Hierarchical Memory-Constrained Operator Scheduling of Neural Architecture Search Networks

Zihan Wang
Shanghai Jiao Tong University
Shanghai, China
wangzh99@sjtu.edu.cn

Chengcheng Wan
University of Chicago
Chicago, Illinois, United States
cwan@uchicago.edu

Yuting Chen*
Shanghai Jiao Tong University
Shanghai, China
chenyt@sjtu.edu.cn

Ziyi Lin
Alibaba Group Inc.
Shanghai, China
cengfeng.lzy@alibaba-inc.com

He Jiang
Dalian University of Technology
Dalian, China
jianghe@dlut.edu.cn

Lei Qiao
Beijing Institute of Control Engineering
Beijing, China
fly2moon@163.com

ABSTRACT

Neural Architecture Search (NAS) is widely used in industry, searching for neural networks meeting task requirements. Meanwhile, it faces a challenge in scheduling networks satisfying memory constraints. This paper proposes HMCOS that performs hierarchical memory-constrained operator scheduling of NAS networks: given a network, HMCOS constructs a hierarchical computation graph and employs an iterative scheduling algorithm to progressively reduce peak memory footprints. We evaluate HMCOS against RPO and Serenity (two popular scheduling techniques). The results show that HMCOS outperforms existing techniques in supporting more NAS networks, reducing 8.7~42.4% of peak memory footprints, and achieving 137~283× of speedups in scheduling.

1 INTRODUCTION

In recent years, Neural Architecture Search (NAS) is widely used in industry. It searches for neural networks meeting task requirements. Many NAS algorithms are proposed to expand design space and enhance their performance with increased accuracy under parameter and computation budgets [16]. The resulting networks are complicated—they may include irregular wirings and multiple levels of architectures, deviating from regular and almost sequential patterns in classic hand-crafted networks such as VGG [17] and ResNet [6].

As Figure 1 shows, when NAS networks are deployed, *schedules* must be figured out. These schedules describe execution plans of the corresponding *computation graphs*, guaranteeing the correctness and as well efficiency of computations. Schedules are strictly constrained by memory—networks can be deployed only when their schedules meet memory constraints.

*Yuting Chen is the corresponding author.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530472

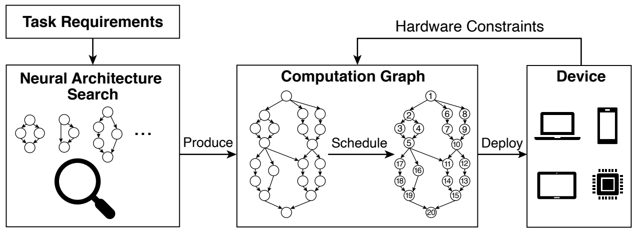


Figure 1: Neural Architecture Search (NAS) techniques search for the best network under task requirements. Represented as a computation graph, the network has to be scheduled before deployment, taking hardware constraints of target devices into account.

However, engineers face two challenges when scheduling NAS networks for memory-constrained devices, including mobile and embedded devices.

Challenge 1: Huge schedule space. The architectural characteristics of NAS networks indicate a huge number of possible schedules for a computation graph. Different schedules lead to different lifetimes of intermediate results produced by operators, which account for most of the runtime memory footprints. As Figure 2 shows, these schedules have significantly different peak memory footprints, and only 4% satisfy a given memory constraint (4.60 MB)[1].

Furthermore, it is time-consuming to exhaustively search in a huge schedule space for schedules with the lowest memory footprint. In Figure 2, the optimal peak memory footprint of NASNet-A [19] (for ImageNet dataset) is 4.13 MB, while none of the schedules with peaks lower than 4.40 MB is sampled within 1M samples.

Challenge 2: Diverse architectures. The NAS networks are usually composed of different cells. A network may contain either sequential or non-sequential cell-level architectures. Figure 3 shows the computation graph of NASNet-A at the operator and the cell levels. Schedules of these networks can possibly *interleave* multiple cells at one time, that is, executing an operator in one cell, then next operator in another cell. This variety is common in practice, requiring general-purposed scheduling algorithms to be designed.

Existing work. Existing deep learning frameworks/compilers (e.g., TensorFlow [1] and TVM [3]) are unaware of memory footprints

[1]We use peak memory footprints of classic networks as constraints. The peak of MobileNetV1 [7] for ImageNet is 4.60 MB.

