SAFE OPTIMIZED STATIC MEMORY ALLOCATION FOR PARALLEL DEEP LEARNING

Ioannis Lamprou [1] **Zhen Zhang** [1] **Javier de Juan** [1] **Hang Yang** [1] **Yongqiang Lai** [2] **Etienne Filhol** [1] **Cedric Bastoul** [1]

ABSTRACT

Parallel training is mandatory in order to maintain performance efficiency and tackle memory constraints for deep neural network (DNN) models. For this purpose, a critical optimization in order to tune a parallelism strategy is to schedule tensors onto device memory in compilation time. In this paper, we present a safe and optimized solver for this problem capturing a general parallel scenario to enable execution in open-source MindSpore framework. The input is a computational graph and a partition of its operators into streams of execution, which may run in parallel. First, we design algorithms to efficiently and provably decide if it is safe, for any two tensors, to reuse memory. Second, given such a set of reuse constraints, as well as a set of contiguous constraints to enable bulk communication among processing elements, we design algorithms to assign an offset to each tensor, such that all constraints are satisfied and total memory is minimized. Our experiments in parallel training of a variety of DNNs demonstrate nearly optimal, improved in some cases, memory consumption compared to state-of-the-art (adapted for our setting) and a sequential execution lower bound. Our algorithms show compilation time gains of up to 44% in determining safety and up to 70% in tensor offset assignment.

1 INTRODUCTION

Modern deep learning frameworks are used to execute *Deep Neural Networks* (DNNs) of a multi-layered structure with each layer consisting of neural operators and tensors, that is, data sent among operators. A significant development is the emergence of more and more DNNs of very large size both in depth and width of layers. Salient examples include T5 (Raffel et al., 2020), GPT-3 (Brown et al., 2020), and PanGu-α (Zeng et al., 2021). Training in parallel is mandatory in order to maintain performance efficiency and tackle memory constraints (Narayanan et al., 2021).

Modern AI accelerator devices are equipped with multiple intensive parallel *Processing Elements* (PEs), but limited memory size (Chen et al., 2020). To enable big model training in such environments, two sets of classic memory variables, *parameters* and *intermediate tensors*, have to be mapped onto different devices. Parameters are kept forever in memory in order to avoid communication overhead. The memory footprint of parameters is computed during compilation as the sum of their sizes. Intermediate tensors are produced and consumed among layers during training. There is no need to allocate exclusive memory per tensor. Tensors may overlap in memory if not needed concurrently.

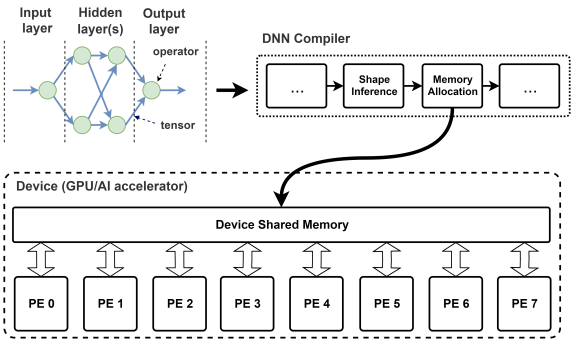


Figure 1: Static Allocation: given a DNN, the compiler decides a memory plan, then maps it onto physical device memory shared by multiple processing elements (PEs).

Traditional dynamic allocators may be used to manage allocation and release of intermediate tensors in runtime (Google, 2022). Handling fragmentation and runtime out-of-memory (OOM) via reallocation or garbage collection are necessary in order to avoid training interruption and relaunching, which are often very expensive. To escape these issues and the overhead they incur, an alternative method is *static allocation*. In this setting, it is a prerequisite that the shape of each intermediate tensor is inferred in compilation time, as shown in Figure 1.

[1]Huawei Technologies France, Paris, France [2]Huawei Technologies Co., Ltd., Shenzhen, China. Correspondence to: Ioannis Lamprou <ioannis.lamprou@huawei.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

With static allocation, the DNN is compiled on host, then it is loaded and executed on the hardware until termination. Most control communication and bidirectional data movement between host and device is avoided; only input data sets and output results are communicated. Moreover, as parameters and intermediate tensors memory footprints are computed in compilation time, we can tune the parallelism strategy in order to reduce the number of devices to the benefit of the final user (Zheng et al., 2022).

We present a safe optimized memory allocation system to enable *parallel* deep learning in the static execution model. The scarcity of static memory allocators for optimized memory management in such contexts creates a big challenge. Optimizing quality and solving performance to catch up with DNNs of growing size is a strong motivation.

Optimizing memory usage for *sequential* execution is well studied, for example in Chainer (Sekiyama et al., 2018) and TFLite (Lee et al., 2019; Pisarchyk & Lee, 2020). In these works, the proposed algorithms allocate tensors into a single buffer in some greedy order. To do so, every tensor already in memory is examined for lifetime overlap with the tensor to be inserted. In total, this yields quadratic time complexity. Yet, related work fails to address generalized *parallel* execution. For the latter, the graph is partitioned into multiple execution streams, which bring about a superset of constraints. Many additional communications are brought in to aggregate the distributed results. Communication tensors are concatenated to use maximal I/O bandwidth, thus also giving rise to *contiguous* constraints (see also tensor fusion in (Sergeev & Del Balso, 2018)). Overall, we tackle two limitations of the state of the art:

1. Global lifetime comparison used to determine safety of memory reuse among tensors in sequential execution is not enough to enable safe parallel deep learning.

2. As DNNs grow large, quadratic time complexity will be a significant burden with respect to parallel strategy tuning during compilation.

In this paper, we make the following contributions:

1. Compute a *provably* safe minimal set of tensor memory reuse (no-overlap) constraints to enable parallel deep learning. We propose two approaches: a standard and an optimized one yielding compilation time benefits of up to 44%.

2. Design novel tensor offset calculation algorithms given safe-reuse and contiguous constraints. We tackle the time complexity wall in practice (up to 70%), while maintaining nearly optimal or improved footprint.

3. Implement our solution in an open-source production framework and provide experimental validation.

Outline. In Section 2, we define *Offset Calculation for Parallel*. In Section 3, we design efficient algorithms to determine memory reuse safety. In Section 4, we design efficient constraints-respecting algorithms to assign tensor offsets such that footprint is minimized. In Section 5, we validate our methods in an open-source framework. We discuss related works in Section 6 and conclude in Section 7.

2 Problem Description

We represent a DNN as a *Directed Acyclic Graph* (DAG), also called a *computational graph*, with nodes representing operators and arcs representing tensors between operators. A tensor t generated at node v is shipped via arcs whose origin is v. If u receives tensor t from v, then arc (v, u) is assigned label t. There may be multiple arcs (tensors shipped) between two operators. Operator "workspace" memory for node v may be modeled by adding a self-loop tensor (v, v).

Our goal is to efficiently store all tensors required during execution, that is, to *minimize* the size of the logical memory buffer. To do so, we must determine dependency relations among tensors. Two tensors not needed at the same time may be stored in overlapping memory. Specifically, the memory occupied by a tensor t may be reused by other tensors *(i)* before the producer operator of t executes and *(ii)* after all operators consuming t execute. Each tensor is assigned a *start offset*. Its end offset is given by *start offset* + *tensor size - 1*. Once fixed, tensor offsets do not change.

Before defining the problem we tackle, let us consider the special case of *sequential* execution of operators. The global sequence in which operators execute directly dictates data dependencies. The sequence can be viewed as a traversal of nodes. Example traversals include breadth-first-search, depth-first-search, etc. A traversal of nodes in some order gives us a *topological sorting*: each node is assigned a natural number corresponding to its execution order in the sequence. Such a topological sorting helps us determine the *lifetime* of each tensor: the time interval during which the tensor needs to be stored. Let $ts(v)$ stand for the order of node v in the topological sorting. We define the lifetime of a tensor t generated at node v and sent to $u_1, u_2, \ldots, u_k$ as $L_t = [ts(v), \max_{i=1,2,\ldots,k} ts(u_i)]$. Two tensors have a no-overlap constraint, if their lifetime intervals overlap.

Definition 1 (Offset Calculation)**.** *Given a topologically sorted DNN, return a start offset for each tensor, such that no two tensors $t_1, t_2 \in T$, where $L_{t_1} \cap L_{t_2} \neq \emptyset$, overlap in memory and the total storage used is minimized.*

In this paper, we consider generalized Offset Calculation, where nodes execute in *parallel* and multiple processing elements share the device memory. Each node is assigned to a *stream* of execution handled by a processing element. No assumption is made about the execution of nodes assigned

to different streams. In order to ensure safety, the set of no-overlap constraints may be larger than in the sequential case. It suffices that two tensors are needed simultaneously in one execution scenario for them to be a no-overlap. However, this execution path may not be realized in practice!

For an introductory example, see Figure 2. The nodes of the DAG are assigned a global topological order $\{1, 2, \ldots, 8\}$, which we use to identify them, whereas tensors are identified by small letter labels $\{a, b, \ldots, i\}$. There are three streams defined: stream 0 containing vertices 1 and 3, stream 1 with vertices $2, 4, 5, 6$, and stream 2 with vertices 7 and 8. Note that a local topological order per stream can be extracted by the sequence in the global topological order: for instance, in stream 0, node 1 will execute before node 3.

According to the global sorting, in the single-stream case, it would be safe to reuse tensors a and f, since the lifetime of a, namely $[1, 3]$, does not overlap the one of f, namely $[4, 6]$. Nonetheless, this is *not* a safe reuse in the multi-stream case: operator 4 might actually execute before, or in parallel to, operator 3, since there is no preguaranteed execution order between streams 0 and 1. In other words, overwriting tensor a by tensor f could result in operator 3 lacking proper input when selected for execution, hence causing a runtime error.

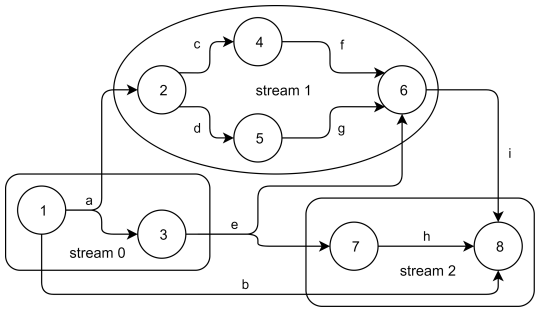


Figure 2: Example DNN DAG partitioned into three streams

Definition 2 (Offset Calculation for Parallel)**.** *Given a multi-stream DNN, return a start offset for each tensor, so that no two tensors overlap in memory, if they might be needed simultaneously in memory during parallel execution, and the total storage used is minimized.*

In the following two sections, we define both subproblems of *Offset Calculation for Parallel* and design algorithms to solve them. First, for any two tensors, we determine whether it is safe to reuse their offset intervals. Second, we assign an offset to each tensor, such that the computed set of no-overlap constraints is respected and footprint is minimized. We also support additional constraints on tensors, which require related tensors to be in spatially contiguous storage.

3 MULTI-STREAM SAFETY

In sequential execution, a tensor's lifetime is associated to two nodes: the one producing it and the last one consuming it. To determine a no-overlap constraint between two tensors, it suffices to examine their corresponding lifetimes.

In this section, our interest lies in the safe parallel execution of nodes, in other words, a *multi-stream* model. The set of nodes is partitioned into *streams* of execution. Each node is assigned to a single stream. Each stream is assigned its own topological sorting, which dictates the sequential order $\{1, 2, \ldots\}$ in which nodes are executed within the stream. To transform a local topological sorting into graph-theoretic dependency, for each $i \in \{1, 2, \ldots\}$, if originally there is no arc $(i, i+1)$, we add such an arc with a fresh label associated with a new tensor of size 0. No predefined assumptions are made on the relative execution order among streams. Given two operators in different streams, a global lifetime comparison is no longer pertinent in order to determine their potential runtime dependency. The introductory example, see Figure 2, acutely reveals the need for a more general set of rules to capture the increased number of conflicts arising during a parallel execution of streams. We are now ready to define the modeling problem by introducing the notion of *safe pairs*. Below, for some finite set X, let $\binom{X}{2}$ denote the set of unordered pairs of elements of X. More formally, we define $\binom{X}{2} := \{\{x_1, x_2\} \mid x_1, x_2 \in X \text{ and } x_1 \neq x_2\}$.

