ACM SenSys 2023

nnPerf

Demystifying DNN Runtime Inference Latency on Mobile Platforms

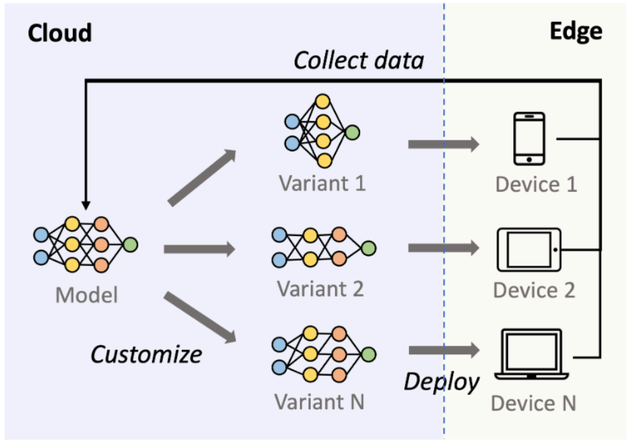
SenSys'23

Haolin Chu, Xiaolong Zheng, Liang Liu, Huadong Ma
Beijing University of Posts and Telecommunications

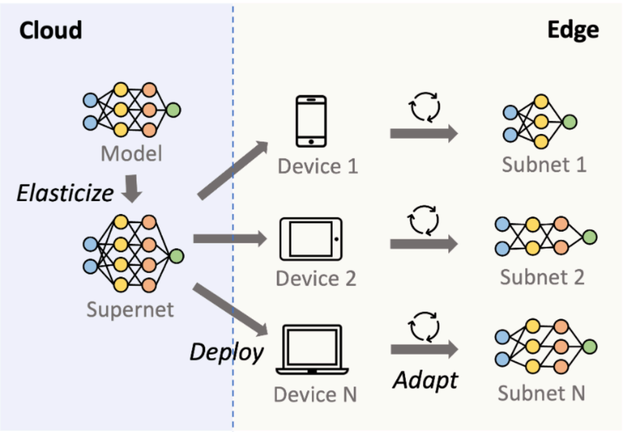
Introduction

Method to optimize model architecture

- **Train-then-deploy**
  - Train on server
  - Fit supernet to mobile devices
- **Search for efficient model**
  - NAS

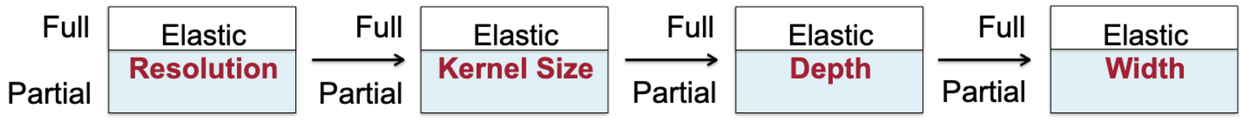


(a) Pre-deployment on-cloud model generation (conventional).



(b) Post-deployment on-device model adaptation (ours).

Latency matters



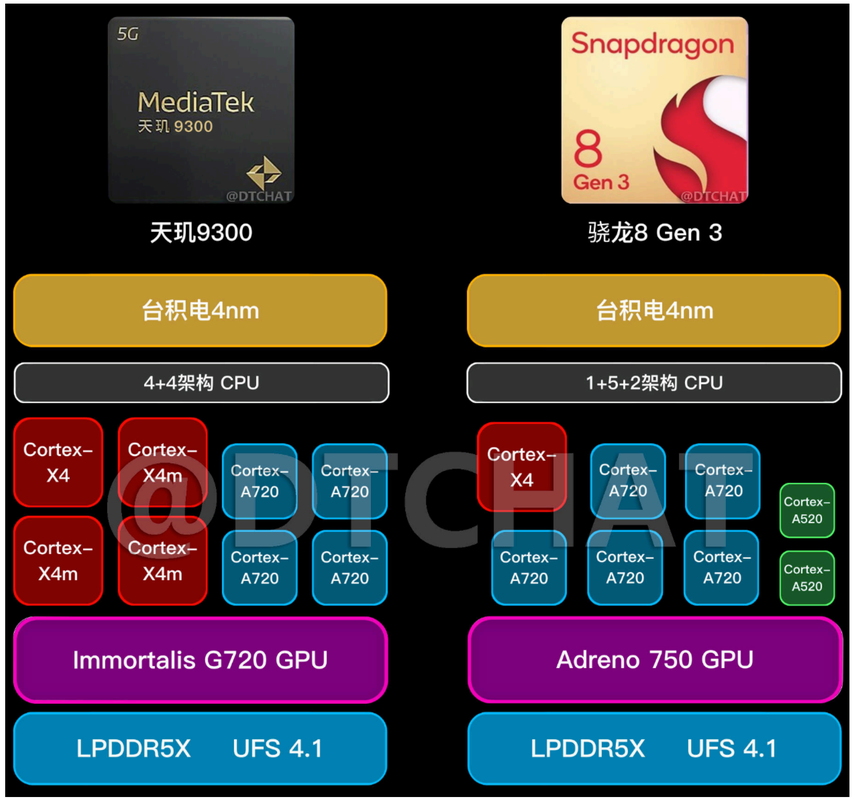
AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments

Once for All: Train One Network and Specialize it for Efficient Deployment

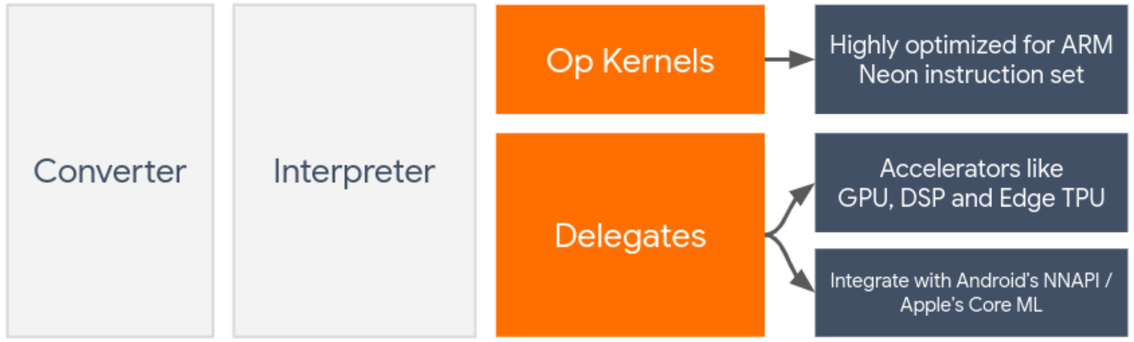
Introduction

Mobile platform's Hardware: variable resources

- **Heterogeneous SoCs**
  - Big’little -> Dynamic
  - CPU, GPU, DSP(NPU) …
- **Dynamic voltage and frequency scaling(DVFS)**
  - based on workload and temperature



Diversity



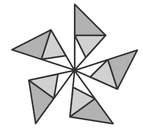
Introduction

Mobile platform's Software: dynamic environment

- **Platform**
  - AI platform
- **Performance scheduling strategy**
  - Energy/performance -> voltage & freq
  - Offloading to which component/core
  - Application & services in foreground & background
  - memory/compute bound

NCNN

TensorFlow Lite



ONNX RUNTIME

Dynamics

Introduction

Resolution: resource-aware profiler

- **Profiler: PC & Mobile**
  - NVIDIA: Nsight
  - mobile: ?
- **Requirement of wonderful profiler**
  - Cross-stack design
    - At the granularity of model/op/**kernel**
  - Realtime and on-device profiling
    - more accurate
    - fine-grained latency feedback

Table 1: Comparison of nnPerf with existing DNN model profilers designed for mobile platforms.

