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

ACM MobiCom '24
Authors: Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University
Shanghai Artificial Intelligence Laboratory
Microsoft Research

汇报人：陆宇翔

2024年11月10日

Outline

- Background & Motivation
- Design
- Implementation
- Evaluation
- Conclusion

Background

近年来，深度神经网络(DNNs)已成为广泛应用于现实世界的流行技术，在多种应用场景都展现出了良好的效果。

驾驶辅助

交通监控

人脸识别

出于对网络开销、数据隐私和推理延迟的考虑，将深度神经网络部署到边缘设备（如智能相机、智能手机和微型物联网设备）上的需求日益增加。

Background

内存是边缘设备的主要瓶颈，直接决定了模型能否在许多边缘设备中使用

V. S.

NVDIA A100 GPU

每个GPU的内存约为80GB
一个典型的GPU服务器可能有8个互相连接的NVDIA A100 GPU

Google Pixel 6 pro 手机

手机只有12GB的内存
在有些操作系统和应用程序限制之下，推理的内存使用量可低至512MB甚至50MB

Background

- **现有的部署方案主要集中于降低计算成本，然而内存通常是硬约束，因此FlexNN以内存作为优先考虑的因素进行设计**
- **目前考虑设备内存有限的解决方案主要有两种思路：**

模型定制

模型压缩、高效模型架构设计、神经架构搜索

系统设计

增强内存管理方案，一种典型做法是layer streaming，也就是交换非紧急的层出内存，只在内存中保留活动的层

Background

- **设计部署在边缘设备上的DNN推理的内存管理方案的挑战：**

各层间不平衡的内存占用

在推理中，模型的峰值内存由最大的层决定， 这使得交换其他较小的层意义不大。

内存管理不可避免的开销

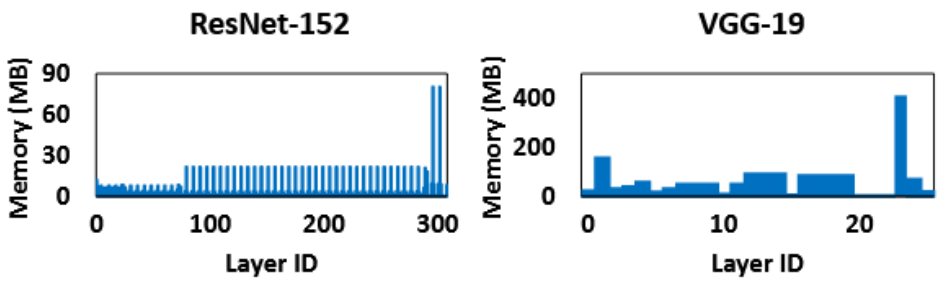
降低深度神经网络推理的内存消耗通常需要额外的操作。导致更多的成本和内存碎片。

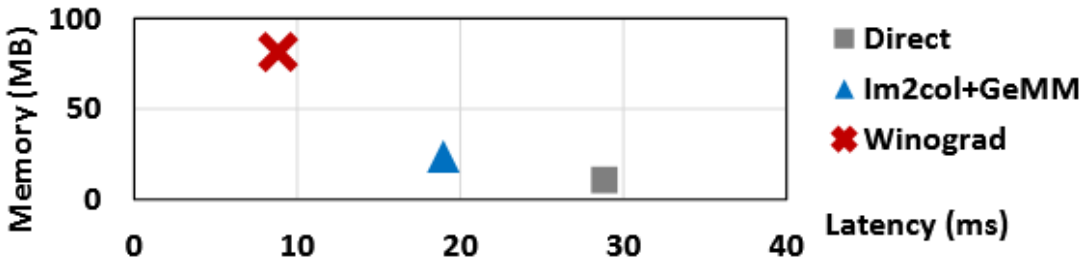
内存预算动态性

相同的深度神经网络推理可能会运行于不同的设备上，这些设备具有不同的可用内存。
此外，移动/边缘设备可能导致可用内存预算频繁变化。

Motivation

- **内存分析：从层内和层间两个层面进行观察**
