H-Rockmate: Hierarchical Approach for Efficient Re-materialization of Large Neural Networks

Julia Gusak, Xunyi Zhao, Théotime Le Hellard, Zhe Li, Lionel Eyraud-Dubois, Olivier Beaumont

H-Rockmate: Hierarchical Approach for Efficient Re-materialization of Large Neural Networks

Julia Gusak
Inria Center at the University of Bordeaux
yulia.gusak@inria.fr

Xunyi Zhao
Inria Center at the University of Bordeaux
xunyi.zhao@inria.fr

Théotime Le Hellard
École Normale Supérieure, Paris
theotime.le.hellard@ens.pl.eu

Zhe Li
Inria Center at the University of Bordeaux
zhe.a.li@inria.fr

Lionel Eyraud-Dubois
Inria Center at the University of Bordeaux
lionel.eyraud-dubois@inria.fr

Olivier Beaumont
Inria Center at the University of Bordeaux
olivier.beaumont@inria.fr

Abstract

Training modern neural networks poses a significant memory challenge, as storing intermediate results during the forward and backward passes demands substantial memory resources. To address this issue while maintaining model accuracy, re-materialization techniques have been introduced to recompute selected intermediate results rather than storing them, thereby adhering to peak memory constraints. The main algorithmic problem is to compute a re-materialization schedule that minimizes the computational overhead within a given memory budget. Our H-Rockmate framework builds upon existing Rockmate solution and overcomes its limitation to work with sequential block structures by proposing a *hierarchical* approach. The framework performs an automatic decomposition of the data-flow graph into a hierarchy of small-scale subgraphs, and finds a re-materialization schedule for the whole graph by recursively solving optimization problems for each subgraph. H-Rockmate allows users to transform their PyTorch models into nn.Modules that execute forward and backward passes efficiently within the specified memory budget. This framework can handle neural networks with diverse data-flow graph structures, including U-Nets and encoder-decoder Transformers. H-Rockmate consistently outperforms existing re-materialization approaches both in terms of average training iteration time and peak memory trade-offs, demonstrating superior memory efficiency in training modern neural networks.

Preprint. Under review.

1 Introduction

Modern Neural Networks (NN) undergo several important evolutions which have consequences on the computation and memory requirements, from the first vision networks like ResNet-50 [Wu et al., 2019] to Natural Language Processing transformer-based models [Vaswani et al., 2017] like GPT. On the one hand, the size of the models and the resolution of the data are increasing, which raises problems for the storage of both weights and activations. On the other hand, the first models had chain-like structures (sequence of convolution layers) and then moved to chains of complex blocks (chains of Transformer blocks for GPT-like models). Finally, recent NN exhibit arbitrarily complex dependency graph structures between layers of the neural network: for example, UNO Wen et al. [2022] is structured as a U-Net of complex blocks, and encoder-decoder transformers Vaswani et al. [2017] feature very long skip connections.

Re-materialization is a well known and efficient technique to limit the memory requirements related to activations during training. The idea is to avoid storing all the necessary activations because of the subsequent dependencies during the Forward phase and the Backward phase (those related to the Stochastic Gradient Descent mechanism). Some activations are computed and then deleted to make room in memory, and they will have to be re-computed later when needed. In the case of simple chains [Beaumont et al., 2019a] and in the case of chains of complex blocks [Zhao et al., 2023], this technique has shown its efficiency: it is often possible to save 50% of memory for a computational overhead of about 10 to 15%.

From a theoretical point of view, the problem is to minimize the computational time required to execute the forward and backward passes under a predefined memory budget. It has been proven in Naumann [2008] that obtaining an optimal re-materialization schedule is NP-Hard in the case of general dependency graphs represented as general data-flow graphs. Some solutions such as TW-Remat [Kumar et al., 2019] or Checkmate [Jain et al., 2020] have nevertheless been designed to deal with the case of general graphs, but both suffer from a number of limitations, as discussed in Section 2. Other approaches like Rotor [Beaumont et al., 2019a] and Rockmate [Zhao et al., 2023] have proposed to find efficient solutions for limited classes of dependency graphs, where a chain structure (of potentially complex blocks) can be identified.

The work presented here is at the convergence of these research lines, with an original approach based on a **hierarchical decomposition** of the computation graph, to find a re-materialization strategy for **any graph of dependencies** between layers. In the case where the graph is too large to be addressed directly by an approach based on Integer Linear Programming, we propose to decompose it into a graph of complex blocks, with potentially even more than two levels in the hierarchy. Efficient solutions for different memory budgets are generated for each of the blocks at the bottom of the hierarchy, using different approaches from the literature. Then, we provide a **new** Integer Linear Programming (ILP) formulation to **efficiently recombine** these low-level solutions into candidate solutions for the higher levels of the hierarchy.

This paper contains the following contributions:

- A data-flow graph decomposition algorithm **H-Partition** that builds a hierarchy of blocks of reasonable sizes (to keep an acceptable computational complexity) while minimizing the memory size of the interfaces between the blocks
- A new linear programming solver H-Ilp adapted to this hierarchical decomposition.
- A general **framework** to integrate previous and future re-materialization strategies at any level of the hierarchy, combining their strengths.

Thanks to the H-Ilp solver, H-Rockmate achieves high efficiency in terms of solution quality, while having a sufficiently fast solving time to be used on the larger models used in practice today. H-Rockmate is fully compatible with the `autograd` mechanism of PyTorch, so that no modification of the code is required to use it. With a single line, the user can automatically control the memory usage of their neural network: `model = HRockmate(model, sample, memory_budget)`. The H-Rockmate framework will be distributed as open-source software upon acceptance of the paper.

The rest of the paper is organized as follows. In Section 2, we review the related work on memory saving strategies for DNN training. H-Rockmate is presented in Section 3 which covers graph decomposition, partial problems resolution and global re-materialization strategy reconstruction. Section 4 demonstrates the efficiency of the proposed method on a large number of networks, and provides an evaluation of the computational overhead induced by re-materialization. Finally, concluding remarks are proposed in Section 5.

2 Related Work

In the rest of the related work, we mostly cover the contributions that rely on the exclusive use of re-materialization. Memory footprint of activations might also be reduced via data parallelism [Das et al., 2016, Zhang et al., 2013] (by distributing mini-batches across several computational resources) and offloading [Rhu et al., 2016, Wang et al., 2018] (which offloads and later retrieves activations from GPU memory to CPU memory). Model parallelism [Huang et al., 2019, Narayanan et al., 2019], in particular in its pipelined version, can be used to minimize the memory requirements by storing weights across several devices. The optimal combination of all these strategies with re-materialization, discussed by Beaumont et al. [2021] in the context of sequential models, is left for future work.

TW-Remat [Kumar et al., 2019], based on a tree-width decomposition of the dependency graph, was initially designed for the case where all computational costs and activation sizes have a unitary weight. A greedy heuristic was later proposed to generalize the algorithm to the cases of non-unitary weights, but without guarantee. Kusumoto et al. [2019] proposed an optimal dynamic programming algorithm restricted to a special class of solutions, where each node is computed at most twice. The XLA framework[1] contains a re-materialization feature based on a greedy heuristic described in Kumar et al. [2019], and shown to be less efficient than TW-Remat. Checkmate [Jain et al., 2020] is based on the solution of an Integer Linear Program (ILP), and computes a generic optimal solution under a set of reasonable assumptions, but the computation cost becomes prohibitive as soon as the number of nodes in the network exceeds 70-90 nodes.

Other approaches have been proposed to find efficient solutions for limited classes of dependency graphs. Beaumont et al. [2019b] relies on dynamic programming in the case of networks whose graph is a sequence of basic operations, covering ResNet-like networks (considering one ResNet block as one operation). This extends in a proven framework the heuristic techniques proposed by Chen et al. [2016]. Then, the case of graphs for which the forward graph is a sequence of complex blocks has been addressed in the Rockmate framework in Zhao et al. [2023], which covers most Transformer-based networks (with one Transformer block considered as a complex block). One of the main limitations of Rockmate is that for architectures with long skip connections (like U-Net [Ronneberger et al., 2015] or encoder-decoder Transformer [Vaswani et al., 2017]), it degenerates to Checkmate and inherits its difficulties to handle large data-flow graphs.

In this paper, we design **an algorithm and a framework to address any kind of network with a static data-flow graph**, with reasonable solving time on networks with up to 2000 nodes. In the experimental evaluation of Section 4, we compare H-Rockmate with the state-of-the-art strategies: Checkmate, TW-Remat, and Rockmate. Table 1 summarizes the comparison of these re-materialization techniques, and emphasizes the benefits of H-Rockmate compared to all previous approaches.

3 H-Rockmate

We consider the *data-flow graph* associated to a neural network where each node is a computation at the tensor level. A *schedule* represents a sequence of basic computations and tensor deletions that performs the forward and backward passes for one training iteration. The optimization problem related to re-materialization is to find a schedule whose peak memory is below a given budget and with the smallest possible execution time.

[1] `http://www.tensorflow.org/xla`

	Approach	Limitations w.r.t. H-Rockmate
Rotor [Beaumont et al., 2019a]	Dyn. Prog.	Chain structure only
TW-Remat [Kumar et al., 2019]	BTD +Greedy	No guarantee on solution quality
Checkmate [Jain et al., 2020]	ILP	Long solving time
Rockmate [Zhao et al., 2023]	Dyn. Prog. + ILP	Solving time is not controlled for general structures

Table 1: Comparison of Different Re-Materialization Strategies, Dyn. Prog. stands for Dynamic Programming, ILP for Integer Linear Programming, BTD for Bounded Tree-width Decomposition.

