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

Xiangyu Li
lixiangy22@mails.tsinghua.edu.cn
Institute for AI Industry Research (AIR), Tsinghua University
Beijing, China

Yuanchun Li
liyuanchun@air.tsinghua.edu.cn
Institute for AI Industry Research (AIR), Tsinghua University
Beijing, China
Shanghai Artificial Intelligence Laboratory
Shanghai, China

Yuanzhe Li
liyuanzhe@air.tsinghua.edu.cn
Institute for AI Industry Research (AIR), Tsinghua University
Beijing, China

Ting Cao
ting.cao@microsoft.com
Microsoft Research
Beijing, China

Yunxin Liu[*]
liuyunxin@air.tsinghua.edu.cn
Institute for AI Industry Research (AIR), Tsinghua University
Beijing, China
Shanghai Artificial Intelligence Laboratory
Shanghai, China

ABSTRACT

Due to the popularity of deep neural networks (DNNs) and considerations over network overhead, data privacy, and inference latency, there is a growing interest in deploying DNNs to edge devices in recent years. However, the limited memory becomes a major bottleneck for on-device DNN deployment, making it crucial to reduce the memory footprint of DNN. The mainstream model customization solutions require intensive deployment efforts and may lead to severe accuracy degradation, and existing deep learning (DL) frameworks don't take memory as a priority. Besides, recent works to enhance the memory management scheme cannot be directly applied because of several challenges, including the unbalanced memory footprint across layers, the inevitable overhead of memory management, and the memory budget dynamicity. To tackle these challenges, we introduce FlexNN, an efficient and adaptive memory management framework for DNN inference on memory-constrained devices. FlexNN uses a slicing-loading-computing joint planning approach, to achieve optimal memory utilization and minimal memory management overhead. We implemented FlexNN atop NCNN, and conducted comprehensive evaluations with common model architectures on various devices. The results have shown that our approach is able to adapt to different memory constraints with optimal latency-memory trade-offs. For example, FlexNN can reduce the memory consumption by 93.81% with only a 3.64% increase in latency, as compared with the original NCNN on smartphones.

[*]Corresponding author.

ACM MobiCom '24, November 18-22, 2024, Washington D.C., DC, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0489-5/24/09.
https://doi.org/10.1145/3636534.3649391

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Artificial intelligence**.

KEYWORDS

Edge Device, Deep Learning, DNN Inference, Memory Management

ACM Reference Format:
Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *The 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), September 30-October 4, 2024, Washington D.C., DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3649391

1 INTRODUCTION

In recent years, deep neural networks (DNNs) have become a popular technique used for a wide range of real-world applications, including driving assistance [12, 14, 15, 24, 48], traffic monitoring [2, 3, 25, 51], and face recognition [9, 30, 39, 43]. Due to considerations over network overhead, data privacy, and inference latency, there is a growing interest in deploying deep neural networks to edge devices, such as smart cameras, smartphones, and tiny IoT devices. Today, there are already many deep learning (DL) frameworks [1, 8, 22, 29] that support on-device inference.

Unlike the cloud-deployed DNN models that are based on powerful GPU clusters, a major bottleneck for on-device DNN deployment is memory. For example, a typical GPU server may have eight interconnected NVIDIA A100 GPUs, each of which has about 80GB of memory [46], but a high-end Android smartphone like the Google Pixel 6 Pro only has 12GB of memory. Moreover, the operating system and the applications may further limit the memory usage of DNN inference to as low as 512MB or even 50MB (more details in § 2.1). Therefore, reducing the memory footprint of DNN inference is crucial, which directly determines whether or not a model can be used in many mobile/edge applications.

The mainstream solution to deploy DL models to memory-constrained devices is model customization. This involves various techniques, including model compression [4, 17, 19, 27], efficient model architecture design [21, 36, 41, 52], and neural architecture search (NAS) [5, 34, 40, 47, 50]. For example, pruning and quantization are the most popular model compression methods that shrink the model size by removing redundant parameters and reducing parameter precision, respectively. Model architecture design and search aim to produce more compact network structures that can fit into limited memory. However, applying these techniques requires intensive development efforts in model design and training, and may lead to severe accuracy degradation due to the reduced model capacity [26].

A more developer-agnostic solution is to reduce DNN memory consumption through system design, which is orthogonal to the model customization approaches. However, the memory management design of existing mobile DL frameworks is mostly inherited from the traditional frameworks, which do not treat memory as a priority. For example, most existing frameworks (e.g., MNN [22], NCNN [29], etc.) simply put all weights in memory during inference, and manage non-persistent tensors through a coarse-grained memory pool dynamically (e.g., with an on-demand strategy).

Due to the limited consideration of memory in existing DL frameworks, researchers have proposed various techniques to enhance the memory management scheme. A typical practice is to swap the non-urgent data to the storage to reduce the memory footprint. Specifically, a straightforward memory swapping strategy for DNN inference is layer streaming, which loads and executes one or multiple layers of the model each time, thereby keeping only the active layer(s) in the memory. There are also several approaches that consider memory layout planning [44] and layer partitioning [26] in on-device training or TEE-based inference scenarios. However, directly applying these techniques to on-device inference faces several challenges, as explained below.

A. Unbalanced memory footprint across layers. In layer-wise streaming inference, the peak memory of model inference is determined by the largest layer. Specifically, the layer that consumes the most memory becomes the bottleneck for the peak memory of inference, which makes it less meaningful to swap other smaller layers.

B. Inevitable overhead of memory management. Reducing the memory consumption of DNN inference usually requires extra operations over the normal inference process. For example, to enable layer streaming, additional memory I/O is required to load weights before computation. Due to the Unified Memory Architecture (UMA) and limited storage I/O bandwidth on edge devices, the overhead is usually non-negligible. Meanwhile, partitioning the layers also causes additional splitting/merging costs of weights and data. It may also lead to more memory fragments that hinder memory management.

C. Memory budget dynamicity. In real-world deployment, the same DNN inference may run on diverse devices in various scenarios with diverse available memory. Moreover, the memory of a mobile/edge device is often shared by multiple applications and functional modules, leading to frequent changes in the available memory budget. This dynamicity imposes challenges on the memory management scheme to fully utilize the available memory. Specifically, using an on-demand memory management strategy may lead to fragmentation and under-utilization of the available memory, while adjusting a static memory management strategy with ahead-of-time planning on the fly may cause significant adaption costs.

To this end, we introduce FlexNN, an efficient and adaptive on-device DNN inference framework for memory-constrained scenarios. FlexNN addresses the above challenges through a *slicing-loading-computing joint planning* approach. The key insights include (1) enabling optimal memory utilization through a fine-grained co-design of the model execution plan and memory management plan; and (2) suppressing runtime memory management overhead via intensive offline preparation.

Specifically, regarding the unbalanced memory footprint between layers, we propose *bottleneck-aware layer slicing* to partition bottleneck layers, which refer to the large layers that determine the peak memory usage of the model. The layer slicing technique also selects the most suitable partitioning approach for each layer based on the intra-layer memory bottleneck. The sliced model is accompanied by a tailored *preload-aware memory layout planner* to reduce memory fragments and achieve actual memory saving.

The overhead of memory management is reduced by careful ahead-of-time planning. For example, we adopt static memory management to achieve optimal memory layout and reduce memory allocation costs, pre-transform model weights to avoid runtime processing, preload weights to reduce runtime I/O waiting time, and optionally retain a portion of the model weights in memory to reduce repetitive I/O operations.

Due to the extensive offline model preprocessing and planning, the runtime cost of adapting to the new memory budget is significantly reduced. Each time the memory budget changes, we can reuse the offline-generated profile and weights, and use a lightweight algorithm to re-generate the execution plan. Occasionally, when the original slicing strategy fails to meet the current memory budget, the system might repeat the preprocessing and profiling of models when necessary. Through this, FlexNN is able to handle memory budget dynamicity with minimal overhead while achieving optimal inference latency.