Systems	e2e latency	DNN Op		DNN Kernel		Process Timeline	On-device	Real-time results
		CPU	GPU	CPU	GPU			
nnPerf	✓	✓	✓	✓	✓	✓	✓	✓
AGI [6] & Perffeto [25]	✓	✓	×	×	×	✓	×	×
Snapdragon Profiler [28]	✓	✓	×	×	×	✓	×	✓
Android Studio profiler [2]	✓	✓	×	×	×	✓	×	✓
Adreno GPU Profiler [3]	✓	✓	×	×	×	✓	×	×
Tensorflow Benchmark [33]	✓	✓	×	×	×	×	×	×
Perfdog [32]	✓	×	×	×	×	✓	×	×
Xcode-Instrument [36]	✓	✓	✓	×	×	✓	×	✓
Soloπ [29] & Emmagee [11]	✓	×	×	×	×	×	✓	×

real-time on-device latency profiler on mobile platforms

Motivation 1

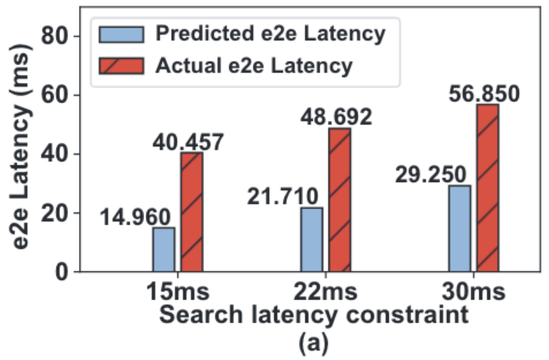
Latency Prediction is Inaccurate

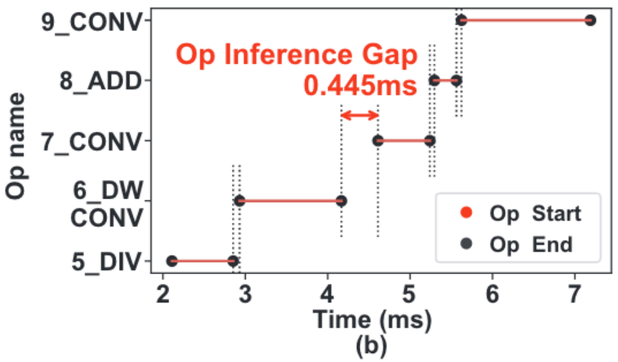
#1 Disadvantage of current method

- No profilers support fine-grained **(kernel-level)** on-device profiling in real-time
  - Mainstreaming method is **add-on of op-level** for e2e latency
  - offline method is inaccurate (LUT)

#2 Reasons of inaccurate latency

- **op latency may change** with the underlying hard-ware resources
  - compiler may choose different kernels
- **ops could be fused together** during compiling
  - implicit glue op (concat/add)
- **data preparation delay** is ignored
  - main cause of latency gap



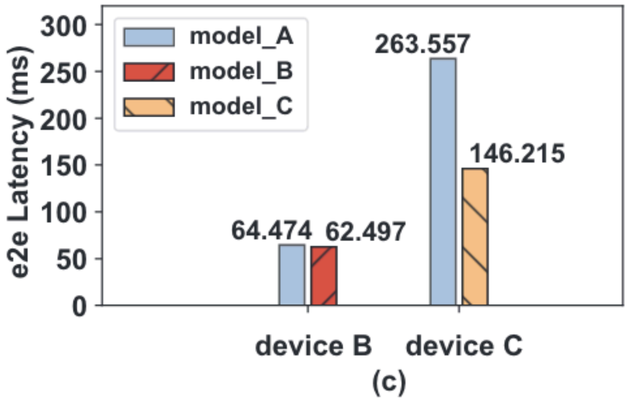


Motivation 2

Hardware Heterogeneity Requires Fine-grained Profiling

#1 Tailored design lacks scalability

- **different mobile devices differ in their computing & memory capacity**
  - **model** customized for a device is unlikely to remain optimal for other devices.
  - Hardware heterogeneity could change **op** executions



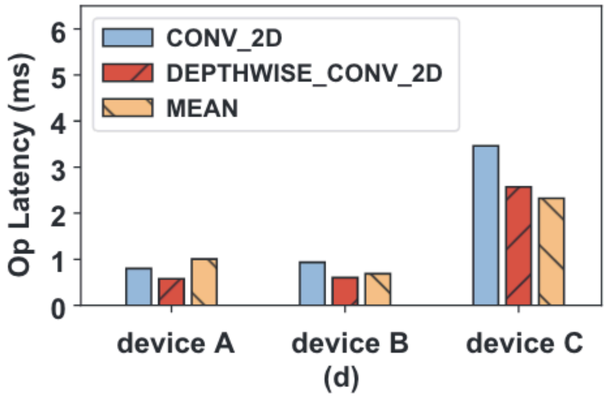


Table 2: List of mobile platforms.

Device	SoC	CPU	GPU	RAM	OS Version
(A) Google Pixel 3XL	Snapdragon 845	Kryo 385	Adreno 630	4GB	Android 12 beta
(B) Samsung Galaxy Note10	Snapdragon 855	Kryo 485	Adreno 640	8GB	Android 11
(C) Xiaomi 10	Snapdragon 865	Kryo 585	Adreno 650	8GB	MIUI 13.0.7
(D) TB-RK3399ProD	Rockchip RK3399	Cortex-A72&A53	Mali-T860	6GB	Android 8.1

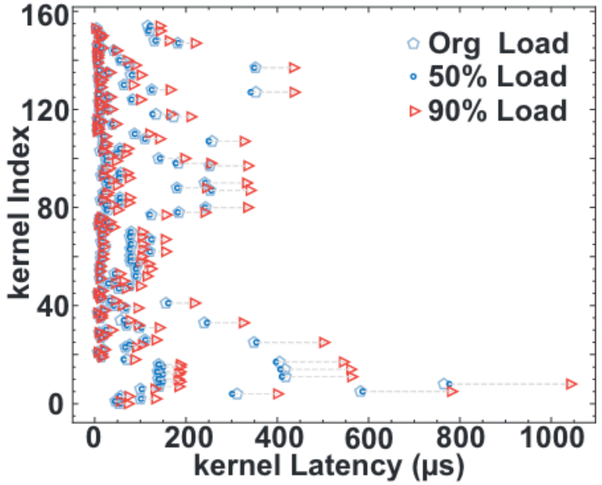
Close the gap between deployment & design → **profile model at fine-grained granularity**