Definition 3 (Safe Pair)**.** *Assume we are given a multi-stream DNN with tensor set T. An unordered pair of tensors $\{t_1, t_2\} \in \binom{T}{2}$ is called a **safe pair** if we do not need to store t_1 and t_2 concurrently in memory for **any** potentially realized parallel execution of the DNN.*

Definition 4 (Multi-Stream Safety)**.** *Given a multi-stream DNN with tensor set T, for each pair $\{t_1, t_2\} \in \binom{T}{2}$ decide whether $\{t_1, t_2\}$ is a safe pair.*

If an unordered pair of tensors is not a safe pair, we refer to it as an *unsafe pair*, which corresponds to the notion of no-overlap constraint we discussed. Note that a decision for each pair in $\binom{T}{2}$ is necessary, since they are all considered in Single Object algorithms (to be defined later in Section 4). Thus, we make all decisions required in Definition 4 a priori and later use them to make decisions on offset assignments.

3.1 Computational Graph Based Approach

A first idea to solve Multi-Stream Safety, independently of stream definitions, is to use tensor ancestor relationships in the DAG to decide on safety of reuse. Intuitively, a tensor t_1 may reuse memory with a tensor t_2, if t_1 is consumed before t_2 is produced or vice versa. In other words, if due to graph topology, all consumers of a tensor execute before the producer of another tensor, then the two tensors may reuse memory regardless of stream definitions.

For each tensor, let us define the set of *ancestor tensors*. To do so, we start by nodes: for each node n, let $AncNodes[n]$ contain all nodes n' such that there is a directed path of length at least 1 from n' to n; we use notation $n' \rightsquigarrow n$ to refer to this. Also, $n \notin AncNodes[n]$, that is, we ignore self-arcs in this definition. For a tensor t, let $DestNodes[t]$ be the set containing all nodes consuming t. For a tensor t produced at node n, let $AncTensors[t]$ contain all tensors t' for which it holds $DestNodes[t'] \subseteq AncNodes[n]$. Tensors t_1 and t_2 may reuse memory if either $t_1 \in AncTensors[t_2]$ or $t_2 \in AncTensors[t_1]$, see Algorithm 1.

Algorithm 1: Ancestor Tensors

Input : A DNN with node set N and tensor set T.
Output : A set $U \subseteq \binom{T}{2}$ of unsafe pairs.

```
1 U ← (T 2);
2 foreach {t1, t2} ∈ (T 2) do
3 |  if t1 ∈ AncTensors[t2] OR t2 ∈ AncTensors[t1] then
4 |  |  U ← U \ {{t1, t2}};
5 return U;
```

Theorem 1. *Algorithm 1 solves Multi-Stream Safety.*

There is one way to optimize the runtime of Algorithm 1 by taking advantage of stream definitions. For a tensor t, we substitute $DestNodes[t]$ by a (potentially much smaller depending on DAG topology) subset $MaxNodes[t]$, which contains a single node per destination stream of t, that is, the one with the maximum local topological order among the destination nodes of t in this stream. Let $N_s \subseteq N$ denote the set of nodes within stream s, $s(n)$ denote the stream within which lies node n and $StartTime[n]$ denote the topological order of node n within $s(n)$. Formally, we have $MaxNodes[t] := \{n \in DestNodes[t] \mid \forall n' \in N_{s(n)}$ it holds $StartTime[n] \geq StartTime[n']\}$.

Consider a destination stream of a tensor t, namely stream s, and let $n^* \in MaxNodes[t]$ denote the destination node of t in stream s with maximum local order. Recall all within-stream arcs $(i, i+1)$ exist as discussed above. Then, for every $n' \in DestNodes[t] \cap N_s$ it holds $n' \rightsquigarrow n^*$ by within-stream arcs. For any node $n \in N$ it follows, if $n^* \rightsquigarrow n$, then $n' \rightsquigarrow n$. By applying this idea to each destination stream of t, we get that if $MaxNodes[t] \subseteq AncNodes[n]$, then $DestNodes[t] \subseteq AncNodes[n]$. On the other hand, since by definition of $MaxNodes$ it holds $MaxNodes[t] \subseteq DestNodes[t]$, then $DestNodes[t] \subseteq AncNodes[n]$ implies $MaxNodes[t] \subseteq AncNodes[n]$.

Overall, the above imply $DestNodes[t] \subseteq AncNodes[n]$ if and only if $MaxNodes[t] \subseteq AncNodes[n]$, therefore we can substitute $DestNodes$ by $MaxNodes$ within the $AncTensors$ check and obtain equivalent result. We refer to this optimized version of Algorithm 1 as 1_{opt}.

Corollary 1. *Algorithm 1_{opt} solves Multi-Stream Safety.*

3.2 Stream Graph Based Approach

Here, we present a set of algorithms to decide Multi-Stream Safety by considering ancestor relationships among streams. The idea is to employ information from the *stream graph*, that is, the implied graph of streams, in order to simplify decisions on safe pairs. Let $G_D = (N(G_D), A(G_D))$ stand for the stream graph corresponding to computational graph $D = (N, A)$. We define the node set $N(G_D) := S$, where S is the set of streams used in D. We define the arc set as $A(G_D) := \{(s, s') \mid \exists\, n \in N_s, n' \in N_{s'}$ such that $(n, n') \in A\}$. We design an algorithm for tensor pairs generated in streams with ancestor relationship (Algorithm 3) and another for tensor pairs generated within the same stream (Algorithm 5).

Let us describe preliminary notions used in the algorithms to follow. A tensor belongs to the stream of the node producing it. The set of tensors produced in stream s is denoted by $T_s \subseteq T$. Respectively, let $N_s \subseteq N$ denote the set of nodes in stream s. A tensor is called *between-streams*, if there is at least one node consuming it which does not belong to the same stream as the node producing it. For any tensor t, and stream s, let $EndTime[t][s]$ denote the maximum order of a node in stream s receiving tensor t. If t is not between-streams, we may simplify this notation to $EndTime[t]$. Respectively, let $StartTime[t]$ denote the lifetime start of tensor t: the order of the node producing t. For example, in Figure 2, we have $StartTime[a] = 1$, $EndTime[a][0] = 3$, and $EndTime[a][1] = 2$.

Regarding streams, we say s_2 is an *ancestor* of s_1, if there exist nodes n_2 in stream s_2 and n_1 in stream s_1 such that $n_2 \rightsquigarrow n_1$. Let $AncStreams[s]$ stand for the set of ancestor streams of s. In Figure 2, we get $AncStreams[0] = \{\}$, $AncStreams[1] = \{0\}$ and $AncStreams[2] = \{0, 1\}$.

For a tensor t_1 produced at node n_1 in stream s_1, and a stream s_2 ancestor of s_1, let $MaxAncId[t_1][s_2]$ denote the maximum topological order among all nodes in s_2 from which there is a directed path to n_1. If there is no such node, $MaxAncId$ is set to 0. Intuitively, t_1 may reuse the memory of any tensor in s_2 that ends not later than its ancestor tensor ending most late in s_2. In Figure 2, we get $MaxAncId[g][0] = 1$ and $MaxAncId[h][0] = 3$, but $MaxAncId[h][1] = 0$. Finally, let $DestStreams[t]$ stand for the set of streams containing at least one node consuming tensor t. In Figure 2, we get $DestStreams[a] = \{0, 1\}$, $DestStreams[c] = \{1\}$, and $DestStreams[e] = \{1, 2\}$.

Before proceeding with detailed algorithm descriptions, in Algorithm 2, we show how they are used together to solve Multi-Stream Safety. Initially, we assume any $\{t_1, t_2\} \in \binom{T}{2}$ is an unsafe pair (line 1) and we denote them as set U. We run Algorithm 3 with all unsafe pairs as input to allow reuse for some tensor pairs in ancestor streams (line 2). The set of remaining unsafe pairs $U' \subseteq U$ is returned. Then,

Algorithm 5 is called with U' as input in order to allow reuse for some tensor pairs generated in the same stream (line 3). If two tensors are in different streams without any ancestor relationship, they are not considered as a pair by either algorithm, so by default they are deemed unsafe.

Algorithm 2: Stream Graph Safety

Input : A DNN with stream set S and tensor set T.
Output : A set $U'' \subseteq \binom{T}{2}$ of unsafe pairs.

1 $U \leftarrow \binom{T}{2}$;
2 $U' \leftarrow AncestorStreamsReuse(S, T, U)$;
3 $U'' \leftarrow SameStreamReuse(S, T, U')$;
4 **return** U'';

Let us describe Algorithm 3. For each stream we consider all tensors in its ancestor streams (lines 1 to 3). Our safety rule depends on whether an ancestor tensor is between streams.

Algorithm 3: Ancestor Streams Reuse

Input : A set of streams S, tensors T, unsafe pairs U.
Output : A subset of U after removing safe pairs of tensors in streams with ancestor relationship.

1 **foreach** $s_1 \in S$ **do**
2 **foreach** $s_2 \in AncStreams[s_1]$ **do**
3 **foreach** $t_2 \in T_{s_2}$ **do**
4 **if** t_2 *is* ***not*** *between streams* **then**
5 **foreach** $t_1 \in T_{s_1}$ **do**
6 **if** $EndTime[t_2] \leq MaxAncId[t_1][s_2]$ **then**
7 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
8 **else** // t_2 is between streams
9 **foreach** $t_1 \in T_{s_1}$ **do**
10 **if** $ValidAncestor(t_2, t_1)$ **then**
11 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
12 **return** U;

If ancestor tensor t_2 is not between streams (lines 4 to 7), for each tensor t_1 in current stream s_1, it suffices to check whether $EndTime[t_2]$ within ancestor stream s_2 does not exceed $MaxAncId[t_1][s_2]$. Intuitively, if t_1 is to overwrite t_2 in memory, then t_2 must not be needed any longer than the execution of the last ancestor node in s_2 needed by t_1. In case it holds $MaxAncId[t_1][s_2]$ equal 0, then there is no safe ancestor node in s_2 for t_1. Therefore, reuse is disallowed since $EndTime[t_2] > 0$ by the local topological order. In Figure 2, tensor h is not allowed to reuse any tensor generated in stream 1, since $MaxAncId[h][1] = 0$.

Contrarily, if ancestor tensor t_2 is between streams (lines 8 to 11), we generalize the rule and request t_2 to be a *valid ancestor* of t_1, see Algorithm 4. In other words, the set of destination streams of t_2 needs to be a *valid subset* of the set containing the ancestor streams of s_1 and s_1 itself, where s_1 is the source stream of t_1. It must hold $DestStreams[t_2] \subseteq AncStreams[s_1] \cup \{s_1\}$ and, for any $s \in DestStreams[t_2]$, the following rule must apply: if $s \in AncStreams[s_1]$, then $EndTime[t_2][s] \leq MaxAncId[t_1][s]$, otherwise, if $s = s_1$, then $EndTime[t_2][s_1] < StartTime[t_1]$. Intuitively, in case a destination stream of t_2 is an ancestor stream of t_1, the same rule as in line 6 in Algorithm 3 needs to be respected. Otherwise, in case the destination is the current stream ($s = s_1$), then all t_2 consumers in s_1 must execute strictly before t_1 is produced in order to ensure safety with certainty. If at least one destination stream of t_2 does not conform to the rule, we cannot ensure safety: t_1 and t_2 may be needed simultaneously in storage during execution.

Algorithm 4: Valid Ancestor (t_2, t_1)

Input : A tensor t_1 produced in stream s_1 and a tensor t_2 produced in (ancestor) stream s_2.
Output : True if t_2 is a valid ancestor of t_1, else False.