We implemented FlexNN atop NCNN, a popular open-sourced mobile inference framework. Although the design is general, we target the mobile CPU in the current implementation due to its dominance in mobile/embedded AI applications. We evaluated FlexNN on various mobile devices, including smartphones and single-board computers (SBCs) with common model architectures. The results have shown that our approach is able to adapt to different memory constraints with optimal latency-memory trade-offs. For example, FlexNN can reduce the memory consumption by 93.81% with only a 3.64% increase in latency, as compared with the original NCNN on smartphones.

We summarize our main contributions as follows:

(1) We design FlexNN, an efficient and adaptive memory management framework for memory-constrained on-device DNN inference.
(2) We introduce a *slicing-loading-computing joint planning* method that can reduce memory consumption of DNN inference with minimal increase in runtime latency.
(3) We propose a *preload-aware memory planning* scheme that can effectively reduce memory fragments and I/O waiting time during inference.
(4) We implement FlexNN atop NCNN and conduct extensive experiments on various edge devices and DNN models. The results demonstrate that FlexNN is able to adapt to different memory budgets with optimal latency-memory trade-offs and minimal overhead.

2 BACKGROUND AND MOTIVATION

2.1 Limited Memory for On-device DNN Inference

While there have been many existing optimizations focused on reducing the computational cost, memory is the real bottleneck for on-device DNN inference. Unlike computation that may just slow down the inference, memory is often a hard constraint that directly determines whether it is feasible or not to run the model. The memory constraint manifests in the following aspects:

Hardware. In the past decade, the growth in memory capacity on mobile/edge devices has significantly lagged behind the advances in computing power, while the computation and memory requirements of a DNN model usually increase linearly with the model capability [33]. For example, from the iPhone 4 in 2010 to the iPhone 14 in 2022, the GPU performance has increased by nearly 1000 times (in terms of FLOPS), and the CPU performance has improved by around 100 times (based on GeekBench 4 scores [23]), but the RAM capacity has only increased by a factor of 8 [13].

Operating system. Since most mobile/edge devices are not dedicated to single tasks, the operating system may impose memory constraints on individual apps. For example, Android imposes an app-wise heap size limit (e.g., 512MB) and employs a Low-Memory Killer for memory management [7].

Applications. Commercial apps are usually multi-functional and may further limit the memory usage of individual in-app functions including DNN inference, to ensure good performance and smooth user experience. For example, it is reported that the grammar-checker model in Microsoft Editor must use less than 50 MB of memory [42].

To mend the gap between the limited memory budget on edge devices and the increasing memory requirement of DNN models, it is desirable to design an inference framework that takes memory optimization as the first priority.

2.2 Memory Footprint of DNN Inference

We conduct in-depth memory profiling of DNN inference from both inter-layer and intra-layer levels and obtain the following observations:

A. Unbalanced layer-wise memory distribution. Results in Figure 1 indicate that DNN models exhibit a highly unbalanced layer-wise memory distribution. For instance, in ResNet-152, 79.6% layers consume no more than 5MB of memory, and 99.4% layers consume no more than 20MB of memory, while only 0.6% layers consume more than 70MB of memory.

B. Latency-memory trade-off in kernel selection. "Kernels" refer to different implementations of a layer in a DNN. When inferring the same layer, a kernel with lower latency

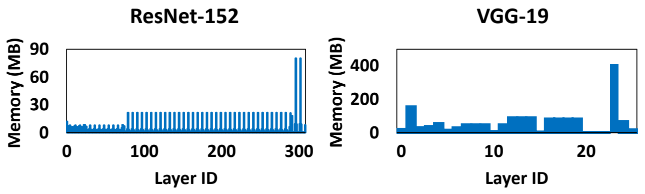


Figure 1: Layer-wise memory usage of DNNs. Few layers have large memory footprints. Layer IDs are consistent with the converted NCNN model format.

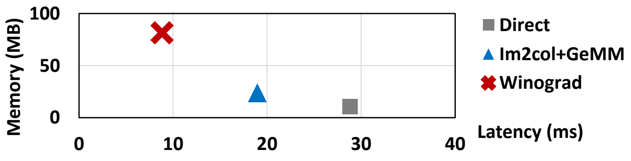


Figure 2: Latency-memory trade-off among 3 kernels of a 3×3 Conv. There is a negative correlation between latency and memory usage.

tends to consume more memory. For example, Winograd, Im2col+GeMM, and Direct Conv are three common kernels of convolutional (Conv) layers [29]. Direct Conv directly calculates the convolution results by definition. Im2col+GeMM flattens the input feature to convert the convolution into a matrix multiplication, which is more hardware-friendly for acceleration. Winograd reduces the runtime computational complexity of 3×3 Conv by transforming weights at the preparation stage. As is shown in Figure 2, Winograd has the lowest latency, but consumes the most memory.

C. Different memory bottlenecks among layers and kernels. The memory footprint of a layer consists of three major parts: activations (i.e., the layer inputs and outputs), weights (i.e., the model parameters), and intermediates (i.e., temporary results when calculating the layer outputs). Specifically, the flattened input (e.g., in Im2col+GeMM Conv) is a common type of intermediate, and the transformed weights (e.g., in Winograd Conv) is a common type of weights. Our profiling results indicate that weights and flattened inputs are two major intra-layer memory bottlenecks. For example, weights typically dominate the fully connected (FC) layers (e.g., by 99.97% in VGG-19's largest FC) and Winograd Conv layers (e.g., by 83.63% in ResNet-152's largest Conv), while the flattened inputs dominate Im2col+GeMM Conv layers (e.g., by 74.75% in VGG-19's 2nd Conv).

2.3 Opportunities

The observations in 2.2 not only cause challenges to DNN's memory management, but also provide opportunities and guide our bottleneck-aware layer slicing design. To address the unbalanced layer-wise memory distribution, it is necessary to conduct intra-layer partitioning to reduce the memory footprint of bottleneck layers. Due to the diverse memory bottlenecks within different layers and kernels, different partitioning approaches should be applied according to layer/kernel characteristics. Besides, since different kernels have different latency-memory trade-offs, it is important to select the kernel that minimizes latency under the given memory budget.

Besides, the unique characteristics of DNN inference workloads enable a better design space for memory swapping. First, the control flow of DNN inference is deterministic as compared with traditional software, making it possible to reduce execution-time management overhead through ahead-of-time planning. Second, the size and lifecycle of tensors during DNN inference exhibit certain patterns, which provide opportunities for optimizing the memory layout.

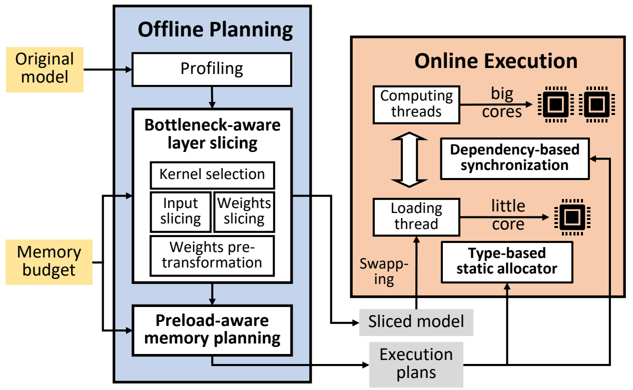


Figure 3: The system overview of FlexNN.

3 FLEXNN DESIGN

3.1 System Overview

As is shown in Figure 3, 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, and the online execution stage that conducts model inference based on offline-generated plans.

