MAGIS: Memory Optimization via Coordinated Graph Transformation and Scheduling for DNN

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

Zijian Ding*
bradyd@cs.ucla.edu
University of California, Los Angeles
United States

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

Chengrui Zhang
zhangchr@stu.pku.edu.cn
Peking University
China

Jingwen Leng
leng-jw@cs.sjtu.edu.cn
Shanghai Jiao Tong University
China

Xuanzhe Liu
xzl@pku.edu.cn
Peking University
China

Yun Liang†
ericlyun@pku.edu.cn
Peking University
China

Abstract

Recently, memory consumption of Deep Neural Network (DNN) rapidly increases, mainly due to long lifetimes and large shapes of tensors. Graph scheduling has emerged as an effective memory optimization technique, which determines the optimal execution, re-computation, swap-out, and swap-in timings for each operator/tensor. However, it often hurts performance significantly and can only manipulate tensors' lifetimes but not shapes, limiting the optimization space. We find that graph transformation, which can change the tensor shapes and graph structure, creates a new trade-off space between memory and performance. Nevertheless, graph transformation are applied separately so far, with primary focus on optimizing performance and not memory.

In this paper, we propose MAGIS, a DNN memory optimization framework that coordinates graph transformation with graph scheduling. MAGIS uses a hierarchical tree to represent Fission Transformation (F-Trans), a type of transformation which can effectively reduce tensor shapes in a sub-graph. To keep the complexity low, we build a light-weight search space based on graph structure analysis. MAGIS decomposes graph scheduling into graph transformation and re-ordering and designs an incremental scheduling algorithm to alleviate the scheduling overhead after each graph transformation step to efficiently coordinate them. Experimental results show that compared to state-of-the-art works, MAGIS only uses 15%~85% of their peak memory usage with the same latency[1] constraint and obtains a better Pareto boundary in dual-objective optimization of memory and performance. Our code is now available at https://github.com/pku-liang/MAGIS.

ACM Reference Format:

Renze Chen, Zijian Ding, Size Zheng, Chengrui Zhang, Jingwen Leng, Xuanzhe Liu, and Yun Liang. 2024. MAGIS: Memory Optimization via Coordinated Graph Transformation and Scheduling for DNN. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651330

1 Introduction

As deep neural networks (DNNs) become more complex in terms of topology and size, the memory consumption of DNNs keeps growing, which poses great challenges for both training and inference. The memory consumption turns out to be more important when larger models come to stage [7, 12, 51]. The memory consumption increase can be attributed to two main factors. First, there are numerous tensors with **long lifetimes**, such as model parameters [7, 12, 15, 40, 51], activations during the training's forward pass [5, 10, 38, 42], and intermediate tensors in complex networks [44, 73, 75]. Second, many tensors have **large shapes**, including large batch sizes for efficient training/inference, long sequence lengths in language models [7, 12, 51], and high resolutions in image-related models [21, 40, 45].

*Work done while the author was a student at Peking University.

†Corresponding author.

[1] In this paper, the terms "performance" and "latency" are interchangeably used, both referring to the time taken by a DNN to complete one inference/training epoch.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651330

Optimizing memory usage for DNNs becomes crucial for both server and mobile computing devices. GPUs, for instance, NVIDIA GeForce RTX 3090, provide only tens to dozens of GB of memory, while the large-batch training or inference sometimes requires several tens or even hundreds of GB of memory. Memory optimization is beneficial for executing large DNN, enabling co-location of multiple tasks in memory [32], and reducing cross-card communications in distributed learning. Similarly, mobile CPUs such as Qualcomm Snapdragon 888 provide only a few tens of GB of memory and many background applications may reside in memory, which greatly limits the space for DNN. Memory optimization is beneficial for deploying DNNs on mobile devices without consuming too much background memory.

Graph scheduling is a class of widely used memory optimization techniques for DNNs, mainly including rematerialization [5, 10, 17, 18, 24, 27–29, 37, 38, 47], swapping [5, 20, 22, 30, 37–39, 41, 42, 57], and re-ordering [3, 22, 58, 72]. Its core idea is to manipulate the lifetimes of tensors by scheduling when each operator/tensor computes, evicts, recomputes, offloads, and reloads, thereby reducing the peak amount of tensors simultaneously residing in memory. However, because of the overhead introduced by re-computation or data transfer, it frequently leads to a notable reduction in performance. Moreover, although it operates the lifetimes of tensors, it does not affect the tensor shapes, which limits its potential optimization space.

On the other hand, graph transformation is a class of optimization techniques based on equivalent transformations of graphs. Existing works have achieved good results in optimizing DNN performance [25, 26, 54, 56, 62]. They employ rule-based sub-graph substitution technique, which can be roughly divided into two types: Aggregation Transformation (A-Trans), like Figure 1 (a), which enhances hardware utilization to improve performance by aggregating small operators into larger ones at the cost of temporally increased memory usage; Interim Transformation (I-Trans), such as Figure 1 (b), which generally exploits algebraic equivalence to provide opportunities for other graph transformations. In addition, we find that the dual of A-Trans, which we call **Fission Transformation (F-Trans)**, like Figure 1 (c), can effectively reduce memory at the cost of lower hardware utilization by splitting some large operators into smaller ones and executing only one of the split parts at a time.

However, graph transformation for memory optimization poses two main challenges. **(1) Complexity introduced by F-Trans.** On one hand, F-Trans leads to rapid growth in the size of the graph (as shown in Figure 1 (c), where the number of nodes almost doubles after transformation), which hinders subsequent optimization; on the other hand, F-Trans itself has a vast search space, as it can be applied to almost every sub-graph. **(2) Correlated graph transformation and graph scheduling.** Graph transformation involves a trade-off between memory and performance (e.g., A-Trans trades memory for performance, and F-Trans does the opposite), but the final memory usage and performance are also traded by graph scheduling. This necessitates the need for efficient coordinated optimization between graph transformation and scheduling, which is challenging since both of them are complicated optimization.

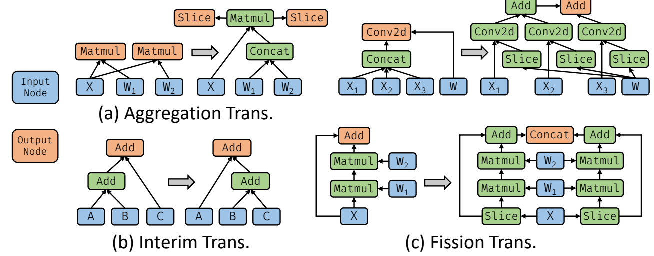


Figure 1. Examples of graph transformations. **(a)** and **(b)** are transformations borrowed from TASO [25], which are used to optimize performance. **(c)** is the dual of Aggregation Trans. and can effectively trade memory with performance.

To tackle these challenges, we propose MAGIS, a DNN memory optimization framework through coordinated graph transformations and scheduling. To address the complexity problem of F-Trans, we propose Fission Hierarchy Tree (F-Tree) to express the graph structure after F-Trans, without actually transforming the graph into a complex structure. Although such design somehow limits the search space, it keeps the complexity low, making it easier for subsequent transformation and scheduling to search for better solutions. We then propose analytic methods to select proper sub-graphs and dimensions for F-Trans to construct a light-weight F-Tree, effectively reducing the search space of F-Trans.

To address the second challenge, our goal is to alleviate the complexity of graph scheduling after each graph transformation step. We firstly decompose re-materialization and swapping into graph transformations and re-ordering, where re-materialization and swapping are two important scheduling techniques which can trade memory with performance, while re-ordering is a scheduling method that optimize memory without hurting performance. Such decomposition moves the memory & performance trade-off completely to the transformation phase, and the scheduling phase can only focus on memory optimization through re-ordering. It makes the scheduling after each graph transformation step much simpler, and fuses the memory & performance trade-offs into the unified search space of graph transformation. Then, we design an incremental graph scheduling algorithm that efficiently obtains a new schedule based on the previous schedule and the current transformation, further reducing scheduling time.

Our contributions can be summarized as follows:

- We design and implement MAGIS, a memory optimization framework based on coordinated graph transformation and graph scheduling.

- We formalize graph fission transformation, represent it based on hierarchy tree, and use graph analysis to reduce its search space.
- We propose transformations and algorithms that efficiently coordinate graph transformation and graph scheduling for memory optimization.

We compare MAGIS with state-of-the-art graph scheduling-based memory optimization frameworks on various DNNs. Experimental results demonstrate that MAGIS can optimize original peak memory usage to 15%~50% with no more than 10% latency overheads. Compared to state-of-the-art methods, MAGIS can optimize peak memory to only 15%~85% of theirs with the same latency constraint, and can achieve a 1.25× speedup over them under the same memory constraint, obtaining a better Pareto boundary in dual-objective optimization of memory and latency. Our code is now available at https://github.com/pku-liang/MAGIS.

