Band: Coordinated Multi-DNN Inference on Heterogeneous Mobile Processors

MobiSys'22

Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, Byung-Gon Chun

Seoul National University

Samsung Research Funding & Incubation Center of Samsung Electronics

FriendliAI

Introduction

Enhanced computing power for mobile processors & rapid development of deep learning algorithms

Mobile applications leverage a wide variety of deep neural networks (DNNs) to solve various tasks

Poor performance under muti-DNN conditions

Face Recognition

AR Games

Introduction

Poor performance under muti-DNN conditions

- Real-time FPS requirement
- Compatibility on heterogeneous processors is terrible
- Apps require concurrent support of tasks
- Apps have various SLOs for real-time response
- TensorFlow Lite , MNN , Mace , and NCNN executing a single DNN as fast as possible

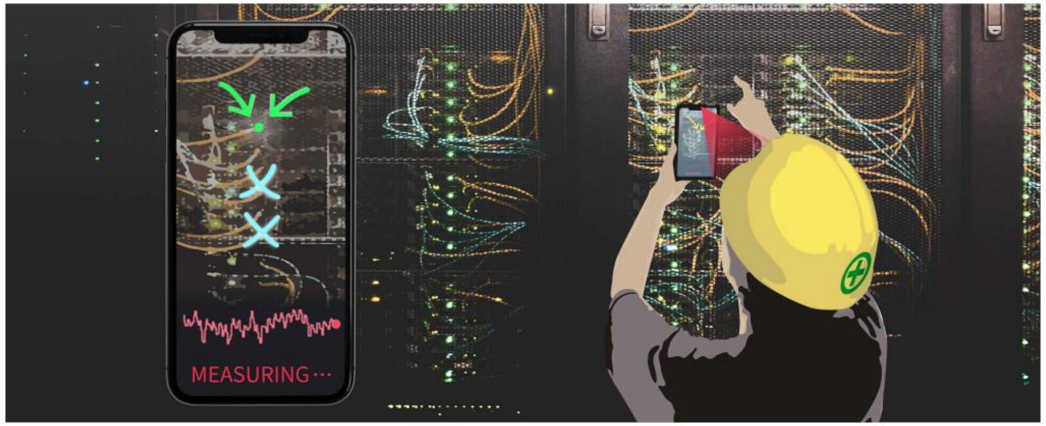


Figure 3: Assembly assistant multi-app scenario. The smartphone simultaneously runs an assembly AR app [13] with heart-rate variability sensing [27] for factory workers.

TensorFlow

MNN

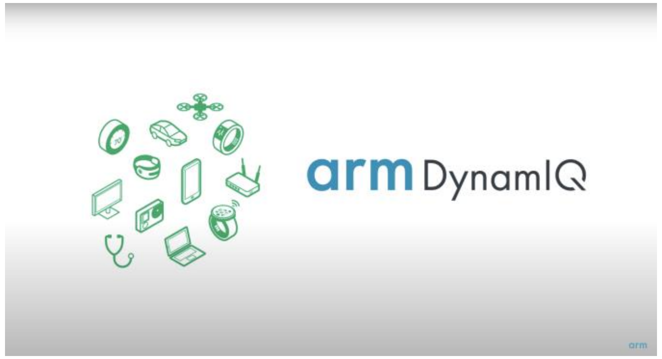
MACE
Mobile AI Compute Engine

Introduction

The heterogeneous design of mobile systemon-chips (SoCs) provides opportunity and challenge:

1）**Architecture improvement ：**brings high performance and flexibility in power and schedule

2） **Specificity of various coprocessor :** heterogeneous make it hard to Coordinate



Introduction

Muti-DNN—workload characteristics

1) Service requirement

2) Dynamic workloads

3) Multiple application

Introduction

Muti-DNN—workload characteristics

1) Service requirement

   i. Minimization of makespan
   ii. Timely responses to latency-critical tasks

ii. Oculus Quest

i. EagleEye

Introduction

Muti-DNN—workload characteristics

1) Service requirement **low latency & timely feedback**

2) Dynamic workloads **vary upon time**

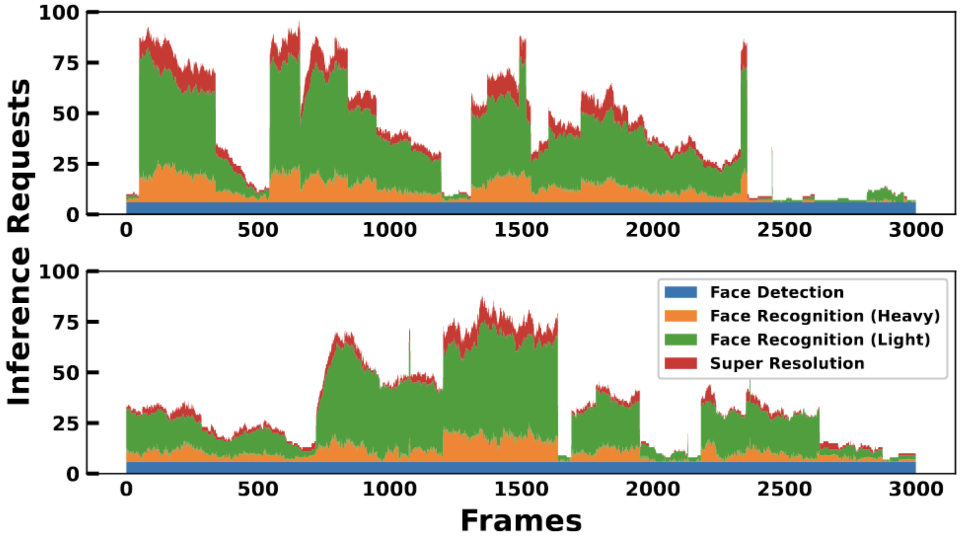


Figure 2: Workload requirements related to scene complexity. The number of required inference requests for the EagleEye Person Finder [57] scenario is shown, based on two different traces from the YouTube Faces dataset [53].

Introduction

Muti-DNN—workload characteristics

1) Service requirement **low latency & timely feedback**

2) Dynamic workloads **vary upon time**

3) Multiple application

execute different combinations of foreground and background sensing apps

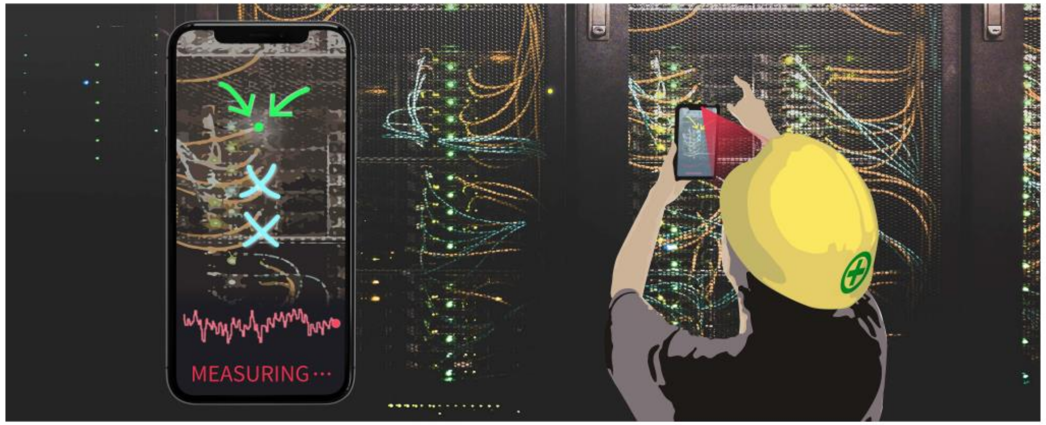


Figure 3: Assembly assistant multi-app scenario. The smartphone simultaneously runs an assembly AR app [13] with heart-rate variability sensing [27] for factory workers.

AR scenario + heart-rate variability (HRV)

Introduction

Muti-DNN—workload characteristics

1) Service requirement **low latency & timely feedback**

2) Dynamic workloads **vary upon time**

3) Multiple application **multi-application parallelism**

Introduction

Problems

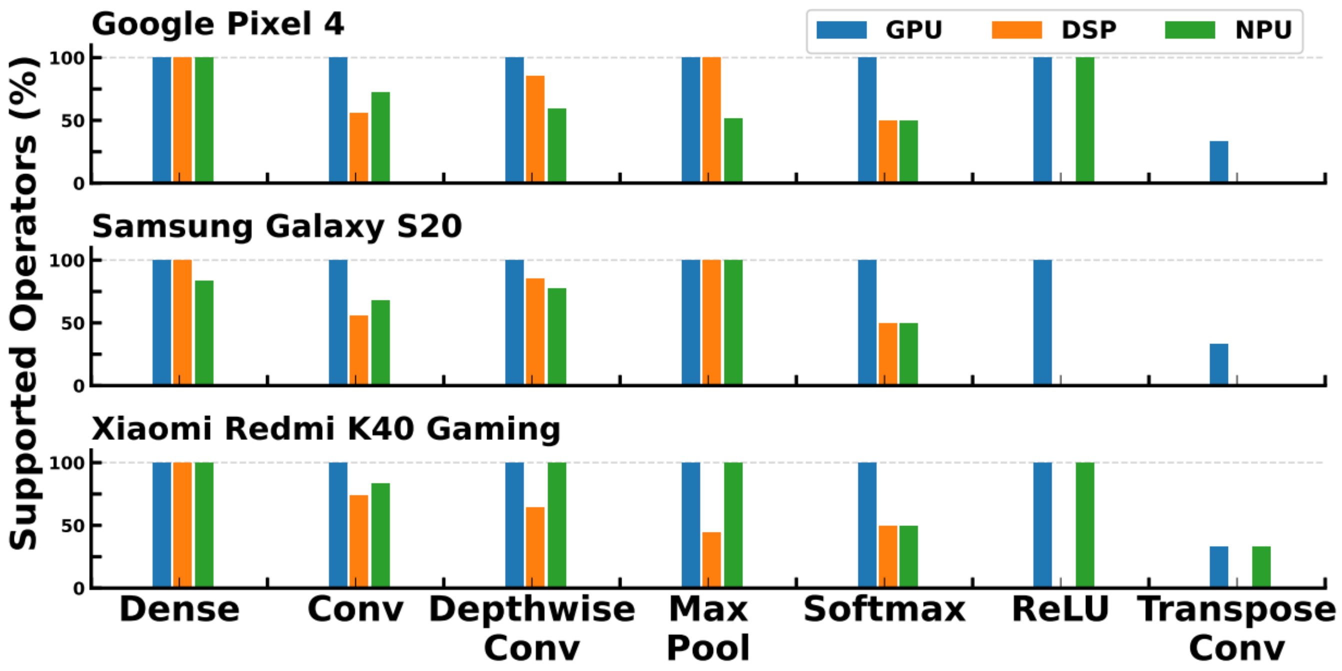
1) Frameworks designate a fast processor and **use only one processor to run a DNN**

Introduction

Problems

1) Frameworks designate a fast processor and **use only one processor to run a DNN**

