MoteNN: Memory Optimization via Fine-grained Scheduling for Deep Neural Networks on Tiny Devices

Renze Chen
Peking University
Beijing, China
crz@pku.edu.cn

Zijian Ding
Peking University
Beijing, China
zijianding@pku.edu.cn

Size Zheng
Peking University
Beijing, China
zhengsz@pku.edu.cn

Meng Li
Peking University
Beijing, China
meng.li@pku.edu.cn

Yun Liang*
Peking University
Beijing, China
ericlyun@pku.edu.cn

ABSTRACT

There has been a growing trend in deploying deep neural networks (DNNs) on tiny devices. However, deploying DNNs on such devices poses significant challenges due to the contradiction between DNNs' substantial memory requirements and the stringent memory constraints of tiny devices. Some prior works incur large latency overhead to save memory and target only simple CNNs, while others employ coarse-grained scheduling for complicated networks, leading to limited memory footprint reduction. This paper proposes MoteNN that performs fine-grained scheduling via operator partitioning on arbitrary DNNs to dramatically reduce peak memory usage with little latency overhead. MoteNN presents a graph representation named Axis Connecting Graph (ACG) to perform operator partition at graph-level efficiently. MoteNN further proposes an algorithm that finds the partition and schedule guided by memory bottlenecks. We evaluate MoteNN using various popular networks and show that MoteNN achieves up to 80% of peak memory usage reduction compared to the state-of-art works with nearly no latency overhead on tiny devices.

ACM Reference Format:
Renze Chen, Zijian Ding, Size Zheng, Meng Li, and Yun Liang. 2024. MoteNN: Memory Optimization via Fine-grained Scheduling for Deep Neural Networks on Tiny Devices. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655922

1 INTRODUCTION

The deployment of deep neural networks (DNNs) on tiny devices like micro-controller units (MCUs) is increasingly common, enabling widespread AI applications in the Internet of Things (IoT) area [10–12, 17, 25, 27, 28]. These tiny devices typically feature limited storage, including an SRAM of no more than 2 MB and an extensible read-only Flash memory of several megabytes. For example, the STM32F767ZI MCU includes a 512 KB SRAM and an extensible Flash with initial capacity of 2 MB. Generally, when deploying DNNs on tiny devices, model weights are allocated to Flash with relatively sufficient capacity [5], while the intermediate tensors during inference must be allocated to the capacity-limited SRAM. However, even networks tailored for resource-limited devices [4, 7, 8, 10, 11, 16, 21, 24] require large intermeidate-tensor memory compared to the limited SRAM of tiny devices. Some of them like MobileNetV2 [16] / BERT-Tiny [9] need large image resolution / long token sequence for accuracy, leading to large tensor sizes. Some of them have complicated structure to enhance model expressiveness [4, 6–8, 21], incurring more intermediate tensors residing in memory during inference.

As shown in Table 1, some libraries & frameworks [1, 14, 19, 23] have been developed for deploying DNNs on tiny devices, but they focus little on the optimization of memory usage of intermediate tensors. Works like TinyEngine [10, 11] reduce memory usage through manually designed patch-based inference, but such approach introduces significant latency overhead and is limited to simple CNN structures, thus hard to be applied to more complex DNNs, such as complicated CNNs like NASNet [4] or transformer models like BERT-Tiny [9]. Some works [3, 22, 26] find that scheduling the execution order of operators in networks with complicated structures can effectively reduce memory usage. However, these memory optimization is limited because they consider only coarse-grained scheduling, that is, scheduling the network in an operator-by-operator manner. We find that exploring the scheduling space inside the operator can reduce memory usage more effectively. By partitioning the operators into smaller ones and scheduling the finer-grained graph, greater memory reduction can be achieved. For example, Figure 1 (a) shows that when we schedule the graph coarsely, we can optimize peak memory usage to at most 768. But if we enable more fine-grained scheduling, as shown in Figure 1 (b), we can reduce peak memory usage to 384. In fact, patch-based inference used in previous works [11, 20] is a simple case of fine-grained scheduling, but it's a specific design and is hard to be applied to networks other than simple structured CNNs.

Although fine-grained scheduling can greatly reduce memory usage, it makes operator shapes smaller and the amount of operators larger, which results in lower hardware utilization and more kernel invocations, incurring latency overhead. Therefore, memory optimization via fine-grained scheduling needs to be performed

*Corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655922

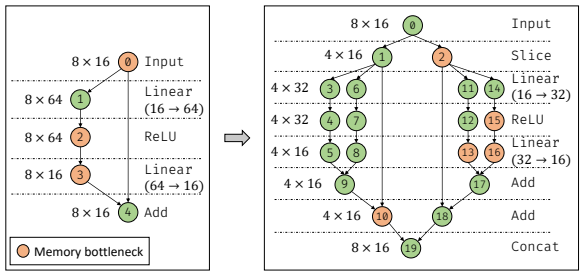

Figure 1: The graph & schedule & peak memory usage before and after fine-grained scheduling with the help of operator partition, where the numbers in nodes indicate execution order.