2 Background & Motivation

Table 1. Notations

Notation	Description/Definition
$\mathcal{V}(G), \mathcal{E}(G)$	operators, dependencies of G
$\mathcal{D}(G), \mathcal{T}(G)$	dimension graph, dominator tree of G
$\texttt{cost}(G), \texttt{cost}(v)$	execution latency of G and $v \in \mathcal{V}(G)$
$\texttt{size}(v)$ or $\lvert v\rvert$	output tensor size of operator v
$G.\texttt{pre}(v), G.\texttt{suc}(v)$	predecessors, successors of $v \in \mathcal{V}(G)$
$G.\texttt{anc}(v), G.\texttt{des}(v)$	ancestors, descendants of $v \in \mathcal{V}(G)$
$\texttt{inps}(G), \texttt{outs}(G)$	inputs, outputs of G
$G.\texttt{sub}(S)$ or $G[S]$	sub-graph of G induced from $S \subseteq \mathcal{V}(G)$
$G.\texttt{inps}(S)$	nodes consumed by $S \subseteq \mathcal{V}(G)$ from outside
$G.\texttt{outs}(S)$	nodes produced by $S \subseteq \mathcal{V}(G)$ for outside

2.1 Computation Graph

Graph Structure. DNN during training or inference process is often represented as "computation graph" G (abbreviated as "graph"). $V = \mathcal{V}(G)$ is the set of operators, each of which has several input tensors and one output tensor, and $E = \mathcal{E}(G) \subseteq V \times V$ is the set of data dependencies between operators. $(v_1, v_2) \in E$ means that the output tensor of v_1 is one of the input tensors of v_2. Related notations used in this paper are shown in Table 1. In cases where there is no ambiguity, we use $\texttt{xxx}(v)$ as an abbreviation for $G.\texttt{xxx}(v)$. Some notations can be derived from other notations, for example, $G.\texttt{inps}(S) = (\bigcup_{v\in S} G.\texttt{pre}(v))\backslash S$, and $G.\texttt{outs}(S) = (\texttt{outs}(G) \cup \bigcup_{v\in\mathcal{V}(G)\backslash S} G.\texttt{pre}(v)) \cap S$. A node u dominates node v if every path from the entry node to v must go through u; and then u is v's dominator. The intermediate dominator of a node v is the dominator of v that is dominated by all the dominators of v except v itself. The dominator tree [4] is the tree where each node's parent is its intermediate dominator in the graph. A computation graph usually has many input nodes (e.g., input tensor, label tensor, and weight tensors), so the dominator tree we use here usually takes the input tensor as the entry. Note that for $T = \mathcal{T}(G)$, T itself is also a graph, and the operations in Table 1 are also applicable to it. For example, the set of child nodes of a node v in T is $T.\texttt{suc}(v)$. The nodes of T also belong to G, i.e., $\mathcal{V}(\mathcal{T}(G)) \subseteq \mathcal{V}(G)$.

Execution Latency. In single machine situation (e.g., single-card GPU), the operators in the graph are generally executed in order, and the order $s = (v_1, v_2, ..., v_n)$ must satisfy the data dependencies between operators. The graph execution latency can be estimated as the sum of the latency of the operators: $\texttt{cost}(G) \approx \sum_{v\in\mathcal{V}(G)} \texttt{cost}(v)$.

Memory Usage. Given a topo-order $s = (v_1, v_2, ..., v_n)$, assuming that i is the timestamp when the ith operator is finished, we can calculate the lifetime of the output tensor of each operator v_i: the start timestamp is $S_i = i - 1$, and the free timestamp is $F_i = \max_{v_j \in suc(v_i)} j$. Based on the lifetime of each tensor, the set of tensors that are active during the execution of v_i is $A_i = \{v_j \mid S_j \le i \le F_j\}$. Then the **active memory usage** during v_i's execution is $M_i = \sum_{u\in A_i} |u|$, and the **peak memory usage** during the execution of graph G is: $M_{peak} = \max_i M_i$. We define **memory hot-spots** as the set of tensors that contribute to the peak memory usage, that is, the tensors that are active when peak memory usage is reached: $H = \bigcup\{A_i \mid i \in \{1, 2, ..., n\} \wedge M_i = M_{peak}\}$.

2.2 Graph Scheduling and Transformation

Graph scheduling is a class of widely used DNN memory optimization techniques, which manipulates the lifetimes of tensors to schedule when to execute (re-ordering [3, 58]), evict & re-compute (re-materialization [5, 10, 18, 24, 27, 37, 38]), and offload & reload (swapping [5, 20, 22, 37, 38, 41, 42, 57]) each operator/tensor without influencing tensor shapes.

Graph transformation is a class of techniques to optimize computation graphs by mutating their structures while preserving semantics. Existing works [25, 26, 56, 62] mainly optimize latency via rule-based sub-graph substitution, which can be categorized into two types: Aggregation Transformation (A-Trans), aggregating small operators into larger ones to trade memory for latency; Interim Transformation (I-Trans), mostly based on algebraic equivalence to provide opportunities for other transformations.

2.3 Motivation

We find that appropriate graph transformations can also improve the memory usage of graphs. For example, as shown in Figure 2 (c), splitting operators reduces peak memory usage at the cost of more operator calls and decreased hardware utilization. With the help of graph transformation, memory optimization of DNNs can be greatly enhanced. For example, in Figure 2 (a), there's a simplified graph structure commonly observed in DNN training or some DNNs with long skip-connections [23, 44, 73, 75]. It has a peak memory usage of 1056 since 33 tensors with size 32 are alive when computing the 33-th operator, which exceeds the memory limit of

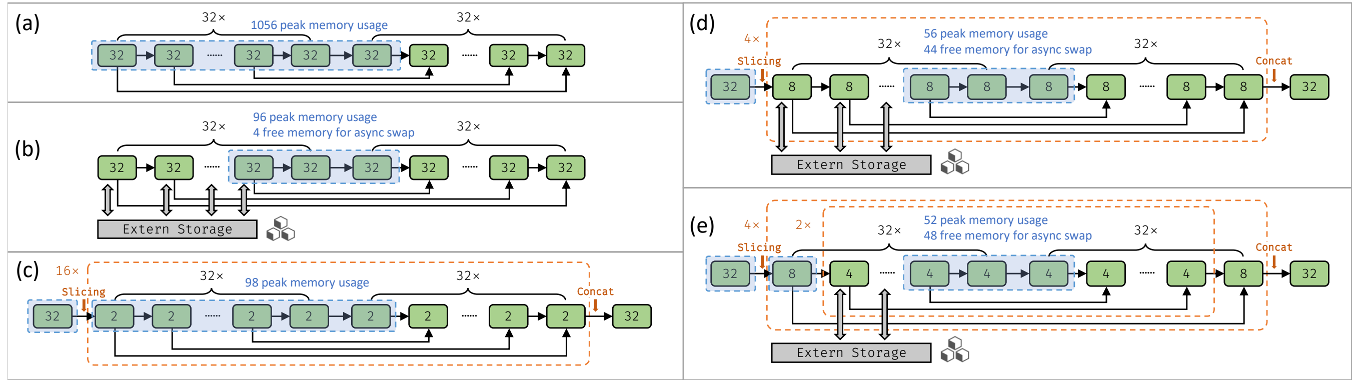


Figure 2. Motivation examples with memory limit of 100. **(a)** Without any optimization. **(b)** Using swapping. **(c)** Using fission transformation. **(d)(e)** Using fission transformation and swapping.

100. In Figure 2 (b), although graph scheduling alone can restrict memory usage to 100 by swapping temporally unused tensors into external storage , it causes long latency due to data transfer. However, incorporating graph transformations, as shown in Figure 2 (d), more memory is saved and asynchronous swapping can be utilized to hide data transfer latency. Although the hardware utilization decreases, the latency penalty can be compensated by the efficiency gain provided by asynchronous swapping in this case.

We name the transformation used in Figure 1 (c) and Figure 2 (c) (d) (e) as **Fission Transformation** (F-Trans), which is the dual of A-Trans and can effectively optimize the memory usage by splitting operators. However, the existing graph transformation techniques based on rule-based sub-graph substitution [25, 26, 56, 62] can not be used for F-Trans. First, F-Trans often greatly increases the graph complexity, hindering subsequent optimization. Second, F-Trans involves a vast search space, since it can be applied to almost any sub-graph. For example, Figure 2 (e) uses two different F-Trans, and even for such a simple network in this example, the search space for feasible F-Trans is huge. Finding efficient ways to represent and search for F-Trans is a challenge.

In addition, coordinating graph transformations with graph scheduling is critical for optimizing memory usage with graph transformations. Figure 2 (c) shows that applying graph transformations alone can optimize memory usage, but excessively fine-grained operator splitting may result in high performance costs. Instead, combining graph transformation and graph scheduling as in Figure 2 (d) can significantly reduce memory usage and achieve shorter latency by jointly balancing the memory and performance trade-offs of both transformation and scheduling.

3 Design Overview

Figure 3 shows the overall design of MAGIS. It accepts a DNN graph and outputs the optimized graph and schedule. MAGIS has four main components: M-State, M-Analyzer, M-Rules, and M-Optimizer.

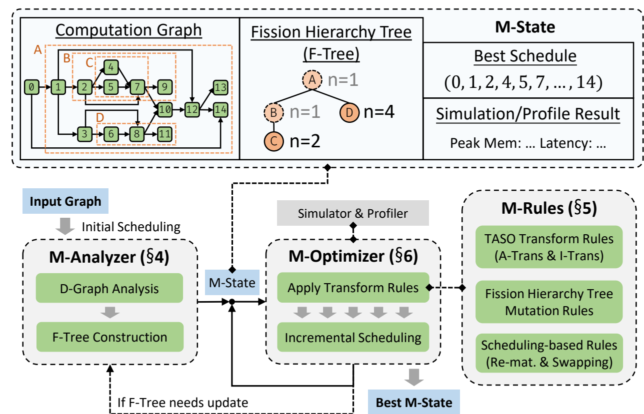
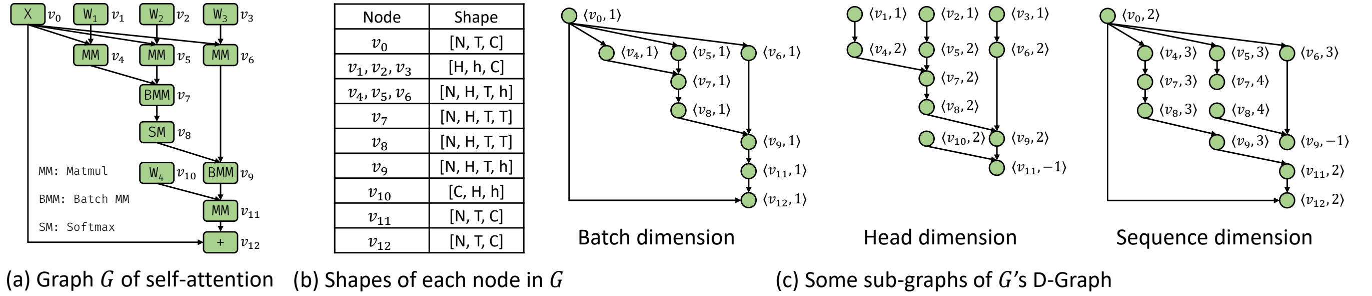


Figure 3. Overview of MAGIS.