2) Many DNN **operators are not fully supported** on every mobile processors



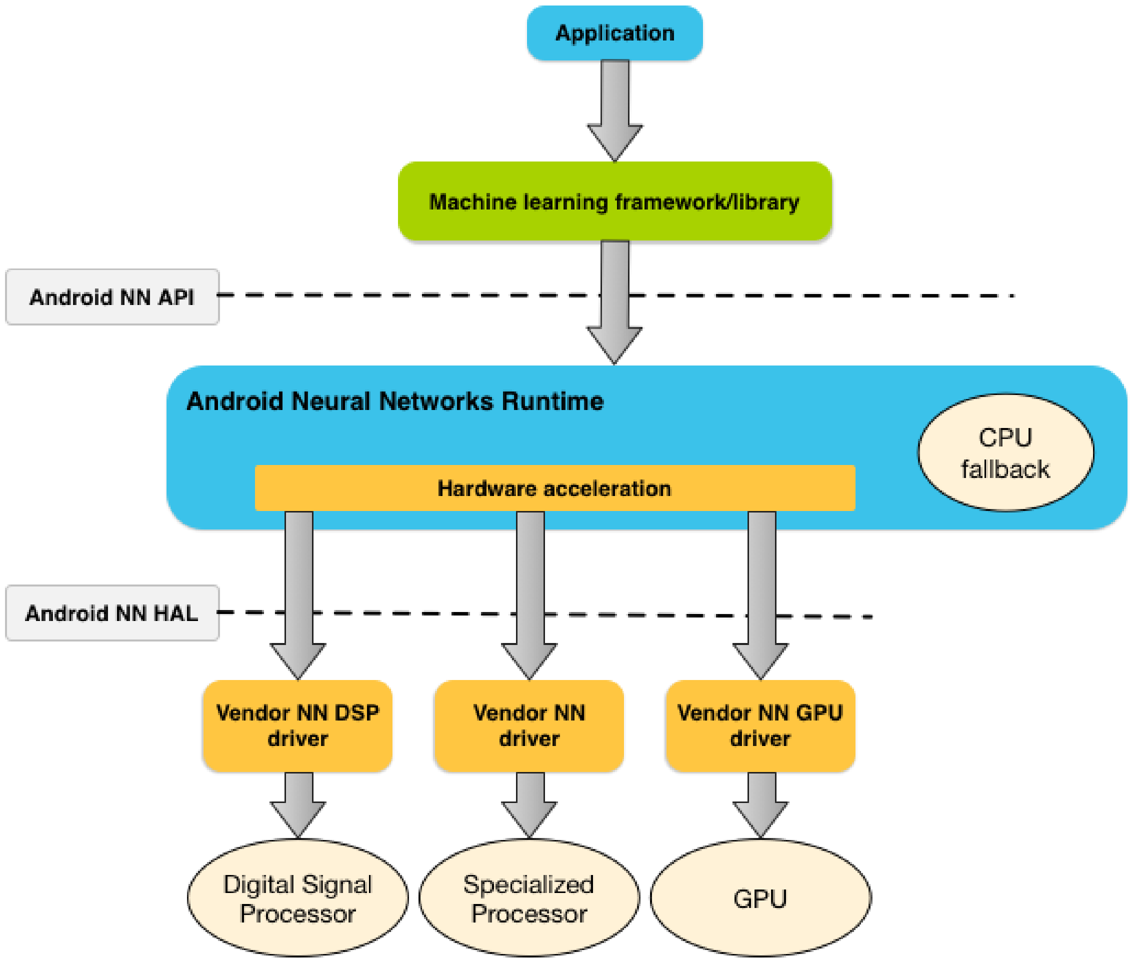
Introduction

Problems

1) Frameworks designate a fast processor and **use only one processor to run a DNN**

2) Many DNN **operators are not fully supported** on every mobile processors

3) None of previous works consider the coordination of **fallback operators**



Introduction

Problems

1) Frameworks designate a fast processor and **use only one processor to run a DNN**

2) Many DNN **operators are not fully supported** on every mobile processors

3) None of previous works consider the coordination of **fallback operators**

Challenges

1) Avoid **contention** from scheduling DNNs on the same processor (***heterogeneous processors performance*** into account)

2) Deal with **fallback operators** that are unsupported on certain processors(***occupation silently***)

3) Consider the **performance fluctuations** of mobile processors(***DVFS***)

Introduction

Problems

1) Frameworks designate a fast processor and **use only one processor to run a DNN**

2) Many DNN **operators are not fully supported** on every mobile processors

3) None of previous works consider the coordination of **fallback operators**

Challenges

1) **Processor contention**

2) **Contention from fallbacks**

3) **Uncertainties in performance**

Solutions

1) Model Partitioning

2) Dynamic Scheduling

Introduction

Why we insist **utilizing heterogeneous processors**

Frameworks mostly focus on **running** *a single DNN* **as fast as possible**, they only utilize *a specific* processor such as the GPU or NPU

↑

GPU's performance difference between mobile and pc platform is **huge**

8 Gen 1-2236 GFLOPS

RTX 3060-12.74 TFLOPS

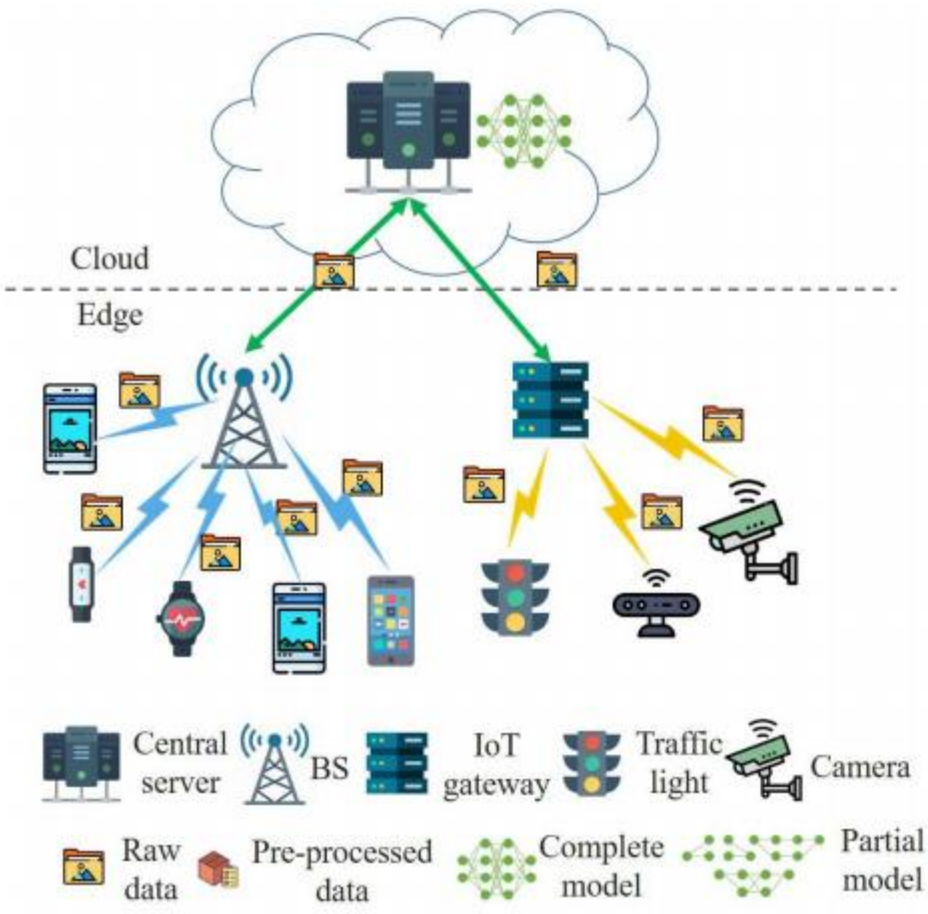
Introduction

Why we insist **utilizing heterogeneous processors**

Can't satisfy the **real-time and latency requirement** asked by **muti-DNN applications**

↑

1. **The latency of offloading** heavy computation to edge or cloud infrastructures is generally too large (**>100 ms**)
2. **Privacy** can't be Guaranteed



Edge computing

Introduction

Why we insist **utilizing heterogeneous processors**

Computing power

balance

Utilize heterogeneous processors

Latency(SLOs)

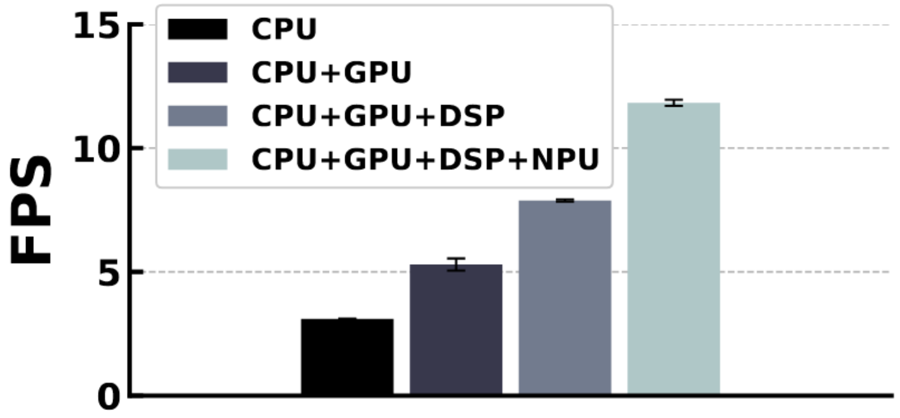


Figure 4: BAND's performance of using heterogeneous processors. The processed frames per second (FPS) rise as more processors are used. Results are from the EagleEye [57] workload, on Google Pixel 4.

Introduction

Challenges

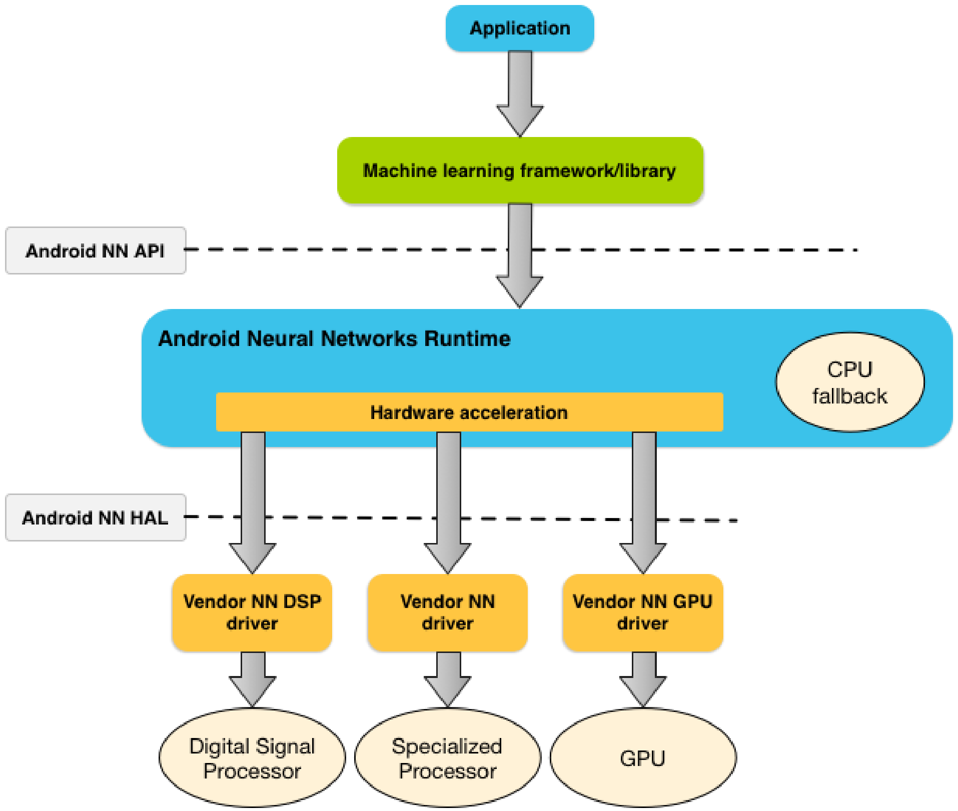
1) **Processor contention**

2) **Contention from fallbacks**

3) **Uncertainties in performance**

Introduction

Challenge 1 Processor contention



Limited number of processor cores and memory bandwidth makes it more likely appear process contention **compared with PC platform**