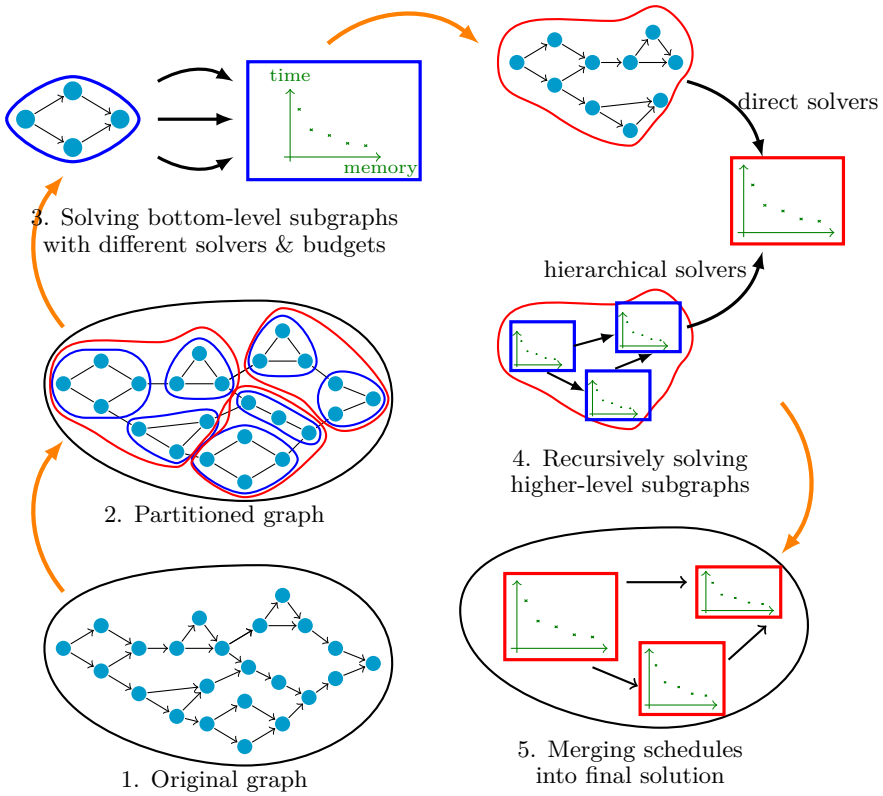


Figure 1: Visualization of the different steps in the H-Rockmate framework.

Figure 1 describe the main steps of H-Rockmate approach. In a first step, H-Rockmate uses the graph building tool developed in Rockmate [Zhao et al., 2023] to obtain the data-flow graph of the input module. In a second step, the **H-Partition** package, described in Section 3.1, is used to recursively partition the graph into subgraphs of tractable sizes. This partitioning is performed recursively in order to ensure that the higher-level graphs are also of reasonable size. Steps 3, 4 and 5 of Figure 1 describe the **H-Solver** algorithm, detailed in Section 3.2.1. Starting with the lowest level of the decomposition, the algorithm computes schedules for each subgraph with different memory budgets, so as to explore different time-memory trade-offs, providing several *options* for the nodes of the higher levels. This procedure continues until the top-level graph, solved for a single memory budget corresponding to the available memory for activations.

3.1 H-Partition

The goal of the partitioning step of H-Rockmate is to reduce the size of the problems that need to be solved, without affecting too much the quality of the overall solution. The result

of this step is a decomposition in a hierarchy of subgraphs, where the nodes of graphs at a given level represent an entire subgraph of the level below. The subgraph sizes are limited by two main parameters: M^l denotes the maximum number of nodes in a lower-level subgraph, and M^t denotes the maximum number of nodes in the top-level graph. Since it is beneficial to allow a larger solving time for the top-level graph, we use $M^t \geq M^l$. Our partitioning algorithm is a greedy heuristic described in Algorithm 1. Each iteration has three main steps: formation of *candidate* groups, selection of a best candidate according to our evaluation criterion, merging of the selected candidate and update of the remaining candidates. Each of these steps is described below.

Input: data-flow graph G
Result: a recursive partition of G
Parameters: max high-level size M^t, max lower-level size M^l, score parameter α
while *G has more than M^t nodes* **do**
  $\mathcal{C} \leftarrow \bigcup_{x \in G}$ candidate group containing all nodes between x and $a(x)$;
  while *$\mathcal{C}$ is non empty and G has more than M^t nodes* **do**
    Select candidate C which minimizes s_α (eq. 1);
    Wrap the nodes of C into a group;
    Update $\mathcal{C}$;
  end
  Consider all groups as subgraphs;
  Update G so that each subgraph is considered as a node;
end
return *partitioned graph*

Algorithm 1: H-Partition algorithm

Formation of candidate groups For any node x in G, we consider $a(x)$, the closest common ancestor to all the direct predecessors of x[2]. We create four candidate groups with all nodes on all paths from $a(x)$ to x, depending on whether x and/or $a(x)$ are included. Any candidate group with more than M^l nodes is discarded.

Candidate selection When selecting the best group among all candidates, our objective is to avoid incurring a too high memory pressure when the group is used as a subgraph. The memory pressure depends directly on the sizes of the input and output values. It also depends, less directly, on the length of the schedule during which they will be alive, which we evaluate through the number of original nodes in the subgraph. We use the following score function $s(C)$ for a candidate C:

$$s_\alpha(C) = \left(\sum_{x \text{ input or output value of } C} \text{memory size of } x \right) \cdot (\#\text{ of original nodes in } C)^\alpha \,, \quad (1)$$

where α is a hyper parameter, whose default value is 0.5.

Update of candidates Once the best candidate C is chosen, it becomes a group: all its nodes will be considered together from now on. However, it is not yet a subgraph: if it is not too large, it might be merged with other groups later in the same phase in order to reduce the number of groups. The remaining candidates are updated: in any candidate C' whose intersection with C is nonempty, we add all the other nodes of C to ensure that all nodes of C remain together. If this union contains more than M^l nodes, this candidate C' is no longer acceptable and is removed.

The Appendix contains a proof that this procedure always results in a valid decomposition, in the sense that no subgraph contains a directed cycle. After the partitioning, H-Rockmate also identifies subgraphs that correspond to the execution of the same code, as is done in Rockmate. This avoids solving several times the same optimization problem.

[2] a common global ancestor is added in case G has several inputs

3.2 H-Solver

3.2.1 Solving framework

The idea of hierarchical re-materialization is that re-materialization schedules can be computed for each subgraph independently, with several memory budgets. These different re-materialization schedules for a given subgraph are called *options*, and each corresponds to a different time vs memory trade-off. Once all subgraphs at a given level have been solved, a schedule for the next level can be computed with our proposed H-ILP. H-ILP is a **linear programming** formulation that provides an optimal schedule within a given memory budget. It can be used on a subgraph of general structure, but using it on large subgraphs can result in unreasonable solving time. It is thus applied only to subgraphs with sufficiently low number of nodes. This algorithm is inspired by RK-CHECKMATE [Zhao et al., 2023], which itself is an extension of CHECKMATE [Jain et al., 2020]. The general idea of extending this linear programming approach to a hierarchical structure is a major contribution of the present paper and is described in detail in Section 3.2.2.

On significantly large graphs, partitioning with only two levels would result in the top-level graph being still too large to be solved with this technique. Our hierarchical approach makes it possible to compute schedules for the levels from bottom to top, handling graphs of arbitrary size, while each sub-problem remains of tractable size.

The H-ILP solver that we propose is very generic and efficient, and it provides very good quality solutions on all kinds of neural networks. Thanks to the genericity and extendability of the H-ROCKMATE framework, it is also possible to combine existing approaches, which further improves the quality of the solutions. The H-ROCKMATE framework currently contains two additional algorithms. First, H-TWREMAT is a wrapper around the TW-REMAT implementation [Shepperd, 2021] of a heuristic based on a **treewidth decomposition** approach [Kumar et al., 2019]. This wrapper and the H-ROCKMATE framework allows this heuristic to be used with PyTorch, whereas it was previously only available for TensorFlow. Second, RK-ROTOR is the **dynamic programming** algorithm from ROCKMATE [Zhao et al., 2023] that also provides an optimal schedule. It is limited to subgraphs whose forward phase has a sequential structure (where each computing node only depends on the previous one), but its solving time is low.

3.2.2 Hierarchical ILP formulation

This section presents H-ILP, the hierarchical adaptation of the RK-CHECKMATE linear programming formulation. We are given an arbitrary graph H, where each compute node represents a subgraph, and where dependencies are carried by *data nodes*, which represent values that can be saved in memory. The H-ILP formulation computes the schedule with minimum running time whose peak memory remains below a specified memory budget.

The actual linear programming formulation of H-ILP follows the same structure as the formulation of RK-CHECKMATE: the schedule is divided into *phases*, and the goal of phase t is to compute node t for the first time. For conciseness, we describe here only the main modifications that allow H-ILP to be used in a hierarchical setting, but we refer the reader to the Appendix for a complete description of the H-ILP formulation.

Options and phantom nodes The novelty of H-ILP compared to RK-CHECKMATE is that each compute node can represent a subgraph of the original graph. Such a compute node can be computed with one of several *options*. Each of these options represents a possible schedule for the forward and backward phases of the subgraph. There is a strong link between the forward and the respective backward computations, and each backward computation should be performed with the same option as its corresponding forward computation. The H-ILP formulation contains additional variables and constraints to specify which option for each compute node is used within each phase.

