# OFFMATE: full fine-tuning of LLMs on a single GPU by re-materialization and offloading

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

# OFFMATE: full fine-tuning of LLMs on a single GPU by re-materialization and offloading

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

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

**Théotime Le Hellard**
École Normale Supérieure, PSL University, Paris
theotime.le.hellard@ens.psl.eu

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

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

## Abstract

We present OFFMATE, an efficient memory-reducing framework to enable fine-tuning large language models on a single GPU. In the same way that PyTorch Dynamo takes a model and automatically changes it to reduce the execution time, OFFMATE takes a model and automatically modifies it to fit memory constraints (e.g. GPU VRAM), while keeping the same numerical results without approximation. OFFMATE uses integer linear programming to combine re-materialization (deleting some intermediate activations and recomputing them when needed), weight and activation offloading (moving data to CPU memory), and CPU optimization in a holistically optimized way, ensuring an efficient usage of available resources. With 10%-50% execution time overhead, OFFMATE has achieved up to $10\times$ GPU memory reduction on billion-size models including Llama, Phi, Bloom and Mistral from HuggingFace. OFFMATE is also designed to be compatible with reduced precision and parameter-efficient fine-tuning techniques, so that the memory benefits can be combined.

## 1 Introduction

Inspired by the transformer structure [30], Large Language Models (LLMs) with different architectures such as GPT [21], Bloom [31], Llama [29] and Mistral [16] have been trained and demonstrate excellent performance on general tasks. Based on these large pre-trained models, which require a significant amount of resources to train, many fine-tuned models have been proposed and contribute to a variety of fields such as law [13], medicine [32], and finance [18]. Compared to training an LLM from scratch, fine-tuning is more accessible to broader communities of artificial intelligence (AI) enthusiasts because it requires less data, less computing power, and less training time.

However, a common challenge for fine-tuning LLMs is the memory bottleneck of the training process. Most billion-parameter models can hardly be stored on consumer graphics cards, which typically have between 8GB and 24GB of video RAM (VRAM). Depending on the choice of hyperparameters and optimization settings, the training process can require far more memory than the VRAM available on a single GPU. While most LLMs are pre-trained using parallel approaches across hundreds of GPUs, many works like QLoRA or ZeRO [10, 25] have been proposed to allow fine-tuning of LLMs

Preprint. Under review.

| **Model**<br>**Memory Req.** | Bloom [31](3B)<br>28.0 GiB | Phi [15](3.8B)<br>43.6 GiB | Mistral [16](7B)<br>80.2 GiB | Llama3 [29](8B)<br>65.3 GiB |
|---|---|---|---|---|
| Original model | 2.35 sample/s | 1.91 sample/s | 0.93 sample/s | 0.97 sample/s |
| ZeRO-Infinity [24] | 0.67 sample/s | 0.64 sample/s | OOM | OOM |
| OFFMATE | 1.68 sample/s | 1.29 sample/s | 0.77 sample/s | 0.80 sample/s |

Table 1: Throughput (samples/second) of fine-tuning popular LLMs in bfloat16 precision with 12GB GPU and 128GB RAM. Original model corresponds to the expected runtime of the model on a hypothetical GPU with the same speed and infinite memory. Out of memory (OOM) of ZeRO-Infinity is due to the limit of CPU RAM. Memory requirement (Memory Req.) includes activation size.

on a single GPU. Most methods reduce memory requirements by simplifying the training process, such as lowering the data precision or reducing the number of trainable parameters. These strategies have proven to be efficient without significantly degrading the accuracy of the resulting model.

This paper focuses on fine-tuning on a single consumer-grade GPU. We introduce OFFMATE to reduce memory requirements for LLMs fine-tuning with a very low overhead in training time. OFFMATE efficiently reduces the memory footprint of both activation and weights during training iterations by selectively combining recomputation of some operations and offloading data and computations to the CPU. This combination makes it possible to train an entire network within the memory footprint of a single transformer block. Without changing the data precision or the optimization settings, OFFMATE enables fine-tuning of billion-size models on a consumer-grade GPU with comparable throughput as the original models as shown in Table 1. By preserving the exact numerical results of the training, OFFMATE is compatible with Parameter Efficient Fine-Tuning (PEFT) methods like LoRA [12] that reduce memory requirements by simplifying the training task.

This work presents the following contributions:

- an efficient, fully asynchronous PyTorch framework for full-duplex communication between GPU and RAM, overlapped with independent computations on both GPU and CPU;
- an optimization algorithm based on an Integer Linear Programming formulation, which optimizes over all techniques for reducing memory requirements;
- the OFFMATE tool, which takes any PyTorch model (including from HuggingFace) and with a one-line instruction seamlessly modifies it to fit into memory without approximation;
- an extensive experimental study highlighting the low overhead of our approach compared to the state-of-the-art solutions.

The paper is organized as follows. We present related work on memory-efficient training in Section 2. We describe the motivation and general concept of the present work in Section 3. In Section 4, we propose our algorithm that automatically produces an optimized combination of offloading and re-materialization, resulting in very low computational overhead even for very tight memory budgets. Finally, in Section 5 we demonstrate the excellent performance achieved by OFFMATE.

## 2 Related work

Training Large Language Models (LLMs) is often performed on multiple GPU devices [24, 27], and the memory footprint is distributed across the GPUs. However, since model fine-tuning enables promising applications on edge devices such as personal assistants [11], many memory-efficient memory strategies have been proposed to enable LLM tuning on devices with limited resources.

Parameter Efficient Fine-Tuning (PEFT) methods are often found useful to reduce the memory requirements of LLM tuning. By efficiently simplifying the training task, many algorithms have demonstrated that it is possible to train with substantially lower resource requirements while achieving similar performance as full model fine-tuning. In particular, the LoRA family [12, 10] has shown that using a small fraction of the training parameters can achieve good performance on various tasks. Quantization has also been found useful to significantly reduce memory requirements in LLM training [9, 19]. Other methods include training only the input embedding layer [1], training hidden states [20], and training with a sparse mask over the weights [28].

Another family of memory efficient training does not rely on making the training task simpler. The gradient checkpointing method [8] reduces the memory footprint by deleting intermediate activations and recomputing them during the backward phase. Inspired by gradient checkpointing, other re-materialization strategies [14, 7, 4, 33, 2] have been proposed that rely on optimization algorithms

to build efficient schedules for recomputation. Approaches that offload some activations from GPU to CPU memory have also been found to be useful in both training [25, 24, 6, 3] and inference [26]. A combination of re-materialization and offloading has been proposed [5, 22], but only considering the memory usage of activations. In the present work, we propose to optimize the combination of activation re-materialization with parameter and activation offloading to reduce all parts of the memory usage during the training process. Our solution is closely related to the ZeRO-Infinity solution [24] in the sense that we rely on the same ingredients to reduce memory consumption: offloading, re-materialization and optimization on CPU.