SDKs for mobile processors like **NNAPI** provide interfaces for **concurrent execution**， but require driver adaptation

Introduction

Challenge 1 Processor contention

Processor (Mobile Device)	The Number of Concurrent Models 1	2	4
Google Edge TPU (Google Pixel 4)	**25.29** ± 0.79	**35.86** ± 10.23	**56.94** ± 22.55
Hexagon DSP (Google Pixel 4)	**25.43** ± 0.55	**37.34** ± 11.41	**61.32** ± 25.08
Qualcomm HTA (Samsung Galaxy S20)	**23.98** ± 1.95	**24.08** ± 2.68	**33.66** ± 10.47
Adreno 650 GPU (Samsung Galaxy S20)	**115.52** ± 0.97	**228.33** ± 3.16	**448.34** ± 7.47
MediaTek APU 3.0 (Xiaomi Redmi K40 Gaming)	**20.34** ± 0.19	**21.35** ± 0.24	**31.61** ± 9.78
Mali-G77 GPU (Xiaomi Redmi K40 Gaming)	**133.36** ± 2.22	**255.49** ± 5.52	**477.91** ± 37.08
Huawei NPU (Huawei Mate 40 Pro)	**10.15** ± 0.14	**14.92** ± 4.26	**23.53** ± 9.57

Table 1: Inference latency variation from concurrent inferences. The mean latency and standard deviation (ms, per model) of running InceptionV4 on various processors are shown, for a varying number of concurrent inferences.

SDKs for mobile processors like **NNAPI** provide interfaces for **concurrent execution**， but require driver adaptation

1. **GPU**’s performance is disappointing as doesn’t benefit from concurrent execution
2. **Other accelerator**(APU) has good performance only when number of models is small

Introduction

Challenge 2 Contention from fallbacks

Each mobile platform(SoC) is equipped with a **unique combination of various processors**

Heterogeneity upon support of operators

GPU's Compatibility is better compared with other coprocessors

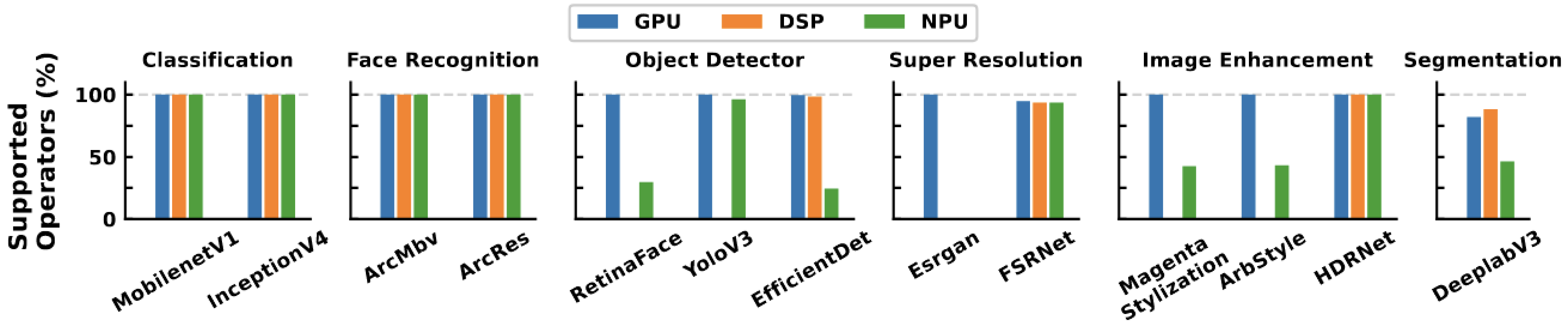


Figure 5: Fallbacks commonly occur across various tasks. Percentage of supported operators on Google Pixel 4 is shown.

Introduction

Challenge 2 Contention from fallbacks

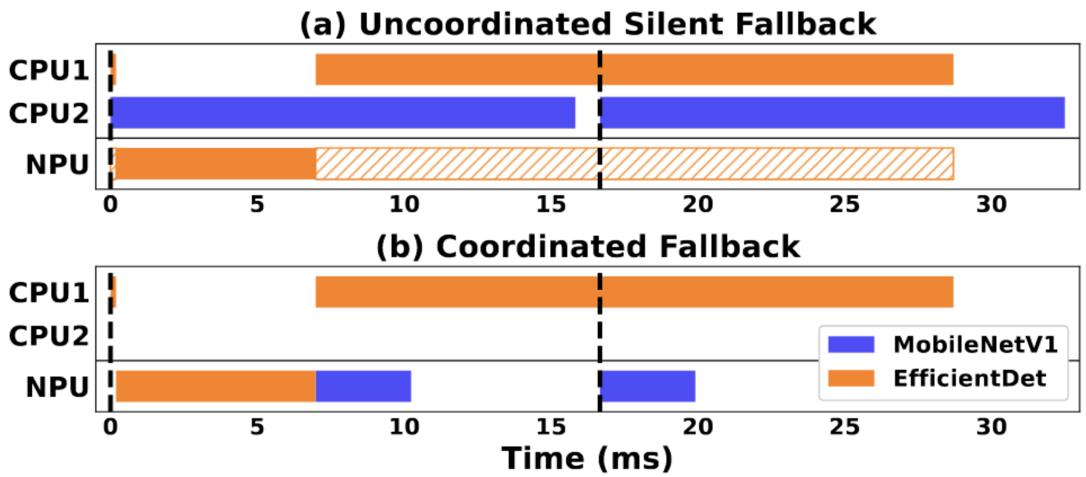


Figure 6: Schedulability limitations of uncoordinated fallbacks. Execution timeline of MobileNetV1 and EfficientDet on the CPU and NPU (Google Pixel 4's Edge TPU). The fallback NPU region of EfficientDet (hatched area) can be used by MobileNetV1 to run faster than on the CPU.

Existing frameworks are only capable of utilizing **a single processor** at a time silently fall back to the **CPU(always)** when running **unsupported operators(fallbacks)**

1. Uncoordinated fallback blocks other operators from accessing an **idle** processor
2. Other **non-CPU accelerators** were not considered as **an option to process the fallback operators**

Introduction

Challenge 3 Uncertainties in performance → Latency estimation is harder

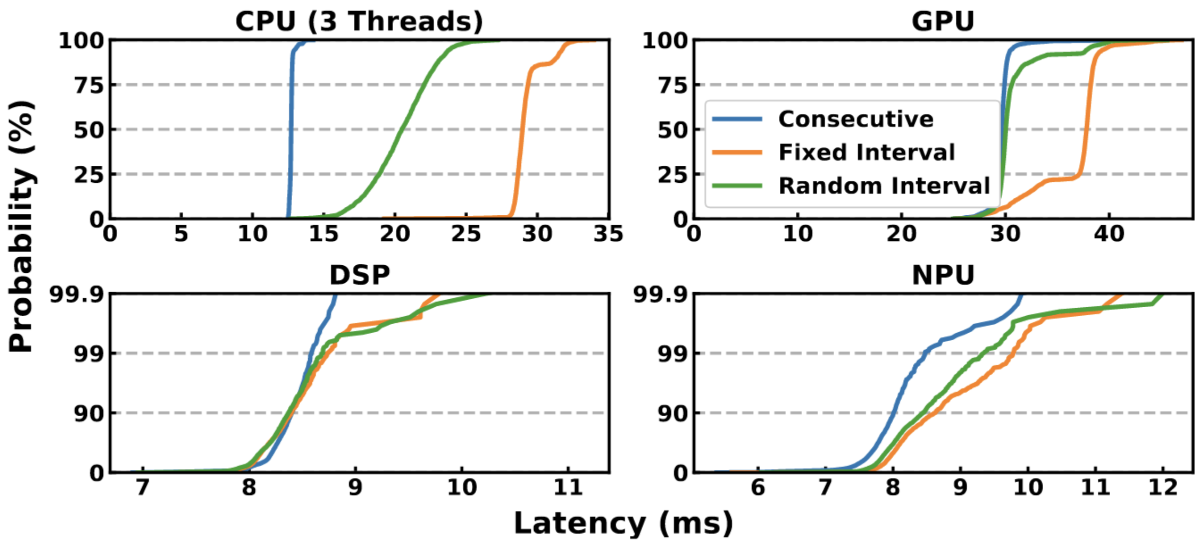


Figure 7: Performance variation of mobile processors. The inference latency of the ArcFace-ResNet50 model on a Google Pixel 4 device is shown as a CDF, with various intervals between inferences.

DVFS mechanisms： balance the power consumption and performance (**BIG.little -> DynamIQ**)
CPU & GPU's **automatic frequency**

Mobile accelerators of separate hardware are **loosely coupled with the main chips**
DSP & NPU's **single-frequency**

BAND Overview

BAND stemed from a key findings.

Fine-grained execution of DNN models can increase the schedulability of non-preemptive heterogeneous processors

CoDL intergrates two techniques

1) Model analysis based on **subgraph partitioning**
2) Fine-grained **subgraph scheduling** based on non-preemptive processors

BAND Overview

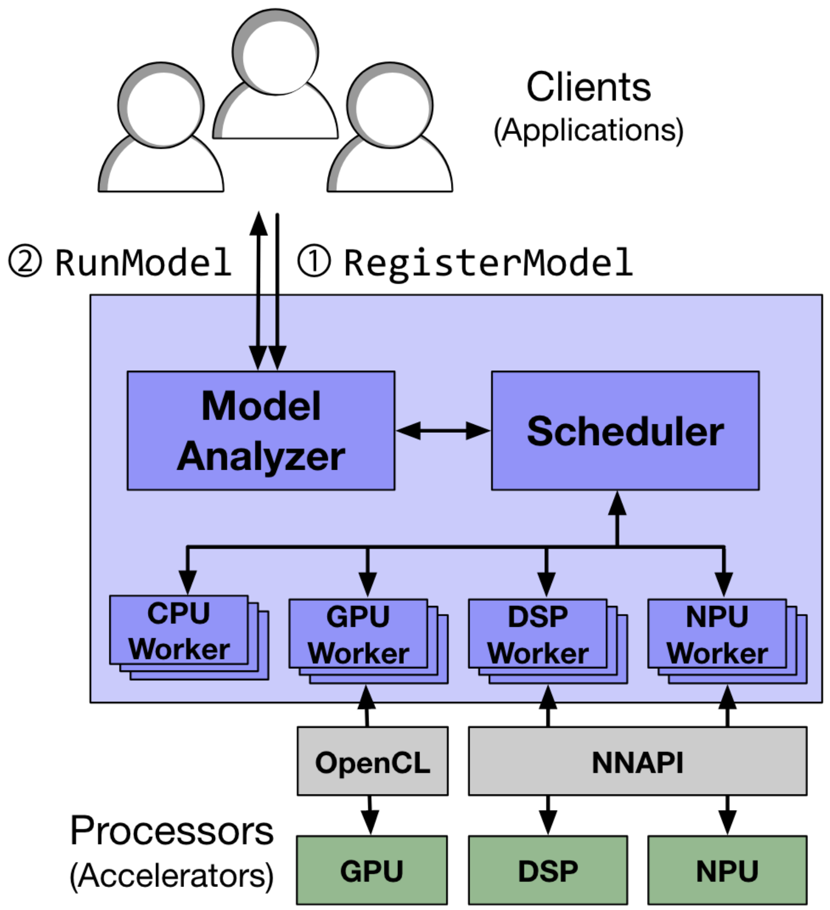


Figure 8: BAND system architecture.

