Accelerating the Training of Large Language Models using Efficient Activation Rematerialization and Optimal Hybrid Parallelism

Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang, *Kuaishou Technology*

https://www.usenix.org/conference/atc24/presentation/yuan

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Accelerating the Training of Large Language Models using Efficient Activation Rematerialization and Optimal Hybrid Parallelism

Tailing Yuan, Yuliang Liu*, Xucheng Ye, Shenglong Zhang, Jianchao Tan,
Bin Chen, Chengru Song, and Di Zhang

Kuaishou Technology, Beijing, China

Abstract

Recent advancements in training large-scale models have centered on optimizing activation strategies and exploring various parallel training options. One research avenue focuses on enhancing activation-related operations, such as offloading and recomputing. However, there is room for further refinement in these strategies to improve the balance between computation and memory utilization. Another line of work explores different training parallelisms, which often require extensive parameter tuning and achieve suboptimal combinations of parallel options.

To tackle these challenges, this paper introduces a novel method for losslessly accelerating the training of large language models. Specifically, two efficient activation rematerialization strategies are proposed: Pipeline-Parallel-Aware Offloading, which maximizes the utilization of host memory for storing activations, and Compute-Memory Balanced Checkpointing, which seeks a practical equilibrium between activation memory and computational efficiency. Additionally, the paper presents an extremely efficient searching method for optimizing parameters for hybrid parallelism, considering both offloading and checkpointing to achieve optimal performance. The efficacy of the proposed method is demonstrated through extensive experiments on public benchmarks with diverse model sizes and context window sizes. For example, the method significantly increases Model FLOPs Utilization (MFU) from 32.3% to 42.7% for a 175B Llama-like model with a context window size of 32,768 on 256 NVIDIA H800.

1 Introduction

Large language models (LLMs) [3, 5, 27, 28] have emerged as powerful tools capable of transforming various domains, including natural language processing, machine translation, and conversational AI applications. These models, however, require extensive computational resources and memory, particularly during training, necessitating the development of strategies to accelerate the training process. Existing work has focused on optimizing various activation strategies and exploring different parallel training options.

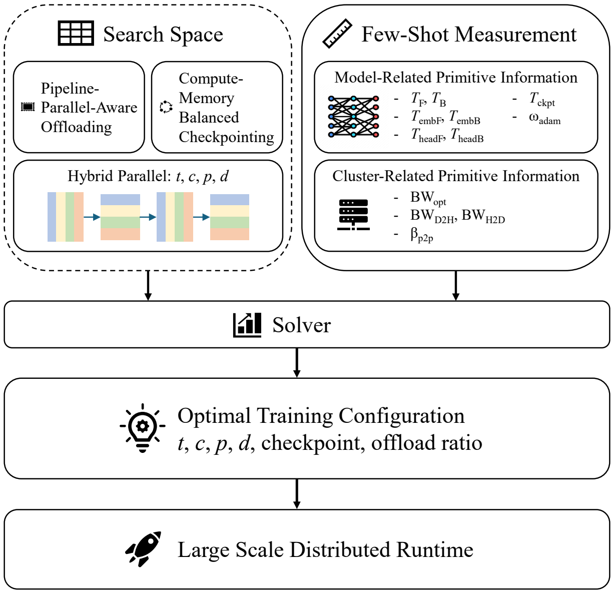

Search Space
Pipeline-Parallel-Aware Offloading
Compute-Memory Balanced Checkpointing
Hybrid Parallel: t, c, p, d
Few-Shot Measurement
Model-Related Primitive Information
- T_F, T_B
- T_{embF}, T_{embB}
- T_{headF}, T_{headB}
- T_{ckpt}
- ω_{adam}
Cluster-Related Primitive Information
- BW_{opt}
- BW_{D2H}, BW_{H2D}
- β_{p2p}
Solver
Optimal Training Configuration
t, c, p, d, checkpoint, offload ratio
Large Scale Distributed Runtime

Figure 1: Overview of our system. Hybrid parallel variables t, c, p, d stand for tensor, context, parallel, and data parallelism sizes. Variables under model-related and cluster-related primitive information are defined in Table 7. Model primitives can be measured with a single layer and cluster primitives can be obtained with only a few nodes. The configuration generated from our system gives optimal performance in real large distributed training runtime.

