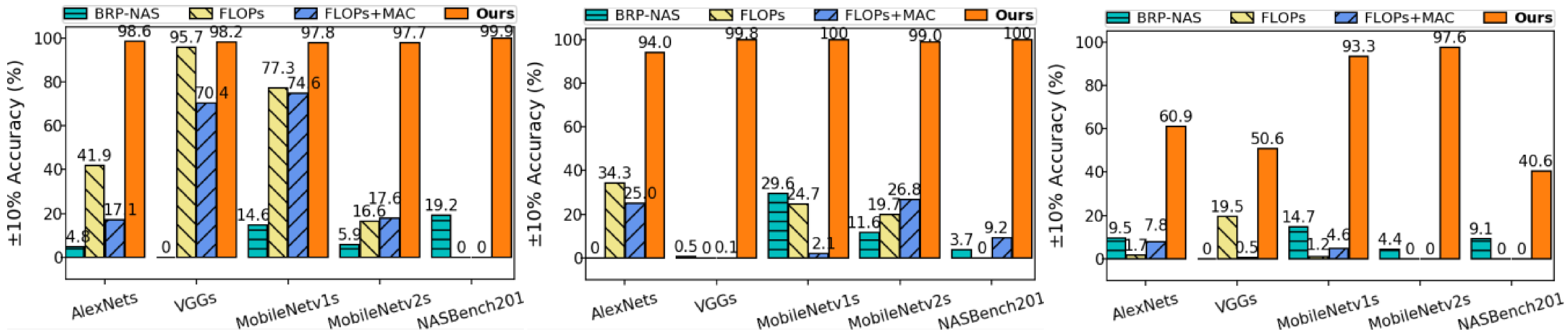


Figure 9: Compared to baseline predictors, nn-Meter achieves much higher ±10% accuracy on unseen models.

Experiments

End-to-end prediction evaluation

Model variants	Mobile CPU RMSE (ms)	Mobile CPU RMSPE (%)	Mobile CPU ±5% Acc.	Mobile CPU ±10% Acc.	Mobile GPU RMSE (ms)	Mobile GPU RMSPE (%)	Mobile GPU ±5% Acc.	Mobile GPU ±10% Acc.	Intel VPU RMSE (ms)	Intel VPU RMSPE (%)	Intel VPU ±5% Acc.	Intel VPU ±10% Acc.
AlexNets	4.02	3.90	81.0%	98.6%	0.93	5.32	72.0%	94.0%	1.17	10.74	23.4%	60.9%
VGGs	185.71	4.84	66.1%	98.2%	12.74	2.97	91.8%	99.8%	85.35	22.25	27.1%	50.6%
DenseNets	7.10	2.76	93.1%	99.9%	1.99	4.52	68.55%	99.9%	2.83	5.89	75.6%	86.3%
GoogleNets	5.69	3.27	85.9%	100%	0.44	1.35	100%	100%	0.94	5.86	39.7%	98.4%
SqueezeNets	7.19	3.59	84.5%	99.9%	1.17	3.85	81.9%	97.9%	1.93	7.08	66.1%	88.5%
ResNets	26.87	4.41	72.3%	98.1%	2.58	3.16	88.8%	99.9%	3.39	7.42	37.9%	84.2%
MobileNetv1s	3.71	4.98	63.8%	97.8%	0.37	2.56	96.9%	100%	1.21	5.90	54.2%	93.3%
MobileNetv2s	3.25	4.84	67.6%	97.7%	0.54	3.93	80.0%	99.0%	1.29	4.26	78.3%	97.6%
MobileNetv3s	2.03	4.34	73.8%	99.0%	0.40	4.02	84.4%	100%	2.47	5.72	47.6%	98.5%
ShuffleNetv2s	2.48	5.01	61.6%	98.3%	-	-	-	-	1.91	6.37	45.6%	91.3%
MnasNets	3.19	5.54	50.9%	99.2%	0.25	1.86	100%	100%	1.76	4.34	77.3%	97.7%
ProxylessNass	3.18	3.44	84.6%	100%	0.61	3.28	95.6%	98.9%	1.97	5.05	65.6%	96.9%
NASBench201	0.44	3.51	82.4%	99.9%	0.12	3.80	75.9%	100%	0.90	18.20	19.3%	40.6%

Table 8: End-to-end latency prediction for 26,000 models on mobile CPU, GPU and Intel VPU.

