HIREMATE: Hierarchical Approach for Efficient Re-materialization of Neural Networks

Julia Gusak [*1] **Xunyi Zhao** [*1] **Théotime Le Hellard** [*2] **Zhe Li** [1] **Lionel Eyraud-Dubois** [1] **Olivier Beaumont** [1]

Abstract

Training deep neural networks (DNNs) on memory-limited GPUs is challenging, as storing intermediate activations often exceeds available memory. Re-materialization, a technique that preserves exact computations, addresses this by selectively recomputing activations instead of storing them. However, existing methods either fail to scale, lack generality, or introduce excessive execution overhead. We introduce HIREMATE, a *hierarchical* re-materialization framework that recursively partitions large computation graphs, applies optimized solvers at multiple levels, and merges solutions into a global efficient training schedule. This enables scalability to significantly larger graphs than prior ILP-based methods while keeping runtime overhead low. Designed for single-GPU models and activation re-materialization, HiRemate extends the feasibility of training networks with thousands of graph nodes, surpassing prior methods in both efficiency and scalability. Experiments on various types of networks yield up to 50-70% memory reduction with only 10-15% overhead, closely matching optimal solutions while significantly reducing solver time. Seamlessly integrating with PyTorch Autograd, HiRemate requires almost no code change to use, enabling broad adoption in memory-constrained deep learning.

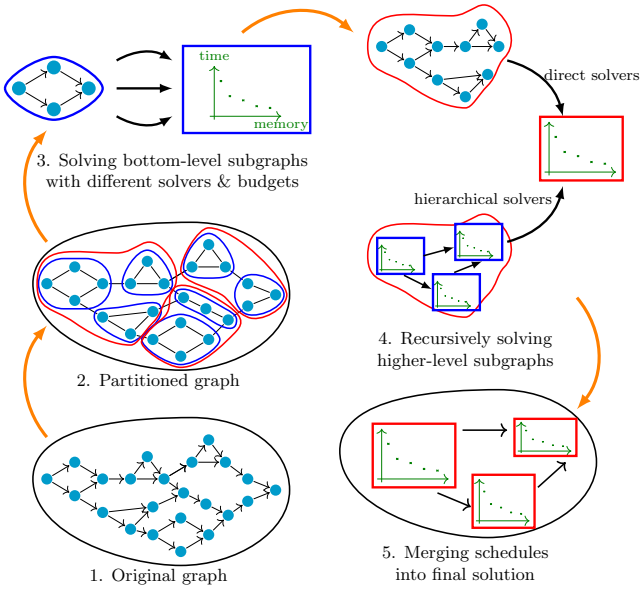


Figure 1. **HIREMATE** takes a PyTorch `nn.Module` and creates a `nn.Module`, which provides same outputs while satisfying peak memory budget constraints B during training. (1) Obtain data-flow graph G from the PyTorch module. (2) Recursively partition G with **H-Partition** (Section 3.1) into small-size subgraphs. (3)-(5) Obtain a re-materialization schedule from G with budget B using **H-Solver** (Section 3.2). (6) Produce a new `nn.Module` whose execution follows the schedule (Appendix).

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.

[*]Equal contribution [1]Inria Center at the University of Bordeaux [2]École Normale Supérieure, PSL University, Paris. Correspondence to: Julia Gusak <yulia.gusak@inria.fr>, Lionel Eyraud-Dubois <lionel.eyraud-dubois@inria.fr>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

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., 2019) and in the case of chains of complex blocks (Zhao et al., 2023), re-materialization 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 NP-Hard in (Naumann, 2008) for 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. These limitations stem either from the computational cost and the scalability of the algorithm that generates the re-materialization strategy, or from the overhead of that re-materialization strategy during the training phase, as discussed in Section 2. Another line of research is to propose re-materialization strategies whose computational cost and overhead are controlled, as in ROTOR (Beaumont et al., 2019) and ROCKMATE (Zhao et al., 2023), however at the cost of generality, by proposing solutions only 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 and we propose a computationally efficient, low-overhead solution that can address general graphs. Our framework HIREMATE is based on a **hierarchical decomposition** approach 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 several levels in the hierarchy to handle very large graphs. 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. Furthermore,HIREMATE 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)`.

To achieve this result, we rely on the following main 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** for integrating any existing (or future) re-materialization strategy at any level of the hierarchy, and combining their strengths.

The rest of the paper is organized as follows. In Section 2, we review the related work on memory saving strategies for DNN training. HIREMATE 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.

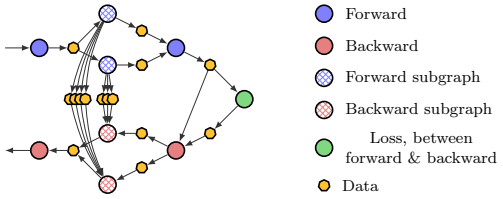


Figure 2. HIREMATE recursively finds schedules for each subgraph. Different schedules correspond to different memory budget constraints and hence have different values for memory/time ratio, peak memory, size of allocated tensors, and execution time.

2. 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). Gradient accumulation can also reduce memory usage by splitting a large batch into smaller sub-batches and accumulating gradients across them, but is limited by the low GPU utilization because of smaller sub-batches, and requires that one smaller sub-batch fits in memory. Model

Table 1. Comparison of Different Re-Materialization Strategies in terms of (i) Generality (ability to handle complex models with arbitrary dependencies), (ii) Scalability (ability to handle models with many operations), (iii) Quality (ability to generate a re-materialization strategy with low overhead during training), and Ease of use (in terms of modifications to be made to the original PyTorch model). Scores range from "-" (true limitation) to "++" (excellent).

	GENERALITY	SCALABILITY	QUALITY	EASE OF USE
CHECKMATE (JAIN ET AL., 2020)	++	-	++	0
TW-REMAT (KUMAR ET AL., 2019)	++	++	0	-
MOCCASIN (BARTAN ET AL., 2023)	++	++	+	-
ROCKMATE (ZHAO ET AL., 2023)	+	+	++	+
HIREMATE (PRESENT PAPER)	++	++	++	++

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 (xla) 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. Moccasin (Bartan et al., 2023) is a constraint programming based solution we show that HIREMATE does actually outperform Moccasin, both in terms of solution time and the quality of the solution produced.

Other approaches have been proposed to find efficient solutions for limited classes of dependency graphs. (Beaumont et al., 2024 (accepted for publication) 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 (where one Transformer block is considered as a complex block). However, for architectures with long skip connections (like U-Net (Ronneberger et al., 2015) or encoder-decoder Transformer (Vaswani et al., 2017)), ROCKMATE degenerates to CHECKMATE and inherits its difficulties to handle large data-flow graphs.