## 3 Motivation

**Memory requirement of large model training** The memory footprint when training a large model consists of two parts: (1) intermediate activations $M_{act}$, whose size depend on the input batch size and which can be rebuilt using re-materialization to reduce the associated peak memory usage; (2) model states, which consist of parameters $M_{param}$, parameter gradients $M_{p\_grad}$, and optimizer states $M_{opt\_st}$. Once the model is selected, the size of $M_{param}$ depends only on the data type of the parameters. The size of $M_{p\_grad}$ is the size of trainable parameters, which can be much smaller than $M_{param}$ when using PEFT methods. The optimizer is the update algorithm which at each iteration computes new values for the parameters based on their gradients; some of these algorithms store data between iterations, like first and second-order momentum for the Adam optimizer [17], and we call this data optimizer states. The size of $M_{opt\_st}$ depends on the choice of the optimizer and is typically a multiplicative factor of $M_{p\_grad}$: for the Adam family of optimizers, the multiplicative factor is 2.

**Memory efficient solutions** Existing memory-efficient approaches in the literature often consider the different sources of memory consumption separately, using specialized algorithms to optimize the decisions. For example, ROCKMATE [33] focuses on reducing $M_{act}$ with a nearly optimal re-materialization strategy. ZeRO-Infinity [25] reduces $M_{opt\_st}$ by storing full-precision optimizer states in CPU RAM and performing Adam optimization steps directly on CPU. In addition to quantization and low-rank approximation, which reduce $M_{param}$ and $M_{opt\_st}$ by reducing precision, Q-LoRA [10] uses Paged Optimizers to swap optimizer states between GPU and CPU RAM while still performing optimization steps on GPU. All PEFT methods that limit the number of trainable parameters can significantly reduce $M_{opt\_st}$.

In this paper, we focus on reducing the memory footprint without any modification to the training result in any way for two reasons: (1) this ensures that the model does not lose any generalization capability compared to the original architecture, so that fine-tuning remains appropriate for all applications where the original model is competitive; (2) for all cases where lower precision and fewer trainable parameters are known to be valid, our solution can be directly applied to this new model and further reduces the memory footprint. Therefore, our proposed solution OFFMATE performs the same training task as the original model and ensures maximum compatibility. OFFMATE leverages the ROCKMATE framework [33] and an Integer Linear Programming (ILP) formulation to efficiently combine all techniques: re-materialization, weight and activation offloading, CPU optimization, and optimizer state paging.

The ROCKMATE framework assumes that the neural network has a *block sequential* structure: the computation is made up of a sequence of blocks, where the inputs of a block are the outputs of the previous blocks. Within each block, arbitrary computational task graphs are supported. In ROCKMATE, the re-materialization optimization problem consists of deciding which activations to delete and which operations to recompute, with the goal of minimizing the overall execution time within a given memory limit. ROCKMATE only minimize the memory used by activations.

**Our approach** In OFFMATE, we adopt the same assumption of a *block sequential* structure that is relevant for many LLMs, but we aim to reduce all sources of memory consumption. To reduce $M_{act}$, we generate multiple re-materialization schedules for each block, using either the RK-CHECKMATE formulation from ROCKMATE or a simple strategy combining re-materialization and activation offloading. Instead of applying the RK-ROTOR solver of ROCKMATE, we combine these schedules with a new linear programming formulation described in Section 4, which incorporates techniques for reducing all other sources of memory consumption.

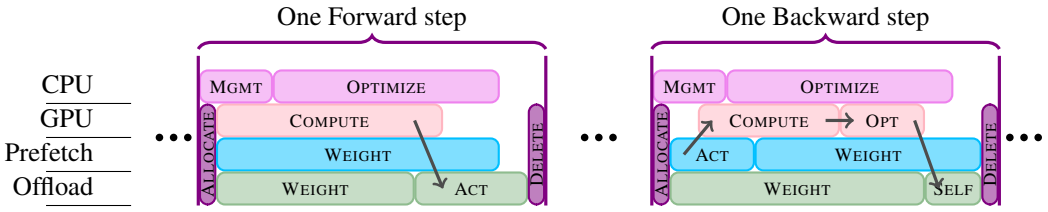


Figure 1: All operations performed in a step. MGMT represents management operations performed on CPU, ACT represents activation offloading or prefetching, and SELF represents offloading the weights of the current layer.

To reduce $M_{param}$, each parameter can be offloaded onto CPU RAM. To reduce $M_{p\_grad}$, each parameter is optimized immediately when the associated gradients are generated and no gradient accumulation is applied. To reduce $M_{opt\_st}$, for parameters which are optimized on GPU, the optimizer states can either be kept on GPU or offloaded to CPU RAM and moved back to GPU before the next optimization step (as proposed by Paged Optimizers [10]).

OFFMATE also considers *CPU optimization*: this allows parameters to be optimized on CPU, in which case optimizer states will always be kept in CPU RAM. This idea was originally proposed in ZeRO-Offload [25], where it was noted that since it is possible to (i) offload the gradient of a parameter, (ii) optimize on the CPU, and (iii) prefetch the new optimized parameter, CPU optimization involves the same amount of data transfers as regular offloading.

The main contribution of OFFMATE is to perform a holistic optimization of all these techniques simultaneously, so that different parameters can be handled by different techniques to minimize the iteration time within a limited memory budget. This combination makes it possible to train the entire model within the memory requirements of a single block: in the context of very tight memory budgets, all data related to other blocks can indeed be offloaded to the CPU RAM. When more memory is available, OFFMATE automatically reduces the amount of offloaded data.

## 4 Method

Using RK-GB introduced in ROCKMATE [33], we implement an efficient data management framework that performs asynchronous full-duplex communication between GPU memory and RAM, overlapped with independent computations on both GPU and CPU.

In this section, we present OFFMATE algorithm based on linear programming that optimizes both the re-materialization decisions to temporarily free memory from activations, and the offloading decisions to reduce memory usage by transferring some model parameters and optimizer states to RAM. We assume that the model has a *block sequential* structure. OFFMATE can incorporate re-materialization schedules of each block of the neural network into a global schedule that can additionally offload the model parameters. Our algorithm relies on a linear programming formulation presented in Section 4.1 and a post-processing phase, presented in Section 4.2, to generate a complete solution.