Experiments

Kernel prediction evaluation

Kernel	CPU		GPU		VPU	
	RMSE (ms)	±10% Acc.	RMSE (ms)	±10% Acc.	RMSE (ms)	±10% Acc.
Conv+bn+relu	6.24	89.1%	6.77	82.0%	18.74	67.9%
DWConv+bn+relu	0.21	97.4%	0.10	98.7%	0.28	89.4%
FC	0.64	94.3%	0.07	96.2%	0.12	93.9%
maxpool	0.12	89.6%	0.06	97.1%	0.21	89.7%
avgpool	1.94	99.0%	0.01	99.7%	0.26	95.4%
SE	0.45	87.1%	0.39	99.8%	0.44	99.0%
hswish	0.16	98.1%	0.01	100%	0.02	100%
channelshuffle	0.14	99.5%	-	-	0.35	100%
bn+relu	0.85	80.7%	0.01	100%	-	-
add+relu	0.10	93.7%	0.003	98.3%	0.02	98.9%
concat	0.09	89.3%	0.42	77.1%	-	-

Table 9: Performance for main kernel predictors.

Experiments

VPU study

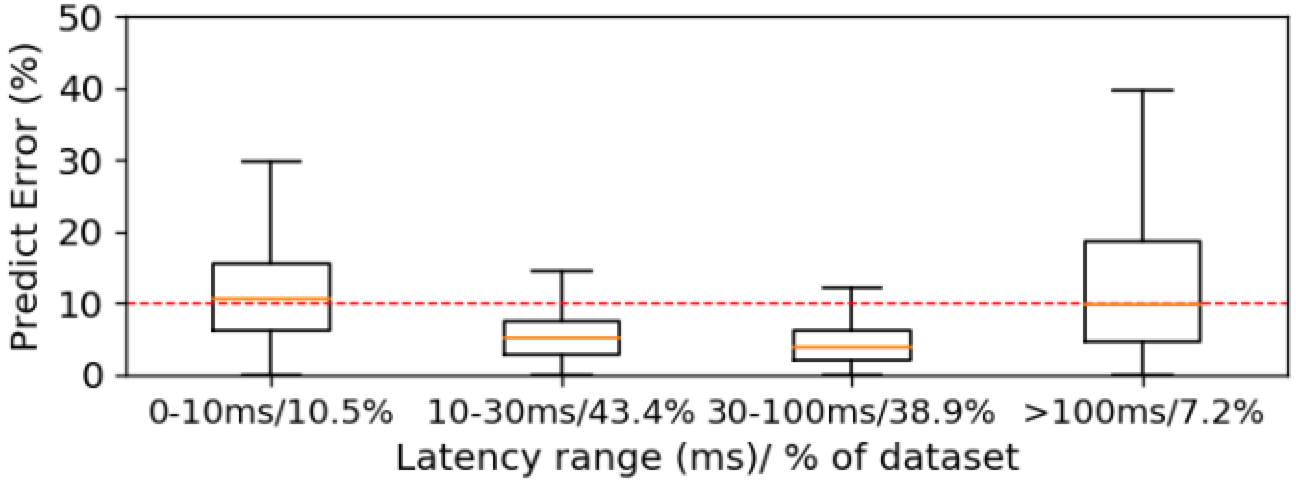
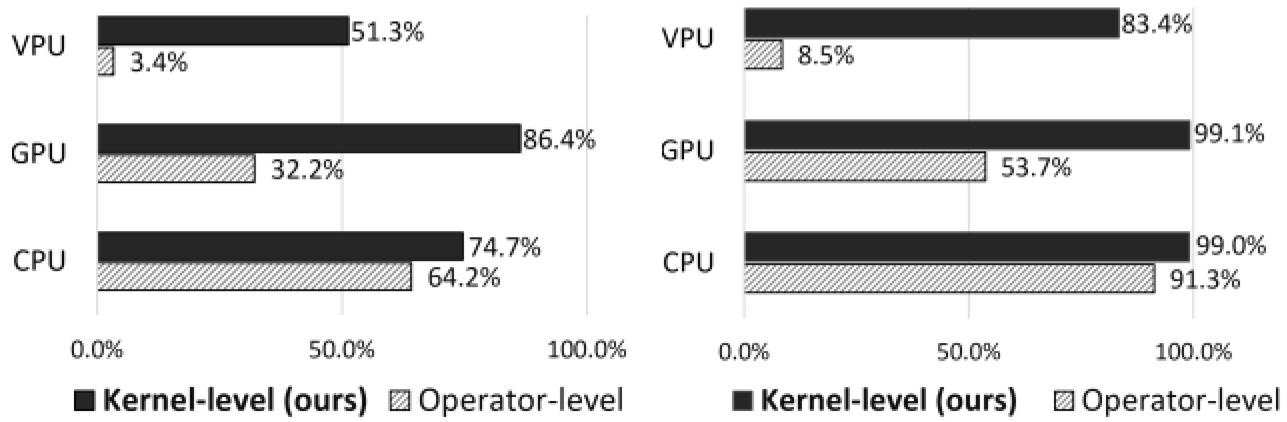