Composition

- A model analyzer
- A central scheduler
- Per-processor workers

BAND Overview

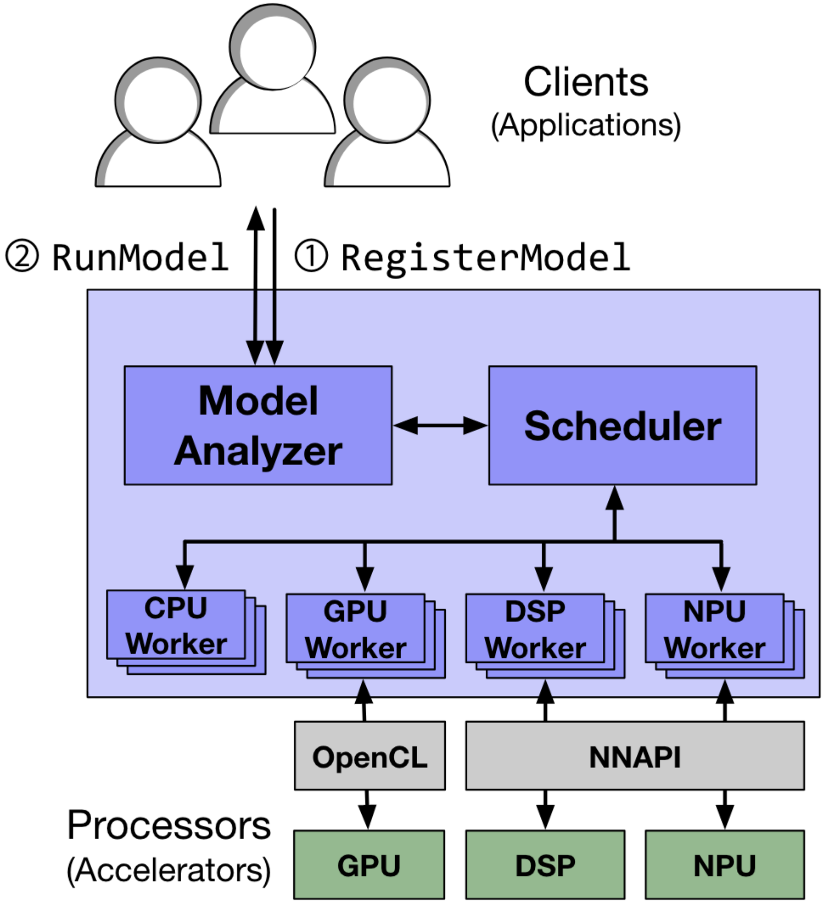


Figure 8: BAND system architecture.

Composition

- **A model analyzer**

  Partitions models into **subgraphs**
- A central scheduler
- Per-processor workers

BAND Overview

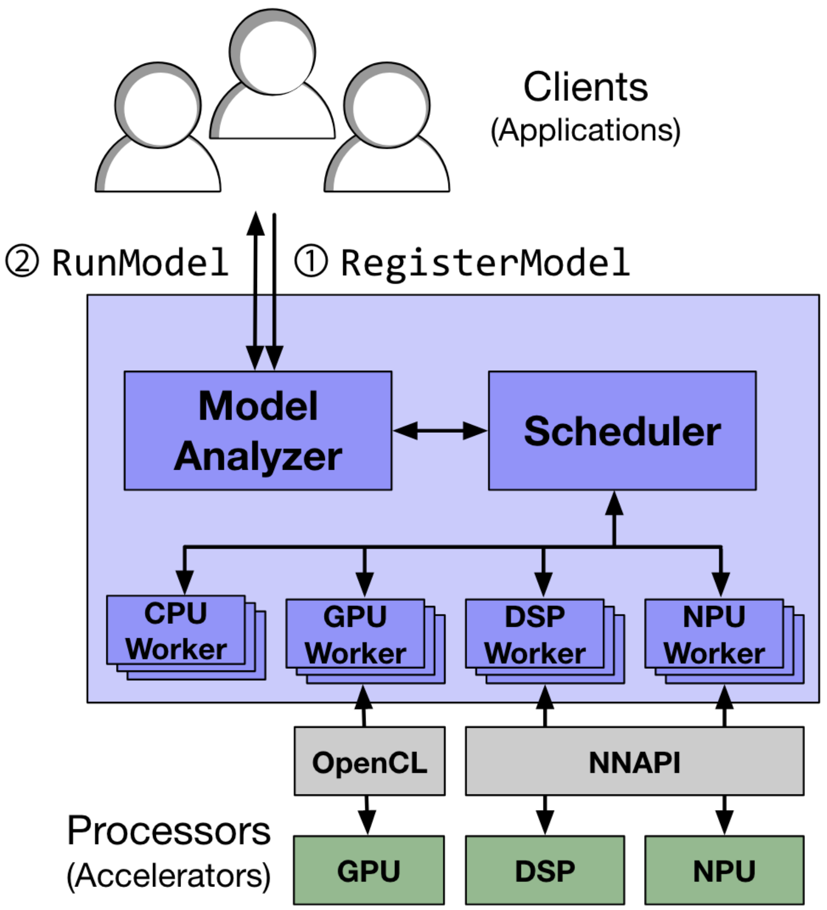


Figure 8: BAND system architecture.

Composition

- A model analyzer
- **A central scheduler**

  decides **which subgraphs** to run on **which workers**
- Per-processor workers

BAND Overview

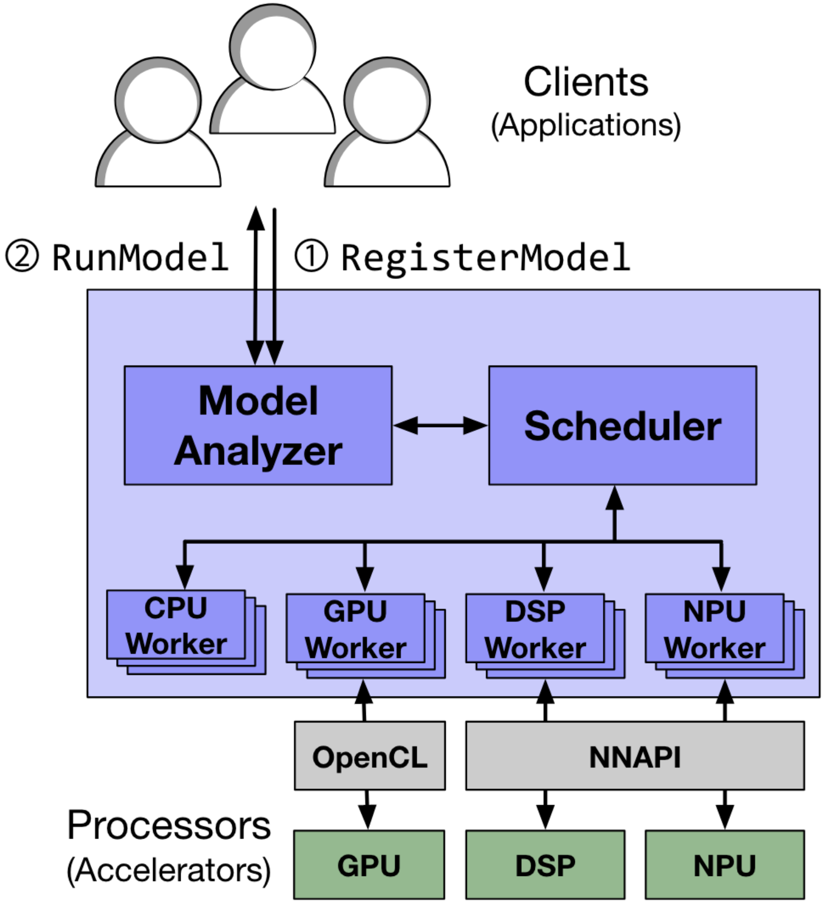


Figure 8: BAND system architecture.

Composition

- A model analyzer
- A central scheduler
- **Per-processor workers**

  1.execute the subgraphs on their respective processors

  2.processor-work thread： one to many correspondence

BAND Overview

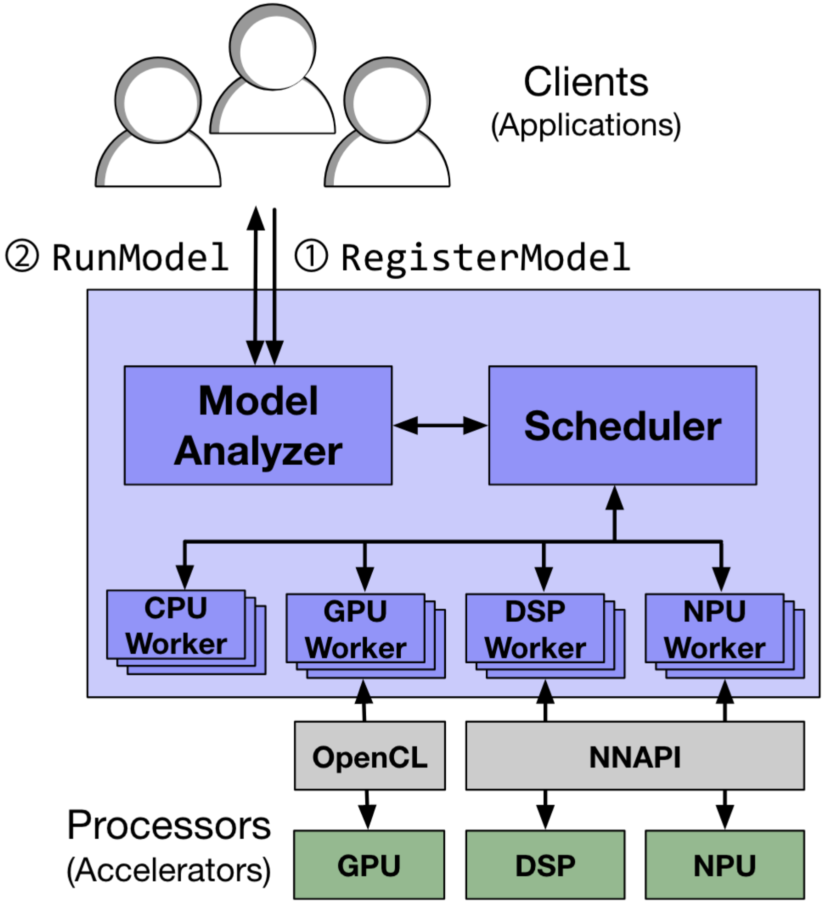


Figure 8: BAND system architecture.

