FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices

Xiangyu Li[1], Yuanchun Li[1,2], Yuanzhe Li[1], Ting Cao[3], Yunxin Liu[1,2]

[1]Institute for AI Industry Research (AIR), Tsinghua University

[2]Shanghai Artificial Intelligence Laboratory [3]Microsoft Research

Edge AI is transforming our lives

The capability of AI has been demonstrated on various tasks.

Edge AI is taking an important role, but its capability is limited.

- **Local DNN inference:** network-free, cost-efficient, privacy-preserving.
- **Challenge:** **resource constraints** on edge devices.

Autonomous Vehicle

Smartphone

Edge Camera

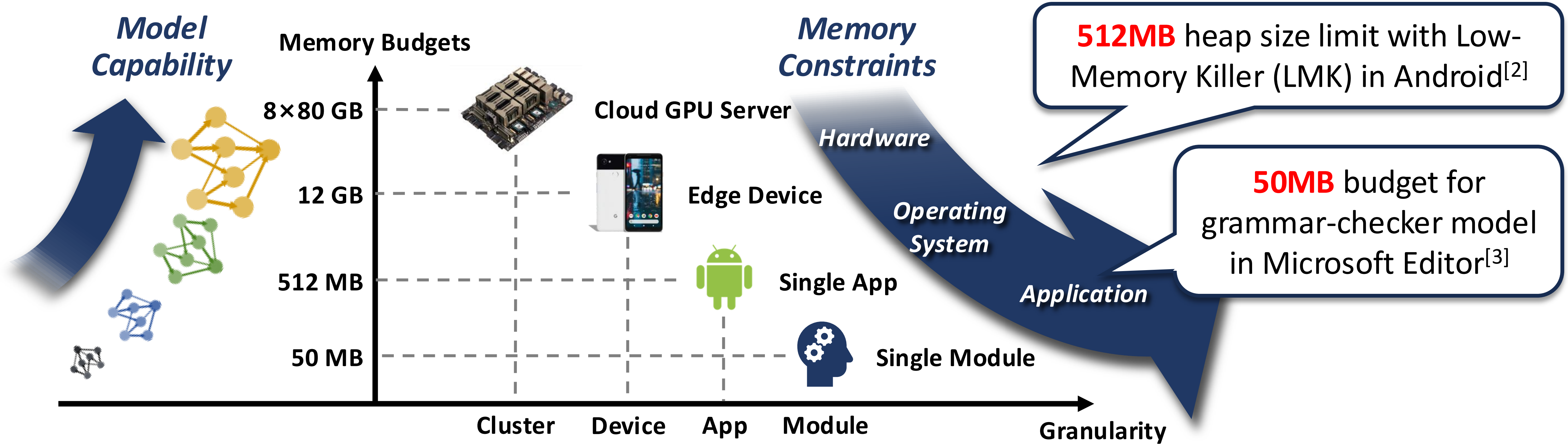
Pictures from Internet.

Memory bottleneck of on-device DNN inference

The improvement of memory has greatly lagged behind computation.

- From iPhone 4 to iPhone 14[1]: GPU (1000×) > CPU (100×) >> RAM (8×)

There are multiple levels of memory constraints besides hardware.



[1] GadgetVersus. https://gadgetversus.com/smartphone/apple-iphone-4-vs-appleiphone-14/, accessed: 2023-08-01.
[2] Android Developers. https://developer.android.com/topic/performance/memory, accessed: 2023-08-01.
[3] Ge Tao, et al. https://www.microsoft.com/en-us/research/blog/achievingzero-cogs-with-microsoft-editor-neural-grammar-checker/, accessed: 2023-08-11.

Existing approaches for memory reduction

Model Customization

- Model compression (*e.g.,* quantization, pruning, …)
- Efficient structure design (*e.g.,* MobileNets[1])
- Neural Architecture Search

deployment efforts ↑
model accuracy ↓

orthogonal

System Design

- Mobile inference frameworks (*e.g.,* NCNN[4], MNN[5], …)
- System Optimizations
  - **layer streaming** (*i.e.,* layer-wise swapping)
  - **Layer partitioning** in TEE-based inference[3]
  - **Memory planning** in on-device training[2]