M-State represents the optimization status, including computation graph, fission hierarchy tree (F-Tree), best schedule, and simulation & profile result. F-Tree represents the hierarchical search space of fission transformation (F-Trans), where a node with $n = 1$ represents a potential sub-graph & dimension candidate for F-Trans, and a node with $n > 1$ represents a sub-graph already been split via F-Trans along some dimension into n parts. M-Analyzer generates the search space of fission transformation (F-Trans), by constructing the fission hierarchy tree (F-Tree) according to the computation graph. M-Optimizer coordinates the graph transformations (including F-Trans) and scheduling to optimize the latency & memory. M-Rules provide the transformations for M-Optimizer, including "TASO rules" used in previous works [25, 26, 56, 62], F-Tree mutation rules for manipulating F-Tree to reflect F-Trans applications on the graph (§5.1) , and scheduling-based rules decomposed from graph scheduling. Note that F-Trans is decoupled as F-Tree and mutation rules applied on the F-Tree. These rules are integrated with others (e.g., TASO rules, scheduling-based rules) in M-Rules, forming a unified optimization space explored by the M-Optimizer.



Node	Shape
v_0	[N, T, C]
v_1, v_2, v_3	[H, h, C]
v_4, v_5, v_6	[N, H, T, h]
v_7	[N, H, T, T]
v_8	[N, H, T, T]
v_9	[N, H, T, h]
v_{10}	[C, H, h]
v_{11}	[N, T, C]
v_{12}	[N, T, C]

(a) Graph G of self-attention (b) Shapes of each node in G (c) Some sub-graphs of G's D-Graph

Figure 4. Example of D-Graph. N, T, C, H, h represents batch-size, seq-len, hidden-dim, num-heads, head-dim respectively.

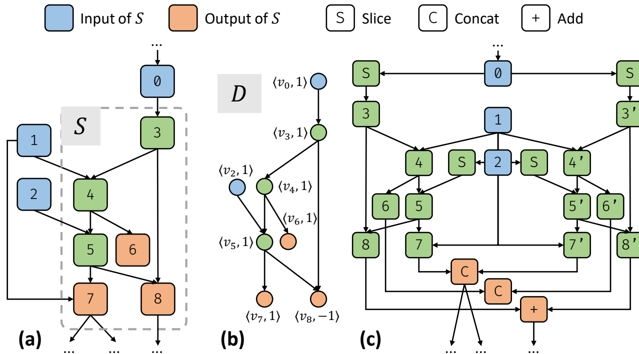


Figure 5. F-Trans $f = (S, D, n)$ $(n = 2)$ in graph G, which is simplified from the training-graph of an MLP. **(a)** Sub-graph $S = \{v_3, v_4, v_5, v_6, v_7, v_8\}$. **(b)** D-Graph D, which represents the batch-dim of S's activation. **(c)** Result graph after F-Trans.

MAGIS takes a computation graph as input. The graph and its initial schedule are analyzed by M-Analyzer, which constructs the F-Tree, outputs the initial M-State and sends the M-State to the M-Optimizer. M-Optimizer applies M-Rules to produce new M-States by mutating some sub-graphs or sub-F-trees. Note that the rules will not choose the sub-graph spanning the boundary of the sub-graphs affected by F-Trans (the sub-graph belonging to the F-Tree node with $n > 1$) for transformation. This is because for a region R already affected by F-Trans, the rules will not transform the sub-graph S that partly intersects with R, as some nodes of S will be split during execution while some not. It then performs fast incremental scheduling on these new graphs, utilizing the mutated graph region of the transformation and prior schedules, to quickly derive near-optimal schedules and associated profile results. Effective M-States are iteratively fed back to M-Optimizer. Besides, if a graph transformation is applied on a sub-graph that has not been affected by F-Trans, M-optimizer will query M-Analyzer to update the F-Tree in the new M-States.

The remainder of this paper is structured as follows: §4 introduces M-Analyzer of MAGIS, §5 discusses M-Rules, and §6 details M-Optimizer.

4 M-Analyzer

In this section, we will first introduce Dimension Graph (D-Graph) and use it to define F-Trans. Then we propose F-Tree as an abstraction of the optimization space/state of F-Trans, and provide an algorithm to construct a light-weight F-Tree considering F-Trans only on some sub-graphs that are selected based on dominator tree and memory hot-spots.

4.1 Dimension Graph

Intuitively, an F-Trans splits a sub-graph along a "dimension" running through it. Therefore, we propose Dimension Graph (D-Graph) to identify the graph-level dimensions.

Given a graph G where $v \in \mathcal{V}(G)$ has s_v dimensions in its output tensor and r_v reduce-axes in its computation, we define D-Graph $D = \mathcal{D}(G)$ where for each $v \in \mathcal{V}(G)$ and $i = -r_v, ..., -2, -1, 1, 2, ..., s_v$, there's $\langle v, i \rangle \in \mathcal{V}(D)$. For each $(u, v) \in \mathcal{E}(G)$, if the ith dimension of u and j^{th} dimension of v correspond to the same spatial-axis[2], then there's $(\langle u, i \rangle, \langle v, j \rangle) \in \mathcal{E}(D)$; and if the ith dimension of u corresponds to the jth reduce-axis of v's computation, then there's $(\langle u, i \rangle, \langle v, -j \rangle) \in \mathcal{E}(D)$. For instance, a `MatMul` operator c (expressed as $c[m, n] = \sum_k a[m, k] \times b[k, n]$, where m, n are c's dimensions, and k is the reduce-axis) with its inputs $a, b \in \texttt{pre}(c)$ provides connections $(\langle a, 1 \rangle, \langle c, 1 \rangle)$, $(\langle a, 2 \rangle, \langle c, -1 \rangle)$, $(\langle b, 1 \rangle, \langle c, -1 \rangle)$, $(\langle b, 2 \rangle, \langle c, 2 \rangle) \in \mathcal{E}(D)$.

Example. Figure 4 (a) illustrates graph G, extracted from a transformer block [55], with shapes detailed in part (b). Part (c) depicts some sub-graphs of $\mathcal{D}(G)$, like one with batch-dimensions from tensors excluding v_1, v_2, v_3, v_{10}, one with head-dimensions from tensors excluding v_0, v_{12}, and another with sequence-dimensions from tensors except v_1, v_2, v_3, v_{10}.

4.2 Fission Transformation

With the help of D-Graph, we can define an F-Trans of graph G as $f = (S, D, n)$, where $S \subseteq \mathcal{V}(G)$, D is the D-Graph to split sub-graph $G[S]$ along, n is the fission number. It has the following constraints: **(1)** $G[S]$ is weakly connected. **(2)** $G[S]$ is convex: $G.\texttt{inps}(S) \cap \bigcup_{v \in G.\texttt{outs}(S)} G.\texttt{des}(v) = \emptyset$. **(3)** The graph after fission has no redundant computation, requiring

[2] Here we do not consider spatial-axis with sliding-window, such as the height axis of a 3×3 convolution; we will improve it in future work.

Algorithm 1: M-Analyzer: F-Tree Construction

```
input  : graph: G; max-level: L
output : fission hierarchy tree: F
1  F := ∅;
2  H := MemoryHotspots(G);
3  for D ∈ connected components of 𝒟(G) do
4      G' := subgraph of G induced from D;
5      T := 𝒯(G');
6      s := GetScores(G', T, H);
7      s_max = max_{v∈𝒱(G')} s[v];
8      if s_max ≤ 0 then continue;
9      for i ∈ {1, 2, ..., L} do
10         V := {v ∈ 𝒱(G') | i/L ≤ s[v]/s_max < (i+1)/L};
11         for v_dom ∈ {v ∈ V | T.des(v) ∩ V = ∅} do
12             S := T.des(v_dom) \ {v_dom};
13             D' := subgraph of D induced from S;
14             f := (S, D', 1);
15             if f is valid then F := F ∪ {f};
16 return F;
```

that $\forall v \in S$, there's exact one $i \in \mathbb{Z}$ s.t. $\langle v, i\rangle \in \mathcal{V}(D)$, and $\forall (u, v) \in \mathcal{E}(G[S])$, $\exists i, j \in \mathbb{Z}$ s.t. $(\langle u, i\rangle, \langle v, j\rangle) \in \mathcal{E}(D)$.

Given an F-Trans $f = (S, D, n)$ of G, the result graph after F-Trans is a graph with n split parts of $G[S]$. $\forall u \in G.\texttt{inps}(S)$, if $\exists i > 0$ s.t. $\langle u, i\rangle \in \mathcal{V}(D)$, then u will be sliced for each split part, otherwise shared by them. $\forall v \in G.\texttt{outs}(S)$, if $\exists i > 0$ s.t. $\langle v, i\rangle \in \mathcal{V}(D)$, then v will be computed by merging the related outputs of split parts, otherwise reducing them. Note that, the split parts are executed sequentially to save memory by timely freeing intermediate tensors of each part at the cost of lower hardware utilization (e.g., parallelism, locality) due to smaller operator shapes.