The solution is a set of *steps* associated to the blocks, where each step performs four operations: executing a forward or backward schedule of a block, offloading some parameters, optimizer states or activations to RAM, prefetching some other previously offloaded data back to GPU memory, and performing an optimization step for some parameters on the CPU. The time required for a step is the maximum time of these four operations. A sketch of the operations performed in a step is provided in Figure 1, highlighting the dependencies between some of these operations, the memory allocation and deletion, and the management overhead incurred on the CPU.

The execution of these steps is *cyclic*: after all steps have been executed, the execution resumes from the first step with a new input batch. This implies that a valid schedule must ensure the consistency of the set of parameters that are offloaded into RAM between the start and end of the schedule. For notational simplicity, when summing values over an interval of time steps, we assume that the index wraps around at 0, so that for $a > b$, $\sum_{t=a}^{b} Y_t$ is $\sum_{t=a}^{2L} Y_t + \sum_{t=0}^{b} Y_t$.

**Assumptions** To make the problem easier to solve, we make the following assumptions:

1. Only one re-materialization schedule is chosen and executed only once for each block. The block thus appears in two steps, for the forward and the backward execution.
2. During the execution of a step, all parameters required by the block are present in the GPU memory for the entire duration of the step.
3. The memory usage of parameters offloaded during a step is only released at the end of that step; symmetrically, the memory usage of parameters prefetched during a step is taken into account from the beginning of the step.
4. It is possible to offload and prefetch an arbitrary fraction of a block's parameters during each step.
5. Optimization of trainable parameters on the GPU is performed immediately after the last gradient has been generated.
6. All operations of activation offloading (resp. prefetching) for a block happen during the corresponding forward (resp. backward) step.
7. No data is deallocated on the CPU: each data stored on the CPU at some point retains its allocated memory space.

Assumption 1 states that the number of steps in the schedule is $2L$, and we know which activations and which parameters are required to be in memory for each step. Assumptions 2 and 3 make it possible to consider offload-oblivious schedules for each block individually. Assumption 4 makes it possible to use continuous variables to represent the fraction of parameters that are prefetched and offloaded at each step. In practice, a parameter in our formulation represents multiple tensors, and we present a grouping algorithm in Section 4.2 to convert these fractional decisions into a group of full tensors to offload. Assumption 5 ensures that we avoid keeping parameter gradients for a long time. Assumption 6 allows us to include activation offloading only at the levels of the sub-schedules, limiting the number of variables in the ILP formulation. Assumption 7 allows us to use pinned memory for efficient and asynchronous transfers between CPU and GPU without incurring costly allocation overhead. In addition, optimizer states for the parameters that are optimized on the CPU remain on the CPU throughout the training.

### 4.1 OFFMATE formulation

**Problem statement** The input to the optimization problem is a set of $L$ blocks, and for each block $i$, (1) a list of $n_i$ re-materialization and activation offloading schedules (called *options*) without taking into account the memory cost of the parameters, and (2) the set $P_i$ of all parameters used by that block. We distinguish between *interface activations*, which are passed from one block to another, and *internal activations*, which are stored in the forward phase of a block to be used during the backward phase. Each option $o$ of block $i$ may store on the GPU a different set of internal activations whose memory usage is denoted as $S_i^o$, and may offload another set of internal activations whose memory usage is denoted as $O_i^o$. We denote by $T_t^o$ the computation time of option $o$ of the $t$-th computation (the forward computation if $t \leq L$, otherwise the backward computation). For options that involve activation offloading, this computation time also includes the delay incurred by waiting for communications to complete (offload for forward computations, prefetch for backward computations).

The total size of a parameter $w$ is denoted by $|w|$, and the size of its trainable part is denoted by $|w|_g$. If several tensor parameters are used by the same blocks, they are considered together as a single parameter, where some parts may be trainable and others not. We denote by $B(t)$ and $A_t$ the executed block and the size of the interface activations and gradients present in memory during step $t$. For a given parameter $w$, we refer to $f_w$ as the first step that makes use of $w$, and to $g_w$ as the step that computes the last gradient related to $w$.

We denote with $\alpha_G$ and $\alpha_C$ the update speeds of optimizing a parameter on the GPU and on the CPU respectively. We denote with $\beta$ the bandwidth of the communication link between the CPU and the GPU, and with $H$ the time required for the CPU to handle the management of all the other operations in the step (submitting the kernels to the GPU), which we estimate as a constant in all steps.

To generate activation-offloading schedules, we use a simple strategy based on *cheap* operations, inspired by the *selective recomputations* from Megatron-LM [27]. We identify operations whose computational load is smaller than the time required to offload and prefetch their output data. These *cheap* operations are recomputed, and all other activations are selected for offloading, resulting in an option $o$ with $S_i^o = 0$.

**Linear Programming Formulation** The formulation involves the following variables:

$$\forall i \leq L, \forall o \leq n_i, \quad Comp_i^o \in \{0,1\} \tag{1}$$
$$\forall t \leq 2L, \quad Time_t \geq 0 \tag{2}$$
$$\forall t \leq 2L, \forall w, \quad Sto_t^w, Ofl_t^w, Prf_t^w, StoO_t^w, OflO_t^w, PrfO_t^w \in [0,1] \tag{3}$$
$$\forall w, \forall f_w < t < g_w, Opt_t^w \in [0,1] \tag{4}$$

$Comp_i^o$ is 1 if block $i$ is executed with option $o$, and 0 otherwise. These variables satisfy $\forall i, \sum_o Comp_i^o = 1$. $Time_t \geq 0$ represents the duration of step $t$. Finally, $Sto_t^w$, $Ofl_t^w$, $Prf_t^w$, $Opt_t^w$ are the fractions of parameter $w$ which are respectively stored on GPU, offloaded, prefetched and optimized on CPU during step $t$. $StoO_t^w$, $OflO_t^w$ and $PrfO_t^w$ represent similar decisions on the optimizer states linked to $w$. We denote as $X_w = \sum_{t=g_w}^{f_w} Opt_t^w$ the fraction of parameter $w$ which are optimized on CPU at some point, which is also the fraction of optimizer states stored on the CPU.