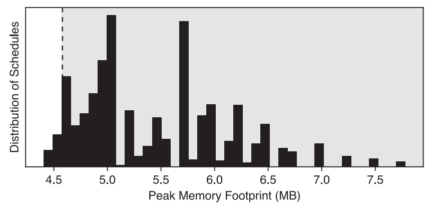
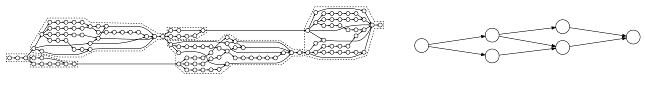


Figure 2: Peak memory footprint distribution of 1M samples in NASNet-A (for ImageNet) schedule space. The memory constraint is 4.60 MB in this example.



(a) Operator level. **(b) Cell level.**

Figure 3: Computation graph at two levels: at the operator level, there are irregular wirings between operators; at the cell level, cells form a non-sequential structure.

w.r.t. different network schedules. Serenity [2], the state-of-the-art, sets initial foot on this problem. It identifies each partial schedule by a signature and develops a DP-based scheduling algorithm. Serenity achieves the schedule with the lowest peak memory footprint on supported networks. Despite its effectiveness, Serenity has two limitations when applied in practice: (1) It only works for *hourglass-shaped* networks where cells are sequentially concatenated; (2) It usually takes hours to produce a schedule for a network.

Our work. Inspired by multiple levels of architecture, we propose HMCOS[2], the first hierarchical approach towards memory-constrained scheduling of NAS networks. HMCOS constructs a *hierarchical computation graph* and *iteratively* schedules operators on the hierarchical graph. The hierarhical graph greatly simplifies the complex architecture of a NAS network and effectively prunes the schedule space. The iterative scheduling algorithm disassembles only the peak-related vertices of the hierarchical graph, and thus manages to produce interleaving schedules without introducing too much workload.

This paper makes the following contributions:

(1) *Hierarchization.* We propose strategies for hierarchization of a computation graph by introducing two levels of hierarchy. The hierarchical graph is much more efficient in scheduling, and also provides opportunities for optimizing peak memory footprints.
(2) *Scheduling.* We design an iterative scheduling algorithm, which progressively reduces peak memory footprints through iterations, and three optimizations to further improve its efficiency.
(3) *Evaluation.* We evaluate HMCOS on several NAS networks. The evaluation results show that HMCOS (1) supports a large family of network architectures; (2) reduces 8.7~42.4% peak memory footprints compared with baseline reverse postorder scheduling; (3) saves 137~283× scheduling time compared with baseline Serenity.

2 PRELIMINARIES

This section introduces several basic concepts related to memory-constrained scheduling of neural networks.

[2]Source code is available at https://github.com/wzh99/HMCOS.

Definition 2.1 (Computation Graph and Schedule). One batch of neural network computation is modeled as a directed acyclic graph (DAG) $G = (V, E)$. Each vertex $v \in V$ represents an operator performing primitive computation. Each directed edge $\langle u, v\rangle \in E$ represents a data dependency between operators u and v.

A schedule $s = \langle v_1, v_2, \ldots, v_n\rangle$ is a topological ordering of G where $n = \#V$ and v_i specifies the operator to be executed at time i.

Definition 2.2 (Value and Operator). A value x is a result produced by an operator in the computation graph. $size(x)$ tells how much memory should be allocated to x. Let $in(v)$ be the input list $\langle x_1, x_2, \ldots, x_m\rangle$ where $m \in \mathbb{N}$ and $out(v)$ the output list $\langle y_1, y_2, \ldots, y_n\rangle$ where $n \in \mathbb{N}$. An operator v represents computation $out(v) = f(in(v))$. It is described as v *uses* values in $in(v)$ and *defines* values in $out(v)$.

Correspondingly, $def(x)$ is the operator defining a given value x, and $use(x)$ is a list of operators using x.

Given a neural network, we need to allocate memory space of each value properly so that the network is safely and correctly executed. Lifetime specifies the shortest time period of a value whose memory space must be available.

Definition 2.3 (Lifetime). Given a schedule s, the lifetime of a value x is an interval $[alloc(x), free(x)]$ where $alloc(x)$ is the time when the its defining operator is called and $free(x)$ is the time when it is *terminally used*, i.e., used by $v_{free(x)}$ and no longer used by following operators. Let $def(x) = v_i$, we have:

$$alloc(x) = i, \quad free(x) = \max_{v_j \in use(x)} j. \tag{1}$$

There are two alternate stages in neural network execution: the stage of executing an operator, and the stage between execution of two operators. We use *stable footprint* and *transient footprint* respectively to describe memory footprints in the two stages.