Composition

- **A model analyzer**
- **A central scheduler**
- Per-processor workers

Model analyzer based on **subgraph partitioning**

The model analyzer's work : examines a registered model and **creates specific subgraphs**.

Benefit: **more possible schedules** can be considered at runtime

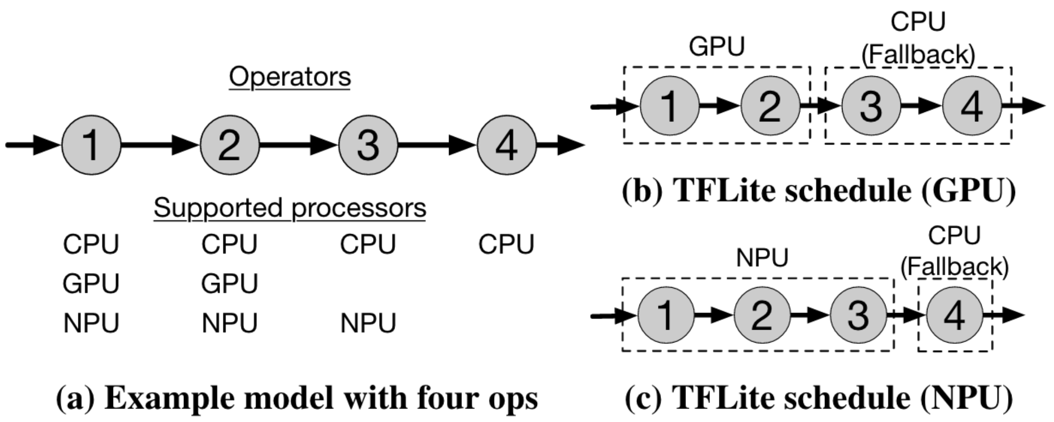


Figure 9: An example model with varying operator support. For a given processor, TensorFlow Lite only creates a single execution schedule.

All operators are always supported by the **CPU**

Model analyzer based on **subgraph partitioning**

The model analyzer's work : examines a registered model and **creates specific subgraphs**.

Benefit: **more possible schedules** can be considered at runtime

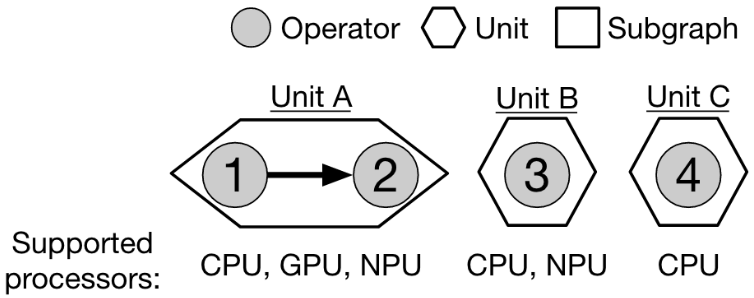
operator-level scheduling
(most fine-grained)

subgraph-level scheduling
(less fine-grained)

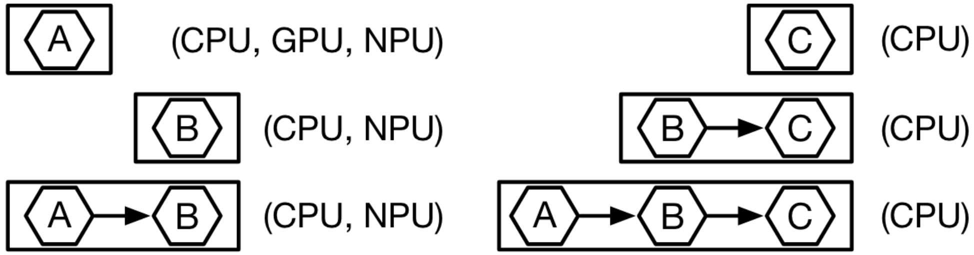
model-level scheduling
(most coarse-grained)

Model analyzer based on subgraph partitioning

1. Create units and subgraphs



(a) Units



(b) Subgraphs

Figure 10: Grouping operators into units and subgraphs.

Step 1 . Generate a list of **units** from the operators (Figure 10a)

Step 2 . Generate **subgraphs** from these units (Figure 10b)

Model analyzer based on **subgraph partitioning**

2. Subgraph usage and memory

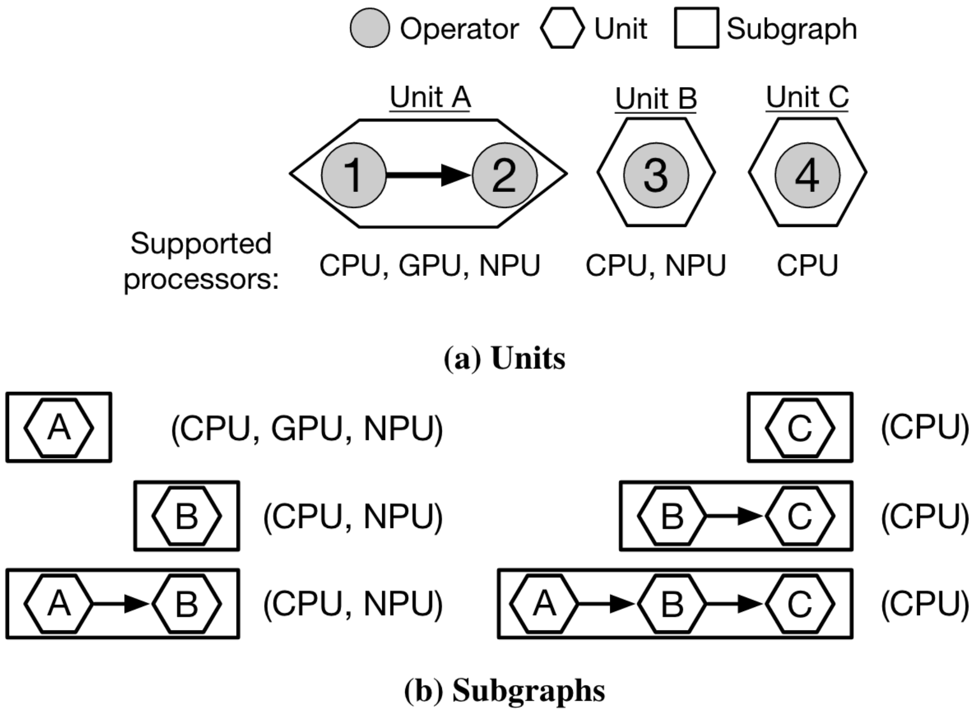
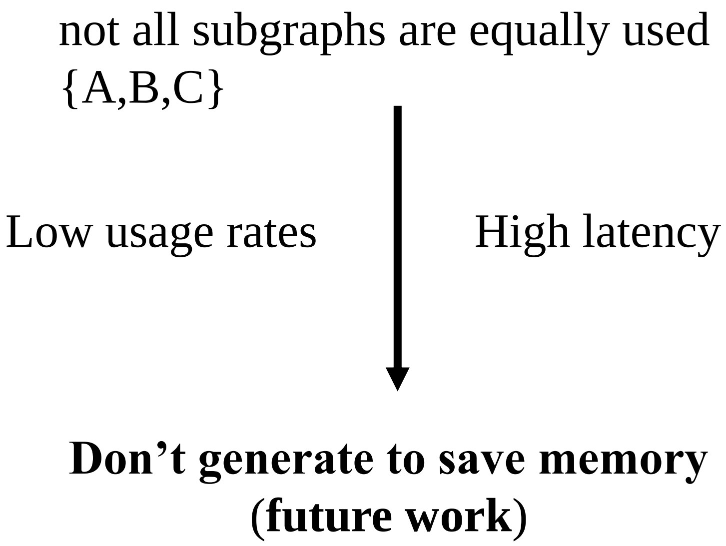


Figure 10: Grouping operators into units and subgraphs.



Central scheduler based on **Subgraph scheduling**

The central scheduler's work :

- examines the executable subgraphs from the units
- selects which subgraph to run
- selects the processor to run the subgraph

Benefit:

- **make full use of the computing resource**
- **Fit uncertainties in performance**

Central scheduler based on **Subgraph scheduling**

1. Scheduling

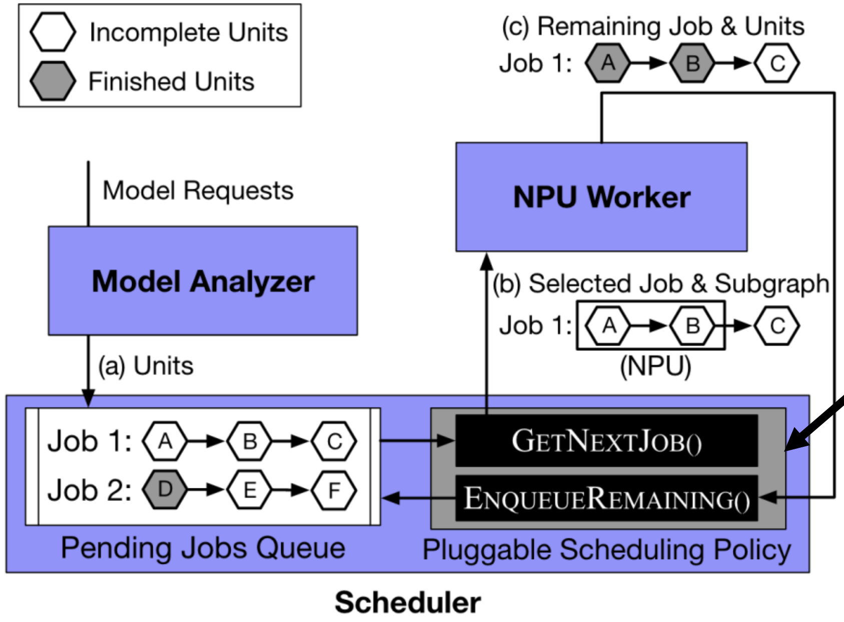


Figure 11: Detailed workflow of Band's scheduler. (a) The scheduler spawns a job for each inference request, with units provided by the model analyzer. (b) The jobs are enqueued into the job queue, and the scheduling policy checks the queue to select the next job to process. The policy also chooses the subgraph and the processor to run. (c) Afterwards, the job's unit execution status is updated, and the job is put back into the job queue if there are any remaining units. The enqueue position of the updated job is determined by the policy.