We now enumerate the constraints. First, the computation of a block $i$ requires all its parameters (5) and the optimization is performed with the last backward step (6).

$$\forall t \leq 2L, \forall w \in P_{B(t)}, \; Sto_t^w \geq 1 \tag{5}$$
$$\forall w, \; StoO_{g_w}^w \geq 1 - X_w \tag{6}$$

Second, parameters and optimizer states not optimized on CPU must either be in the GPU or offloaded (7, 8), and bringing a parameter or optimizer state back to the GPU requires prefetching it (9, 10).

$$\forall t, w, \; Sto_t^w + \sum_{t'=g_w}^{t} Ofl_{t'}^w \geq 1 \tag{7}$$
$$StoO_t^w + \sum_{t'=g_w}^{t} OflO_{t'}^w \geq 1 - X_w \tag{8}$$
$$\forall t, w, \; Sto_{t+1}^w \leq Sto_t^w + Prf_t^w \tag{9}$$
$$StoO_{t+1}^w \leq StoO_t^w + PrfO_t^w \tag{10}$$

Third, performing the optimization on CPU requires fetching the optimized part of the parameter (11), the number of parameters optimized on CPU is at most the number of offloaded parameters (12), and parameters waiting to be optimized cannot be prefetched (13).

$$\forall w, \; |w|_g \sum_{t'=g_w}^{f_w} Opt_{t'}^w \leq |w| \sum_{t'=g_w}^{f_w} Prf_{t'}^w \tag{11}$$
$$\forall t, \forall w, \; |w|_g \sum_{t'=g_w}^{t} Opt_{t'}^w \leq |w| \sum_{t'=g_w}^{t} Ofl_{t'}^w \tag{12}$$
$$\forall t, \forall w, \; |w|_g (X_w - \sum_{t'=g_w}^{t} Opt_{t'}^w) \leq |w|(1 - Prf_t^w - Sto_t^w) \tag{13}$$

We denote by $k$ the number of optimizer states per trainable value, which depends on the optimizer. We can express the global memory constraint as (17), where $W_t$, $O_t$, and $S_t$ represent respectively the memory usage during step $t$ of parameters, optimizer states, and stored internal activations:

$$W_t = \sum_w |w| \left(Sto_t^w + Prf_t^w\right) \tag{14}$$
$$O_t = \sum_w k|w|_g \left(StoO_t^w + PrfO_t^w\right) \tag{15}$$
$$S_t = \sum_{i \leq B(t)} \sum_o S_i^o \cdot Comp_i^o \tag{16}$$
$$\forall t, \; W_t + O_t + S_t + A_t \leq M_{GPU} \tag{17}$$

We also express a memory constraint for the memory of the CPU. It contains all the parameters of the model, the offloaded activations and optimizer states, as well as the optimizer states and gradient of each parameter optimized on CPU:

$$\sum_w |w| + \sum_{i,o} O_i^o \cdot Comp_i^o + \sum_w |w|_g \sum_t \left(k \cdot OflO_t^w + (k+1) \cdot Opt_t^w\right) \leq M_{CPU} \tag{18}$$

To avoid GPU idle time, we allow optimization operations on the CPU to be performed during the training iteration, overlapping with computation and communication operations. Our formulation also allows a parameter to be offloaded in the same step as the one it is used, but these operations cannot overlap. By denoting as $L_t$ the time spent offloading parameters and optimizer states for the block of step $t$, we can express the time of step $t$ as:

$$L_t = \frac{1}{\beta} \sum_{w \in P_{B(t)}} |w| Ofl_t^w + |w|_g OflO_t^w$$

$$\text{(GPU Fwd)} \quad \forall t \leq L, Time_t \geq \sum_o Comp_{B(t)}^o T_t^o \tag{19}$$
$$\text{(GPU Bwd)} \quad \forall t > L, Time_t \geq \sum_o Comp_{B(t)}^o T_t^o + \frac{1}{\alpha_G} \sum_{w \in P_{B(t)}} (1 - X_w)|w|_g + L_t \tag{20}$$
$$\text{(Prefetch)} \quad \forall t, Time_t \geq \frac{1}{\beta} \sum_w \left(|w| Prf_t^w + k|w|_g PrfO_t^w\right) \tag{21}$$
$$\text{(Offload)} \quad \forall t, Time_t \geq \frac{1}{\beta} \sum_w \left(|w| Ofl_t^w + k|w|_g OflO_t^w\right) \tag{22}$$
$$\text{(CPU)} \quad \forall t, Time_t \geq H + \frac{1}{\alpha_C} \sum_w |w|_g Opt_t^w \tag{23}$$

The objective is then to minimize the overall duration $\sum_t Time_t$.

**Batch size selection** This formulation, like all other approaches for re-materialization, assumes that the batch size is fixed, so that the forward and backward computation times are measured using samples of real size. We present in Appendix A a variant of OFFMATE ILP which also considers batch size as a variable to be optimized, under the assumption that computation times depend linearly on the batch size. Users interested in fine-tuning tasks which are not sensitive to batch size can thus enable batch size tuning to achieve better overall throughput.

### 4.2 Post-processing

The schedule obtained with ILP is feasible because it fits in memory, but Assumption 4 assumes that we can deallocate an arbitrary fraction of a tensor's memory. Doing this in practice requires copying the other part to a buffer. However, allocating this buffer and copying the data induces a significant overhead, both in memory and execution time. For an efficient execution, we decide to offload and prefetch only full tensors, and rely on the fact that in most models, the parameters of a block are stored in multiple tensors. The goal of the post-processing step is, for each parameter $w$ and each time step $t$, to select a subset of tensors from $w$, ensuring that at each step, we offload and delete at least as much memory as required by the ILP solution, and prefetch at most what the solution requires.

This is done with a greedy algorithm which repeatedly solves a particular case of the knapsack problem: given a set of tensors and a target value, find a subset whose size is larger but as close as possible to the target value. The algorithm starts with step $g_w$ where all tensors are present in GPU memory, and successively selects tensors to offload so that the total offloaded size at step $t$ is at least $|w| \sum_{t'=g_w}^{t} Ofl_t^w$, and symmetrically for the prefetched tensors. Then tensors for CPU optimization are selected, and finally we run the same algorithm for optimizer states. Because of space limitations, we provide further details in Appendix B. This results in a new solution whose memory usage is never larger than the memory usage of the original ILP solution, and which only transfers entire tensors.

## 5 Experiments

In this section, we evaluate the performance of OFFMATE on real fine-tuning scenarios. OFFMATE relies on `torch.export()` to obtain the task graph of the model. All experiments are run on PyTorch 2.3 and LLMs are loaded from HuggingFace Transformers v4.36. To avoid memory fragmentation issues, we set the target memory budget to 2GB less than the total available GPU memory for ILP solving. Unless otherwise specified, all experiments are performed on a computer with an NVIDIA RTX 3060 GPU and an Intel Core i3-10105F CPU, where the GPU-to-CPU bandwidth is measured to be 10.4GB/s. The Integer Linear Program is solved using the default solver provided in the open source PULP library[1]. For all experiments in this section, we use three options for each block and limit the ILP solving time to 20 minutes to provide ready-to-use results. We also assume all the trainable parameters are updated with the Adam optimizer [17] during each iteration.