The offline planning stage involves two major modules: bottleneck-aware layer slicing and preload-aware memory planning. After profiling tensor-wise memory size and lifecycle, layer slicing is conducted to reduce layer-wise memory footprints. Besides weights slicing and input slicing which are two slicing approaches, the layer slicing module also involves kernel selection and weights pre-transformation to reduce runtime overhead. The preload-aware memory planning is conducted after the layer slicing to provide a detailed memory plan. It uses a lightweight algorithm that reduces fragments and I/O waiting time at runtime, as well

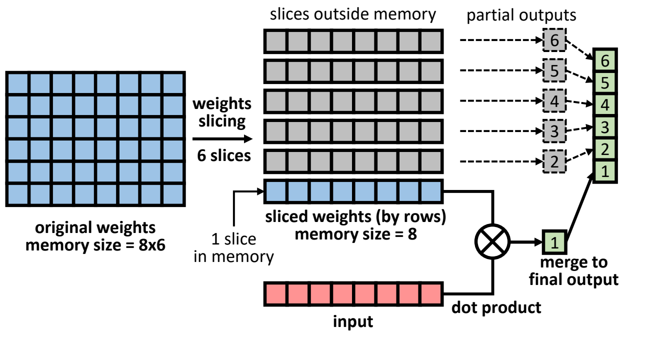


Figure 4: Weights slicing example of 8×6 FC. The weights are partitioned into six slices to save 83% of memory usage in this example. The FC weight size scales with input and output sizes, which can get very large.

as avoiding large adaption overhead. When the memory budget is sufficiently large, FlexNN also fixes a portion of model weights in memory to further reduce the I/O cost.

In the online execution stage, FlexNN conducts model inference efficiently with the offline-generated plans and pre-transformed weights. Specifically, FlexNN uses a dependency-based synchronization scheme and a type-based allocator to ensure the correctness of runtime parallel execution.

Although a complete planning is required at initialization, the layer slicing part is usually skipped in subsequent adaptions to avoid the overhead. When the memory budget changes, FlexNN firstly checks if the previously sliced model satisfies the new memory budget. If so, FlexNN will bypass layer slicing, and only conduct memory planning. Otherwise, the layer slicing is still required to meet the new memory budget.

3.2 Bottleneck-Aware Layer Slicing

The goal of layer slicing is to reduce the memory consumption of each individual layer with fine-grained partitioning, and thus reduce the peak memory consumption. Moreover, our bottleneck-aware layer slicing performs layer-wise peak memory reduction with runtime latency consideration.

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

Flattened inputs and weights are two major bottlenecks of layer-wise memory footprint. To address the different memory bottlenecks of layers and kernels, we introduce two approaches of layer slicing with different latency-memory trade-offs (evaluated in § 5.3): *weights slicing* and *input slicing*, as described below.

Weights slicing partitions layer weights into several slices, and loads one slice each time to reduce the memory footprint. As is shown in Figure 4, the FC weights are partitioned by rows, while the input remains unchanged. The sliced weights are loaded and multiplied with the input slice by slice, producing a partial output for each slice. The partial outputs are merged together to obtain the final output.

The number of slices in weights slicing should be minimized under the memory budget. The reason is that weights slicing divides a large I/O task into multiple smaller ones interleaved with computations, which significantly increases the scheduling overhead when the number of slices is large. We implement weights slicing by splitting one layer into several sub-layers in the computational graph, with each sub-layer carrying one slice of weights.

Input slicing partitions the flattened input (e.g., in Im2col + GeMM Conv), and keeps one slice in memory each time to reduce the memory footprint. In Figure 5, the Im2col-flattened input is partitioned by flattened channels, without changing the weights. The input is flattened and multiplied with the weights slice by slice to produce partial outputs, which are merged in the end to obtain the final output.

The number of slices in input slicing should be maximized within a platform-dependent threshold, which is different from weights slicing. This is because input slicing is achieved by generating a portion of the flattened input each time instead of swapping with the disk, thus avoiding additional I/O scheduling overheads. The total latency will not significantly increase with the number of slices unless the slice size becomes too small to fully utilize hardware acceleration (e.g., SIMD). This imposes a platform-dependent constraint on the maximum number of slices.

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.

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 trade-off in the kernel selection of the same layer, the kernel with optimal latency might not satisfy the memory constraint. Therefore, FlexNN firstly calculates the expected memory footprint of latency-saving kernels after slicing, and switches to memory-efficient kernels if the latency-saving kernel fails to meet the memory constraint.

For example, a 3×3 Conv has multiple available kernels including Direct Conv, Im2col+GeMM Conv, and Winograd Conv. FlexNN firstly tries the Winograd kernel with weights slicing to minimize latency. If failed, FlexNN then tries the Im2col+GeMM kernel with input slicing. If Im2col+GeMM also fails, FlexNN falls back to direct convolution.

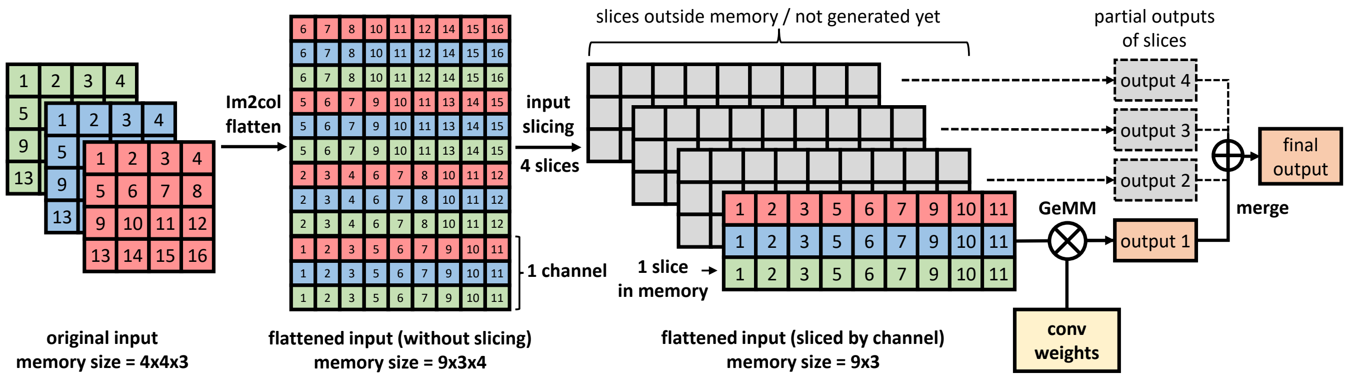


Figure 5: Input slicing example of Im2col+GeMM 3×3 Conv. The Im2col-flattened input is partitioned into four slices to save 75% memory usage in this example. The flattened input size scales linearly with $C_{in} \times W_{out} \times H_{out}$, which can get very large.

Weights pre-transformation is conducted after kernel selection and input/weights slicing, to transform specific weights before execution. The transformation includes not only reshaping or reformatting, but also matrix multiplications for kernels like Winograd Conv, which cause non-negligible overheads. The pre-transformation allows FlexNN to directly load the transformed weights at runtime, and thus avoids the runtime processing overhead.

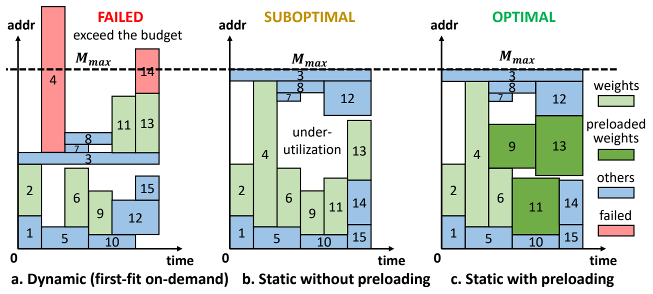


Figure 6: Comparison between different memory management strategies. Our preloading-aware static memory management can simultaneously reduce memory fragments and I/O waiting time. The numbers in tensor blocks represent the order of allocation.

3.3 Preload-Aware Memory Planning

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, while the static memory management strategies in existing works don't take preloading into consideration, leading to suboptimal plans. To address both issues, FlexNN employs static memory management with preload-aware memory planning, thereby reducing both fragments and I/O waiting time (Figure 6). We will firstly formulate the preload-aware memory planning problem, and then introduce our lightweight planning algorithm.