1 **if** $DestStreams[t_2] \not\subseteq AncStreams[s_1] \cup \{s_1\}$ **then**
2 **return** *False*;
3 **foreach** s *in* $DestStreams[t_2]$ **do**
4 **if** $s \in AncStreams[s_1]$ **then**
5 **if** $EndTime[t_2][s] > MaxAncId[t_1][s]$ **then**
6 **return** *False*;
7 **if** $s = s_1$ **then**
8 **if** $EndTime[t_2][s_1] \geq StartTime[t_1]$ **then**
9 **return** *False*;
10 **return** *True*;

To showcase the valid ancestor rule, consider two examples in Figure 2. Tensor h does not form a safe pair with tensor a. Although it holds $DestStreams[a] = \{0, 1\} \subseteq \{0, 1, 2\} = AncStreams[2] \cup \{2\}$ and $EndTime[a][0] = 3 \leq 3 = MaxAncId[h][0]$, we notice $EndTime[a][1] = 2 > 0 = MaxAncId[h][1]$. Indeed, node 7 may execute before node 2 is done consuming a, so it is unsafe to have h overwrite a in storage. In another case, consider tensor i in stream 1 and a in stream 0. It holds $DestStreams[a] = \{0, 1\} \subseteq \{0, 1\} = AncStreams[1] \cup \{1\}$. For any stream where a is consumed, there is no violation to the valid ancestor rule. We get $EndTime[a][0] = 3 \leq 3 = MaxAncId[i][0]$, and $EndTime[a][1] = 2 < 6 = StartTime[i]$, so $\{a, i\}$ is safe.

Lemma 1. *Algorithm 4, $ValidAncestor(t_2, t_1)$, returns True if and only if for all $n' \in DestNodes[t_2]$ it holds $n' \rightsquigarrow n_1$, where $n_1 \in N_{s_1}$ is the node producing t_1.*

Let us now describe Algorithm 5. For any pair of tensors generated in the same stream, where both of them are not between-streams, it suffices to employ the standard lifetime comparison, as in the single-stream case (lines 14 to 16). Formally, in this case, a pair $\{t_1, t_2\}$ is safe if and only if $[StartTime[t_1], EndTime[t_1]] \cap [StartTime[t_2], EndTime[t_2]] = \emptyset$. Two between-stream tensors generated in the same stream are allowed to reuse memory if one is a valid ancestor of the other (lines 5 to 7) using the same check as in Algorithm 3. If only one of them is between-streams, they are deemed a safe pair only if the between-streams tensor starts strictly after the lifetime end of the within-stream tensor, or before,

but also satisfying valid ancestor in this case (lines 8 to 13). In Figure 2, tensors b and e, both generated in stream 0, are an unsafe pair. Neither b is a valid ancestor of e, since $DestStreams[b] = \{2\} \not\subseteq \{0\} = AncStreams[0] \cup \{0\}$, nor e is a valid ancestor of b, since $DestStreams[e] = \{1, 2\} \not\subseteq \{0\} = AncStreams[0] \cup \{0\}$. On the other hand, in stream 1, tensor i may reuse tensors c and d, since $StartTime[i] = 6 > 5 = EndTime[d]$, but it may not reuse f, since $StartTime[i] = 6 \not> 6 = EndTime[f]$.

Algorithm 5: Same Stream Reuse

Input : A set of streams S, tensors T, unsafe pairs U.
Output : A subset of U after removing safe pairs of tensors generated in the same stream.

1 **foreach** $s \in S$ **do**
2 **foreach** $\{t_1, t_2\} \in \binom{T_s}{2}$ **do**
3 **if** $StartTime[t_1] == StartTime[t_2]$ **then**
4 **continue;**
5 **if** t_1, t_2 *are between streams* **then**
6 **if** $ValidAncestor(t_1, t_2)$ *OR* $ValidAncestor(t_2, t_1)$ **then**
7 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
8 **else if** t_1 *is between streams* **then**
9 **if** $StartTime[t_1] > EndTime[t_2]$ *OR* $(StartTime[t_1] < StartTime[t_2]$ *AND* $ValidAncestor(t_1, t_2))$ **then**
10 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
11 **else if** t_2 *is between streams* **then**
12 **if** $StartTime[t_2] > EndTime[t_1]$ *OR* $(StartTime[t_2] < StartTime[t_1]$ *AND* $ValidAncestor(t_2, t_1))$ **then**
13 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
14 **else** // t_1, t_2 are *not* between streams
15 **if** $L_{t_1} \cap L_{t_2} = \emptyset$ **then**
16 $U \leftarrow U \setminus \{\{t_1, t_2\}\}$;
17 **return** U;

Theorem 2. *Algorithm 2 solves Multi-Stream Safety.*

4 Offset Assignment

In the preceding section, we saw how safety of memory reuse among tensors is decided by introducing the notion of safe pairs in the multi-stream model. We now examine the problem of assigning a start offset for each tensor in the logical buffer given a set of unsafe pairs and a set of contiguous constraints, thus capturing a general parallel deep learning scenario of static execution.

Parallel deep learning systems may have requirements to allocate certain tensor sequences in contiguous memory space (Zeng et al., 2021) in order to enable bulk I/O, e.g., reading/writing in one call for communication operators in multi-core execution. Contiguous allocation is similar to the *tensor fusion* problem (Sergeev & Del Balso, 2018) needed to avoid tiny all-reduce operations. Concatenation operators provided by frameworks have certain restrictions, e.g., they require matching tensor shapes (except in the concatenating dimension), and might not cover the general case we present. Indeed, there are test cases in MindSpore (Chen, 2021) where some tensors in a contiguous list do not make up safe pairs with the exact same set of tensors. Concatenating such a contiguous list into one tensor having the union of their reuse (no-overlap) constraints will yield a worse result than the one attained by our algorithms: in an example test, ResNet50 training peak memory is worsened by 5%.

Formally, for some natural number k, we say tensors $t_1, t_2, \ldots, t_k$ are bound by a *contiguous constraint*, if the start offset of t_i must be equal to the end offset of t_{i-1} plus one for $2 \leq i \leq k$. Multiple sets of tensors may be bound by contiguous constraints of arbitrary length. To avoid infeasible cases, we assume a tensor participates in at most one contiguous constraint. Below, for tensor $t \in T$, let $offset(t)$ denote its start offset in the logical buffer (to be decided) and $size(t)$ its size (given as input). Both values are non-negative integers. We let $I_t = [offset(t), offset(t) + size(t) - 1]$ denote the offset interval t occupies in the buffer. We now define the problem this section tackles.

Definition 5 (Offset Assignment for Parallel)**.** *Given a set of tensors T, a set of unsafe pairs $U \subseteq \binom{T}{2}$ and an arbitrary set of contiguous constraints $\{C_1, C_2, \ldots, C_l\}$, return a value $offset(t)$ for every $t \in T$ such that*

- *for all $\{t_1, t_2\} \in U$, it holds $I_{t_1} \cap I_{t_2} = \emptyset$,*
- *for any contiguous constraint C_i with $k_i \geq 2$ tensors $t_{i,1}, t_{i,2}, \ldots, t_{i,k_i}$, for all $j = 2, 3, \ldots, k_i$, it holds $offset(t_{i,j}) = offset(t_{i,j-1}) + size(t_{i,j-1})$, and*
- *$\max_{t \in T}(offset(t) + size(t))$ is minimized.*

In its essence, Offset Assignment for Parallel, shortly OAP, is a scheduling or allocation problem. Hardness can be inferred by its relationship to traditional problems in this setting. To demonstrate this, we use the classic NP-hard *Dynamic Storage Allocation* problem (DSA), see Problem SR2 in (Garey & Johnson, 1979) for reference.

Theorem 3. *Offset Assignment for Parallel is NP-hard.*

Let $G_c = (N(G_c), E(G_c))$ be the undirected *conflict graph* whose node set corresponds to the set of tensors, that is, $N(G_c) = T$, where there is an edge connecting any unsafe pair of tensors, that is, $E(G_c) = \{\{t_1, t_2\} \in \binom{T}{2} \mid \{t_1, t_2\}$ is unsafe $\}$. The conflict graph constructed in the proof of Theorem 3 is an *interval graph*. Considering our definition of streams, this would be the case when the input DNN is to be executed in a single stream. In that case, unsafe pairs correspond to lifetime intersections derived by the global topological sorting. At this point, we make no assumption on the topology of G_c for OAP and prepare general-use algorithms. Whether some stream definitions imply specific conflict graph topology is left for future work.

4.1 Algorithm Design

We implement two (families of) algorithms solving OAP. In *Single Object*, a single buffer, that is, *memory object*, is maintained like in state of the art Offset Calculation. In *Many Objects*, the logical buffer is partitioned into many objects. Their number is not known a priori, but decided on the fly by the algorithm. Each object is defined by the size of its largest, spanning, tensor. Each tensor is allocated such that its interval spans offsets within one object.

The intuition behind introducing Many Objects is that it runs faster than Single Object in practice, since searching for feasible gaps to place the next tensor is executed at each (small) object until a suitable one is found. For many tensors we need not examine the whole (large) memory space. Also, we are able to avoid many iterations in case a tensor forms an unsafe pair with the object-spanning tensor. Contrarily, Single Object comes with more freedom on scheduling, and, in some cases, this yields slightly better solutions.

Either algorithm takes as input a set of *blocks of tensors* and *unsafe pairs*. A block of tensors is either a single tensor or a sequence of tensors bound by a contiguous constraint. The algorithms are parameterized by *sorting strategy*, that is, the order in which tensors are considered by the algorithm, and *fitting strategy*, that is, the way to decide the start offset of a tensor among all available offsets. Algorithmic behavior is *deterministic* in the sense that ties in sorting and fitting are resolved based on unique order indices.

We employ essential routines in the algorithm descriptions to follow. $Sort$ returns the blocks given as input ordered in some way, e.g., by decreasing size. $Avoid$ takes as input a block to allocate, an object, and the set of unsafe pairs, and returns the set of offsets where we must not assign the new block to ensure safety. $Merge$ receives a list of offset intervals and returns a minimal stack of intervals, where no two intervals overlap, yet the intervals span the same offsets. Fit receives a block, the set of forbidden start offsets for it, and an object, and returns the start offset decided for this tensor (or none if an offset cannot be assigned in the Many Objects case). Fitting is either *Best-Fit* (place tensor at smallest-size gap possible) or *First-Fit* (place at first available gap possible). $SetOffsets$ takes a block and its assigned offset and assigns accordingly the offsets of tensors within the block. $offset$, respectively $size$, returns the start offset, respectively size, for a tensor, block or object. Objects and blocks are presented as linked lists of blocks and tensors, respectively. We use $head$ to denote the first element and $length$ for the number of elements in a list.

In Algorithm 6, we define *Single Object*. There is a single object, initially empty (line 2).We iterate over input blocks given by $Sort$ (lines 3 to 8). For each block, we determine the set $avoid\_intervals'$, where it cannot be placed due to unsafe pair constraints with a block already in object (lines 4 to 5). The block is assigned an offset according to fitting and is added to the object (lines 6 to 7). Within-block tensor offsets are set (line 8). For an example, see Figure 3.

Algorithm 6: The Single Object algorithm

Input : A set of *blocks* of tensors and *unsafe* pairs.
A sorting and fitting function $Sort$, Fit.
Output : A mapping from blocks to start memory offsets, and the size of total memory used.