Table 1: Comparison with prior-art works

Work	Network Structure	Memory Reduction	Latency Overhead	Methods
CMSIS-NN [14] TFLite-Micro [19]	Arbitrary	No	No	Library & Runtime Support
TinyEngine [10, 11] FDT [20]	Simple CNN	High	Medium	Patch-based Inference
Serenity [3] HMCOS [26]	Arbitrary	Medium	No	Coarse-grained Scheduling
MoteNN (Ours)	**Arbitrary**	**High**	**Low**	**Graph-level Analysis, Fine-grained Scheduling**

under certain latency constraint. But there are quite a lot of partition schemes for each operator in the network, which together form a large combinatorial optimization space. *How to search for a network partition scheme and corresponding schedule to minimize peak memory under the latency constraint is challenging.* Meanwhile, for operators with overlapped sliding-windows such as convolution or pooling, partitioning the output operator may cause overlapped input dependency, leading to a large amount of re-computation overhead [11, 20]. *How to deal with the overlapped sliding-windows poses another challenge.*

To address these challenges, we propose MoteNN, a sophisticated memory optimizer designed for efficient fine-grained scheduling of DNNs on tiny devices. We find that since peak memory usage is determined by the maximum memory usage during execution, many combinations of partition strategies for operators have the same effect. Therefore, it is inefficient to explore each partition strategy for each operator separately. We introduce a graph representation called Axis Connecting Graph (ACG) to consider partitioning operators at the graph level, significantly reducing the optimization space. With the help of ACG, we design an optimization algorithm to select sub-graphs based on the current memory bottleneck to further reduce the optimization space. In addition, to avoid the large computation overhead caused by overlapped sliding-windows, we can carefully manage and allocate memory for the input blocks shared by multiple output blocks to reduce computation overhead.

In summary, this paper makes the following contributions:

- We propose MoteNN, a memory optimizer using fine-grained scheduling for DNN.
- We propose a graph representation named "Axis Connecting Graph" (ACG) to model graph-level partition scheme for building fine-grained graph.
- We design an optimization algorithm based on ACG and memory bottleneck to partition and schedule network to reduce its peak memory usage.

We evaluate MoteNN extensively with popular vision and language models, e.g., MobileNetV2 [16], MCUNetV2 [10], BERT-Tiny [9], etc, and networks with complex structures, e.g., NASNet [4]. Experiment results show that MoteNN can reduce up to 80% peak memory usage compared to state-of-the-arts scheduling frameworks with less then 5% latency overhead, enabling many powerful networks deployed on memory-limited tiny devices.

2 PRELIMINARIES

Computation Graph. A DNN is often represented as a graph G. $V = \mathcal{V}(G)$ represents operators, where each operator has several input tensors and one output tensor $E = \mathcal{E}(G) \subseteq V \times V$ represents the dependencies between operators, such as $(v_1, v_2) \in E$ indicates that operator v_2 depends on v_1 (usually, this means that the output tensor of v_1 is one of the input tensors of v_2).

We use $\texttt{pre}(v)$, $\texttt{suc}(v)$, $\texttt{sp-axes}(v)$, $\texttt{re-axes}(v)$, $v@i$, $v\$i$, and $\texttt{size}(v)$ to represent the predecessors, successors, spatial-axes, reduce-axes, i^{th} spatial-axis, i^{th} reduce-axis, and output tensor size of $v \in V$, respectively. Here spatial-axes means the axes/dimensions of output tensor, while reduce-axes refers to the loop axes for reduction. For example, in $\texttt{Matmul}$ $C[m,n] = \sum_k A[m,k] \times B[k,n]$, m, n are spatial-axes while k is reduce-axis. Note that there's $\texttt{size}(v) = \prod_{v@i \in \texttt{sp-axes}(v)} |v@i|$, where $|v@i|$ is the length of $v@i$.

Graph Schedule & Memory Usage & Memory Bottlenecks. A topo-order $T = (v_1, v_2, ..., v_n)$ of $\mathcal{V}(G)$ is a schedule of graph G. Assuming that i represents the time point when the i^{th} operator is completed, we can get the start/end of the lifetime of each operator v_i: $\texttt{start}(i) = i-1$, $\texttt{end}(i) = \max_{v_j \in \texttt{suc}(v_i)} j$. The set of tensors that are alive during the execution of operator v_i is $\texttt{alive}(i) = \{v_j \in T \mid \texttt{start}(j) \le i \le \texttt{end}(j)\}$. The memory usage of operator v_i during execution is $M_i = \sum_{u \in \texttt{alive}(i)} \texttt{size}(u)$; and the **peak memory usage** during the execution of G is $M_{peak} = \max_{1 \le i \le n} M_i$. The tensors that contribute to the peak memory usage are called **memory bottlenecks**: $V_{mb} = \bigcup \{\texttt{alive}(i) \mid M_i = M_{peak}\}$.