In the realm of checkpointing, methods such as [4, 13] have been designed to save memory by swapping computing resources, thus minimizing memory usage on a single GPU. However, these methods are either limited to networks with residual connections due to the linearization assumption or suffer from high search complexity and sensitivity to parallel configurations, resulting in high search overhead and difficulty

[*] Corresponding author: Yuliang Liu (liuyuliang@kuaishou.com)

in direct application to large models.

Current approaches to hybrid parallelism tuning for large models can be categorized into two main strategies. One strategy [12, 32, 36] employs mathematical modeling and optimization techniques, such as dynamic programming or integer linear programming, to identify the optimal parallel configuration. This method assumes a direct proportionality between computational costs and the amount of computation or communication, which may not always be accurate. The other strategy [8] involves a brute-force enumeration of parallelism parameters. While this approach circumvents the complexities of modeling, it is resource-intensive due to the extensive experimentation needed to evaluate numerous parameter combinations.

To address these challenges, this paper introduces a novel approach to losslessly accelerate the training of large language models by optimizing activation strategies and exploring optimal hybrid parallel training, as shown in Figure 1. Specifically, we propose two efficient activation rematerialization strategies: Pipeline-Parallel-Aware Offloading, which maximizes the utilization of host memory for storing activations, and Compute-Memory Balanced Checkpointing, which seeks a practical equilibrium between activation memory and computational efficiency.

Our Pipeline-Parallel-Aware Offloading algorithm addresses the challenge of efficiently managing the organization of activations. It operates on the principle that the host-to-device bandwidth may be a limiting factor in offloading, and thus, it carefully schedules offloading and reloading of activations to minimize performance overhead. It follows a scheduling granularity of "pipeline stages", ensuring that the time to finish both offloading and computing is determined by the slower process, allowing for efficient overlap.

The Compute-Memory Balanced Checkpointing algorithm tackles the challenge of balancing memory cost and computational efficiency. It identifies the minimum computational expenditure required to reconstruct each activation tensor while adhering to a specified memory budget. This method focuses on the size of stored activation, as it is more significant than temporary memory generated by at most a few layers. By enumerating the set of stored activations, we derive the Pareto frontier of memory cost and computation cost, from which we select a compute-memory balanced solution.

To optimize the vast search space of parallelism parameters, we propose an efficient method that involves measuring cluster-related primitive information and the model-related primitive information, such as the forward and backward time of each transformer layer, the time of pipeline parallel peer-to-peer communication, the bandwidth of optimizer communication, and so on. These measurements are then used to build a performance model that accurately predicts the time of each iteration, allowing us to exhaustively search for the optimal combination of parameters that minimize the time of each iteration while satisfying memory constraints.

The contribution is highlighted as follows:

1. Pipeline-Parallel-Aware Offloading is proposed to schedule offloading and reloading of activations, following the pipeline parallel schema, fully utilizing host memory to store activations with negligible overhead.

2. Compute-Memory Balanced Checkpointing is proposed to balance memory cost and computation cost to achieve the Pareto optimality.

3. We propose an efficient searching method to find the optimal hybrid parallelism parameters using the performance model measured from cluster-related primitive information and model-related primitive information.

4. Extensive experiments demonstrate the superiority of the proposed method. Remarkably, our method significantly increases Model FLOPs Utilization (MFU) from 32.3% to 42.7% for a 175B Llama-like model with a context window size of 32,768 on 256 NVIDIA H800 GPUs.

2 Related Work

2.1 Checkpointing

Activating checkpoints [4, 13] is a strategy designed to save memory by swapping computing resources, thus minimizing memory usage on a single GPU. When applied to a sequence of layers, this method retains only the input of the first layer for backpropagation. During the backward pass, intermediate outputs are recomputed as needed for gradient calculations. This approach significantly reduces the memory consumption associated with intermediate activations, freeing up memory to support larger models. Tools such as Rotor [1], Pofo [2], Checkmate [11], and POET [20] have been developed to automate the decision-making process of applying activation checkpoints in training pipelines, optimizing strategies to achieve the best runtime performance within a memory budget. Activation checkpointing does not involve tensor sharding, which is compatible with other parallelization methods.

2.2 Hybrid Parallelism

Data parallelism Data parallelism [34] is widely adopted for distributed learning. Each GPU processes its assigned dataset partition, and gradient synchronization occurs during the backward pass. Frameworks such as PyTorch DistributedDataParallel [16] and Horovod [25] facilitate the incorporation of data parallel training into existing codebases with minimal code changes. A critical limitation of data parallelism is the redundancy in memory usage, as each GPU retains a full copy of the model weights. To address this issue, DeepSpeed introduces a series research work on zero-redundancy optimizer [21, 22, 24, 30] that partitions the optimizer state, gradients, and model parameters during data-parallel training.