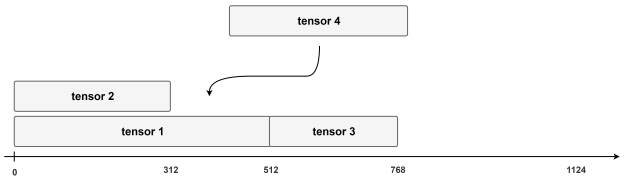
1 $offset \leftarrow \emptyset$;
2 $object \leftarrow [\,]$;
3 **foreach** $block \in Sort(blocks)$ **do**
4 $\quad avoid\_intervals \leftarrow Avoid(block, object, unsafe)$;
5 $\quad avoid\_intervals' \leftarrow Merge(avoid\_intervals)$;
6 $\quad offset(block) \leftarrow Fit(block, avoid\_intervals', object)$;
7 $\quad object.push\_back(block)$;
8 $\quad SetOffsets(block, offset(block))$;
9 **return** $(offset, \max_{b \in blocks}(offset(b) + size(b) - 1))$;

In Algorithm 7, we define *Many Objects*. In this case, we maintain a list of objects, initially empty (line 2), which partition the logical buffer. Like in Single Object, we iterate over blocks in some sorted sequence (lines 4 to 26). For each block, we examine the objects currently present in the object list (lines 6 to 18). As before, we extract the set of offset intervals, where it is prohibited to place this tensor (lines 11 to 12). Note this might not be necessary for all objects. If there is an unsafe pair constraint with the object-spanning block, then $avoid\_intervals$ contains the whole object. Since a fit is not possible, there is no need to examine any other block in it. We include this optimization in lines 7 to 10. If there is at least one available offset for the tensor in this object, we fit the tensor there according to the fitting strategy and complete the object iteration (lines 13 to 18). If there exists no object where the tensor can be placed, then a new object is created with this tensor defining its offset interval, and it is placed at the end of the object list (lines 19 to 26). For an example, see Figure 4.

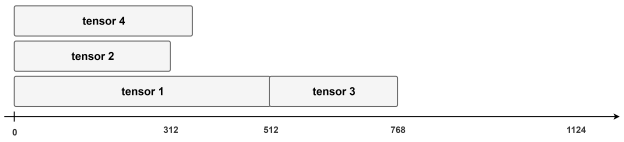
5 Empirical Evaluation

Setup. We implemented the proposed algorithms within MindSpore (Chen, 2021) backend and ran training tests on an Atlas 800 (model 9010) server featuring 8 Ascend 910 AI accelerators (Liang, 2020; Liao et al., 2021), 32 GB high bandwidth memory and 2.24 pflops fp16 computing power.

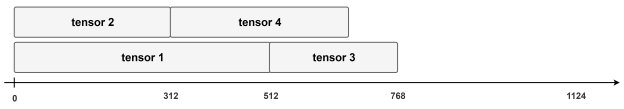
Workload. We experimented with ten large-size DNNs available in MindSpore Model Zoo (MindSpore, 2023) . We ran MindSpore training process on instances of BERT (Devlin et al., 2019) with roughly 4,400 (base), 8,200 (large), and 9,400 tensors (nezha), a Transformer (Vaswani et al., 2017) (3,500 tensors), and a Tiny-BERT (Jiao et al., 2019) (1,800). We also tested ResNet (He et al., 2016)



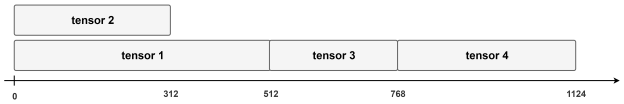
(a) Tensors 1, 2, 3 (sizes 512, 312, 256) are allocated at offsets 0, 0, 512, respectively. Tensor 4 (size 356) is next to be assigned.



(b) If $\{4, 1\}$ and $\{4, 2\}$ are not unsafe, then tensor 4 is assigned at offset 0 as well.



(c) If $\{4, 2\}$ is unsafe, but $\{4, 1\}$ and $\{4, 3\}$ are safe, then tensor 4 is assigned at offset 312.



(d) If either $\{4, 1\}$ or $\{4, 2\}$ is unsafe, and $\{4, 3\}$ is unsafe, tensor 4 is assigned at offset 768 and object end moves from 767 to 1123.

Figure 3: Example Single Object (First-Fit) allocation

Algorithm 7: The Many Objects algorithm

Input : A set of *blocks* of tensors and *unsafe* pairs.
A sorting and fitting function $Sort$, Fit.
Output : A mapping from blocks to start memory offsets, and the size of total memory used.

$offset \leftarrow \emptyset$;
$object\_list \leftarrow [\,]$;
$mem\_end \leftarrow -1$;
foreach *block* $b \in Sort(blocks)$ **do**
  $allocated \leftarrow False$;
  foreach $object \in object\_list$ **do**
    $b' \leftarrow object.head()$;
    if $length(b) == 1$ *AND* $length(b') == 1$ **then**
      if $\{b.head(), b'.head()\} \in unsafe$ **then**
        continue;
    $avoid\_intervals \leftarrow Avoid(b, object, unsafe)$;
    $avoid\_intervals' \leftarrow Merge(avoid\_intervals)$;
    $offset(b) \leftarrow Fit(b, avoid\_intervals', object)$;
    if $offset(b) \neq \emptyset$ **then**
      $allocated \leftarrow True$;
      $object.push\_back(b)$;
      $SetOffsets(b, offset(b))$;
      break;
  if $\boldsymbol{not}$ $allocated$ **then**
    $object \leftarrow \boldsymbol{new}$ $Object$;
    $offset(object) \leftarrow mem\_end + 1$;
    $offset(b) \leftarrow mem\_end + 1$;
    $object.push\_back(b)$;
    $SetOffsets(b, offset(b))$;
    $object\_list.push\_back(object)$;
    $mem\_end \leftarrow mem\_end + size(b)$;
return $(offset, \max_{b \in blocks} offset(b) + size(b) - 1)$;

(1,500), a ResNet-based Face Recognition net (8,500), a YOLOv3-based (Redmon & Farhadi, 2018) Face Detection net (4,200), PanGu-α with 2.6 billion parameters (Zeng et al., 2021) (20,000), and MobileNetv2-quantitative (Sandler et al., 2018) (3,900).

5.1 Multi-Stream Safety

For each DNN, we receive a multi-stream computational graph as produced by another MindSpore component, which runs before memory allocation to define multi-streaming. Roughly speaking, nodes are assigned to a few categories, for example, common, communication and other. A stream only contains nodes of a single category. Based on hardware constraints, there is an upper limit on the number of nodes to fit within a stream. The graph is traversed, per category, and a new stream is created when the current one fills up.

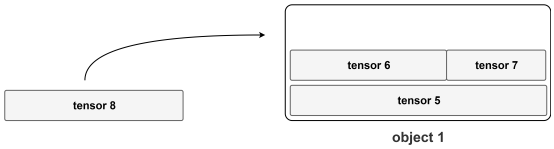
In Table 1, we demonstrate the performance, execution time, of the algorithms designed in Section 3. The ideas proposed in Algorithm 2 outperform the original idea proposed even in the optimized version of Algorithm 1. The solving time required for this component is significantly reduced for all DNNs tested by at least 21% and up to 44%.

5.2 Offset Assignment

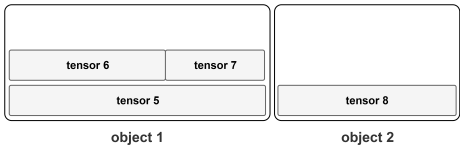
We implemented, then executed, a variety of Single Object instances based on ideas described in state-of-the-art. Note that pure state-of-the-art algorithms do not support general unsafe pairs and contiguous constraints. We adapt the ideas presented in previous works to run in our general parallel deep learning scenario. We incorporate these ideas within Single Object by performing necessary preprocessing and modifying the order in which tensors are considered. The results for Offset Assignment are given in Tables 2 and 3.

In Greedy Breadth (Pisarchyk & Lee, 2020), tensors are sorted in decreasing order of *breadth* of their source node, i.e., the sum of sizes of tensors whose global lifetimes include the execution order of the node. In Greedy Size (Pisarchyk & Lee, 2020), tensors are sorted by decreasing size. In Greedy Source (Lee et al., 2019), tensors are sorted by increasing execution order of their source node.

The heuristic in (Sekiyama et al., 2018) is not implementable (at least in a obvious way) in the multi-stream setting, since



(a) If tensor 8 forms an unsafe pair with at least one tensor, then it cannot be allocated in object 1 due to lack of safe space.



(b) Object 2 is created. Tensor 8 defines its interval of offsets.

Figure 4: Creation of new object by Many Objects

Table 1: Experimental results for Multi-Stream Safety on popular DNNs (solving time in milliseconds)

Network	Algorithm 1_{opt}	Algorithm 2	Speedup
BERT-base	957	620	35.21%
BERT-large	4043	2289	43.38%
BERT-nezha	5275	2959	43.91%
FaceRecognition	1376	845	38.59%
PanGu-α (2.6B)	13845	10359	25.18%
ResNet-50	32	20	37.50%
Tiny-BERT	143	96	32.87%
FaceDetection	693	546	21.21%
Transformer	720	568	21.11%
MobileNetv2	57	42	26.32%

it is based on maintaining a two-dimensional strip packing with lifetime and offset axes, and global lifetime can no longer capture the superset of unsafe pairs generated by multi-streaming. Instead, we tested Greedy Unsafe, with tensors sorted by decreasing number of unsafe pairs in which they participate. Intuitively, the decision for a tensor with many constraints should be made early, whereas few constraints imply a lot of freedom for offset assignment.

With regard to fitting, state-of-the-art methods fall under the paradigm of Best-Fit. (place tensor in smallest-size available gap). Besides that, we chose to experiment with traditional First-Fit, as an alternative (place tensor in first available gap). Other fitting methods we have tested did not yield improved results, therefore we omit them.

We experimented with the Many Objects algorithm with Greedy Size sorting and both fitting strategies. Note that all other sortings tested with Single Object were also tested with Many Objects, but they yielded significantly inferior memory footprint solutions, thus they are not of practical interest and we do not include them in our results. The intuition is that Many Objects is efficient when tensors are sorted in decreasing order of size, since this results into a sequence of footprints also of decreasing size. Due to tensors being ordered by size, each new object created will be at most as large as the one before it. This is not the case for other sortings, which yield a size-wise unbalanced object sequence, which does not perform well experimentally.

Up to our knowledge, there is no relevant literature in scheduling capturing contiguous constraints, so it is not possible to provide any baseline other than a theoretical lower bound and adapted Offset Calculation state of the art. All algorithms receive the same set of constraints as input. We provide memory footprint solutions in gigabytes and solving time in milliseconds. We compute the memory relative error of Many Objects compared to Single Object, and of the overall best solution compared to the global *lifetime lower bound* corresponding to a single stream execution of the DNN (the maximum total size of tensors alive at any time step of sequential execution). For solving time, we compute the improvement we get with Many Objects.

Many Objects provides nearly optimal solutions consistently deviating by less than 0.3% from the best Single Object. This translates to a loss of only a few megabytes compared to the state-of-the-art, thus no real practical overhead in networks of large size. In some cases, for example, Tiny-BERT and ResNet, Many Objects even discovers better solutions of up to 0.7%. Due to its design, Many Objects runs in practice in much less time than Single Object. Indeed, if we consider the solving time gains presented in the tables, these are roughly at least 30% and up to 70%. In addition, Many Objects Greedy Size does not require any preprocessing time to compute breadth or number of conflicts.

PanGu-α is a network designed to test the limitations of parallel deep learning. The graph includes 678 contiguous constraints, whereas all other graphs include roughly up to 20. In this extreme case, the time to handle contiguous constraints dominates the solving time (within the $Avoid()$ subroutine), so it is expected Many Objects and Single Object solving times converge. Even so, we still manage to get about 20% average solving time gain. Also, the rigorous handling of contiguous constraints in our designed algorithms is necessary to fit large models in memory (MindSpore, 2021), see details in Table 4. For 13 billion parameters, memory usage is decreased from 31.72 GB (causing an out-of-memory in device) to 25.08 GB.

Besides Many Objects, we observe how the introduction of First-Fit and Greedy Unsafe provides us with significant gains for certain neural networks. First-Fit finds the best solution for 9 out of 10 DNNs in Single Object and 8 out of 10 in Many Objects. Greedy Unsafe provides an improved solution over other Single Object algorithms for 2 DNNs.