  - 不平衡的内存层分布
  - 内核选择中的延迟与内存平衡
  - 各个层和内核的内存瓶颈不同





Outline

Background & Motivation

Design

Implementation

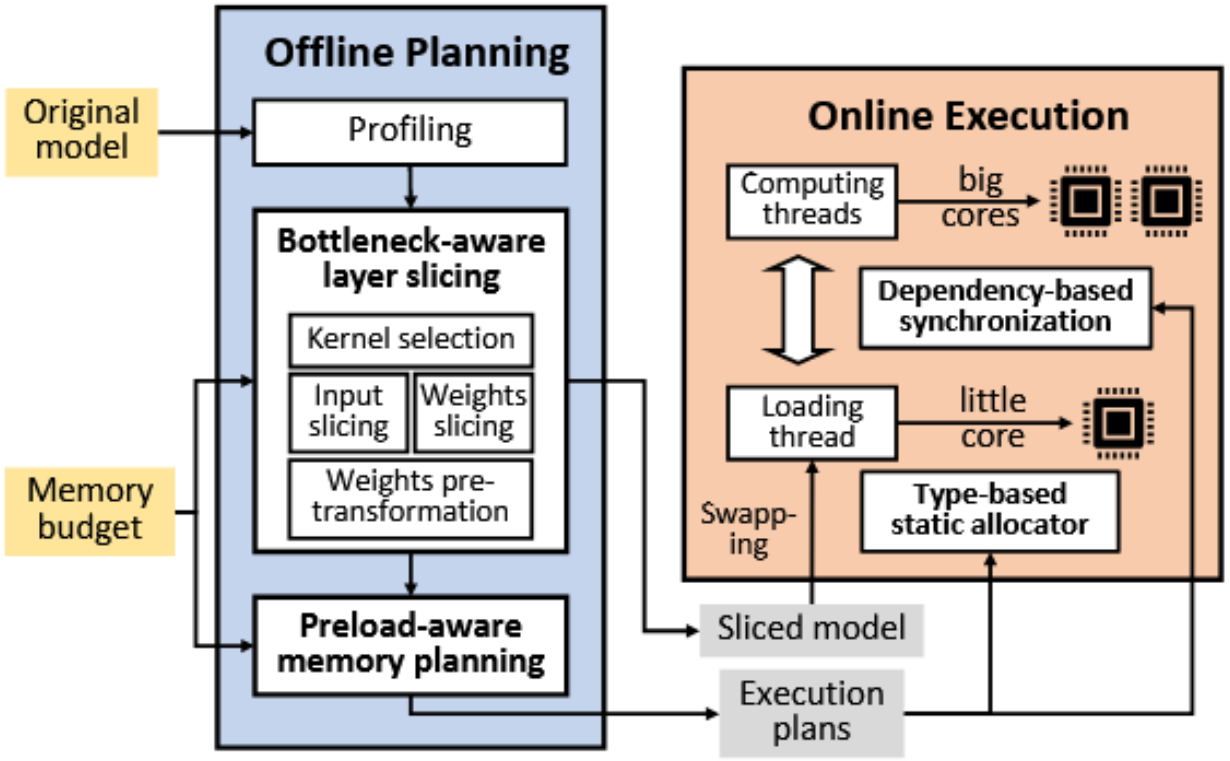
Evaluation

Conclusion

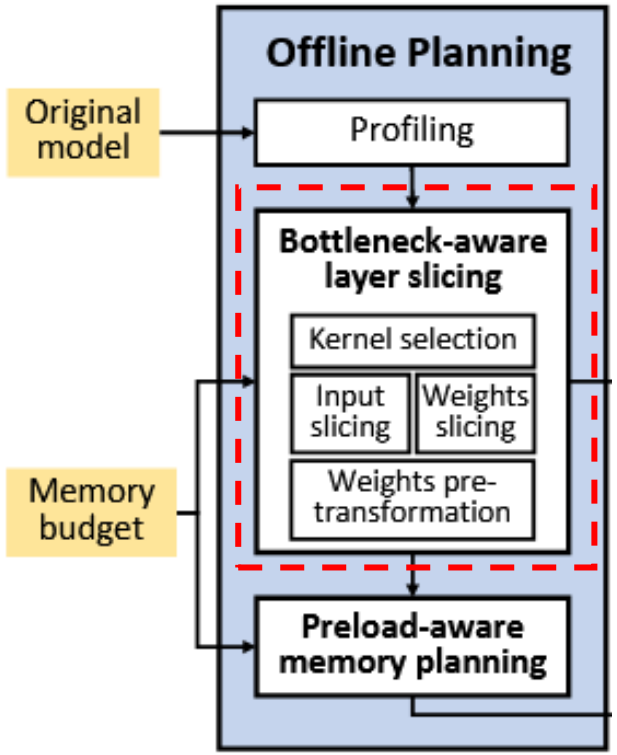
Design

FlexNN consists of two stages:

- **The offline planning stage** that performs slicing-loading-computing joint planning according to the memory budget and the given model;
- **The online execution stage** that conducts model inference based on offline-generated plans.



Design: Offline Planning



The offline planning stage involves two major modules: **Bottleneck-aware layer slicing** and **Preload-aware memory planning**.

Bottleneck-aware layer slicing

The bottleneck-aware layer slicing involves 3 steps:

(1) **kernel selection** that chooses the most suitable kernel implementation;

(2) **weights/input slicing** that performs partitioning based on the selected kernel;

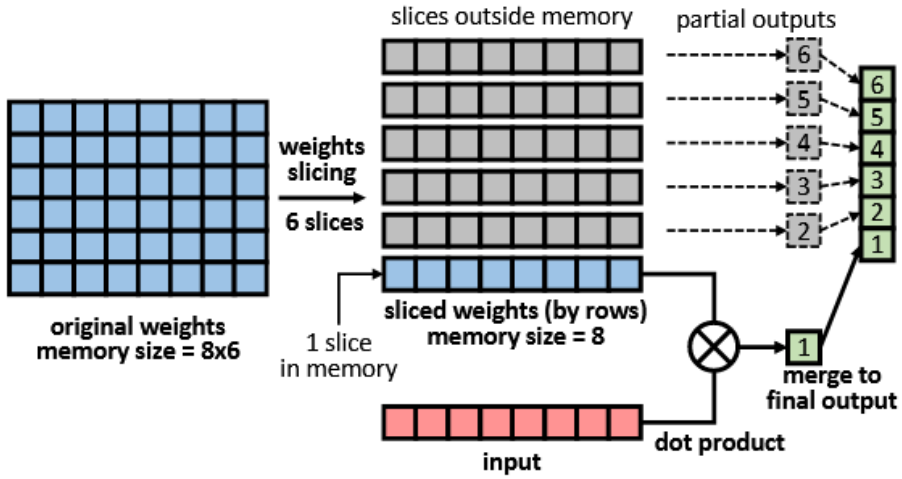
(3) **weights pre-transformation** that avoids runtime processing overhead.

Flattened inputs and **weights** are two major bottlenecks of layer-wise memory footprint. Two approaches of layer slicing: **weights slicing and input slicing**.