Tensor parallelism and context parallelism Tensor parallelism involves the partitioning of parameters and inputs within an operator, which is a technique used to distribute the computational load across multiple devices. In the context of the Transformer [29], Megatron-LM [26] takes advantage of the structure of transformer networks to remove a synchronization point. Additionally, there are some researches [14,17] that focus on partitioning along the sequence dimension, which makes activations further split along the context window. Sequence parallelism [14] distributes all activations along tensor parallel groups, while tensor parallelism only distributes part of activations. Research [17] distributes activations to a new kind of parallel groups that are orthogonal to tensor parallel groups, and this technique is named "context parallelism" in Megatron-LM.

Pipeline parallelism Pipeline parallelism partitions a model by its layers, distributing different layers across various devices for concurrent execution. This approach can introduce "bubbles" due to load imbalance and idle periods. To address the issue of bubbles in pipeline parallelism, numerous studies [9,15,18,35] have explored solutions from the perspective of asynchronous updates. However, this approach can lead to issues with model performance. In this work, we employ an interleaved schedule [19], which offers improvements over the basic schedule [7,10,18] approach in terms of both bubble size and memory consumption.

Hybrid parallelism tuning The current landscape of distributed parameter search for large models can be broadly categorized into two main approaches. One approach [12,32,36] involves modeling static compute graph and then employing some search algorithms, such as dynamic programming or integer linear programming, to identify the optimal solution. The other approach [8] involves enumerating distributed parameters and selecting the best solution based on empirical performance comparisons. However, the first approach has its limitations. It assumes that computational or communication costs are directly proportional to the amount of computation or communication, which may not always hold true. More critically, it struggles to accurately model the impact of hardware-specific computations that are overlapped by communication [23]. The second approach, which relies on exhaustive enumeration and empirical measurements, circumvents the modeling difficulties encountered by the first approach. However, this method can be time-consuming and costly due to the need to experiment with a large number of possible parameter combinations.

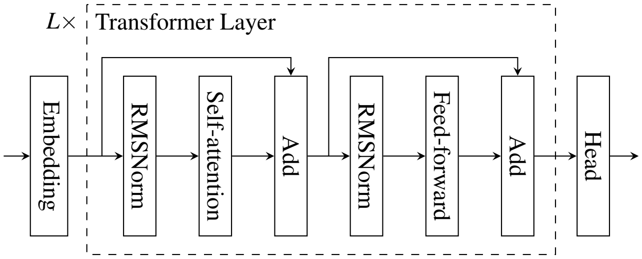


Figure 2: Structure of Llama series of models.

Table 1: Notation table.

Notations of model		Notations of hybrid parallelism	
L	# transformer layers	t	tensor parallel size
h	hidden dimension size	c	context parallel size
H	intermediate size	p	pipeline parallel size
a	# attention heads	d	data parallel size
g	# query groups	v	# pipeline stages per GPU
V	vocabulary size	l	# layers per stage
B	global batch size	b	micro-batch size
s	sequence length		

3 Preliminaries

Model definition A large language model usually consists of an embedding layer, L transformer layers, and a head layer. Figure 2 gives the structure of the Llama series of models. The dimension of hidden state of transformer layer is denoted h. The dimension of outputs of activation functions in feed-forward networks is named "intermediate size" and denoted H. See Table 1 for more notations that define the model.

Hybrid parallel groups During the hybrid parallel training, GPUs in a cluster are partitioned into a Cartesian product of tensor-parallel group, context-parallel group, pipeline-parallel group, and data-parallel group, of which the group sizes are denoted t, c, p, and d respectively, thus #GPUs $= tcpd$.

Memory footprint Memory consumption on each device can be categorized into the following four parts. Terms in the polynomials that are two or more orders of magnitude smaller than the others are omitted.

1) Model weights and gradients. The weight of each transformer layer is denoted to

$$P = \left(2 + \frac{2g}{a} + \frac{3H}{h}\right) h^2 \tag{1}$$

In hybrid parallelism, L transformer layers are split by interleaved pipeline parallelism. Each device holds v stages and each stage includes l transformer layers, thus $L = pvl$. Then weights and gradients are distributed to t devices along tensor-parallel group. The size on each device is

$$M_m = \frac{6}{t}\left(vlP + Vh \cdot \mathbb{1}_{r_{\mathrm{pp}} \in \{0,p-1\}}\right) \tag{2}$$

where $\mathbb{1}$ is known as indicator function and r_{pp} is defined as pipeline parallel rank. The coefficient $6 = 2 + 4$ consists of

BF16 weights and FP32 gradients. Change the coefficient if other data precision formats are specified.