Such static rematerialization strategies are very efficient for models with fixed computation graphs, where memory usage can be optimized offline for efficient execution. Nevertheless, they struggle to handle models with dynamic or input-dependent control flow. In contrast, dynamic rematerialization strategies (Kirisame et al., 2020) adapt to runtime behavior by making decisions on-the-fly, allowing them to support a broader range of model architectures. However, this flexibility often comes at the cost of higher runtime overhead and less predictable performance. Notably, dynamic approaches such as POET (Patil et al., 2022) and MegTaiChi (Hu et al., 2022) combine rematerialization with additional mechanisms like paging.

In this paper, we focus specifically on static strategies and 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 HIREMATE with the state-of-the-art strategies: CHECKMATE, MOCCASIN, TW-REMAT, and ROCKMATE. Table 1 summarizes the comparison of these re-materialization techniques.

In the experimental evaluation of Section 4, we compare HIREMATE with the state-of-the-art strategies: CHECKMATE, TW-Remat, and ROCKMATE. Table 1 summarizes the comparison of these re-materialization techniques with several metrics. Generality denotes the ability to handle complex models with arbitrary dependencies, scalability denotes the ability to handle models with many operations. Quality measures the ability to generate a re-materialization strategy with low overhead during training, and ease of use is expressed in terms of modifications to be made to the original PyTorch model.

3. HIREMATE

Problem statement We associate each neural network with a *data-flow graph*, where each node represents a tensor-level computation. A *schedule* is a sequence of elementary computations and tensor deletions that performs the forward and backward passes for one training iteration.

Let F_i and B_i denote the forward pass and backward pass associated with layer i and D_i denotes the deletion of the output of F_i. Then, for a model that is a sequence of 3 layers, a schedule corresponding to the standard autodiff training can be written as $F_1F_2F_3B_3B_2B_1$, while $F_1F_2D_1F_3B_3D_3F_1F_2B_2D_2F_1B_1$ is another valid schedule, yielding a smaller peak memory since it does not require storing all intermediate activations simultaneously. **The re-materialization optimization problem** (2) **is to find a schedule whose peak memory is below a given budget and whose execution time is as small as possible.**

Figure 1 describes the main steps of the HIREMATE approach. In the first step, using the graph building tool adapted from (Zhao et al., 2023), **a data flow graph of the input module is obtained**. In the second step, the **H-partition** algorithm (see Section 3.1) **recursively partitions the graph into subgraphs** of manageable sizes. This partitioning is done recursively to ensure that all parent graphs are also of manageable size. Steps 3, 4, and 5 of Figure 1 describe the **H-Solver** algorithm (see Section 3.2), which **builds a training schedule**. Starting at the lowest level of the decomposition, the algorithm computes schedules for each subgraph (Figure 2) with different memory budgets to explore different time-memory tradeoffs, providing several *options* (*i.e* ways to (re)compute operations and store activations during forward and backward passes through the subgraph) for the nodes at higher levels. This procedure continues until the top-level graph is solved (*i.e* schedule is found) for a single memory budget corresponding to the overall memory available for activations. It is important to emphasize that the current implementation of the general scheme described above and in Figure 1 is fully modular. Thus, **HIREMATE allows the injection of custom partitioners as well as solvers at any level of the graph**.

3.1. H-Partition

The goal of HIREMATE's partitioning step is to reduce the size of the problems to be solved without compromising the quality of the overall solution. The result of this step is a decomposition into a hierarchy of subgraphs, where a node at a given level represent an entire subgraph at the level below (see Figure 7 in Appendix). Note that the original graph is partitioned without requiring a topological order; topological sorting is only performed later, within each subgraph, during the schedule generation phase (see Section 3.2). The subgraph sizes are bounded 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 advantageous to allow a longer solution time for the top-level graph, we use $M^t \geq M^l$. Our partitioning algorithm is a greedy bottom-to-top heuristic described in Algorithm 1. Each iteration has three main steps: forming *candidate* groups, selecting the best candidate according to our evaluation criterion, merging the selected candidate, and updating the candidates. Each step is described below.

Algorithm 1 H-Partition Bottom-to-Top 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 while
  Consider all groups as subgraphs
  Update G so that each subgraph is considered as a node
end while
return partitioned graph

Forming candidate groups For each node x in G, we consider $a(x)$, the closest common ancestor of all direct predecessors of x (a common global ancestor is added in case G has multiple entries). 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.

Selecting the best candidate When selecting the best group among all candidates, our goal is to avoid incurring too much memory pressure when the group is used as a subgraph. The memory pressure depends directly on the size of the input and output values. It also depends, although not as directly, on the length of the schedule during which they will be alive, which we evaluate by the number of compute nodes in the group. We use the following score function $s(C)$ for a candidate C:

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

where α is a hyperparameter whose default value is 0.5.

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

Section A in Appendix contains a proof that this procedure always leads to a valid decomposition, in the sense that no subgraph contains a directed cycle. After partitioning, HIREMATE identifies subgraphs that correspond to the execution of the same piece of code to avoid solving the same optimization problem multiple times.

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 multiple possible memory budgets. The resulting re-materialization schedules for a given subgraph are called *options*, and each corresponds to a different tradeoff between computation time and required memory. **Once all subgraphs at a given level of hierarchy have been resolved, a schedule for the upper level can be computed with our proposed H-ILP**. H-ILP is an Integer Linear Program (ILP) formulation that provides an optimal schedule within a given memory budget. It can be used on a subgraph of arbitrary structure, but using it on large subgraphs can lead to unreasonably long solution times. Therefore, it is only applied to subgraphs with a sufficiently small number of nodes. This algorithm is inspired by RK-CHECKMATE (Zhao et al., 2023) and CHECKMATE (Jain et al., 2020). The general idea of extending an ILP-based approach to a graph of arbitrary size and structure, which is hierarchically decomposed into subgraphs of manageable size, is one of the major contributions of the present work and is described in detail in Section 3.2.2.

For significantly large graphs, partitioning with only two levels would result in the top-level graph still being too large to be solved with this technique. Our hierarchical approach makes it possible to introduce more than two levels into the hierarchy and compute schedules for the levels from bottom to top, handling graphs of arbitrary size while keeping each subproblem of manageable size.

The H-ILP solver we propose is very generic and efficient, and it provides very good quality solutions on all types of neural networks. **Thanks to the genericity and extensibility of the HIREMATE framework, it allows combining different solvers**, which further improves the quality of the solutions. All solvers can be used at any level, but those not acting hierarchically ignore the underlying partition. Each solver comes with an *applicability* criterion to decide whether it is suited for a given graph. For each subgraph, all applicable solvers are used to produce schedules.

The HIREMATE framework currently includes 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 HIREMATE framework allow this heuristic to be used with PyTorch where previously it was only available for TensorFlow. Second, RK-ROTOR is the **dynamic programming** algorithm from (Zhao et al., 2023), which also provides very good schedules in the case of (forward) graphs consisting of a sequence of possibly complex subgraphs. It is therefore limited in genericity, but its computational time is low. The RK-ROTOR solver is only activated when we detect that the graph can be decomposed into a sequence.