Design: Offline Planning

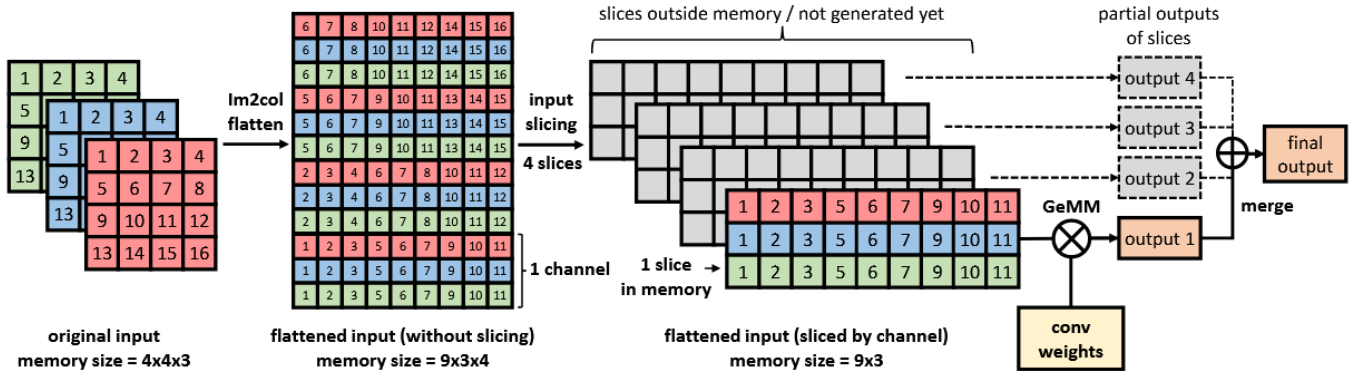
Weights slicing

It partitions layer weights into several slices, and **loads one slice each time** to reduce the memory footprint.



Input slicing

It partitions the flattened input, and **keeps one slice in memory each time** to reduce the memory footprint.

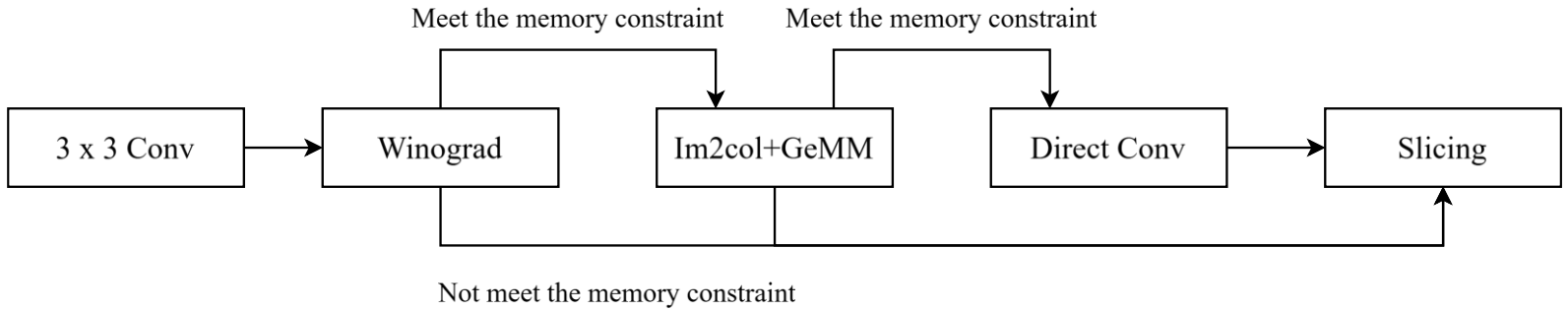


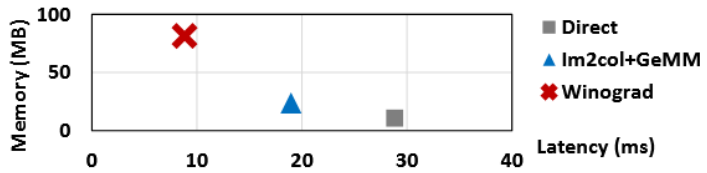
The choice between the two approaches **is determined by the bottleneck of the target layer**.
Weights slicing is more suitable for weights-dominated layers such as FC and Winograd Conv, while input slicing is more suitable for intermediates-dominated layers such as Im2col+GeMM Conv.

Design: Offline Planning

Kernel selection

Kernel selection is conducted **before** the actual partitioning process, because the choice of slicing approaches depends on the selected kernel.
Since there is a **latency-memory tradeoff** in the kernel selection of the same layer, the kernel with optimal latency might not satisfy the memory constraint.

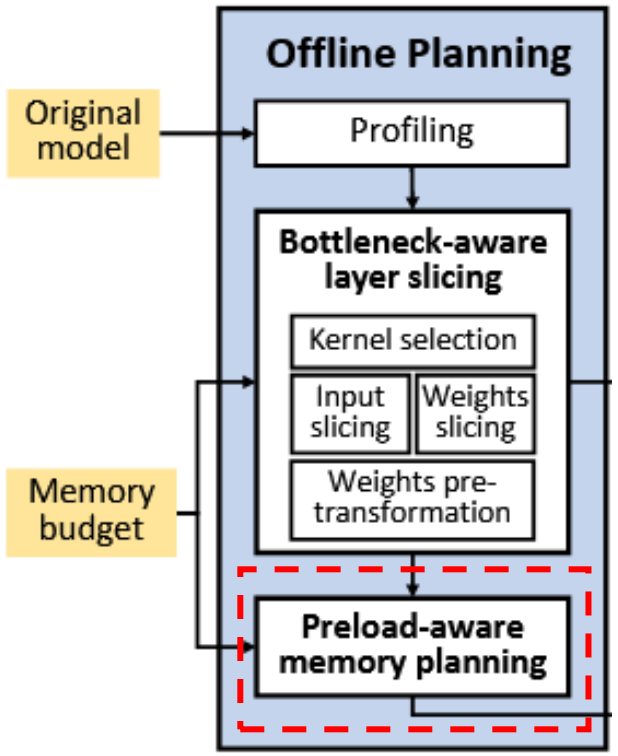




Weights pre-transformation

Weights pre-transformation is conducted after kernel selection and input/weights slicing, **to transform specific weights before execution**.
The pre-transformation allows FlexNN to **directly load the transformed weights** at runtime, and thus **avoids the runtime processing overhead**.