Example. Figure 5 demonstrates an example of F-Trans $f = (S, D, n)$ with $n = 2$. v_1 is a weight tensor, so there's no $\langle v_1, i\rangle \in \mathcal{V}(D)$; so in the result graph v_1 is shared by each split part. Other inputs, v_0 and v_2, are sliced for each part. v_8 is the gradient of v_1, computed by adding along batch-dim, so $\langle v_8, -1\rangle \in \mathcal{V}(D)$; so in the result graph v_8 is computed by adding the outputs of each split part. Other outputs, v_6 and v_7, are computed by concatenating the outputs of each part.

4.3 Fission Hierarchy Tree

Directly applying F-Trans to a graph will significantly increase the complexity, especially when the fission number is large. Since each F-Trans divides a graph into several isomorphic sub-graphs, we can save only one of them. Instead of transforming the original graph directly, we construct a fission hierarchy tree (F-Tree). Each tree-node in the F-Tree records a F-Trans $f = (S, D, n)$. For any tree-node $f = (S, D, n)$ and its parent $f' = (S', D', n')$, we have $S \subseteq S'$. Figure 3 displays an example of F-Tree, where each node represents a sub-graph surrounded by a dashed box in the left-side graph and the n next to the node is the fission number. When $n = 1$, it indicates that the node is an fission candidate, and when $n > 1$, it indicates that the subgraph of the node has been split into n parts by F-Trans. Such abstraction significantly reduces the complexity of subsequent graph transformation and scheduling.

However, the search space for F-Trans on graph G is still large, reaching up to $O(2^{|\mathcal{V}(G)|^2})$ since almost any convex sub-graph can be a fission candidate. Indeed, arbitrarily applying F-Trans does not guarantee peak memory reduction. Effective memory saving can be achieved only when F-Trans targets sub-graphs containing memory hot-spots (§2.1).

Analysis. For an F-Trans $f = (S, D, n)$ of graph G, with memory hot-spots as H and $I = G.\texttt{inps}(S)$. M_0 and M_f represent the peak memory usages before and after F-Trans, shown in Equation (1). Since inputs I reside in memory when executing split sub-graphs, M_f should combine their sizes $\sum_{v\in I} |v|$ with $\sum_{v\in H\setminus S} |v|$ (sizes of memory hot-pots beyond S) into $\sum_{v\in(H\setminus S)\cup I} |v|$. The peak memory reduction after F-Trans, i.e., $M_0 - M_f$, is shown in Equation (2).

$$M_0 = \sum_{v\in H} |v| \qquad M_f \approx \sum_{v\in(H\setminus S)\cup I} |v| + \sum_{v\in H\cap S} \frac{|v|}{n} \quad (1)$$

$$M_0 - M_f = \sum_{v\in H\cap S}(1 - \tfrac{1}{n})|v| - \sum_{v\in I\setminus H} |v| \quad (2)$$

Metric. We can observe that to make $M_0 - M_f$ larger, we need to ensure that S includes more memory hot-spots, while I consumes less memory. To minimize input memory usage of F-Trans, we select a node and consider the sub-graph dominated by it as the fission candidate, ensuring the sub-graph has only one entry node [3]. We define a metric called "memory heat", representing the total size of hot-spots in a sub-graph dominated by a node. Given the graph G with dominator tree $T = \mathcal{T}(G)$ and memory hot-spots H, we calculate v's memory heat with Equation (3), where $H \cap T.\texttt{des}(v)$ are the memory hot-spots dominated by v. We then assign a score for each node v as shown in Equation (4), estimating the potential peak memory reduction after F-Trans on the sub-graph dominated by v, where the first term is the reduction of the sizes of memory hot-spots, and the second term is the sizes of input nodes which should reside in memory during the execution of each split part after F-Trans. We typically set $n = 2$ to ensure that just splitting the sub-graph into two parts also yields benefits.

$$\texttt{heat}(v) = \sum_{w\in H\cap T.\texttt{des}(v)} |w| \quad (3)$$

$$\texttt{score}(v) = (1 - \tfrac{1}{n})\texttt{heat}(v) - \sum_{u\in G.\texttt{inps}(T.\texttt{des}(v))\setminus H} |u| \quad (4)$$

Algorithm. Based on the metrics discussed above, we propose Algorithm 1 to construct an F-Tree. The main idea is identifying nodes with scores (Equation (4)) distributed in different intervals, since a higher score indicates more peak memory reduction of F-Trans, but may also imply larger latency overhead. The hyper-parameter L controls the number of intervals and the F-Tree's max-level. This algorithm

[3]Strictly, weight tensors may also be input nodes, as discussed in §2.1

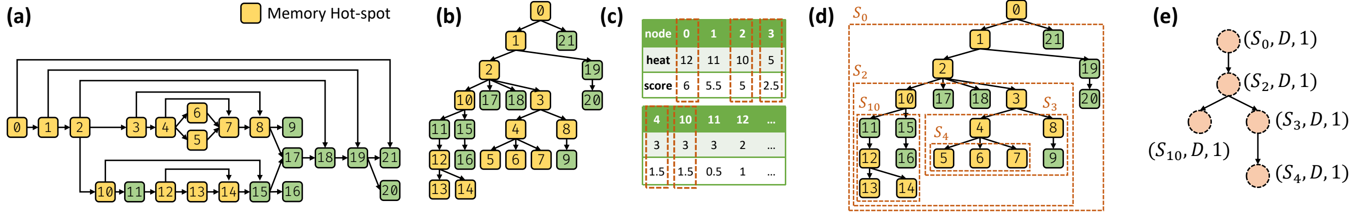


Figure 6. Example of F-Tree construction based on Algorithm 1 (with $L = 5$). Each tensor has a size of 1. **(a)** G' in Algorithm 1 line 4. **(b)** Dom $T = \mathcal{T}(G')$. **(c)** Scores calculated based on Equation (3) (4), where nodes in orange boxes are selected dominators (v_{dom} in Algorithm 1 line 11). **(d)** Selected sub-graphs (S in Algorithm 1 line 12). **(e)** Constructed F-Tree.

inputs graph G and max-level L, iterating over connected components D of $\mathcal{D}(G)$ (line 3), extracting sub-graph G' and its dominator tree T (lines 4-5), then calculating scores based on Equation (3) (4) (line 6). Upon obtaining the maximum score s_{max} (line 7), it segments $[0, 1]$ into L intervals, selecting nodes in different intervals based on normalized scores $s[v]/s_{max}$ (lines 10-11), and generating fission candidates from sub-graphs dominated these nodes (lines 12-15). The F-Tree is constructed from these sub-graphs.

Example. Figure 6 gives an example of F-Tree construction for a computation graph simplified from the training graphs of various models. For demonstration, we only show one connected component $D \in \mathcal{D}(G)$ here. Part (a) is the G' in Algorithm 1 at line 4. Part (b) shows dominator tree $T = \mathcal{T}(G')$. Part (c) shows the calculated results of `heat` and `score` based on Equation (3) (4). Here $L = 5$, so there are 5 normalized score intervals [0.2, 0.4), [0.4,0.6), [0.6,0.8), [0.8,1), [1,1], and the nodes in dashed boxes are selected. Part (d) shows the selected sub-graph nodes as fission-candidates. Part (e) shows the finally constructed F-Tree.

5 M-Rules

M-Rules in MAGIS borrow the rules of Aggregation Transformation (A-Trans) and Interim Transformation (I-Trans) from previous works like TASO [25], shown by Figure 1 (a) (b). We call these TASO Rules, which can be used to optimize latency. Beside of these, in this section, we will introduce F-Tree Mutation Rules and Scheduling-based Rules to further optimize memory and latency.

5.1 Fission Hierarchy Tree Mutation Rules

All tree-nodes $f = (S, D, n)$ of the initial F-Tree constructed by Algorithm 1 have $n = 1$. We refer them as disabled nodes, whose sub-graphs have not performed F-Trans. Node with $n > 1$ is called enabled node, which means its sub-graph has already performed F-Trans and is split into n parts. The F-Tree Mutation Rules mainly change the n of the F-Tree node to apply F-Trans to the graph. They include:

- **Enabling Rule.** It enables a disabled leaf node of F-Tree or a parent node of an enabled node without enabled ancestors, as shown in Figure 7 (a).
- **Lifting Rule.** It disables an enabled node without enabled ancestors and enables its parent node, as shown in Figure 7 (b).
- **Disabling Rule.** It disables an enabled node that has no enabled descendant node, as shown in Figure 7 (c).
- **Mutating Rule.** It increases an enabled node's fission number n to the next number that can divide the dimension length, as shown in Figure 7 (d).

With the help of M-Analyzer and above rules, we decouple F-Trans into F-Tree construction before optimization phase and F-Tree mutation during optimization phase. It can be observed that we actually start enabling leaf nodes first and gradually move towards nodes closer to the root. Since applying fission on the nodes closer to the root has a greater impact on memory and latency, we start from the leaves for smaller mutation steps and smoother search.

5.2 Scheduling-based Rules

We introduce two additional operators, `Store` and `Load`, to represent swapping behaviour in graph scheduling. Based on this, we add four rules as follows:

- **Re-materialization Rule.** It separates one user B from an operator A with multiple users and lets it use a recalculated operator A', as shown in Figure 8 (a) (b).
- **De-re-materialization Rule.** It is the dual of the re-materialization rule and combines two operators A and A' of the same type with the same inputs into a single operator, as shown in Figure 8 (c) (d).
- **Swapping Rule.** It inserts `Store` and `Load` between an operator A and one of its users B to represent that A will be swapped-out to external storage first, and then swapped-in when B needs to use it, as shown in Figure 8 (e).
- **De-swapping Rule.** It is the dual of the swapping rule and removes `Store` and `Load` between two operators, as shown in Figure 8 (f).