3.2.2. HIERARCHICAL ILP FORMULATION

Consider an arbitrary graph H, where each computation node represents a subgraph (Figure 2), and where dependencies are carried by *data nodes* representing values that can be stored in memory. The H-ILP formulation computes the minimum runtime schedule whose peak memory remains below a given memory budget. The computation nodes are numbered in topological order.

In the linear programming formulation of H-ILP, the schedule is divided into *phases*, and the goal of phase t is to compute node t for the first time. For the sake of brevity, we describe here only the main ideas that allow H-ILP to be used in a hierarchical setting, but refer the reader to the Section B of Appendix for a complete description of the H-ILP formulation.

Compute options and phantom nodes The innovation of H-ILP 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 associated subgraph. There is a tight coupling between the forward computation and its corresponding 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 is used for each computation node within each phase.

In H-ILP, we also introduce an explicit representation of the data stored in memory between a forward computation and its corresponding backward computation. 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 used by any other computation 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 stored 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 includes additional variables that specify which option of each phantom node is in memory at each stage.

Objective Let $R^t_{i,o}$ equals 1 if node i is computed with option o during phase t and 0 otherwise. Then the objective is to minimize the total execution time expressed as

$$\min \sum_{i,t,o} R^t_{i,o} * \text{time of computing option } o \text{ for node } i \quad (2)$$

subject to constraints on auxiliary variables specified in Section B.3 of Appendix.

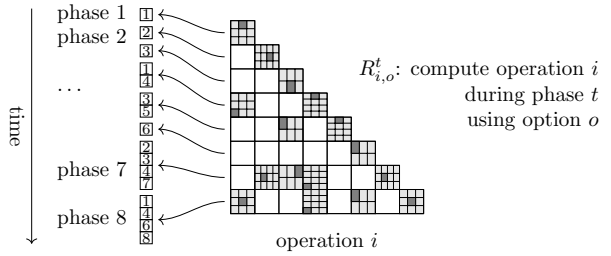


Figure 3. Example of the solution. $R^t_{i,o}$ equals 1 if node i is computed with option o during phase t.

Correction terms for memory usage In (Zhao et al., 2023), 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 this paper, a subgraph may share its input values with other subgraphs. To achieve good performance, H-ILP allows subschedules to delete their input values and the gradients associated to their outputs; however, if another subgraph needs these values, the higher-level algorithm adjusts the subschedules. This decision has an impact on the memory consumption of the corresponding subgraphs.

We have added constraints to the formulation to ensure that the memory peak is correctly evaluated in all cases. These constraints are introduced only when solving the top-level graph, since they make the ILP harder to solve, and only the top-level graph solution is required to fulfill accurately the GPU memory bound. 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 associated to option o for node k. In practice, this number of constraints remains small enough so that the H-ILP formulation can be 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 hyperparameter N_o that imposes a limit on the total number of options. To stay within this limit, we may need to select only a few options from all the schedules generated at the lower level. To do so, we split N_o to assign a number of options to each node proportional to the number of basic operations within the associated subgraph. Then, we greedily select the schedules whose memory peaks are farther apart from each other: starting with the schedule with the highest memory peak, then the one with the lowest memory peak, then the one closest to the middle, and so on.

3.3. Complexity Analysis

The complexity of using the H-ILP solver depends mostly on the number N of nodes in the graph. Obtaining the dataflow 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 of $O(N^2 \log N)$. This step is also fast 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 $N = 10^5$ takes 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 linear complexity. In fact, the hierarchical decomposition has a logarithmic depth, and the total number of subgraphs is $O(N/M^t)$. Solving a 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. Moreover, all ILPs at the same level are completely independent and can be run 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 a Transformer with 36 encoders and decoders have 2500 nodes and are solved in 15 and 150 minutes, respectively.

Figure 4. Experiments for varying graph size: (top) iteration time (bottom) solving time

4. Experimental Evaluation

Experimental settings The HIREMATE framework is designed to be used for end-to-end training. Since re-materialization guarantees that all computed values (including the gradients) are the same as without any recomputation, we focus in our experiments on measuring peak memory usage and iteration time for a single training iteration. We perform a warm-up phase consisting of five initial runs; the subsequent ten runs are used to evaluate the peak memory and computation time, providing reliable estimates of 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 shown in the plots. The experiments were performed on an NVIDIA Quadro RTX8000 GPU with 48 GB of memory and an NVIDIA V100 GPU with 16 GB of memory, using PyTorch 2.0.1, CUDA 11.6, and Gurobi 9.5.0. We intentionally report performance on settings where the experimental platform can run the original model, so that we can compare our results with the training time obtained with regular PyTorch Autodiff. All experiments can be scaled up by increasing image or batch size, to a point where training requires using HIREMATE. Additional experiments including an ablation study, varying batch sizes and sequence lengths, as well as a dozen different architectures are available in the Appendix.

Efficiency In Figure 4 we compare the solving time and performance of ROCKMATE, CHECKMATE, MOCCASIN, and H-ILP. Since solving an ILP has exponential complexity, it is important to limit the size of the problem. A clear disadvantage 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, leading to unacceptable solving times for graphs that do not follow this structure. This is typically the case in the encoder-decoder transformer (Vaswani et al., 2017), as shown on the bottom of Figure 4. Specifically, ROCKMATE takes 15 hours to obtain a solution for a 4-layer encoder-decoder structure transformer, while H-ILP achieves better results within 12 minutes. Note that H-ILP is run once before the entire model training, so 12 minutes is clearly acceptable. By controlling the total number of nodes in each graph with the H-partition algorithm, H-ILP is able to efficiently limit the solving time without compromising solution quality. It is important to note that though MOCCASIN claims to beat CHECKMATE, it does not provide an optimal solution in general, because the number of rematerializations is bounded in practice (to 2, to limit complexity), and the time allocated to the constraint programming solver is also bounded (to 1h, because the time to reach a solution can be arbitrarily long).

Performance Figure 5 shows that H-ILP consistently outperforms TW-REMAT in iteration time for a given budget. On the encoder-decoder (Figure 5(c)), TW-REMAT is able to find schedules for smaller memory budgets than H-ILP, but for UNet the situation is the reverse. The HIREMATE algorithm refers to the complete framework, including the RK-ROTOR and TW-REMAT solvers. By integrating TW-REMAT, HIREMATE overcomes this limitation of H-ILP and provides efficient solutions over the entire range of memory budgets. Figures 5(a) and 5(b) show that while H-ILP may perform slightly worse than ROCKMATE, H-ILP achieves similar performance to ROCKMATE, a baseline approach specifically tailored for such architectures.

Overall, HIREMATE acts as a general solution that includes H-ILP, ROTOR, ROCKMATE, and CHECKMATE. Our experimental results consistently show that HIREMATE performs on par with these baseline algorithms, demonstrating its versatility and effectiveness in optimizing memory utilization for a wide range of network architectures.