**Fine-tuning tasks** In this experiment, we study the performance of OFFMATE on four fine-tuning tasks: Phi-2(2.7B) [15] and Llama2(7B) [29] in floating-point 32 precision, Llama2(13B) in bfloat16, and Llama2(7B) with LoRA [12]. We evaluate the performance of OFFMATE compared to the following approaches:

1. PyTorch (Extrapolated): assuming the peak memory and iteration time of PyTorch execution depends linearly on the number of hidden layers, we measure the PyTorch execution on small number of layers until it is out of memory, then extrapolate to predict the expected result on the full-size model.
2. Paged Optimizers proposed in QLoRA [10], using the implementation from HuggingFace Trainer. This experiment does not include quantization or low-rank adapters, only swapping the optimizer states between GPU and RAM.
3. ZeRO-2 [23] which uses ZeRO stage 2 and offload the optimizer states to RAM; ZeRO-Infinity [24] which uses ZeRO stage 3 to offload optimizer states and parameters, with

[1] https://github.com/coin-or/pulp

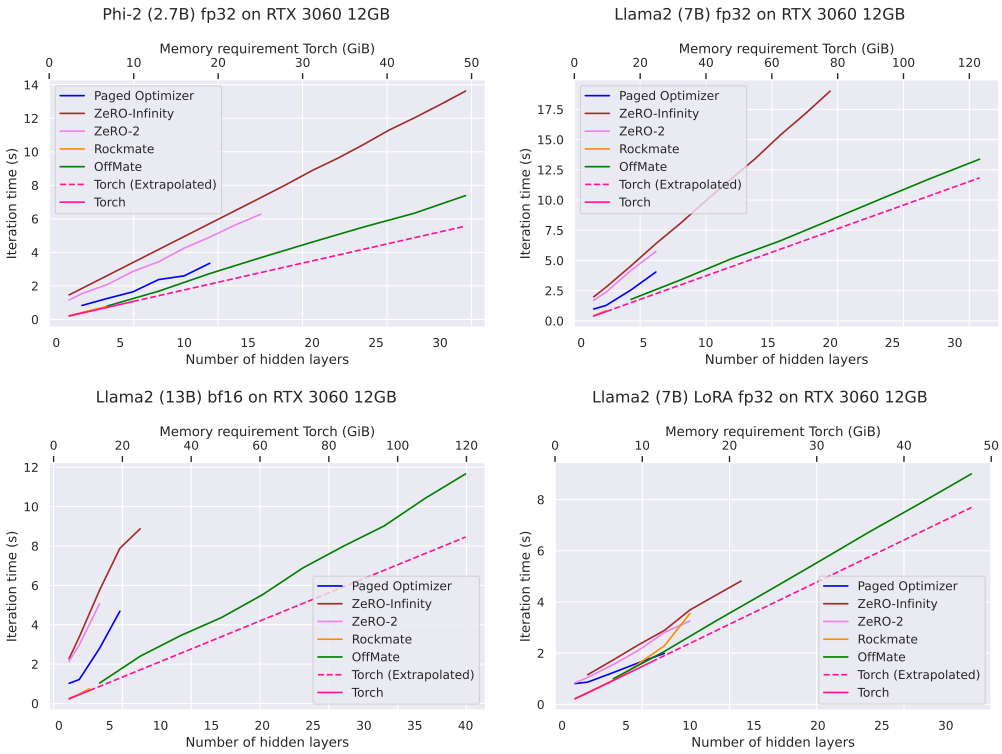


Figure 2: Average iteration time vs. number of hidden layers (full size models, with 32 or 40 layers, correspond to the right-most point in each plot). Batch size is 4 and sequence length is 512. The extrapolated PyTorch memory usage is marked on top of each graph. With LoRA, less than $1\%$ parameters are trainable thus $M_{p\_grad}$ and $M_{opt\_st}$ are negligible.

default configuration. NVMe offload is not used in our experiments. Gradient checkpointing is also enabled through HuggingFace Trainer.

4. Rockmate [33] which can significantly reduce the memory usage from activations.

Since not all approaches can perform fine-tuning on the complete model, we present on Figure 2 the iteration time as a function of the number of hidden layers included in the model, where the right-most point corresponds to the full-size model. Figure 2 shows that OFFMATE is able to perform all the fine-tuning tasks with significantly lower time overhead. Specifically, OFFMATE is able to fine-tune a full-size Llama2 (7B) on a consumer-grade GPU, reducing the memory usage from 120 GB to 10 GB with only $13.7\%$ overhead in the execution time comparing to the expected time without memory constraint.

Paged Optimizer, ZeRO-2 and Rockmate do not reduce all sources of memory and thus cannot perform fine-tuning as soon as the model size gets too big. ZeRO-Infinity has a more aggressive approach where all data are indiscriminately offloaded; this enables processing larger models, but induces a significant time cost: the time overhead of ZeRO-Infinity can reach more than 200%. It also induces Out of memory (OOM) on CPU RAM as shown in Table 1. OFFMATE avoids this problem by applying the CPU RAM constraint 18 in the ILP formulation.

**Step visualization** We provide on Figure 3 a trace obtained from `torch.profiler` for the execution of one backward step on Llama2 (7B), to be compared with the theoretical Figure 1. This figure highlights how OFFMATE is able to efficiently overlap both computation and communication to limit the idle time in practice. We also provide in Appendix D a trace of the complete execution of all steps, which shows the high resource utilization and the similarity between the expected schedule and the actual execution. A CPU management time of H=50ms is used in ILP constraint 23.

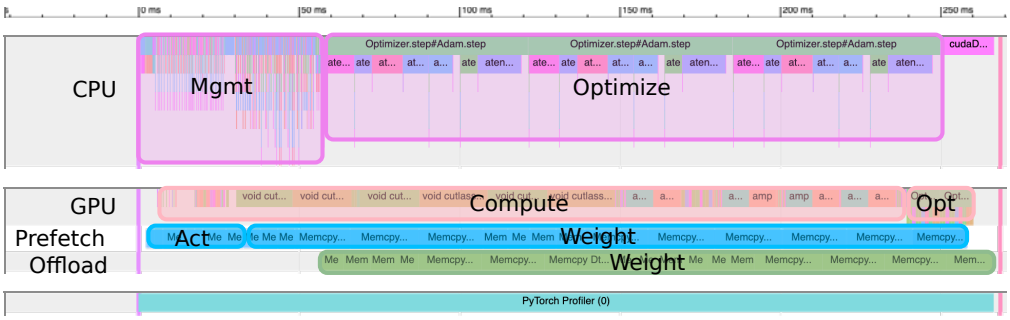


Figure 3: Annotated trace of a single backward step from Llama2 (7B).

