TinyMem: Boosting Multi-DNN Inference on Tiny AI Accelerators with Weight Memory Virtualization

Changmin Jeon*
Seoul National University
Seoul, Republic of Korea
wisechang1@snu.ac.kr

Taesik Gong*
UNIST
Ulsan, Republic of Korea
taesik.gong@unist.ac.kr

Juheon Yi
Nokia Bell Labs
Cambridge, United Kingdom
juheon.yi@nokia-bell-labs.com

Fahim Kawsar
Nokia Bell Labs & University of Glasgow
Cambridge, United Kingdom
fahim.kawsar@nokia-bell-labs.com

Chulhong Min
Nokia Bell Labs
Cambridge, United Kingdom
chulhong.min@nokia-bell-labs.com

Abstract

As wearable devices continue to integrate deeper into our everyday lives, the importance of tiny AI accelerators in enabling efficient multi-DNN inference becomes increasingly evident. However, we identified a critical bottleneck in deploying multi-DNN models on these accelerators: weight memory loading time. This challenge is exacerbated by the unique characteristics of tiny AI accelerators, including a 2-dimensional weight memory layout and heterogeneous processors.

We propose TinyMem, a memory-efficient system designed to optimize weight memory management and reduce latency in multi-DNN inference to address these challenges. At the core of TinyMem is the *2D Weight Memory Coordination*, which strategically aligns weights across accelerator cores, maximizing both weight reuse and preloading to minimize end-to-end latency. TinyMem increases total model throughput by 1.97-5.01× compared to state-of-the-art tiny AI accelerator frameworks, significantly improving the efficiency of multi-DNN inference in wearable devices.

CCS Concepts

• **Computer systems organization** → **Embedded software**; • **Computing methodologies** → *Planning with abstraction and generalization.*

Keywords

Tiny AI Accelerator, Memory Management, Multi-DNN Inference

ACM Reference Format:
Changmin Jeon, Taesik Gong, Juheon Yi, Fahim Kawsar, and Chulhong Min. 2025. TinyMem: Boosting Multi-DNN Inference on Tiny AI Accelerators with Weight Memory Virtualization. In *The 26th International Workshop on Mobile Computing Systems and Applications (HOTMOBILE '25), February 26–27, 2025, La Quinta, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3708468.3711888

*This work was done while the authors were affiliated with Nokia Bell Labs.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

HOTMOBILE '25, February 26–27, 2025, La Quinta, CA, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1403-0/25/02
https://doi.org/10.1145/3708468.3711888

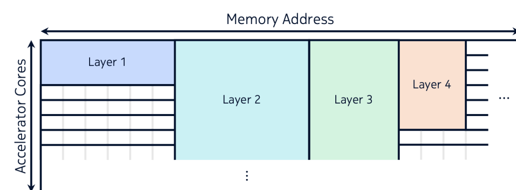


Figure 1: 2D weight memory layout in tiny AI accelerators. Each layer occupies a rectangular region.

1 Introduction

The advent of tiny AI accelerators (e.g., Analog MAX78000 [3], Greenwaves GAP-9 [17]) has brought significant advancements in enabling efficient deep neural network (DNN) execution on resource-constrained microcontroller units (MCUs). They are designed to offload computationally intensive tasks to dedicated hardware, providing substantial acceleration for DNN inference while maintaining low power consumption. Because of their small form factor (e.g., MAX78000: 8mm×8mm), these tiny AI accelerators are increasingly being integrated into wearable and IoT devices to support various AI-driven applications [2, 13, 15]. We envision that multi-DNN inference will be a key requirement to support concurrent app services (e.g., audio keyword spotting, PPG heart rate detection, and IMU action recognition on a single earbud). For instance, healthcare systems like the Galaxy Ring [14] simultaneously monitor multiple vital signs, such as heart rate and blood oxygen levels, by leveraging advanced DNN models, while sensor-based IoT systems analyze contextual information and user behavior in real-time to adapt dynamically to their environment [11].