Example. In Figure 1 (a), during v_3's execution there are three tensors alive: v_0, v_2, and v_3. Therefore, $M_3 = 8*16+8*64+8*16 = 768$, which is the peak memory usage of this graph under such schedule. Thus v_0, v_2, v_3 are memory bottlenecks. In Figure 1 (b), during v_{16}'s execution there are five tensors alive: v_2, v_{10}, v_{13}, v_{15}, and v_{16}. Therefore, $M_{16} = 4*16+4*16+4*16+4*32+4*16 = 384$, which is the peak memory usage of this graph under such schedule. Thus $v_2, v_{10}, v_{13}, v_{15}, v_{16}$ are memory bottlenecks.

3 METHODS

3.1 Insights of MoteNN

We aim to optimize the peak memory usage, which refers to the maximum memory occupancy during execution. Our methods are based on two insights.

Insight 1: *It is inefficient to consider the operator partition separately for each operator.* Partitioning separately will introduce much redundancy and is hard to model the connections among operators. For instance, in Figure 2 (b), splitting the two $\texttt{Linear}$ operators

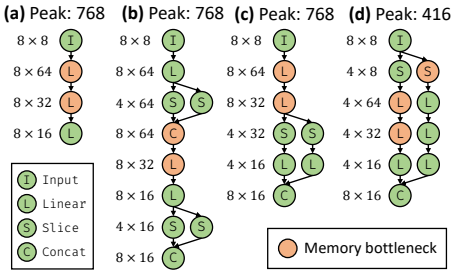

Figure 2: Different partition choices lead to different peak memory.

separately in (a) has no impact on the peak memory. We need to consider the partition and scheduling of operators at graph level.

Insight 2: *It is important to focus on memory bottlenecks.* Only partitioning memory bottlenecks tensors can optimize peak memory. For instance, the tensors of size 8×32 and 8×64 are memory-bottlenecks in Figure 2 (a). The partition of non-bottleneck tensors like (c) doesn't impact the peak memory, while (d) splits bottlenecks of (a) and nearly halves the peak memory. In practice, memory bottlenecks are mostly inside certain cells [3, 22, 26], which are appropriate sub-graphs for operator partitioning.

3.2 Workflow Overview

Figure 4 shows the workflow of MoteNN. It accepts a model description like ONNX as input. The analyzer analyzes the model and builds an Axis Connecting Graph (ACG) based on the axis mappings of each operator and the dependency between operators. The optimizer then searches for feasible partition schemes and corresponding schedules based on ACG and memory-bottlenecks. It uses a profiler & simulator to measure the latency of operators with specific shapes and estimate the latency of the entire graph. Then, MoteNN generates code based on the searched fine-grained graph and operator schedule. Finally, the generated code is compiled into a binary file for deployment on the target device.

3.3 Axis Connecting Graph (ACG)

Based on **Insight 1**, we need to consider operator partition at the graph level. When considering partition at the operator level, we need to get the axes (spatial-axis, reduce-axis) of an operator and partition along these axes. Likewise, to consider operator partition and scheduling at the graph level, we need a data structure to represent the "axes" of a subgraph. To this end, we propose Axis Connecting Graph (ACG) that represents the connection among the axes of different operators in a graph. The "axis" of a subgraph in the computation graph is a subgraph of the ACG. ACG represents the inter-axis dependencies of operators in the form of a graph, which allows us to consider operator partition from graph-level, greatly reducing the optimization space that needs to be explored.

Given a graph G, we define its **Axis Connecting Graph (ACG)** as $A = \mathcal{A}(G)$, where

(1) For $\forall v \in \mathcal{V}(G), v@i \in \texttt{sp-axes}(v)$, there's $v@i \in \mathcal{V}(A)$.
(2) For $\forall v \in \mathcal{V}(G), v\$i \in \texttt{re-axes}(v)$, there's $v\$i \in \mathcal{V}(A)$.
(3) For $\forall (u, v) \in \mathcal{E}(G), i, j \geq 0$, if $|u@i| = |v@j|$ and $u@i, v@j$ correspond to a same spatial-loop, then $(u@i, v@j) \in \mathcal{E}(A)$.
(4) For $\forall (u, v) \in \mathcal{E}(G), i, j \geq 0$, if $u@i$ and $v\$j$ correspond to a same reduce-loop, then $(u@i, v\$j) \in \mathcal{E}(A)$.

For instance, consider a `Linear` operator node v with shape $N \times K$ and its input node u with shape $N \times C$. The first axis of u and v correspond to a same spatial-axis, and the second axis of u corresponds to the reduce-axis of v, yielding $(u@0, v@0), (u@1, v\$0) \in \mathcal{E}(A)$. Figure 3 (a) presents the graph of an inverted-bottleneck in MobileNetV2 [16], and (b) shows sub-graphs of its ACG, where there are four connected sub-graphs, corresponding to the height-axis of `Input`, width-axis of `Input`, channel-axis of `Input`, and channel-axis of `DW-Conv`, respectively.