Controlling Sub-Optimality With our hierarchical approach, the H-ILP solver does not compute an optimal solution to the re-materialization problem. We compare our results with RK-CHECKMATE, which provides an optimal solution for a given topological order of operations. The selected results in Table 2 show that H-ILP achieves performance close to this optimal solver on a network size where it remains tractable to compute. In another experiment, we investigate the impact of hierarchy depth in the H-partition by restricting the size of each subgraph, thereby increas-

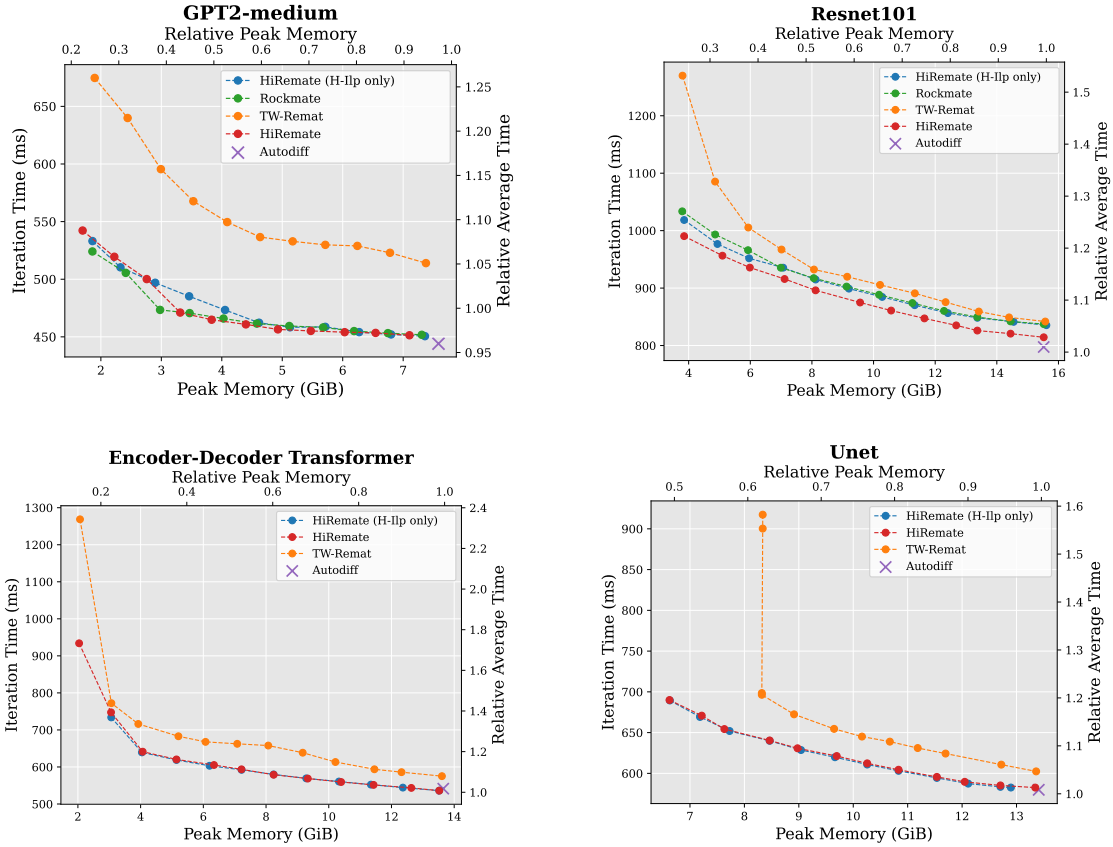


Figure 5. Experiments on different network architectures: (top) sequential-like neural networks (bottom) non-sequential neural networks, which ROCKMATE cannot solve efficiently.

ing the depth. As shown in Figure 6, the solution quality achieved by H-ILP shows only a slight degradation as the hierarchy becomes deeper. This indicates that HIREMATE effectively preserves performance despite deeper partitioning. It is important to note, however, that increasing the depth significantly affects the complexity and tractability of the problem as shown in Section 3.3, making this result particularly notable: H-ILP manages to avoid quality degradation in settings where solving the problem becomes considerably more challenging. The HIREMATE framework allows the user to use several trade-offs between solution quality and solving time, by adjusting the partition granularity, the number O of memory budget levels used in lower-level subgraphs, or the number N_o of options retained when solving one subgraph.

5. Discussion and Conclusion

This paper introduces the HIREMATE framework, which offers both theoretical and practical advances in re-materialization for PyTorch models. HIREMATE provides a solution that can find very effective solutions in terms of overhead during training, comparable to those of the literature for problems of small size or with a particular graph structure, but without these limitations. On the practical side, HIREMATE integrates seamlessly with PyTorch, improving the memory-time tradeoff and efficiency, and is fully compatible with PyTorch Autograd. The theoretical contributions include a hierarchical approach and an original linear programming formulation H-ILP. Although HIREMATE focuses on computational graphs with primitive operations, the hierarchical approach with linear complexity is potentially applicable to other intensive tasks targeting

Table 2. Comparison of H-ILP and RK-CHECKMATE overhead in terms of iteration time

Network	Budget (GB)	Checkmate	H-ILP
UNet	7.1	5.32%	5.44%
2-layer Tformer	7.0	5.32%	5.34%
MLP-Mixer	6.0	2.47%	2.44%
5-layer GPT2	1.25	6.02%	9.24%

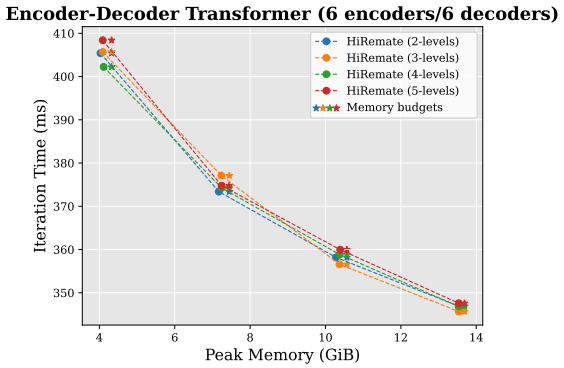


Figure 6. Experiments on 6-layer encoder-decoder Transformer with different depth of hierarchical structure.

graphs resulting from tiling. The framework also incorporates previous approaches, ensuring state-of-the-art results. The remaining limitation of HIREMATE is that it is not adapted to dynamic neural network architectures, where the the structure of the computational graph changes based on runtime inputs. However, HIREMATE can be extended to support mildly dynamic graphs by unrolling fixed-length control flow, precomputing schedules for a small number of known variants, or applying scheduling locally to submodules with static behavior. A critical point is that the code is fully modular, and it is possible for external contributors to introduce a new graph partitioner or solver at any level of the hierarchical decomposition. We hope that this modularity will stimulate research and further improvement of HIREMATE. Future research may include exploring how to integrate offloading into the optimization problem to reduce the need for recomputation, and how to optimally combine rematerialization with pipelined model parallelism. Advances in these areas hold promise for improving the performance and efficiency of deep learning systems.

Acknowledgments

The research has been partially supported by the EUPEX (European Pilot for Exascale) project, which received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101033975.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

XLA. http://www.tensorflow.org/xla.