2) Optimizer states and main weights. Optimizer is further distributed in data-parallel groups, whose size on each device is

$$M_o = \frac{12}{tcd}\left(vlP + Vh \cdot \mathbb{1}_{r_{\text{pp}} \in \{0,p-1\}}\right) \tag{3}$$

where the coefficient $12 = 4+4+4$ is the sum of FP32 main weights and two FP32 Adam states.

3) Maximum living activation size. Activation is distributed in both tensor-parallel groups and context-parallel groups. In this paper, sequence-parallelism is always enabled whenever $t \geq 2$ to ensure activations are fully distributed along tensor-parallel groups; FlashAttention [6] is adopted to reduce the memory overhead of the attention. For convenience, we use the term "activation block" to refer to all activations stored by one pipeline stage (which contains l transformer layers) for one micro-batch. The size of the activation block is

$$M_b = \frac{1}{tc}\left(12 + \frac{4g}{a} + \frac{8H}{h}\right) lbsh \tag{4}$$

In interleaved pipeline-parallelism, the number of forward steps before the first backward step is $(vp + p - 2r_{\text{pp}} - 1)$. Thus the maximum living activation size on each device is

$$M_a = (vp + p - 2r_{\text{pp}} - 1)M_b \tag{5}$$

4) Other buffers and overheads. The size of this part is implementation-dependent, including CUDA context overhead, memory management fragments, NCCL buffers, cuBLAS workspaces, and temporary variables. If the buffer size is not enough during training, an out-of-memory (OOM) error occurs, or the program may run under borderline conditions (e.g., frequent garbage collection, and frequent device synchronization).

4 Motivation

4.1 Activation Bottleneck

During the training of large language models, the GPU memory is faced with significant challenges, particularly in training models with a long context window. For example, training Llama-175B with a 32k context window takes at least $M_a = 171.5$ GB activation memory on the first rank, regardless of the hybrid parallel parameters employed (assuming $tc \leq 8$). The configurations of the models used for the experiments are presented in Table 2.

There are two straightforward methods to address the GPU memory issue, but they come with considerable side effects. The first approach to solve this problem is full checkpointing, but it leads to $1/3$ additional computation cost. Another way to solve this problem is increasing the tensor parallelism or context parallelism sizes, but it incurs substantial communication overhead and a reduction in computational intensity. To illustrate this issue, we present the estimated memory requirements and computation times for various parallelism configurations in Table 3. Due to practical limitations in GPU memory capacity for training models with the specified hybrid parallel parameters, we initially measured the forward and backward pass times for a single transformer layer. Subsequently, we approximate the time per iteration based on Equation (1) of [19]:

$$T_{\text{rough}} = (mv + p - 1)l(T_{\text{F}} + T_{\text{B}}) \tag{6}$$

where $m = B/(bd)$ is the number of micro-batches for each data-parallel group, $T_{\text{F}} + T_{\text{B}}$ is the forward and backward time of one transformer layer.

Table 2: The configurations of the models in experiments.

Model	L	h	H	a	g	V
Llama-175B	96	12288	32768	96	96	32005
Llama-65B	80	8192	22016	64	64	32005
Llama2-70B	80	8192	28672	64	8	32005

Table 3: Roughly estimated computation times and memory requirements for different hybrid parallel parameters. $B = 256$, $b = 1$, $l = 2$, #GPUs $= 256$.

Model	s	t	c	p	T_{rough} (s)	$M_m + M_o$ (MB)	M_a (MB)	Can run?
175B	4k	8	1	8	10.20	23,750	24,640	✓
175B	4k	4	1	8	8.91	39,583	49,280	OOM
65B	4k	2	2	8	3.70	26,899	28,200	✓
65B	4k	2	1	8	3.47	26,899	56,400	OOM
70B	16k	4	4	4	19.51	27,962	27,864	✓
70B	16k	4	2	4	18.13	27,962	55,728	OOM

4.2 Challenge of Hybrid Parallel Tuning

Problem 1 (Basic problem). Given the model, sequence length s, global batch size B, and the number of nodes, searching a group of hybrid parallel parameters $(t, c, p, l, \text{ckpt})$ to minimize the time of each iteration T.