3.3.1 Problem Formulation. The memory layout planning is commonly formulated to the 2D Bin Packing (2DBP) problem with fixed time coordinate [28, 44]. In this formulation, each tensor is abstracted as a rectangle (called "tensor block" in the following formulation) on a two-dimensional plane of memory address (y-axis) and time (x-axis). The x range of the tensor block represents the tensor's *lifecycle* (the logical time, i.e., layer ID from allocation to de-allocation), and the y range represents the memory space it occupies. The goal is to find an optimal layout for a given set of tensors with known lifecycle and memory size.

Our preload-aware memory planning problem can also be formulated as a variant of the 2DBP problem. The main difference is that, in the aforementioned formulation, all tensors have known lifecycle, so the only objective is to decide their memory addresses. However, in our problem, both the allocated address and the allocation time (i.e., the logical time to start preloading) of weights need to be determined by the planning result. For example, in Figure 6.c, weights 9, 11, and 13 start preloading before they are required by computation, and thus have a longer lifecycle than in Figure 6.b, which don't consider preloading. To address this, we formulate our planning problem to a 2DBP problem with half-fixed time coordinate as described below:

Given: (1) A list of n tensors with known properties: the allocation time without preloading (the starting layer IDs) $\{s_i\}$, the de-allocation time (the ending layer IDs) $\{e_i\}$, the memory size $\{m_i\}$, and the memory type $\{type(i)\}$, which includes weights, activations and intermediates. Specifically, activations refer to inputs and outputs of all layers, and intermediates refer to the computational intermediates within individual layers, which is different from activations. Since

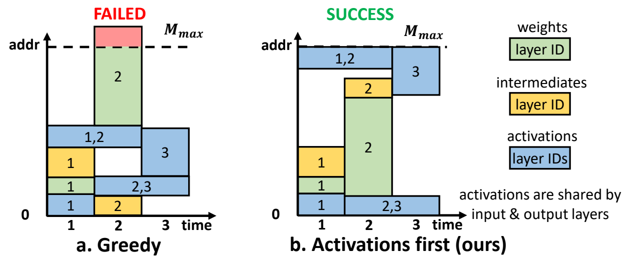


Figure 7: Plan activations first to avoid fragments. The numbers in tensor blocks represent the layer IDs they belong to.

we implement weights slicing by splitting layers into sub-layers, we treat each sub-layer as an individual layer in the memory planning, so the sub-layers have different layer IDs. (2) A memory budget M_{max}.

Objective: Find an optimal memory plan that minimizes the inference latency while satisfying the constraints. The plan is represented as a list of time-address pairs $\Omega = \{\langle time_i, addr_i \rangle\}$, where $time_i$ is the time of tensor i's allocation with preloading and $addr_i$ is the allocated memory address.

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).

3.3.2 *Planning Algorithm.*

We employ the lightweight Algorithm 1, which leverages the memory access patterns of DNN inference, to minimize both memory fragments and I/O waiting time. The algorithm decomposes memory planning in a unified buffer into several steps, based on different tensor types: weights, activations, and intermediates. As is illustrated in Figure 8, the algorithm involves the following key steps:

Plan activations. FlexNN prioritizes planning the activations due to their long lifecycle. Activations have longer lifecycle than other types of tensors since they typically serve as one layer's output and one or multiple layers' inputs. For example, in Figure 7, layer 1's output is layer 2's input, marked as "1,2". As is illustrated in Figure 7, the activation-first planning effectively avoids fragmentation by firstly placing all activations at both ends of the memory buffer, thereby maintaining a continuous block of memory in the middle.

Plan weights with preloading (layer-wise). For each layer, FlexNN plans the weights before intermediates. It greedily searches available memory that minimizes the allocation time of weights, thereby reducing I/O waiting time as much as possible.

Plan intermediates (layer-wise). For each layer, after the weights' layout is determined, FlexNN greedily fills the remaining memory with intermediates.

Re-plan weights and intermediates without preloading (layer-wise). The greedy weights preloading strategy might also cause fragments and lead to intermediate planning failures. Therefore, for each layer, when preload-aware planning fails, FlexNN will re-plan this layer's weights and intermediates without weights preloading.

Algorithm 1 Lightweight memory planning algorithm

```
Input: memory budget $M_{max}$, tensors profiles: allocation
    time $\{s_i\}$, de-allocation time $\{e_i\}$, memory size $\{m_i\}$, mem-
    ory type $\{type(i)\}$
Output: the memory plan $\Omega = \{\langle time_i, addr_i \rangle\}$
  empty plan $\Omega$
  $\Omega \leftarrow$ PlanActivations($\Omega, \{s_i\}, \{e_i\}, \{m_i\}, M_{max}$)
  for $l$ in range(layer count) do
      backup plan $\Omega$
      $\Omega \leftarrow$ PreloadWeights($\Omega, \{s_i\}, \{e_i\}, \{m_i\}, M_{max}$)
      $\Omega \leftarrow$ PlanIntermediates($\Omega, \{s_i\}, \{e_i\}, \{m_i\}, M_{max}$)
      if $\Omega$ fails then
          restore plan $\Omega$
          $\Omega \leftarrow$ PlanNoPreload($\Omega, \{s_i\}, \{e_i\}, \{m_i\}, M_{max}$)
  ▷ Re-schedule weights and intermediates w.o. preloading.
          if $\Omega$ fails then
              return Error.                         ▷ Schedule fails.
          end if
      end if
  end for
  return $\Omega$                                   ▷ Schedule succeeds.
```

3.4 Online Execution Design

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, as described below.

The gap between tensor-wise planning and layer-wise execution. The memory plan determines fine-grained dependencies between tensors, but the runtime conducts a relatively coarse-grained layer-wise inference. Since the execution of one layer involves multiple tensors with different lifecycle, there might be tensor-wise memory conflicts if the loading and computing order of layers is not carefully controlled.

The gap between logical time and actual execution time. The plan determines only the logical time of memory allocation and release for each tensor, but the actual time is unknown.

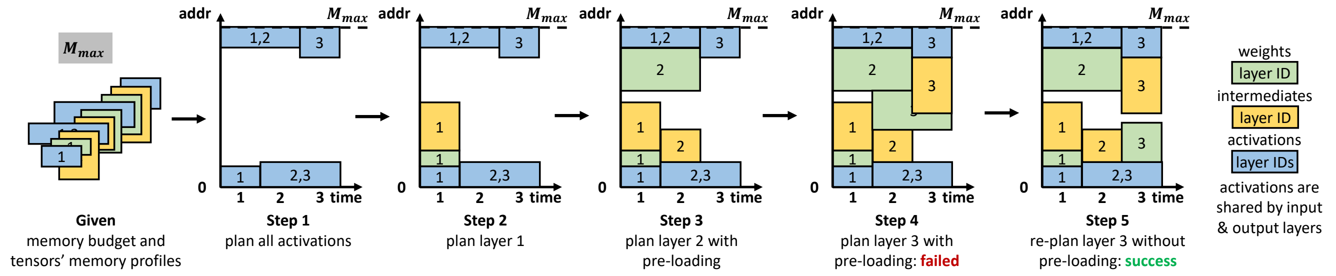

Given
memory budget and tensors' memory profiles
Step 1
plan all activations
Step 2
plan layer 1
Step 3
plan layer 2 with pre-loading
Step 4
plan layer 3 with pre-loading: failed
Step 5
re-plan layer 3 without pre-loading: success
weights
layer ID
intermediates
layer ID
activations
layer IDs
activations are shared by input & output layers

Figure 8: The complete workflow of preload-aware memory planning with a 3-layer example. The numbers in tensor blocks represent the layer IDs they belong to.