Finally, in the context of overall training performance, there is evidence on our optimized memory management enabling

Table 2: Training experiments: peak memory in GB, solving time (milliseconds) in italic – First Part

	BERT-base	BERT-large	BERT-nezha	FaceRecognition	PanGu-α (2.6B)
Naïve Allocation	42.7816	83.3553	61.5739	77.6916	1349.1400
Single Object (SO)					
Greedy Breadth (Best-Fit)	**13.5119** *(597)*	**24.9171** *(3587)*	14.8293 *(4309)*	16.1210 *(3029)*	**18.4541** *(16318)*
Greedy Breadth (First-Fit)	**13.5119** *(603)*	**24.9171** *(3555)*	14.8293 *(4273)*	16.1210 *(2841)*	**18.4541** *(16320)*
Greedy Size (Best-Fit)	13.5121 *(637)*	25.0130 *(3716)*	14.8232 *(4520)*	15.8100 *(2409)*	**18.4541** *(16017)*
Greedy Size (First-Fit)	13.5121 *(616)*	25.0040 *(3790)*	**14.7778** *(4443)*	15.7867 *(2936)*	**18.4541** *(13451)*
Greedy Source (Best-Fit)	13.5148 *(612)*	24.9320 *(3565)*	14.9117 *(4013)*	15.8746 *(2927)*	19.5846 *(14928)*
Greedy Source (First-Fit)	13.5148 *(571)*	24.9320 *(3482)*	14.9117 *(4132)*	16.0055 *(2975)*	20.2817 *(15564)*
Greedy Unsafe (Best-Fit)	13.6894 *(584)*	25.1032 *(3499)*	14.8282 *(3559)*	**15.7456** *(3220)*	21.4649 *(15298)*
Greedy Unsafe (First-Fit)	13.6894 *(581)*	25.1016 *(3571)*	14.8426 *(4037)*	15.7815 *(3063)*	21.6111 *(15931)*
Many Objects (MO)					
Greedy Size (Best-Fit)	**13.5121** *(316)*	**24.9172** *(1953)*	**14.7854** *(2060)*	**15.7797** *(862)*	**18.4541** *(12781)*
Greedy Size (First-Fit)	**13.5121** *(315)*	**24.9172** *(2226)*	**14.7854** *(2310)*	**15.7797** *(875)*	**18.4541** *(12390)*
Lower Bound	13.5119	24.9171	14.6860	15.7456	18.4541
Memory Error					
Best MO to Best SO	0.00148%	0.00040%	0.05143%	0.21656%	0.00000%
Best to Lower Bound	0.00000%	0.00000%	0.62509%	0.00000%	0.00000%
Solving Time Gain					
Best MO to Best SO	44.83%	43.91%	48.67%	69.66%	7.89%
Avg. MO to Avg. SO	47.43%	41.89%	47.49%	70.31%	18.69%

MindSpore to train a model with larger batch size [1].

6 Related Work

There are online mentions about unsafe pair computation for parallel deep learning platforms, e.g., in MXNet's website. However, we are unaware of any general offline model determining all unsafe pairs in the context of multi-streaming.

For sequential offset assignment, MXNet (Chen et al., 2015) use two simple greedy static allocation algorithms: one simulating the graph traversal and one based on parallel execution dependencies. In (Sekiyama et al., 2018), the authors propose a Best-Fit approach on the basis of a heuristic for two-dimensional strip packing (Burke et al., 2004). The algorithm chooses an offset and determines the tensor to be allocated there. Experiments in Chainer showed significant memory consumption reductions. In (Lee et al., 2019), the authors propose heuristics to preplan a shared objects memory allocation for efficient use in GPU hardware. They propose a variation of greedy Best-Fit. Their algorithm is shown to provide good results for various DNNs. Also, they propose to reduce the given DAG to another (auxiliary) graph and run a minimum-cost flow algorithm on it, then translate to a memory plan in the original DAG. This approach performs better for a limited number of DNNs tested in Tensorflow Lite (Shuangfeng, 2020). Pisarchyk and Lee (Pisarchyk & Lee, 2020) propose new allocation algorithms in the single-stream case. The algorithms use Best-Fit with greedy tensor order, either by size or by operator breadth, Inference on six popular DNNs shows up to 11% benefit over past approaches. In (Fang et al., 2021), an idea related to Many Objects is proposed, as a component in a transformer serving system, to balance memory footprint and malloc/free efficiency. The motivation is to release small objects not used in a current inference. In contrast to us, they use default or scaled object sizes. They identify best-fit gaps in memory by a slight modification of the algorithm proposed in (Pisarchyk & Lee, 2020) and showcase their algorithm's performance during BERT inferences.

We mention few other lines of research relevant to DNN memory management. In *DNN compression* (Han et al., 2015) the goal is to prune arcs, in order to slightly modify network behavior and gain memory reduction. In *memory swapping* (Huang et al., 2020; Le et al., 2019) the idea is to swap tensors from fast small-capacity memory to slow high-capacity memory in order to maintain the fast memory's footprint under control. Recent works (Jain et al., 2020; Kirisame et al., 2021; Peng et al., 2020) consider *tensor recomputation*. Such approaches could be used as preprocessing to our module, as we may receive the recomputation-aware modified static training graph as input.

7 Conclusion

We implement a safe optimized memory allocation solver to enable parallel deep learning under the static execution model in an open-source production framework. We define

[1] "Support Safe Optimized Memory Allocation Solver on Ascend to improve the memory-reuse, the batch size of Bert large model (128 sequence length) is increased from 160 to 208." at https://github.com/mindspore-ai/mindspore/blob/master/RELEASE.md

Table 3: Training experiments: peak memory in GB, solving time (milliseconds) in italic – Second Part

	ResNet-50	Tiny-BERT	FaceDetection	Transformer	MobileNetv2
Naïve Allocation	3.3598	5.17475	13.5162	34.2267	64.1832
Single Object (SO)					
Greedy Breadth (Best-Fit)	1.4909 *(41)*	0.71281 *(105)*	3.21740 *(763)*	**7.54506** *(302)*	**17.6506** *(544)*
Greedy Breadth (First-Fit)	1.4909 *(73)*	0.75969 *(99)*	**3.19949** *(773)*	**7.54506** *(362)*	**17.6506** *(551)*
Greedy Size (Best-Fit)	1.4211 *(49)*	0.70469 *(100)*	3.20868 *(841)*	**7.54506** *(323)*	17.6682 *(592)*
Greedy Size (First-Fit)	**1.4133** *(49)*	0.70708 *(97)*	3.20813 *(869)*	**7.54506** *(322)*	17.6637 *(625)*
Greedy Source (Best-Fit)	1.4917 *(43)*	0.74554 *(107)*	3.20907 *(791)*	7.78331 *(301)*	17.6881 *(595)*
Greedy Source (First-Fit)	1.4917 *(42)*	0.74581 *(106)*	3.21292 *(773)*	7.78526 *(300)*	17.6887 *(594)*
Greedy Unsafe (Best-Fit)	1.4207 *(45)*	0.70400 *(97)*	3.19961 *(805)*	7.57963 *(314)*	17.6613 *(692)*
Greedy Unsafe (First-Fit)	1.4286 *(46)*	**0.70180** *(100)*	3.19961 *(851)*	7.57963 *(311)*	17.6613 *(695)*
Many Objects (MO)					
Greedy Size (Best-Fit)	1.4251 *(29)*	**0.69726** *(43)*	**3.20942** *(406)*	**7.54506** *(169)*	**17.6662** *(162)*
Greedy Size (First-Fit)	**1.4132** *(28)*	0.69698 *(44)*	3.21070 *(441)*	**7.54506** *(136)*	**17.6662** *(154)*
Lower Bound	1.4056	0.68938	3.19949	7.54506	17.6423
Memory Error					
Best MO to Best SO	**-0.00708%**	**-0.64690%**	0.31036%	0.00000%	0.08838%
Best to Lower Bound	0.54069%	1.14306%	0.00000%	0.00000%	0.04705%
Solving Time Gain					
Best MO to Best SO	31.71%	55.67%	46.79%	54.67%	71.69%
Avg. MO to Avg. SO	41.24%	57.09%	47.60%	51.87%	74.14%

Table 4: Training experiment: PanGu-α large model

	PanGu-α (8B)	PanGu-α (13B)
Baseline (without our solution)	27.36	31.72
Our Best Result	14.76	25.08
Lower Bound	14.68	24.95
Memory Error		
Our result to Baseline	-46.05%	-20.92%
Our result to Lower Bound	0.54%	0.55%

a general problem also capturing contiguous constraints. We design efficient algorithms to decide provably safety reuse based on the stream graph: we show up to 44% time gains compared to an optimized conventional approach. We design offset assignment algorithms to optimize memory consumption for large DNNs: Many Objects yields time gains up to 70% compared to adapted state of the art.

Future directions include improving compilation time and solution quality, but also relevant static execution aspects. The choice of multi-streaming and the defined global/local topological ordering, affects the global memory allocation assignment and it is a reasonable future question to consider.

REFERENCES

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

Burke, E. K., Kendall, G., and Whitwell, G. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research*, 52(4):655–671, 2004.

Chen, L. *Deep Learning and Practice with MindSpore*. Springer Nature, 2021.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, Y., Xie, Y., Song, L., Chen, F., and Tang, T. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020. ISSN 2095-8099. doi: https://doi.org/10.1016/j.eng.2020.01.007. URL https://www.sciencedirect.com/science/article/pii/S2095809919306356.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Escoffier, B., Monnot, J., and Paschos, V. T. Weighted coloring: further complexity and approximability results. *Information Processing Letters*, 97(3):98–103, 2006.

Even, G., Halldórsson, M. M., Kaplan, L., and Ron, D. Scheduling with conflicts: online and offline algorithms. *Journal of scheduling*, 12(2):199–224, 2009.

Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

Garey, M. R. and Johnson, D. S. *Computers and intractability*, volume 174. Freeman San Francisco, 1979.

Google. Tcmalloc, 2022. URL https://github.com/google/tcmalloc.

Hà, M. H., Ta, D. Q., and Nguyen, T. T. Exact algorithms for scheduling problems on parallel identical machines with conflict jobs. *arXiv preprint arXiv:2102.06043*, 2021.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1341–1355, 2020.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 497–511, 2020.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021.

Kowalczyk, D. and Leus, R. An exact algorithm for parallel machine scheduling with conflicts. *Journal of Scheduling*, 20(4):355–372, 2017.

Le, T. D., Imai, H., Negishi, Y., and Kawachiya, K. Automatic gpu memory management for large neural models in tensorflow. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pp. 1–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367226. doi: 10.1145/3315573.3329984. URL https://doi.org/10.1145/3315573.3329984.

Lee, J., Chirkov, N., Ignasheva, E., Pisarchyk, Y., Shieh, M., Riccardi, F., Sarokin, R., Kulik, A., and Grundmann, M.

On-device neural net inference with mobile gpus. *arXiv preprint arXiv:1907.01989*, 2019.

Liang, X. Chapter 3 - hardware architecture. In Liang, X. (ed.), *Ascend AI Processor Architecture and Programming*, pp. 75–100. Elsevier, 2020. ISBN 978-0-12-823488-4. doi: https://doi.org/10.1016/B978-0-12-823488-4.00003-5. URL https://www.sciencedirect.com/science/article/pii/B9780128234884000035.

Liao, H., Tu, J., Xia, J., Liu, H., Zhou, X., Yuan, H., and Hu, Y. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 789–801, 2021. doi: 10.1109/HPCA51647.2021.00071.

Lin, L. and Lin, Y. Machine scheduling with contiguous processing constraints. *Information Processing Letters*, 113(8):280–284, 2013. ISSN 0020-0190. doi: https://doi.org/10.1016/j.ipl.2013.02.004. URL https://www.sciencedirect.com/science/article/pii/S0020019013000495.

Mallek, A. and Boudhar, M. A branch-and-bound algorithm for the problem of scheduling with a conflict graph. In *2020 International Conference on Decision Aid Sciences and Application (DASA)*, pp. 778–782. IEEE, 2020.

MindSpore. Official news - 07, 2021. URL https://mindspore.cn/news/newschildren?id=367.

MindSpore. Models, 2023. URL https://gitee.com/mindspore/models.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.

Pemmaraju, S. V., Raman, R., and Varadarajan, K. R. Buffer minimization using max-coloring. In *SODA*, volume 4, pp. 562–571. Citeseer, 2004.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 891–905, 2020.