Motivation 3

Impact of Resource Dynamics

#1 Impact of APP activities

- **APP's priority changes** -> alter memory & computing resources
  - latency **variation grows** significantly (b,c)
  - **context switching have impact** as well (d)
  - **memory load settings have impact** as well (a)



(a)

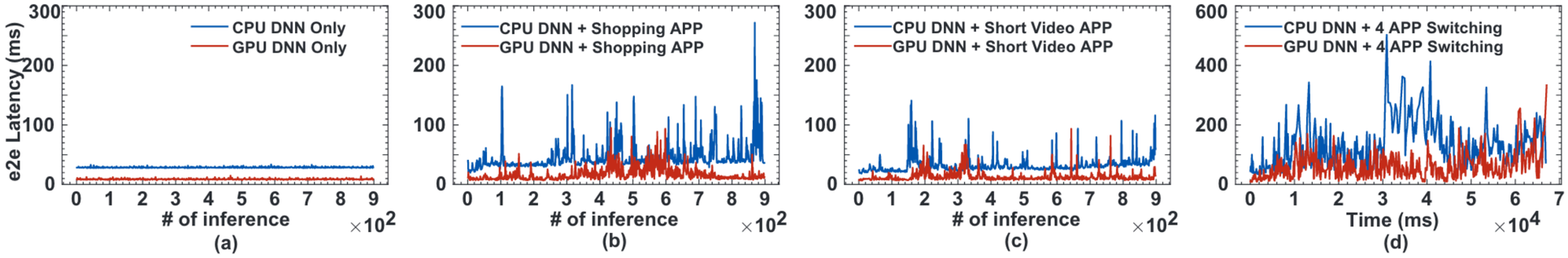


Figure 2: The impact of resource dynamics on DNN run-time latency. (a): the e2e latency of MobienetV3-large in the absence of a co-running APP. (b): the model e2e latency in the presence of a shopping APP. (c): the model e2e latency in the presence of a short video APP. (d): the model e2e latency in the presence of switching between four mobile applications.

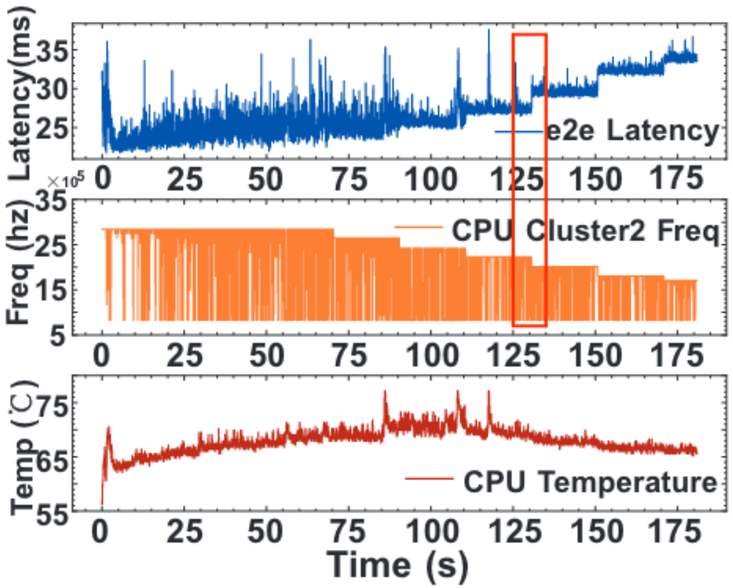
Motivation 3

Impact of Resource Dynamics

#2 Impact of thermal throttling

- **DVFS**
  - Dynamic voltage and frequency scaling
  - mobile devices suffer from **overheating** (different from PC/server)
  - affect **computing resource** - CPU & GPU freq

latency strongly correlates with the freq of the underlying hardware accelerator



(b)

Motivation 4

The Need for a Cross-Stack Profiler

#1 Course-grained granularity

- model(e2e), op, **kernel**-level

#2 Offline with an auxiliary device

- Existing profilers run on auxiliary device (PC host) via adb
  - make **on-device resource-aware optimization challenging**
  - PC can only analysis **after inference is finished** via the data transformed via adb in batch

Overview of nnPerf's design

Overview of nnPerf

- **Cross-stack latency profiling** → **accurate multi-level profiling**
  - e2e model inference profiling
  - op-level profiling (on-demand timer injection)
  - **kernel-level profiling**
- **light-weight time synchronization** → **Align CPU&GPU profiling results**
  - Unified time wrapper
  - **Time synchronization**

nnPerf: Cross-Stack Latency Profiling

Accurate e2e model inference latency profiling

- **Current method**
  - API of model compiler framework/offline code injection
  - time cost on **library invokes & language conversion** (Java JNI -> TFLite C++)
- **nnPerf solution**
  - embed **trace timers** in ML framework **low-level C++ library**
  - build **asynchronous output module** to record **log files**
    - save **timestamps in real-time**
    - also log **op's name & kernel merging information (for op, kernel-level)**
    - user **access without root**

nnPerf: Cross-Stack Latency Profiling

On-demand timer injection design for operator-level profiling

- **Current method**
  - function pointer *OpInvoke* is pushed into *execution_plan_queue* once the op is selected
  - record the time pointer stays in queue
  - time cost on **data preparation** (tensor copy) between two ops
- **Challenges**
  - can't trace down to C++ core and **set timer in advance for op (unaware before compiling)**
  - can't just blindly inject to **C++ library (TFLite support *custom_op* from custom library)**

nnPerf: Cross-Stack Latency Profiling

On-demand timer injection design for operator-level profiling

- **nnPerf solution**
  - **adaptive on-demand** timer injection at model's **compiling stage**
    - TFLite go through *executoin_plan_queue* with pointer *FindOp*
    - get ID of op -> get op's name via <*op_name, op_id*> -> locate op's execution function *Invoke()*
    - inject timer to op's core execution function *Eval()*
    - log timestamps & op id to log files

nnPerf: Cross-Stack Latency Profiling

Down to the kernel-level latency profiling

- **Current method**
  - not available yet
- **Challenges**
  - **Closed-source kernel implementation**
    - can’t trace code to locate the function call of TFLite C++ core (e2e-level) or timer injection (op-level) **(GPU SDKs closed-source)**
  - **Unknown kernel-fusion rules**
    - can’t know kernel fusion situation before inference **(device-dependent)**

nnPerf: Cross-Stack Latency Profiling

Down to the kernel-level latency profiling

- **nnPerf solution**
  - **customized event registration** mechanism
    - motivated by *cl_events* in OpenCL library -> *nnPerf_event*