Definition 2.4 (Memory Footprint). For a schedule s of a computation graph G with n operators, let $\langle \mu^s_1, \mu^s_2, \ldots, \mu^s_n\rangle$ and $\langle \mu^t_0, \mu^t_1, \ldots, \mu^t_n\rangle$ be two footprint sequences, where $\mu^s_{1\le i\le n}$ is the *stable footprint* and $\mu^t_{0\le i\le n}$ is the *transient footprint* at time i:

$$\mu^s_i = \sum_{alloc(x)\le i\le free(x)} size(x), \tag{2}$$

$$\mu^t_i = \sum_{alloc(x)\le i\le free(x)-1} size(x). \tag{3}$$

The *peak memory footprint* μ_{peak} of schedule s is defined as the maximum of stable footprints: $\mu_{peak} = \max_i \mu^s_i$.

3 HMCOS DESIGN

The key idea of HMCOS is to hierarchically schedule a computation graph for achieving the lowest peak memory footprint. As shown in Figure 4, HMCOS contains two major components: (1) *Hierarchization of computation graphs* (§3.1), where operators in a computation graph is organized into a hierarchy; (2) *Iterative scheduling algorithm* that iteratively schedules operators on hierarchical graphs (§3.2).

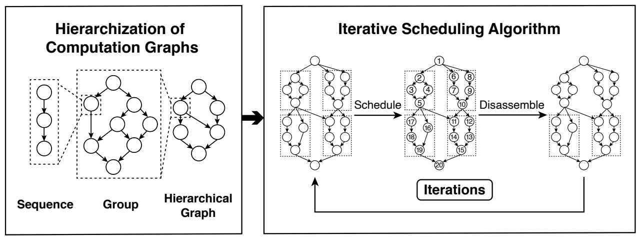

Hierarchization of Computation Graphs
Sequence
Group
Hierarchical Graph
Iterative Scheduling Algorithm
Schedule
Disassemble
Iterations

Figure 4: Overview of HMCOS.

3.1 Hierarchization of Computation Graphs

HMCOS hierarchizes computation graph to effectively prune schedule space before actual scheduling is performed. Thus we extend a computation graph to a hierarchical one.

Definition 3.1 (Hierarchical Graph). A hierarchical graph is a graph $G = (V, E)$ where each vertex $v \in V$ can either represent an *operator* or a *hierarchical vertex*. A hierarchical vertex v can be further expanded to another hierarchical graph $\mathcal{G}(v)$.

HMCOS introduces two levels, sequence and group, into hierarchical graphs.

3.1.1 Sequence. Operators can be joined as sequences. Despite diversity in wirings of NAS networks, sequential patterns are ubiquitous. If each of these sequences is treated as a single vertex, the schedule space is effectively reduced.

First, a sequence must look "sequential" in graph structure. It must be a list of operators $\langle v_1, v_2, \ldots, v_n \rangle$ where each operator has only one predecessor and one successor.

Second, the memory footprints of a sequence must not hinder cutting peaks. Let its memory footprints be $\mu^s_{1 \le i \le n}$ and $\mu^t_{0 \le i \le n}$. They must satisfy two conditions:

Condition 1. $\mu^s_1 \ge \mu^s_{2 \le i \le n}$.

Reason. If the first operator has the highest stable footprint in a sequence, the sequence is always safe to schedule as a whole. There is no risk of lifting the peak memory footprint after the first operator.

Condition 2. $\mu^t_i \ge \mu^t_j$ where $1 \le i < j \le n$.

Reason. A sequence should have non-increasing transient footprints. Otherwise it hinders cutting peak memory footprint when scheduling another hierarchical vertex, since it does not provide the lowest possible initial transient footprint for that vertex.

3.1.2 Group. NAS networks are usually built upon cells. Cells encapsulate certain degree of architectural complexity inside them; they also indicate different periods of computation, thus helping recognize the critical, peak-related parts. However, the scope of a cell is not memory-aware; a hierarchical graph created from cells may have high peak memory footprint, requiring more iterations to reduce.