With the help of ACG, we can represent the operator partition at graph level. For a graph G and its subgraph $S \subseteq G$, we define a **Partition Scheme** $p = (S, A, V_I, V_O, n)$, where A is a connected sub-graph of $\mathcal{A}(S)$, $V_I \subseteq \mathcal{V}(S)$ represents the input nodes of S (which can be explicitly split by `Slice`), $V_O \subseteq \mathcal{V}(S)$ represents the output nodes of S (which can be explicitly merged by `Concat`), and n is the number of partitions. Figure 5 shows how to partition a sub-graph based on a partition scheme. There are two input and two output nodes. u_1 is split while u_2 is not. v_1 is split along the spatial-axis, while v_2 is split along the reduce-axis. The subgraph is split into two parts, with u_1 being split into two parts using a `Slice` operator, and u_2 being reused in both parts. Parts of v_1 are merged using a `Concat` operator in the end, while parts of v_2 are accumulated using an `Add` operator. Figure 3 (c) (d) shows two examples of partitioning sub-graphs of (a) along sub-ACG of (b).

3.4 Handling Overlapped Sliding-window

In operators like `Conv2d` with kernel sizes greater than 1, a sliding-window reduction occurs along the height/width axis, enlarging receptive fields. If multiple `Conv2d` operators are sequentially used in a sub-graph, this enlargement effect accumulates, significantly increasing computation [11]. Figure 6 (a) demonstrates consecutive 3×3 `Conv2d`, (b) illustrates block dependencies when partitioning the sub-graph into n segments starting from the output, and (c) depicts the partitioned sub-graph with its additional computation and peak memory ratios. To lower memory usage, increasing n is necessary, but this leads to more extra computation. An alternative method, used by some hardware-level accelerator design [15, 18], infers block boundaries based on dependencies and splits overlapping parts into separate blocks, as shown in Figure 6 (d), avoiding extra computation but slightly increasing memory usage. However, this approach implemented in software-level can result in many small operators, negatively affecting performance, as shown in Table 3. Meanwhile, both methods above have limitations in complex CNNs with varying numbers of sliding-window operators in different paths. As shown in Figure 6 (f), the receptive fields from v_5 to v_1 differ in the two paths due to the different numbers of `Conv2d`. *It is hard to partition sub-graph in such situation with the above methods.*

We propose a compromised solution, which keeps the partition granularity consistent across layers, as shown in Figure 6 (e). Before each block is executed, we use a `BlockConcat` kernel to merge the input blocks that it depends on (for each input block, it will only copy the data portion consumed by the output block). The overhead introduced by `BlockConcat` can be generally ignored as shown in Table 3. This method incurs a little more memory usage than the previous ones. In most networks, cell layers (L of Figure 6) typically ranges from 2 to 6 [2, 4, 6–8, 10, 11, 13, 16, 21, 24]. Here, we take

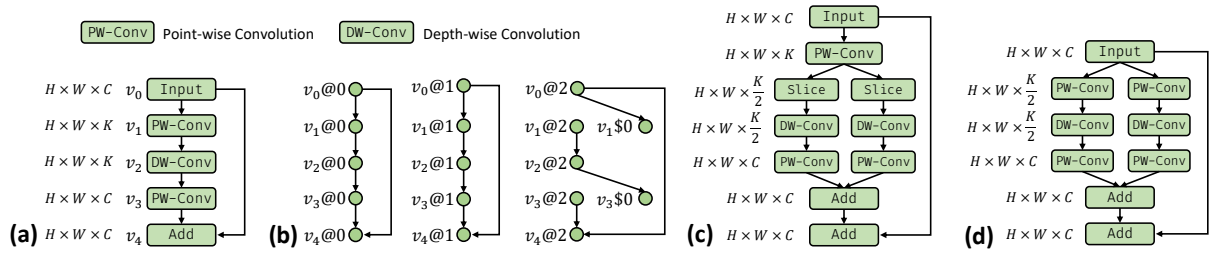

Figure 3: Example of Axis Connecting Graph (ACG). (a) Graph of an inverted-bottleneck in MobileNetV2 [16]. (b) ACG of graph in (a). (c) Partition sub-graph $\{v_1, v_2, v_3\}$ into two parts along ACG $\{v_1@2, v_1@2, v_3\$0\}$. (d) Partition sub-graph $\{v_0, v_1, v_2, v_3\}$ into parts along ACG $\{v_1@2, v_1@2, v_3\$0\}$.