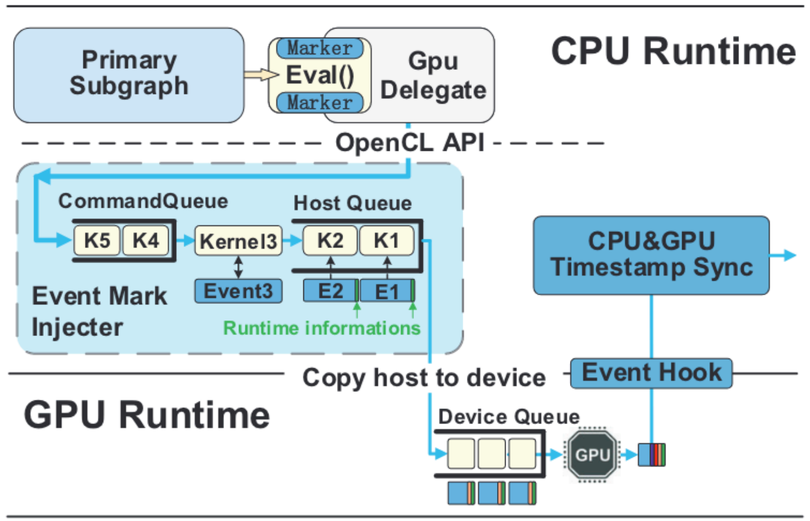


Figure 4: An illustration of kernel execution profiling. We register a customized event for each kernel during the enqueue-dequeue process.

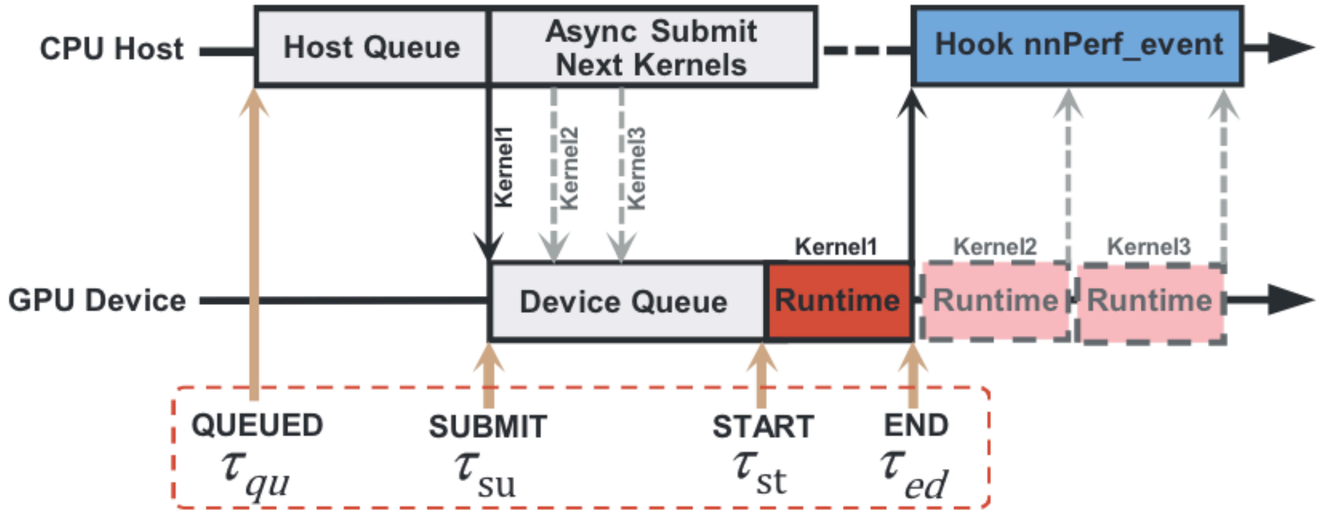


Figure 5: An illustration of kernel lifecycle.

nnPerf: Time Synchronization

Cause of problem

- **CPU-GPU co-processing**
  - CPU - tensor & kernel preparation
  - GPU - kernel execution
- **Different underlying clock sources & timestamp APIs**
  - different core - different clock sources & freq
  - chaos in profile timeline & timestamp results

nnPerf: Time Synchronization

Unified Time Wrapper

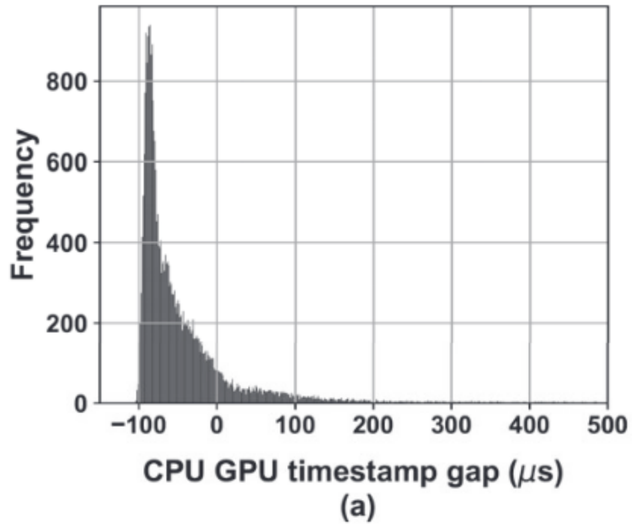
- **Current method**
  - **Timing API** *gettimeofday()* of Tensorflow Benchmark tools **(only support ms precision)**
  - **clock resource** such as *RDTSC & basic_TSC*
    - affected by **out-of-order execution & DVFS -> non monotonic timestamps**
- **nnPerf solution**
  - use **nanosecond-granularity time event** by C++ *Chrono* library
  - replace original counter with **monotonic clock resource** *arch_sys_counter*

monotonic & fine-grained
clock resource

nnPerf: Time Synchronization

Time synchronization

- **Challenges**
  - **Profile disordered timelines errors caused by dynamics clock offset**
    - DVFS change freq -> affect clock offset
  - **Additional overhead in black-box GPUs impeded realtime profiling performance**
    - **hidden code** exist between CPU host & black box GPU (data copy/work-items assignment)



nnPerf: Time Synchronization

Time synchronization

- **nnPerf solution**
  - **light-weight time synchronization** mechanism
    - $\Delta t = \tau_{qu} - t_{qu}$, $\delta = \tau_{qu} - t'_{qu} - \Delta t$ ⟶ **LLS**

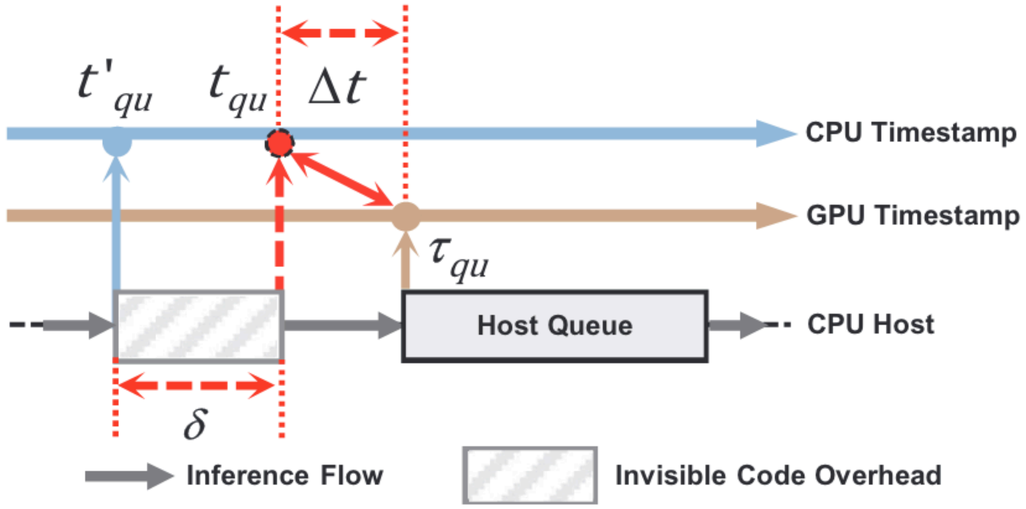


Figure 6: Time offset between CPU and GPU clock.