Despite these benefits, tiny AI accelerators still face limitations in efficiently supporting multi-DNN inference. The key challenge is the unique hardware characteristics of tiny AI accelerators. For acceleration, these accelerators often have dedicated memory and accelerator cores for model inference. Each core is equipped with its own dedicated weight memory space to ensure efficient parallel processing and avoid memory contention. However, as these are mostly designed for optimizing a single model, on-accelerator memory is typically too small to hold multiple models simultaneously (e.g., 442 KB of weight RAM on the accelerator in MAX78000). Furthermore,

weights for each layer must maintain the same offset across all cores, simplifying hardware design but imposing additional constraints on memory allocation. This constraint forces models to be loaded and unloaded repeatedly when running multiple DNNs. This repeated loading and unloading operation is costly due to high data transfer overhead between the CPU's RAM and the accelerator's weight memory, which significantly impacts end-to-end latency for multi-DNN inference.

By investigating the operational characteristics of tiny AI accelerators, we found unexplored opportunities to boost multi-DNN inference throughput by reducing the loading operations for model weights: weight preservation and weight preloading. (a) **Weight preservation**: Tiny AI accelerators often have multiple accelerator cores to parallelize processing. To accelerate execution, memory on these accelerators has a 2-dimensional layout (Figure 1) that assigns dedicated weight memory to each accelerator core to minimize memory management overhead (details in §2.3), thereby allowing parallel access to memory in addition to parallel processing. While this architecture is effective for accelerating a single DNN inference, it also leaves a lot of unused space. In multi-DNN inference, this unused space allows other models to preserve their weights even when not being processed, thereby avoiding the need to reload preserved parts when switching between models. (b) **Weight preloading**: Since AI accelerators have dedicated processors, the CPU becomes idle while model inference is running on the accelerator cores. In multi-DNN inference scenarios, the CPU can use this idle time to preload the next model's weights. However, due to interdependencies between models, it is almost infeasible for developers to manually handle these operations and achieve optimal performance without system-level optimization.

To address these challenges, we propose TinyMem, a system designed to boost multi-DNN inference on tiny AI accelerators. TinyMem virtualizes the weight memory within the accelerator, optimizing the management of model weights to reduce latency. When models are added to TinyMem, it dynamically analyzes the layout of their weight memory and holistically orchestrates weight preservation and preloading operations to minimize weight loading cost. To further enhance performance, TinyMem incorporates **weight packing**, which effectively packs model weights in a 2-D layout. To make holistic decisions, TinyMem employs a joint optimizer that considers the combined effects of these techniques.

We evaluated the effectiveness of TinyMem using the Analog MAX78000 with five models. Our experimental results show that TinyMem increases total model throughput by 1.97-5.01× compared to state-of-the-art tinyAI accelerator frameworks.

2 Background and Motivation

2.1 TinyML and AI Accelerators

TinyML is a field focused on deploying AI models on resource-limited MCUs to achieve power efficiency, low latency, and privacy. Memory is the primary bottleneck, and much of the research centers on reducing model size through pruning, quantization, and neural architecture search [10]. Specialized tiny AI accelerators, such as Analog MAX78000 [3], Greenwaves GAP-9 [17], and Google Coral Micro, have been developed to improve efficiency by offloading computation to dedicated hardware. For instance, MAX78000

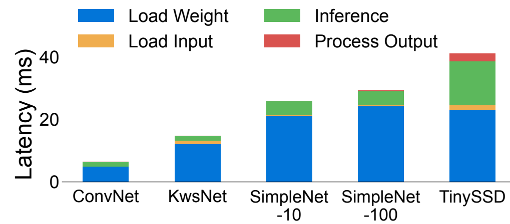


Figure 2: DNN inference latency breakdown. The weight loading time dominates the inference latency.

features a neural network accelerator with 64 parallel cores and dedicated memory for weights and inputs. A recent benchmark [5] showed that MAX78000 reduces inference latency for face detection and keyword spotting by 54.3× and 61.5× compared to the STM32F7 with Cortex-M7 at 216 MHz [16]. This paper focuses on Analog MAX78000 [3], one of the most widely used tiny AI accelerators in research [5, 6, 12].