In H-ILP, we also introduce an explicit representation of the data saved in memory between a forward computation and its corresponding backward. We call them *phantom nodes*, and we update the formulation by considering them as special data nodes, with two specificities. First, a phantom node is always created by its forward computation, always deleted after its

backward computation, and is not required by any other computing node. In the formulation, we take advantage of this by not including additional variables expressing whether the phantom node is deleted or not. Second, the values saved in a phantom node (and thus the associated memory size) depend on the option used for the forward and backward computations. For this reason, the formulation contains additional variables that specify which option of each phantom node exists in memory during each phase.

Correction terms for memory usage In ROCKMATE, to achieve good performance, the input values of a block are deleted as soon as they are no longer needed, instead of being kept until the end of the block. This is possible due to the sequential structure, where each block is the only user of its input values. In the more general context of the present paper, a subgraph might share its input values with other subgraphs. To obtain good performance, H-ILP allows sub schedules to delete their input values and the gradients of the outputs; but if another subgraph needs these values, the higher level algorithm adapts the sub schedules. This decision has an impact on the memory peak of corresponding subgraphs.

We have added constraints to the formulation to ensure that the memory peak is correctly evaluated in all cases. These constraints are only introduced when solving the top-level graph since they make the ILP harder to solve, and only the top-level graph solution needs to be accurately bounded by GPU memory. For each computation node k, the number of these constraints for node k is bounded in the worst case by $\min(l_{k,o}, 2^{\text{degree of node } k})$, where $l_{k,o}$ is the number of operations of the schedule of option o for node k. In practice, this number of constraints remains low enough that the H-ILP formulation is solved in reasonable time.

Schedule selection The number of binary variables in the H-ILP formulation depends linearly on the total number of options of all nodes. To avoid wasting resources when several very similar options are available for a given node, we include in H-ILP a hyper-parameter N_o, which provides a limit on the total number of options. To stay within this limit, we may have to select some options among all the schedules generated at the lower level.

To do so, we first allocate a number of options to each node, based on the total N_o: each node will receive a number of options proportional to the number of basic operations inside the corresponding subgraph. Then, we greedily select the schedules whose memory peaks are further from each other: starting from the schedule with the highest memory peak, then the one with lowest memory peak, then the one closest to the middle, and so on.

3.3 Complexity Analysis of H-Rockmate

The complexity of H-ROCKMATE depends mostly on the number N of nodes in the graph. Obtaining the data-flow graph with RK-GB has a complexity $O(N)$ and is very fast in practice. With our current implementation, recursively partitioning the graph with H-Partition has a complexity $O(N^2 \log N)$. This step is also quick for graph sizes up to $N = 1000$, but handling very large graphs would require more work on the graph algorithms: recursively partitioning a graph of size $N = 10^5$ requires 2 hours on an Intel Xeon Gold processor.

The most time-consuming step is the computation of re-materialization schedules with H-ILP. Thanks to our hierarchical approach, this step actually has a linear complexity. Indeed, the hierarchical decomposition has logarithmic depth, and the total number of subgraphs is $O(N/M^t)$. Solving one subgraph of size M^l for a number O of budget options only requires solving O Integer Linear Programs, whose sizes and solving times do not depend on N. In addition, all ILPs at the same level are completely independent and can be performed in parallel. As detailed in the Appendix, even without parallelization, our current implementation is able to handle the largest versions of neural networks used in practice: the forward-backward graphs of GPT with 96 layers and Transformer with 36 encoders and decoders have 2500 computation nodes and are solved respectively in 15 and 150 minutes.

4 Experimental Evaluation

Experimental settings For all models, we measure the peak memory usage and the execution time required for a single training iteration. We conduct a warm-up phase consisting

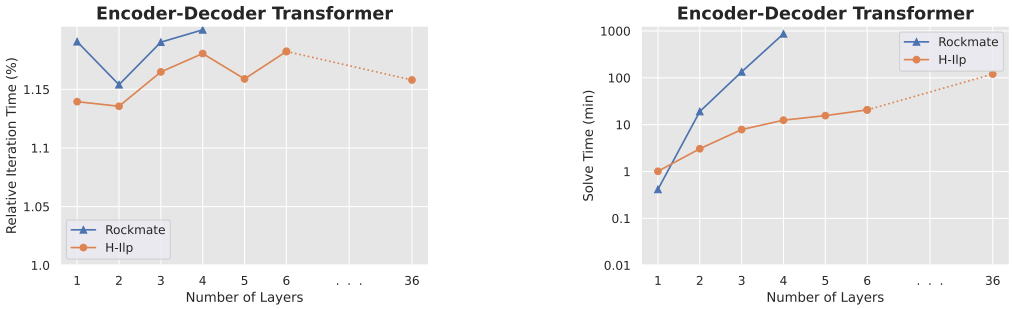


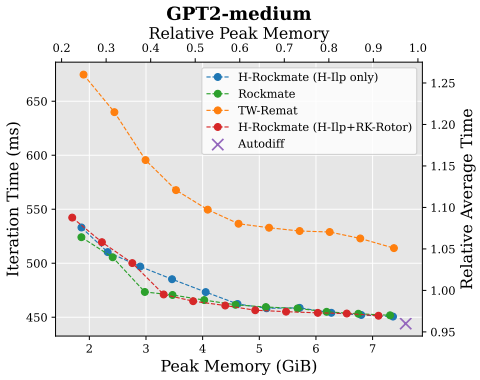
Figure 2: Experiments for varying graph size: (left) iteration time (right) solving time

of five initial runs; the subsequent ten runs are used to evaluate the peak memory and computational time, providing reliable estimates of the performance. Measurements show that the standard error of the iteration time is at least two orders of magnitude smaller than the mean. As a result, error bars are not depicted in the plots. Experiments use an NVIDIA Quadro RTX8000 GPU with 48GB of memory and an NVIDIA V100 GPU with 16GB of memory, with PyTorch 2.0.1, CUDA 11.6, and Gurobi 9.5.0. Additional experiments for more architectures and more ablation studies are available in the Appendix.

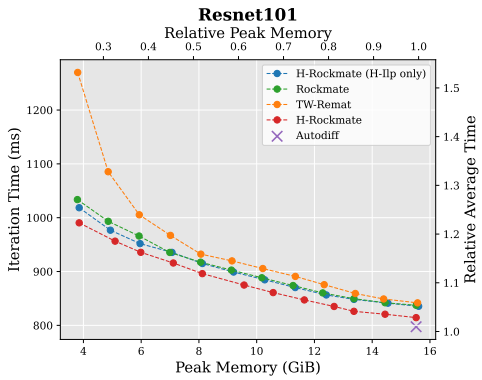
Efficiency In Figure 2, we compare the solving time and performance of Rockmate and H-Rockmate. Since solving an ILP has exponential complexity, it is important to limit the size of the problem. A clear drawback of Rockmate is that it needs to see the graph as a sequence of blocks, which for general neural networks means that some blocks are very large. In Encoder-Decoder Transformer Vaswani et al. [2017], there is no clear sequential structure for Rockmate to partition the graph, and this leads to very large solving times as shown on the right of Figure 2. Specifically, Rockmate takes 15 hours to obtain a solution for a 4-layer encoder-decoder structure transformer, while H-Rockmate achieves better results within 12 minutes. By controlling the total number of nodes in each graph with the H-Partition algorithm, H-Rockmate is able to efficiently limit the solving time without compromising solution quality.

Performance Figure 3 demonstrates that H-Ilp consistently outperforms TW-Remat in terms of iteration time for a given budget. On the Encoder-Decoder (Figure 3c), TW-Remat is able to find schedules for smaller memory budgets than H-Ilp, but for UNet the situation is reversed. By integrating TW-Remat, H-Rockmate effectively addresses this limitation of H-Ilp and provides efficient solutions over the complete range of memory budgets. Figures 3a and 3b show that H-Ilp obtains similar performance as Rockmate, a baseline approach specifically tailored for such architectures. Overall, H-Rockmate acts as a general solution that encompasses H-Ilp, Rotor, Rockmate, and Checkmate. Our experimental results consistently demonstrate that H-Rockmate performs on par with these baseline algorithms, showcasing its versatility and effectiveness in optimizing memory utilization for a wide range of network architectures.

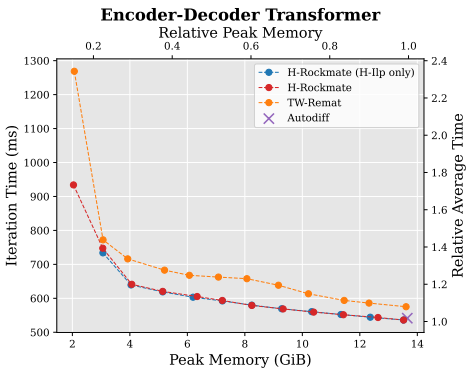
Optimality w.r.t. partitioning Without graph partitioning, H-Ilp is equivalent to rk-Checkmate which provides an optimal solution given a predefined topological order of operations. While partitioning might exclude certain re-materialization solutions on the overall network, Table 2 shows that H-Ilp obtains performance close to this optimal solver. In another experiment, we control the depth of hierarchy in H-partition by limiting the size of each subgraph. The results in Figure 4 show that H-Ilp maintains similar results in terms of solution quality when the depth of the hierarchy increases. It suggests that the combination of H-Partition and H-Ilp can find near-optimal solutions even on larger data-flow graphs by reducing the search space of re-materialization solutions in a smart way.



