Efficient On-Device Deep-Learning
towards Ubiquitous Intelligence on the Edge

Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University
Computing paradigm shifts

- Centralized
- Distributed
- Intelligent cloud
- Intelligent cloud + edge

Diagram showing the transition from mainframe to personal computing to distributed systems and intelligent cloud.
Distributed devices and data

- **Smart Devices**: 20B IoT devices
- **Smart City**: 250 PB per day
- **Connected Factory**: 1 PB per day
- **Stadium**: 200 TB per game
- **People**: 1.5 GB per day
- **Smart Home**: 50 GB per day
- **Autonomous Vehicle**: 5 TB per day
- **Smart Office**: 150 GB per day
On-device (ubiquitous) intelligence (DL)

- Data explosion from fast growing edge devices
  - E.g., smart surveillance cameras, self-driving cars

- Strong needs of on-device intelligence
  - Low latency
  - High availability and reliability
  - Strong privacy protection
  - Low cost

- Edge devices becoming increasingly powerful
  - Emerging high-perf, low-power, low-cost AI ASIC
Empower every app & device with AI/DL

Affordable AI models tailored for diverse hardware

Highly-optimized software stack & efficient hardware for AI

Security & privacy, model protection, explainable AI, debugging

On-device, continuous, collaborative learning loop
Innovations of on-device DL stack

Efficient neural network (NN) design

Edge NN Frameworks

AI Chips
NN design and deployment

**NN Design**

- Design Space:
  - # of layers, op structure, channel, …
  - constraints (e.g., FLOPs)

- Manual Design
- NAS
- Pruning

**Model Deployment**

- Quantization
- Conv
- Re-quantize
- BN
- Re-quantize
- ReLu
- Re-quantize
- Dequantization

**Gap**

- Current NN design does not consider platform features

**Framework opt. e.g. op fusion**

- CPU
- GPU
- DSP
- TPU
- NPU
- …
Does less FLOPs mean less latency?

- **MobileNetV3**
  - Latency: 4 ms
  - Model accuracy: 74.7%
  - 209M FLOPs

- **MobileNetEdgeTPU**
  - Latency: 3.6 ms
  - Model accuracy: 75.6%
  - 990M FLOPs

Less FLOPs $\neq$ less latency, but can harm model accuracy.
Does a fast model run fast on every hardware?

MobileNetV3 vs MobileNetV2:
- Cortex A76 CPU: 25% faster

MobileNetV3 vs MobileNetV2:
- VPU: 71% faster
A Behavior Study for Neural Networks

- **# of Channels**: The latency of Conv increases in a step pattern with the # of output channels.
- **Block**: The relative latency of a NN block varies greatly on different platforms.
- **Activation Function**: Activation functions can have big impact on latency, particularly for Swish and HardSwish.
- **Kernel Size**: The Conv latency increases much less with kernel size on AI accelerators than on the CPU.
- **Quantization**:
  - The use of INT8 on the NPU achieves > 11× speedup, while CPU only achieves < 3.6×
  - INT8 can dramatically decrease inference accuracy of various models
Efficient NN design must consider hardware characteristics.

How to get a good model?
Efficient NN design for diverse edge hardware

Profiling and modeling

HW-specific predictors of latency and energy

NN Design

Design Space: # of layers, op structure, channel, … constraints latency, energy

Manual Design

NAS

Pruning

Model deployment

Models

Edge TPU

VPU

HPU

NPU

KPU
Existing work on latency prediction

• **FLOPs**-based prediction
  • Pros: very simple
  • Cons: not a direct metric of inference latency

• **Operator**-level prediction
  • Pros: stable primitive operators (conv2d, pooling, activations...)
  • Cons: unaware of graph-level optimizations

• **Model**-level prediction
  • Pros: learn graph-level optimization automatically
  • Cons: cannot generalize to unseen model structures
Challenge: framework optimizations

- Backend-independent opt.
  - Constant folding
  - Common expression elimination
  - ...
- Backend-dependent opt.
  - Operator fusion
  - ...

Diagram:

- Designed model
  - Backend independent opt.
  - Backend dependent opt.
    - CPU backend1 (eg Eigen lib.)
    - CPU backend2 (eg NNPack lib.)
    - GPU backend1 (eg OpenCL)
    - ...
    - Movidius backend
Impact of operator fusion