With the help of the rules above, we can decompose graph scheduling into graph transformation and re-ordering, where transformation phase decides what operators need to be recomputed / swapped, and re-ordering decides when to recompute / swap. Then the trade-offs between memory and latency can be moved to graph transformation phase, and

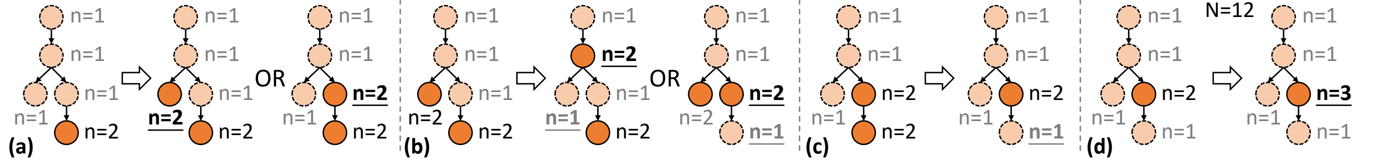


Figure 7. Illustrations of F-Tree Mutation Rules. **(a)** Enable an F-Tree node. **(b)** Lift an F-Tree node. **(c)** Disable an F-Tree node. **(d)** Increase the fission number n (with dimension length $N = 12$).

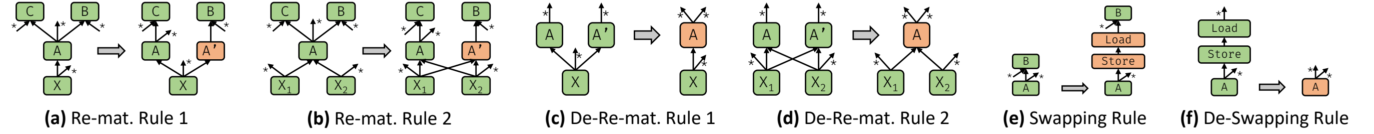


Figure 8. Scheduling-based Rules, representing the transformations decomposed from graph scheduling. The edges marked with an asterisk (*) represent zero or multiple edges

Algorithm 2: M-Optimizer: Incremental Scheduling

input : old, new graph: G_{old}, G_{new};
old mutated sub-graph nodes: S_{old};
schedule of old graph: ψ_{old}

output : schedule of new graph: ψ_{new}

1 **function** `GetRescheduleInterval`(G, S, ψ):
2   **function** `ExtendBound`(i, d):
3         $\hat{n} = \infty$; $v := \psi[i]$; $l := 0$;
4   **while** $l < 20 \wedge (\hat{n} > 10 \vee \mathrm{nw}(v) < 4) \wedge \mathrm{nw}(v) < \hat{n}$ **do**
5             $\hat{n} := \mathrm{nw}(v)$; $i := i + d$; $v := \psi[i]$; $l := l + 1$;
6   **return** i;
7     $I_S := \{i \mid i = 1, \dots, |\psi| \text{ if } \psi[i] \in S\}$;
8   **return** `ExtendBound`$(\min I_S, -1)$, `ExtendBound`$(\max I_S, 1)$;
9 beg, end := `GetRescheduleInterval`$(G_{old}, S_{old}, \psi_{old})$;
10 $S_{new} := \mathcal{V}(G_{new}) \setminus (\psi_{old}[:\mathrm{beg}] \cup \psi_{old}[\mathrm{end}:])$;
11 $\Psi :=$ {`DpSchedule`(S) | $S \in$ `GraphPartition`(S_{new})};
12 **return** `Merge`$(\psi_{old}[:\mathrm{beg}]$, `MergeSubSched`$(\Psi)$, $\psi_{old}[\mathrm{end}:])$;

graph scheduling phase only needs to consider re-ordering that generally has no effect on total execution latency. Such decomposition makes the scheduling after each graph transformation step much simpler.

Heuristic. Considering the Re-materialization Rule and Swapping Rule can be applied to almost any operator, resulting in a large search space that slows down optimization, in the actual sub-graph pattern-matching process, these two rules can be selectively applied, *filtering out sub-graphs that do not contain memory hot-spots.*

6 M-Optimizer

In this section, we first introduce incremental scheduling, to efficiently generate the optimal schedule for the transformed graph using information from the mutated sub-graph and the previous schedule. We then present the top-level search algorithm, which prioritizes M-States based on both memory and latency and transforms current best M-States using M-Rules to generate new M-States.

6.1 Incremental Scheduling

To obtain memory usage and performance of a graph, we need to perform graph scheduling. Performing full graph scheduling after each graph transformation is expensive. To address this issue, we design an incremental scheduling algorithm that determines the subset of the graph that needs to be rescheduled based on the previous scheduling and the sub-graph scope impacted by the previous graph transformation. This approach allows us to perform scheduling only on the necessary sub-graphs, reducing the overhead of scheduling.

Algorithm 2 presents the details. It first obtains the sequence of operators that need to be rescheduled in the original graph by using `GetRescheduleInterval` (line 9). Next, the corresponding sub-graph S_{new} is obtained for this sequence in the new graph (line 10), which is then partitioned into several sub-graphs that can be independently scheduled using `GraphPartition` (line 11). The scheduling of each sub-graph is performed using the dynamic programming-based algorithm in previous work [3] (line 11), and finally, the resulting schedules are combined to form the schedule for the new graph, which is integrated with the schedule for the original graph (line 12).

`GetRescheduleInterval` is a crucial processes in Algorithm 2, designed to find the interval in the original schedule that needs to be rescheduled. The interval should not be too small, otherwise the rescheduled result would be sub-optimal or even incorrect. Also, the interval should not be too large, otherwise the rescheduling process will consume too much time. Trading between the optimization quality and time cost is important.

We introduce narrow waist (NW) value $\mathrm{nw}(v)$ of a node v to solve it. For a graph G and a node $v \in \mathcal{V}(G)$, $\mathrm{nw}(v)$ is defined as $|\mathcal{V}(G)| - |G.\mathrm{anc}(v)| - |G.\mathrm{des}(v)| - 1$, i.e., $|\mathcal{V}(G) \setminus G.\mathrm{anc}(v) \setminus G.\mathrm{des}(v)| - 1$. The NW value can be used to measure the number of nodes that are independent of the given node. A lower $\mathrm{nw}(v)$ implies that more nodes are dependent on v and v depends on more nodes, which makes

Algorithm 3: M-Optimizer: Search Algorithm

```
input  : input graph G; memory constraint M;
         F-Tree max-level L;
output : optimized M-State μ_best
1  function BetterThan(μ1, μ2, δ = 1):
2      return (max(μ1.mem, M), μ1.lat) <
         (max(δ × μ2.mem, M), δ × μ2.lat);
3  function GraphHash(G):
4      for v ∈ topo-order(G) do
5          x_v := hash(hash(v) ⊕ (⨁_{u∈G.pre(v)} x_u));
6      return hash(Σ_{v∈G} x_v);
7  μ_best :=InitState(G); X := ∅;
8  Q :=PriorityQueue({μ_best}, BetterThan);
9  while Q ≠ ∅ do
10     μ := Q.pop(); x :=GraphHash(μ.G);
11     if x ∈ X then continue;
12     X := X ∪ {x};
13     if μ's F-Tree needs update then
14         μ :=Analyze(μ, L); # Algorithm 1
15     for μ' ∈ApplyTransformRules(μ) do
16         μ' :=ApplyIncrementalSchedule(μ'); # Algorithm 2
17         if BetterThan(μ', μ_best) then μ_best := μ' ;
18         if BetterThan(μ', μ_best, 1.1) then Q.push(μ') ;
19 return μ_best;
```

v a suitable dividing point for topological ordering problem. Specifically, all the nodes that v depends on should be scheduled before v, and all the nodes that are dependent on v should be scheduled after v, providing a natural partition of the scheduling problem. Also, after we find the optimal schedules separately for $G.\mathtt{anc}(v)$ and $G.\mathtt{des}(v)$, the peak memory consumption is guaranteed to be less than $M_{opt} + \sum_{v\in\mathcal{V}(G)\setminus G.\mathtt{anc}(v)\setminus G.\mathtt{des}(v)} |v|$, where M_{opt} represents the peak memory achieved under the optimal scheduling of G. If $\mathtt{nw}(v) = 0$, then the scheduling problem for the graph can be divided into two completely independent subproblems at v. We design a heuristic algorithm based on the NW value to select interval whose boundary NW values are as small as possible (line 2-6), where the constants 20, 10, 4 are empirical hyper-parameters which perform well in practical. The idea behind GraphPartition is to use nodes with $\mathtt{nw}(v) \leq 1$ as dividing points to partition each connected component of the given graph into multiple sub-graphs.

6.2 Top-level Search Algorithm

MAGIS adopts a greedy search algorithm to optimize graphs. There are two modes of optimization supported by MAGIS: optimizing latency given memory limit or optimizing memory given latency limit. Algorithm 3 shows the search algorithm for the former mode.

The inputs of Algorithm 3 consist of a graph G, a given memory limit M, and F-Tree max-level L. We first schedule and analyze the given graph to obtain an initial M-State (line 9). Then we construct a priority queue for storing M-State (line 10) where the priority is determined by the BetterThan function (line 1-4) that compares latency first when both M-States satisfy the memory limit M; otherwise, it compares memory (note that we compare $(a, b) < (c, d)$ with lexicographical order). We then iteratively pop an M-State μ (line 12) and apply M-Rules to generate a series of new M-State (line 17). The Analyze function (line 16) will update the F-Tree in M-State μ if its previously mutated sub-graph is not influenced by F-Trans. We perform incremental scheduling on the newly generated M-State μ'. Then we will push μ' to queue if it's not worse than μ_{best} in a relaxed condition (controlled by a small coefficient δ, empirically set to 1.1). To prevent redundant search, we borrow the idea of Weisfeiler-Lehman Test [48] to hash a given graph (line 5-8, line 12-14), where ⊕ means bytes concatenation operation.