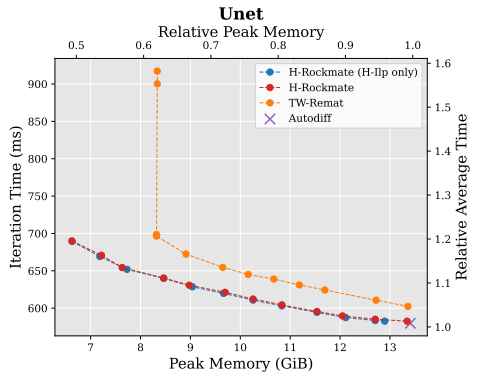
(a) GPT2-medium



(b) Resnet101



(c) Encoder-Decoder Transformer



(d) U-Net

Figure 3: Experiments on different network architectures: (top) sequential-like neural networks (bottom) non-sequential neural networks, which ROCKMATE cannot solve efficiently.

Budget (GB)	H-ILP	RK-CHECKMATE
0.8	4.714%	4.706%
0.9	3.597%	3.531%
1.0	2.480%	2.430%
1.1	1.369%	1.369%

(a) 2-layer Encoder-Decoder Transformer

Budget (GB)	H-ILP	RK-CHECKMATE
6.1	8.586%	8.010%
7.1	5.473%	5.462%
8.1	3.191%	2.988%
9.1	1.110%	1.040%

(b) U-Net

Table 2: Simulation results comparing H-ILP and RK-CHECKMATE: for different budgets we show the overhead in terms of iteration time compared to Autodiff.

5 Discussion and Conclusion

In conclusion, this paper introduces the H-ROCKMATE framework, which offers both theoretical and practical advancements in re-materialization for PyTorch models. The theoretical contributions include a hierarchical approach and an original linear programming formulation H-ILP for very efficient solutions without restrictive assumptions. Although H-ROCKMATE focus on computational graph with primitive operations, the hierarchical approach with linear complexity can be useful in other intense tasks targeting at tiling graphs. The framework also incorporates previous approaches, insuring state-of-the-art results. On the practical part, H-ROCKMATE seamlessly integrates into PyTorch, improving memory-time trade-off and efficiency, and is fully compatible with PyTorch Autograd. The main limitation of H-ROCKMATE is that it is not adapted to dynamic neural network architectures, where the structure of the computational graph depends on the input.

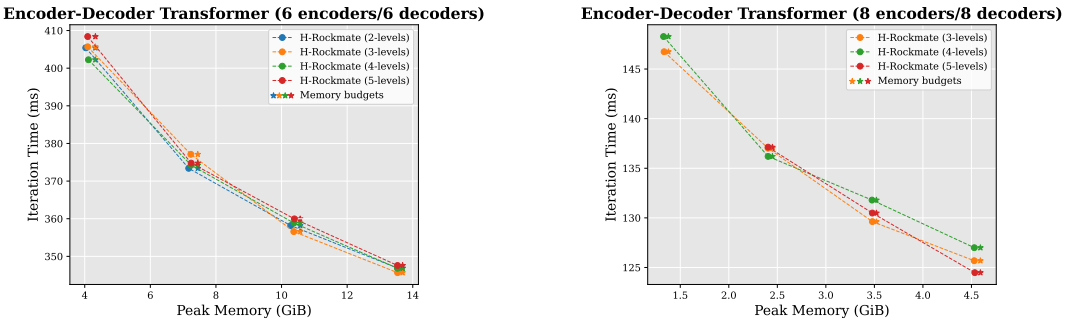


Figure 4: Experiments on 6-layer and 8-layer encoder-decoder Transformer with different depth of hierarchical structure.

Future research can explore how to integrate offloading into the optimization problem to reduce the need for re-computations, and also how to combine re-materialization with pipelined model parallelism in an optimized way. Advancements in these areas hold promise for enhancing performance and efficiency in deep learning systems.

References

O. Beaumont, L. Eyraud-Dubois, J. Hermann, A. Joly, and A. Shilova. Rotor, 2019a. URL `https://gitlab.inria.fr/hiepacs/rotor`.

O. Beaumont, L. Eyraud-Dubois, J. Hermann, A. Joly, and A. Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory, 2019b. URL `https://arxiv.org/abs/1911.13214`.

O. Beaumont, L. Eyraud-Dubois, and A. Shilova. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34: 23844–23857, 2021.

T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

R. Kumar, M. Purohit, Z. Svitkina, E. Vee, and J. Wang. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019.

M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

U. Naumann. Call tree reversal is np-complete. In *Advances in automatic differentiation*, pages 13–22. Springer, 2008.

M. A. Rahman, Z. E. Ross, and K. Azizzadenesheli. U-no: U-shaped neural operators. *arXiv preprint arXiv:2204.11127*, 2022.

M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 18. IEEE Press, 2016.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.

N. Shepperd. Fine tuning on custom datasets, 2021. URL `https://github.com/nshepperd/gpt-2/tree/finetuning/twremat`.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. *SIGPLAN Not.*, 53 (1):41–53, Feb. 2018. ISSN 0362-1340. doi: 10.1145/3200691.3178491.

G. Wen, Z. Li, K. Azizzadenesheli, A. Anandkumar, and S. M. Benson. U-fno—an enhanced fourier neural operator-based deep-learning model for multiphase flow. *Advances in Water Resources*, 163:104180, 2022.

Z. Wu, C. Shen, and A. Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019.

S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663. IEEE, 2013.

X. Zhao, T. Le Hellard, L. Eyraud-Dubois, J. Gusak, and O. Beaumont. Rockmate: an Efficient, Fast, Automatic and Generic Tool for Re-materialization in PyTorch. In *ICML 2023*, Honolulu (HI), United States, July 2023. URL `https://hal.science/hal-04095305`.

H-Rockmate: Hierarchical Approach for Efficient Re-materialization of Large Neural Networks

Appendix

A H-Partition algorithm

Input: data-flow graph G
Result: a recursive partition of G
Parameters: max high-level size M^t, max lower-level size M^l, score parameter α

while *G has more than M^t nodes* **do**
  $\mathcal{C} \leftarrow \bigcup_{x \in G}$ candidate group containing all nodes between x and $a(x)$;
  while *$\mathcal{C}$ is non empty and G has more than M^t nodes* **do**
    Select candidate C which minimizes s_α (eq. 1);
    Wrap the nodes of C into a group;
    Update $\mathcal{C}$;
  end
  Consider all groups as subgraphs;
  Update G so that each subgraph is considered as a node;
end
return *partitioned graph*

Algorithm 2: H-Partition algorithm

In this section, we analyze the H-Partition algorithm, whose description is reproduced in Algorithm 2. We prove that the partition computed by this algorithm is always valid, in the sense that the resulting subgraphs do not contain any cycle. When merging a subgraph into a node for the higher level, all edges entering or exiting a vertex of the subgraph are attached to the resulting node. To ensure that this does not result in a cycle, we guarantee that all of the subgraphs are *convex* in the graph theoretic sense, as defined below.

Convexity We first provide some graph notations. Given two nodes a and b, we write $a \to b$ if there is a direct edge from a to b, and $a \rightsquigarrow b$ if there is a path of any length from a to b. Paths of length 0 are also valid, so that $a \rightsquigarrow a$ is always true. With these notations, we can define the *convexity* of a subgraph:

Definition 1. *A subgraph C of a graph G is* convex *if for any two elements a, b in C, C contains all nodes of G on any path from a to b. This can be written as:*

$$\forall a, b \in C, \forall u \in G, (a \rightsquigarrow u \text{ and } u \to b) \Rightarrow u \in C$$

Merging a convex subgraph C into a node n does not create new cycles into the graph: such a new cycle would be a path starting at n and going back to n, going through another node $u \notin C$. If C is convex, any path from a node of C to another node of C only goes through nodes of C, which ensures the absence of cycles.

Candidate groups The candidate groups C_x formed on line **??** of Algorithm 2 contain all nodes on all paths from $a(x)$ to x, where $a(x)$ is the common ancestor to all direct predecessors of x. They have the following property, where $h(C_x) = a(x)$:

Property 1. *A subset C of nodes is a* valid *candidate group, if and only if there exists a* head *$h(C)$ such that:*

$$\text{If } u \in C \text{ and } v \to u, \quad \text{then } v \in C \text{ or } v = h(C) \tag{2}$$
$$\text{If } u \in C, \quad \text{then } h(C) \rightsquigarrow u \tag{3}$$

This property ensure their convexity:

Lemma 1. *Any candidate group C which satisfy Property 1 is convex.*

Proof. Consider a and b in C, and u in G such that $a \rightsquigarrow u$ and $u \to b$. There are two cases:

- If $u \neq h(C)$, then since $b \in C$ and $u \to b$, according to (2) we have $u \in C$.
- If $u = h(C)$, then since $a \in C$, by (3), we have $u \rightsquigarrow a$. Since G is acyclic, this implies $u = a \in C$.

□

Update of candidates Once the best candidate C is chosen, it becomes a group: all its nodes will be considered together from now on. The remaining candidates are updated: in any candidate C' whose intersection with C is nonempty, we add all the other nodes of C to ensure that all nodes of C remain together. The following results show that the resulting set of nodes is still a valid candidate group; in particular it is also convex.

Lemma 2. *If C and C' are valid candidate groups with $C \cap C' \neq \emptyset$, then $h(C) \rightsquigarrow h(C')$ or $h(C') \rightsquigarrow h(C)$. Furthermore, if $h(C) \neq h(C')$, then the first case implies $h(C') \in C$ and the second case implies $h(C) \in C'$.*

Proof. Let $u \in C \cap C'$, and consider $v \in G$ such that $v \to u$. If no such v exists, then u is the source of G and $u = h(C) = h(C')$. If $v \in C \cap C'$, we can start over with $u = v$.