- **Operator fusion** has a great impact on inference latency

```
Model graph
Conv ➔ Active
Conv+ Active
```

```
 Backend implementation

_kernel conv_2d_1x1 ( ) {
 for(i=0;i<out.row;i++)
 for(j=0;j<out.col;j++)
 for(cout=0;cout<out.chan;cout++)
 for(cin=0;cin<in.chan;cin++)
 out[i][j][cout]+=in[i][j][cin]*filter[cout][cin];
}

_kernel active ( ) {
 for(i=0;i<out.row;i++)
 for(j=0;j<out.col;j++)
 for(c=0;c<out.chan;c++)
 out[i][j][c]=active(in[i][j][c]);
}
```

```
Operator fusion
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Latency</th>
<th>Operator sum</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>45.57ms</td>
<td>51.23ms</td>
<td>12.42%</td>
</tr>
<tr>
<td>GPU</td>
<td>10.18ms</td>
<td>12.31ms</td>
<td>20.92%</td>
</tr>
<tr>
<td>VPU</td>
<td>22.64ms</td>
<td>33.86ms</td>
<td>49.56%</td>
</tr>
</tbody>
</table>

MobileNetv2
nn-Meter: kernel-level latency prediction

• Kernel: the basic execution unit on a device
  • Can be a single operator or a fusion of multiple operators

• Divide a whole model into kernels, conduct kernel-level prediction
  • Model latency is the sum of all kernels

• Problems:
  1. How to detect kernels? (Kernel Detection)
  2. How to predict accurately for each kernel? (Adaptive Data Sampling)
nn-Meter tech#1: automatic kernel detector

Fusion rule detection for black-box devices

- A set of test cases
- For every two operators, we generate 3 graphs
- Compare the latency difference

\[
\text{Op1 and op2 are fusible if: } T_{\text{op1}} + T_{\text{op2}} - T_{\text{(op1, op2)}} > \alpha \cdot \min(T_{\text{op1}}, T_{\text{op2}})
\]
nn-Meter tech#1: Automatic kernel detector

Fusion rule detection for black-box devices
  • A set of test cases:
  • For every two operators, we generate 3 graphs
  • Compare the latency difference

Kernel search by the fusion rules
  • Apply the fusion rules to search maximum fused operators in target model

A resnet18 block example
Kernel-latency prediction: challenges

• Large sample space, e.g., Conv
  Collected from 24 widely used CNN models from PyTorch model zoo, Conv has $1 \times 10^9$ of configurations to sample!

<table>
<thead>
<tr>
<th>dimension</th>
<th>sample space</th>
</tr>
</thead>
<tbody>
<tr>
<td>input $HW$</td>
<td>224, 112, 56, 32, 28, 27, 14, 13, 8, 7, 1</td>
</tr>
<tr>
<td>kernel size $K$</td>
<td>1, 3, 5, 7, 9</td>
</tr>
<tr>
<td>stride $S$</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>$C_{cin}$</td>
<td>range(3, 2160)</td>
</tr>
<tr>
<td>$C_{out}$</td>
<td>range(16, 2048)</td>
</tr>
</tbody>
</table>
Kernel-latency prediction: challenges

- Non-linear latency on edge devices
- Random sampling misses crucial data points
nn-Meter tech#2: adaptive data sampler

Sample the most beneficial data (kernel configuration) instead of random sampling

- Sample configurations that are likely to be considered in model design
  - Prior possibility distribution: learned from model zoo
  - Fine-grained sampling around inaccurate prediction data
nn-Meter Evaluation

• **Prediction accuracy**: 99.0% (CPU), 99.1% (Adreno640 GPU), 99.0% (Adreno630 GPU) and 83.4% (Intel VPU)

• **Generalization performance on unseen model graphs**
  • Comparison baselines: FLOPs, FLOPs+MAC, BRP-NAS (GCN),
  • On average: nn-Meter achieves 89.2%, significantly better than FLOPs (22.1%), FLOPs+MAC (17.1%), and BRP-NAS (8.5%)
We got a good model.

How does it run on real devices?
Low utilization of mobile big.Little CPU

CPU utilization % for CNN

Unbalanced task distribution by OS inter and intra core clusters

Big core cluster

Little core cluster
Why is distribution unbalanced on the CPU?

Execution flow of matrix multiplication

1) Block partition for parallelism
2) Copy blocks into continuous memory space
3) Schedule tasks to thread queues

- Ignore hardware asymmetry
- Ignore resource constraints
- Ignore the interference-prone environment
- Redundant data copy
- Ignore hardware asymmetry
- Ignore data locality
AsyMo: optimize DL inference on big.Little CPU

- Accelerate edge DL inference with lower energy cost
Cost-model-based block partition

Cost for a task: computation + memory
\[ Cost_{task} = t_{\text{comp}} \cdot mc \cdot kc \cdot nc + t_{\text{mem}} \cdot (mc \cdot kc + kc \cdot nc + 2 \cdot mc \cdot nc) \]

Cost for a sequential unit: \( Cost_{seq} = K/kc \cdot Cost_{task} \)

Cost for parallel calculation: parallel task number x \( Cost_{seq} \)
\[ Cost_{par} = \frac{1}{\text{Thread}\#} \cdot \frac{M}{mc} \cdot \frac{N}{nc} \cdot Cost_{seq} \]
\[ = \frac{1}{\text{Thread}\#} \cdot \left( t_{\text{comp}} \frac{MNK}{nc} + t_{\text{mem}} \left( \frac{N}{nc} \frac{MK}{mc} + \frac{M}{mc} \frac{NK}{kc} + \frac{2}{kc} \frac{MN}{kc} \right) \right) \]

Other cost: unparallel + task schedule + framework
\[ Cost_{other} = p \cdot Cost_{seq} + t_{\text{sched}} \cdot \frac{M}{mc} \cdot \frac{N}{nc} \cdot \frac{K}{kc} + t_{\text{fram}} \]

Total cost: \( Cost = Cost_{par} + Cost_{other} \)
Optimized execution flow of matrix multiplication

One-run initialization

- Block partition
- Params layout
- Copy features

Inference run

- Tasks scheduling and run

Big core cluster

Little core cluster

Pin thread on core

Better data locality

No work stealing from big to little
Performance and energy improvement

Asymo vs TensorFlow on Kirin 970 + Android 9 Pie

Both @max CPU frequency

Pre-copy params enable parallel implementation

TensorFlow @ OS frequency setting
Asymo @ picked efficient CPU frequency
The world is a computer
Thanks!

liuyunxin@air.tsinghua.edu.cn
https://yunxinliu.github.io/