Pisarchyk, Y. and Lee, J. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288*, 2020.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Redmon, J. and Farhadi, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Sekiyama, T., Imamichi, T., Imai, H., and Raymond, R. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Sethi, R. Complete register allocation problems. *SIAM journal on Computing*, 4(3):226–248, 1975.

Shuangfeng, L. Tensorflow lite: On-device machine learning framework. *Journal of Computer Research and Development*, 57(9):1839–1853, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Zeng, W., Ren, X., Su, T., Wang, H., Liao, Y., Wang, Z., Jiang, X., Yang, Z., Wang, K., Zhang, X., Li, C., Gong, Z., Yao, Y., Huang, X., Wang, J., Yu, J., Guo, Q., Yu, Y., Zhang, Y., Wang, J., Tao, H., Yan, D., Yi, Z., Peng, F., Jiang, F., Zhang, H., Deng, L., Zhang, Y., Lin, Z., Zhang, C., Zhang, S., Guo, M., Gu, S., Fan, G., Wang, Y., Jin, X., Liu, Q., and Tian, Y. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *arXiv preprint arXiv:2104.12369*, 2021.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Zuckerman, D. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pp. 681–690, 2006.

A SECTION 3: MISSING PROOFS

Proof of Theorem 1. We prove $\{t_1, t_2\} \in \binom{T}{2}$ is a safe pair if and only if either $t_1 \in AncTensors[t_2]$ or $t_2 \in AncTensors[t_1]$. Below, let $n_1, n_2 \in N$ denote the nodes producing t_1, t_2, respectively.

Without loss of generality, assume $t_1 \in AncTensors[t_2]$. By definition it holds $DestNodes[t_1] \subseteq AncNodes[n_2]$. There exists a directed path of length at least 1 from any node consuming t_1 to the node producing t_2. That is, regardless of multi-streaming, t_1 is needed in storage only strictly before t_2 is produced, so they form a safe pair. The same argument works for the symmetrical case $t_2 \in AncTensors[t_1]$. Note it cannot be the case both $t_1 \in AncTensors[t_2]$ and $t_2 \in AncTensors[t_1]$, since it implies there exist $n' \in DestNodes[t_1]$ and $n'' \in DestNodes[t_2]$, such that $n_1 \rightsquigarrow n' \rightsquigarrow n_2 \rightsquigarrow n'' \rightsquigarrow n_1$, a contradiction since the graph is acyclic.

If instead $t_1 \notin AncTensors[t_2]$ and $t_2 \notin AncTensors[t_1]$, then by definition it holds $DestNodes[t_1] \not\subseteq AncNodes[n_2]$ and $DestNodes[t_2] \not\subseteq AncNodes[n_1]$. By definition of $AncNodes$, there exists a node $n' \in DestNodes[t_1]$, such that $n' \not\rightsquigarrow n_2$, and a node $n'' \in DestNodes[t_2]$, such that $n'' \not\rightsquigarrow n_1$. We consider the possible cases:

- If $n' = n_2$ or $n'' = n_1$, a consumer of one tensor is the producer of the other, so $\{t_1, t_2\}$ is an unsafe pair.
- If $n_2 \not\rightsquigarrow n'$ or $n_1 \not\rightsquigarrow n''$, then without loss of generality let us consider the case $n_2 \not\rightsquigarrow n'$. Since it holds both $n' \not\rightsquigarrow n_2$ and $n_2 \not\rightsquigarrow n'$ (and $n' \neq n_2$), it follows there is no dependency between n' and n_2 and they might execute in parallel. Thus, $\{t_1, t_2\}$ is unsafe.
- If $n_2 \rightsquigarrow n'$ and $n_1 \rightsquigarrow n''$, we consider two subcases.

  If $n_1 \rightsquigarrow n_2$, then t_1 is produced before t_2 is produced. But, since $n_2 \rightsquigarrow n'$ and $n' \in DestNodes[t_1]$, t_1 must remain stored until the execution of n', which is after the execution of n_2 producing t_2. As a result, t_1 and t_2 must be saved in storage at the same time, which makes them an unsafe pair. The case $n_2 \rightsquigarrow n_1$ is symmetrical by considering $n_1 \rightsquigarrow n''$ and $n'' \in DestNodes[t_2]$.

  If $n_1 \not\rightsquigarrow n_2$ and $n_2 \not\rightsquigarrow n_1$, the nodes are independent and might execute in parallel, so $\{t_1, t_2\}$ is unsafe.

□

Proof of Lemma 1. In case it holds $DestStreams[t_2] \not\subseteq AncStreams[s_1] \cup \{s_1\}$, the call returns False (Algorithm 4, lines 1 to 2). In this case, there is a stream s^* receiving t_2 such that it holds $s^* \notin AncStreams[s_1] \cup \{s_1\}$. Let $n^* \in DestNodes[t_2]$ stand for a destination node of t_2 within s^*. By definition of $AncStreams$, it follows $n^* \not\rightsquigarrow n_1$.

If instead $DestStreams[t_2] \subseteq AncStreams[s_1] \cup \{s_1\}$, to return True, we consider all streams $s \in DestStreams[t_2]$ (Algorithm 4, lines 3 to 9).

Assume $s \in AncStreams[s_1]$ (Algorithm 4, lines 4 to 6). Let $EndTime[t_2][s] = n_3$ and $MaxAncId[t_1][s] = n_4$. We get $n_4 \rightsquigarrow n_1$, since n_4 is an ancestor of n_1 and $s \neq s_1$. For any $n' \in DestNodes[t_2] \cap N_s$, it holds $n' \rightsquigarrow n_3$ by within-stream arcs. The algorithm does not return False if and only if $n_3 \rightsquigarrow n_4$ (case $EndTime[t_2][s] < MaxAncId[t_1][s]$) or $n_3 = n_4$ (case $EndTime[t_2][s] = MaxAncId[t_1][s]$). Either case implies $n' \rightsquigarrow n$.

In the case $s = s_1$, let $EndTime[t_2][s_1] = n^*$. We do not return False if and only if $n^* \rightsquigarrow n_1$, equivalent to $EndTime[t_2][s_1] < StartTime[t_1]$, which implies $n' \rightsquigarrow n_1$ for any $n' \in DestNodes[t_2] \cap N_s$, since $n' \rightsquigarrow n^*$ by within-stream arcs (Algorithm 4, lines 7 to 9). □

Proof of Theorem 2. We consider all possible cases for any $\{t_1, t_2\} \in \binom{T}{2}$. Below, let t_1 be produced by node $n_1 \in N_{s_1}$ and t_2 by $n_2 \in N_{s_2}$. Recall in Theorem 1 we have proved $\{t_1, t_2\}$ is safe if and only if $t_1 \in AncTensors[t_2]$ or $t_2 \in AncTensors[t_1]$. Also, by definition, $t_1 \in AncTensors[t_2]$ is equivalent to $DestNodes[t_1] \subseteq AncNodes[n_2]$, which is equivalent to $n^* \rightsquigarrow n_2$ for all $n^* \in DestNodes[t_1]$.

- In case it holds $s_1 \neq s_2$, $s_1 \notin AncStreams[s_2]$ and $s_2 \notin AncStreams[s_1]$, by definition it follows neither $t_1 \in AncTensors[t_2]$ nor $t_2 \in AncTensors[t_1]$, thus $\{t_1, t_2\}$ is an unsafe pair. In Algorithms 3, 5, the pair is not considered in any loop, thus eventually it holds $\{t_1, t_2\} \in U''$ (Algorithm 2, line 4).
- Assume $s_1 \neq s_2$ and, without loss of generality, the case $s_2 \in AncStreams[s_1]$. The tensor pair $\{t_1, t_2\}$ is considered in Algorithm 3.

  If t_2 is not between streams, let $n' = EndTime[t_2]$ and $n'' = MaxAncId[t_1][s_2]$. By within-stream arcs, for any $n^* \in DestNodes[t_2]$, where $n^* \neq n'$, we get $n^* \rightsquigarrow n'$. By assumption, it holds $n'' \rightsquigarrow n_1$. By the algorithm, the pair is deemed safe if and only if it holds $n' \rightsquigarrow n''$ or $n' = n''$ (Algorithm 3, line 6). Either way, we infer $n^* \rightsquigarrow n_1$ for all $n^* \in DestNodes[t_2]$.

  If t_2 is between streams, $ValidAncestor(t_2, t_1)$ is called (Algorithm 3, line 10). By Lemma 1, $\{t_1, t_2\}$ is deemed safe if and only if it holds $n^* \rightsquigarrow n_1$ for all $n^* \in DestNodes[t_2]$.
- Assume $s_1 = s_2 = s$. If t_1, t_2 are both between streams, by Lemma 1, they are deemed safe if and only if there is a valid ancestor relationship in either way (Algorithm 5, lines 5 to 7). That is, the algorithm decides safety if and only if it holds either $n^* \rightsquigarrow n_1$ for all $n^* \in DestNodes[t_2]$ or $n^{**} \rightsquigarrow n_2$ for all $n^{**} \in$

$DestNodes[t_1]$. As a side note, observe in this case $ValidAncestor(t_1, t_2)$ or $ValidAncestor(t_2, t_1)$ may only be true when the stream graph is not a DAG. This case could emerge depending on stream definitions. $ValidAncestor(t_2, t_1)$ cannot return true, unless it holds $DestStreams[t_2] \setminus \{s\} \subseteq AncStreams[s]$. As t_2 is between streams, it holds $DestStreams[t_2] \setminus \{s\} \neq \emptyset$. Since t_2 is generated within s, t_2 is sent from s to $s' \subseteq AncStreams[s]$, which means $s \in AncStreams[s']$, so there is a cycle in the stream graph.

Consider the case only one of t_1, t_2 is between streams (Algorithm 5, lines 8 to 13), say t_2 without loss of generality. $\{t_1, t_2\}$ is deemed safe if t_2 is generated *strictly* after t_1 is last consumed within s by some node $n' = EndTime[t_1]$, that is, if $n_1 \rightsquigarrow n^* \rightsquigarrow n' \rightsquigarrow n_2$ for any $n^* \in DestNodes[t_1]$. If $StartTime[t_1] \leq StartTime[t_2] \leq EndTime[t_1]$, then $n_2 \rightsquigarrow n'$ or $n_2 = n'$ and therefore $n' \not\rightsquigarrow n_2$, so $\{t_1, t_2\}$ is correctly deemed unsafe. If $StartTime[t_1] > StartTime[t_2]$, similarly to before, by Lemma 1, $\{t_1, t_2\}$ is safe if and only if $ValidAncestor(t_2, t_1)$ is True.

If t_1 and t_2 are not between streams, then all nodes consuming them are within s. Let $n' = EndTime[t_1]$ and $n^* \in DestNodes[t_1]$, where $n^* \neq n'$ if such an n^* exists. Similarly, let $n'' = EndTime[t_2]$ and $n^{**} \in DestNodes[t_2]$, where $n^{**} \neq n''$ if such an n^{**} exists. In Algorithm 5, lines 14 to 16, the pair is deemed safe if and only if $[n_1, n'] \cap [n_2, n''] = \emptyset$. Equivalently, by within-stream arcs, we get either $n_1 \rightsquigarrow n^* \rightsquigarrow n' \rightsquigarrow n_2 \rightsquigarrow n''$ or $n_2 \rightsquigarrow n^{**} \rightsquigarrow n'' \rightsquigarrow n_1 \rightsquigarrow n'$.

□

B Section 3: Missing Discussion

Multi-Stream Safety is polynomial-time solvable by using either of Algorithms 1, 2. This is an *offline* problem. For each pair of tensors, we decide a priori whether or not they are a safe pair. Given $|T|$ tensors, there are $\binom{|T|}{2} \subseteq \mathcal{O}(|T|^2)$ pairs to consider. We allocate one bit for each such decision. All such information is required for some offset assignment methods, see Section 4. This approach differs to other, online, scenarios where memory reuse is decided on the fly.