We now have $u \in C \cap C'$, and $v \notin C \cap C'$ with $v \to u$. We have three cases:

- If $v \in C$ and $v \notin C'$: from (2) applied to C', we have $v = h(C') \in C$, and from (3) applied to C we get $h(C) \rightsquigarrow h(C')$.
- Symmetrically, if $v \notin C$ and $v \in C'$, we get $h(C') \rightsquigarrow h(C)$.
- If $v \notin C$ and $v \notin C'$: from (2) applied to both C and C', we get $v = h(C) = h(C')$.

□

Theorem 1. *If C and C' are valid candidate groups with $C \cap C' \neq \emptyset$, then $D = C \cup C'$ is a valid candidate group.*

Proof. From Lemma 2, we know that $h(C) \rightsquigarrow h(C')$ or $h(C') \rightsquigarrow h(C)$. We define the head of D as $h(D) = h(C)$ in the first case, and $h(D) = h(C')$ otherwise. For simplicity, we assume in the following that $h(C) \rightsquigarrow h(C')$; the other case is symmetrical.

It is clear that D satisfies (3): consider any $u \in D$. If $u \in C$, then $h(D) = h(C) \rightsquigarrow u$ by (3) applied to C. If $u \in C'$, then $h(D) \rightsquigarrow h(C')$ by assumption and $h(C') \rightsquigarrow u$ by (3), so that in both cases $h(D) \rightsquigarrow u$.

We now show that D satisfies (2). Let $u \in D$ and $v \to u$ with $v \notin D$. We distinguish two cases:

- If $u \in C'$, then since $v \notin C'$, by (2) applied to C' we get $v = h(C')$; and since $v \notin C$, the contrapositive of Lemma 2 yields $h(C') = h(C) = h(D)$. Thus $v = h(D)$.
- If $u \in C$, then $v \notin C$ and (2) applied to C yield directly $v = h(C) = h(D)$.

□

This completes the validity proof of Algorithm 2: all candidate groups in $\mathcal{C}$ satisfy Property 1 all along the execution of the algorithm, both when they are created (line **??**) and when they are updated (line **??**). This implies that all subgraphs created line **??** are convex.

Identification of identical subgraphs To further improve efficiency of H-Rockmate, we rely on and improve an idea from Rockmate: once the graph is partitioned, we identify all identical subgraphs that correspond to the execution of exactly the same code on values with the same shape[3]. A schedule computed for one of these identical subgraphs can be used for any of them, that significantly reduces the solving time on networks with a large number of identical blocks, such as GPT. This is performed in a more efficient way than in Rockmate, by expressing each graph in a canonical way and by relying on a hash table.

B H-Ilp hierarchical formulation

In this section, we present H-Ilp, the hierarchical adaptation of the Checkmate linear programming formulation.

B.1 Context

We assume that we have an arbitrary graph H, where each compute node represents a subgraph. Like in rk-Checkmate, dependencies are carried by *data nodes*, that represent *values* that can be saved in memory. A value is said to be *alive* at some time in a schedule if it is stored in memory at that time. The memory usage at a given time in a schedule is the sum of the memory sizes of all values alive at that time, and the *peak memory* of a schedule is the largest memory usage over the length of the schedule. The H-Ilp formulation computes the schedule with minimum running time whose peak memory remains below a specified memory budget B. We denote by T the number of compute nodes, by I the number of data nodes. Compute nodes are numbered in a topological order.

B.1.1 Options and phantom nodes

The novelty of H-Ilp compared to rk-Checkmate is that each compute node can represent a subgraph of the original graph. Such a compute node can be computed with one of several options. Each of these options represents a possible schedule for the forward and backward phases of the subgraph. There is a strong link between the forward and the respective backward computations, and each backward computation should be performed with the same option as its corresponding forward computation. A pair of a forward and the corresponding backward nodes are called a *layer*. For a layer j, its forward and backward compute nodes are denoted F_j and B_j respectively.

In H-Ilp, we also introduce an explicit representation of the data saved in memory between a forward computation and its corresponding backward. We call them *phantom nodes*, and we update the formulation by considering them as special data nodes, with two specificities:

- a phantom node is always created by its forward computation, can only be deleted by its backward computation, and is not required by any other computing node. In the formulation, we can take advantage of this by not including additional variables expressing whether the phantom node is deleted or not.
- the values saved in a phantom node (and thus the associated memory size) depend on the option used for the forward and backward computations. For this reason, the formulation contains additional variables that specify which option of each phantom node exists in memory during each phase.

In this formulation, we consider schedules in which for a given phantom node, only one option is present in memory at a given time. However, it is possible that a phantom node is produced several times with different options during the course of the schedule.

B.1.2 Note about input dependencies

Option-specific dependencies An output value of the forward computation (i.e., a data node which is computed during forward and used by another compute node) is never included

[3]This does not require to solve the difficult graph isomorphism problem, since we can order nodes according to their execution in the original code.

in the phantom node. However, it happens that an output value is also used within the forward computation to produce other results. An example could be:

```
def compute(a):
  x = f(a)
  y = g(x)
  return x, y
```

In this example, the value `x` is both an output of the layer and used to produce `y`. In that case, the backward schedule might choose either to use `x` as input to be able to perform the backward of `g()` (if having it in memory between forward and backward fits in the budget), or to recompute it during backward. The implication is that for a given layer, each option leads to specific dependencies for the backward compute node, depending on which inputs is used by the corresponding schedule. If option o of a computation node k depends on value d, we denote this as $d \xrightarrow{o} k$.

Multiple predecessors Just like in RK-CHECKMATE, a data node can have several predecessors. This happens in backward when computing gradients: each computation is a *contribution* to the same memory slot (gradients are accumulated). A successor of such a data node can only be processed if all its contributions have been computed.

B.2 Formulation

Like in the RK-CHECKMATE formulation, the schedule is divided into T phases. The goal of phase t is to compute node t for the first time. In the following, we denote compute nodes with index k, data nodes with index d, options with index o, phases with index t and layers (a pair of forward and corresponding backward nodes) with index j.

$\mathcal{F}$ is the set of final data nodes. The graph contains a specific *loss* node which represents the computations that takes place between the forward and backward passes of our graph. If G is the main highest-level graph, this represents the computations of the loss for the training; if G is any subgraph, this also contains other computations from the rest of the graph. The index of the loss node is l.

Variables The H-ILP formulation only contains binary variables, which can take the value 0 or 1.

$R^t_{k,o}$ is 1 if and only if node k is computed with option o during phase t.

P^t_d is 1 if and only if data node d is present in memory before phase t.

$S^t_{k,d}$ for k predecessor of d is 1 if and only if the contribution of compute node k has been included in data node d before phase t.

$Sp^t_{j,o}$ is 1 if and only if the phantom of layer j is saved with option o before phase t.

$C^t_{k,d}$ is 1 if and only if data node d is created when computing node k during phase t

$D^t_{k,d}$ is 1 if and only if data node d is deleted after computing node k during phase t

Objective The objective is to minimize the total running time, expressed as

$$\min \sum_{i,t,o} R^t_{i,o} * \text{time of computing option } o \text{ for node } i$$

B.2.1 Constraints for options

In addition to the constraints already present in RK-CHECKMATE, the H-ILP formulation contains constraints relative to the choice of options and the management of phantom nodes.

Namely:

$$\forall t, \forall k, \quad \sum_o R^t_{k,o} \leq 1 \qquad \text{at most one option for each computation} \quad (4)$$

$$\forall t, \forall j, \quad \sum_o Sp^t_{j,o} \leq 1 \qquad \text{only one option of a phantom is in memory} \quad (5)$$

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \leq Sp^t_{j,o} + R^t_{F_j,o} \qquad \text{a phantom node is only be created by its } F_j \quad (6)$$

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \geq Sp^t_{j,o} + R^t_{F_j,o} - R^t_{B_j,o} \qquad \text{a phantom node is only deleted by its } B_j \quad (7)$$

$$\forall t, \forall j, \forall o, \quad R^t_{B_j,o} \leq Sp^t_{j,o} + R^t_{F_j,o} \qquad \text{computing } B_j \text{ requires the phantom node} \quad (8)$$

B.2.2 Constraints common with rk-Checkmate

For reference, we also describe here the constraints of RK-CHECKMATE that are still present in H-ILP. For any compute node k and any phase t, we denote by $CR^t_k = \sum_o R^t_{k,o}$ the equivalent of the R variable of RK-CHECKMATE, which is equal to 1 if node k is computed during phase t (with any option).