limited optimization

cannot directly apply

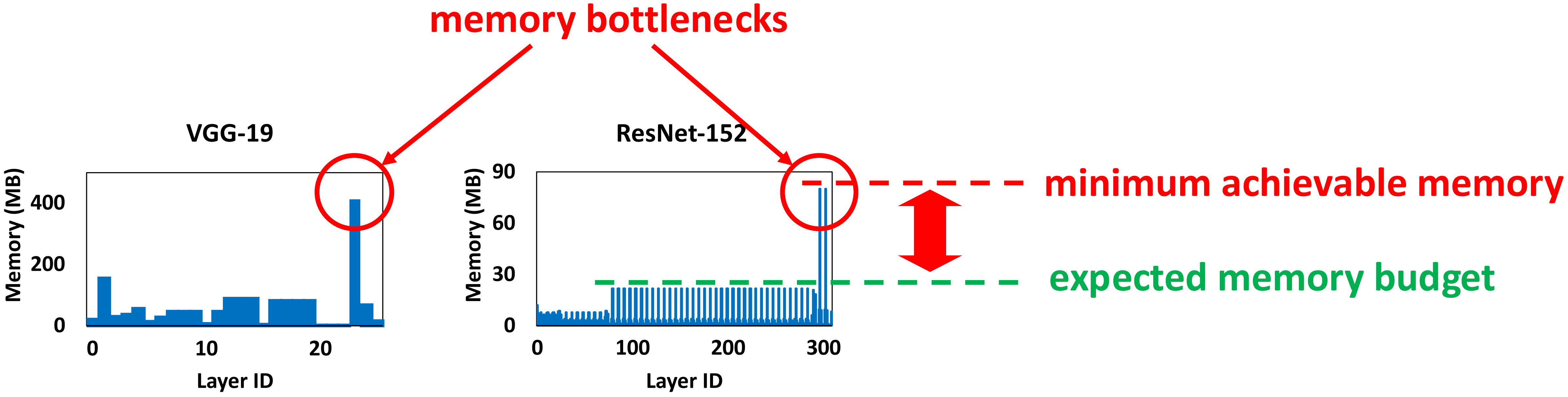
challenges

[1] Mark Sandler, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." CVPR 2018.
[2] Qipeng Wang, et al. "Melon: Breaking the memory wall for resource-efficient on-device machine learning." MobiSys 2022.
[3] Taegyeong Lee, et al. "Occlumency: Privacy-preserving remote deep-learning inference using SGX." MobiCom 2019.
[4] Hui Ni, and The ncnn contributors. Ncnn. 2017, https://github.com/Tencent/ncnn.
[5] Xiaotang Jiang, et al. "MNN: A universal and efficient inference engine." MLSys 2020.

Challenge 1: unbalanced memory footprints

Only a few layers have large memory footprints!

- In ***layer streaming***, the “largest” layers determine the peak memory.
- They become memory bottlenecks of the whole model.

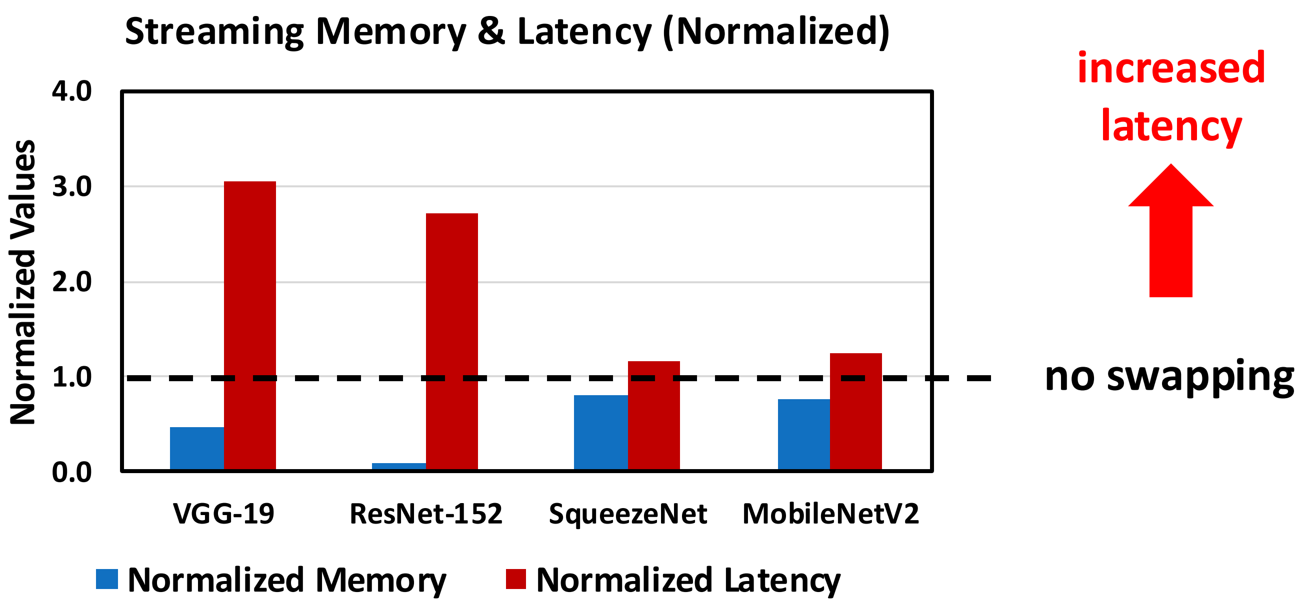


Layer-wise memory footprint of DNNs. Only a few layers have large memory footprints.
Layer IDs are consistent with the converted NCNN model format.

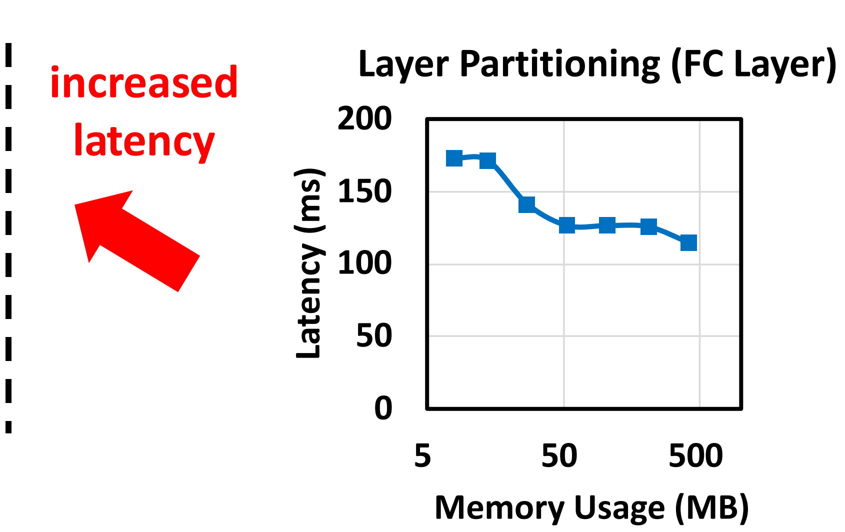
Challenge 2: memory management overhead

Memory reduction requires extra operations.