Scheduling policy is a pluggable component

Default : least slack time (LST)

- Selecting a job from the job queue
- Scheduling the remaining units of a job
- Handling processors with the thermal shutdown

Central scheduler based on **Subgraph scheduling**

1. Scheduling

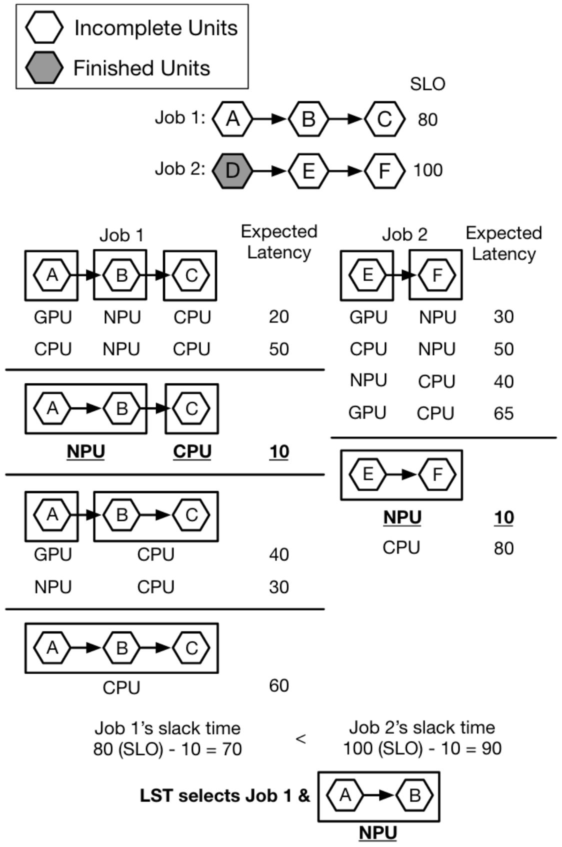


Figure 12: The Least Slack Time (LST) policy. The policy selects the job with the least slack, which is calculated by subtracting the expected latency from the SLO. The latency of the shortest subgraph sequence is regarded as that job's latency, and the first subgraph of that sequence is returned.

- **Selecting a job from the job queue**

  Each scheduling policy has its own unique logic for selecting a job and its subgraph (Figure 11b)
- Scheduling the remaining units of a job
- Handling processors with the thermal shutdown

for **n** subgraphs with **p** compatible processors each

$O(p^n)$ (for a single job) $\longrightarrow$ $O(pn^2)$

LST policy

Central scheduler based on **Subgraph scheduling**

1. Scheduling

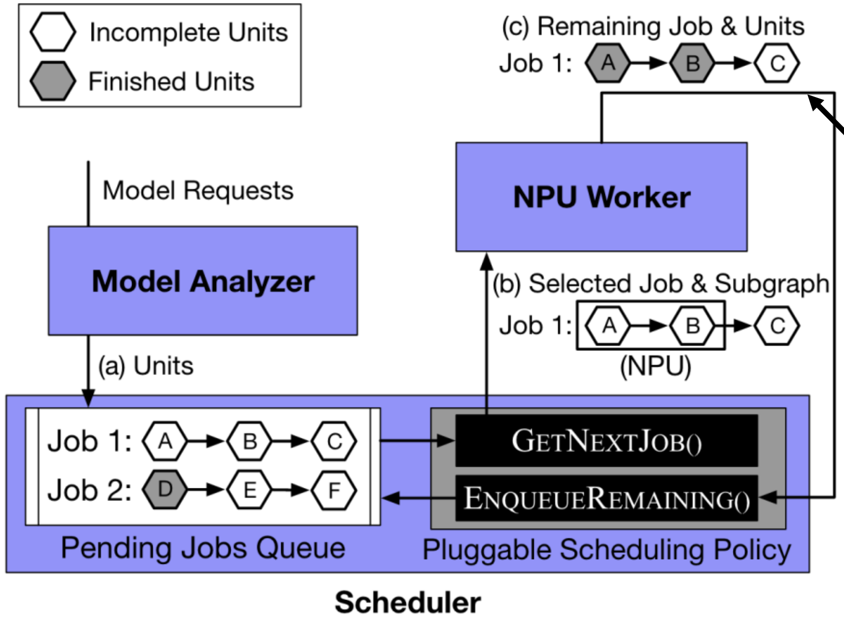


Figure 11: Detailed workflow of BAND's scheduler. (a) The scheduler spawns a job for each inference request, with units provided by the model analyzer. (b) The jobs are enqueued into the job queue, and the scheduling policy checks the queue to select the next job to process. The policy also chooses the subgraph and the processor to run. (c) Afterwards, the job's unit execution status is updated, and the job is put back into the job queue if there are any remaining units. The enqueue position of the updated job is determined by the policy.

- Selecting a job from the job queue
- **Scheduling the remaining units of a job**
- Handling processors with the thermal shutdown

Depending on schedule policy

Central scheduler based on **Subgraph scheduling**

1. Scheduling

- Selecting a job from the job queue
- Scheduling the remaining units of a job
- **Handling processors with the thermal shutdown**

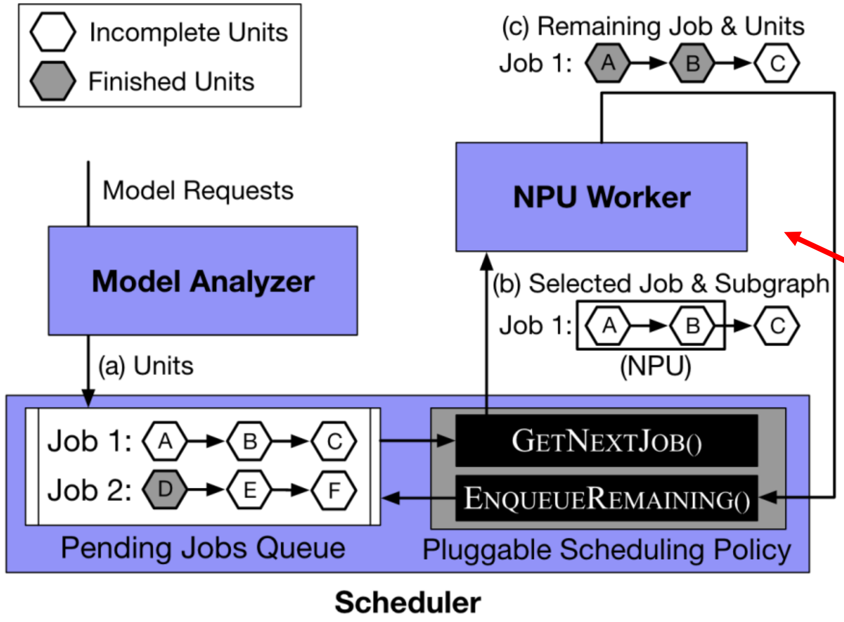


Figure 11: Detailed workflow of BAND's scheduler. (a) The scheduler spawns a job for each inference request, with units provided by the model analyzer. (b) The jobs are enqueued into the job queue, and the scheduling policy checks the queue to select the next job to process. The policy also chooses the subgraph and the processor to run. (c) Afterwards, the job's unit execution status is updated, and the job is put back into the job queue if there are any remaining units. The enqueue position of the updated job is determined by the policy.

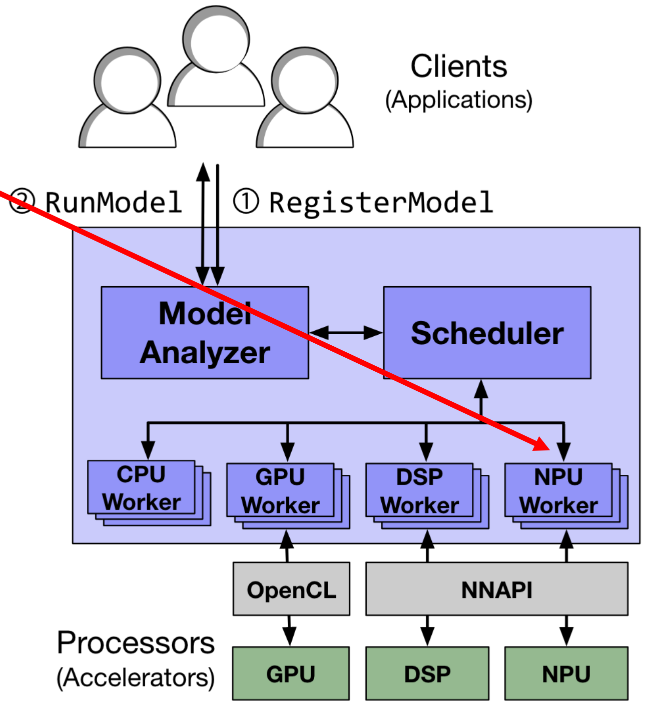
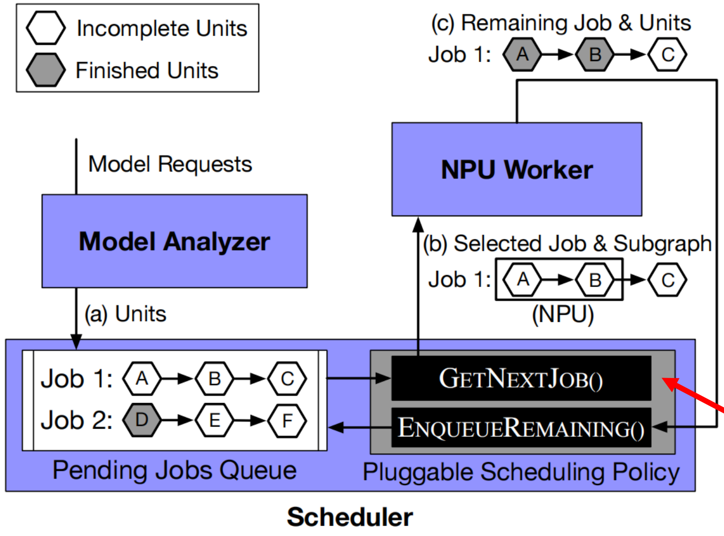


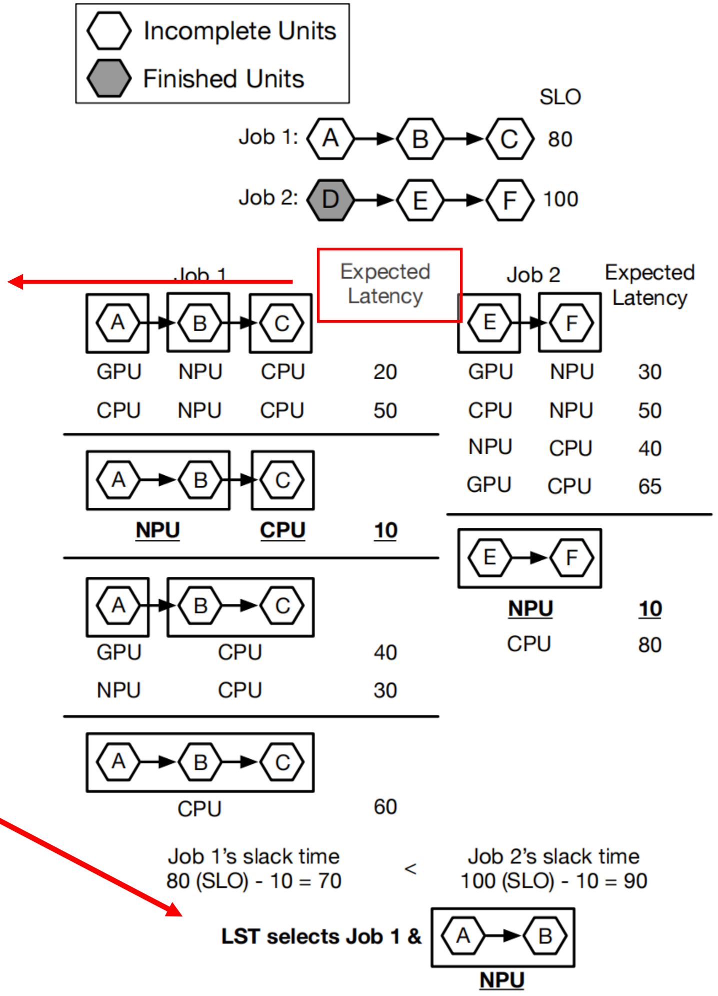
Figure 8: BAND system architecture.