Validity constraints

$\forall t, \forall k > t,$	$CR^t_k = 0$	Node $k > t$ can not be computed in phase t
$\forall t, \forall j$ s.t. $F_j > t, \forall o,$	$Sp^t_{j,o} = 0$	No phantom j from node $F_j > t$ is saved before phase t
$\forall k \to d, \forall t \leq k,$	$S^t_{k,d} = 0$	No result of node k can be saved in any phase before phase k
$\forall d, \forall t \leq \min\{k \mid k \to d\},$	$P^t_d = 0$	Data node d is not in memory before any of its predecessors
$\forall d \in \mathcal{F}, \forall k \to d,$	$S^T_{k,d} + CR^T_k = 1$	After the last phase, all *final* data nodes should be in memory
$\forall t,$	$CR^t_t = 1$	Node t is executed in phase t
	$\sum_t CR^t_l = 1$	The *loss* node is executed only once

Data dependencies

$\forall t, \forall k \to d,$	$S^t_{k,d} \leq P^t_d$	Data node d with at least one contribution k is alive
$\forall t < T, \forall k \to d,$	$S^{t+1}_{k,d} \leq S^t_{k,d} + CR^t_k$	New contribution k to node d only appears by being computed
$\forall t, \forall k \to d \to k',$	$CR^t_{k'} \leq CR^t_k + S^t_{k,d}$	Computing node k' requires all contributions k to input node d
$\forall t, \forall j, \forall o, \forall k \to d \xrightarrow{o} B_j$	$R^t_{B_j,o} \leq CR^t_k + S^t_{k,d}$	Option-specific dependencies

Alive status of values A computing node k is *related* to a data node d if $k \to d$ or $d \to k$. We denote this with $k \leftrightarrow d$. For a data node d, only computing nodes k that are related to d can affect its alive status. For any t, if k is related to d, we denote with $A^t_{k,d}$ the alive status of node d during phase t after computing node k (and also after performing all deletions mandated by variables D). In phase t after computing node k, a data node d is alive if it was stored before phase t or created in phase t before node k, and not deleted until then, so that we can write:

$$\forall t, \forall k \leftrightarrow d, \quad A^t_{k,d} \doteq P^t_d + \sum_{k' \to d, k' \leq k} C^t_{k',d} - \sum_{k' \leftrightarrow d, k' \leq k} D^t_{k',d}$$

In the above equation and in the following, we use $\doteq$ to denote an alias definition, so that $A^t_{k,d}$ can be replaced by the right-hand side in any constraint, whereas the $=$ sign is used to denote a constraint that is added to the formulation.

Constraints relative to liveness

$\forall t, \forall k \leftrightarrow d,$	$0 \leq A^t_{k,d} \leq 1$	Data node d is either alive or not
$\forall t, \forall k \rightarrow d,$	$A^t_{k,d} \geq CR^t_k - D^t_{k,d}$	d is alive if computed and not deleted
$\forall t, \forall k \rightarrow d,$	$C^t_{k,d} \leq CR^t_k$	value d can only be created by a node k that is really computed
$\forall t < T, \forall d,$	$P^{t+1}_d = A^t_{\max\{k \mid k \leftrightarrow d\},d}$	Value d is alive after phase t iff it is alive after its last related node k

One additional constraint states that a value d is deleted after computing node k in phase t if it is not used afterwards: neither by later computing nodes $k' > k$ in the same phase t, nor in the next phase $t+1$. This can be stated as:

$$\forall t, \forall k \leftrightarrow d, \quad D^t_{k,d} = 1 \text{ if and only if } CR^t_k = 1 \text{ and } P^{t+1}_k = 0 \text{ and } \sum_{d \rightarrow k', k' > k} CR^t_{k'} = 0$$

However, this constraint is not linear. It can be linearized in the same way as in the original CHECKMATE paper [Jain et al., 2020, Section 4.5]: if we denote by $h_{k,d} = 2 + |\{k' | d \rightarrow k', k' > k\}|$ the number of equalities in the above statement, it is equivalent to:

$$\forall t, \forall k \leftrightarrow d, \qquad D^t_{k,d} \geq CR^t_k - P^{t+1}_k - \sum_{d \rightarrow k', k' > k} CR^t_{k'}$$

$$\forall t, \forall k \leftrightarrow d, \quad h_{k,d}(1 - D^t_{k,d}) \geq 1 - CR^t_k + P^{t+1}_k + \sum_{d \rightarrow k', k' > k} CR^t_{k'}$$

Evaluating memory usage The memory budget constraints of RK-CHECKMATE are also updated to take into account the memory sizes of the phantom nodes where appropriate. For this purpose, we denote by U^t_k the memory usage after computing node k in phase t. For any value d, let s_d be the amount of memory required to store d; and for any layer j and option o, let $p_{j,o}$ be the amount of memory required to store option o of the phantom node of layer j.

Then U^t_k can be expressed as a linear combination of the formulation variables. Indeed, the memory usage before starting phase t is:

$$M_t \doteq \sum_d s_d \cdot P^t_d + \sum_{j,o} p_{j,o} \cdot Sp^t_{j,o}.$$

Then, the increment when computing a forward node $k = F_j$ is:

$$IF_k \doteq \sum_{k \rightarrow d} s_d \cdot C^t_{k,d} - \sum_{k \leftrightarrow d} s_d \cdot D^t_{k,d} + \sum_o p_{j,o} R^t_{k,o}.$$

When computing a backward node $k = B_j$, the increment is:

$$IB_k \doteq \sum_{k \rightarrow d} s_d \cdot C^t_{k,d} - \sum_{k \leftrightarrow d} s_d \cdot D^t_{k,d} - \sum_o p_{j,o} \left(R^t_{F_j,o} + Sp^t_{j,o} - Sp^{t+1}_{j,o} \right)$$

The expression within the parenthesis is equal to 1 if the phantom node j is deleted, and 0 otherwise. Indeed, since B_j is the only compute node that can use it, $Sp^{t+1}_{j,o} = 0$ means that phantom node j can be deleted right after B_j. Constraint (7) ensures that if $R^t_{B_j,o} = 0$, then the expression within the parenthesis is also 0.

Finally, we can express U^t_k iteratively (like in the CHECKMATE and RK-CHECKMATE formulations):

$$\begin{aligned} \forall t, \quad & U^t_0 \doteq M_t + IF_0 \\ \forall t, \forall k = F_j, \quad & U^t_k \doteq U^t_{k-1} + IF_k \\ \forall t, \forall k = B_j, \quad & U^t_k \doteq U^t_{k-1} + IB_k \end{aligned}$$

Memory budget constraints Thanks to the U_k^t definitions, we can express constraints to ensure that the memory usage is always within the memory budget B. If we detail a single step k of some phase t, it corresponds to: (a) allocating memory for the newly created values (according to $C_{k,d}^t$ variables), (b) computing node k, (c) freeing the memory of the deleted values (according to variables $D_{k,d}^t$). The highest memory usage in this sequence is during (b), but the memory usage U_k^t corresponds to after (c). In addition, the computation of node k with some option o might incur a memory overhead (by allocating temporary values), which we denote by $m_{k,o}$. In total, in RK-CHECKMATE, the memory budget constraints are written as:

$$\forall t, \forall k, \quad U_k^t + \sum_o m_{k,o} \cdot R_{k,o}^t + \sum_{k \leftrightarrow d} s_d \cdot D_{k,d}^t \leq B \tag{9}$$

B.2.3 Correction terms

However, in H-ILP each compute node k represents not a single basic computation, but a *sequence* of basic computations (defined by the corresponding schedule). After the H-ILP formulation is solved, the actual schedule is modified: the memory deallocations for the values freed in step (c) above are performed as early as possible, possibly during the schedule of node k (in the middle of step b).

This means that the memory overhead $m_{k,o}$ during the computation of option o of node k might depend on whether some values are alive before or after computing node k. For example, if the corresponding schedule deletes a value in the middle of computation, its memory overhead $m_{k,o}$ assumes that the deletion is delayed until the end of the schedule. If that value is actually not needed later in the higher-level schedule computed by H-ILP, it will be deleted within the schedule, which may or may not change the memory overhead.

In the following, we present how we modify the memory budget constraint to account for this kind of situation. Consider a specific phase t, and an option o (and thus a schedule) for node k. For simplicity of presentation, let us consider only inputs; the situation with outputs is similar and symmetric. Consider a sub-step i of the schedule. We compute the memory overhead at this sub-step as $m_{k,o}^i$, assuming that value deletion happens after the computation of this schedule of node k. We denote by $\mathcal{F}_i$ the set of values which are not used in the following sub-steps of that schedule. Within the schedule computed by H-ILP, values in $\mathcal{F}_i$ are deleted after sub-step i if and only if they are deleted after step k. Hence, the *actual* memory usage of sub-step i is $m_{k,o}^i - \sum_{d \in \mathcal{F}_i} s_d \cdot D_{k,d}^t$. The corresponding memory constraint is:

$$\forall t, \forall k, \forall o, \forall i, \quad U_k^t + m_{k,o}^i \cdot R_{k,o}^t + \sum_{k \leftrightarrow d} s_d \cdot D_{k,d}^t - \sum_{d \in \mathcal{F}_i} s_d \cdot D_{k,d}^t \leq B$$

We write one constraint for each option and each sub-step. Since all the correction terms are negative, and all $m_{k,o}$ are at least 0, if $R_{k,o}^t$ is 0, this constraint is weaker than (9). We write such a constraint for each sub-step of the schedule, and this provides a more precise assessment of the memory usage of the solution. The case of output values is the same, except that we care whether the output value is created during the computation of node k, which is represented with variable $C_{k,d}^t$.

An interesting remark is that it is not necessary to write one constraint for each sub-step: if the set of inputs not needed after sub-steps i and j are the same ($\mathcal{F}_i = \mathcal{F}_j$), we can keep only one of both constraints (the one with the larger memory usage $m_{k,o}^i$). The number of constraints is thus bounded by $\min(\text{number of sub-steps}, 2^{|\{\text{inputs}\}|+|\{\text{outputs}\}|})$. In practice, the number of different constraints remain low enough. In addition, these constraints are only introduced when solving the top-level graph, where the constraint to remain under budget B is required to be as accurate as possible.

C Implementation details of the H-Rockmate framework

We provide here some implementation details about our framework, and in particular on the differences compared to ROCKMATE. In addition to the partitioning and solving steps,

described in detail above, there are two other steps in H-ROCKMATE: the graph building procedure, and the execution process.