Problem 2 (Throughput-maximization problem). Given the model, sequence length s, the range of satisfying global batch size $B \in [B_{\min}, B_{\max}]$, and a maximum number of nodes, searching a group of hybrid parallel parameters $(t, c, p, l, \text{ckpt})$ to maximize throughput Bs/T.

Here is some prior knowledge about reducing the searching space of performance tuning. The micro-batch size can be fixed to $b = 1$ to reduce parallel size thus achieving higher performance [8]. Cross-node tensor parallelism should be avoided because the communication size of tensor parallelism is $20lbsh/c$ for each pipeline stage on each device, which is $5lt$ times larger than the communication size of pipeline parallelism. Similarly, for models that do not employ grouped query attention (GQA), inter-node context parallelism should also be avoided, as it entails a $12lbsh/t$ communication size for each pipeline stage on each device. Our parameter tuning method can work well without the above constraints; however, these conditions were determined in the early stages of our research.

Even with the aforementioned prior knowledge to help shrink the search space, it is still very large, as shown in Table 4. Exhaustive searching for Problem 2 by enumerating combinations of $(B, \#\text{nodes})$ will repeatedly invoke the procedure of Problem 1.

Table 4: Searching space of Problem 1.

#GPUs	Llama-175B		Llama-65B		Llama2-70B	
	$\#(t,c)$	$\#(t,c,p,l)$	$\#(t,c)$	$\#(t,c,p,l)$	$\#(t,c)$	$\#(t,c,p,l)$
64	10	141	10	86	14	106
192	10	287	10	86	14	106
240	10	175	10	125	14	141
256	10	160	10	90	22	178
1024	10	160	10	90	30	250
7680	10	310	10	190	34	514

5 Method

This section begins with an exposition of two methods for rematerializing activations. Subsequently, it presents an approach capable of identifying optimal hybrid parallel parameters, which are designed to simultaneously take the rematerialization of activations into account.

5.1 Pipeline-Parallelism-Aware Offloading

As shown in Figure 3, we take advantage of the flow of interleaved pipeline parallelism [19] according to two principles to design our offloading scheme: 1) Offloading starts as soon as possible after the end of each pipeline stage forward. 2) Reloading starts at the beginning of the previous pipeline stage backward.

5.1.1 Implementation

Scheduling granularity The granularity of scheduling offloading and reloading is “pipeline stages”. In large language models, all pipeline stages have the same computation time and activation size (also the same transmission time). Thus the time to finish both offloading and computing is decided by the slower one between computation time and transmission time, and the faster one can be completely overlapped.

Event serialization The timing to serialize offloading events is worth considering because GPU memory allocations that are under-transferring cannot be reused. We use `cudaStreamWaitEvent` to make each offloading wait for the previous offloading finishes. This ensures the activation of at most one pipeline stage is under offloading.

Ping-pong reloading Two buffers are used in reloading: one is used as the target of reloading, and the other is used by the current backward step. At the next backward step, the roles of two buffers are swapped. Activation tensors are in-place constructed from the previous reloaded buffer.

Bandwidth utilization enhancing To achieve the highest bandwidth between GPU and host, we bind the Non-Uniform Memory Access (NUMA) node for each process and use page-locked memory for CPU buffers.

5.1.2 Memory Size

Additionally, the offload ratio α $(0 \leq \alpha \leq 1)$ is used to control how much activation is offloaded to host memory.

With the offloading schema, on the first pipeline parallel rank, the maximum number of offloaded activation blocks is $(vp + p - 3)$, one activation block is under offloading, one activation block is generated by the current forward step, and two buffers are used by reloading. Peak GPU memory usage is

$$M_{\text{gpu}} = M_m + M_o + ((vp + p - 3)(1 - \alpha) + 2 + 2\alpha)M_b \quad (7)$$

On the host side, there are at most $(vp + p - 3)$ activation blocks that are offloaded (including one activation block is under reloading), and one activation block is under offloading. Peak host memory usage is

$$M_{\text{host}} = (vp + p - 2)\alpha M_b \quad (8)$$

We select offload ratio α as less as possible for two reasons: Memory copy between host and device may slow down computation due to the competition for resources; offloading may be not completely overlapped by computation. First solve α using Equation 7 with maximum available GPU memory size. Then compute M_{host} and check whether it is out of host memory.