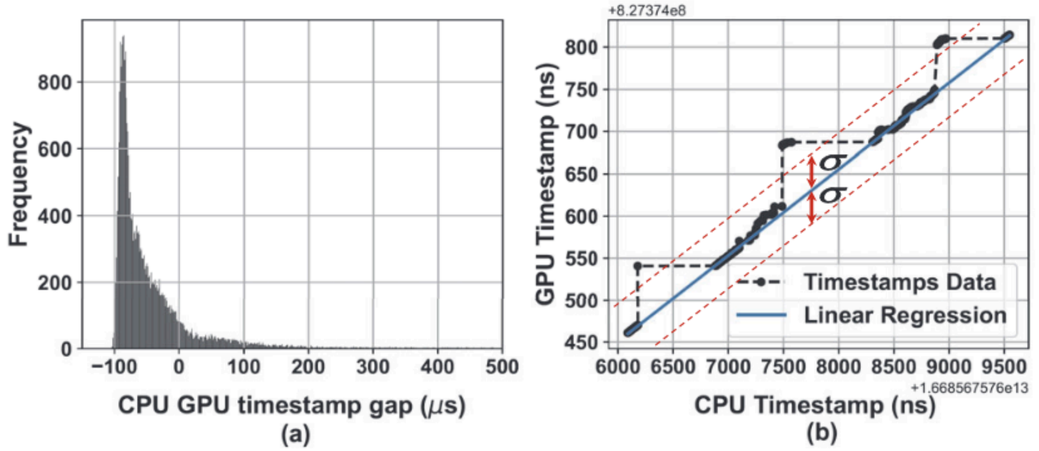


Figure 7: (a) Distribution of the measurement offset. (b) Leveraging linear regression to mitigate the offset.

nnPerf: Time Synchronization

Time synchronization

- **nnPerf solution**
  - **light-weight time synchronization**
    - $\Delta t = \tau_{qu} - t_{qu}$, $\delta = \tau_{qu} - t'_{qu} - \Delta t$

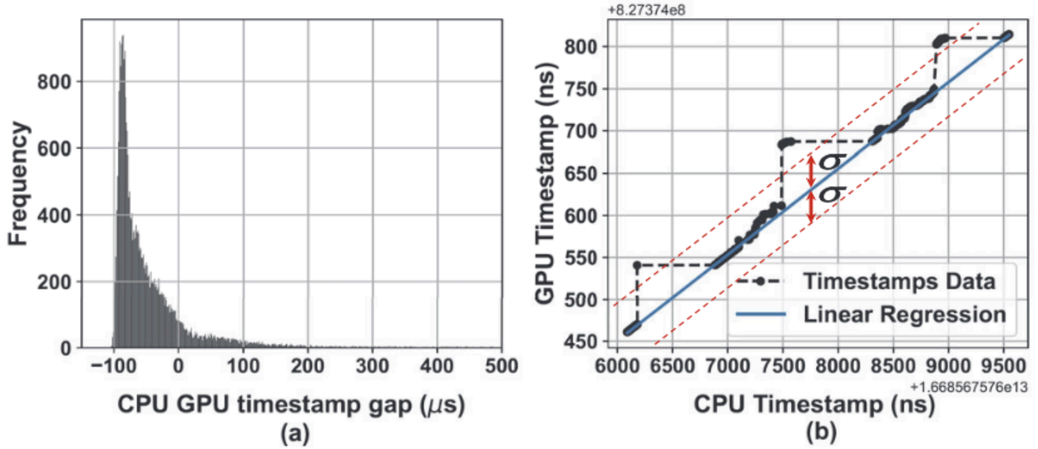


Figure 7: (a) Distribution of the measurement offset. (b) Leveraging linear regression to mitigate the offset.

Algorithm 1 Dynamic Fitting Strategy

```
1:  Input :Pair<t'_qu,τ_qu> 𝒯, *sycn_q, *n, σ, k
2:  Output :Synchronized GPU timestamp
3:  function SYNC(𝒯, &sycn_q ,σ, &n, k)
4:      n ++; // Sliding window count
5:      if | 𝒯.second - 𝒯.first |≤ σ then
6:          sycn_q.pop(); sycn_q.push(𝒯);
7:          if n > k then // Re-fit with recent 𝒯
8:              n = 1;
9:              LR.Regression(sycn_q); // LLS fitting
10:             queue<pair<double, double» empty;
11:             swap (empty, sycn_q); // Clear sycn_q queue
12:         end if
13:         return LR.Predict(𝒯.second);
14:     else if | 𝒯.second - 𝒯.first |> σ then
15:         // Reuse previous fitting results
16:         return LR.Predict(𝒯.second)
17:     end if
18: end function
```

nnPerf: Time Synchronization

Time synchronization

- **nnPerf solution**
  - cold-start: **empty kernel injector**
    - $$\sigma = 2\sqrt[2]{\frac{1}{3}[(\Delta t_1 - m)^2 + (\Delta t_2 - m)^2 + (\Delta t_3 - m)^2]}$$

$$m = \frac{1}{3}(\Delta t_1 + \Delta t_2 + \Delta t_3).$$

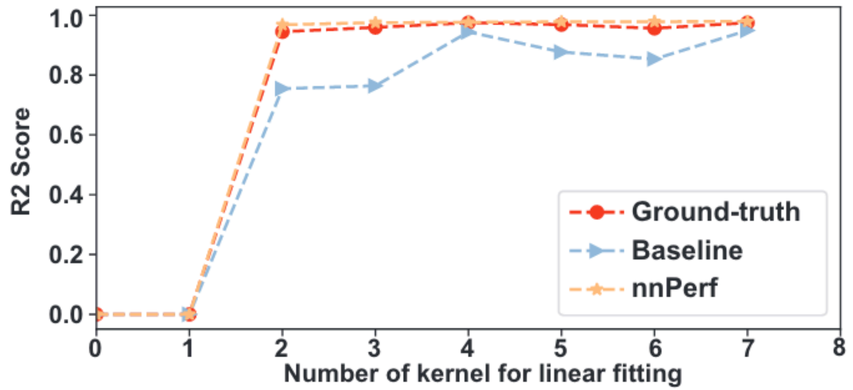


Figure 8: Linear fitting in different numbers of kernels. Averaging over 50 inferences on a Xiaomi10 phone.