Design: Offline Planning



Dynamic

Dynamic (e.g., on-demand) memory management strategies in traditional DL frameworks may suffer from **increasing fragments** due to a lack of global memory information.

Static

Static memory management strategies in existing **don't take preloading into consideration**, leading to **suboptimal** plans.

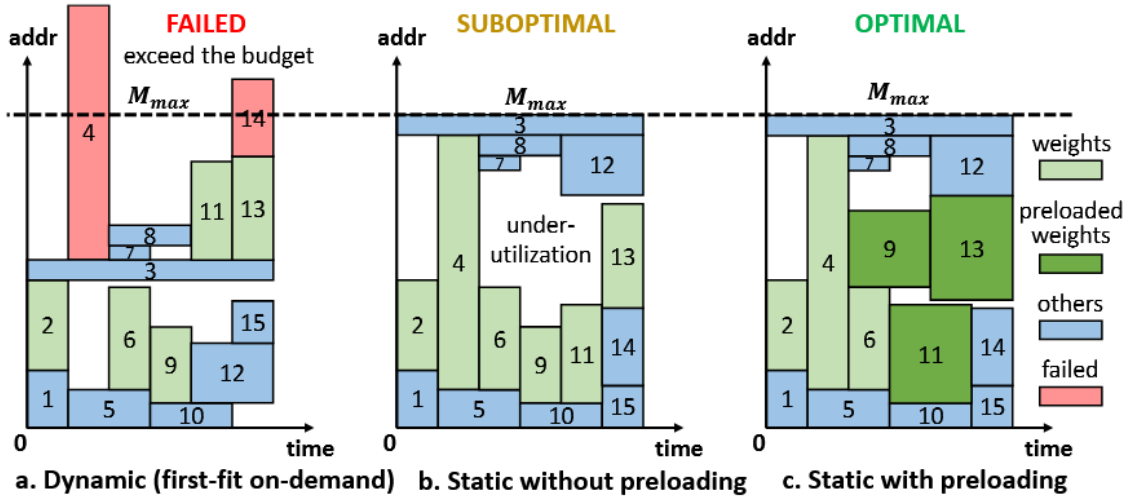
To address both issues, FlexNN employs **static memory management with preload-aware memory planning**.

Design: Offline Planning

Problem Formulation

The memory layout planning is commonly formulated to the **2D Bin Packing (2DBP) problem** with fixed time coordinate.

Our preload-aware memory planning problem can also be formulated as a variant of the 2DBP problem.



a. Dynamic (first-fit on-demand) b. Static without preloading c. Static with preloading

Given:

(1) A list of n tensors with known properties: the allocation time without preloading $\{s_i\}$, the de-allocation time $\{e_i\}$, the memory size $\{m_i\}$, and the memory type $\{type\ (i)\}$ (weights, activations and intermediates).

(2) A memory budget M_{max}.

Objective:

Minimizes the inference latency while satisfying the constraints. The plan is represented as a list of time-address pairs $\Omega = \{\langle time_i, addr_i \rangle\}$.

Constraints:

(1) Lifecycle constraint: $1 \le time_i \le s_i$ for all weight tensors (allowing for preloading), and $time_i = s_i$ for all other types of tensors.

(2) Memory usage constraint: for all time τ_l from τ_1 to τ_L, where L is the number of time steps, i.e., number of layers, $\sum_{time_i \le \tau_l \le e_i} m_i < M_{max}$.

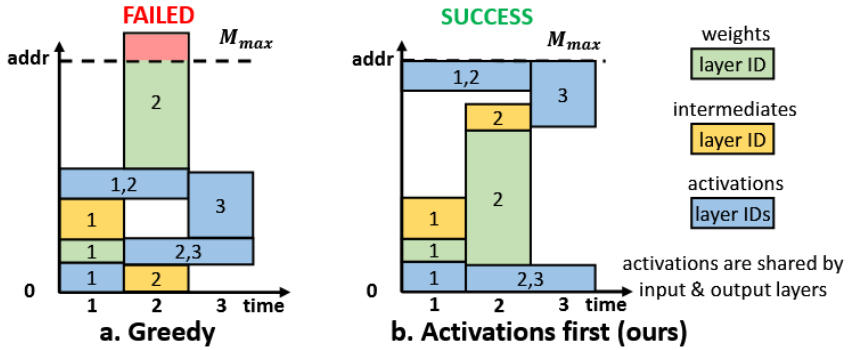
(3) Memory address constraint: $0 \le addr_i \le M_{max} - m_i$ for all i, l.

(4) Memory non-overlapping constraint: either $(time_i \ge e_j) \vee (time_j \ge e_i)$ (lifecycle non-overlapping) or $(addr_i + m_i \le addr_j) \vee (addr_j + m_j \le addr_i)$ (memory address non-overlapping)

Design: Offline Planning

Planning Algorithm

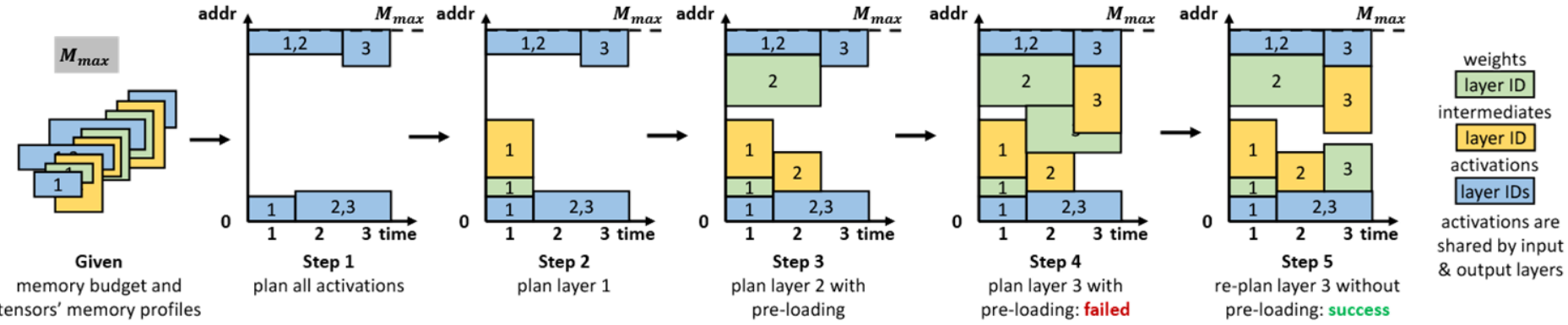
We employ the lightweight Algorithm, which leverages the memory access patterns of DNN inference, to minimize both memory fragments and I/O waiting time.