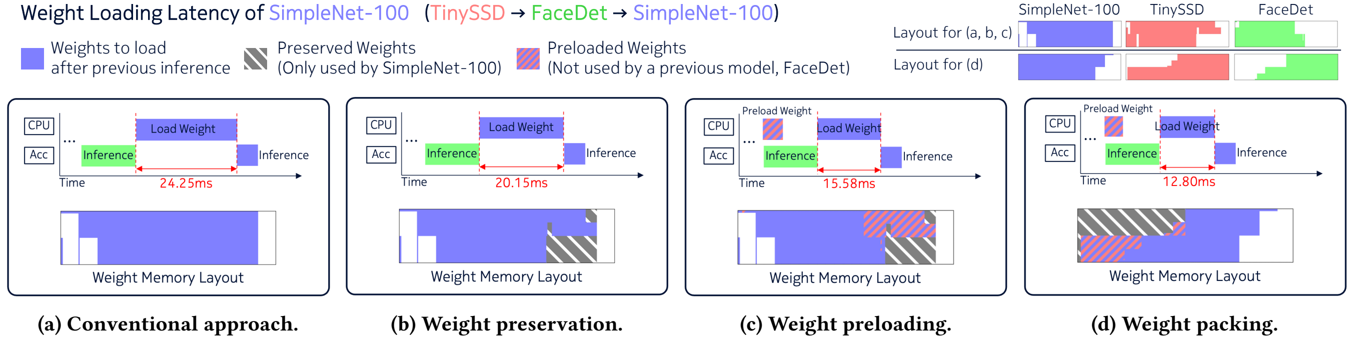
2.2 Limitations in Multi-DNN Inference

While tiny AI accelerators enable extremely efficient DNN execution, their support for multi-DNN inference is still limited due to memory constraints. A few recent studies [4, 8, 9] designed solutions for mobile devices and MCUs, focusing on reducing weight loading overhead, but lack generality as they require model retraining to share weight data values across multiple models. In addition, they do not account for the unique characteristics of tiny AI accelerators. To better understand this performance bottleneck, we conducted an exploratory study. Figure 2 illustrates the end-to-end latency breakdown for various models on MAX78000. "Load weight" and "Load input" refer to loading model weights and input data from SRAM to the AI accelerator's memory. The results show the significant acceleration of MAX78000 for a single DNN inference due to fast processing on input data loading, inference, and output processing. Note that model weights need to be loaded only once during the initial run.

However, the results also highlight inefficiencies in multi-DNN execution on AI accelerators. Due to limited on-accelerator memory, only one model can be supported at a time, requiring frequent loading and unloading of model weights. For instance, MAX78000's 442 KB of weight RAM is significantly smaller than the 141 GB on a Cloud GPU (NVIDIA H100) or the 8 GB on a Mobile GPU (iPhone 15). When an application needs to continually run both SimpleNet-100 and TinySSD sequentially, which cannot be loaded simultaneously on MAX78000, their weights must be loaded and unloaded repeatedly. As shown in Figure 2, the significant acceleration of inference makes the memory operations for model weights a critical bottleneck. For example, while separate inferences of SimpleNet-100 and TinySSD achieve throughputs of 186 and 40 inferences per second, respectively, running them sequentially in an alternating manner reduces throughput to 13, which is significantly lower than the average throughput.

2.3 Weight Memory of Tiny AI Accelerators

To explore optimization opportunities for multi-DNN inference on tiny AI accelerators, we investigate the characteristics of weight



(a) Conventional approach. (b) Weight preservation. (c) Weight preloading. (d) Weight packing.

Figure 3: Design insights for TinyMem.

memory on tiny AI accelerators. One distinct feature of memory in tiny AI accelerators is the 2-dimensional layout that assigns dedicated memory to each accelerator core, unlike the 1-dimensional space commonly found in CPUs and GPUs. Each core is equipped with its own dedicated weight memory space to avoid memory contention and ensure efficient parallel processing. Furthermore, for each layer, weights must maintain the same offset across all cores, which simplifies the memory controller design and allows synchronized operations across the cores.

For powerful parallel processors, weights are typically loaded into a 1-dimensional space first and then managed by internal memory controllers that distribute them to the processor's dedicated memory. However, tiny AI accelerators require a more direct and efficient approach, loading weights directly into a 2-dimensional space to minimize overhead. Figure 1 depicts this 2-dimensional layout. For each layer, the corresponding weight memory is allocated in a 2D format, where one axis represents the parallel accelerator cores required for processing the weights, and the other represents the address space. This 2D memory layout brings several performance benefits. It reduces the number of registers required for configuring AI accelerators, simplifies the internal circuit design, and results in a more efficient hardware architecture. Weights can also be loaded in parallel by allocating dedicated memory to each core before executing each layer, which is beneficial for channel or weight-wise parallelism during inference.