**Ablation study** In Table 2 we show the ablation study about the performance of OFFMATE on Llama2 (7B) on two different machines. Specifically, we compare the results when CPU optimization or activation offload are disabled. With the same model size, fitting all the parameters in memory is much harder for the first machine. It explains why both CPU optimization and activation offloading are crucial to achieve the best performance of OFFMATE. For the second machine, due to the larger difference between CPU and GPU optimization speed, the gain from applying CPU optimization is smaller.

| **GPU:VRAM** RTX 3060 12GB | **CPU:RAM** Intel i3-10105F 128GB | **Bandwidth** 10.4GB/s | $\alpha_C$ 1.1GB/s | $\alpha_G$ 16.6GB/s |
|---|---|---|---|---|
| **Method** | **RAM usage** | **VRAM usage** | **Time** | **Overhead** |
| OFFMATE | 99.0 GB | 9.9 GB | 13.5 s | 13.7% |
| w/o CPU optim | 88.4 GB | 9.1 GB | 14.4 s | 21.6% |
| w/o Act offload | 96.2 GB | 10.0 GB | 15.7 s | 32.3% |
| **GPU:VRAM** RTX 3090 24GB | **CPU:RAM** Ryzen 3975WX 256GB | **Bandwidth** 18.8GB/s | $\alpha_C$ 1.6GB/s | $\alpha_G$ 40.9GB/s |
| **Method** | **RAM usage** | **VRAM usage** | **Time** | **Overhead** |
| OFFMATE | 111.3 GB | 21.7 GB | 10.3 s | 24.3% |
| w/o CPU optim | 107.2 GB | 21.2 GB | 10.6 s | 27.4% |
| w/o Act offload | 103.5 GB | 21.2 GB | 11.8 s | 42.3% |

Table 2: Comparing the results with Llama2 (7B) on two different machines (top and bottom). $\alpha_C$ and $\alpha_G$ represents the optimization speed on CPU and GPU. Input size is $(4, 512)$ for the first machine and $(8, 512)$ for the second machine, which cost 40.8GB and 81.7GB of activation size. Overhead is defined as the extra measured iteration time divided by PyTorch (Extrapolated) time shown in Figure 2. RAM usage does not include the system usage.

## 6 Limitations

OFFMATE does not support NVMe offload as ZeRO-Infinity. As shown in Table 2, the RAM size could become a bottleneck to fine-tune very large models. OFFMATE does not perform any approximation in training, but optimizing on CPU may change the results depending on the precision. Indeed, due to differences in machine computation, the same task running on CPU or GPU may produce slightly different results, and we confirm that the results obtained by OFFMATE are within this range. Furthermore, the ILP of OFFMATE requires that all parameters of a block are available when it is executed. A more fine-grained structure within blocks could help reduce the minimum memory required by offloading in the middle of block computations. Finally, we rely on a number of assumptions to limit the solving time of the ILP formulation. Different trade-offs between realism and tractability could be explored and may result in more efficient solutions.

## 7 Conclusion

This paper presents OFFMATE, a framework for efficiently reducing memory requirements when fine-tuning Large Language Models on a single consumer-grade GPU. OFFMATE relies on integer linear programming to combine different memory-reducing approaches in a holistically optimized manner, ensuring efficient use of all available resource (computing, storage, communication). Experiments show that OFFMATE significantly outperforms the State-of-the-Art approaches and enables efficient fine-tuning of LLMs from HuggingFace Transformers. For Llama2 (7B) model, OFFMATE achieves a $10\times$ reduction in GPU memory at the cost of a $13.7\%$ increase in training time, without modifying the training task (e.g., lowering data precision or reducing the number of trainable parameters). We also show that OFFMATE can be combined with parameter-efficient fine-tuning methods such as LoRA. OFFMATE is an easy-to-use framework that can enable fine-tuning on resource-constrained machines and is expected to have a significant impact for individual AI researchers. Future research efforts could add support for gradient accumulation or allow fine-grain offloading within a block to further reduce the minimum memory requirement.

## References


[1] S. An, Y. Li, Z. Lin, Q. Liu, B. Chen, Q. Fu, W. Chen, N. Zheng, and J.-G. Lou. Input-tuning: Adapting unfamiliar inputs to frozen pretrained models. *arXiv preprint arXiv:2203.03131*, 2022.

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

[3] O. Beaumont, L. Eyraud-Dubois, and A. Shilova. Optimal gpu-cpu offloading strategies for deep neural network training. In *European Conference on Parallel Processing*, pages 151–166. Springer, 2020.

[4] O. Beaumont, J. Herrmann, G. Pallez, and A. Shilova. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A*, 378(2166):20190049, 2020.

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

[6] O. Beaumont, L. Eyraud-Dubois, A. Shilova, and X. Zhao. Weight Offloading Strategies for Training Large DNN Models. working paper or preprint, Feb. 2022. URL `https://hal.inria.fr/hal-03580767`.

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

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

[9] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[11] X. L. Dong, S. Moon, Y. E. Xu, K. Malik, and Z. Yu. Towards next-generation intelligent assistants leveraging llm techniques. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5792–5793, 2023.

[12] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[13] Q. Huang, M. Tao, Z. An, C. Zhang, C. Jiang, Z. Chen, Z. Wu, and Y. Feng. Lawyer llama technical report. *arXiv preprint arXiv:2305.15062*, 2023.

[14] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization, 2019.

[15] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi, et al. Phi-2: The surprising power of small language models. *Microsoft Research Blog*, 2023.

[16] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[17] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] Y. Li, S. Wang, H. Ding, and H. Chen. Large language models in finance: A survey. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pages 374–382, 2023.

[19] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[20] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022.

[21] R. OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2:13, 2023.

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

[23] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[24] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. 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*, pages 1–14, 2021.

[25] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[26] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[27] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[28] Y.-L. Sung, V. Nair, and C. A. Raffel. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*, 34:24193–24205, 2021.

[29] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

[31] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[32] C. Wu, X. Zhang, Y. Zhang, Y. Wang, and W. Xie. Pmc-llama: Further finetuning llama on medical papers. *arXiv preprint arXiv:2304.14454*, 2023.

