Boosting Rematerialization Training via Execution Mode Splitting Modeling on Convex Optimized Dynamic Programming

Yu Tang*
Lujia Yin*
National University of Defense Technology
Changsha, China

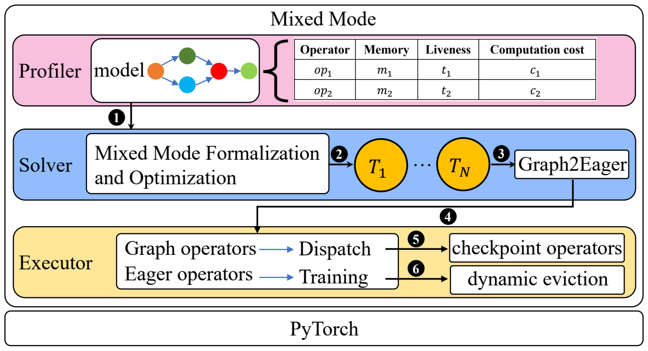
Qiao Li
Xiamen University
Xiamen, China

Yiming Zhang
Shanghai Jiao Tong University
Shanghai, China

GPU memory wall problem has become a significant bottleneck hindering further development of large-scale models [1, 4]. Many approaches focus on scheduling layers or operators in the computation graph for rematerialization or swapping to other devices to reduce GPU memory usage [2, 4, 6]. However, these methods, built on either graph mode or eager mode of deep learning frameworks, lack necessary flexibility and memory efficiency, or they suffer from poor training performance, respectively [3]. Graph mode follows a declarative programming approach while eager mode follows an imperative programming approach. Balancing memory consumption and training efficiency is challenging [2]. Specifically, graph mode encounters severe challenges in practical applications, including: 1) **Inadequate Memory Utilization**: current efficiency optimizations in graph mode may conflict with memory optimizations. For instance, fusing multiple small operators into a larger one may prevent separate eviction, consuming significant memory when residing on the GPU. 2) **Inadaptability to Control Flow**: in large-scale models, execution paths frequently change according to runtime conditions, such as conditional branches and loops [5]. The mismatch between the static computation graph and the dynamic nature of the control flow prevents tracking and training large-scale models in graph mode. 3) **Introducing prolonged checkpoint solving time**: the just-in-time compilation feature of graph mode requires predetermining checkpoints when the GPU budget is insufficient. However, as the size of large models grows, the high complexity of these algorithms significantly delays checkpoint solving, resulting in prolonged end-to-end training times.

To solve the above issues, we introduce Mixed Mode, as shown in Figure 1, that facilitates the training of large-scale models with limited GPU memory by integrating the strengths of both graph mode and eager mode. Our key insight is that while graph mode may exhibit suboptimal memory optimization or control flow failure in some operator subsequences, we try to automatically identify these subsequences and convert them to eager mode. This idea preserves the high performance of the overall graph mode

*Both authors contributed equally to this research.



Operator	Memory	Liveness	Computation cost
op_1	m_1	t_1	c_1
op_2	m_2	t_2	c_2

Figure 1. Mixed Mode execution overview. "Graph operators" means operators which are executed in graph mode and "Eager operators" means operators which are executed in eager mode.

while leveraging the local eager mode to adjust memory usage and eliminate the interference of the control flow.

Overview. Figure 1 shows the architecture of our Mixed Mode, which consists of three components, Profiler, Solver, and Executor. First, Profiler collects each operator's memory, liveness time, and computational cost in the input model. These values are sent to the Solver (❶). Then, Solver performs the Mixed Mode Formalization and solves the problem using our designed DP algorithm. The Solver determines the execution mode for each operator(❷), and marks operators with different labels according to the execution mode (❸). In Solver, we also use our Graph2Eager Conversion to improve the adaptability of control flow operators (❹). Then, Executor performs the execution of operators. Those graph operators will be executed according to their dispatch labels (❺) and eager operators will be evicted if OOM occurs (❻).

Mixed Mode Formulation. The input model could be translated into an operator sequence consisting of N operators. For each operator $o_i (i = 0, \cdots, N-1)$, there is computation cost c_i and memory consumption m_i. We define $seq(i, j)$ as the operator subsequence starting from o_i up to o_{j-1} $(i < j)$. Our goal is to minimize the total execution cost of the operator sequence $seq(0, N)$ under the constraint of memory budget M. We define $Opt(i, j, k)$ to represent the total execution cost of $seq(i,j)$ and with the memory consumption no more than k and operator o_{j-1} outputs to and resides in GPU memory. Obviously, $Opt(0,N,M)$ is the optimization objective. To solve it, we model the problem to a subsequence splitting DP (dynamic programming). First, we induce the subproblem and the recurrence formula of the DP. For any given state $Opt(0,i,k)$, its recurrence formula is articulated in Equation (1).