To reduce the overhead of performance measurement, we implement a simulator with an operator performance cache. It saves the actual execution latency of operators, and uses a simulation approach to obtain the overall performance and memory usage of the whole graph with a schedule. When considering asynchronous swapping, re-ordering involving Store/Load operators can also slightly affect latency. To address this, our re-ordering strategy is to place the Store as early as possible and place the Load as late as the data transfer latency can be just hidden.

Table 2. Workloads for Evaluation

Name	Batch	Other Configuration
ResNet-50 [19]	64	image-size=224
BERT-base [12]	32	sequence-length=512
ViT-base [15]	64	image-size=224, patch-size=16
U-Net [45]	32	image-size=256
U-Net++ [73]	16	image-size=256
GPT-Neo-1.3B [6]	32	sequence-length=512
BTLM-3B [13]	32	sequence-length=512

7 Evaluation

7.1 Experiment Setup

We use rustworkx [52] to implement MAGIS's graph data structure. We implement a code generation backend to generate Python code calling PyTorch APIs based on the graph and schedule. We use PyTorch's CUDA Stream API to implement asynchronous Store and Load. The data is swapped between GPU memory and CPU memory. Although our current implementation targets NVIDIA GPU, our methods can be easily ported to other platforms.

Our main baselines for comparison are: (1) PyTorch [36]: unoptimized graphs are directly converted into PyTorch code after simple topo-order scheduling, acting as the baseline for memory usage and execution latency. Note that basic memory saving are applied for this baseline, that is, future-unused tensors are deleted immediately. (2) POFO [5]: state-of-the-art work for memory optimization of networks with simple

structures and linearly connected cells, considering both re-materialization and swapping. We use the open-sourced implementation of POFO [4]. (3) DTR [27]: state-of-the-art work using re-materialization technology for memory optimization of arbitrary networks. We use the implementation of DTR in MegEngine [1] (its eager mode and PyTorch both call cuBLAS & cuDNN for computation on NVIDIA GPUs with the same performance). (4) XLA [46]: state-of-the-art DNN compiler using a greedy re-materialization algorithm for memory optimization. (5) TVM [9] (Relay [43]): state-of-the-art DNN compiler, performing basic memory saving to reclaim future-unused tensors. (6) Torch-Inductor [2] (TI): state-of-the-art DNN compiler leveraging OpenAI Triton [50], performing basic memory saving to recycle tensors that are no longer used in the future.

Table 2 shows the workloads we use for evaluation. We select the training processes of the following networks as experiment workloads: (1) Classic CNN classification network: ResNet [19], with linear inter-cell connection and simple intra-cell structure. (2) Classic transformer networks: BERT [12] and ViT [15], with linear inter-cell connection and complicated intra-cell structure. (3) Image segmentation networks with long skip-connections: U-Net [45] and U-Net++ [73], with complicated inter-cell connections (U-Net++ is even more complex than U-Net) and simple intra-cell structure. (4) Large language models: GPT-Neo-1.3B [6] and BTLM-3B [13], with much larger weights and deeper structures compared with classic transformer networks. Note that the workloads diversely span from language models to vision models, from large models to small models. The data type is `bf16` for GPT-Neo & BTLM, and `tf32` for others.

The platform we use for our experiments is an Intel workstation equipped with 20 Intel(R) Xeon(R) Silver 4210R CPUs, an NVIDIA GeForce RTX 3090 GPU, CUDA version 11.6, cuDNN version 8.4.0, PyTorch version 2.1.0, MegEngine version 1.12.3, TensorFlow version 2.15.0, and TVM version 0.14.0. The max-level parameter L of Algorithm 3 is 4 by default. For every optimization process, we run MAGIS with a time budget of 3 minutes. For each baseline, we first use TASO rules (mainly the A-Trans rules which merge operations like the QKV-projections in a transformer-block into a single operation and split the result later) to optimize the network to ensure a fair comparison. We measure the peak memory usage of the optimization results of MAGIS, PyTorch, POFO, and TI via `torch.cuda.max_memory_allocated`; for DTR, we use `megengine.get_max_allocated_memory`; for XLA, we use `tf.config.experimental.get_memory_info`; for TVM, we hack the memory allocation information of its memory planner. Note that, since baseline PyTorch cannot run the workload settings of GPT-Neo and BTLM in the experiment platform because of out-of-memory, we measure its latency and peak memory using MAGIS's simulator.

[4]https://gitlab.inria.fr/hiepacs/rotor/-/tree/offload

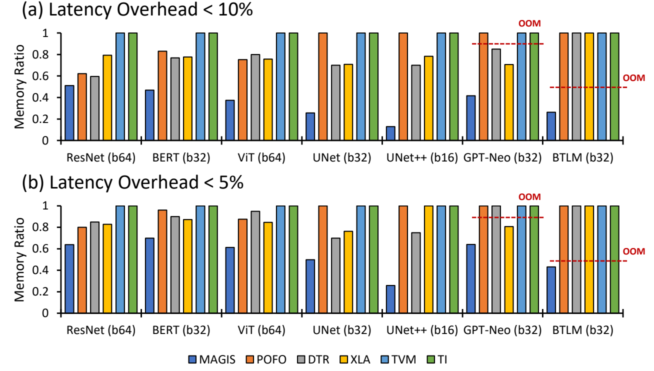


Figure 9. Peak memory ratio compared to un-optimized PyTorch (lower is better). "OOM" means the memory usage exceeds the memory limit of our experiment platform.

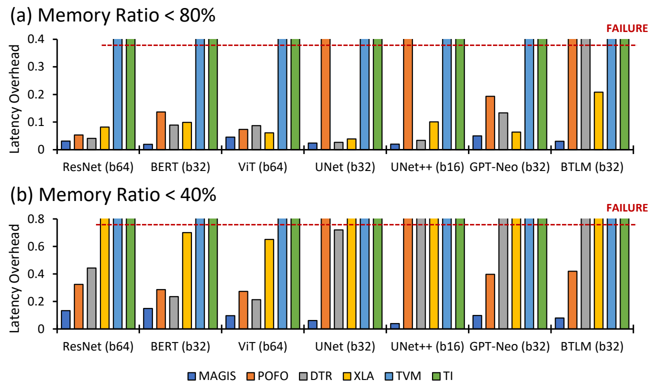


Figure 10. Latency overhead compared to PyTorch without optimization (lower is better). "FAILURE" means the memory ratio cannot be optimized to meet the constraint.

7.2 Experiment Results

7.2.1 Memory Optimization with Latency Constraint.

We first evaluate the memory optimization effects of MAGIS and baselines under 10% and 5% latency overhead constraints. Results are shown in Figure 9. With 10% latency overhead limit, MAGIS optimizes peak memory to 15%~60% of PyTorch's, outperforming other baselines (60% at best). TVM and TI only perform basic memory saving like the PyTorch baseline, so their optimized memory ratios are near to 100%. MAGIS's memory is 15%~80% of POFO's, 20%~85% of DTR's, and 15%~70% of XLA's. At 5% latency overhead limit, MAGIS optimizes peak memory to 25%~70% of PyTorch's, 25%~80% of POFO's, 35%~80% of DTR's, and 25%~80% of XLA's.

For ResNet, when the latency overhead limit is 10% (5%), MAGIS's peak memory is around 80%~85% (75%~80%) of POFO & DTR & XLA. MAGIS's results are closed to baselines', mainly because ResNet has a simple structure, and the benefits brought by our methods are limited.

For BERT and ViT, MAGIS achieves 50%~70% (65%~75%) of the baselines' memory at 10% (5%) latency overhead constraint. The relative results of MAGIS are better than ResNet

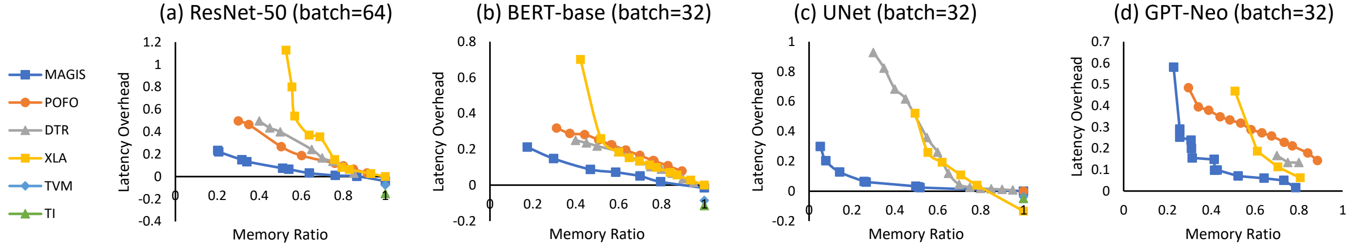


Figure 11. Latency & memory curves of MAGIS and baselines. MAGIS can achieve Pareto optimal in almost all cases.

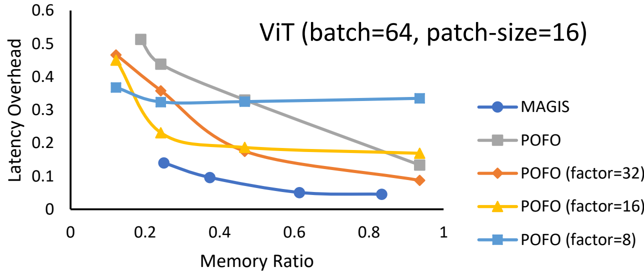


Figure 12. Comparing MAGIS with POFO. The network used by POFO has been pre-processed with micro-batching (with different factors).

due to more intra-cell complexity of transformer networks. MAGIS performs better on ViT than on BERT due to shorter sequence length of ViT. Sequence length has a larger impact on latency than on peak memory, making it more challenging to optimize the memory under a given latency constraint with longer sequence. For UNet and UNet++, MAGIS achieves 15%∼35% (25%∼70%) of baselines' memory at 10% (5%) latency overhead constraint. MAGIS performs better on these two networks compared to other networks due to more complex inter-cell structures which provide more optimization space for graph transformation. For GPT-Neo and BTLM, MAGIS maintains ≤40% (≤60%) of PyTorch's memory at 10% (5%) latency overhead limit. Only XLA avoids OOM among baselines for GPT-Neo. All baselines are OOM for BTLM under both constraints.