C.1 Graph builder

In H-ROCKMATE, we use the same RK-GB module as in ROCKMATE to obtain a data-flow graph from an arbitrary PyTorch `nn.Module`. This module has four steps:

1. obtain the forward *basic graph* with `torch.jit`, in which each node represents exactly one computational operation
2. obtain a *simplified* graph in which all operation which do not produce a new `Tensor` are merged with the operation that produced the involved `Tensor`. Typically, all `view` and `size` calls, and all in-place operations have this behavior.
3. building the *backward* graph, and
4. measuring the time and memory requirements of each operation.

In ROCKMATE, there is another step where the simplified forward graph is decomposed into a sequence of blocks (so that each block can be solved with RK-CHECKMATE, and the sequence can be solved with RK-ROTOR). Once all blocks are identified, ROCKMATE performs pairwise comparisons to identify the blocks with similar structure, so that RK-CHECKMATE is only applied once for all identical blocks.

In H-ROCKMATE, the H-Partition step is a replacement for this "sequence-building" step. In the following, we provide technical details about three parts of RK-GB which were adapted (and sometimes improved) for the H-ROCKMATE framework: the identification of identical subgraphs, the management of "soft" dependencies, and the dependencies on the inputs of the model.

C.1.1 Efficient identical graphs recognition

Instead of pairwise comparison as is done in ROCKMATE, in the H-ROCKMATE framework the identical subgraphs are identified by using a canonical representation and a hash table. As a reminder, in both cases, identifying identical graphs is made possible by the fact that we can rely on the order of execution of the code; and the main use case of this feature is to identify when the same code is executed several times in the model. RK-GB is designed to be fully deterministic, so that it provides the same result on the same executed code.

We provide now a description of the identical subgraph recognition in H-ROCKMATE:

- We first identify identical compute nodes from the original (not partitioned) graph. Two nodes are identical if they have the same code (modulo variable renaming) and if all inputs and parameters have the same characteristics (shape, type, ...). Each compute node is thus associated with an *equivalence class*.
- Subgraphs are determined by the list of nodes they contain. To obtain the canonical representation of a subgraph, we go through this list of nodes, in the topological ordering inferred from the code execution. Each node is represented by its equivalence class, and by the indices (in the subgraph) of their predecessor nodes. The canonical representation of the subgraph is the concatenation of the representation of all its nodes.
- The obtained representation is inserted in a hash table: if a subgraph with the same representation exists, they will be identified as identical. Only one of these identical subgraphs is solved within H-Solver; it is called the *representative*.
- H-ROCKMATE also features a "translation" function that renames variables appropriately to convert a schedule for the representative into a schedule for each of the subgraphs in the class.

This procedure can be performed in linear time, and is thus very efficient in practice.

C.1.2 Soft dependencies

Consider a situation where a computation node k' does not depend on the `Tensor` x computed by a previous node k, but it depends on the characteristics of x (typically, the result of a `size(x)` call). In the simplification process, the `size()` call is merged within the node k. But having node k' depend on node k would result in undesired behavior in a situation where k has been computed once and removed from memory: k would be recomputed to enable computing k', whereas k' only requires the size of k.

The solution introduced by Rockmate is to consider this particular situation a *soft dependency* during the simplification process. These soft dependencies are used to produce consistent topological orderings. After the simplification process, for solving the blocks with rk-Checkmate, all soft dependencies are removed from the graph. The dependency of k' on the size of x is ensured by the topological ordering: the first computation of k will take place before any computation of k'. Since the data carried by a soft dependency is not a *Tensor*, it does not occupy any memory space, and is never removed.

Soft dependencies with H-partition However, in H-Rockmate, the H-Partition algorithm has to take the soft dependencies into account for the convexity considerations. Without the soft dependency from k to k', the predecessor j of k might be placed in the same subgraph as k', but without k. This subgraph would be executed *before* the subgraph of k (since $j \to k$), which would break the soft dependency. For this reason, the soft dependencies are retained all through the H-Partition algorithm, and ignored only during the H-Solver part.

C.1.3 Dependencies on inputs of the model

The `Tensor` values provided as input to the model have to remain in memory throughout the whole execution, for any possible schedule. It is not useful to take into account their memory sizes, since they cannot be managed by H-Rockmate. In addition, if such an input value has `requires_grad=False`, then no backward computation is associated either. In H-Rockmate, unlike Rockmate, the corresponding dependency is ignored. This enables H-Rockmate to have a more efficient discovery of sequential sub-parts of the model.

Consider for example the `torch.nn.Transformer` model, with 6 encoder and 6 decoder layers. This model has two inputs, `src` used at the first encoder layer, and `tgt` used at the first decoder layer. Since Rockmate takes into account the dependency carried by the `tgt` value, it is unable to identify the sequential structure of the encoding part: there is a dependency from the input node to the first node of the decoder. As a result, Rockmate sees this model as one large block of 219 computing nodes.

In contrast, with this change in the H-Rockmate framework, the same model can be decomposed into 24 blocks, the biggest of which contains 131 nodes (corresponding to the decoder part). This is still too large to be solved directly with rk-Checkmate, but this change enables H-Rockmate to run the Rockmate algorithm on a `nn.Transformer` with 4 layers. Beyond the benefit for Rockmate, having fewer dependencies also provides more opportunities for partitioning in the H-Partition algorithm.

C.2 Execution of the schedule

Once a schedule has been computed by the H-Solver, the H-Rockmate framework generates a new `nn.Module` which follows that execution schedule. For performance and compatibility reasons, this part of H-Rockmate is completely new and not based on the corresponding Rockmate package rk-Exec.

C.2.1 The execution procedure

The rk-Exec package converts the complete schedule produced by rk-Rotor into a single very large `string` of Python commands, which is then executed with the `exec()` built-in function. This has significant complexity in the code, and results in significant overhead

for the running time. For example, running the `nn.Transformer` model with this approach results in almost doubling the execution time.

We use a different approach in H-Rockmate, where the schedule is interpreted on the fly and each operation is performed in a controlled environment. This enables H-Rockmate to efficiently execute schedules for large models, as demonstrated in the experimental section.

C.2.2 Autograd compatibility

The result of Rockmate does not respect the PyTorch convention for `nn.Module`: the backward phase must be explicitly triggered, and the produced module can not be safely re-used with different inputs. In addition, Rockmate schedules have the possibility to delete the output `Tensor` during the backward pass, which might break the code of the user if they need to re-use it afterwards.

This possibility is removed in H-Rockmate, and the implementation provided by H-Rockmate is completely compatible with the `autograd` mechanism of PyTorch. For example, the following code works as expected:

```
rematMod = HRockmate(model,sample,budget)
inp, tgt = next(dataset)
y = linear(inp)
z1 = rematMod(y)
z2 = rematMod(x)
z = z1 + z2
loss = loss_function(z, tgt)
loss.backward()
```

D Ablation Study

Through this ablation study, we aim to shed light on the key factors influencing the performance of H-Rockmate and highlight its superiority over existing approaches in terms of peak memory - training time trade-off across different batch sizes and input data resolutions. The findings of this study contribute to a deeper understanding of the underlying mechanisms and efficacy of H-Rockmate, further solidifying its position as a leading solution in the field of re-materialization.

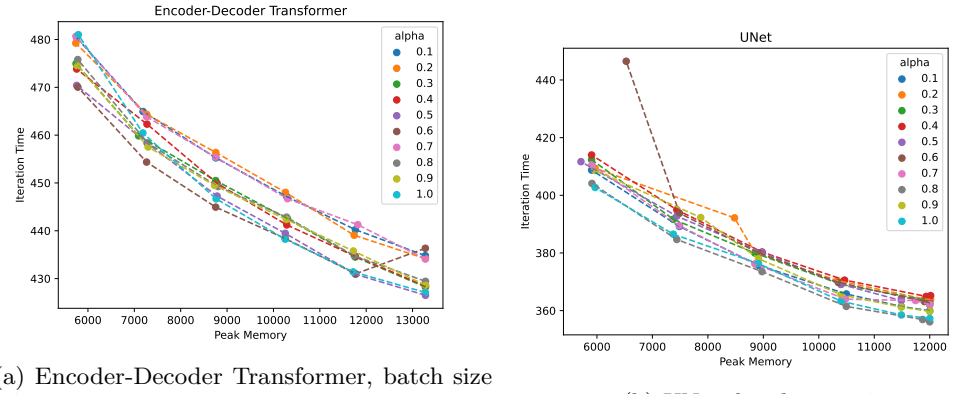
D.1 H-Rockmate hyper-parameters

In this section, we conduct an ablation study to examine the influence on the performance of H-Rockmate of the α hyper-parameter, which impacts the quality of the partitioning. By systematically varying the value of α, we aim to gain insights into its effects on the overall performance of H-Rockmate. This analysis provides valuable information for optimizing the hyper-parameter configuration to achieve the best possible results in different scenarios.

On Figure 5, we observe that the parameter α has a small but measurable impact on the quality of the solution returned by H-Rockmate. However, the value which provide the most efficient solution depends on the graph of the model, and we can not conclude on an optimal value for α. In the rest of the plots in this paper, we use the default $\alpha = 0.5$ which provide reasonable results in most cases.