Bartan, B., Li, H., Teague, H., Lott, C., and Dilkina, B. Moccasin: Efficient tensor rematerialization for neural networks. In *International Conference on Machine Learning*, pp. 1826–1837. PMLR, 2023.

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

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

Beaumont, O., Eyraud-Dubois, L., Hermann, J., Joly, A., and Shilova, A. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *ACM Transactions on Mathematical Software (TOMS)*, 2024 (accepted for publication). URL https://arxiv.org/abs/1911.13214.

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

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

Hu, Z., Xiao, J., Deng, Z., Li, M., Zhang, K., Zhang, X., Meng, K., Sun, N., and Tan, G. Megtaichi: Dynamic tensor-based memory management optimization for dnn training. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pp. 1–13, 2022.

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

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

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

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

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

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

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

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

Patil, S. G., Jain, P., Dutta, P., Stoica, I., and Gonzalez, J. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pp. 17573–17583. PMLR, 2022.

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

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

Ronneberger, O., Fischer, P., and Brox, T. 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*, pp. 234–241. Springer, 2015.

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

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

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

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

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

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

Zhao, X., Le Hellard, T., Eyraud-Dubois, L., Gusak, J., and Beaumont, O. 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.

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

Appendix

A. H-Partition algorithm

Algorithm 2 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 while
  Consider all groups as subgraphs
  Update G so that each subgraph is considered as a node
end while
return partitioned graph

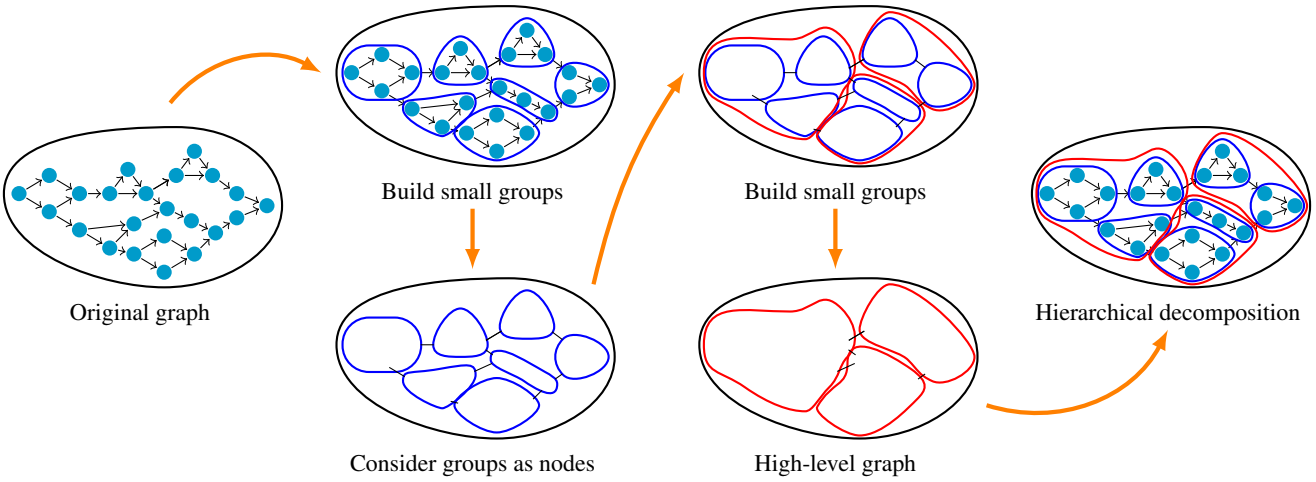


Figure 7. Visualization of recursive partitioning of the forward graph with 3 levels of hierarchy

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 A.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 5 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:

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

$$\text{If } u \in C, \quad \text{then } h(C) \rightsquigarrow u \tag{4}$$

This property ensure their convexity:

Lemma A.2. *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 (3) we have $u \in C$.
- If $u = h(C)$, then since $a \in C$, by (4), 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 A.3. *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 (3) applied to C', we have $v = h(C') \in C$, and from (4) 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 (3) applied to both C and C', we get $v = h(C) = h(C')$.

□

Theorem A.4. *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 A.3, 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 (4): consider any $u \in D$. If $u \in C$, then $h(D) = h(C) \rightsquigarrow u$ by (4) applied to C. If $u \in C'$, then $h(D) \rightsquigarrow h(C')$ by assumption and $h(C') \rightsquigarrow u$ by (4), so that in both cases $h(D) \rightsquigarrow u$.

We now show that D satisfies (3). 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 (3) applied to C' we get $v = h(C')$; and since $v \notin C$, the contrapositive of Lemma A.3 yields $h(C') = h(C) = h(D)$. Thus $v = h(D)$.
- If $u \in C$, then $v \notin C$ and (3) 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 5) and when they are updated (line 9). This implies that all subgraphs created line 11 are convex.

Identification of identical subgraphs To further improve efficiency of HIREMATE, we rely on and improve an idea from (Zhao et al., 2023): 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[1]. 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 provide details on H-ILP formulation, the hierarchical re-materialization approach based on solving linear programming problem.

B.1. Context

We assume that we have an arbitrary graph H, where each compute node represents a subgraph. Similar to RK-CHECKMATE from (Zhao et al., 2023), 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. COMPUTE 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 (Figure 8). 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.
- values saved in an 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.

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

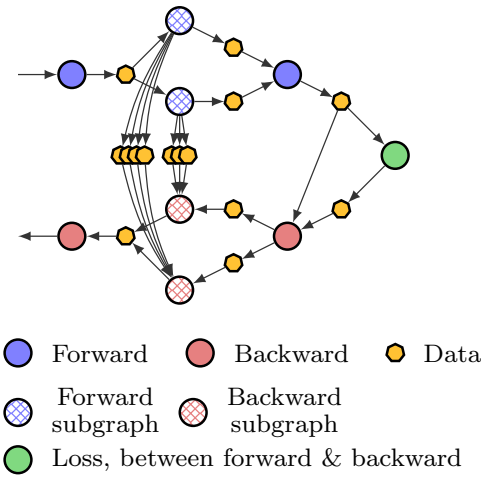


Figure 8. HIREMATE recursively finds schedules for each subgraph. Different schedules correspond to different memory budget constraints (options) and hence have different values for memory/time ration, peak memory, size of the saved data, and time for the backward computation.