- **Layer streaming:** additional I/O and processing for repetitive weight loading.
- **Layer partitioning:** splitting/merging overhead and more memory fragments.



Memory and Latency Impacts of Naïve Layer Streaming.
The values are normalized to those without swapping.



Memory and Latency Trade-off of Naïve Layer Partitioning.

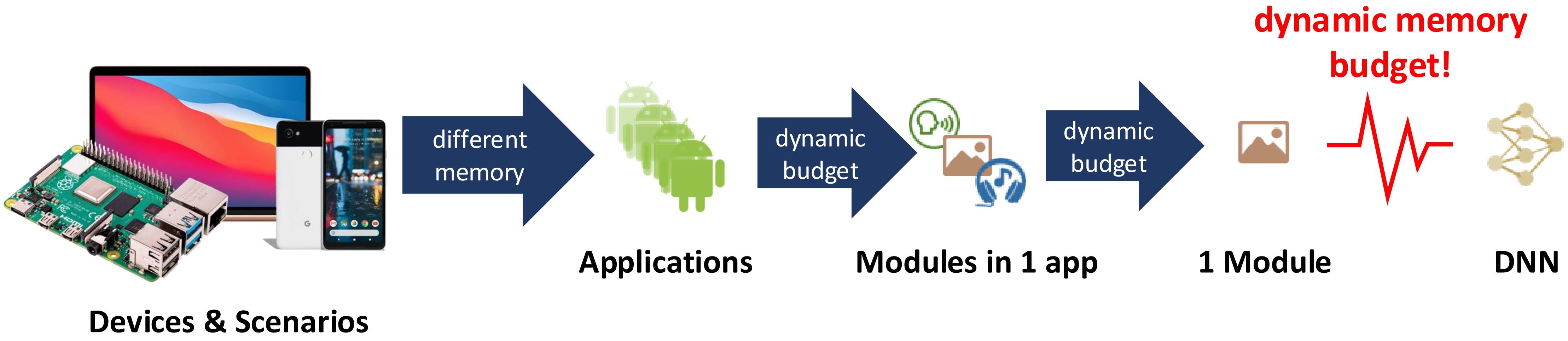
Challenge 3: memory budget dynamicity

In real-world applications, the memory budget can frequently change.

- Diverse devices and scenarios.
- Influences from other apps and modules.

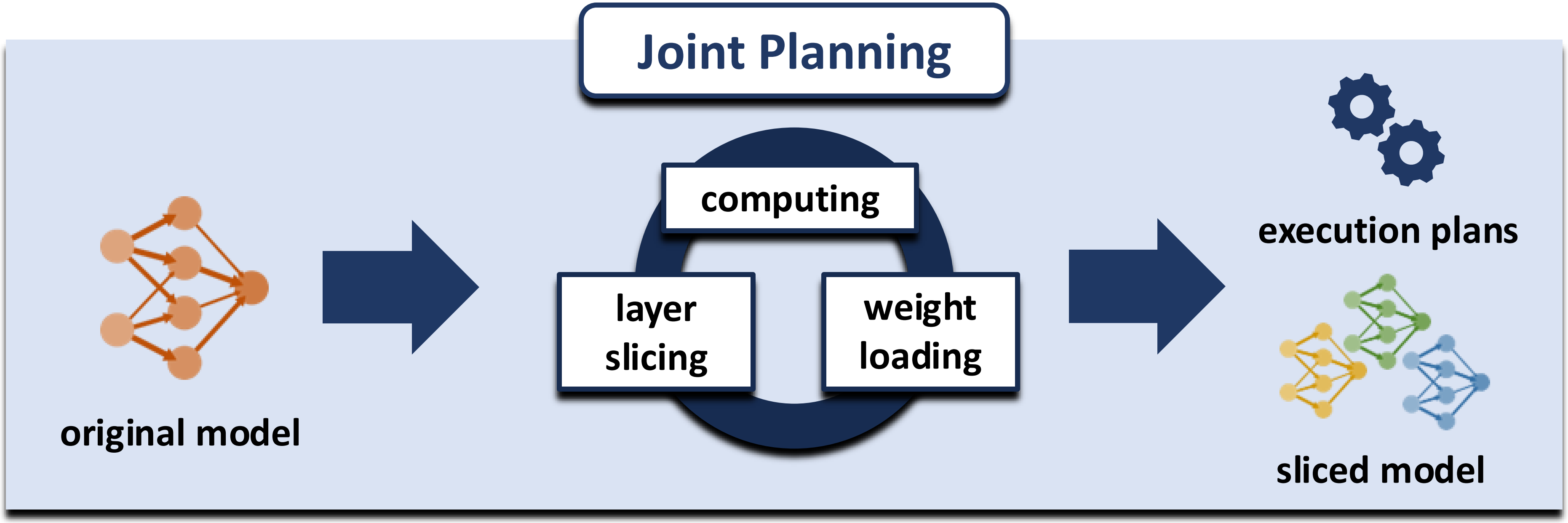
It is important to fully utilize the memory while not exceeding the budget!

- Existing memory management schemes cannot address this issue.