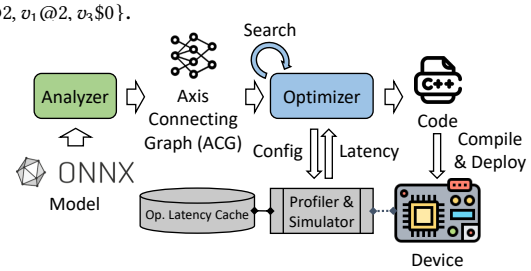

Figure 4: Workflow overview of MoteNN

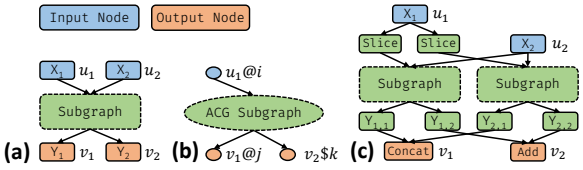

Figure 5: Illustration of graph partition along ACG. (a) Graph for partition. (b) ACG that partitioning along $(i, j, k \geq 0)$. (c) Result fine-grained graph after partition.

$L = 4$, and the peak memory ratio of our solution is $\frac{n+16}{2n}$, which is larger than the method in Figure 6 (c) by $\frac{7}{n} - \frac{7}{H}$. As n increases, i.e., when the memory usage is lower, the difference becomes smaller.

3.5 Optimization Algorithm

With the aid of ACG, we design a beam search algorithm based on **Insight 2** to optimize peak memory under a given latency constraint, as shown in Algorithm 1. It iteratively partitions the graph (line 5-17) by identifying memory bottlenecks (line 6), selecting the cell containing bottlenecks (lines 8), and the connected sub-ACG containing bottlenecks within a cell (lines 9). Maximum heap **P** maintains β best optimization states (line 12). Each state P contains many partition schemes p. Each sub-ACG A can generate several partition schemes $p = (S, A, V_I, V_O, n)$ with different partition numbers n, which are appended to current states to construct new ones (line 11). Top-β new states are kept for future iterations (line 13-16). We define `LessThen` (line 3-4) for the heap (line 12), which compares peak memory first if latency meets the constraint otherwise compares latency first (line 4).

Details. `GenSchemes` generates different partition schemes $p = (S, A, V_I, V_O, n)$ for a given ACG A by enumerating factors n of the axis-length of A. `Schedule` is responsible for scheduling the fine-grained graph. We run this function frequently, and the graph it handles is complex after partitioning. Therefore, we choose the fastest reverse-post-order algorithm with linear complexity in terms of the network size; it can achieve near-optimal peak memory in most cases [26]. `Apply` applies the partition schemes on a given graph to produce a new fine-grained graph. `PeakMem` and `MemBottlenecks` are implemented using the formulas in Section 2. `Latency` measures the performance of the graph.

Algorithm 1: Memory-bottleneck-aware Beam Search

input : Graph: G. Latency constraint ratio: δ. Beam width: β
output: Fine-grained graph after operator partition: G'. Schedule: T'

1 $\mathbf{C}_{rest}$:= all cells of G; $\mathbf{P} := \{\{\text{DummyPartitionScheme}\}\}$;
2 $G' := G$; $T' := \texttt{Schedule}(G')$; $L_\top := \texttt{Latency}(G') \times \delta$;
3 **def** `LessThen`(P_1, P_2):
4 **return if** $P_1.L \leq L_\top \wedge P_2.L \leq L_\top$ **then** $(P_1.M, P_1.L) < (P_2.M, P_2.L)$ **else** $(P_1.L, P_1.M) < (P_2.L, P_2.M)$;
5 **while** *true* **do**
6 $V_{mb} := \texttt{MemBottlenecks}(G', T')$;
$\mathbf{C}_{mb} := \{C \in \mathbf{C}_{rest} \mid \mathcal{V}(C) \cap V_{mb} \neq \emptyset\}$;
7 **if** $\mathbf{C}_{mb} = \emptyset$ **then break**;
8 $C := \text{argmax}_{C \in \mathbf{C}_{mb}} \Sigma_{v \in (\mathcal{V}(C) \cap V_{mb})} \texttt{size}(v)$; $\mathbf{C}_{rest} := \mathbf{C}_{rest} \setminus \{C\}$;
9 $\mathbf{A}_{mb} := \{A \in$ all valid connected sub-graphs of $\mathcal{A}(C) \mid \exists v \in V_{mb}, i \geq 1$ s.t. $(\texttt{sp-axes}(v) \cup \texttt{re-axes}(v)) \cap \mathcal{V}(A) \neq \emptyset\}$;
10 **for** $A \in \mathbf{A}_{mb}$ *sorted by depth* **do**
11 $\mathbf{P}' := \{P \cup \{p\} \mid P \in \mathbf{P}, p \in \texttt{GenSchemes}(A)\}$;
12 $\mathbf{P} := \texttt{MaxHeap}(\texttt{LessThen})$;
13 **for** $P \in \mathbf{P}'$ **do**
14 $G' := \texttt{Apply}(G, P)$; $T' := \texttt{Schedule}(G')$;
15 $P.M := \texttt{PeakMem}(G', T')$; $P.L := \texttt{Latency}(G')$;
16 $\mathbf{P}.\texttt{push}(P)$; **if** $|\mathbf{P}| > \beta$ **then** $\mathbf{P}.\texttt{pop\_largest}()$;
17 $P := \mathbf{P}.\texttt{get\_smallest}()$; $G' := \texttt{Apply}(G, P)$; $T' = \texttt{Schedule}(G')$;
18 **return** G', T';