$$\begin{aligned} Opt(0,i,k) = \min_{0<\delta\leq k} (&Opt(0,i,k-1), \\ &Opt(0,j,k-\delta)+F(i,j,\delta)). \end{aligned} \tag{1}$$

For any mixed mode sequence $seq(0,i)$ under k memory budget, its solution $Opt(0,i,k)$ can be obtained by concatenating a subproblem's solution of $seq(0,j)$ (whose cost is $Opt(0,j,k-\delta)$ and memory budget is $k-\delta$) with a single execution mode subsequence (whose cost is $F(i,j,\delta)$, and memory budget is δ), where j is the concatenating point. Here, we define $E(i,j,\delta)$ as the eager execution mode cost and $G(i,j,\delta)$ as the graph execution mode cost, and set:

$$F(i,j,\delta) = \min(E(i,j,\delta), G(i,j,\delta)). \tag{2}$$

Graph2Eager Conversion. For the graph mode subsequence with control flow operators after operator fusion, we convert the execution mode of control flow operators from graph to eager. We accomplish the conversion by shifting control flow operators, initially part of the graph mode, one position left or right, allowing them to integrate into an eager mode subsequence. We revise the calculation of $\boldsymbol{F}$ from Equation (2) to Equation (3). For a subsequence $seq(i,j)$, it contains a control flow operator o_p $(i \leq p < j)$, its optimal cost $F(i,j,\delta)$ is obtained by:

$$\begin{aligned} F(i,j,\delta) = \min_{0<\epsilon<\delta} (&E(i,p+1,\delta-\epsilon)+G(p+1,j,\epsilon), \\ &G(i,p,\epsilon)+E(p,j,\delta-\epsilon), E(i,j,\delta)). \end{aligned} \tag{3}$$

This adjustment reallocates the budget, originally designated for a single execution mode subsequence, across operator sequences with two distinct execution modes.

Convex Optimization on DP. We employ dynamic programming to solve smaller subproblems by splitting the operator sequence into two parts. This splitting spans two dimensions: sequence length and memory budget. Since $\boldsymbol{Opt}$ and $\boldsymbol{F}$ are concave functions, there exists a unique optimal splitting point for i and δ in Equation (2). We refer to these points as the **equilibrium point**. Before we introduce the sequence splitting optimization and the memory partition optimization, we introduce a type of optimization called quadrangle inequality. Since Equation (1) holds for any enumerated variables k and δ, we consider $F(i,j,\delta)$ as a set of binary functions with respect to i and j. We use the symbols $w_\delta(i,j)$ to replace $F(i,j,\delta)$. The prerequisite for using quadrangle inequality optimization is Assumption 1,

Assumption 1. $\forall \delta, 0<i<j<N, w_\delta(i,j)+w_\delta(i+1,j+1) \leq w_\delta(i+1,j)+w_\delta(i+1,j+1) \models$

In simpler terms, although the total length is the same for those two pairs of operator sequences $< seq(i,j), seq(i+1,j+1) >$ and $< seq(i+1,j), seq(i,j+1) >$, the latter includes a longer operator sequence $seq(i,j+1)$, which needs more memory to store operators. The performance of $seq(i,j+1)$ declines more sharply. Thus, although Assumption 1 is very strong and almost impossible to strictly hold in practical applications, the satisfaction probability should be quite high. After using quadrangle inequality optimization, the overall complexity of $O(N^2M^2)$ could be reduced to $O(N \log NM^2)$.

On examining the enumeration of the memory budget δ, we consider that Equation (1) features a unique extremum point for δ. Both $Opt(0,i,k-\delta)$ and $F(i,j,\delta)$ are concave functions with respect to δ, where we place the derivation process of the function variable substitution of $\boldsymbol{Opt}$ from k to δ. Thus, their sum is also a concave function with respect to δ. For single-variable concave functions, the minimum value can be efficiently found using the ternary search. Consequently, the overall complexity is further reduced from $O(N \log NM^2)$ to $O(N \log NM \log M)$, making the DP computation viable.

Conclusion and Future Work. In conclusion, we propose Mixed Mode for boosting large-scale model rematerialization training, which overcomes the drawbacks of limited application scenarios of graph mode and low efficiency of eager mode. We utilizes the dynamic programming to achieve the optimal sequence splitting and reduce its complexity through convex optimization.

References

[1] Gholami Amir, Yao Zhewei, Kim Sehoon, Mahoney Michael W, and Keutzer Kurt. 2021. AI and Memory Wall. *RiseLab Medium Post* (2021).

[2] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations.*

[3] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 891–905.

[4] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21).* New York, NY, USA, Article 59, 14 pages.

[5] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. Torch.fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[6] Xunyi Zhao, Théotime Le Hellard, Lionel Eyraud-Dubois, Julia Gusak, and Olivier Beaumont. 2023. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in pytorch. In *International Conference on Machine Learning.* PMLR, 42018–42045.