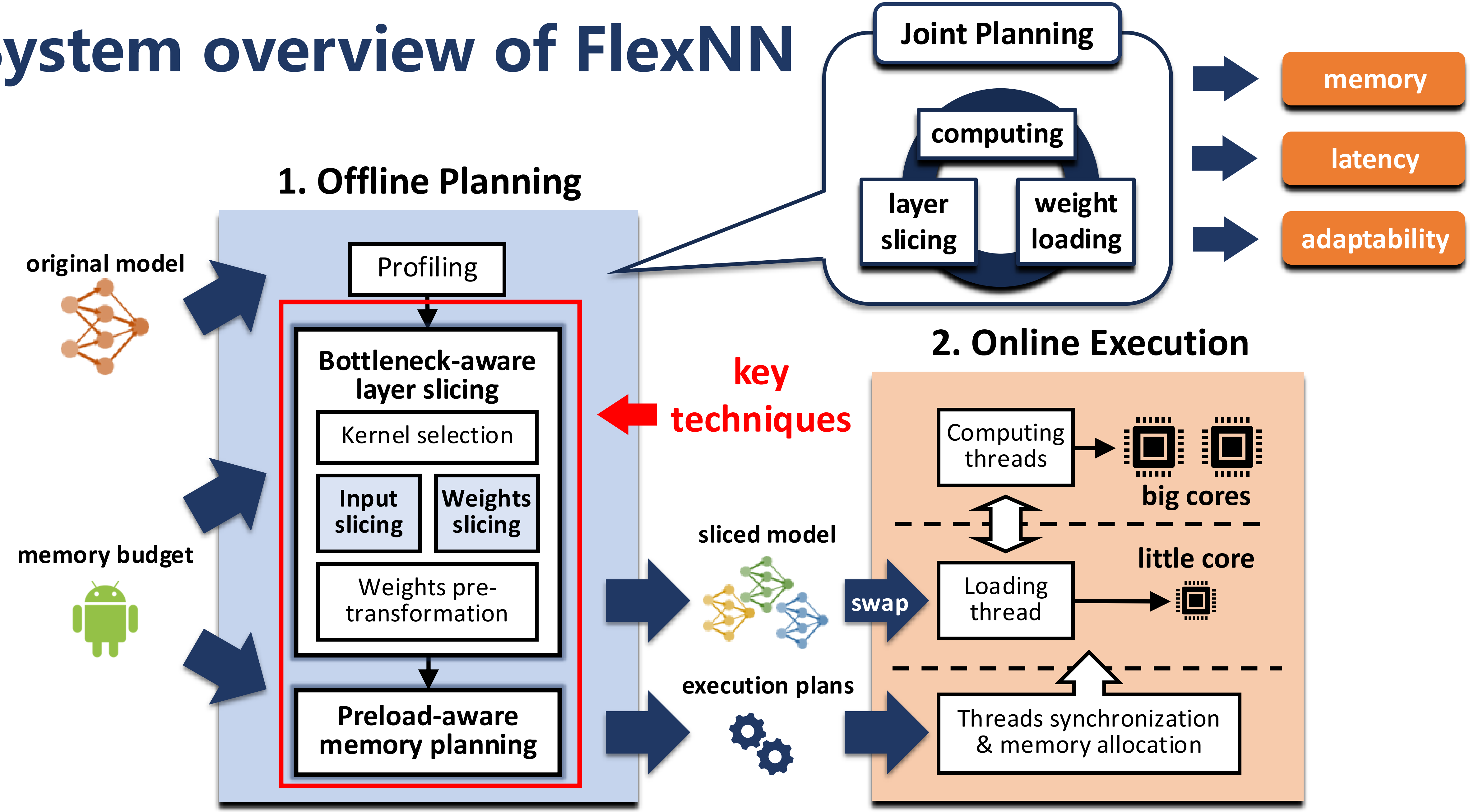


Our approach: fine-grained joint planning

Challenges	*Goals*	*Our approaches*
Unbalanced memory distribution	memory	Layer slicing
Memory management overhead	latency	Offline planning & preparation
Memory budget dynamicity	adaptability	Efficient planning algorithm



System overview of FlexNN

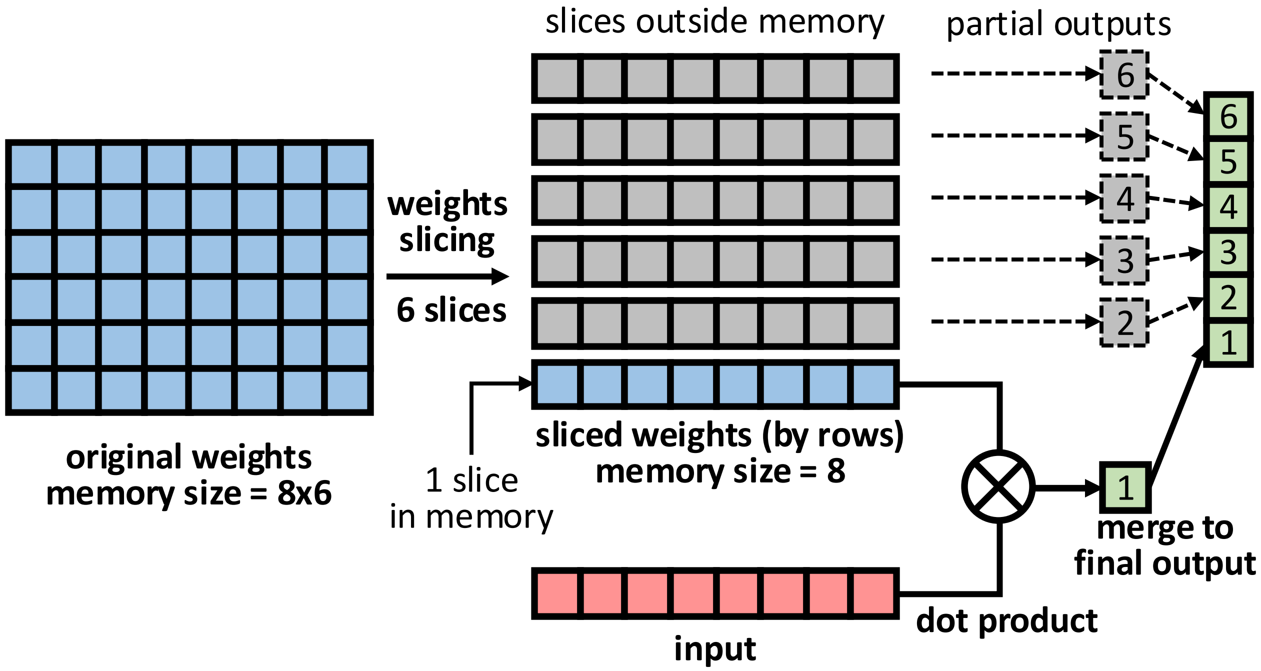


Weights slicing: Fully-Connected (FC) example

Implementation: split a large FC layer into sub-layers.