4 EVALUATION

4.1 Experiment Setup

Table 2: Networks for evaluation

Network Names	Input Size
NASNet-A [4], DARTS [7]	$32 \times 32 \times 3$ (CIFAR-10)
NASNet-A [4], DARTS [7], FPNAS [8]	$224 \times 224 \times 3$ (ImageNet)
MobileNetV2 [16], MCUNetV2 [11]	$224 \times 224 \times 3$ (ImageNet)
BERT-Tiny [9]	512×128

We select `STM32H7A3ZI-Q` MCU with `ARM Cortex-M7` core as our test platform. It has 1.4 MB SRAM but can simulate lower memory constraints. We utilize CMSIS-NN [14] as the operator library for MoteNN and baselines and implement a kernel of `BlockConcat` proposed in Section 3.4 for MoteNN. We use Mbed with `gcc-arm` as the compilation system. We set $\beta = 8$ in Algorithm 1 of MoteNN. We use `Intel(R) Xeon(R) Silver 4210R` CPU to run optimization processes, each of which is completed within 2 minutes.

We evaluate MoteNN with 8 popular networks shown in Table 2. For NASNet-A [4] and DARTS [7], different datasets (input sizes)

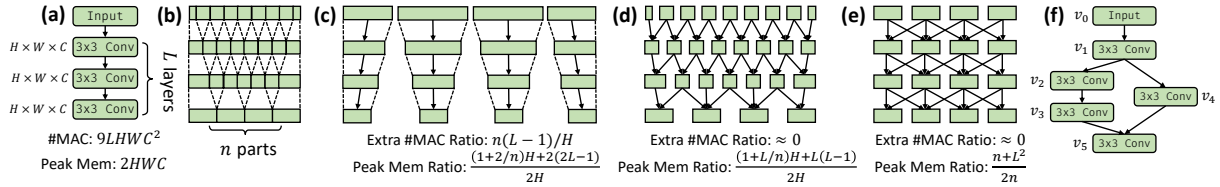

Figure 6: (a) L consecutive 3×3 Conv2d. (b) Dependency between blocks if we split the output into n parts. (c) (d) (e) Three solutions to split the subgraph. "Extra #MAC Ratio" means the ratio of extra #MAC to original one. "Peak Memory Ratio" means the ratio of current peak memory to original one. (f) Example of a subgraph where different paths have different numbers of Conv2d, hard to be handled by solutions in (c) and (d).

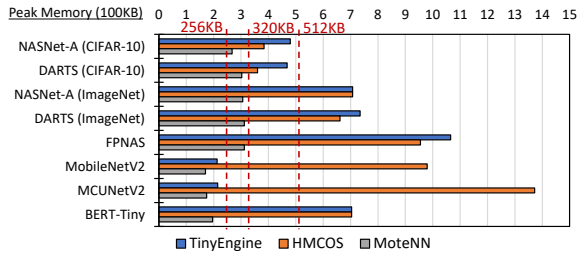

Figure 7: Comparison of minimum peak memory usage achieved by MoteNN and baselines on different networks (int8 quantization).

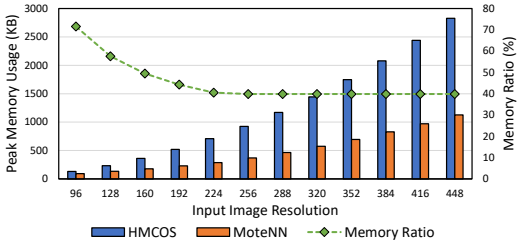

Figure 8: Peak memory (int8) with different input image resolution for NASNet-A (ImageNet). "Memory ratio" refers to the ratio of memory usage between MoteNN and HMCOS.

correspond to different network architectures. For MCUNetV2 [11], we use the released MCUNetV2-SE-Large model. These networks have either large tensor sizes or complex topologies, suffering from high peak memory during inference although they are tailored for resource-limited devices.

We compare MoteNN with two state-of-the-art open-sourced memory optimizers for DNN on tiny devices: TinyEngine [10, 11] and HMCOS [26]. TinyEngine proposes patch-based inference to optimize memory of simple CNNs. HMCOS is the state-of-the-art memory-aware graph scheduler for complicated networks.

4.2 Peak Memory Usage Evaluation

We evaluate and compare the peak memory usage optimized by TinyEngine, HMCOS, and MoteNN. We run MoteNN with latency overhead constraint $\delta = 1.05$ in Algorithm 1, that is, the execution latency overhead introduced by fine-grained scheduling should be no more than 5%. To demonstrate MoteNN's effectiveness, we set several memory constraints (256KB, 320KB, 512KB, 1MB, and 2MB), which are common for tiny devices.