7.2.2 Latency Optimization with Memory Constraint.

We then conduct experiments to compare the latency optimization effects of MAGIS and other works under 80% and 40% peak memory limits of un-optimized PyTorch. Results are shown in Figure 10. TVM & TI cannot optimize all the workloads into 80% memory ratio. POFO almost cannot optimize UNet & UNet++. DTR's processes for UNet++, GPT-Neo, and BTLM take too long with a 40% memory limit, and XLA also cannot optimize these workloads under such constraints. We mark these cases as "FAILURE" in the figure. With an 80% limit, MAGIS reduces latency overhead to ≤5%, better than POFO (≤40% for BTLM, ≤20% for others), DTR (≤15%), and XLA (≤20%). At 40% memory limit, MAGIS maintains ≤15% overhead, while POFO stays at ≤40%, DTR reaches ≤45% for ResNet/BERT/ViT and ≤70% for UNet, and XLA caps at ≤70%. Similar to the previous experiments, MAGIS performs the best on UNet/UNet++, followed by ResNet/BERT/ViT, achieving a 1.25× speedup over DTR for UNet under the 40% memory limit. Among the baselines, only POFO can optimize GPT-Neo and BTLM under the 40% memory limit, but with much higher latency overhead than MAGIS. Note that although TASO rules bring some peak memory overhead, the overhead is small (around 5% on average) since these rules only enlarge local memory footprint. The baselines that fail to meet the memory constraints in Figure 10 still fail without applying TASO rules.

7.2.3 Trade-off Curves of Latency & Memory.

We plot the memory & latency trade-off curves in Figure 11. Note that XLA, TVM, and TI may achieve lower latency than the PyTorch baseline when there's no memory constraint, resulting in points below the horizontal line. When memory ratio is 1, MAGIS is faster than PyTorch but slower than XLA, TVM, and TI, due to MAGIS currently not implementing the sophisticated compilation optimizations like operator fusion as these compilers do. From the results, it can be observed that MAGIS's curve remains mostly below the baselines' curve. This indicates that we have achieved a better Pareto boundary in the dual-objective optimization of memory and latency, which means that, given a latency constraint, MAGIS achieves lower memory consumption, or given a memory constraint, it achieves lower latency.

We observe XLA's curve is nearly linear but experiences substantial latency overhead under low memory limits, since when memory limit is tight, re-computing one operator might depend on another operator re-materialization. The re-materialization used by DTR is better than XLA's greedy algorithm, enabling a near-linear trade-off between memory and latency even under tight memory limits. POFO's curve is also near-linear as it also adopts swapping, which balances memory and latency in a near-linear ratio. When the memory constraint it not tight, MAGIS's curve is near-linear with a slope lower than baselines since it also employs graph transformations such as F-Trans to balance memory and latency. However, under strict memory limits, MAGIS's curve becomes increasingly steep because even F-Trans incurs large overhead to optimize memory within tight constraint, caused by poor locality of on-chip memories due to small operators split from F-Trans.

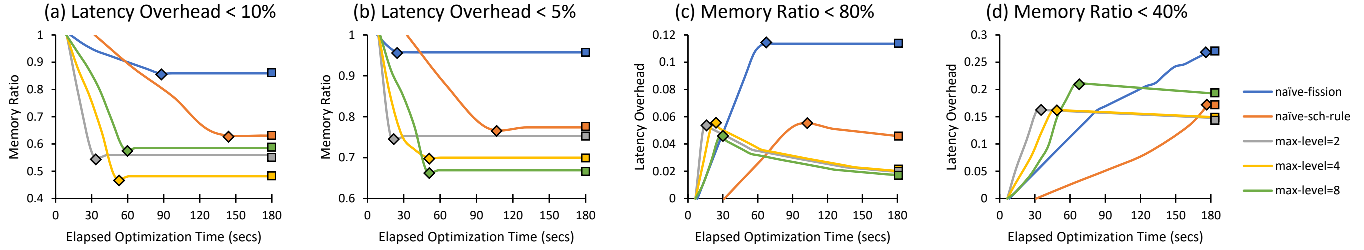


Figure 13. Heuristic breakdown of MAGIS when optimizing BERT workload in 3 minutes with the constraints used in §7.2.1 and §7.2.2. The diamond "◇" in a curve is the time point after which its optimization result meets the constraint. The square "□" in a curve is the time point with the best optimization result.

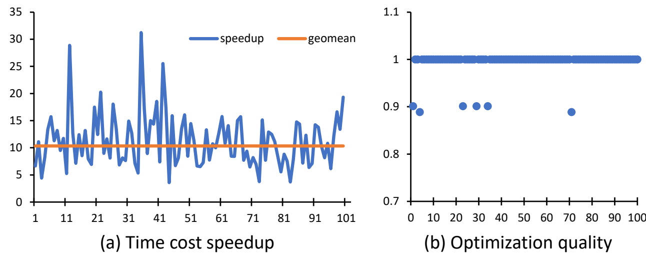


Figure 14. Comparison between incremental scheduling (IS) and full scheduling (FS). **(a)** Speedup of IS's scheduling time relative to FS's scheduling time. **(b)** Schedule result quality of IS compared to FS.

	Total	Trans.	Sched.	Simul.	Hash	Filtered	Others
Count	/	7148	924	924	7148	6224	/
Cost (secs)	60	2.52	3.70	8.71	44.82	/	0.25

Figure 15. Optimization time cost breakdown of MAGIS when optimizing ViT (batch 64) in 1 minutes. "Filtered" means the duplicated graphs filtered by hash test.

7.2.4 Comparison to Micro-batching. We examine the effect of graph transformation on memory optimization by integrating it into baselines. We select ViT as workload and POFO as target baseline. We focus on F-Trans due to its substantial memory impact. We apply micro-batching to ViT, dividing the whole graph (factors: 32, 16, 8) along batch-dimension to simulate a simple F-Trans. The split sub-graph are fed to POFO, and execution latency is multiplied by the sub-graph count.

Figure 12 reveals that graph transformation optimization enhances POFO's performance under stringent memory constraints. Under different memory limits, POFO performs best with different factors. This indicates that there are different trade-off spaces between graph transformation and scheduling. MAGIS outperforms both optimized and original POFO due to better coordinating transformation and scheduling.

7.2.5 Heuristic Ablation. The heuristics used in MAGIS main include: **H1**) F-Tree construction (Algorithm 1) discussed in §4.3; **H2)** heuristic used for schedule-based rules mentioned in §5.2; **H3)** hyper-parameter L (Algorithm 1 and 3) to control the max-level of F-Tree. We conduct a breakdown experiment with five settings: ① **naïve-fission**: disabling **H1** by randomly selecting valid sub-graph & dimension for F-Trans; ② **naïve-sch-rules**: disabling **H2** by matching schedule-based rules on the whole graph; ③/④/⑤ **max-level=2/4/8**: setting hyper-parameter L as 2/4/8 (default is $L = 4$ for other settings). We evaluate them on BERT workload under constraints used in §7.2.1 and §7.2.2 with a time budget of 3 minutes. Figure 13 depicts the curves of their elapsed optimization time and the historical best results during searching.

naïve-fission performs the worst due to limited F-Trans optimization of memory, causing up to 70% and 45% higher peak memory consumption and 10%-12% higher latency overhead than the best setting. **naïve-sch-rule** outperforms **naïve-fission** due to enabling **H1**. But it lags behind others since it disables **H2**, slowing down search convergence and making it challenging to find better results within the time budget. Settings excluding **naïve-fission** and **naïve-sch-rule** generally yield better outcomes, with **max-level=4** exhibiting the best overall performance. **max-level=2** restricts the F-Trans search space due to shorter F-Tree, thereby reducing optimization potential. Conversely, **max-level=8** expands the search space, slowing search and making optimization more difficult; in (d), **max-level=8** is even inferior to **naïve-sch-rule** (with $L = 4$).

7.2.6 Optimization Time. Figure 15 illustrates the time costs of different processes in a 1-minute ViT (batch 64) training optimization using MAGIS. Processes include Transformation ("Trans."), Scheduling ("Sched."), Simulation ("Simul."), Hash Test ("Hash"). "Filtered" indicates the number of graphs filtered out after hash test. "Trans." contributes a minor 2.52s overhead, while "Sched." stands at 3.7s. "Simul.", necessitated by operator performance data collection, exhibits the highest average overhead at 8.71s. "Hash" incurs the highest total overhead (44.82s), mainly due to filtering duplicate graphs, effectively reducing other processes' overhead.

7.3 Evaluation of Incremental Scheduling

We evaluate incremental scheduling (IS) against full scheduling (FS) in terms of speed for 10 randomly generated DNNs

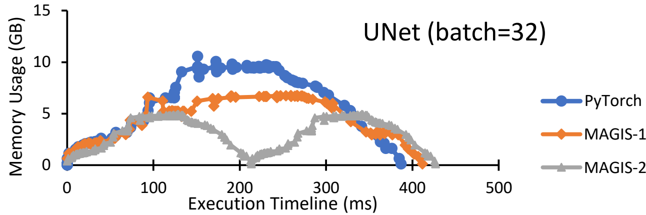


Figure 16. Execution time & memory usage for UNet.

with structures resembling NASNet [75]. Using TASO's [25] graph transformation rules, we conduct 100 rounds of transformations (10 rounds per DNN) after an initial scheduling. Both IS and FS employ the DP algorithm from [3] (`DpSchedule` in Algorithm 2). Figure 14 (a) illustrates IS's speed advantage over FS, achieving a speedup of $4 \sim 30\times$ ($10\times$ in average) across 100 tests. Figure 14 (b) presents the optimization quality of IS, measured as the ratio of peak memory usage optimized by IS to that optimized by FS. In 94 out of 100 tests, IS attains the same level of optimality as FS.