For this reason, we design *groups* on sequences to alleviate the cost of scheduling. Sequences are grouped by taking four steps:

(1) Create initial groups from cells. The scopes of cells are provided by the NAS algorithm.
(2) For the output vertex v^o_i of each cell c_i, detect all vertices dominated by it and create set D_i of these vertices. Here a dominator tree is constructed by the technique in [11].
(3) For each D_i, find a connected subset C_i of D_i satisfying: (a) $v^o_i \in C_i$, (b) $\sigma_i = \sum_{\langle x,k \rangle \in OU(C_i)} size(x) \le \sum_{\langle x,k \rangle \in OU(v^o_i)} size(x)$, and (c) σ_i is the smallest. Here for a hierarchical vertex v, $OU(v) = \{\langle x, \#uses\_outside(x, v) \rangle\}$ is a mapping where x is a value defined inside $\mathcal{G}(v)$, and *uses_outside* produces a set of x's uses outside $\mathcal{G}(v)$.
(4) If such C_i is found, extract vertices $C_i - \{v^o_i\}$ from their original groups and create a new group for them.

3.2 Iterative Scheduling Algorithm

We design an iterative scheduling algorithm to progressively reduce peak memory footprints. As shown in Figure 4, the scheduling algorithm accepts a hierarchical graph and produces a schedule of it. The hierarchical graph is disassembled and rescheduled according to its peaks in memory footprints. The process repeats until the hierarchical graph cannot be further disassembled.

Algorithm 1 shows the iterative scheduling algorithm. It locates and disassembles the peak-related hierarchical vertices. Let $X_i = \{x \mid alloc(x) \le i \le free(x)\}$ be the set of alive values during i-th operator's execution. Suppose $\mu_{peak} = \mu^s_k$, HMCOS finds all *peak values*, i.e., alive values at the peak $X_{peak} = X_k$. For each peak value $x \in X_{peak}$, HMCOS finds its defining operator and the related sequence and group of x as well. Then it disassembles the corresponding hierarchical vertices. Only groups need to be disassembled to expose schedules interleaving groups. Sequences remain unchanged as they do not hide optimization opportunities.

During scheduling, memory footprints are computed incrementally by computing differences values $\Delta\mu^+_{1 \le i \le n}$ and $\Delta\mu^-_{1 \le i \le n}$:

$$\Delta\mu^+_i = \mu^s_i - \mu^t_{i-1} = \sum_{y \in out(v_i)} size(y), \tag{4}$$

$$\Delta\mu^-_i = \mu^s_i - \mu^t_i = \sum_{i = \max\limits_{v_j \in use(x)} j} size(x). \tag{5}$$

Equation 4 sums up size of each value y in $out(v_i)$. Equation 5 identifies whether a value x is *terminally used* by operator v_i. To compute Equation 5, HMCOS maintains a mapping $\mathcal{U}_i = \{\langle x, \#uses\_after(x, v_i) \rangle\}$ at each time i, where x is an alive value and *uses_after* retrieves a set of x's uses after v_i.

Algorithm 2 shows the scheduling algorithm on a fixed hierarchical graph. `ScheduleGraph` resembles the DP-based algorithm in Serenity and supports hierarchical graphs. `ScheduleVertex` processes a hierarchical vertex v: If v is a sequence, the algorithm directly computes memory footprints for it, using Equation 4 and 5. Otherwise, it calls `ScheduleGraph` to schedule $\mathcal{G}(v)$.

We then propose three optimizations to further improve efficiency of the scheduling algorithm.

Optimization 1: *Adaptive soft budgeting.*

This optimization prunes sub-optimal schedules as soon as possible. It reduces schedule space by defining a memory budget τ and eliminating partial schedules whose peak memory footprints are larger than τ. Different from Serenity, which relies on a hard-to-set hyperparameter to search for an appropriate budget, HMCOS employs μ_{peak} of last iteration for budget of current iteration. In Algorithm 2, `ScheduleVertex` and `ScheduleGraph` return `null` if the schedule satisfying the budget cannot be produced.

Algorithm 1: Iterative scheduling.

```
Input: hierarchical graph $G$
Output: schedule $s$, peak memory footprint $\mu_{peak}$
$s \leftarrow \langle\rangle, \mu_{peak} \leftarrow \infty$;
while true do
    $\mathcal{R} \leftarrow$ ScheduleGraph($G$, {});
    Compute $X_{peak}$ from $\mathcal{R}.s$;
    for $x \in X_{peak}$ do
        Find sequence $v_s$ that contains $def(x)$;
        if $v_s$ is in some group $v_g$ then disassemble $v_g$;
    if no group is disassembled then break;
    $s \leftarrow \mathcal{R}.s, \mu_{peak} \leftarrow \mathcal{R}.\mu_{peak}$;
```

Optimization 2: *Scheduling strategy selection.*

This optimization saves scheduling work on noncritical parts of the graph. HMCOS employs the DP-based method on groups with large memory footprints, while adopts simple scheduling method (e.g., reverse postorder) on the others. HMCOS computes peak memory footprint with simple scheduling and compares it with previous peak. It adopts simple scheduling if the peak is not lifted.