D.2 Performance for Different Input Sizes

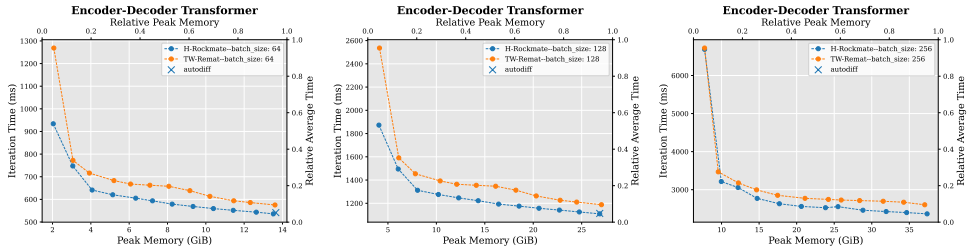
Furthermore, we show performances of our module H-Rockmate, across different batch sizes and input data resolutions. We focus on the Transformer and UNet models, and we vary the batch sizes and the input size (in terms of sequence length for Transformer, and in terms of image resolution for UNet). Figure 6 reports results for sequence length 200 and varying batch sizes. Figure 7 reports results for batch size 64 and varying sequence lengths. Figure 8 reports results for fixed resolution 256x256, and varying batch sizes. Figure 9 reports results for fixed batch size 64 and varying resolutions. In addition, we also provide on Figure 10 the results for the MLPMixer model with batch size 64 and resolution 256x256. The behavior of H-Rockmate is consistent across all cases, and significantly outperforms TW-Remat.



(a) Encoder-Decoder Transformer, batch size 64

(b) UNet, batch size 64

Figure 5: Experiments with H-ILP for different graph partitioning defined by α hyper-parameter

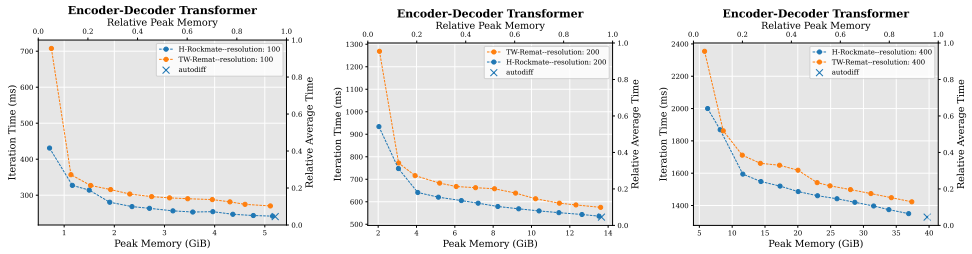


(a) Batch size 64

(b) Batch size 128

(c) Batch size 256

Figure 6: Experiments Encoder-Decoder Transformer with different batch sizes

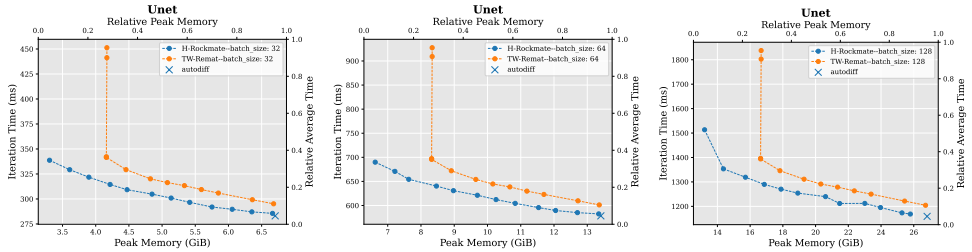


(a) Sequence length 100

(b) Sequence length 200

(c) Sequence length 400

Figure 7: Experiments Encoder-Decoder Transformer with different sequence lengths

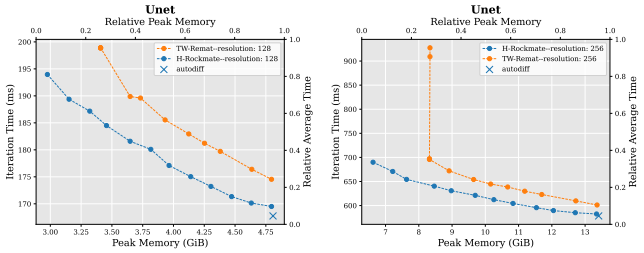


(a) Batch size 32

(b) Batch size 64

(c) Batch size 128

Figure 8: Experiments UNet with different batch sizes and resolution 256x256



(a) Resolution 128x128 (b) Resolution 256x256

Figure 9: Experiments on UNet with batch size 64 and different resolutions

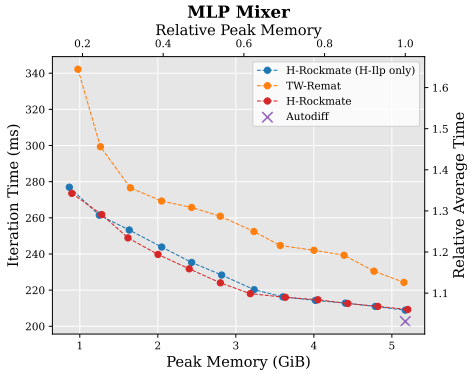


Figure 10: Experiments MLPMixer

E Broad study on a variety of models

We assess the robustness of our approach by analyzing its behavior on a variety of models. Table 3 provides results on GPT-medium and GPT-largest (which have 24 and 96 layers), UNet, MLPMixer, RegNet32, ResNet101, Transformer with 6 encoder and decoder layers, Transformer with 36 encoder and decoder layers, 1D and 3D FNO Li et al. [2020] models, UFNO Wen et al. [2022] and UNO Rahman et al. [2022].

Table 3a describes the computational graphs of each model:

- the number of nodes of the computational graph before simplification;
- the number of nodes in the forward computational graph, after simplification, which is the input to the H-Partition algorithm;
- the number of nodes in the forward-backward computational graph;
- the total number of computation and data nodes in the final computational graph, which is the input to the H-Solver algorithm.

Table 3b describes the result of the H-Partition algorithm:

- the depth of the hierarchical decomposition (without counting the top-level graph, so that CHECKMATE corresponds to depth 0 and ROCKMATE corresponds to a depth of 1);
- the number of subgraphs;
- the number of *unique* subgraphs after identifying identical subgraphs;
- the size of the largest subgraph (in terms of number of nodes).

Table 3c describes the result of H-ROCKMATE on each model:

- the total execution time of the H-Rockmate framework (including building the graph, partitioning and solving);
- the budget provided to H-Rockmate;
- the relative memory usage (compared to the peak memory of the `autodiff` solution) of the resulting `nn.Module` created by H-Rockmate, as measured in a real execution;
- the "predicted" value of the running time of the computed schedule, which is the sum of the measured execution times of all operations in the schedule, expressed relatively to the execution time of the `autodiff` solution;
- the execution time measured from running the `nn.Module` created by H-Rockmate, also expressed relatively to the execution time of the `autodiff` solution.

We can see that H-Rockmate obtains robust results on all these very different models: it is able to reduce memory usage by a factor 2, for a cost in execution time that varies between 7% (for RegNet32) and 22% (for UNO). All graphs except for UNet, FNO 1d and FNO 3d are much too large for Checkmate to solve. The Transformer, U-FNO and UNO models can not be meaningfully sequentialized, so that Rockmate also fails on these graphs. In contrast, the solving time of H-Rockmate remains below one hour for most graphs, and below three hours for the largest Transformer network, which is totally acceptable.

	Size before simplification	Size fwd only	Size fwd and bwd	Size computation and data nodes
GPT 24 (medium)	1858	367	734	1614
GPT 96 (largest)	7402	1447	2894	6366
UNet	73	50	101	205
MLPMixer	203	125	250	549
RegNet32	245	173	347	721
ResNet101	346	211	423	851
Transformer 6-6	1030	224	417	991
Transformer 36-36	6160	1334	2457	5851
FNO 1d	71	42	82	167
FNO 3d	317	64	121	257
U-FNO	567	118	221	467
UNO	997	129	229	495

(a) Size of the computation graphs

	Number of levels = depth	Number of subgraphs	# unique subgraphs	Largest subgraph
GPT 24 (medium)	2	28	8	15
GPT 96 (largest)	3	61	21	10
UNet	1	9	9	15
MLPMixer	1	15	6	10
RegNet32	1	15	10	15
ResNet101	1	19	10	15
Transformer 6-6	2	23	20	17
Transformer 36-36	15	128	67	19
FNO 1d	1	10	5	8
FNO 3d	1	12	12	10
U-FNO	1	11	9	17
UNO	3	12	12	19

(b) Results of H-partition

	Total solving time (min)	Memory budget (GB)	Observed relative peak memory	Predicted relative average time	Observed relative average time
GPT 24 (medium)	7.9	1.7	36.6%	122.3%	120.3%
GPT 96 (largest)	15	4.5	47.2%	119.6%	126.0%
UNet	9	5	51.3%	113.5%	113.3%
MLPMixer	1.5	4.5	53.3%	107.3%	109.0%
RegNet32	5	3.5	41.1%	107.0%	105.8%
ResNet101	5.5	4.5	42.6%	115.3%	115.3%
Transformer 6-6	20	3.5	46.9%	117.6%	114.8%
Transformer 36-36	147	4.9	44.6%	118.4%	
FNO 1d	2.5	5	53.1%	114.7%	111.0%
FNO 3d	5.3	4	47.2%	117.4%	104.1%
U-FNO	16.3	5.2	56.7%	116.3%	116.5%
UNO	46	5.2	58.5%	121.9%	116.7%

(c) Solving time and performances

Table 3: Test of H-Rockmate over 12 models, with a memory budget around half the `autograd` memory usage. All models passed a sanity check: both forward and backward passes produce the exact same result as the original module. Experiments are done on a NVIDIA P100 GPU with 16GB. See column headers and model definitions on previous page.