However, despite these benefits, it reveals that tiny AI accelerators do not yet optimize their memory usage, even though memory is a key constraint of these devices. Since weight memory is assigned as a rectangular region and sequentially layer by layer, there is still a lot of unused space. In the next section, we propose a novel method to accelerate multi-DNN inference by leveraging these memory characteristics of tiny AI accelerators.

3 TinyMem

3.1 Insights for Efficient Multi-DNN

As discussed in §2.2, the key bottleneck of multi-DNN inference on tiny AI accelerators is the weight loading operation when switching between models (see Figure 3a). To reduce this overhead, we design TinyMem based on insights from tiny AI accelerator characteristics, specifically the 2D weight memory layout and the idle CPU cores during model inference. Unlike existing weight sharing techniques [4, 8, 9], which focus on sharing weight values across models, TinyMem addresses memory allocation and placement, making it orthogonal and compatible with these approaches.

Weight preservation (Figure 3b). The 2D weight memory layout leaves unused memory space, especially for layers with fewer channels (i.e., a small number of corresponding accelerator cores). Instead of unloading all weights when switching models, we skip the unloading step to preserve the memory and only reload weights that were overwritten by other models. This approach is particularly effective when the same model is executed repeatedly, reducing the need to reload weights or when models store their weights in distinct memory regions. These scenarios minimize the likelihood of overwriting preserved weights, ensuring that the preserved weights can be reused without reloading.

Weight preloading (Figure 3c). Since model inference runs on dedicated processors in the AI accelerator, we can preload the next model's weights using the idle CPU into memory regions not used by the current model. This parallelization of weight loading and inference reduces end-to-end latency, thereby improving the total throughput of model inferences. This approach is effective when multiple models are executed alternately in a non-repetitive manner, particularly with three or more models.

Weight packing (Figure 3d). The typical method of placing model weights into the weight memory on tiny AI accelerators is to arrange the weights horizontally layer by layer from the left and vertically accelerator core by core from the top [1], as shown in Figure 1. However, when multiple models run, this approach reduces the opportunities for weight preservation and preloading as these models are placed starting from the top-left corner and will largely overlap. By dynamically adjusting the placement of weights, weight packing prevents memory fragmentation, making it particularly advantageous for models with layers of different-sized weights. It also enhances the performance of preservation and preloading techniques, especially when the models do not fully occupy the weight memory.

3.2 Operational Flow

Due to the lack of memory management support in tiny AI accelerators, model developers need to manually manage the weight memory on these accelerators. As shown in Figure 4a, when loading model weights, they need to specify the accelerator cores and destination address of the weight memory for each layer based on the 2D layout of the weight memory (Figure 1). This manual

```
# Layer 0
- out_offset: 0x4000
  processors: 0x0000000000000007
  ...
# Layer 1
- out_offset: 0x0000
  processors: 0xffffffffffffffff
  ...
```

(a) Today's practice.

```
TinyMem.load_weights([
    # Weight values and sizes
    kernels_of(checkpoint_0),
    kernels_of(checkpoint_1),
    kernels_of(checkpoint_2),
])
```

(b) TinyMem.

Figure 4: Weight loading interfaces. TinyMem automatically schedules the weight memory layout with given checkpoints.

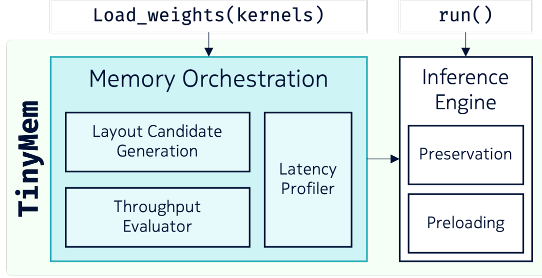


Figure 5: TinyMem operational flow.

management is not only cumbersome but also nearly impossible to optimize for runtime performance because of the interdependencies between concurrently running models.