Optimization 3: *Group schedule memoization.*

This optimization reuses scheduling result inside and across iterations. The optimal schedule of a group g depends on (1) the hierarchical graph $\mathcal{G}(g)$ it expands to, and (2) whether values defined outside the group are *terminally used* in g. Only when a value is terminally used, its size is counted in some $\Delta\mu_i^-$, affecting its successive memory footprints (Equation 5). We define *value context* $\Gamma(g, \mathcal{U}_i)$ to characterize the environment of group scheduling:

$$\Gamma(g, \mathcal{U}_i) = \{\langle x, k = \mathcal{U}_i[x]\rangle \mid \langle x, k\rangle \in IU(g)\}. \tag{6}$$

Here for a hierarchical vertex v, $IU(v) = \{\langle x, \#uses\_inside(x, v)\rangle\}$ is a mapping where x is a value defined outside $\mathcal{G}(v)$, and *uses_inside* returns a set of x's uses inside $\mathcal{G}(v)$. During scheduling, if a group g with context $\Gamma(g, \mathcal{U}_i)$ has been optimally scheduled, the schedule is reused.

4 EVALUATION

4.1 Setup

Networks. We evaluate HMCOS with four popular NAS networks: NASNet-A [19], AmoebaNet-A [15] and DARTS [12] for both CIFAR-10 and ImageNet datasets; and RandWire [18] for ImageNet dataset. We arbitrarily take three RandWire architectures with parameter WS(4, 0.75). These networks have configurations for small computation regime, and can potentially be deployed on resource-constrained devices.

Baselines. We compare HMCOS with reverse postorder (RPO) scheduling and Serenity. RPO is a static scheduling method commonly used in deep learning compilers. Serenity is the state-of-the-art in memory-constrained operator scheduling. As Serenity is not publicly available, we implement our own version of Serenity.

Metrics. We evaluate HMCOS with following metrics:

(1) *Peak memory footprints* of schedules. We set memory constraints as 1.80 MB for CIFAR-10[3] and 4.60 MB for ImageNet.

[3]The peak memory footprint of DenseNet [8] for CIFAR-10 is 1.80 MB.

Algorithm 2: Scheduling a hierarchical graph.

```
Input: hierarchical graph $G$
Output: schedule $s$, peak memory footprint $\mu_{peak}$
function ScheduleVertex($v, \mathcal{U}_0$):
    if $v$ is a sequence $\langle v_1, v_2, \ldots, v_n\rangle$ then
        $\mu_0^t \leftarrow 0, \mu_{peak} \leftarrow -\infty$;
        for $i \in 1..n$ do
            $\mu_i^s, \mu_i^t, \mathcal{U}_i \leftarrow$ ComputeFootprint($v_i, \mu_{i-1}^t, \mathcal{U}_{i-1}$);
            $\mu_{peak} \leftarrow \max(\mu_{peak}, \mu_i^s)$;
        return $s, \mu_{peak}, \mu_n^t, \mathcal{U}_n$;
    else if $v$ is a group $g$ then
        return ScheduleGraph($\mathcal{G}(g), \mathcal{U}_0$);

function ScheduleGraph($G, \mathcal{U}_0$):
    $z_0 \leftarrow$ ZeroIndegree($\langle\rangle, G$);
    $\mathcal{M}_0 \leftarrow \{\langle z_0, (\langle\rangle, -\infty, 0, \mathcal{U}_0)\rangle\}$;
    $n \leftarrow \#G.V$;
    for $k \in 1..n$ do
        $\mathcal{M}_k \leftarrow \{\}$;
        for $\langle z_{k-1}, (s_{k-1}, \mu_{peak}, \mu_{-1}^t, \mathcal{U}_{-1})\rangle \in \mathcal{M}_{k-1}$ do
            for $w \in z_{k-1}$ do
                $\mathcal{R} \leftarrow$ ScheduleVertex($w, \mathcal{U}_{-1}$);
                $s_k \leftarrow s_{k-1} \cdot \mathcal{R}.s$;
                $\mu_{peak} \leftarrow \max(\mu_{peak}, \mu_{-1}^t + \mathcal{R}.\mu_{peak})$;
                $\mu_{-1}^t \leftarrow \mu_{-1}^t + \mathcal{R}.\mu_{-1}^t$;
                $z_k \leftarrow$ ZeroIndegree($s_k, G$);
                if $z_k \notin keys(\mathcal{M}_k) \lor \mu_{peak} < \mathcal{M}_k[z_k].\mu_{peak}$ then
                    $\mathcal{M}_k[z_k] \leftarrow (s_k, \mu_{peak}, \mu_{-1}^t, \mathcal{U}_{-1})$;
    return $\mathcal{M}_n[\{\}]$;
```