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 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 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

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 (Figure 8), 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 node 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

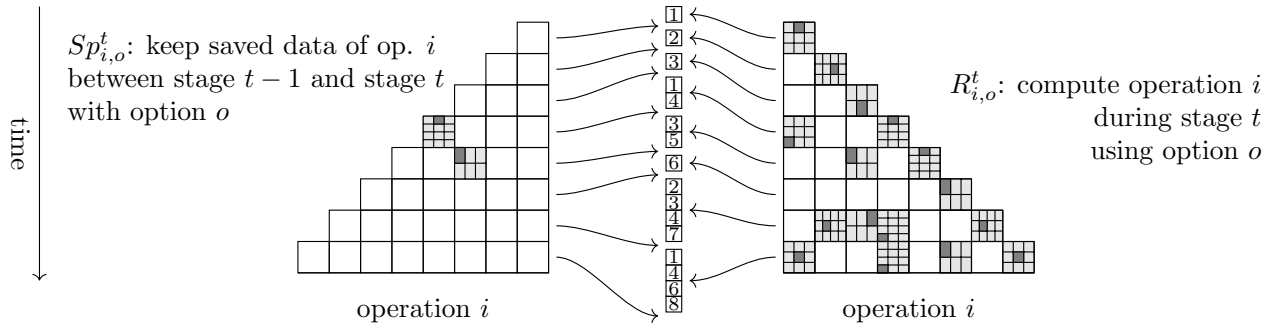


Figure 9.

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.3. Constraints of H-ILP

Constraints for Options H-ILP formulation contains constraints relative to the choice of options and the management of phantom nodes. Namely:

$$\forall t, \forall k, \quad \sum_o R^t_{k,o} \leq 1 \qquad \text{at most one option for each computation} \tag{5}$$

$$\forall t, \forall j, \quad \sum_o Sp^t_{j,o} \leq 1 \qquad \text{only one option of a phantom node is in memory} \tag{6}$$

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \leq Sp^t_{j,o} + R^t_{F_j,o} \qquad \text{a phantom node is only be created by its } F_j \tag{7}$$

$$\forall t, \forall j, \forall o, \quad Sp^{t+1}_{j,o} \geq Sp^t_{j,o} + R^t_{F_j,o} - R^t_{B_j,o} \qquad \text{a phantom node is only deleted by its } B_j \tag{8}$$

$$\forall t, \forall j, \forall o, \quad R^t_{B_j,o} \leq Sp^t_{j,o} + R^t_{F_j,o} \qquad \text{computing } B_j \text{ requires the phantom node} \tag{9}$$

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 (Zhao et al., 2023), which is equal to 1 if node k is computed during phase t (with any option).

Validity constraints

$\forall t, \forall k > t,$	$CR_k^t = 0$	Node $k > t$ can not be computed in phase t
$\forall t, \forall j$ s.t. $F_j > t, \forall o,$	$Sp_{j,o}^t = 0$	No phantom j from node $F_j > t$ is saved before phase t
$\forall k \to d, \forall t \leq k,$	$S_{k,d}^t = 0$	No result of node k can be saved in any phase before phase k
$\forall d, \forall t \leq \min\{k \mid k \to d\},$	$P_d^t = 0$	Data node d is not in memory before any of its predecessors
$\forall d \in \mathcal{F}, \forall k \to d,$	$S_{k,d}^T + CR_k^T = 1$	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
	$\sum_t CR_l^t = 1$	The *loss* node is executed only once

Data dependencies

$\forall t, \forall k \to d,$	$S_{k,d}^t \leq P_d^t$	Data node d with at least one contribution k is alive
$\forall t < T, \forall k \to d,$	$S_{k,d}^{t+1} \leq S_{k,d}^t + CR_k^t$	New contribution k to node d only appears by being computed
$\forall t, \forall k \to d \to k',$	$CR_{k'}^t \leq CR_k^t + S_{k,d}^t$	Computing node k' requires all contributions k to input node d
$\forall t, \forall j, \forall o, \forall k \to d \xrightarrow{o} B_j$	$R_{B_j,o}^t \leq CR_k^t + S_{k,d}^t$	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 $A_{k,d}^t$ 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:

$$\forall t, \forall k \leftrightarrow d, \quad A_{k,d}^t \doteq P_d^t + \sum_{k' \to d, k' \leq k} C_{k',d}^t - \sum_{k' \leftrightarrow d, k' \leq k} D_{k',d}^t$$

In the above equation and in the following, we use $\doteq$ to denote an alias definition, so that $A_{k,d}^t$ 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

$\forall t, \forall k \leftrightarrow d,$	$0 \leq A_{k,d}^t \leq 1$	Data node d is either alive or not
$\forall t, \forall k \to d,$	$A_{k,d}^t \geq CR_k^t - D_{k,d}^t$	d is alive if computed and not deleted
$\forall t, \forall k \to d,$	$C_{k,d}^t \leq CR_k^t$	value d can only be created by a node k that is really computed
$\forall t < T, \forall d,$	$P_d^{t+1} = A_{\max\{k \mid k \leftrightarrow d\},d}^t$	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:

$$\forall t, \forall k \leftrightarrow d, \quad D_{k,d}^t = 1 \text{ if and only if } CR_k^t = 1 \text{ and } P_k^{t+1} = 0 \text{ and } \sum_{d \to k', k' > k} CR_{k'}^t = 0$$

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

$$\forall t, \forall k \leftrightarrow d, \qquad D_{k,d}^t \geq CR_k^t - P_k^{t+1} - \sum_{d \to k', k' > k} CR_{k'}^t$$

$$\forall t, \forall k \leftrightarrow d, \quad h_{k,d}(1 - D_{k,d}^t) \geq 1 - CR_k^t + P_k^{t+1} + \sum_{d \to k', k' > k} CR_{k'}^t$$

Evaluating memory usage In contrast to RK-CHECKMATE, the memory budget constraints of H-ILP are updated to take into account the memory sizes of the phantom nodes where appropriate. For this purpose, we denote by U_k^t 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_k^t can be expressed as a linear combination of the formulation variables. Indeed, the memory usage before starting phase t is:

$$M_t \doteq \sum_d s_d \cdot P_d^t + \sum_{j,o} p_{j,o} \cdot Sp_{j,o}^t.$$

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

$$IF_k \doteq \sum_{k \to d} s_d \cdot C_{k,d}^t - \sum_{k \leftrightarrow d} s_d \cdot D_{k,d}^t + \sum_o p_{j,o} R_{k,o}^t.$$

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

$$IB_k \doteq \sum_{k \to d} s_d \cdot C_{k,d}^t - \sum_{k \leftrightarrow d} s_d \cdot D_{k,d}^t - \sum_o p_{j,o} \left(R_{F_j,o}^t + Sp_{j,o}^t - Sp_{j,o}^{t+1} \right)$$

The expression within the parenthesis is equal to 1 if the allocated node j is deleted, and 0 otherwise. Indeed, since B_j is the only compute node that can use it, $Sp_{j,o}^{t+1} = 0$ means that phantom node j can be deleted right after B_j. Constraint (8) ensures that if $R_{B_j,o}^t = 0$, then the expression within the parenthesis is also 0.