Algorithm 1 Dynamic Fitting Strategy

Input :Pair<t'_{qu},τ_{qu}> $\mathcal{T}$, *sycn_q, *n, σ, k
Output :Synchronized GPU timestamp
function SYNC($\mathcal{T}$, &sycn_q ,σ, &n, k)
    n ++; // Sliding window count
    if | $\mathcal{T}$.second - $\mathcal{T}.first$ |≤ σ **then**
        sycn_q.pop(); sycn_q.push($\mathcal{T}$);
        if $n > k$ **then** // Re-fit with recent $\mathcal{T}$
            n = 1;
            LR.Regression(sycn_q); // LLS fitting
            queue<pair<double, double» empty;
            swap (empty, sycn_q); // Clear sycn_q queue
        end if
        return LR.Predict($\mathcal{T}$.second);
    else if | $\mathcal{T}$.second - $\mathcal{T}.first$ |> σ **then**
        // Reuse previous fitting results
        return LR.Predict($\mathcal{T}$.second)
    end if
end function

Implementation

Implementation of nnPerf

- **TFLite Android ARchive file (ARR)**
- **nightly-snapshot library**

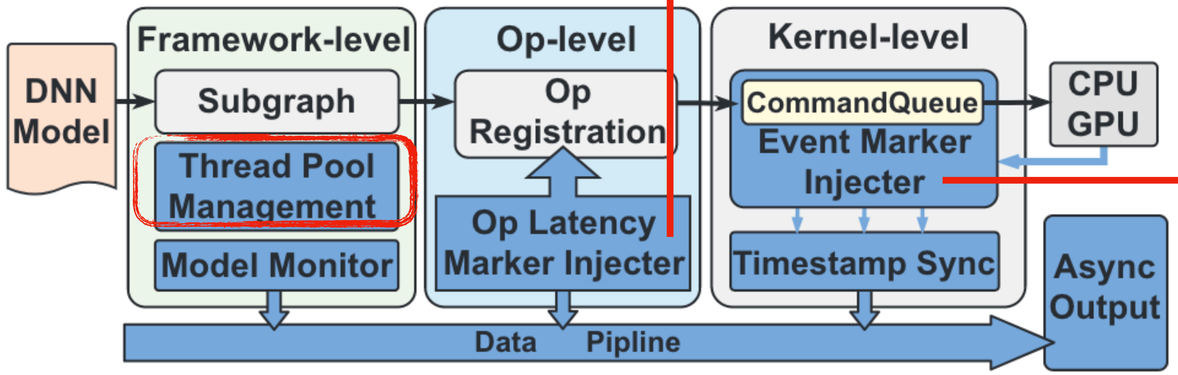


Figure 9: nnPerf implementation breakdown. Each design module in nnPerf is highlighted in blue.

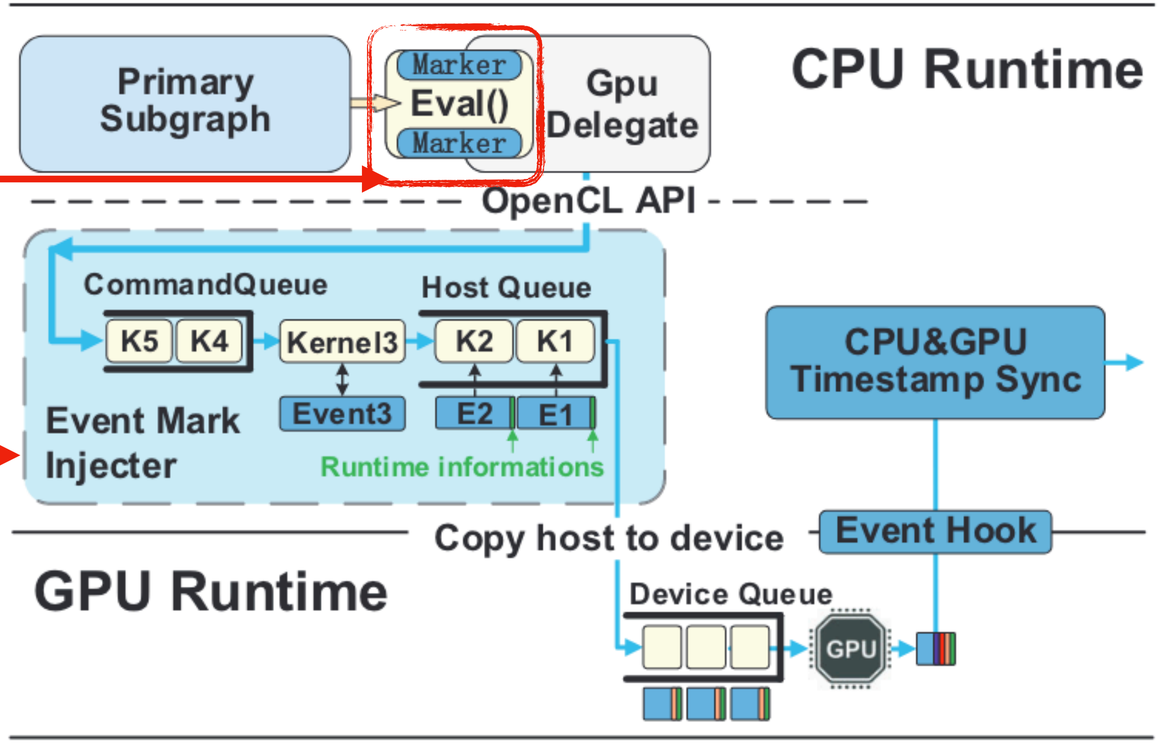


Figure 4: An illustration of kernel execution profiling. We register a customized event for each kernel during the enqueue-dequeue process.

Evaluation

Setup

Table 2: List of mobile platforms.