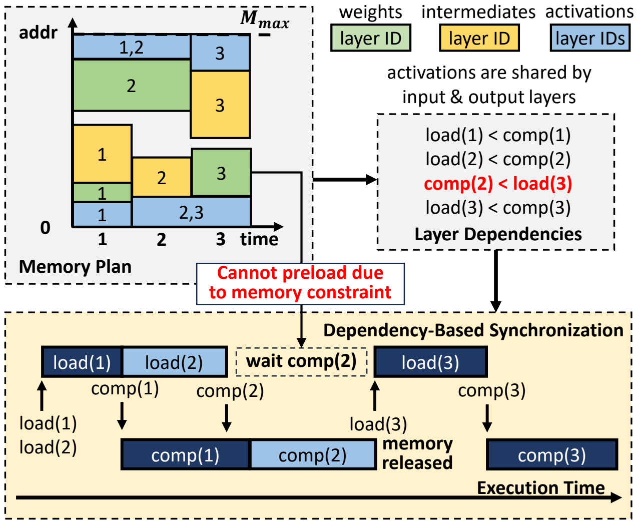


Figure 9: Dependency-based synchronization example of a 3-layer model. The loading of layer 3 depends on the computation of layer 2 since they have an intersection of memory address.

This uncertainty may change the actual allocation order of tensors, and thus cause wrong allocation results.

To mend the gaps, we introduce a *dependency-based synchronization* scheme to schedule the loading and computing of layers, and a *type-based static allocator* to ensure the correctness of allocations. The layer dependency plan for dependency-based synchronization and the memory allocation plan for the type-based allocator are obtained from the memory planning results with simple post-processing.

Dependency-based synchronization. FlexNN captures the memory dependency between layers to convert the tensor-wise memory plan to the layer-wise execution plan. Specifically, we divide the execution of a layer into loading and computing tasks, then manage the execution order of tasks through layer dependencies.

The execution of one layer typically consists of three steps: weights loading, preparation, and computing. Since FlexNN conducts pre-transformation at the offline planning stage, the preparation step can be skipped. We further abstract the weights loading and computing processes of layers to "loading" tasks that conduct only weights loading, and "computing" tasks that conduct all other computations. Both the computing threads and the loading thread are accompanied by a task queue, i.e., the loading task queue and the computing task queue, where they greedily fetch available tasks. Computing and loading threads synchronize through task queue management to satisfy the layer dependencies.

Specifically, there are two types of dependencies at the task level. We denote the loading and computing tasks of layer i as "$load(i)$" and "$comp(i)$" respectively, and use $>$ and $<$ to represent the dependency between tasks, e.g., $load(1) < comp(1)$ means layer 1's computing depends on layer 1's loading. (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. Dependency (1) naturally holds for all layers, while dependency (2) can be resolved from the memory planning results with simple post-processing. Figure 9 shows an example of how our synchronization scheme ensures the execution order through layer dependencies.

Type-based static allocation. The key to ensuring correct memory allocation results at runtime is to define the order of allocations, which may be challenging in multi-thread execution. Fortunately, we observe that each thread will only allocate the memory for certain types of tensors. Specifically, the memory spaces of weights are allocated solely within the loading thread, while the spaces of activations and intermediates are allocated solely within the computing thread. Hence, the memory allocation order within each type of tensor is fixed. Therefore, we uniquely identify each tensor by its type and the count within the type (e.g., `Weights-5`) to ensure the correct order of allocations. The mapping between the assigned tensor ID and the allocated address can be obtained through lightweight post-processing of the memory planning results.

4 IMPLEMENTATION

We implemented a prototype of FlexNN atop NCNN (commit #4594) [29] with 12.3k LoC added. NCNN is an open-source inference engine that has highly optimized kernels on Arm CPUs and supports many real-world mobile apps on the backend, and we target Arm CPUs in the current FlexNN implementation. There are three main reasons for choosing NCNN for the implementation: (1) NCNN is highly optimized for mobile devices, outperforming most other frameworks like TFLite [1]. (2) NCNN supports a good set of models, including CNN, RNN, LSTM, Transformers, etc. (3) NCNN also has good community support and a clear code structure, which provides convenience for our development. Note that our techniques focus on the general process of data loading and computation of DNN inference and thus should be easy to port to other frameworks.

Framework modifications. Except the lifecycle-aware memory planner, which is framework-independent, all other modules require modifications to the inference framework. As for offline planning, we implement layer slicing based on NCNN's model writer tool to modify the computational graph and write transformed weights. In terms of the runtime engine, we implement the parallel preloading pipeline with dependency-based synchronization and modify the native NCNN allocators to a type-based static memory allocator, which changes the original memory management scheme. Specifically, the static allocator allocates a continuous buffer at initialization and manages the buffer during inference. We also modify a few of NCNN's operators. For example, we implement input slicing by adjusting the Im2col+GeMM Conv kernel.

Cooperating with Armv8-A and NEON. Arm CPUs are widely used on various types of edge devices, and the Armv8-A architecture is now the de facto standard on smartphones. Therefore, we target Armv8-A CPUs in our implementation and deployment, although FlexNN's design is compatible with any other architecture. NEON is an advanced SIMD architecture extension that can provide up to 32 128-bit registers in AArch64, the 64-bit execution environment of Armv8-A [6]. Specifically, NEON supports up to 4×32-bit operations per instruction, and NCNN's kernels use basic computing blocks of shapes such as 8×4 and 8×8 with NEON acceleration. Therefore, when slicing layers, we always keep the widths/heights/channels of sliced inputs/weights as multiples of 8 if possible, to fully leverage NEON's acceleration and thus avoid performance degradation.

5 EVALUATION

We evaluate FlexNN on six DNN models and three devices to demonstrate its ability to achieve efficient and adaptive DNN inference with constrained memory.

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

Table 1: Device specifications in our evaluation.

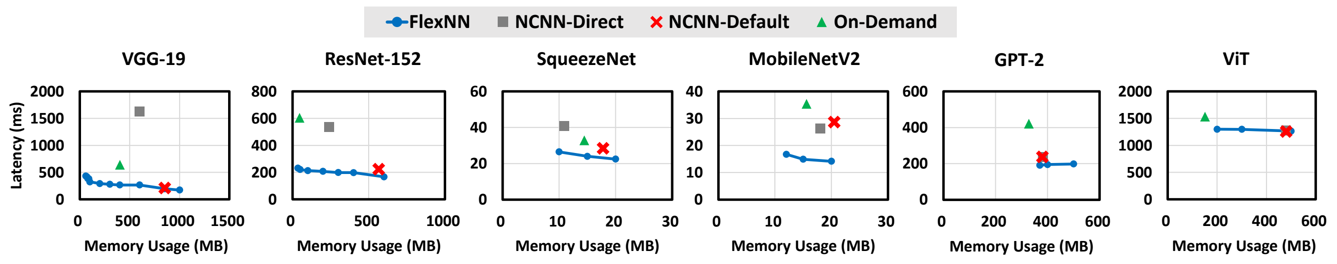
5.1 Experimental Setup

Models. We evaluate FlexNN with six widely-used DNN models, including ResNet-152 [18], VGG-19 [38], Vision Transformer (ViT) [10], GPT-2 [32], MobileNetV2 [36] and SqueezeNet [21]. As FlexNN doesn't modify the model weights, we use the pre-trained models (FP32) and random inputs in our experiments.