Figure 5 shows the operational flow of TinyMem. For runtime memory orchestration, it provides developers with abstractions for underlying weight memory operations: `load_weight()` and `unload_weight()`, as shown in Figure 4b. Layout of added models' weights is optimally adjusted in the *Memory orchestration* component of TinyMem. To this end, it generates layout candidates for each model by changing the position of each layer's weight. Then, it selects the optimal set of layouts, which minimizes the end-to-end latency to run all running models, thereby maximizing the total inference throughput. The decision also includes the order of model inferences as the order affects the end-to-end latency due to its impact on preloading operation. For selection, TinyMem adopts the *Latency profiler* that estimates the latency of weight loading and model inference at runtime. Once *run()* is called, the *Inference engine* executes multi-DNN inference based on the orchestration decision, i.e., preserving weights as needed and preloading the next model's unpreserved weights during inference.

3.3 Holistic Memory Coordination

We explain how we make decisions for holistic memory coordination using the aforementioned insights: weight preservation, preloading, and packing. Although we explain these insights together for clarity, note that preservation and preloading are the runtime operations required during multi-DNN inference. The coordination decisions required for these operations are (a) each model's weight memory layout and (b) the switching order of models. Packing is a strategy for generating the weight memory layout, which enhances the effectiveness of weight preservation.

However, making an optimal coordination decision is not trivial because the decision to maximize the benefit from preservation sometimes contradicts the decision for preloading. To maximize the

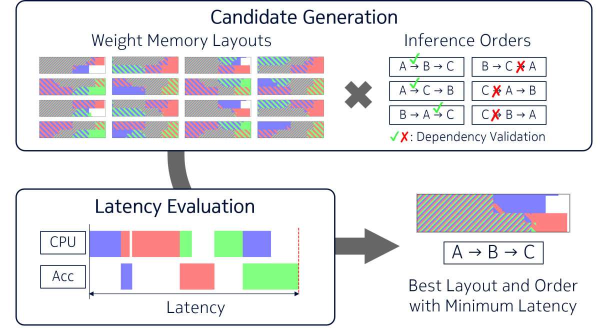


Figure 6: Operational flow for memory coordination. This scenario assumes model C follows model A.

Algorithm 1 Holistic Memory Coordination

Inputs: $S = \{s_{i,j}\}$: Weight size of layer j of model i, $L_{\text{inf}} = \{l_{\text{inf},i}\}$: Inference latency of model i, k: Weight loading time constant per byte, D: Model dependency constraints
Output: $L = \{(x_{i,j}, y_{i,j})\}$: Layouts, O: Inference order

1: $\mathbb{L}_i \leftarrow$ 4 flipped layouts of packed layout of model i
2: $\mathbb{L} \leftarrow \prod_i \mathbb{L}_i$ ▷ Generate all layout combinations across models
3: $\mathbb{O}_{\text{all}} \leftarrow$ All permutations of the models
4: $\mathbb{O}_{\text{val}} \leftarrow \text{filter_valid}(\mathbb{O}_{\text{all}}, D)$ ▷ Filter orders based on dependencies
5: $\mathbb{T} \leftarrow \{\}$ ▷ Map to store throughputs
6: **for all** $L \in \mathbb{L}$ **do** ▷ Iterate layouts
7: **for all** $O \in \mathbb{O}_{\text{val}}$ **do** ▷ Iterate valid orders
8: $Lat \leftarrow 0$ ▷ Initialize latency
9: **for all** $i \in O$ **do** ▷ Iterate models
10: $l_{\text{preload}} \leftarrow k \times \text{Preload Bytes}(S, L, O, i)$
11: $l_{\text{postload}} \leftarrow k \times \text{Postload Bytes}(S, L, O, i)$
12: $Lat \leftarrow Lat + (l_{\text{preload}} - l_{\text{inf},i-1})^+ + l_{\text{postload}} + l_{\text{inf},i}$
13: $\mathbb{T}[L, O] \leftarrow 1/Lat$ ▷ Compute throughput
14: **return** $\arg\max_{L,O} \mathbb{T}[L, O]$

benefit of weight preservation, model weights should be placed in a way of minimizing the overlap across all running models' weights. In contrast, for preloading, the main factor that affects its benefit is the overlap between two consecutive models. While one model runs, TinyMem preloads the next model's unpreserved weights, but these are constrained to memory regions not occupied by the current or preserved by the next model. Layer-wise preloading is also viable, as it only requires considering two consecutive layers, but it increases scheduling complexity and overhead due to frequent interruptions. Thus, when models compete for limited weight memory, minimizing overlap across all running models for preservation may not be ideal for preloading a specific pair of models.