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

# A Batch size selection

**Changes to the formulation** In the OFFMATE ILP formulation, like in all other approaches for re-materialization in the literature, we assume that the batch size is fixed, so that the forward and backward computation times are measured using samples of real size. However, we can observe that only $T_t^o$, $S_t^o$ and $O_t^o$ are affected by the batch size of the training task. All parameter-related operations (offload, prefetch, optimization, update) only depend on the model size and the bandwidth between the GPU and RAM.

We now present an adaptation of OFFMATE to choose the best batch size by making the appropriate assumption that forward and backward operations take time proportional to the batch size. This adaptation is targeted towards some fine-tuning tasks which are not sensitive to batch size, whose users may allow batch size tuning to achieve better throughput.

The natural way to do this would be to add a variable $b$ to the ILp formulation and multiply all computation times by $b$. However, this would result in a quadratic (non linear) formulation. See for example the global memory constraint 17, which would become:

$$\forall t,\, W_t + O_t + b \sum_{i \leq B(t)} \sum_o S_i^o \cdot Comp_i^o + b \cdot A_t \leq M_{GPU},$$

where both $b$ and $Comp_i^o$ are variables in the formulation. Instead, we change the scale and divide all parameter-related times and memory by $b$. For this equation, this yields:

$$\begin{aligned} \forall t,\, \tfrac{1}{b} W_t + \tfrac{1}{b} O_t + \textstyle\sum_{i \leq B(t)} \sum_o S_i^o \cdot Comp_i^o + \cdot A_t \leq \tfrac{1}{b} M_{GPU}, \\ \text{where} \quad \tfrac{1}{b} W_t = \textstyle\sum_w |w| \left( \tfrac{1}{b} Sto_t^w + \tfrac{1}{b} Prf_t^w \right) \end{aligned}$$

This can be done while keeping a linear formulation by introducing a continuous variable $r \in [0, 1]$, which represents $\frac{1}{b}$ and is interpreted as a fraction of the parameters to be used. We then view all parameter-related variables as bounded in $[0, r]$ instead of $[0, 1]$: instead of writing $\frac{1}{b} Sto_t^w$ with $Sto_t^w \in [0, 1]$, we write $Sto_t^w$ with $Sto_t^w \in [0, r]$. The resulting formulation remains linear, with nearly the same constraints as before.

Some constraints where the constant 1 is used to represent the entirety of the parameters need to be modified, as follows:

$$\forall t \leq 2L, \forall w,\, Sto_t^w, Ofl_t^w, Prf_t^w, StoO_t^w, OflO_t^w, PrfO_t^w \leq r \tag{3'}$$

$$\forall w, \forall f_w < t \leq g_w,\, Opt_t^w \leq r \tag{4'}$$

$$\forall w \in P_{B(t)},\, Sto_t^w \geq r - X_w \tag{5"}$$

$$\forall w,\, StoO_{g_w}^w \geq r \tag{6"}$$

$$\forall t,\, Sto_t^w + \textstyle\sum_{t'=g_w}^{t} Ofl_{t'}^w \geq r \tag{7"}$$

$$\forall t,\, StoO_t^w + \textstyle\sum_{t'=g_w}^{t} OflO_{t'}^w \geq r - X_w \tag{8"}$$

$$\forall t, \forall w,\, |w|_g (X_w - \textstyle\sum_{t'=g_w}^{t} Opt_{t'}^w) \leq |w|(r - Prf_t^w - Sto_t^w) \tag{13'}$$

$$\forall t,\, W_t + O_t + S_t + A_t \leq M_{GPU} \cdot r \tag{17"}$$

$$\forall t > L,\, Time_t \geq \textstyle\sum_o Comp_{B(t)}^o T_t^o + \frac{1}{\alpha_G} \sum_{w \in P_{B(t)}} (r - X_w) |w|_g + L_t \tag{20'}$$

**Experimental evaluation** Figure 2 was obtained with a fixed batch size of 4 to simplify the comparison. In this experiment, we evaluate how OFFMATE is able to maximize throughput by choosing a suitable batch size for the given task. Figure 4 displays the throughput obtained with OFFMATE on a fully-trainable Phi-2 model with different batch sizes, where the optimal batch size selected by OFFMATE is shown as the red dot. The difference between the measured and the scheduled throughput is due to CUDA synchronization, whose cost is more significant when then computation time is smaller. The difference between the scheduled throughput and the ILP solution is due to the post-processing described in Section 4. The OFFMATE formulation underestimates the overlap between the GPU backward and optimization, which explains why the throughput of the ILP solution is lower than the real case. These differences make it challenging for the ILP to obtain the optimal batch size, but we can see that in practice the best throughput is obtained for a batch size between 8 and 12, and that the solution returned by the ILP is within this interval.

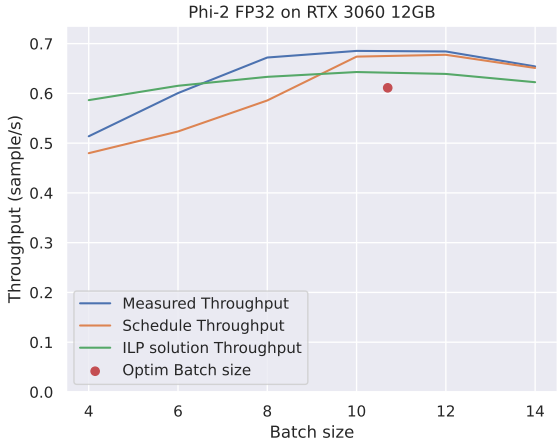


Figure 4: Throughput vs. batch size for the Phi-2 model (2.7B). Optimal batch size selected by OFFMATE is shown as a red dot.

Note that the throughput of OFFMATE is not strictly monotonically increasing with the batch size, since larger batch size increases the memory pressure and requires more re-materialization or offload and prefetch overhead.

## B Post-processing

As mentioned in Section 4.2, the solution from the Linear Programming formulation may specify arbitrary fractions of parameters to be offloaded and prefetched at each step. OFFMATE performs a post-processing step to select a subset of tensors from each parameter $w$, ensuring that at each step, we offload and delete at least as much memory as required by the OFFMATE ILP solution, and prefetch at most what the solution requires.

Algorithm 1 presents the main algorithm of this post-processing step. It is based on a SELECT function, which solves a particular case of the knapsack problem: given a set of tensors and a target value, find a subset whose size is larger but as close as possible to the target value. This problem can be solved efficiently and optimally with dynamic programming.