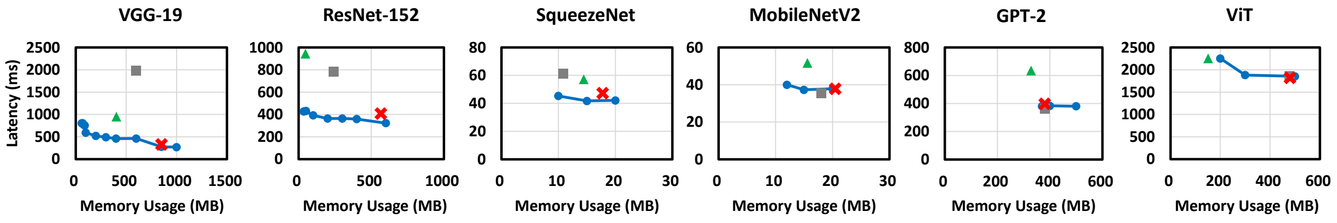
Platforms. We conduct the evaluations on two types (single-board computers and smartphones) of edge devices with different hardware specifications, all equipped with Armv8-A CPUs. The detailed specifications are listed in Table 1. We always use the big cores for computing in our evaluations. Specifically, we use two big cores on Pixel 6 Pro, four big cores on Mix 2S, and three cores on Raspberry Pi. As for loading, we use one middle core on Pixel 6 Pro and one little core on Mix 2S. Since Raspberry Pi doesn't have little cores, one big core is reserved for loading.

Metrics. The measurements cover most of the metrics that indicate the actual performance of the system in real-world applications, including memory usage, inference latency, and energy consumption. The memory usage is measured via the Linux *pmap* command, and the real-time power consumption is estimated by multiplying the battery's current and voltage, which can be obtained through the `sysfs` interface in Linux.

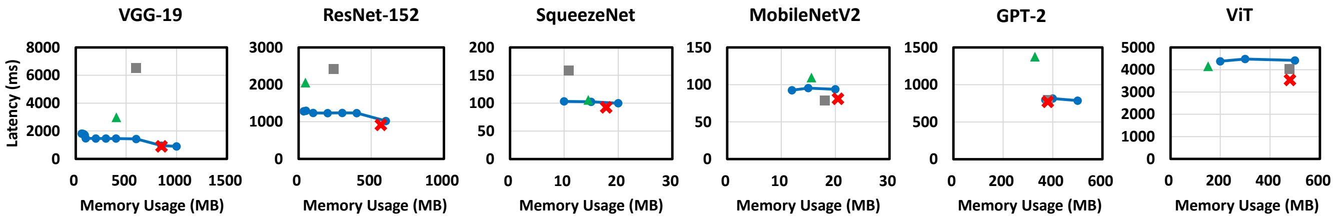
Baselines. All of our baselines are based on NCNN for two main reasons. (1) We implement FlexNN atop NCNN, which is indeed one of the best performing mobile frameworks as aforementioned in Section 4. (2) To the best of our knowledge, FlexNN is the first mobile inference framework to take memory as a priority, so there are no similar works to compare with. We use the NCNN with the default configuration (denoted as "NCNN-Default") as one of our baselines. One way to reduce the memory consumption in NCNN is to disable the Im2col+GeMM and Winograd kernels, both of which would increase memory usage for lower latency. We call this baseline as "NCNN-Direct". Another common strategy to reduce memory footprint is layer-wise swapping and on-demand loading. We also implement this strategy atop NCNN and include it as a baseline, denoted as "On-Demand". All baselines use the same number of big cores as FlexNN for computing. The little cores are not used in baselines, since



(a) Google Pixel 6 Pro



(b) Xiaomi Mix 2S



(c) Raspberry Pi 4B

Figure 10: End-to-end latency and memory results on different devices and models.FlexNN's curve on the down-left side indicates the trade-offs under different memory budgets, while the other points of the baselines provide no trade-offs. Therefore, FlexNN achieves better latency-memory trade-offs.

the unbalanced load would slow down the inference (e.g., by up to 48% on Mix 2S) in NCNN if the big cores and little cores are used together.

5.2 End-to-end Performance

The end-to-end evaluation covers three edge devices and 6 DNN models of various types. In Figure 10, for each device and model, the results of FlexNN are represented by a curve, in which each data point represents the memory and latency achieved by FlexNN under a memory constraint, while the baselines are represented by individual data points as they have fixed memory usage. We analyze the results from the following aspects.

Peak memory reduction. The left end of a curve represents the minimum memory budget enabled by FlexNN. FlexNN supports lower memory budgets than NCNN-Default and NCNN-Direct on all models, and lower than On-Demand in most cases. For instance, when compared to NCNN-Default, NCNN-Direct, and On-Demand on Pixel 6 Pro, FlexNN reduces the memory usage by up to 92.95%, 90.01%, and 85.13% respectively on VGG-19, and by up to 93.81%, 85.45%, and 25.21% respectively on ResNet-152. As a straightforward way of streaming, On-Demand requires less memory than other baselines for most models. Due to our layer slicing techniques, FlexNN reduces the layer-wise memory footprint, further reducing the overall memory usage compared to the On-Demand strategy.

Latency reduction. In addition to reducing memory usage effectively, FlexNN also achieves acceptable inference latency. Under the same memory budgets, FlexNN consistently achieves lower or at least comparable latency to the baselines, especially under low memory budgets. For example, on Pixel 6 Pro, compared to NCNN-Default, NCNN-Direct, and On-Demand, FlexNN reduces the latency by approximately 6.37%, 83.63%, and 58.33% on VGG-19, and by approximately 11.08%, 61.11%, and 63.33% on ResNet-152.

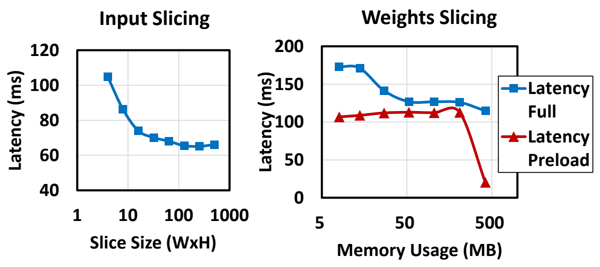


Figure 11: Layer-wise latency and memory trade-off of layer slicing measured on Pixel 6 Pro. Weights slicing is tested on an FC, and input slicing is tested on a 3×3 Conv. Memory usage of input slicing is proportional to the slice size. "Latency Full" represents latency without preloading.

FlexNN maintains latency comparable to NCNN-Default even in low-memory situations through pre-transforming and preloading weights. For instance, it achieves a 93.81% peak memory reduction only at the cost of a 3.64% latency increase on ResNet-152, Pixel 6 Pro.

It is noticed that FlexNN even outperforms NCNN-Default when the memory budget is sufficiently large. This is because FlexNN can fix all weights in memory when given the same memory budget as NCNN-Default, which entirely avoids the I/O cost. Then other latency-reduction designs including pre-transformation and static memory management can further reduce the latency.

Variance across models and devices. FlexNN achieves good performance on CNNs of diverse model sizes, but the improvement on transformer-based models is not significant. It is also noticed that the performance of FlexNN on Raspberry Pi is slightly poorer than on the other devices. Both two observations originate from the change of system bottleneck. On-device DNN inference is bound by computing in most cases, so that FlexNN can hide the loading time with computing time. However, when the model is dominated by I/O (e.g., Transformers) or the device experiences I/O performance bottlenecks (e.g., the Raspberry Pi has an average reading speed of 257 MB/s, compared to 718 MB/s on Mix 2S) , the inference task becomes I/O-bound, which prevents further reduction in latency.

5.3 Latency and Memory Trade-off

In addition to achieving better overall latency and memory compared to the baselines, FlexNN itself exhibits a latency and memory trade-off. As shown in Figure 10, when the memory budget increases, FlexNN effectively utilizes the spare memory to reduce latency. This trade-off comes from two aspects: (1) With a larger memory budget, FlexNN can use a larger slice size and faster computational kernels to

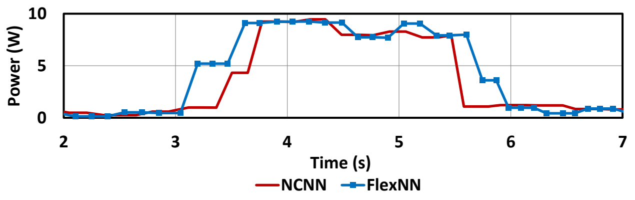


Figure 12: Real-time power consumption of FlexNN and NCNN on Pixel 6 Pro, both inferencing VGG-19 for 8 loops.

obtain layer-wise acceleration. (2) For a given slicing strategy and kernel selection, a larger memory budget allows FlexNN to preload more weights, resulting in model-wise acceleration.