To address this, we jointly optimize the weight memory layout and model switching order, selecting the best combination based on the expected throughput. Algorithm 1 outlines the process.

Candidate generation. Due to the excessive number of possible layout configurations, even for a single model–where each layer's weight can be arbitrarily placed within the entire weight memory space– considering all possible combinations incurs a significant system cost. To effectively reduce the search scope, we devise a simple but effective method. The key idea is to make each model's weight memory as compact as possible, with the rationale that

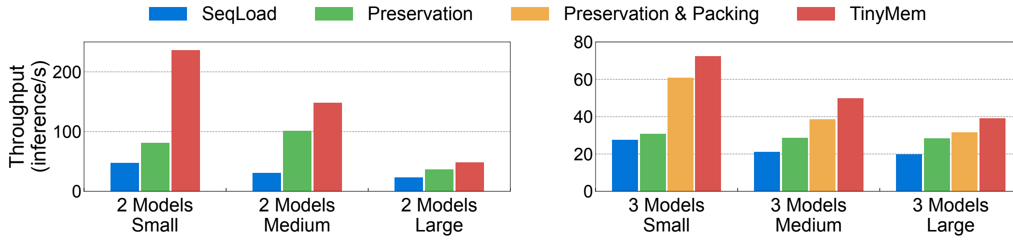


Figure 7: Multi-DNN inference throughput comparison.

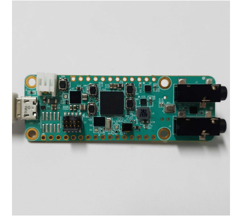


Figure 8: MAX78000 Feather board.

this will minimize the chance of overlap with other models. We begin by packing each model independently using the MaxRect algorithm [7], which arranges model weights to minimize the size of the rectangular region containing the model weights. For each model, we generate four layout candidates by shifting and flipping the weight layout into four corners of the entire weight memory. These layouts are then combined across models to generate all possible layout combinations, resulting in 4^n candidates for n models. Separately, we generate all possible execution orders for the models and filter them based on the given dependencies. For example, if model A must be executed before model B, only the orders satisfying this condition are retained. Finally, the final candidate set is formed by testing all combinations of layout candidates and valid execution orders.

Throughput evaluation. To evaluate the throughput of each candidate, we first estimate an end-to-end latency to run all running models. The latency of a model is determined by the sum of weight loading time and inference time, and the end-to-end latency is computed as the sum of all model's latency (line: 11). The weight loading time of a model consists of two parts: (a) preloading weights during the previous model's inference and (b) post-loading the rest weights (unpreserved and unpreloaded) after the inference. Since today's tiny AI accelerators do not support the parallel execution of multiple models due to limited memory and fixed operation pipelines, we estimate the model throughput by taking the reciprocal of the end-to-end latency.

Latency estimation:. (a) We model weight loading latency by quantifying the time taken for memory copy operations between RAM and on-accelerator memory. Since the latency shows a linear relationship to the weight memory size (p-value$< 1 \times 10^{-15}$), we model the weight loading time as $k \times WeightSize$, where k is a time coefficient obtained offline. The proportionality constant k varies by device type. For instance, on MAX78000, k is measured as 68.455 ns/byte. (b) The model inference latency is estimated using an offline profiling approach. We construct a latency table in advance by profiling each model's inference latency, which is then used at runtime for latency prediction. The Tiny AI Accelerator's deterministic nature, stemming from fixed hardware execution paths, ensures minimal inference time variance, making the offline profiling-based approach effective.

4 Evaluation

4.1 Evaluation Setup

Hardware. We prototyped TinyMem on MAX78000 Feather board, a development platform for MAX78000 [3]. It has 512 KB of data memory, 442 KB of weight memory, and 2 KB of bias memory on the CNN accelerator.

Table 1: List of models used in our evaluation.