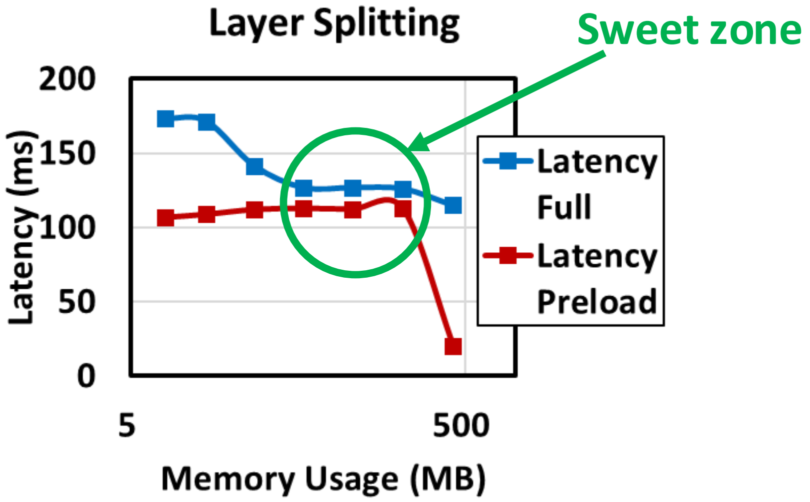
Slicing strategy: *maximize* the slice size below the memory budget.

- Reduce number of layers to schedule and run.



Weight slicing example of 8x6 FC.

The weights are partitioned into 6 slices to save 83% of memory usage.



Layer-wise latency-memory trade-off of weight slicing.

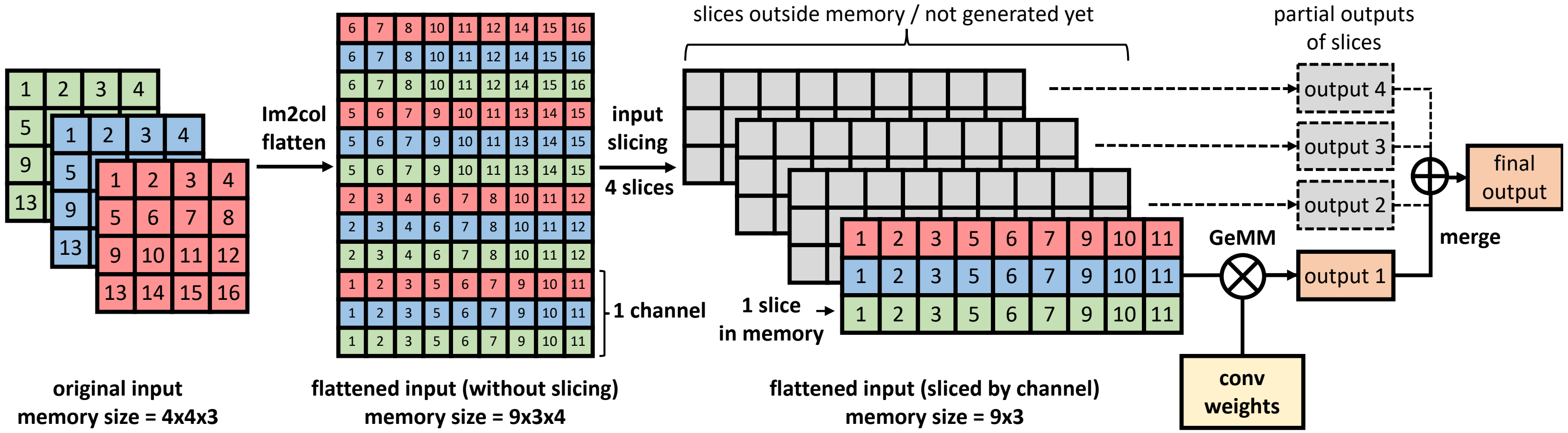
Fully-connected (FC) layers involve simple linear operations (e.g., matrix multiplications).

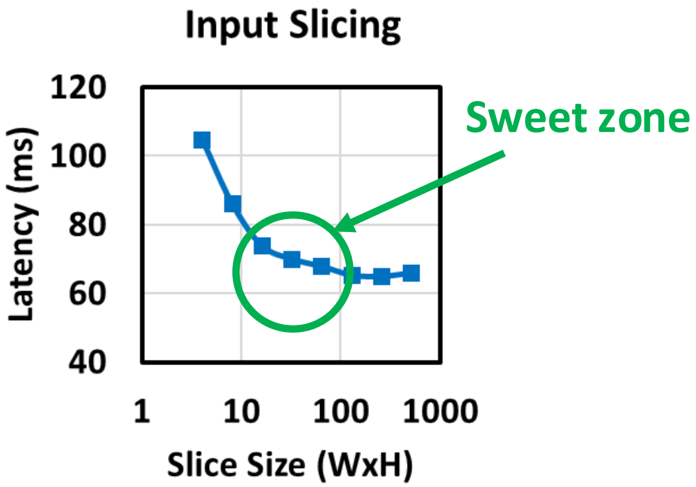
Input slicing: Im2col + GeMM example

Implementation: the GeMM input is flattened and calculated slice by slice.