Figure 11 further demonstrates this trade-off using a single-layer example. On one hand, both input slicing and weights slicing exhibit lower latency when more memory is available. For input slicing, as the slice size increases, the decrease in latency becomes smaller. Hence, a fixed slice size (e.g., 32 in our implementation) is chosen to leave more memory space for preloading. For weights slicing, the latency without preloading ("Latency Full") noticeably increases as memory decreases. However, when preloading is performed, the latency ("Latency Preload") remains almost unchanged as long as the computing time could be hidden by the loading time.

5.4 System Overhead

The overhead of FlexNN consists of two major parts: offline planning overhead and runtime preloading overhead. We measure the system overheads and discuss them separately in this subsection.

Offline planning overhead. The offline planning overhead includes storage for transformed weights, time for profiling, time for layer slicing, and time for memory planning. We measure each term and list them in Table 2. On most edge devices, there is sufficient storage capacity (typically ranging from tens to hundreds of GB) to accommodate the transformed weights. Regarding the time overhead, profiling and layer slicing are typically performed only at initialization, and only memory planning needs to be repeated each time the memory budget changes. Consequently, the time overhead for adaptation typically remains within one second, which satisfies most scenarios. However, the extra cost of re-slicing and re-planning is required when the slicing result fails to meet the new memory budget or when there is an allocation error, which should occur rarely as our approach effectively controls the memory usage.

Runtime I/O overhead. The runtime overhead of FlexNN mainly comes from the additional I/O to load model weights. Since we have made an end-to-end comparison of latency in § 5.2, we will focus on the energy cost here. According to

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

Table 2: Storage and time overheads of FlexNN. Profiling and layer slicing can be bypassed, while memory planning (within 1s) is required each time the memory budget changes.

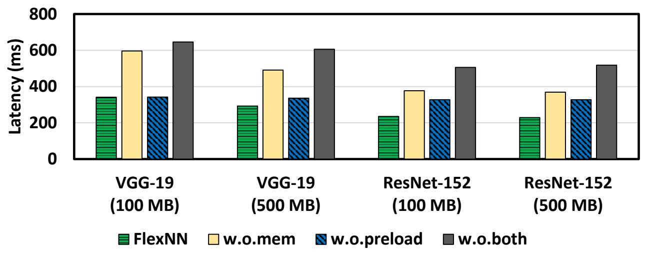


Figure 13: Ablation study. "w.o.mem" stands for FlexNN without static memory management; "w.o.preload" stands for FlexNN without parallel preloading; "w.o.both" stands for FlexNN without both components.

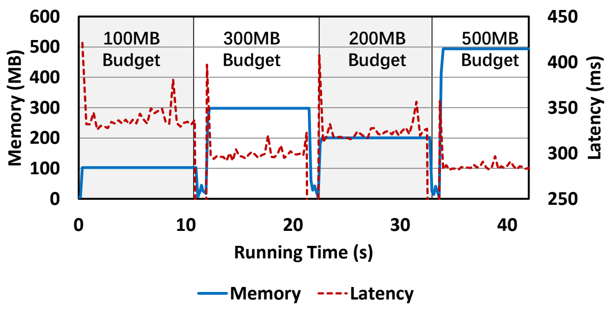


Figure 14: Real-time latency and memory usage of FlexNN under changing memory budgets. FlexNN conducts 32 times of inference on VGG-19 for each memory budget.

Figure 12, the peak power consumption of FlexNN is similar to NCNN, but the larger inference latency of FlexNN results in higher total energy consumption. Specifically, FlexNN increases the energy consumption on Pixel 6 Pro by 1.02% on ResNet-152, and by 28.39% on VGG-19. The additional energy cost is acceptable considering the highly limited memory.

5.5 Ablation Study

We also conduct an ablation study to separately demonstrate the impact of static memory management and parallel preloading on reducing latency. "w.o.mem" replaces FlexNN's static memory allocator with NCNN's native memory allocator, while "w.o.preload" replaces FlexNN's preloading with on-demand loading. "w.o.all" encompasses both modifications. The results in Figure 13 demonstrate that both the static memory management and parallel preloading effectively reduce inference latency under given memory budgets.

5.6 Adapting to Memory Budget Changes

We also implement a demo to demonstrate the real-time performance of FlexNN under varying memory budgets. Figure 14 shows the real-time memory usage and inference latency of FlexNN when sequentially adapting to four different memory budgets. When the memory budget changes, FlexNN pauses the inference process, releases memory, generates an execution plan for the new memory budget, reallocates memory, and finally resumes the inference. The entire adaptation process takes approximately 1 second in total. As is shown in Figure 14, FlexNN accurately meets the memory budgets and achieves smaller inference latency under larger memory budgets. Since there is no warm-up, the first loop after each adaptation has a longer latency due to the cold start overhead. Note that in the real-world deployment, we expect the memory budget changes and adaption strategy to be controlled by the developer, which is not the focus of this paper.

6 DISCUSSION

We hereby discuss the applicability and effectiveness of FlexNN across other scenarios.

Different memory bottleneck. FlexNN adaptively deals with the cases when the weights or inputs are the layer-wise memory bottleneck, which is common across most models. Its effectiveness may be limited when the activation becomes the major memory bottleneck (e.g., when the weights/inputs have been highly compressed), because our current design does not support activation slicing, and the input slicing in § 3.2 is now restricted to flattened inputs such as in Im2col+GeMM. This issue can be mitigated by further supporting activation partitioning and swapping, which we leave for future work.

Different models. Our current implementation is mainly focused on reducing the memory footprint of typical CNNs (such as VGG, ResNet, and MobileNet) due to the dominance of CNNs on mobile devices. Although we have tested FlexNN on Transformer-based models including ViT and GPT-2, their results are less significant. There are two main reasons. (1) The layer slicing support for the Multi-head Attention (MHA) in Transformer-based models is not as well as the convolutional layers, which requires non-trivial engineering efforts to improve in future work. (2) The inference of Transformer-based models requires more intensive weights loading as compared to CNNs, which brings larger I/O overhead in the streaming-based inference. Therefore, it is more challenging to apply the memory swapping mechanism to these models. Nevertheless, our slicing and memory management design is applicable to all types of models.

Different precisions. Though we only implement and evaluate FlexNN with FP32-based models now, FlexNN's design is agnostic to data precision. It does not require modification of the overall framework to add support for other precision, while the transplantation of operators still entails certain amounts of engineering effort. In general, FlexNN possesses the ability to be compatible with compressed models.

Different hardware. The current implementation only targets mobile CPUs due to its dominance in mobile/embedded AI applications, but it is feasible to migrate the implementation to different backends including mobile GPU, NPU, DSP, and Tensor Cores, as the slicing and joint-planning design of FlexNN is generic and doesn't rely on specific hardware like CPU. However, the following factors should be taken into consideration for the migration and real-world deployment. (1) Our approach involves fine-grained memory management, which might require low-level APIs to support the implementation. (2) The additional data movement cost of utilizing heterogeneous hardware should be considered, as the streaming-based inference design brings additional I/O of model weights. (3) Higher energy consumption of FlexNN might be a limiting factor for battery-operated devices.

7 RELATED WORK

System support for memory-constrained inference. Although the number of studies focusing on streaming inference on mobile devices is limited, there are other types of relevant research. A typical line of work [20, 31, 35, 49] reduces the GPU memory requirement through swapping with the main memory on cloud or edge servers. FlexGen [37] further introduces disk swapping to support large generative models on a single GPU. Occlumency [26] conducts layer partitioning and streaming to fit CNNs into the limited TEE memory. Melon [44] reduces the memory footprint of on-device training through offline planning with a lifetime-aware memory pool. These works lack joint planning of layer partitioning, memory layout, and computing-loading overlapping, and directly applying these approaches in our scenario may lead to suboptimal performance. Nevertheless, they offer valuable insights on memory optimization for edge devices.