Algorithm 1 Lightweight memory planning algorithm

Input: memory budget M_{max}, tensors profiles: allocation time $\{s_i\}$, de-allocation time $\{e_i\}$, memory size $\{m_i\}$, memory type $\{type(i)\}$
Output: the memory plan $\Omega = \{\langle time_i, addr_i \rangle\}$

```
empty plan Ω
Ω ←PlanActivations(Ω, {s_i}, {e_i}, {m_i}, M_max)
for l in range(layer count) do
    backup plan Ω
    Ω ←PreloadWeights(Ω, {s_i}, {e_i}, {m_i}, M_max)
    Ω ←PlanIntermediates(Ω, {s_i}, {e_i}, {m_i}, M_max)
    if Ω fails then
        restore plan Ω
        Ω ←PlanNoPreload(Ω, {s_i}, {e_i}, {m_i}, M_max)
    ▷ Re-schedule weights and intermediates w.o. preloading.
        if Ω fails then
            return Error.                    ▷ Schedule fails.
        end if
    end if
end for
return Ω                                     ▷ Schedule succeeds.
```

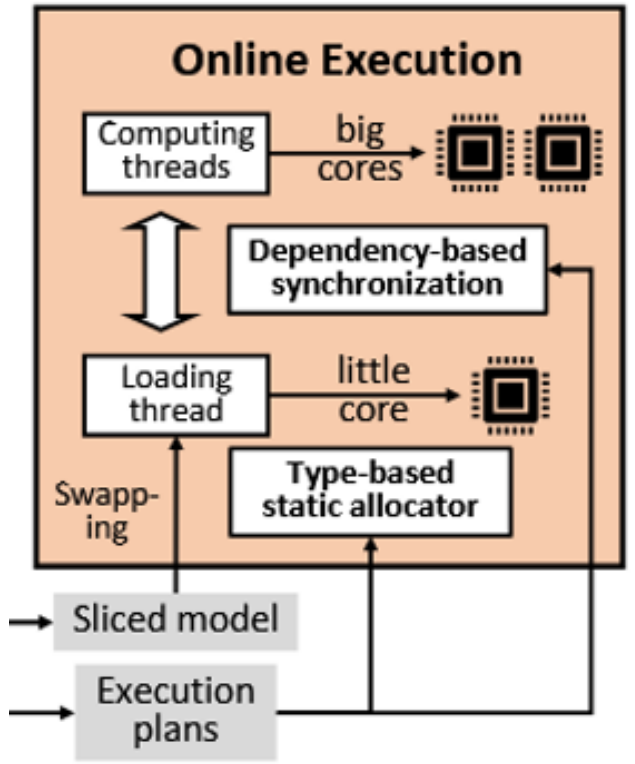


Design: Online Excution

The online execution stage aims to conduct the actual model inference while correctly following the plans determined in the offline planning stage. It mainly needs to address **two gaps** between the planning results and the actual execution：

- The gap between tensor-wise planning and layer-wise execution.
- The gap between logical time and actual execution time.

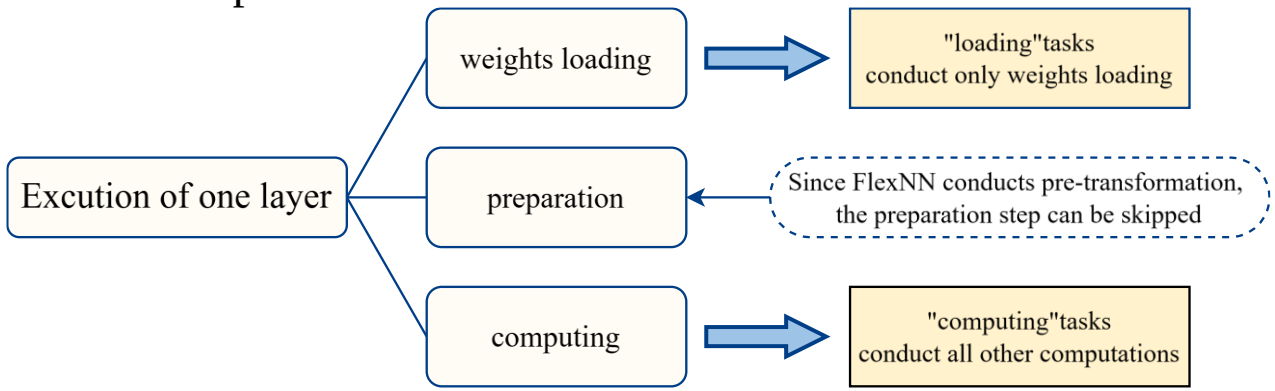
Solution:
Dependency-based synchronization scheme to schedule the loading and computing of layers, and **a type-based static allocator** to ensure the correctness of allocations.