Figure 10: Prediction errors on the VPU. X-axis label: latency range/group size percentages of the dataset.

Experiments

Kernel-level prediction



(a) ±5% accuracy (b) ±10% accuracy

Figure 11: Operator-level approach achieves much lower ±5% and ±10% accuracy on three devices.

Experiments

Sampling performance

Device	Random Sampling		Adaptive Sampling	
	RMSE	±10% Acc.	RMSE	±10% Acc.
CPU	25.47 ms	21.92%	10.13 ms	71.78%
GPU	1.67 ms	48.70%	1.19 ms	75.34%
VPU	7.87 ms	23.98%	7.58 ms	54.33%

Table 10: Under the same amount of sampled data, we achieve better performance than random sampling.

Experiments

Generalization performance

Models (measure on Adreno630)	Adreno640 predictor rmse (ms)	rmspe (%)	±5% Acc.	±10% Acc.	Adreno630 predictor rmse (ms)	rmspe (%)	±5% Acc.	±10% Acc.
AlexNets	8.02	25.40	0.6%	2.3%	0.87	3.61	86.5%	97.9%
VGGs	154.50	24.85	0%	0%	19.47	3.10	89.5%	99.9%
DenseNets	4.44	7.80	18.4%	84.0%	2.41	4.58	67.1%	100%
GoogleNets	6.71	17.03	0%	0%	1.45	3.75	95.8%	100%
SqueezeNets	8.02	19.21	0.4%	3.0%	1.14	3.30	87.3%	100%
ResNets	33.82	21.36	1.3%	11.0%	2.52	2.65	93.4%	100%
MobileNetv1s	0.28	1.92	98.3%	100%	0.23	1.68	99.7%	100%
MobileNetv2s	0.85	5.43	55.6%	97.2%	0.41	3.42	85.7%	99.5%
MobileNetv3s	0.87	8.25	5.2%	87.5%	0.56	6.14	31.5%	99.6%
MnasNets	0.74	5.45	42.2%	99.8%	0.31	2.26	99.9%	100%
ProxylessNass	0.86	4.33	71.5%	100%	0.21	1.11	100%	100%
NASBench201	0.77	14.94	3.2%	17.5%	0.36	6.22	48.1%	90.6%

Table 11: Two different latency predictors for model inference on the Adreno GPU 630.

Experiments

System overhead

	CPU	GPU	VPU
total measure time	2.5 *days*	1 *day*	4.4 *days*

Table 12: Time cost of nn-Meter.

Conclusion

Advantages:

- Kernel detector is a nice idea to detect optimized kernels in black-box frameworks.
- Predictor introduces an adaptive random sampling method to achieve good trade-off between accuracy and complexity.

Disadvantages:

- Only support CNN currently.
- Retrain needed for new version of backends or networks.
- No consideration of dynamic environment.
- No detailed consideration of heterogenous computing.
- No modeling for power consumption.
- Black-box model based predictor.
- No applicable experiments on SOTA NAS or model compression to evaluate its performance.

Thank You !

Nov 14, 2022

Presented by Mengyang Liu