Model Name	Weight Layout	Size (MB)	Weight. Latency	Inf. Latency	Input Type
SimpleNet-100		4.0	24.0 ms	4.58 ms	Image
SimpleNet-10		3.5	20.8 ms	4.57 ms	Image
SimplerNet-100		4.5	11.5 ms	2.59 ms	Image
KwsNet		2.6	12.1 ms	1.44 ms	Audio
ConvNet		0.6	4.9 ms	1.41 ms	Image

Table 2: Model combinations used in the experiments.

Workload	Small	Medium	Large
2 Models	KwsNet ConvNet	SimpleNet-10 ConvNet	SimpleNet-100 KwsNet
3 Models	SimplerNet-100 KwsNet ConvNet	SimpleNet-10 KwsNet ConvNet	SimpleNet-100 KwsNet ConvNet

DNNs. We use five models for evaluation as summarized in Table 1. As MAX78000 supports only int8 quantized models, we quantized pre-trained PyTorch model weights using Analog ai8x-synthesizer [1].

Workloads. We use six types of workloads generated by combining different numbers and sizes of models. Table 2 shows the model combinations used in the experiments. *Large*, *Medium*, and *Small* represent the memory characteristics of each combination based on the total size of models.

Baselines. `SeqLoad` is a baseline which sequentially loads and unloads all model weights. `Preservation` and `Preservation & Packing` sequentially apply our weight memory preservation and packing techniques without preloading.

Metric. We use throughput, the total number of inferences per second across all models, assuming they run in equal frequency.

4.2 Performance Analysis

Figure 7 and Table 3 show that TinyMem consistently outperforms baselines for both 2 Model and 3 Model Workloads as well as for different model complexities. We analyze the results in detail.

Overall performance. Overall, TinyMem achieves 2.12-5.01× and 1.97-2.62× higher throughput than `SeqLoad` for 2 Models and 3 Models Workload, respectively. TinyMem significantly reduces the bottleneck of multi-DNN inference, particularly in weight loading

Table 3: Throughput on MAX78002 for 3 Model Workload (SimpleNet-100, SimplerNet-100, SimpleNet-10).

SeqLoad	Preservation	Preservation & Preloading	TinyMem
8.04	12.77	13.92	12.81

time. TinyMem consistently outperforms the baseline, showing that reducing weight loading time is key to improving throughput.

Performance breakdown. All our techniques—weight preservation, packing, and preloading—contribute to throughput gain, achieving average improvements of 1.95×, 1.67×, and 1.24×, respectively, for 2-model and 3-model cases. Note that in the 2 Model case, there is no available memory space for preloading between inferences, resulting in identical performance for the `Preservation & Packing` and TinyMem configurations. This is because, in the 2-model case, memory is either (1) shared by both models, leaving no unused space for preloading, or (2) used exclusively by one model, in which case the space can be preserved rather than requiring runtime preloading. Weight preservation has the most significant impact, as it directly addresses the core bottleneck of weight reloading by keeping weights in memory when switching between models. This is particularly effective in smaller workloads or when memory usage is well below capacity, avoiding costly reloading.

Once weight preservation has reduced the weight loading time, preloading becomes more effective. In Figure 2, the inference latency is much shorter than the weight loading time, which typically limits the potential gains from preloading, as pipelining is most effective when task lengths are balanced. Without preservation, preloading alone would have a limited effect since the system is dominated by the long weight loading time. However, after preservation reduces this bottleneck, preloading can better parallelize inference and weight loading, improving overall throughput. As workloads increase, either due to more models or larger model sizes, the effectiveness of preservation diminishes as memory becomes constrained, making preloading and packing increasingly important to sustain high throughput.

Impact of workload. TinyMem achieves higher throughput across diverse workloads, both in terms of the number of models and complexities. We observe higher gain for smaller models (e.g., up to 5.09× throughput for the 2-model small). This is because smaller models leave more memory space for weight preservation, reducing the need for reloading.

TinyMem still outperforms `SeqLoad` for larger models, while the throughput gain is slightly lower (up to 2.62×). As model size increases, weight preservation becomes more challenging due to limited memory, making preloading more critical. In Large workloads, preservation becomes difficult, and the benefits of preloading become more pronounced. This demonstrates the importance of balancing preservation and preloading as model complexity grows. Similar trends hold as the number of models increases.