Finally, we can express U_k^t iteratively (similar to CHECKMATE and RK-CHECKMATE formulations):

$$\begin{aligned} \forall t, \quad & U_0^t \doteq M_t + IF_0 \\ \forall t, \forall k = F_j, \quad & U_k^t \doteq U_{k-1}^t + IF_k \\ \forall t, \forall k = B_j, \quad & U_k^t \doteq U_{k-1}^t + IB_k \end{aligned}$$

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:

$$\forall t, \forall k, \quad U_k^t + \sum_o m_{k,o} \cdot R_{k,o}^t + \sum_{k \leftrightarrow d} s_d \cdot D_{k,d}^t \leq B \tag{10}$$

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 $m^i_{k,o} - \sum_{d\in\mathcal{F}_i} s_d \cdot D^t_{k,d}$. The corresponding memory constraint is:

$$\forall t, \forall k, \forall o, \forall i, \quad U^t_k + m^i_{k,o} \cdot R^t_{k,o} + \sum_{k\leftrightarrow d} s_d \cdot D^t_{k,d} - \sum_{d\in\mathcal{F}_i} s_d \cdot D^t_{k,d} \leq B$$

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 $R^t_{k,o}$ is 0, this constraint is weaker than (10). 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 $C^t_{k,d}$.

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 $m^i_{k,o}$). The number of constraints is thus bounded by $\min($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 HIREMATE 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 HIREMATE: the graph building procedure, and the execution process.

C.1. Graph builder

In HIREMATE, 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 HIREMATE, 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 HIREMATE 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 HIREMATE 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 HIREMATE:

- 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*.

- HIREMATE 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 HIREMATE, 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 HIREMATE. In addition, if such an input value has `requires_grad=False`, then no backward computation is associated either. In HIREMATE, unlike ROCKMATE, the corresponding dependency is ignored. This enables HIREMATE 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 HIREMATE 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 HIREMATE 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 HIREMATE framework generates a new `nn.Module` which follows that execution schedule. For performance and compatibility reasons, this part of HIREMATE 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.

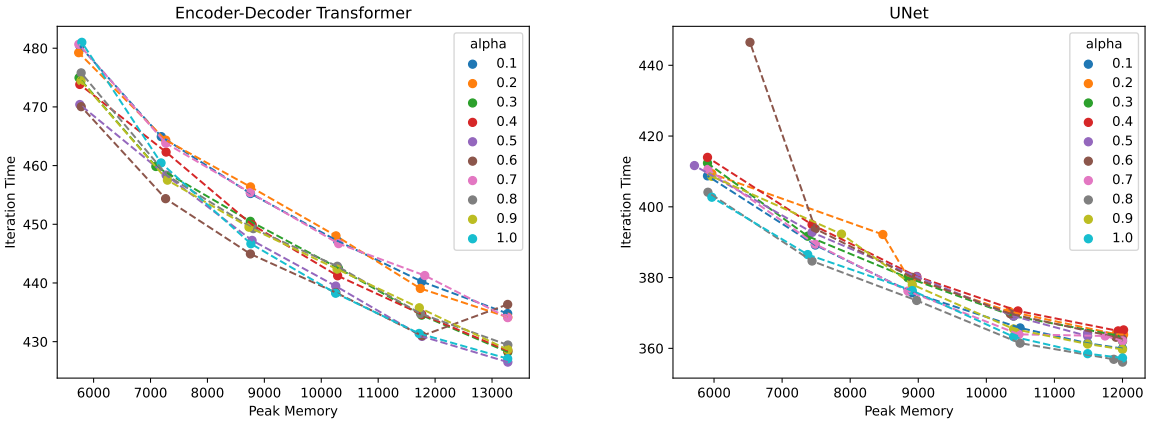
We use a different approach in HIREMATE, where the schedule is interpreted on the fly and each operation is performed in a controlled environment. This enables HIREMATE to efficiently execute schedules for large models, as demonstrated in the experimental section.