Slicing strategy: *minimize* the slice size above a certain threshold.

- Memory saving with negligible overhead.
- Cannot fully utilize parallel acceleration (e.g., SIMD) if the slice size is too small.





Input slicing example of Im2col+GeMM 3x3 Conv.
The Im2col-flattened input is partitioned into 4 slices to save 75% of memory usage.

Layer-wise latency-memory trade-off of input slicing.

Im2col + GeMM is a commonly used Conv kernel, which involves flattenning the input tensor (*i.e.,* Im2col), and multiplication to the weights (*i.e.,* GeMM).

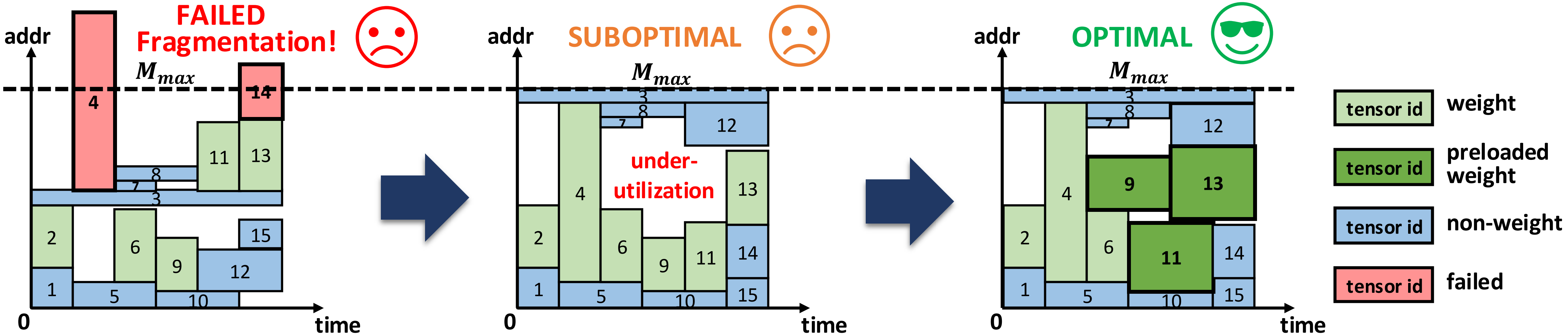
Preload-Aware Memory Planning

Preliminary: *2DBP (2D Bin Packing, NP-Hard)* formulation of memory planning.

- Allocating tensors (space & time) is equal to placing rectangles in a plane.
- Mapping: time* → x value; memory address → y value

Our case: a more complex *2DBP* variant which is aware of weight preloading.

- Difference: the allocation time of weight tensors is planning output, not input.
- **Concern: significant planning cost when adapting to a new memory budget.**



A. Dynamic (no planning) B. Common planning C. Preload-aware planning (our formulation)

* Logical time is this case refers to the layer index, since the runtime scheduling is in the granularity of layers.

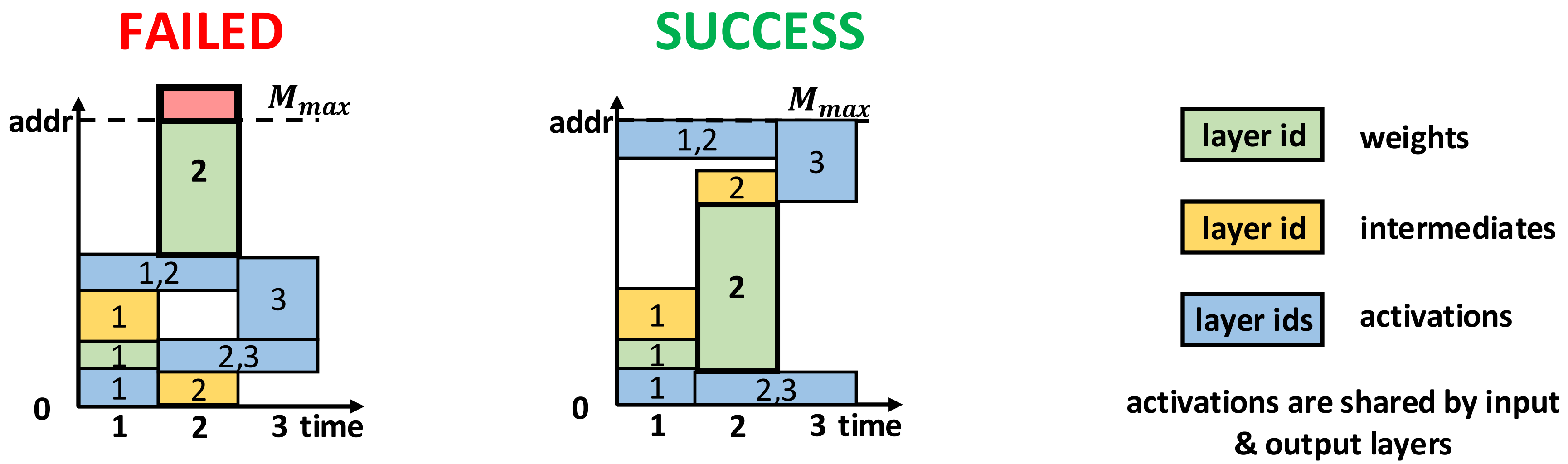
Our approach: prioritize activations

Insight: fragments are caused by *long-lifecycle* tensors.

- *Activations* have longer lifecycle than the others during inference.