Device	SoC	CPU	GPU	RAM	OS Version
(**A**) Google Pixel 3XL	Snapdragon 845	Kryo 385	Adreno 630	4GB	Android 12 beta
(**B**) Samsung Galaxy Note10	Snapdragon 855	Kryo 485	Adreno 640	8GB	Android 11
(**C**) Xiaomi 10	Snapdragon 865	Kryo 585	Adreno 650	8GB	MIUI 13.0.7
(**D**) TB-RK3399ProD	Rockchip RK3399	Cortex-A72&A53	Mali-T860	6GB	Android 8.1

Evaluation

System performance — model & op-level profiling accuracy

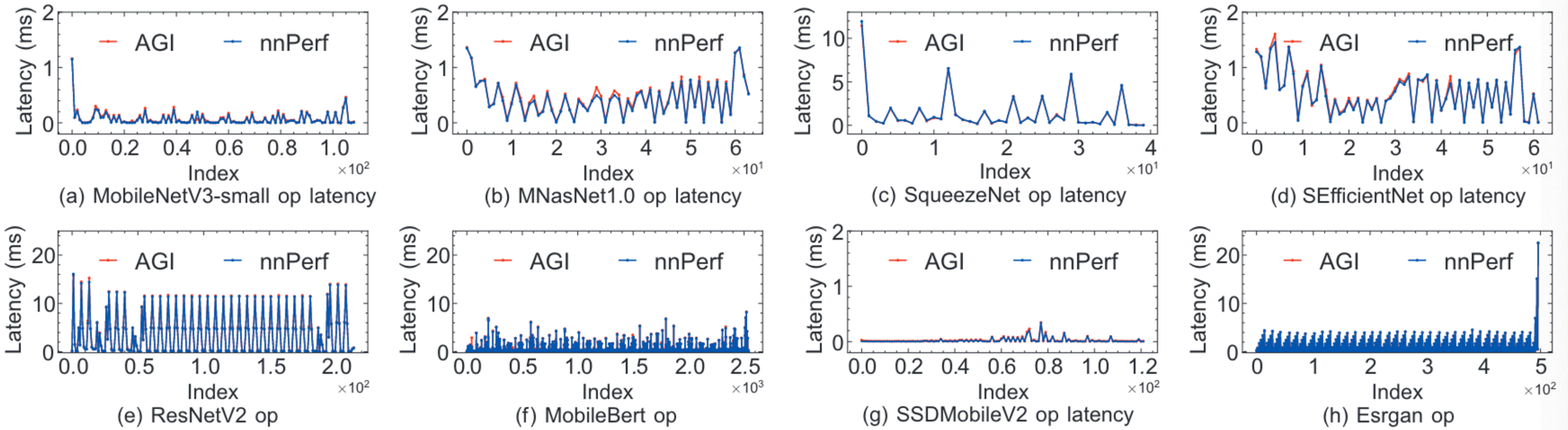


Figure 10: Comparison of real-time on-device nnPerf and offline with an auxiliary device AGI profiled op inference latency.

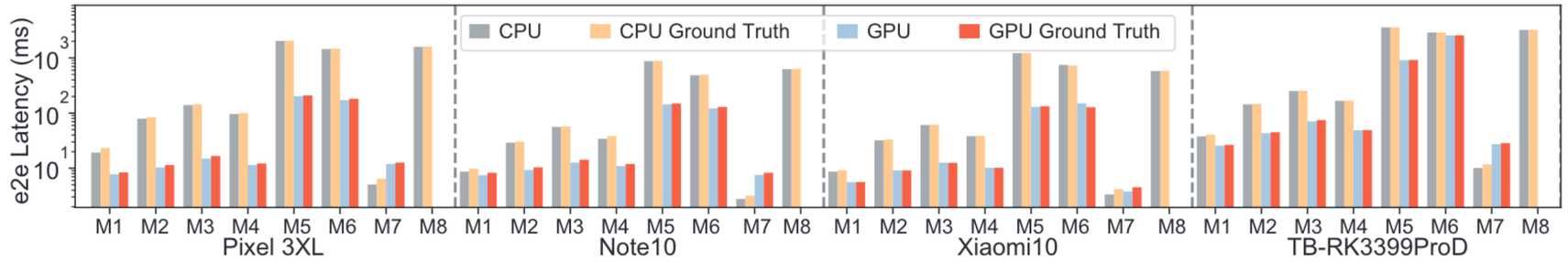


Figure 11: e2e latency compare with ground truth (AGI) on different devices (M1 to M8 are mobilenetv3-small, mnasNet1.0, squeezeNet, efficientNet, resNetV2 101, mobileBert, ssd-mobileV2 and esrgan in sequence).

Evaluation

System performance —kernel-level profiling accuracy

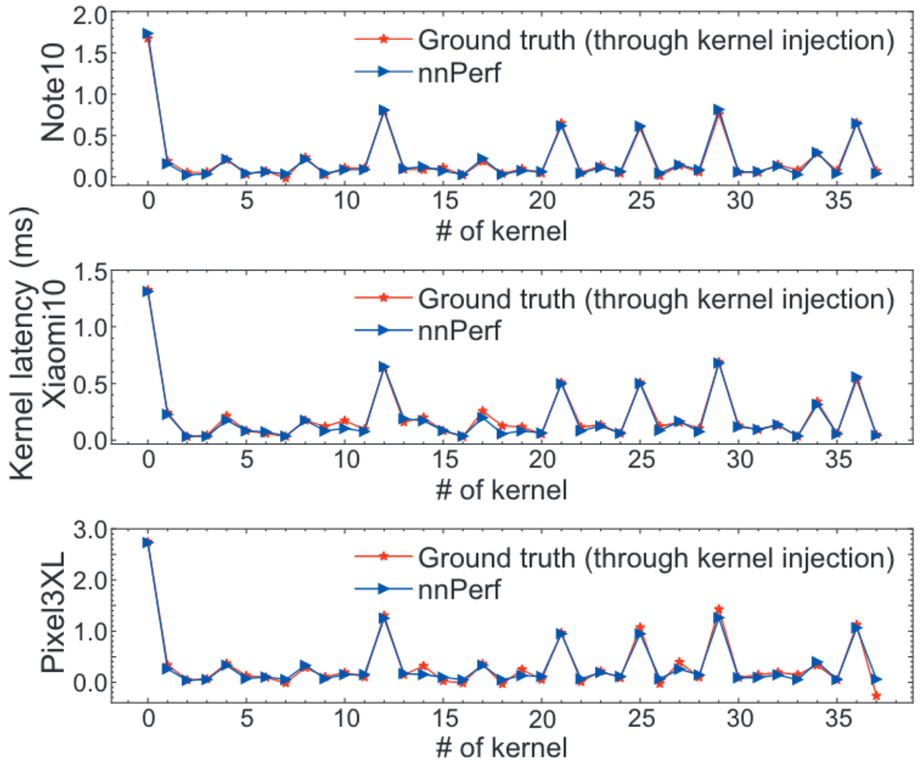


Figure 12: Comparison of nnPerf and groundtruth method profiled kernel inference latency. The results are averaged over 40,0000+ inferences.