Impact of device. To demonstrate the generalizability of TinyMem across devices with different weight memory sizes, we conducted experiments on MAX78002, which has 5.3× larger weight memory than MAX78000. Table 3 shows that TinyMem achieves up to 1.73× higher throughput compared to `SeqLoad`. This gain is primarily from weight preservation, which increases throughput to 12.77 inferences per second (a 1.59× improvement). Preloading further enhances performance, raising throughput to 13.92 inferences per second (an additional 1.09× gain), while weight packing results in a slight decrease in performance. This outcome arises because the larger weight memory of MAX78002 allows models designed for MAX78000 to occupy a much smaller portion of the available memory. As a result, weight preservation plays a stronger role, similar to small model cases, while the benefits of preloading and packing are diminished since memory constraints are less of a concern. We expect larger gains when larger models are used.

5 Conclusion

In this paper, we introduced TinyMem, a memory-efficient system that optimizes weight memory management for multi-DNN inference on tiny AI accelerators. With weight preservation, preloading, and packing, TinyMem reduces weight loading time and end-to-end latency, boosting throughput by up to 5.01× over existing frameworks. These results show TinyMem's potential to improve multi-DNN inference efficiency in resource-constrained environments like wearable devices. Future work will focus on extending these techniques to more complex models and hardware platforms.

References

[1] Inc. Analog Devices. 2024. ai8x-synthesis: Quantization and Synthesis for ADI's MAX78000 and MAX78002 Edge AI Devices. https://github.com/analogdevicesinc/ai8x-synthesis. Accessed: 2024-10-11.
[2] Ananta Narayanan Balaji and Li-Shiuan Peh. 2023. AI-On-Skin: Towards Enabling Fast and Scalable On-body AI Inference for Wearable On-Skin Interfaces. *Proceedings of CHI* 7, EICS (2023), 1–34.
[3] Analog Devices. 2024. MAX78000. https://www.analog.com/en/products/max78000.html. Accessed: 2024-10-10.
[4] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of ACM MobiCom*. 115–127.
[5] Taesik Gong, Si Young Jang, Utku Günay Acer, Fahim Kawsar, and Chulhong Min. 2023. Collaborative inference via dynamic composition of tiny ai accelerators on mcus. *arXiv preprint arXiv:2401.08637* (2023).
[6] Taesik Gong, Fahim Kawsar, and Chulhong Min. 2024. DEX: Data Channel Extension for Efficient CNN Inference on Tiny AI Accelerators. In *NeurIPS '24*.
[7] Jukka Jylänki. 2010. A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing. *retrived from http://clb. demon. fi/files/RectangleBinPack. pdf* (2010).
[8] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of ACM MobiSys*. 175–190.
[9] Seulki Lee and Shahriar Nirjon. 2022. Weight separation for memory-efficient and accurate deep multitask learning. In *IEEE PerCom*. IEEE.
[10] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *NeurIPS* 33 (2020).
[11] Yi Lu, Lejia Zhou, Aili Zhang, Siyu Zha, Xiaojie Zhuo, and Sen Ge. 2024. Application of Deep Learning and Intelligent Sensing Analysis in Smart Home. *Sensors* 24, 3 (2024), 953.
[12] Arthur Moss, Hyunjong Lee, Lei Xun, Chulhong Min, Fahim Kawsar, and Alessandro Montanari. 2022. Ultra-low power DNN accelerators for IoT: Resource characterization of the MAX78000. In *AIChallengeIoT in conjunction with ACM SenSys*. 934–940.
[13] OmniBuds [n. d.]. OmniBuds. https://omnibuds.tech/. Accessed: 15 Mar. 2024.
[14] Samsung Electronics. 2025. Galaxy Ring. https://www.samsung.com/us/rings/galaxy-ring/. Accessed: 2025-01-14.
[15] Shift Moonwalkers [n. d.]. Shift Moonwalkers. https://shiftrobotics.io/. Accessed: 15 Mar. 2024.
[16] STMicroelectronics. 2024. STM32F7 Series. https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html. Accessed: 2024-10-12.
[17] GreenWaves Technologies. 2024. GAP9 processor for hearables and smart sensors. https://greenwaves-technologies.com/gap9_processor/. Accessed: 2024-10-10.