For a given parameter $w$, Algorithm 1 starts with step $g_w$ where all tensors are present in GPU memory, and successively selects tensors to offload so that the total offloaded size at step $t$ is at least $|w| \sum_{t'=g_w}^{t} Ofl_t^w$. At each step, if some memory need to be freed, some tensors are chosen from the set of already offloaded but not yet deleted tensors, again ensuring that at least the required amount of data is deleted. On the contrary, if the amount of available data is too low, some tensors are chosen from the set of deleted tensors, ensuring that at most the required amount of data is prefetched. This results in a new solution whose memory usage is never larger than the memory usage of the original ILP solution, and which only transfers entire tensors.

Once Algorithm 1 has selected which tensors to offload and prefetch, we identify candidate tensors for CPU optimization: those that are offloaded to the CPU after the backward computation, and are prefetched before the forward computation. We use the SELECT function once to globally select a sufficiently large set of tensors that will be optimized on CPU. Once this is done for all parameters $w$, the optimization operations of the selected tensors are greedily scheduled into the time steps in order of increasing block index, since blocks with lower index have a smaller range between backward and forward computation.

Finally, we have to select which optimizer states to offload and prefetch. For this purpose, we run Algorithm 1 on optimizer states, with $StoO_t^w$ and $OflO_t^w$ as input, and where $\mathcal{T}_w$ is the set of optimizer states that have not been selected for optimization on CPU.

**Algorithm 1** Greedy grouping for parameter $w$

**Input:** $Sto_t^w, Ofl_t^w$ for all $t$, $\mathcal{T}_w$ = set of all tensors in $w$
**Output:** *Offload*, *Prefetch*, *Delete*
def SELECT(set, value) = knapsack()
Initialize $ofl = \emptyset$
Initialize $alive = \mathcal{T}_w$
**for** $t \in steps$ **from** $g_w$ **do**
  **if** $|w| \cdot \sum_{t'=g_w}^{t} Ofl_{t'}^w > |ofl|$ **then**
    set = $\mathcal{T}_w \setminus ofl$
    *Offload*[t] = SELECT(set, $|w| \cdot \sum_{t'} Ofl_{t'}^w - |ofl|$)
    $ofl = ofl \cup$ *Offload*[t]
  **end if**
  **if** $|w| \cdot Sto_t^w < |alive|$ **then**
    set = $ofl \cap alive$
    *Delete*[t] = SELECT(set, $|alive| - |w| \cdot Sto_w^t$)
    $alive = alive \setminus$ *Delete*[t]
  **end if**
  **if** $|w| \cdot Sto_t^w > |alive|$ **then**
    set = $\mathcal{T}_w \setminus alive$
    kept = SELECT(set, |set| $- |w| \cdot Sto_w^t + |alive|$)
    *Prefetch*[t] = set $\setminus$ kept
    $alive = alive \cup$ *Prefetch*[t]
  **end if**
**end for**

## C Grouping of CPU memory allocations

In the ILP formulation of Section 4, Constraint 18 expresses the constraint for CPU memory usage. The left side of Constraint 18 only contains data that need to be present in memory during an unmodified PyTorch execution, so the total amount of memory cannot be higher than the extrapolated memory usage on an infinite-memory GPU. However, we observed in practice that the CPU memory usage can be higher. This is due to the way PyTorch allocates pinned memory on CPU: to increase the possibility of reusing pinned memory buffers, PyTorch rounds up the data size to the next power of 2, which leads to high memory overhead in allocation when the desired tensor size is not a power of 2.

To obtain the best possible communication performance in OFFMATE, all the offloading is realized by moving data to a pinned memory buffer on the CPU memory, which is not reallocated between iterations. Indeed, unlike parameter gradients and activations which may appear in different time periods on GPU memory, all the CPU memory allocations are performed at the first iteration and never released until the training is over. This PyTorch behavior is thus not useful for OFFMATE, and can incur a significant CPU memory overhead. For example, Llama2 (7B) with float32 and Llama2-13B with bfloat16 from the experiments described in Figure 2 both result in out-of-memory problems on 128GB RAM.

To overcome this issue, we include a heuristic in OFFMATE to reduce this memory overhead on CPU. Except for the parameters scheduled to be optimized by CPU, all other CPU allocations correspond to tensors that need to be offloaded from the GPU: they are not involved in computation but merely used for storing the data. Therefore, we analyze the sizes of all the required CPU allocations and merge them with a heuristic algorithm to obtain groups of size almost $2^n$, and directly ask PyTorch to allocate the grouped data. All offload and prefetch operations are then performed with the corresponding parts of the allocations on CPU. This heuristic grouping allows OFFMATE to use at most as much CPU memory as an original, unmodified PyTorch execution on CPU.

## D Full execution trace

We show on Figure 5 a trace of the execution of a complete iteration for the Llama2 (7B) model (forward and backward passes). The top part of the figure displays the expected schedule produced

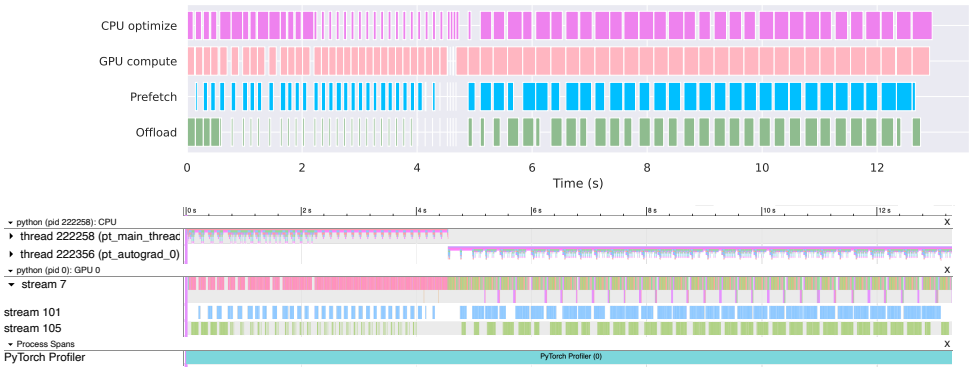


Figure 5: Visualization of the simulated operations schedule on Llama2 (7B), corresponding to the real measurement shown on Figure 2 with 32 layers. The top figure shows the simulated time cost of different operations during each step, and the bottom one shows the traced results with `torch.profiler`.

by the OFFMATE optimization algorithm with each step visible, and the bottom part shows the trace obtained with `torch.profiler`. This trace highlights the high resource utilization along the entire iteration and the similarity between the expected schedule and the actual execution.