Design: Online Excution

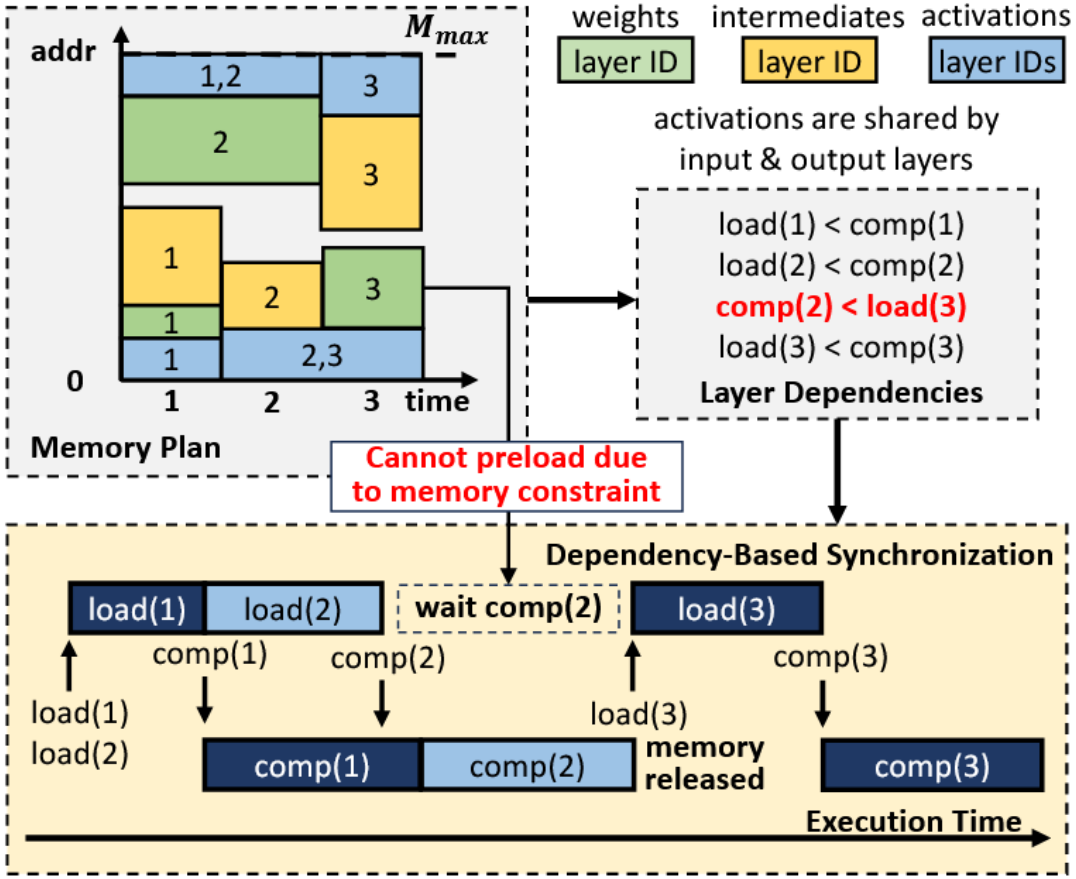
Dependency-based synchronization

FlexNN captures the **memory dependency** between layers to convert the tensorwise memory plan to the layer-wise execution plan.



There are two types of dependencies at the task level, use > and < to represent the dependency between tasks.

(1) **Loading before computing**, denoted as *load* (*i*) < *comp* (*i*): layer *i*'s computing depends on layer *i*'s loading.
(2) **Computing before loading**, denoted as *comp* (*i*) < *load* (*j*): layer *j*'s loading depends on layer *i*'s computing if they have intersection of memory address.



Design: Online Excution

Type-based static allocation

It is designed to ensure correct memory allocation results at runtime.
The key to achieve it is to **define the order of allocations.**
Based on the observation that each thread will only allocate memory for a specific type of tensor, FlexNN **uniquely identify** each tensor by its type and the count within the type (e.g., Weights-5) .

Outline

Background & Motivation

Design

Implementation

Evaluation

Conclusion

Implementation

- **在NCNN上实现了FlexNN的原型**
- **NCNN是一个开源推理引擎,在 Arm CPUs 上具有高度优化的内核，并在后端支持许多实际的手机应用程序**
- **目前FlexNN是在Arm CPU上实现部署的**

- **选择NCNN的三个主要原因：**
  - NCNN 针对移动设备进行了高度优化，优于大多数其他框架；
  - NCNN 支持一系列良好的模型，包括 CNN、RNN、LSTM、Transformers等；
  - NCNN 还拥有良好的社区支持和清晰的代码结构。

Implementation

- **FlexNN对NCNN上的很多模块进行了修改，包括：**
  - 基于模型写入工具实现层划分，以修改计算图并写入转换后的权重；
  - 在运行时引擎中，实现基于依赖关系的同步并行预加载；
  - 将 NCNN 的原生分配器修改为基于类型的静态内存分配器，实现在初始化时分配一个连续的缓冲区，并在推理过程中管理该缓冲区；
  - 修改了NCNN的一些算子。
- **FlexNN联合 Armv8-A 和 NEON进行部署。Arm CPU 广泛应用于各种类型的边缘设备，Armv8-A 架构现在是智能手机事实上的标准。因此，在FlexNN的实现和部署中，将目标对准了 Armv8-A CPU。NEON 是一种先进的单指令多数据(SIMD)架构扩展，在执行层划分时，如果可能的话，始终保持输入/权重划分的宽度/ 高度/通道数为 8 的倍数，以充分利用 NEON 的加速，避免性能下降。**

Outline

Background & Motivation

Design

Implementation

Evaluation

Conclusion

Evaluation

实验设置

- **模型：**使用六个广泛使用的深度神经网络模型对 FlexNN 进行评估，包括 ResNet-152、VGG-19、Vision Transformer(ViT)、GPT-2、MobileNetV2 和 SqueezeNet。
- **平台：**在两种不同规格的硬件设备上（单板计算机和智能手机）进行了评估，都配备了Armv8-A CPU，并使用big cores进行计算。
- **指标：**涵盖了表明系统在实际应用中实际性能的大多数指标，包括内存使用率、推理延迟和能耗。
- **Baseline：**所有Baseline基于NCNN，主要原因有两个：