In Algorithm 1, also in 1_{opt}, we need to compute the set $AncNodes[n]$ for each node $n \in N$, in order to perform our ancestor tensor tests. We do so by performing a top to bottom graph traversal. Let $parents(v)$ denote the set of nodes which produce a tensor arriving at v. Then, for every node $n \in N$, it holds $n \in AncNodes[v]$ if and only if either $n \in parents(v)$ or $\bigvee_{n' \in parents(v)} (n \in AncNodes[n'])$ is true. We use a bitset-based implementation to perform the logical operations during the graph traversal efficiently.

In Algorithm 2, we perform more specific checks by considering the stream graph. We simplify our comparisons by the use of precomputed values $StartTime, EndTime$. There is no need to maintain the ancestor nodes of each node, since we use $MaxAncId$ instead. Also, tensors neither with the same source stream nor with source streams in ancestor relationship are not handled by Algorithms 3, 5; they are an unsafe pair by default. Thus, we get a (possibly significant) reduced number of iterations compared to the Algorithm 1. All comparisons require local topological sortings, not a global one, since we compare nodes within the same stream. Preprocessing is done in linear time by graph traversal.

C Section 4: Missing Proof

Definition 6 (Dynamic Storage Allocation)**.** *Given a set of items A to be stored, each $a \in A$ having a size $s(a) \in \mathbb{N}$, an arrival time $r(a) \in \mathbb{N}$ and a departure time $d(a) \in \mathbb{N}$, where $d(a) > r(a)$, find an allocation $\sigma : A \to \mathbb{N}$ such that the maximum right endpoint among allocated storage intervals $I(a) = [\sigma(a), \sigma(a) + s(a))$ is minimized and for every $a, a' \in A$ if it holds $I(a) \cap I(a') \neq \emptyset$, then either $d(a) \leq r(a')$ or $d(a') \leq r(a)$.*

Proof of Theorem 3. Given an instance of DSA, that is, an item set A and values $s(a), r(a), d(a)$ for each $a \in A$, we construct an instance of OAP with the items being tensors, where any two tensors a, a' form an unsafe pair if $[r(a), d(a)) \cap [r(a'), d(a')) \neq \emptyset$. The set of contiguous constraints is left empty. By construction, an allocation function for DSA uses storage D if and only if the same allocation function uses storage D in OAP. □

Let us discuss how Offset Assignment for Parallel relates to traditional coloring and scheduling problems, if we assume the set of contiguous constraints is empty. In case all sizes are equal, OAP reduces to *Graph Coloring*, that is, minimizing the number of colors assigned to vertices such that no two neighboring vertices are assigned the same color. This reduction implies hardness of approximation within $n^{1-\epsilon}$ ratio with $\epsilon > 0$ (Zuckerman, 2006). Also, we may reduce OAP to *Scheduling with Conflicts* (Even et al., 2009) for an unbounded number of machines: given a set J of n jobs with processing times $\{p_j\}_{j \in J}$ and a conflict graph $G_c = (J, E)$ over the jobs, assign time intervals on the machines such that *(i)* each job $j \in J$ is assigned an interval of length p_j on a single machine, *(ii)* intervals on the same machine do not overlap, *(iii)* for any $(j, j') \in E$, the intervals assigned to j and j' do not overlap, and *(iv)* the largest endpoint of an assigned time interval (makespan) is minimized. With regard to OAP, a tensor corresponds to a job and its size to a processing time. The number of machines is unbounded, since there is no restriction on the number of tensors whose assigned offset intervals may over-

lap. To visualize the connection, consider OAP items to be rectangles of width 1 and length equal to their size. A solution to OAP stacks rectangles on top of each other when sharing offsets. A horizontal strip of width 1 is matched to a distinct machine for the Scheduling with Conflicts definition. In Figure 5, we visualize the connection of OAP to Scheduling with Conflicts.

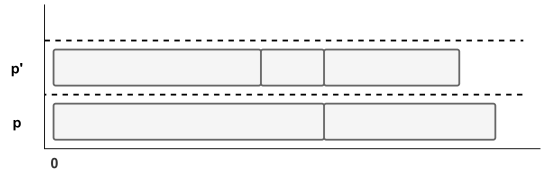


Figure 5: From OAP to Scheduling with Conflicts: The bottom two jobs are assigned to machine p, while the top three jobs are assigned to p'.

D Section 4: Missing Subroutines

Avoid. In Algorithm 8, we present $Avoid$, the subroutine responsible for generating the set of offset intervals, which cannot be selected as a start offset for a new block of tensors.

If the block to be allocated contains a single tensor (lines 2 to 9), then it suffices to ensure safety between this new tensor and all tensors already allocated. For every block in object, we consider all its tensors sequentially (lines 6 to 9). If there is an unsafe pair with the new tensor, the interval occupied by the allocated tensor is pushed back in the list.

On the other hand, in case the new block forms a contiguous list and contains multiple tensors, the check we need becomes more complex. Again, we iterate over all blocks already in object (lines 11 to 27). We loop sequentially over all tensors within an allocated block (lines 13 to 27). For each allocated tensor, we loop over all tensors within the new block in order to compute forbidden offset intervals for the new block's first tensor (lines 16 to 26). Variable acc accumulates the total size of tensors in the new block that have already been examined in this loop. If the allocated tensor and the new tensor are an unsafe pair, we need to determine the proper offset interval, which must be forbidden for placement of the new block's first tensor. To do so, we use auxiliary variables $start$ and end (lines 18 to 19). If new_tensor and $tensor$ are an unsafe pair, then new_tensor cannot overlap with interval $[offset(tensor) - size(new\_tensor) + 1, offset(tensor) + size(tensor))$ in order for no part of new_tensor to occupy the same space as $tensor$. Since new_tensor is a member of a contiguous block, the corresponding offset interval we forbid for new_block is displaced by an acc distance to the left. Note we use open right endpoint for the intervals we push to match the functionality of our Fit subroutine to be detailed later. The only thing left to make sure is that the forbidden interval is within the limits of the current object we examine. If (part of) the interval is outside the object, we need not declare it forbidden as such space will not be examined anyway for the current object. Note that $end \leq offset(tensor) + size(tensor) \leq offset(object) + size(object)$, since an allocated tensor is always within object limits. It suffices to check how $start$ and end compare with respect to the leftmost limit of the object, namely $offset(object)$. If $start \geq offset(object)$, then the whole interval is within the object and so we push it in the avoid list (lines 20 to 21). Otherwise, if it partially intersects with the object, when $end > offset(object)$, then we push back the corresponding part (lines 22 to 24).

For an example of the above, see Figure 6. Block ABC is already allocated in the object with tensors A, B, C, at offsets 0, 10, 15, respectively. Tensor sizes are 10, 5, and 3,

Algorithm 8: Avoid

Input : The block of tensors new_block to be allocated, the $object$, and the $unsafe$ pairs.
Output : A list of offset intervals forbidden for the head tensor of new_block due to unsafe pairs.

```
1  avoid_intervals ← [];
2  if length(new_block) == 1 then
3      new_tensor ← new_block.head();
4      foreach block ∈ object do
5          tensor ← block.head();
6          while tensor ≠ NULL do
7              if {new_tensor, tensor} ∈ unsafe then
8                  avoid_intervals.push_back(
                   [offset(tensor),
                   offset(tensor) + size(tensor)));
9              tensor ← tensor.next();
10 else // length(new_block) > 1
11     foreach block ∈ object do
12         tensor ← block.head();
13         while tensor ≠ NULL do
14             acc ← 0;
15             new_tensor ← new_block.head();
16             while new_tensor ≠ NULL do
17                 if {new_tensor, tensor} ∈ unsafe then
18                     start ← offset(tensor)
                         − size(new_tensor) + 1 − acc;
19                     end ← offset(tensor)
                        + size(tensor) − acc;
20                     if start ≥ offset(object) then
21                         avoid_intervals.push_back(
                           [start, end));
22                     else
23                         if end > offset(object) then
24                             avoid_intervals.push_back(
                               [offset(object), end));
25                 acc ← acc + size(new_tensor);
26                 new_tensor ← new_tensor.next();
27             tensor ← tensor.next();
28 return avoid_intervals;
```

respectively. A new block DEF is considered by $Avoid$: D of size 8, followed by E of size 6, followed by F of size 6. Assume $\{E, B\}$ is unsafe. This means E cannot be placed on offset interval $[offset(B) - size(E) + 1, offset(B) + size(B)) = [10 - 6 + 1, 10 + 5) = [5, 15)$, since it will overlap with B. For the head tensor D, we displace to the left by 8 (the size of D is the accumulator value at this point). Thus, the forbidden interval becomes $[-3, 7)$, which finally becomes $[0, 7)$ to stay within object limits.

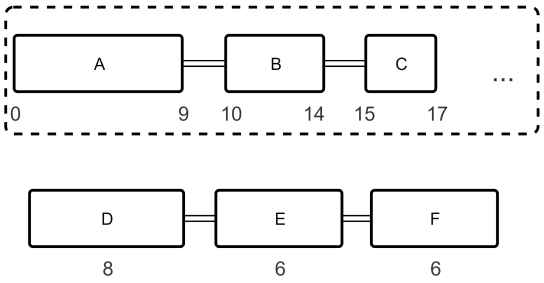


Figure 6: $Avoid$ in case of many-tensors new_block.

Merge. Subroutine $Merge$, see Algorithm 9, is essentially a preprocessing step for Fit to avoid iterating over the same (parts of) intervals multiple times. It takes as input the interval list generated by $Avoid$ and outputs a stack of merged intervals, sorted by decreasing order of right endpoint.

Initially, the interval list $avoid\_intervals$ is sorted by increasing left endpoint (line 2). Note that this step is necessary due to the functionality of $Avoid$ discussed above for the case of new block with multiple tensors. The first interval is pushed onto the stack (line 5). Then we iterate over all remaining intervals in the list in the sorted order (lines 6 to 14). We compare the right endpoint of the interval at the top of the stack with the left endpoint of the new interval. If the latter is larger, then there is no overlap, hence we simply push the new interval at the top of the stack (lines 9 to 10). Otherwise, there is an overlap, and we update the right endpoint of the interval at the top of the stack, if the new interval partially intersects it to the right and is not fully contained in it (lines 11 to 13). Note the sorted order in which intervals are considered guarantees correctness as we scan offsets from left to right.

Fit. We provide details about the implementation of Fit called in our algorithms, that is, in Algorithm 6, line 6, for Single Object, and in Algorithm 7, line 13, for Many Objects. In Algorithm 10, we give the pseudocode. Given a block b to be assigned into an object and the set of forbidden offsets for the head tensor of b, we choose an offset for b, according to a fitter policy, such that safety is respected.

Recall that the offset intervals to be avoided, namely $avoid\_intervals$, is returned by $Merge$ called just before.

Algorithm 9: Merge

Input : A list of offset intervals $avoid\_intervals$.
Output : A minimal size stack of offset intervals in decreasing order of right endpoint, spanning the same offsets as $avoid\_intervals$.

```
1  if length(avoid_intervals) == 0 then
2  |  return [];
3  Sort avoid by order of increasing left endpoint;
4  interval ← avoid_intervals.head();
5  stack.push(interval);
6  interval ← interval.next();
7  while interval ≠ NULL do
8  |  top ← stack.top();
9  |  if top.right < interval.left then
10 |  |  stack.push(interval);
11 |  else
12 |  |  if top.right < interval.right then
13 |  |  |  top.right ← interval.right;
14 |  interval ← interval.next();
15 return stack;
```

$Merge(\cdot)$ returns a stack of intervals in decreasing order of their right endpoint. Due to merging there is no overlap among any intervals within $avoid\_intervals$. The top of the stack contains the maximum-right-endpoint interval among the ones to be avoided. Before we proceed, note we introduce variable $compare\_size$ (lines 2 to 5), since the forbidden intervals refer to the first tensor in a block.