7.4 Case Study

We use UNet as a case study to demonstrate the optimization effect. Figure 16 depicts UNet's training time & memory with PyTorch, MAGIS-1 (memory limited at 80% of PyTorch's peak), and MAGIS-2 (limited at 60%). Both PyTorch and MAGIS-1 display initial memory increase followed by decrease due to activation saving during forward phase and activation releasing during backward phase. MAGIS-1 has lower peak memory thanks to re-materialization, swapping, and F-Trans, but incurs higher latency. MAGIS-2 exhibits dual memory peaks, caused by a F-Trans covering the whole graph. This reduces peak memory further compared to MAGIS-1, yet increases latency overhead.

8 Related Work

In this section, we briefly introduce the related work of MAGIS, mainly including techniques of graph scheduling (re-materialization, swapping, and re-ordering) as well as graph transformation. Some other DNN compilers are also discussed in this section.

Re-materilization evicts some intermediate tensors and re-computing them later when needed. It was first applied in deep learning by [10, 17, 18]. Graph-theoretic analysis is used in [28, 29]. Checkmate [24] uses Integer Programming (IP) for optimization. DTR [27] uses heuristic strategies to optimize re-mat. of dynamic graphs. MONeT [47] co-optimize re-mat. and operator implementations.

Swapping stores some tensors on external storage and reloads them later when needed. vDNN [42], Capuchin [38], and SuperNeurons [57] use it for DNN training on GPUs. SwapAdvisor [22] co-optimizes re-ordering, memory allocation, and swapping. TFLMS [30] represents swapping by special operators and control-flow edges. POET [37] uses IP to combine re-mat. and swapping for training on mobile devices. POFO [5] uses Dynamic Programming (DP) to combine re-mat. and swapping. ZeRO-Offload [41] combines swapping with distributed training. AutoTM [20] and ZeRO-Infinity [39] use persistent memory as external storage.

Re-ordering finds proper topo-order of DNNs to optimize memory. Serenity [3] uses DP for optimization. SwapAdvisor [22] considers both re-ordering and swapping. HMCOS [58] hierarchically searches optimal ordering. Zhong et al. [72] use IP with variable pruning to speedup optimization.

Graph Transformation originates from compiler's super optimization [34]. It gradually optimizes the graph with a sub-graph mutated at each step. MetaFlow [26] uses backtracking algorithm, and TenSAT [62] employs equality saturation [60] for searching. TASO [25] generates transformation rules automatically based on program synthesis. PET [56] proposed partial equivalent transformation. Unity [54] integrates distributed parallel optimization into graph transformation. Turner et al. [53] combine graph transformation with neural architecture search. Compared to previous work, MAGIS can trade the latency and memory optimization. Regarding transformation types, MAGIS investigates the formalization and search for fission transformation. We also propose the re-materialization and swapping rules derived from graph scheduling, enhancing the coordination between graph transformation and scheduling.

Other DNN Compilers. Besides the works mentioned above, many other DNN compilers have been proposed [2, 8, 9, 11, 14, 16, 31, 33, 43, 46, 49, 50, 59, 61, 63–71, 74] in recent years. For example, AutoTVM [11], FlexTensor [70], Ansor [64], and Roller [74] automatically generate/explore tuning space of a single operator or a small sub-graph; UNIT [59], AMOS [66], and TensorIR [16] automatically map operators onto hardware accelerators with specialized tensor instructions; Rammer [33], HFuse [31], and IOS [14] fuse parallel operators to increase hardware utilization; DNNFusion [35], AStitch [71], and Apollo [63] fuse chained operators to reduce data movement; BOLT [61], Chimera [69], SET [8], TileFlow [67], and Welder [49] additionally explore fusion space for compute-intensive operators.

9 Conclusion

We propose MAGIS, a DNN optimizer for memory & latency with a systematic design of fission transformation effective coordination between graph transformation and scheduling. Experimental results show that compared to state-of-the-art methods, MAGIS only uses 15% $\sim$ 85% memory with same latency constraint and obtains a better memory & latency Pareto boundary.

Acknowledgments

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

References

[1] Megengine: A fast, scalable and easy-to-use deep learning framework. https://github.com/MegEngine/MegEngine, 2020.

[2] torch.compiler - PyTorch 2.1 documentation. https://pytorch.org/docs/2.1/torch.compiler, 2023.

[3] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. *MLSys*, 2:44–57, 2020.

[4] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools.* pearson Education, 2007.

[5] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *NIPS*, volume 34, pages 23844–23857, 2021.

[6] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, 2021.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda Askell. Language models are few-shot learners. *NIPS*, 33:1877–1901, 2020.

[8] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. In *ISCA*, pages 1–17, 2023.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, pages 578–594, 2018.

[10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *NIPS*, 31, 2018.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL*, pages 4171–4186, 2019.

[13] Nolan Dey, Daria Soboleva, Faisal Al-Khateeb, Bowen Yang, Ribhu Pathria, Hemant Khachane, Shaheer Muhammad, Zhiming, Chen, Robert Myers, Jacob Robert Steeves, Natalia Vassilieva, Marvin Tom, and Joel Hestness. Btlm-3b-8k: 7b parameter performance in a 3b parameter model, 2023.

[14] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. *MLSys*, 3:167–180, 2021.

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, and Sylvain Gelly. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2020.

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *ASPLOS*, pages 804–817, 2023.

[17] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *TOMS*, 26(1):19–45, 2000.

[18] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *NIPS*, 29, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[20] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *ASPLOS*, pages 875–890, 2020.

[21] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. In *NIPS*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020.

[22] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *ASPLOS*, pages 1341–1355, 2020.

[23] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *CVPR*, 2018.

[24] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. *MLSys*, 2020.

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, pages 47–62, 2019.

[26] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. *MLSys*, 1:27–39, 2019.

[27] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization. *ICLR*, 2021.

[28] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient Rematerialization for Deep Networks. *NIPS*, 32, 2019.

[29] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *NIPS*, volume 32, 2019.

[30] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting, 2019.

[31] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic Horizontal Fusion for GPU Kernels. In *CGO*, 2020.

[32] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In *ATC*, pages 161–175, 2021.

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with {rTasks}. In *OSDI*, pages 881–897, 2020.

[34] Henry Massalin. Superoptimizer: a look at the smallest program. *ASPLOS*, 15(5):122–126, 1987.

[35] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *PLDI*, pages 883–898, 2021.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS*, 2019.

[37] Shishir G. Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph E. Gonzalez. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *ICML*, 2022.

[38] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *ASPLOS*, pages 891–905, 2020.

[39] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme

scale deep learning. In *SC*, pages 1–14, 2021.

[40] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *ATC*, pages 551–564, 2021.

[42] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*, pages 1–13, 2016.

[43] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level compiler for deep learning, 2019.

[44] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis With Latent Diffusion Models. In *CVPR*, pages 10684–10695, 2022.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, pages 234–241, 2015.

[46] Amit Sabne. Xla: Compiling machine learning for peak performance. 2020.

[47] Aashaka Shah, Chao-Yuan Wu, Jayashree Mohan, Vijay Chidambaram, and Philipp Kraehenbuehl. Memory Optimization for Deep Networks. In *ICLR*, 2022.

[48] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *JMLR*, 12(9), 2011.

[49] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *OSDI*, 2023.

[50] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *MAPL*, pages 10–19, 2019.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[52] Matthew Treinish, Ivan Carvalho, Georgios Tsilimigkounakis, and Nahum Sá. rustworkx: A high-performance graph library for python. *JOSS*, 7(79):3968, 2022.

[53] Jack Turner, Elliot J. Crowley, and Michael O'Boyle. Neural Architecture Search as Program Transformation Exploration. *ASPLOS*, 2021.

[54] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, and Jamaludin Mohd-Yusof. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *OSDI*, pages 267–284, 2022.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[56] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *OSDI*, pages 37–54, 2021.

[57] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In *PPoPP*, pages 41–53, 2018.

[58] Zihan Wang, Chengcheng Wan, Yuting Chen, Ziyi Lin, He Jiang, and Lei Qiao. Hierarchical memory-constrained operator scheduling of neural architecture search networks. In *DAC*, pages 493–498. Association for Computing Machinery, July 2022.

[59] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. UNIT: Unifying Tensorized Instruction Compilation. In *CGO*, pages 77–89, 2021.

[60] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and Extensible Equality Saturation. *POPL*, 5:1–29, 2021.

[61] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. *MLSys*, 4, April 2022.

[62] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *MLSys*, 2021.

[63] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic Partition-based Operator fusion through Layer by Layer Optimization. *MLSys*, 4, 2022.

[64] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, pages 863–879, 2020.

[65] Size Zheng, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training. *TPDS*, 33(11):3220–3232, 2021.

[66] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA*, pages 874–887, 2022.

[67] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis. In *MICRO*, 2023.

[68] Size Zheng, Siyuan Chen, and Yun Liang. Memory and Computation Coordinated Mapping of DNNs onto Complex Heterogeneous SoC. In *DAC*, pages 1–6, 2023.

[69] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion. In *HPCA*, pages 1113–1126, 2023.

[70] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS*, pages 859–873, 2020.

[71] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS*, 2022.

[72] Shuzhang Zhong, Meng Li, Yun Liang, Runsheng Wang, and Ru Huang. Memory-aware Scheduling for Complex Wired Networks with Iterative Graph Optimization. In *ICCAD*, 2023.

[73] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. In *DLMIA*, pages 3–11, 2018.

[74] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. {ROLLER}: Fast and Efficient Tensor Compilation for Deep Learning. In *OSDI*, pages 233–248, 2022.

[75] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2018.