1）在NCNN基础上实现的FlexNN，并且NCNN已经是性能最好的移动推理框架之一

2）FlexNN是第一个优先考虑内存的移动推理框架，因此没有类似工作可比较

Device Name	CPUs	RAM (GB)
Google Pixel 6 Pro	2x2.80 GHz Cortex-X1 2x2.25 GHz Cortex-A76 4x1.80 GHz Cortex-A55	12
Xiaomi Mi Mix 2S	4x2.8 GHz Kryo 385 Gold 4x1.8 GHz Kryo 385 Silver	6
Raspberry Pi 4B	4x1.8 GHz Cortex-A72	8

作为Baseline的方法包括：

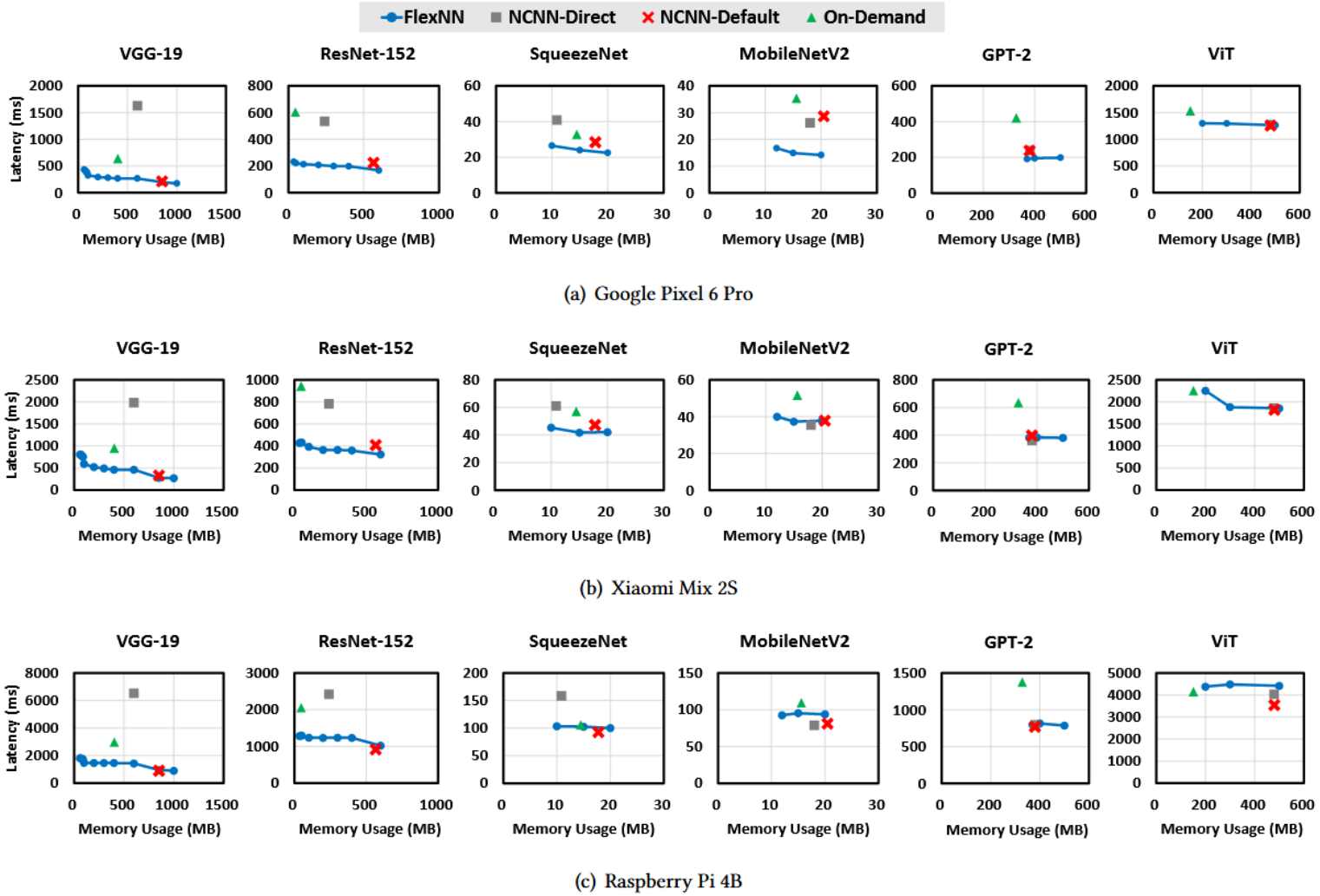
- 默认的NCNN（“NCNN-Default”）
- 禁用Im2col+GeMM和Winograd内核以减少内存的“NCNN-Direct”
- 实施分层交换和按需加载策略的“On-Demand”

所有Baseline都使用与 FlexNN 相同数量的big cores进行计算，而不使用little cores，因为若同时使用，由于不平衡的负载会减慢NCNN推理速度

Evaluation

端到端性能

评估涵盖了三个边缘设备和 6 种不同类型的 DNN 模型。



(a) Google Pixel 6 Pro

(b) Xiaomi Mix 2S

(c) Raspberry Pi 4B

- 默认的NCNN(“NCNN-Default”)
- 禁用Im2col+GeMM和Winograd内核以减少内存的“NCNN-Direct”
- 实施分层交换和按需加载策略的“On-Demand”