Model customization for memory-constrained inference. In order to support DNN inference under resource constraints, numerous works have been devoted to model customization techniques, including model compression [4, 17, 19, 27], efficient model structure design [21, 36, 41, 52] and neural architecture search (NAS) [5, 34, 40, 47, 50]. Pruning reduces the number of model parameters by selectively removing unimportant connections, thereby reducing computation and memory requirements. Similarly, quantization reduces the model size and computation load by using low-precision representations for its parameters. Meanwhile, hardware-aware NAS automatically searches the most efficient model architecture for given platforms and tasks. To address the challenge of memory budget dynamicity on mobile devices, researchers have also proposed to dynamically scale the model on the device [11, 16, 45]. While these techniques effectively reduce the computation and memory requirements of the model, they may also compromise the model's capacity and robustness. Meanwhile, these approaches and ours are orthogonal. Running the customized models with FlexNN can further reduce the memory footprint.

8 CONCLUSION

In this paper, we designed and implemented FlexNN, an efficient and adaptive memory management framework for memory-constrained on-device DNN inference. FlexNN achieves optimal memory utilization and minimal memory management overhead through a slicing-loading-computing joint planning approach. Our evaluation results have shown that FlexNN is able to adapt to different memory budgets with optimal latency-memory trade-offs and minimal adaption overhead.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China (No.2022YFF0604501), NSFC (No.62272261 and 62302262), Xiaomi Foundation, and a grant from the Guoqiang Institute, Tsinghua University.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,

Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow, Large-scale machine learning on heterogeneous systems. https://doi.org/10.5281/zenodo.4724125

[2] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. 2021. Deep learning for network traffic monitoring and analysis (NTMA): A survey. Computer Communications 170 (2021), 19–41.

[3] Zeeshan Ahmad, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, and Farhan Ahmad. 2021. Network intrusion detection system: A systematic study of machine learning and deep learning approaches. Transactions on Emerging Telecommunications Technologies 32, 1 (2021), e4150.

[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. arXiv preprint arXiv:1908.09791 (2019).

[5] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. arXiv preprint arXiv:1812.00332 (2018).

[6] developer.arm.com. 2023. NEON and Floating-Point architecture. https://developer.arm.com/documentation/den0024/a/AArch64-Floating-point-and-NEON/NEON-and-Floating-Point-architecture, accessed: 2023-08-25.

[7] Android Developers. 2023. Manage your app's memory. https://developer.android.com/topic/performance/memory, accessed: 2023-08-01.

[8] MACE Developers. 2023. XiaoMi/mace: MACE is a deep learning inference framework optimized for mobile heterogeneous computing platforms. https://github.com/XiaoMi/mace, accessed: 2023-08-01.

[9] Changxing Ding and Dacheng Tao. 2015. Robust face recognition via multimodal deep face representation. IEEE transactions on Multimedia 17, 11 (2015), 2049–2058.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929 (2020).

[11] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (2018).

[12] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. 2019. Deep learning-based image recognition for autonomous driving. IATSS research 43, 4 (2019), 244–252.

[13] GadgetVersus. 2023. Apple iPhone 4 vs Apple iPhone 14 Benchmarks, Specs, Performance Comparison and Differences. https://gadgetversus.com/smartphone/apple-iphone-4-vs-apple-iphone-14/, accessed: 2023-08-01.

[14] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. Journal of Field Robotics 37, 3 (2020), 362–386.

[15] Abhishek Gupta, Alagan Anpalagan, Ling Guan, and Ahmed Shaharyar Khwaja. 2021. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. Array 10 (2021), 100057.

[16] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y. Chen. 2021. LegoDNN: Block-Grained Scaling of Deep Neural Networks for Mobile Vision. In Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (New Orleans, Louisiana) (MobiCom '21). Association for Computing Machinery, New York, NY, USA, 406–419. https://doi.org/10.1145/3447993.3483249

[17] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015).

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.

[19] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In Proceedings of the European conference on computer vision (ECCV). 784–800.

[20] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 1341–1355.

[21] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv:1602.07360 (2016).

[22] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In MLSys.

[23] Poole John. 2016. Geekbench 4. https://www.geekbench.com/blog/2016/08/geekbench-4/, accessed: 2023-08-06.

[24] Sampo Kuutti, Richard Bowden, Yaochu Jin, Phil Barber, and Saber Fallah. 2020. A survey of deep learning applications to autonomous vehicle control. IEEE Transactions on Intelligent Transportation Systems 22, 2 (2020), 712–733.

[25] Donghwoon Kwon, Hyunjoo Kim, Jinoh Kim, Sang C Suh, Ikkyun Kim, and Kuinam J Kim. 2019. A survey of deep learning-based network anomaly detection. Cluster Computing 22 (2019), 949–961.

[26] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In The 25th Annual International Conference on Mobile Computing and Networking. 1–17.

[27] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. 2019. Metapruning: Meta learning for automatic neural network channel pruning. In Proceedings of the IEEE/CVF international conference on computer vision. 3296–3305.

[28] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 123–137.

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

[30] Omkar Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep face recognition. In BMVC 2015-Proceedings of the British Machine Vision Conference 2015. British Machine Vision Association.

[31] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 891–905.

[32] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask

learners. OpenAI blog 1, 8 (2019), 9.

[33] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 446–459.

[34] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. ACM Computing Surveys (CSUR) 54, 4 (2021), 1–34.

[35] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 1–13.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4510–4520.

[37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput generative inference of large language models with a single gpu. arXiv preprint arXiv:2303.06865 (2023).

[38] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[39] Yi Sun, Ding Liang, Xiaogang Wang, and Xiaoou Tang. 2015. Deepid3: Face recognition with very deep neural networks. arXiv preprint arXiv:1502.00873 (2015).

[40] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2820–2828.

[41] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning. PMLR, 6105–6114.

[42] Ge Tao, Ting Cao, Si-Qing Chen, and Qiong(Emma) Ning. 2023. Achieving Zero-COGS with Microsoft Editor Neural Grammar Checker. https://www.microsoft.com/en-us/research/blog/achieving-zero-cogs-with-microsoft-editor-neural-grammar-checker/, accessed: 2023-08-11.

[43] Mei Wang and Weihong Deng. 2021. Deep face recognition: A survey. Neurocomputing 429 (2021), 215–244.

[44] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services. 450–463.

[45] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Ya-Qin Zhang, and Yunxin Liu. 2023. AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments. (2023).

[46] Tsu William. 2020. Introducing NVIDIA HGX A100: The Most Powerful Accelerated Server Platform for AI and High Performance Computing. https://developer.nvidia.com/blog/introducing-hgx-a100-most-powerful-accelerated-server-platform-for-ai-hpc/, accessed: 2023-08-25.

[47] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 10734–10742.

[48] Yang Xing, Chen Lv, Huaji Wang, Dongpu Cao, Efstathios Velenis, and Fei-Yue Wang. 2019. Driver activity recognition for intelligent vehicles: A deep learning approach. IEEE transactions on Vehicular Technology 68, 6 (2019), 5379–5390.

[49] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. 2019. Efficient memory management for gpu-based deep learning systems. arXiv preprint arXiv:1903.06631 (2019).

[50] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. 2020. Fast hardware-aware neural architecture search. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. 692–693.

[51] Weibin Zhang, Yinghao Yu, Yong Qi, Feng Shu, and Yinhai Wang. 2019. Short-term traffic flow prediction based on spatio-temporal analysis and CNN deep learning. Transportmetrica A: Transport Science 15, 2 (2019), 1688–1711.

[52] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the IEEE conference on computer vision and pattern recognition. 6848–6856.