(2) *Arena size* of each schedule with arena-based allocation, which is adopted by popular edge deep learning frameworks, such as TensorFlow Lite[4]. It allocates a huge memory arena upfront and locates memory space of values by offsets. The arena size and offsets of values need to be precomputed from a schedule. We adopt the allocator in TensorFlow Lite to plan the memory.

(3) *Scheduling time* of each run. To fairly compare Serenity with HMCOS, we disable adaptive soft budgeting optimization of Serenity because it requires setting a hyperparameter which affects its scheduling efficiency.

4.2 Result

4.2.1 Performance. We evaluate supportability, effectiveness and efficiency of HMCOS against RPO and Serenity on all of the networks.

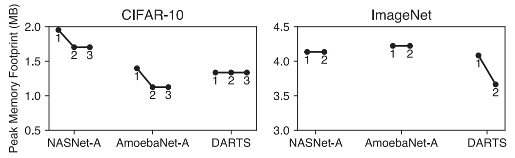
Supportability. HMCOS supports all the networks in our experiments. Comparatively, Serenity does not support NASNet-A, AmoebaNet-A and DARTS because cells in these networks are not sequentially concatenated; Serenity supports RandWire because of its hourglass-shaped architecture.

Effectiveness. Table 1 shows peak memory footprints of schedules achieved by RPO, Serenity and HMCOS. The result suggests that HMCOS is effective in reducing peak memory footprints, thus enabling deployment of more networks under memory constraints.

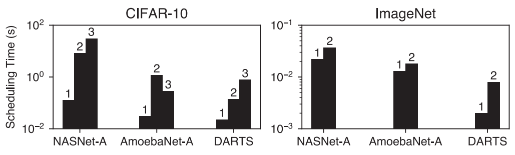
[4]https://www.tensorflow.org/lite/

Table 1: Performance of RPO (marked as R), Serenity (marked as S) and HMCOS (marked as H) on several NAS network architectures. Reduction rates of peak memory footprints and arena sizes are computed relative to RPO. For each peak memory footprint, we mark ✓ if it satisfies the memory constraint and × if it does not.

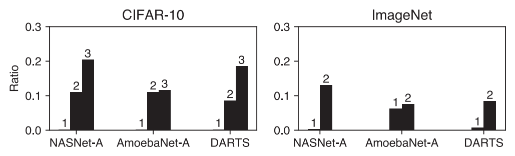
Dataset	Network	Peak memory footprint (MB)							Arena size (MB)				Scheduling time (s)		
		R		S		H		Reduction	R	S	H	Reduction	S	H	Speedup
CIFAR-10	NASNet-A	2.25	×	2.25	×	1.70	✓	24.3%	2.50	2.50	2.06	17.5%	-	40.05	-
	AmoebaNet-A	1.75	✓	1.27	✓	1.13	✓	35.7%	1.85	1.67	1.27	31.8%	-	1.57	-
	DARTS	2.32	×	1.69	✓	1.34	✓	**42.4%**	2.71	1.95	1.62	**40.3%**	-	0.97	-
ImageNet	NASNet-A	5.06	×	4.40	✓	4.13	✓	18.3%	5.45	4.79	4.93	9.5%	-	0.207	-
	AmoebaNet-A	4.92	×	4.29	✓	4.22	✓	14.2%	5.30	4.67	4.88	7.9%	-	0.065	-
	DARTS	4.91	×	4.53	✓	3.66	✓	25.3%	5.29	4.91	4.53	14.5%	-	0.042	-
	RandWire 1	5.37	×	4.43	✓	4.43	✓	17.4%	6.06	4.90	4.90	19.2%	5659	20	**283×**
	RandWire 2	5.37	×	4.90	×	4.90	×	8.7%	5.60	5.13	5.13	8.3%	5358	39	137×
	RandWire 3	5.83	×	4.20	✓	4.20	✓	28.0%	5.83	4.67	4.67	20.0%	10247	68	151×



(a) Peak memory footprint.