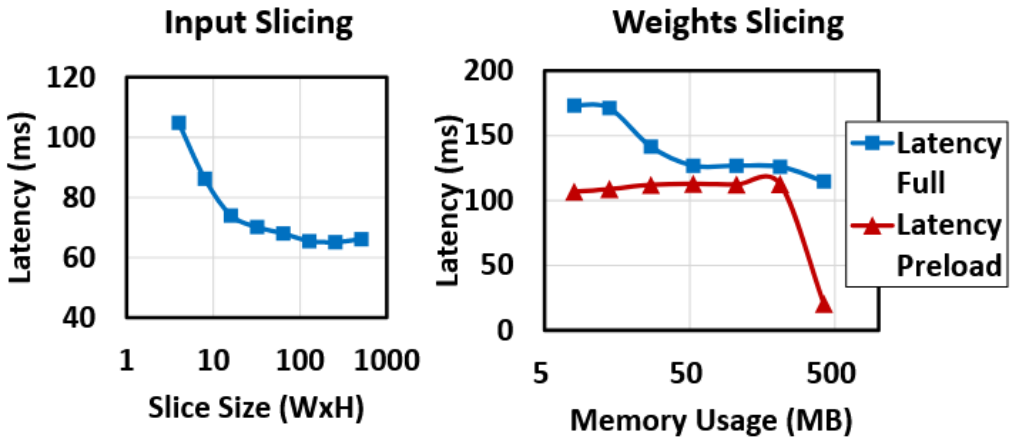
Evaluation

延迟与内存的权衡

与Baseline相比，FlexNN本身就表现出了延迟和内存的权衡。
这种权衡来自于两个方面：

(1) 有更大的内存预算，FlexNN就会使用大的切片尺寸和更快的计算内核以降低延迟

(2) 对于选定的切片策略和计算内核，较大的内存预算允许FlexNN预加载更多权重以加速推理



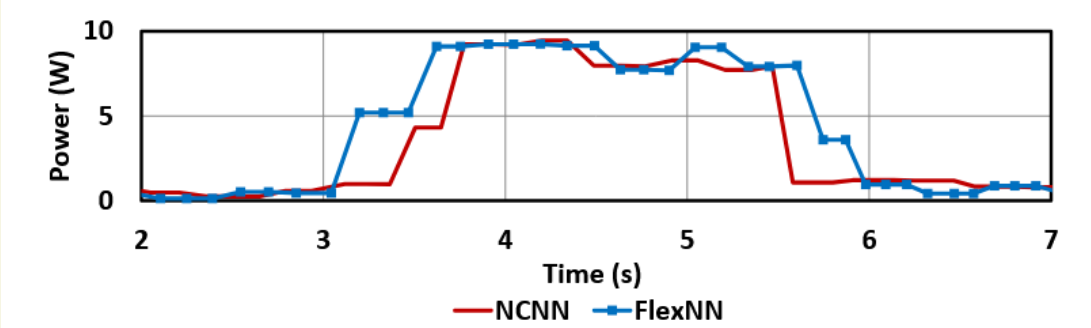
系统开销

系统开销主要包含两个部分：
离线规划开销和运行时I/O开销

离线规划开销

Model	Memory Budget (MB)	Transformed Weights Storage (MB)	Layer Slicing Cost (ms)	Profiling Cost (ms)	Memory Planning Cost (ms)
VGG-19	100	781	2,458.96	689.25	5.27
ResNet-152	100	547	2,228.76	540.94	864.67
Vision Transformer	300	337	888.84	1,438.84	390.55

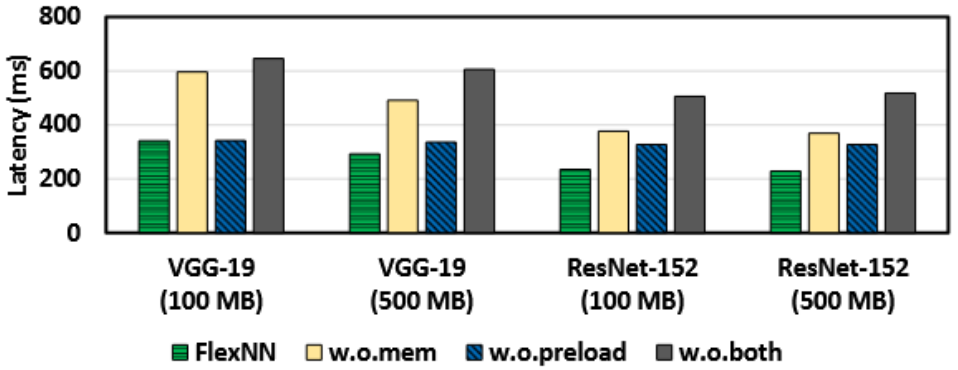
运行时I/O开销



Evaluation

消融实验

实验展示了静态内存管理和并行预加载对减少延迟的影响。
实验结果表明，在给定的内存预算下，静态内存管理和并行预加载都能有效减少推理延迟。



适应内存预算变化

实验展示了 FlexNN 在不同内存预算下的实时性能，FlexNN依次适应四种不同内存预算。
当内存预算发生变化时，FlexNN 会暂停推理过程，释放内存，为新的内存预算生成执行计划，重新分配内存，最后恢复推理。整个适应过程大约需要 1 秒钟。

Outline

Background & Motivation

Design

Implementation

Evaluation

Conclusion

Conclusion

- FlexNN是一种用于内存受限的本地设备深度神经网络推理的高效自适应内存管理框架。
- 利用切片-加载-计算联合规划方法实现了最优内存利用率和最小的内存管理开销。

Idea

- 如文章中所针对的主要是CNN的多种模型，而对大模型如GPT-2，VIT的效果并不理想，能否在类似大模型中进行改良?
- FlexNN在处理内存瓶颈时，主要针对权重或者输入的进行划分，而没有考虑如果内存的瓶颈不在计算层而是在激活层的可能性。
- FlexNN是一种内存管理方案，这与改变模型结构层面的优化方案应是正交的，是否可以实现两个方面的联合优化?

谢谢大家
请老师同学批评指正