Evaluation

System performance — time synchronization accuracy

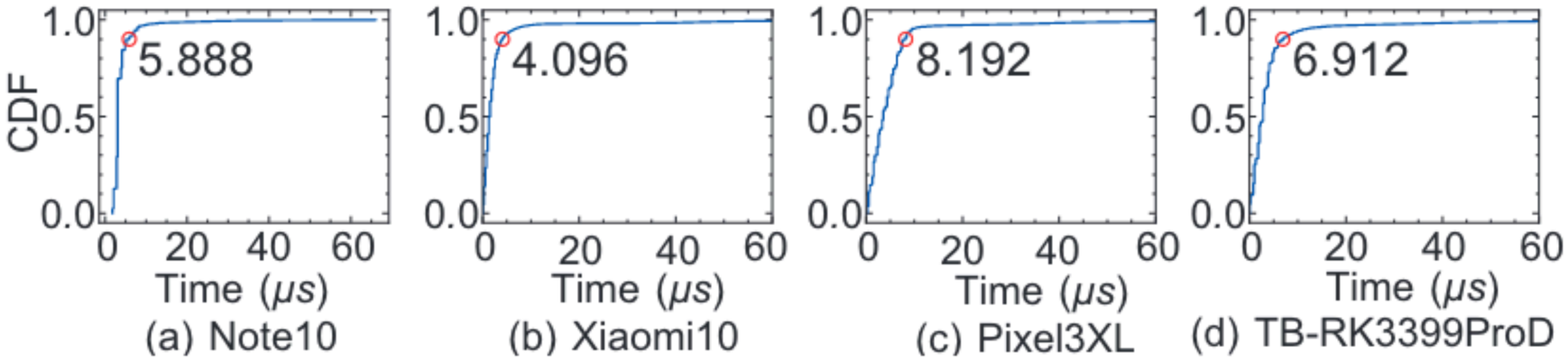


Figure 13: The CDF of per-kernel time offset after time synchronization on different platforms.

Evaluation

System overhead — latency overhead

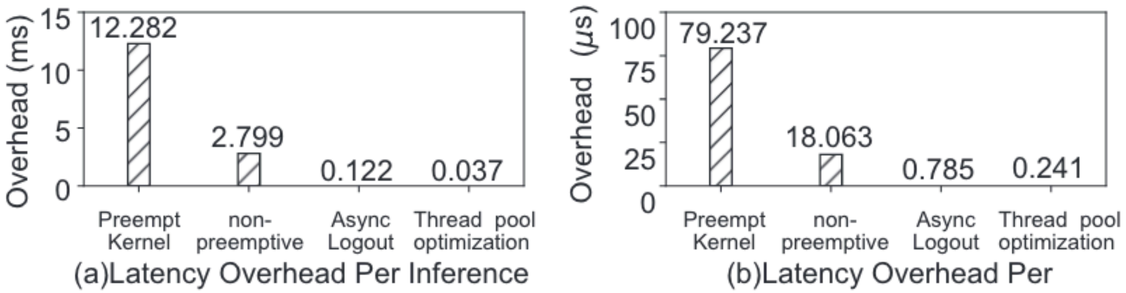


Figure 14: Extra inference latency due to nnPerf.

Table 3: Latency Overhead on different devices.

Device	Per Inference	Per Kernel	Percentage
(A) Google Pixel 3XL	0.020ms	0.201μs	**0.104**%
(B) Samsung Note10	0.020ms	0.227μs	**0.231**%
(C) Xiaomi 10	0.017ms	0.230μs	**0.116**%
(D) TB-RK3399ProD	0.037ms	0.241μs	**0.098**%

Evaluation

System overhead — time synchronization overhead

Table 4: 90-th percentile result of the time synchronization overhead under different CPU frequencies.

Device	825MHz	1056MHz	1497MHz	1804MHz
(**A**) Google Pixel 3XL	3.802μs	3.958μs	4.167μs	4.323μs
(**B**) Samsung Note10	3.646μs	3.802μs	3.542μs	3.177μs
(**C**) Xiaomi 10	4.323μs	4.167μs	3.281μs	3.698μs
(**D**) TB-RK3399ProD	3.802μs	3.958μs	4.167μs	4.323μs

Evaluation

Case study — OFA

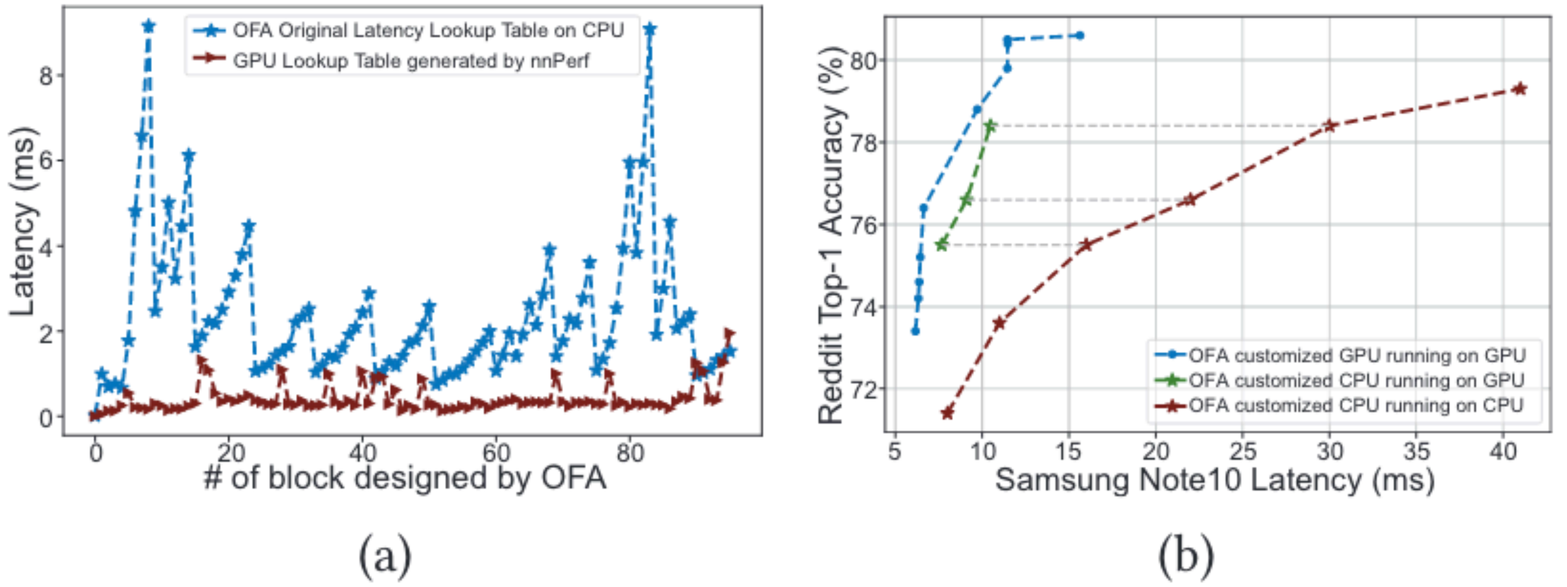


Figure 15: (a) Op-level latency table vs. kernel-level latency table. (b) Migrate OFA to kernel-granularity.

Conclusion

- **The first online, on-device** DNN model inference profiler for **e2e, op, kernel-level latency on CPU & GPU**
- Help understand model inference activity at fine-grained granularity

Thoughts

- **Profiling (dynamic) or prediction (fixed) ?**
- **MLsys'24 latency predictor (embed network & hardware) - without freq/core selection**
- **NSDI'24 LitePred - without core selection**
- **config -> transfer learning & Few-shot learning**

https://github.com/nnperfwins/nnPerf

Thanks

2024-5-24

Presented by Guangtong Li