(b) Scheduling time.



(c) Ratio of freed sequences.

Figure 5: Performance and workload in each iteration of HMCOS. The index of the iteration is marked on each data point.

First, RPO fails to produce schedules under memory constraints at most of the time. Compared with RPO, HMCOS achieves 8.7~42.4% of reduction in peak memory footprints, and almost always produces schedules satisfying memory constraints.

Second, we slightly extend Serenity's divide-and-conquer strategy to handle cells in NASNet-A, AmoebaNet-A and DARTS. As this strategy ignores interleaving schedules, Serenity always produces schedules with higher peak memory footprints than HMCOS for such architectures.

HMCOS also benefits arena-based allocation by producing schedules with lower peak memory footprints. Table 1 lists arena sizes of schedules produced by RPO, Serenity and HMCOS. HMCOS reduces arena sizes by 7.9~40.3% compared with RPO. On most of the architectures, HMCOS produces schedules with arena sizes smaller than or equal to Serenity.

Efficiency. Table 1 shows scheduling time of Serenity and HMCOS. All of the architectures are scheduled by HMCOS within two minutes, and five of them are even done in two seconds. For three RandWire architectures, HMCOS achieves 137~283× speedup compared with Serenity. Note that we do not measure scheduling time of RPO on all architectures and Serenity on NASNet-A, AmoebaNet-A and DARTS, because they do not achieve peak memory footprints as low as HMCOS in these cases.

4.2.2 Iteration. We next evaluate how each iteration reduces the peak memory footprint of an architecture. We carry out experiments on NASNet-A, AmoebaNet-A and DARTS because they require at least two iterations of scheduling.

Figure 5 shows the peak memory footprints, scheduling time and ratio of freed sequences in each iteration of the scheduling. It takes only three iterations to terminate for all CIFAR-10 architectures and two iterations for all ImageNet architectures. Figure 5(a) shows that peak memory footprints are reduced in later iterations for half of the architectures. It denotes that iterative scheduling significantly reduces peak memory footprints. Figure 5(b) shows that even though disassembly increases scheduling workload, the increase in scheduling time is still acceptable. Figure 5(c) shows that at most 20% of the sequences are freed from their original groups—only a small part of the graph needs to be carefully scheduled. It explains why the scheduling time is acceptable after disassembly.

4.2.3 Ablation Study. We further investigate how the components of HMCOS affect its effectiveness and efficiency. Table 2 lists configurations used in our study. ④ is a complete HMCOS configuration, while others have fewer components than ④. ①~④ add levels of hierarchization to the configuration. ③ directly organizes sequences as cells in the original architectures instead of creating groups by the method in §3.1.2. Each of ⑤~⑦ removes one scheduling optimization.

We evaluate these configurations on NASNet-A, AmoebaNet-A and DARTS, and compute peak memory footprints for all schedules. The results show that the effectiveness of HMCOS is affected by the group level hierarchization strategy, rather than the sequence level hierarchization and the three optimizations. NASNet-A for CIFAR-10 has peak memory footprint 1.83 MB under ③ instead of 1.70 MB under other configurations. This indicates that organizing sequences as groups is better than directly following the cells in the original architectures, because groups catch opportunities for optimization.

Scheduling time under different configurations are listed in Table 3. For ③ and ④, we compute their speedup ratio compared with their previous configurations (② and ③ respectively). For ⑤~⑦, we compute their slowdown ratio compared with ④. It can be seen that sequence level hierarchization makes scheduling of these architectures tractable in acceptable time. Group level hierarchization

Table 2: Configurations during the ablation study.

Configuration	①	②	③	④	⑤	⑥	⑦
Hierarchization	None	Sequence	Sequence+Cell	Complete	Complete	Complete	Complete
Scheduling	Complete	Complete	Complete	Complete	w/o Adaptive soft budgeting	w/o Scheduling strategy selection	w/o Group schedule memoization

Table 3: Time (in seconds) for scheduling architectures. ↑ and ↓ represent speedup and slowdown ratios, respectively.