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 HIREMATE, and the implementation provided by HIREMATE 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()
```



(a) Encoder-Decoder Transformer, batch size 64

(b) UNet, batch size 64

Figure 10. Experiments with H-ILP for different graph partitioning defined by α hyper-parameter

D. Ablation Study

Through this ablation study, we aim to shed light on the key factors influencing the performance of HIREMATE 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 HIREMATE, further solidifying its position as a leading solution in the field of re-materialization.

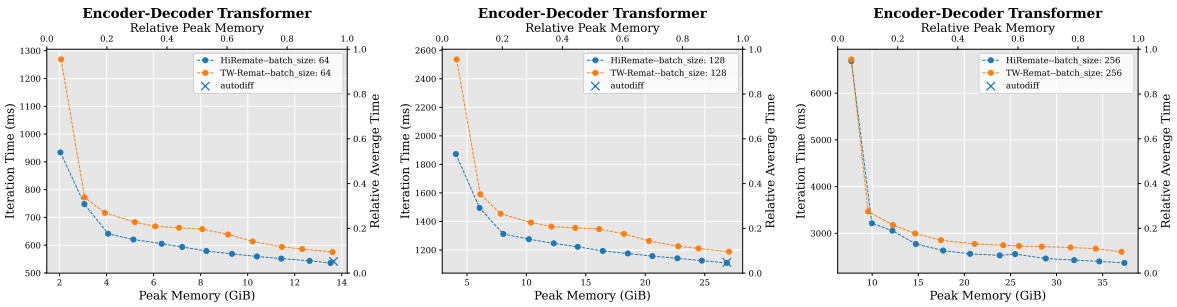
D.1. HIREMATE hyper-parameters

In this section, we conduct an ablation study to examine the influence on the performance of HIREMATE 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 HIREMATE. This analysis provides valuable information for optimizing the hyper-parameter configuration to achieve the best possible results in different scenarios.

On Figure 10, we observe that the parameter α has a small but measurable impact on the quality of the solution returned by HIREMATE. 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.

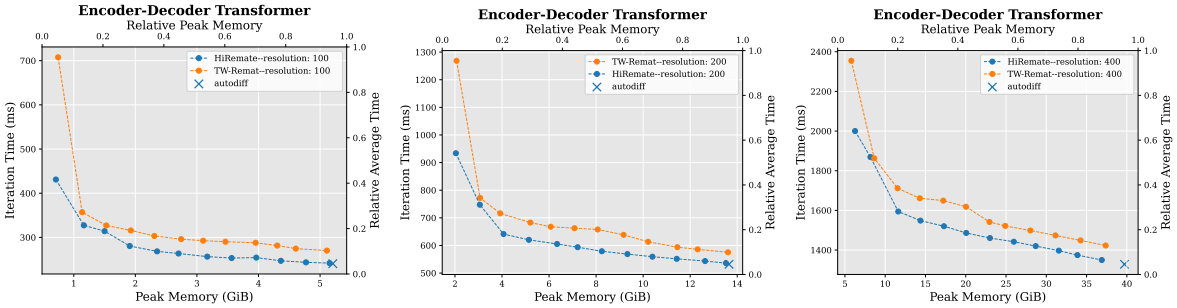
D.2. Performance for Different Input Sizes

Furthermore, we show performances of our module HIREMATE, 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 11 reports results for sequence length 200 and varying batch sizes. Figure 12 reports results for batch size 64 and varying sequence lengths. Figure 13 reports results for fixed resolution 256x256, and varying batch sizes. Figure 14 reports results for fixed batch size 64 and varying resolutions. In addition, we also provide on Figure 15 the results for the MLPMixer model with batch size 64 and resolution 256x256. The behavior of HIREMATE is consistent across all cases, and significantly outperforms TW-Remat.



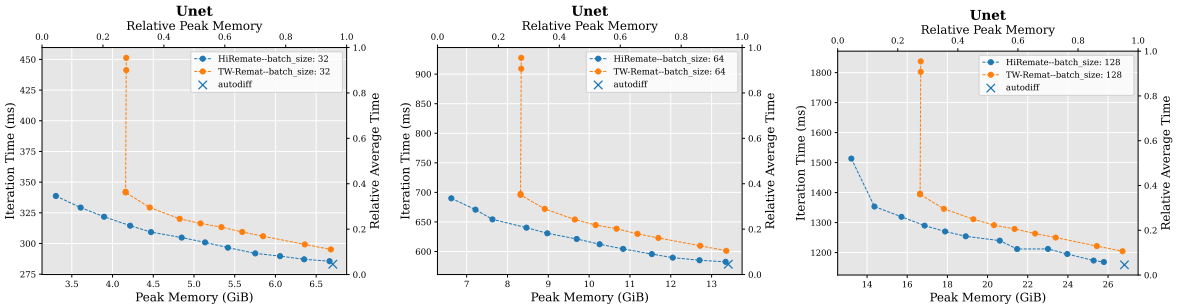
(a) Batch size 64 (b) Batch size 128 (c) Batch size 256

Figure 11. Experiments Encoder-Decoder Transformer with different batch sizes



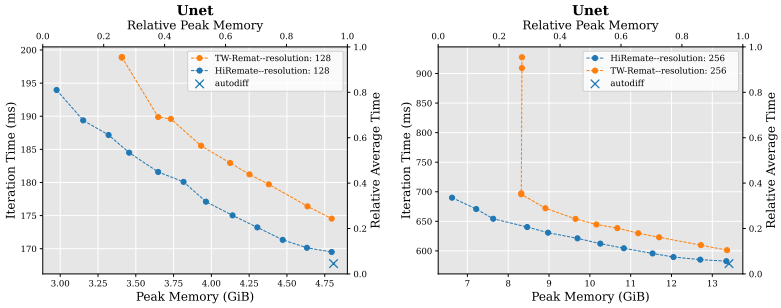
(a) Sequence length 100 (b) Sequence length 200 (c) Sequence length 400

Figure 12. Experiments Encoder-Decoder Transformer with different sequence lengths



(a) Batch size 32 (b) Batch size 64 (c) Batch size 128

Figure 13. Experiments UNet with different batch sizes and resolution 256x256



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

Figure 14. Experiments on UNet with batch size 64 and different resolutions

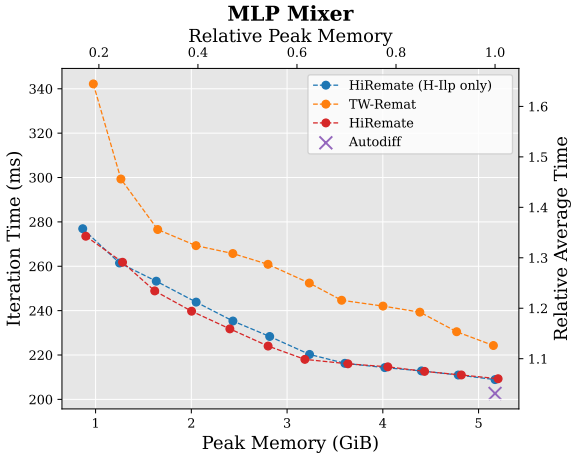


Figure 15. Experiments MLPMixer

Network	Budget	Checkmate	H-ILP
ResNet34	5.0e9	1.0138	1.0184
	4.0e9	1.0360	1.0394
	3.5e9	1.0517	1.0554
	3.0e9	1.0712	1.0761
UNet	9.1e9	1.0090	1.0090
	8.1e9	1.0293	1.0317
	7.1e9	1.0532	1.0544
	6.1e9	1.0776	1.0830
2-layer Tformer	9.0e9	1.0195	1.0194
	8.0e9	1.0355	1.0356
	7.0e9	1.0532	1.0534
	6.0e9	1.0700	1.0702
MLP-Mixer	8.0e9	1.0050	1.0044
	7.0e9	1.0137	1.0137
	6.0e9	1.0247	1.0244
	5.0e9	1.0466	1.0545
5-layer GPT2	1.4e9	1.0113	1.0117
	1.25e9	1.0602	1.0924
	1.1e9	1.1515	1.1929
	1.0e9		1.2593

Table 3. Comparison of H-ILP and the optimal solution of RK-CHECKMATE: overhead in terms of iteration time compared to Autodiff for different models and different budget values

E. Comparison with optimal solvers

We evaluate the quality of the solution returned by H-ILP by comparing with the optimal solver RK-CHECKMATE on small graphs, where the optimal solver is tractable to compute. We provide in Table 3 a broad comparison over several different models and different budget values. We observe on a wide variety of networks and budgets that the quality of the solution produced by H-ILP remains very close to the optimal value provided by RK-CHECKMATE when it is possible to compute it.

F. Broad study on a variety of models

We assess the robustness of our approach by analyzing its behavior on a variety of models. Table 4 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 4(a) 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 4(b) describes the result of the H-Partition algorithm:

- the depth of the hierarchical decomposition. The top-level count as the first level, so CHECKMATE, which works directly takes the whole graph, corresponds to a depth 1, and ROCKMATE, that takes a chain of blocks, corresponds to a depth of 2.
- 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 4(c) describes the result of HIREMATE on each model:

- the total execution time of the HIREMATE framework (including building the graph, partitioning and solving);
- the budget provided to HIREMATE;
- the relative memory usage (compared to the peak memory of the `autodiff` solution) of the resulting `nn.Module` created by HIREMATE, 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 HIREMATE, also expressed relatively to the execution time of the `autodiff` solution.

We can see that HIREMATE 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 HIREMATE remains below one hour for most graphs, and below three hours for the largest Transformer network, which is totally acceptable.

Table 4. Test of HIREMATE 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.

(a) Size of the computation graphs

	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

(b) Results of H-partition

	NUMBER OF LEVELS = DEPTH	NUMBER OF SUBGRAPHS	# UNIQUE SUBGRAPHS	LARGEST SUBGRAPH
GPT 24 (MEDIUM)	3	28	8	15
GPT 96 (LARGEST)	4	61	21	10
UNET	2	9	9	15
MLPMIXER	2	15	6	10
REGNET32	2	15	10	15
RESNET101	2	19	10	15
TRANSFORMER 6-6	3	23	20	17
TRANSFORMER 36-36	16	128	67	19
FNO 1D	2	10	5	8
FNO 3D	2	12	12	10
U-FNO	2	11	9	17
UNO	4	12	12	19

(c) Solving time and performances

	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%