Central scheduler based on **Subgraph scheduling**

1. Execution Time Profiles



Affected by DVFS



Central scheduler based on **Subgraph scheduling**

1. Execution Time Profiles

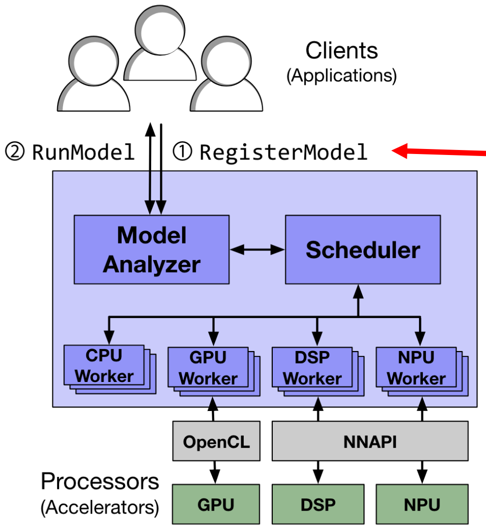
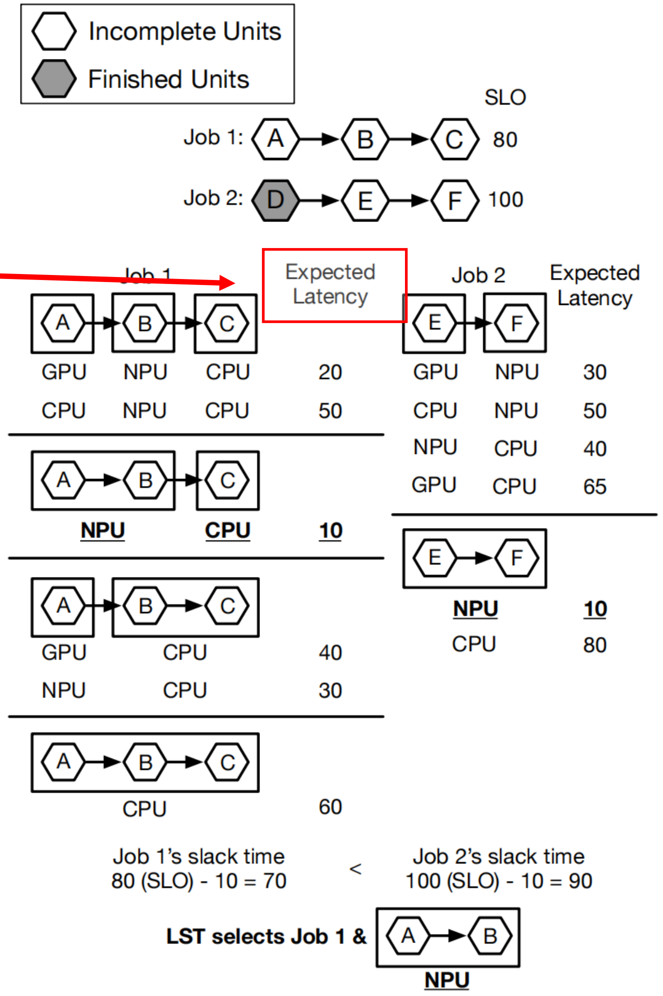


Figure 8: BAND system architecture.



- runs the model a few times to retrieve **baseline execution time values**,
- **estimates** the execution **times of** the model's **subgraphs** based on the baseline execution time.
- constantly make **online adjustments** to reflect the current workload pattern

Central scheduler based on **Subgraph scheduling**

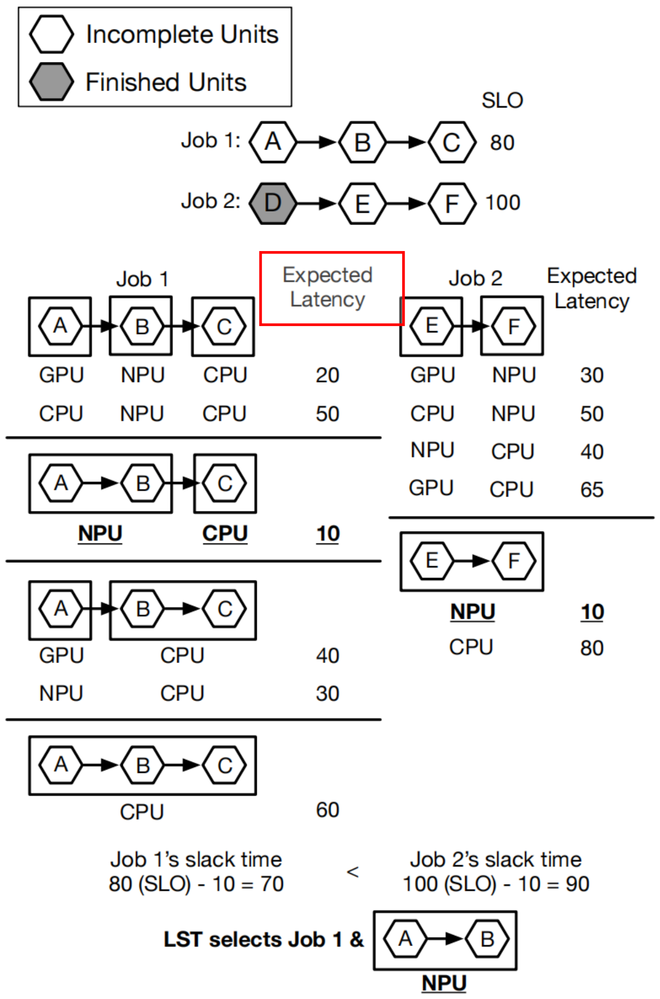
1. Execution Time Profiles

Assuming the execution time of a subgraph is roughly **proportional to**

$$FLOPs + \beta \times tensor\_size,$$

$$\beta = \begin{cases} \frac{1{,}000}{bytes} & \text{For accelerators} \\ \frac{10}{bytes} & \text{For CPUs} \end{cases}$$

$$time_{profiled} \leftarrow \alpha\, time_{new} + (1 - \alpha)\, time_{profiled}$$



Evaluation

Setup

Google Pixel 4（(Qualcomm Snapdragon 855 + Google Edge TPU）

Xiaomi Redmi K40 Gaming（MediaTek Dimensity 1200）

Samsung Galaxy S20（Qualcomm Snapdragon 865）

Baseline： TensorFlow Lite 2.3.0

Hardware & Software

Evaluation

Single-App: Back-to-Back Inference

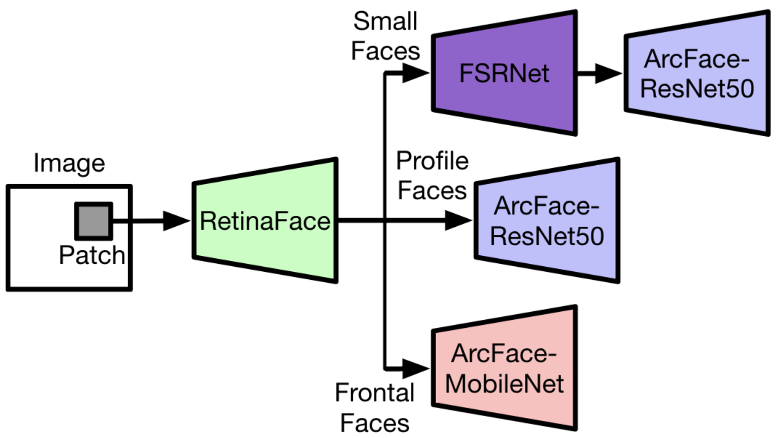
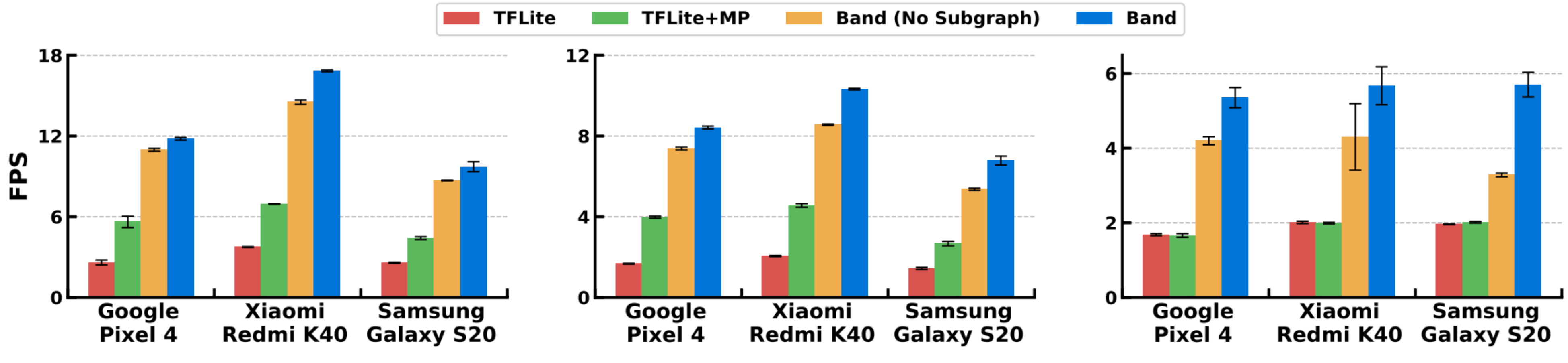


Figure 13: The EagleEye [57] workflow. After running the RetinaFace [18] face detection model on a patch of the input image, a different DNN is run on the faces of that same patch depending on the detected face types. The exact number of patches and faces are different for each frame.

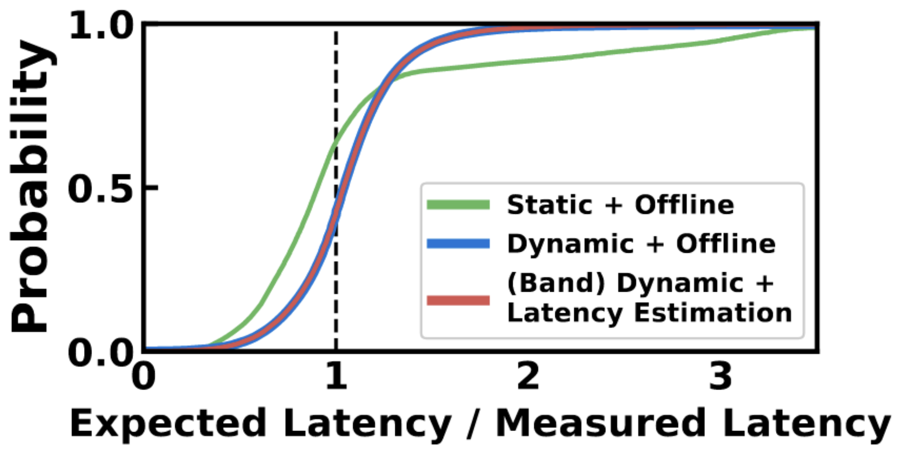


(a) EagleEye (b) EagleEye (Crowded) (c) Distream

Figure 14: Processed FPS for varying workloads on various mobile devices.