Solution: plan *activations* first.

1. Plan all the activations in the model.
2. Layer-wisely plan weights and intermediates.



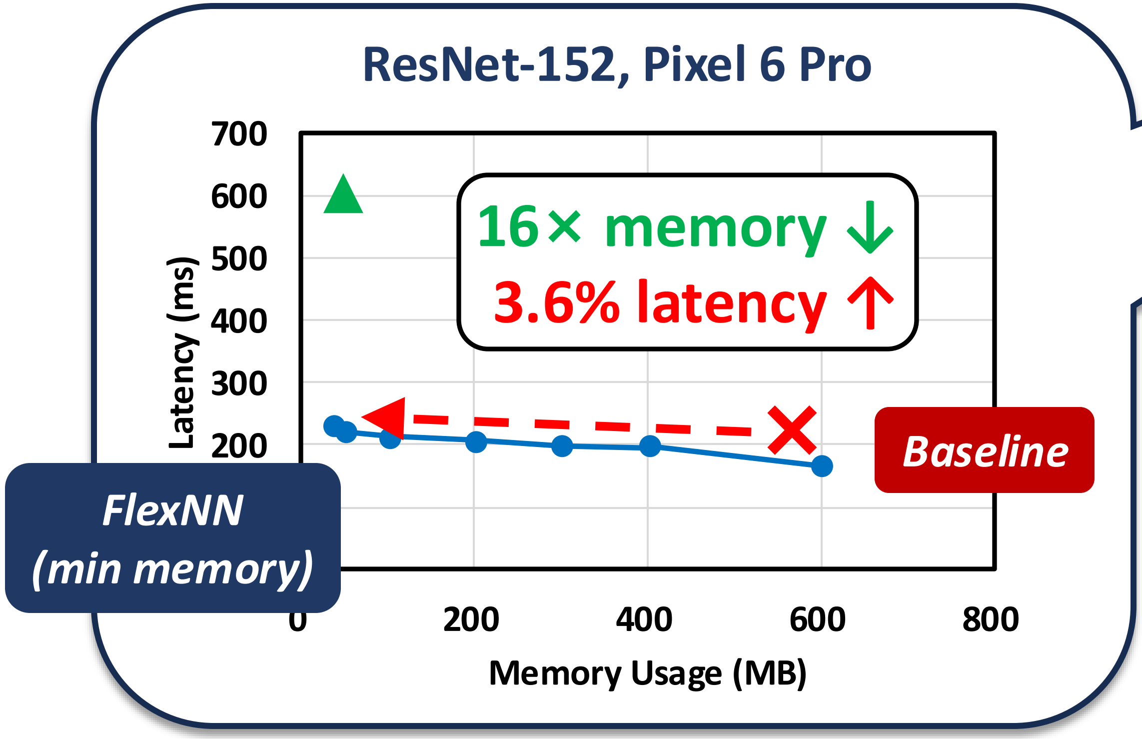
a. Greedy

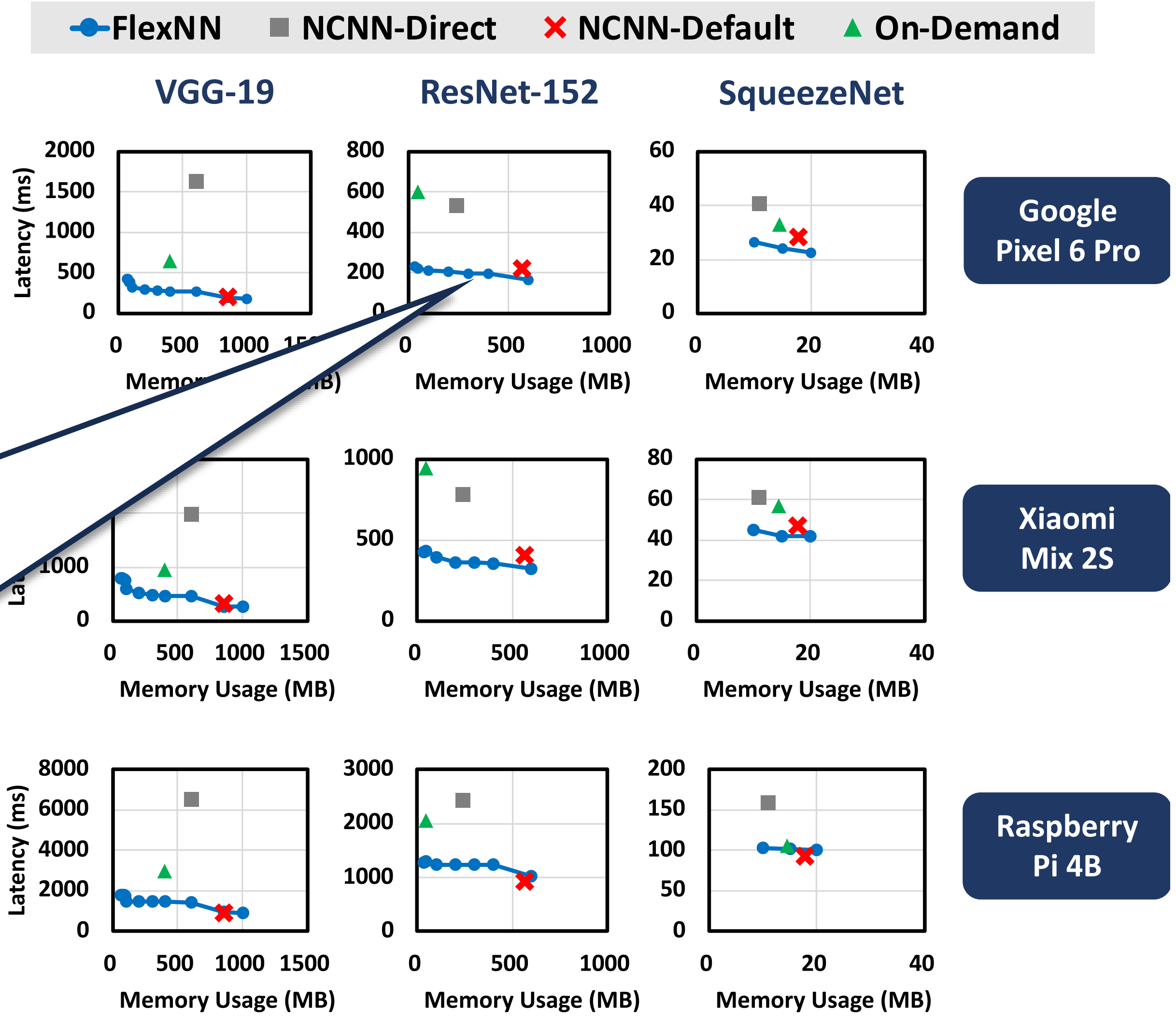
b. Activations first (ours)