Dataset	Network	①	②	③		④		⑤		⑥		⑦	
		Time	Time	Time	↑	Time	↑	Time	↓	Time	↓	Time	↓
CIFAR-10	NASNet-A	>3,600	836.39	271.01	3.09×	40.05	6.77×	49.05	1.22×	40.12	1.00×	47.02	1.17×
	AmoebaNet-A	>3,600	99.70	8.80	11.32×	1.57	5.61×	5.42	3.45×	1.70	1.09×	1.83	1.17×
	DARTS	>3,600	19.58	5.06	3.87×	0.97	5.20×	1.11	1.15×	1.05	1.06×	1.04	1.08×
ImageNet	NASNet-A	>3,600	457.79	2.16	212×	0.207	10.42×	0.96	4.63×	1.09	5.25×	0.212	1.02×
	AmoebaNet-A	>3,600	76.53	0.73	105×	0.065	11.25×	0.29	4.51×	0.20	3.02×	0.069	1.06×
	DARTS	1,139	117.30	0.023	422×	0.042	0.55×	0.063	1.50×	0.20	4.64×	0.042	1.00×
	Geomean	-	-	-	32.9×	-	4.83×	-	2.31×	-	2.10×	-	1.08×

further raises scheduling efficiency. The three optimizations are complementary in improving efficiency of scheduling.

5 RELATED WORK

Neural Architecture Search. Most of the studies on NAS [12, 15, 19] propose methods to efficiently search for a specific network architecture in a manually-designed search space. Some other techniques, such as RandWire [18], switch from searching for specific architectures to network generators. Many of these NAS networks have characteristics that existing systems are usually unaware of, such as irregular wirings [2] and low computation density per operator [4]. To enable efficient deployment of these networks, there is plenty of room for improvement in existing systems.

Memory optimization. Neural networks are memory-consuming. Extensive studies have been done to optimize memory footprints of training [10, 14]. With increasing demand for deploying neural networks on resource-constrained devices, it is also important to optimize memory footprints of *inference* on these devices. There exist many other approaches, including model compression [5] and graph rewriting [9]. Comparatively, Serenity [2] and HMCOS take a different tack in that they optimize peak memory footprints of networks through memory-constrained scheduling.

Inter-operator scheduling. There are two levels of scheduling for execution of neural networks: intra-operator and inter-operator. Intra-operator scheduling orchestrates computation inside an operator, while inter-operator scheduling produces an execution plan of operators in the computation graph. Existing studies [4, 13] mainly focus on reducing latency by leveraging inter-operator parallelism. Serenity performs inter-operator scheduling for reducing peak memory footprints of irregularly wired networks. HMCOS also performs inter-operator scheduling, but in a hierarchical, and thus much more efficient style.

6 CONCLUSION

Memory-constrained scheduling is important for deploying NAS networks on memory-constrained devices. This paper proposes HMCOS for scheduling NAS networks with memory constraints. HMCOS constructs a hierarchical computation graph and iteratively schedules operators on the hierarchical graph. Our evaluation shows supportability, effectiveness and efficiency of HMCOS in scheduling NAS networks.

ACKNOWLEDGMENTS

This research is supported by National Natural Science Foundation of China (Grant No. 62032004). This work was supported by CCF-Huawei Innovative Research programme (TC20210701006/CCF2021-admin-270). Yuting Chen was also supported by Alibaba Group through Alibaba Innovative Research (AIR) programme. This work is also partially supported by the National Key R&D Program of China (2018YFB1003903).

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
[2] Byung Hoon Ahn et al. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. In *MLSys*, Vol. 2. 44–57.
[3] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. 579–594.
[4] Yaoyao Ding et al. 2021. IOS: Inter-Operator Scheduler for CNN Acceleration. In *MLSys*, Vol. 3. 167–180.
[5] Song Han et al. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*.
[6] Kaiming He et al. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.
[7] Andrew G. Howard et al. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]
[8] Gao Huang et al. 2017. Densely Connected Convolutional Networks. In *CVPR*. 2261–2269.
[9] Zhihao Jia et al. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *MLSys*, Vol. 1. 27–39.
[10] Marisa Kirisame et al. 2021. Dynamic Tensor Rematerialization. In *ICLR*.
[11] Thomas Lengauer et al. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141.
[12] Hanxiao Liu et al. 2019. DARTS: Differentiable Architecture Search. In *ICLR*.
[13] Lingxiao Ma et al. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *OSDI*. 881–897.
[14] Xuan Peng et al. 2020. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *ASPLOS*. 891–905.
[15] Esteban Real et al. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*. 4780–4789.
[16] Pengzhen Ren et al. 2021. A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions. *ACM Comput. Surv.* 54, 4, Article 76 (2021).
[17] Karen Simonyan et al. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.
[18] Saining Xie et al. 2019. Exploring Randomly Wired Neural Networks for Image Recognition. In *ICCV*. 1284–1293.
[19] Barret Zoph et al. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*. 8697–8710.