Evaluation

Analysis on Profiling



(a) CDF of the ratio of expected to measured latency

	Static + Offline	Dynamic + Offline	BAND + Noise ± 30%	BAND
FPS	9.12 ± 0.11	9.46 ± 0.09	9.31 ± 0.13	9.46 ± 0.08
Profiling time (s)	76	76	4.8	4.8

(b) Processed FPS and profiling time-varying profiling methods

Figure 15: Effect of different profiling methods. FPS and latencies are measured by running the EagleEye workload 5 times with Samsung Galaxy S20.

Evaluation

In-depth Analysis of Subgraphs—Single Model Scalability

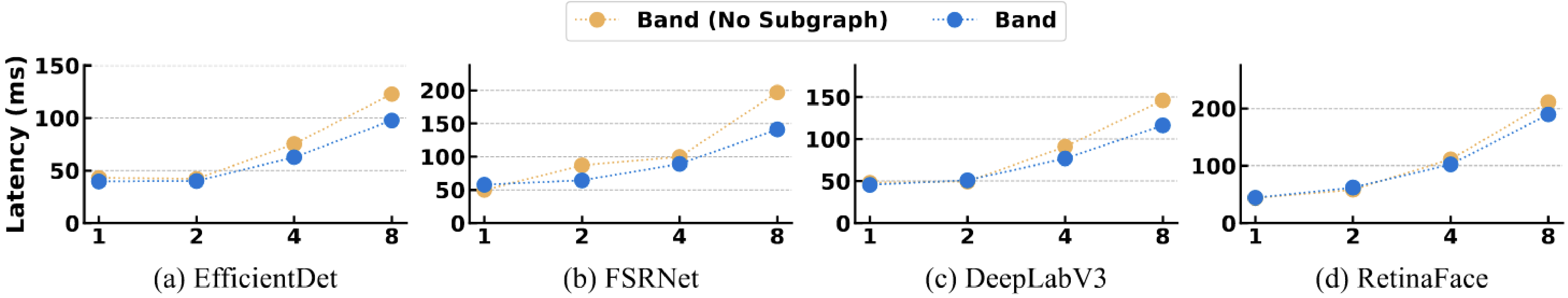
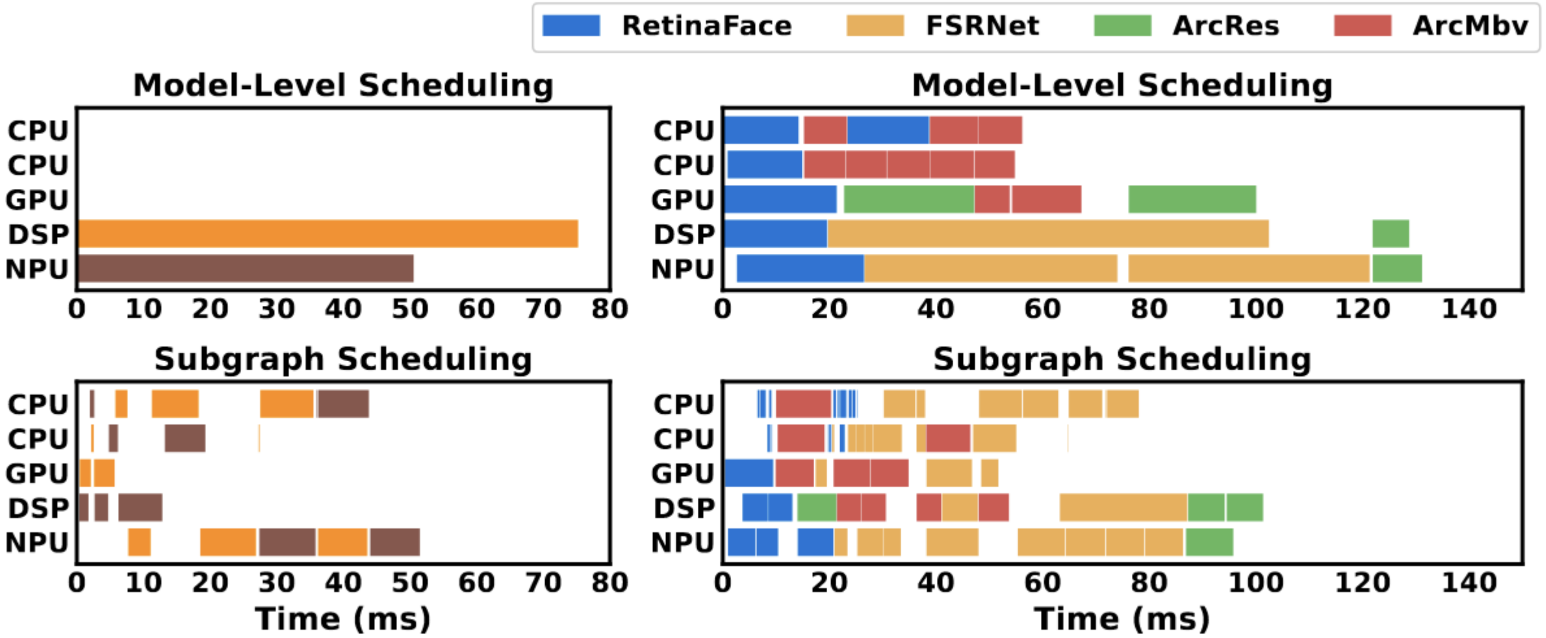


Figure 16: Frame latencies of running multiple instances of a model. For models with largely varying subgraph execution times such as EfficientDet and FSRNet, the gap between BAND and BAND (No Subgraph) is significant.

Evaluation

In-depth Analysis of Subgraphs— Timeline Analysis



(a) Single frame with 2×FSRNet (b) Single frame of EagleEye workload

Figure 17: Subgraph scheduling timelines. Timelines where subgraph scheduling finds better schedules than model-level scheduling are shown. Both timelines are measured on Google Pixel 4.

Evaluation

In-depth Analysis of Subgraphs— Power consumption

	TFLite+MP	BAND
Power (W)	7.60	**7.99**
FPS	4.11	**8.71**

Table 2: Average power consumption in Google Pixel 4 while processing the EagleEye (Crowded) workflow. Power consumption was measured using a Monsoon Power Monitor.

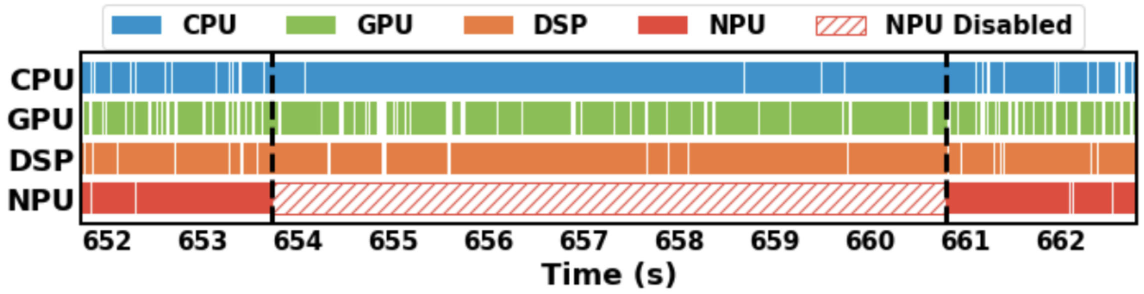


Figure 19: EagleEye scheduling timeline on Google Pixel 4 before and after the NPU becomes unavailable due to throttling.

BAND Overview

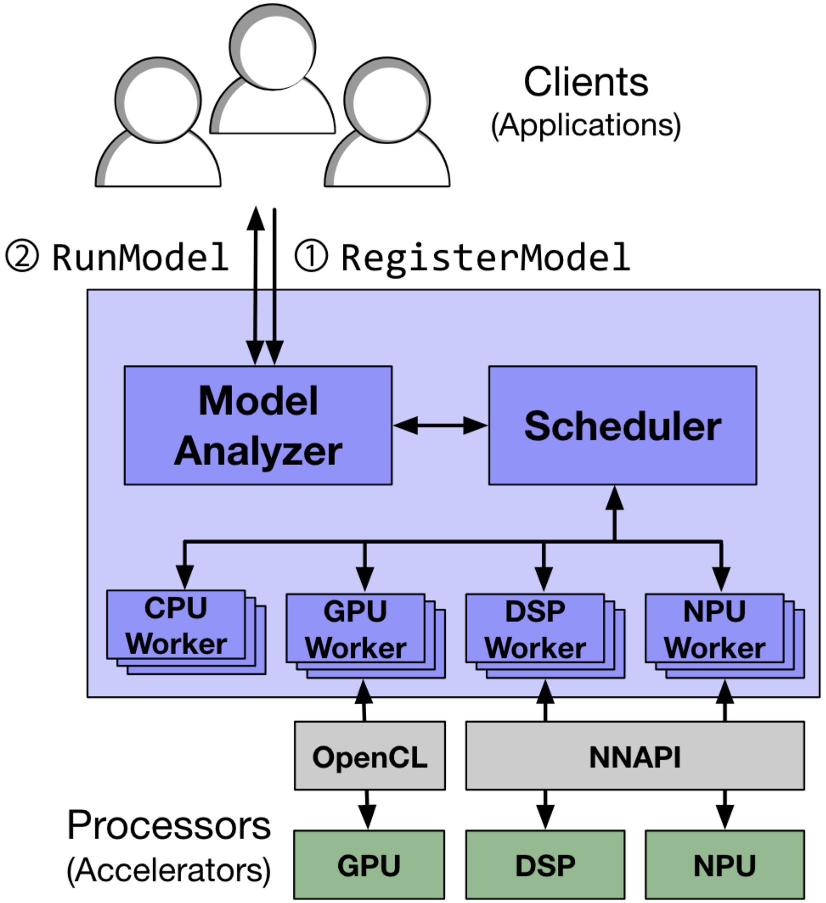


Figure 8: BAND system architecture.

Composition

- A model analyzer
- A central scheduler
- Per-processor workers

Conclusion

Advantages:

1) online adjustments method performance fluctuations caused by DVFS
2) make full use of heterogeneous computing resource by scheduling fallback operators
3) Initialization of new model is faster as system doesn't run all of the subgraphs but Proportional split
4) subgraph-level scheduling (less fine-grained)
5) achieve flexibility in schedule with non-preemptive scheduling method

Disadvantages:

1) solutions for linear smoothing function is too simple, lack of Robustness
2) Only in heavy workloads the BAND performance can be better than model-level scheduling
3) Model analyzer : not all subgraphs are equally used, therefore lead to invalid memory occupation

Thanks

2022-11-28

Presented by Guangtong Li