Comparison between greedy and activation first planning.

Implementation & end-to-end evaluation

- **Implementation:** atop ***NCNN***[1] with 12.3k LoC added (support ARMv8 CPUs).
  - Code: https://github.com/xxxxyu/FlexNN.
- **Evaluation:** covers ***6 models*** and ***3 devices*** of different types.





End-to-end latency and memory results (partial).
FlexNN **achieves better latency-memory trade-offs.**

[1] Hui Ni, and The ncnn contributors. Ncnn. 2017, https://github.com/Tencent/ncnn.

Adaptability evaluation under changing budgets

FlexNN is able to adapt to new memory budget in ~ 1s.

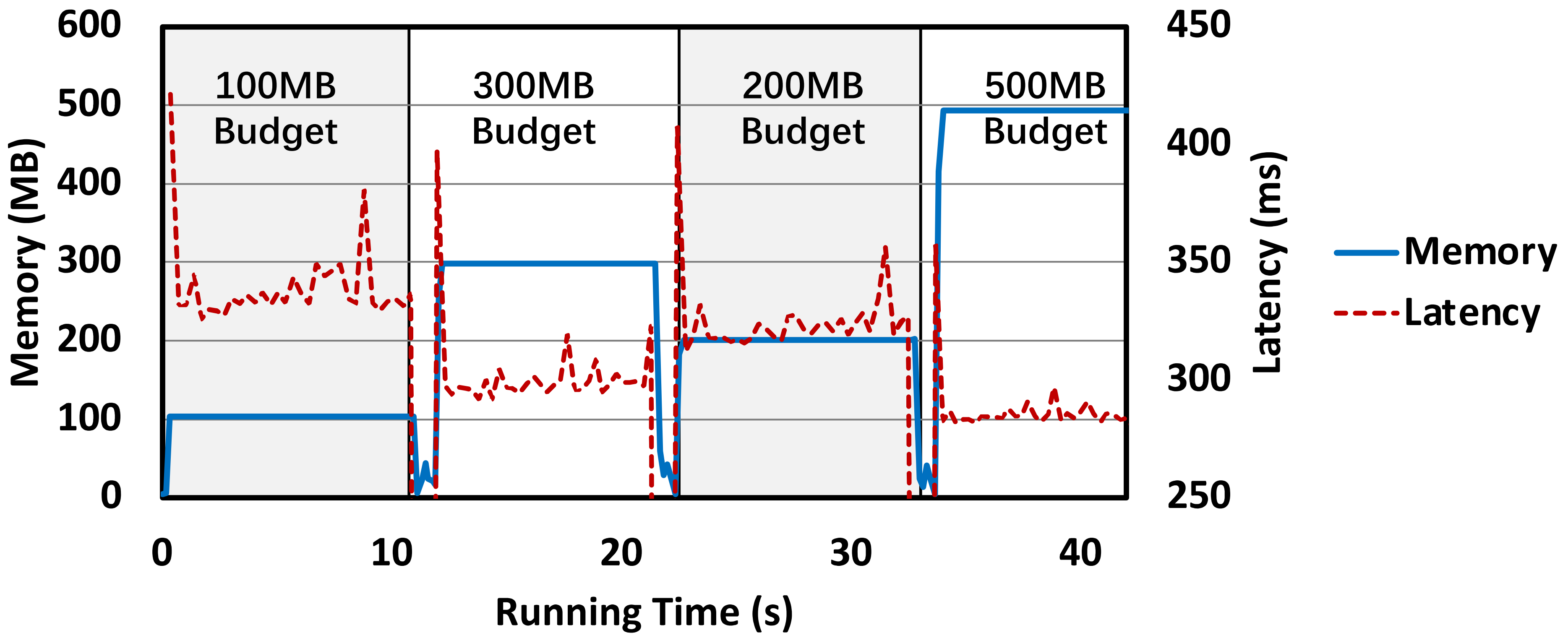


Figure 3. Real-time latency and memory under changing memory budgets.

Summary on FlexNN

Scenario: memory-constrained DNN inference on edge devices.

Goal: fast **adaption** to the given **memory** budget, with acceptable **latency** overhead.

Key Idea: streaming inference with slicing-loading-computing joint planning.

Techniques: bottleneck-aware layer slicing & preload-aware memory planning.

Key Results: **16× memory ↓** with **3.6% latency ↑**, and **~ 1s adaption.**

Thanks!

https://github.com/xxxxyu/FlexNN *lixiangy22@mails.tsinghua.edu.cn*