Results. Figure 7 shows the evaluation results (with int8 as quantization precision). For all networks, MoteNN achieves the smallest peak memory usage. MoteNN achieves average 46% (up to 72%) peak memory usage reduction compared to TinyEngine, and average 58% (up to 87%) peak memory usage reduction compared to HMCOS. We also observe that MoteNN can meet the 320KB memory limit for all the networks, while TinyEngine can satisfy such constraint for only MobileNetV2 and MCUNetV2, and HMCOS cannot satisfy such limit for any test networks. Furthermore, MoteNN can meet the 256KB memory limit for 3 networks, while TinyEngine can satisfy it for only 2 networks. If we change the quantization precision from 8-bit to 16-bit or 32-bit, MoteNN can easily meet memory constraints of 1MB or 2MB respectively on all 8 benchmark networks, while TinyEngine / HMCOS can only satisfy 1MB (2MB) constraint on 4 / 2 networks for 16-bit (32-bit) quantization precision.

We observe that for NASNet, DARTS and FPNAS, HMCOS performs better than TinyEngine. This is because TinyEngine can not optimize networks with complicated structures. But for MobileNetV2 and MCUNetV2, TinyEngine is better than HMCOS as HMCOS cannot exploit intra-operator scheduling space. The reason why MoteNN can achieve significant peak memory usage reduction is that it performs fine-grained scheduling via graph-level tensor partition, exploiting much more scheduling opportunities inside each tensor compared with TinyEngine and HMCOS.

Memory Usages with Different Input Sizes. Note that for NASNet and DARTS with dataset CIFAR-10, memory footprint reduction of MoteNN compared to HMCOS is at most 30%, while for dataset ImageNet, such reduction is at least 50%. Such variation is mainly because MoteNN can exploit more memory reduction opportunities via fine-grained scheduling on networks with larger shapes. To further demonstrate the impact of input sizes, we compare MoteNN with HMCOS under different input image resolutions. Figure 8 shows the peak memory usage optimized by MoteNN and HMCOS, as well as the ratio between the two, when changing the input image resolution of NASNet-A (ImageNet). As the input image resolution increases, the benefits of MoteNN over HMCOS increases and gradually reaches a saturation point as the resolution continues to increase. This is consistent with the formula for peak memory ratio in Figure 6 (e), $\frac{n+L^2}{2n}$, as the number of partitions n increases with higher input image resolutions, causing the value of $\frac{n+L^2}{2n}$ to decrease, but at a slower rate.

4.3 Latency Overhead Evaluation

Full Network Latency Overhead. In this section, we analyze the latency overhead introduced by MoteNN. We set the latency overhead constraint δ in Algorithm 1 as 1.05, 1.02, 1.01 (5%, 2%, 1% latency overhead) and collect some sample points of memory ratio (the ratio between peak memory optimized by MoteNN and HMCOS) and latency overhead of MoteNN when optimizing NASNet-A (ImageNet). As shown in Figure 9, to optimize memory ratio to 60%, the latency overhead is less than 1%, which is negligible. However, to further optimize it, the graph should be more fine-grained,

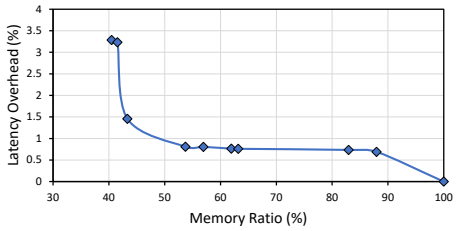

Figure 9: Peak memory and latency overhead curve of NASNet-A (ImageNet). "Memory ratio" refers to the ratio of memory usage between MoteNN and HMCOS.

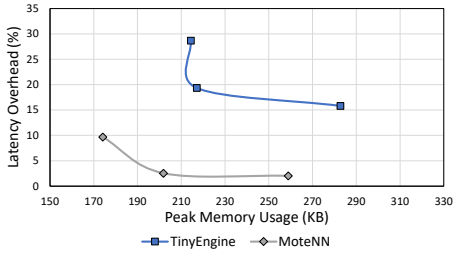

Figure 10: Peak memory usage and latency overhead curves of MCUNetV2-SE-Large optimized by TinyEngine and MoteNN

resulting in more kernel invocations and lower hardware utilization due to smaller operators. Therefore, the latency overhead increases rapidly when the memory ratio is reduced.

Single Operator Latency Overhead. We further conduct an experiment to evaluate the performance degradation caused by partitioning a `Conv2d` operator along both height-axis and width-axis. The results are shown in Table 3. We can observe that as the factor decreases, it incurs more latency overhead. For instance, reducing the factor from 16 to 8 only incurs less than 0.5% extra latency overhead, while reducing it from 2 to 1 leads to 40% extra latency overhead. Table 3 also shows that the latency of all `BlockConcat` kernels, relative to the total latency, is typically less than 2%. However, for extremely small partition factor, the ratio goes up because of too many kernel invocations.