During fitting, we iterate $avoid\_intervals$ to find feasible *gaps* between consecutive forbidden intervals, where the block may be assigned (Algorithm 10, lines 7 to 14). The *upper* limit of a gap is the left endpoint of the interval to the right, whereas the *lower* limit is the right endpoint of the next-in-stack interval to the left. The initial upper limit is the end of object (line 6), which is the system infinity value for Single Object. If the tensor fits between lower and upper limits (line 10), and the block fits within the object (line 11) if it were to be placed there (a check necessary for Many Objects footprints), then this a candidate to place the block, and we maintain in $offset\_candidates$ the start offset of this gap and its size (line 13). The final lower limit is the beginning of the object (line 15). We repeat the same check (lines 16 to 19) for this last offset candidate. Finally, if non-empty, offset candidates are sorted according to a fitter call (line 22). The first start offset given by the sorting is selected as the start offset of block b (lines 23 to 24).

The fitters we test in the scope of this paper are *First-Fit* (Algorithm 11), and the most commonly used *Best-Fit* (Algorithm 12). In First-Fit, we choose the candidate with the lowest start offset, whereas in Best-Fit we choose the one with smallest gap size. Determinism is ensured by maintaining unique gap indices and using it as third sorting criterion.

One might correctly identify that the generalized Fit scheme

Algorithm 10: Fit

Input : A block of tensors b. A stack of non-overlapping offset intervals, in decreasing order of right endpoint, with which b's first tensor must not overlap ($avoid\_intervals$). An *object* and a *fitter*.
Output : A value for $offset(b)$.

$offset\_candidates \leftarrow \emptyset$;
if $length(b) == 1$ **then**
  $compare\_size \leftarrow size(b)$;
else
  $compare\_size \leftarrow size(b.head())$;
$upper \leftarrow object.right$;
while $avoid\_intervals$ *is* ***not*** *empty* **do**
  $interval \leftarrow avoid\_intervals.pop()$;
  $lower \leftarrow interval.right$;
  if $lower + compare\_size \leq upper$ **then**
    if $lower + size(b) \leq object.right$ **then**
      $gap \leftarrow upper - lower$;
      $offset\_candidates.insert((lower, gap))$;
  $upper \leftarrow interval.left$;
$lower \leftarrow offset(object)$;
if $lower + compare\_size \leq upper$ **then**
  if $lower + size(b) \leq object.right$ **then**
    $gap \leftarrow upper - lower$;
    $offset\_candidates.insert((lower, gap))$;
if $offset\_candidates == \emptyset$ **then**
  return $\emptyset$;
$offset\_candidates \leftarrow Sort(offset\_candidates, fitter)$;
$(lower', gap') \leftarrow offset\_candidates[0]$;
return $lower'$;

Algorithm 11: First-Fit

Input : Two start offset and gap pairs, namely $(lower', gap')$ and $(lower'', gap'')$.
Output : True if $(lower', gap')$ is sorted before $(lower'', gap'')$ in First-Fit, otherwise False.

return $lower' < lower''$ OR
  $(lower' = lower''$ AND $gap' < gap'')$;

we implement in Algorithm 10 does a lot of extra work in the case of First-Fit. Indeed, in this case, it suffices to order $avoid\_intervals$ by increasing left endpoint, and break the loop once the first feasible offset candidate is identified, thus saving on the number of iterations and the need for sorting. Regardless, we choose to stick with the generalized design in Algorithm 10, since it allows us to parameterize it by any conceivable fitter function even besides First-Fit and Best-Fit. So, it provides a basis for future experimentation with respect to fitting. Also, we implemented the simplified design for First-Fit and the observed solving time gains were insignificant for most DNNs.

Set Offsets. This subroutine, Algorithm 13, performs a sequential assignment of tensor offset values within a block, once the start offset for the whole block is determined. The offset value for each tensor is calculated by displacing *start_offset* by the total size of tensors already considered.

Algorithm 12: Best-Fit

Input : Two start offset and gap pairs, namely $(lower', gap')$ and $(lower'', gap'')$.
Output : True if $(lower', gap')$ is sorted before $(lower'', gap'')$ in Best-Fit, otherwise False.

return $gap' < gap''$ OR
  $(gap' = gap''$ AND $lower' < lower'')$;

Algorithm 13: SetOffsets

Input : A *block* of tensors and its *start_offset*.
Output : Assign tensor offsets within the block.

$current\_offset \leftarrow start\_offset$;
$tensor \leftarrow block.head()$;
while $tensor \neq NULL$ **do**
  $offset(tensor) \leftarrow current\_offset$;
  $current\_offset \leftarrow current\_offset + size(tensor)$;
  $tensor \leftarrow tensor.next()$;

E Section 4: Example of Many Objects vs Single Object

In Section 5 experiments, Many Objects returned a better solution than Single Object for two popular DNNs, namely Tiny-BERT and ResNet-50. Here, we present a toy example to demonstrate how this may be possible depending on DNN topology and the corresponding set of unsafe pairs. In Figure 7, we consider a toy DNN in the single-stream case, where operators execute in increasing order of their assigned (global) topological sorting.

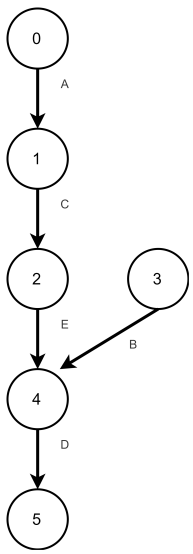
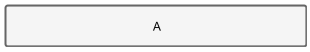


Figure 7: Toy example for algorithms' comparison.

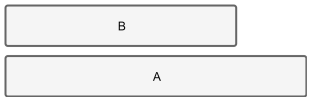
There are five tensors produced, labeled by alphabet letters. The labeling is made based on on the decreasing order of

sizes we assign them, and corresponds to the order in which the tensors are sorted in the heuristics. There is tensor A of size 1024, B of size 768, C of size 640, D of size 512 and E of size 256. From the given DNN, we extract the lifetimes as follows: $A \to [0, 1]$, $B \to [3, 4]$, $C \to [1, 2]$, $D \to [4, 5]$, $E \to [2, 4]$. Given these lifetimes, the set of unsafe pairs arising is $\{\{A, C\}, \{B, D\}, \{B, E\}, \{C, E\}, \{D, E\}\}$.

In Figure 8, we run Single Object (Algorithm 1), whereas in Figure 9, we run Many Objects (Algorithm 2). In both cases, tensors are sorted by decreasing size to be considered by the algorithms and First-Fit is used as fitter within Fit. Essentially, the two algorithms diverge at the fourth step. Due to lack of space in the first object, Many Objects allocated tensor D in the second object. This leaves a gap between B and D, which has just enough size for E to be placed in it in the next step. On the other hand, Single Object has to use extra memory equal to the size of E. Given the placement after step 4, and the set of unsafe pairs, E is placed at the end of memory space.

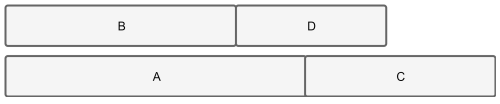


(a) Step 1: A assigned at offset 0.

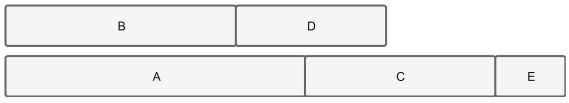


(b) Step 2: B assigned on top of A, since no unsafe pair.

(c) Step 3: C assigned after A, due to unsafe pair.



(d) Step 4: D assigned after B as no unsafe pair with A and C.

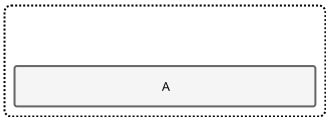


(e) Step 5: E assigned after D due to unsafe pairs with B, D, C.

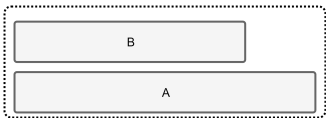
Figure 8: Single Object for toy DNN uses size 1920.

F Section 5: Footprint Lower Bound

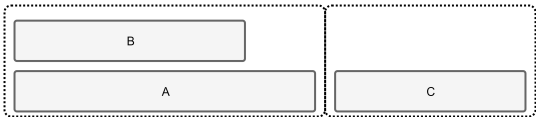
Assume we execute either Algorithm 1 or 2 to obtain the set of unsafe pairs. In a graph-theoretic perspective, consider an auxiliary undirected graph, where each node corresponds to a tensor and has a weight equal to the tensor size, and two nodes have an edge connecting them if they form an unsafe pair. Consider a clique, that is, a complete subgraph, in



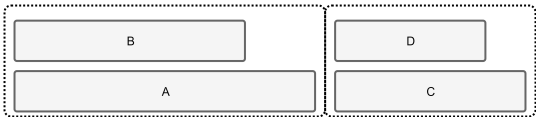
(a) Step 1: A defines a new object of size 1024.



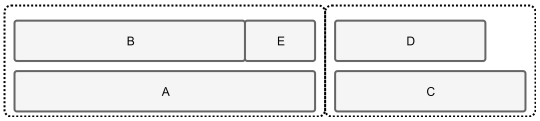
(b) Step 2: B fitted inside object of A, since no unsafe pair.



(c) Step 3: C defines a new object (640), since $\{A, C\}$ is unsafe.



(d) Step 4: D does not fit in first object since $\{B, D\}$ is unsafe.



(e) Step 5: E fits in first object.

Figure 9: Many Objects for toy DNN uses size 1664.

this auxiliary graph: no two tensors participating in a clique may reuse memory. Hence, a natural lower bound we might consider is the maximum weight of a clique. However, not only is this a notorious NP-hard problem (Garey & Johnson, 1979), it also might be far from the optimal solution of memory allocation. Depending on the set of unsafe pairs, tensors outside the max-weight clique may not reuse the memory of tensors forming that clique.

Given the above, a more "trustworthy", and tractable, lower bound is the lifetime based lower bound. Consider a single-stream execution with a global topological sorting $\{1, 2, \ldots, n\}$, where n is the number of nodes. Each tensor is associated with a single lifetime start and end. For each "time step" $i \in \{1, 2, \ldots, n\}$, let sum_i denote the sum of sizes of tensors whose lifetime includes i. A lower bound for the single-stream model is given by $\max_i sum_i$, and it captures the maximum memory needed over all steps of execution. In this paper, we assume all local topological sortings are extracted by the same global sorting of the DAG. In other words, the multi-stream graph is derived from the single-stream one. The unsafe pairs set of the multi-stream case is always a superset of the single-stream case. This lifetime lower bound serves as a natural comparison for our

computed memory plans in the multi-stream setting and we use it as a baseline to evaluate our footprint size quality in Section 5. Note this lower bound disregards the set of (many for some networks) contiguous constraints. Therefore, it might underestimate the solution quality. Nonetheless, it is experimentally proven valuable as shown in Tables 2, 3.

G SECTION 6: OTHER RELATED WORK

We discuss a related model in literature, where tensors are required to be grouped into *shared objects* of memory. Each tensor is assigned to a single shared object and a shared object holds *at most* one tensor at each point in time. In this case, the problem becomes equivalent to the NP-hard *Max-Coloring* problem in interval graphs (Escoffier et al., 2006; Pemmaraju et al., 2004) and it is a generalization of the seminal register allocation problem (Sethi, 1975).

While multiple works investigate Scheduling with Conflicts, see Section 4, they focus on exact solutions (Kowalczyk & Leus, 2017; Hà et al., 2021) and branching approaches (Mallek & Boudhar, 2020), which take a significant amount of time even for instances of very small size (a few tens or hundreds of jobs) and cannot be used to compare with the speed we achieve in our experiments (few seconds even for huge graphs with tens of thousands tensors). Besides, constraints dealing with contiguity have not been thoroughly investigated in scheduling literature. We are only familiar with the work in (Lin & Lin, 2013), where jobs partaking in a contiguous constraint must be processed successively, but unlike our case, there is no restriction on their desired order.

In this paper, our focus is to optimize memory management for large training workloads, hence we present the relevant results. We know there exist other experts within the MindSpore community who have experimented with our solution on inference workloads for applications, for example in MEGA-Fold inference to increase sequence length[2].

[2] https://cloud.tencent.com/developer/article/2095936