Comparison to Latency Overhead of TinyEngine. We present a comparison between MoteNN and TinyEngine in terms of peak memory usage and latency overhead in MCUNetV2-SE-Large. For TinyEngine, we obtain three optimization results with patch sizes of 2, 4, and 7. For MoteNN, we set the latency overhead constraint $\delta = 1.1$ and show three optimization states during search. Figure 10 shows the results. We can observe that both TinyEngine and MoteNN can easily reduce peak memory from the original 1470KB to within 320KB. However, TinyEngine's latency overhead is much larger than MoteNN due to the significant computation overhead caused by the patch-based inference, as shown in Figure 6 (c). In contrast, our approach can achieve more memory reduction with much less computation overhead.

5 CONCLUSION

This paper proposes MoteNN, a method designed to optimize peak memory usage during the deployment of DNNs on tiny devices. It introduces Axis Connecting Graph to model graph-level axis relations, and efficiently searches graph partition & schedule based on memory bottlenecks. Our evaluation results show that MoteNN can reduce up to 80% peak memory usage of popular DNNs compared to state-of-the-art works with nearly no latency overhead.

Table 3: Performance of partitioning 3×3 `Conv2d` along the height-axis and width-axis. The operator has a shape of `height=32,width=32,in-channel=8,out-channel=8`.

Height & Width Partition Factor	16	8	4	2	1
Latency before Partition (us)	6394				
Latency after Partition (us)	6416	6448	6528	6656	9216
Total Latency Overhead	0.34%	0.84%	2.09%	4.09%	44.13%
`BlockConcat` Latency (us)	28	48	108	281	819
Latency Ratio of `BlockConcat`	0.44%	0.74%	1.66%	4.23%	8.89%

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No.U21B2017.

REFERENCES

[1] Alessandro Capotondi et al. 2020. CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices. *IEEE TCAS-II* (2020), 871–875.
[2] Andrew G. Howard et al. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.
[3] Byung Hoon Ahn et al. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. *MLSys* (2020), 44–57.
[4] Barret Zoph et al. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*.
[5] Colby Banbury et al. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In *MLSys*. 517–532.
[6] Esteban Real et al. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*.
[7] Hanxiao Liu et al. 2019. DARTS: Differentiable Architecture Search. In *ICLR*.
[8] Jiequan Cui et al. 2019. Fast and Practical Neural Architecture Search. In *ICCV*.
[9] Jacob Devlin et al. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL* (2019), 4171–4186.
[10] Ji Lin et al. 2020. MCUNet: Tiny Deep Learning on IoT Devices. *NIPS* (2020).
[11] Ji Lin et al. 2021. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. *NIPS* (2021).
[12] Josse Van Delm et al. 2023. HTVM: Efficient Neural Network Deployment On Heterogeneous TinyML Platforms. In *DAC*. 1–6.
[13] Kaiming He et al. 2016. Deep residual learning for image recognition. In *CVPR*.
[14] Liangzhen Lai et al. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs.
[15] Manoj Alwani et al. 2016. Fused-layer CNN accelerators. In *MICRO*. 1–12.
[16] Mark Sandler et al. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *CVPR*.
[17] Muhammad Shafique et al. 2021. TinyML: Current Progress, Research Challenges, and Future Roadmap. In *DAC*. 1303–1306.
[18] Qingcheng Xiao et al. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *DAC*. 1–6.
[19] Robert David et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *MLSys* (2021), 800–811.
[20] Rafael Stahl et al. 2023. Fused depthwise tiling for memory optimization in tinyml deep neural network inference. *tinyML Research Symposium* (2023).
[21] Saining Xie et al. 2019. Exploring Randomly Wired Neural Networks for Image Recognition. In *ICCV*.
[22] Shuzhang Zhong et al. 2023. Memory-aware Scheduling for Complex Wired Networks with Iterative Graph Optimization. In *ICCAD*.
[23] Tianqi Chen et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *OSDI*. 578–594.
[24] Xiangzhong Luo et al. 2022. You only search once: on lightweight differentiable architecture search for resource-constrained embedded platforms. In *DAC*. 475–480.
[25] Yanchi Dong et al. 2023. A Model-Specific End-to-End Design Methodology for Resource-Constrained TinyML Hardware. In *DAC*. 1–6.
[26] Zihan Wang et al. 2022. Hierarchical memory-constrained operator scheduling of neural architecture search networks. In *DAC*. 493–498.
[27] Zhongzhi Yu et al. 2023. NetBooster: Empowering Tiny Deep Learning By Standing on the Shoulders of Deep Giants. In *DAC*. 1–6.
[28] Sudeep Pasricha. 2023. Lightning Talk: Efficient Embedded Machine Learning Deployment on Edge and IoT Devices. In *DAC*. 1–2.