5.1.3 Overlap

As long as the offload ratio α increases, offloading may be not completely overlapped by computation. The pipeline schedule consists of 3 phases: warm-up phase that contains only forward, steady phase that contains pairs of forward and backward, and cooldown phase that contains only backward. Non-overlapped offloading/reloading will first occur in the warm-up phase. The reason is forward is about 2× faster than backward, thus steady phase has 3× time to do offload and reload, and the cooldown phase has 2× time to do reload.

There are $vp - 1$ steps in the critical path of the warm-up phase. The first forward step overlaps nothing, and the second to the p-th steps overlaps $T_{\text{embF}} + lT_{\text{F}}$, where T_{embF} is forward

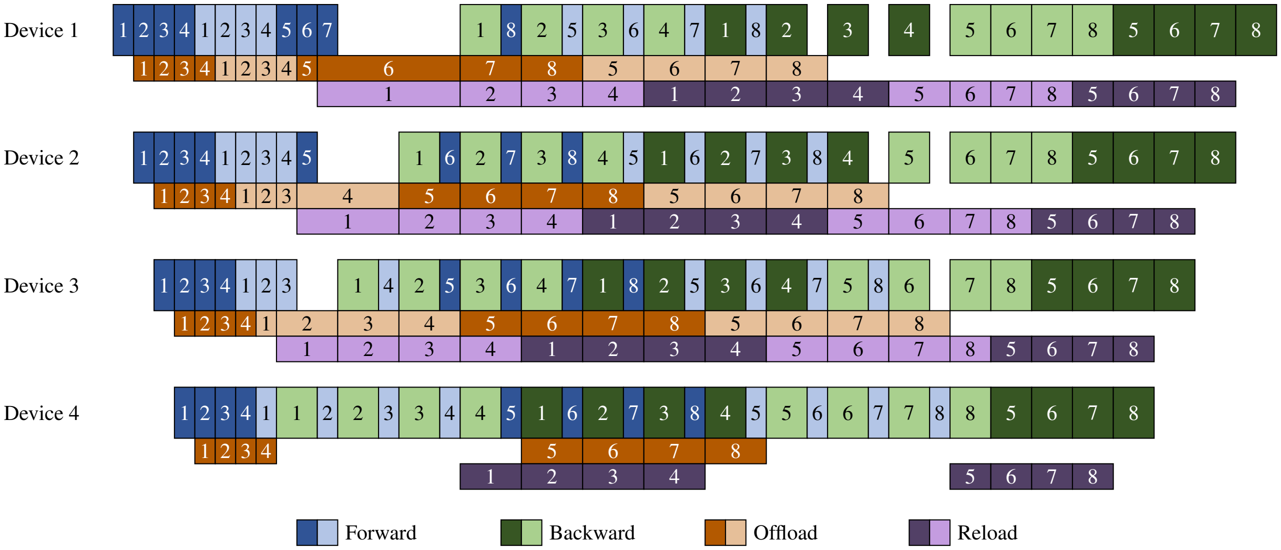


Figure 3: Schedule of pipeline-parallelism-aware offloadding and reloading. The pipeline parallel size is 4, each device has 2 virtual pipeline stages. Dark blocks are corresponding to the first virtual pipeline stage, light blocks are corresponding to the second virtual pipeline stage.

time of the embedding layer. Other $vp-p-1$ steps overlaps lT_{F}. The overhead at the warm-up phase is

$$\underbrace{(p-1)\max(0,\alpha M_b/\mathrm{BW}_{\mathrm{DtoH}}-T_{\mathrm{embF}}-lT_{\mathrm{F}})}_{\text{overlapped by 2nd to }p\text{-th forwards}}+ \\ \underbrace{(vp-p-1)\max(0,\alpha M_b/\mathrm{BW}_{\mathrm{DtoH}}-lT_{\mathrm{F}})}_{\text{overlaped by next forwards}} \tag{9}$$

where $\mathrm{BW}_{\mathrm{DtoH}}$ is the device to host memory copy bandwidth when all GPUs do the same memory copy in parallel. The overhead at the steady phase and cooldown phase can be calculated similarly, see Appendix B.3 for equations.

In summary, although we can make use of host memory, there is some overhead if offload too many activations. The activation size will be reduced in the next section.

5.2 Compute-Memory Balanced Checkpointing

In this section, we initially determine the minimum computational expenditure for each enumerated memory budget. We then deploy a checkpointing strategy that balances computational requirements and memory constraints, functioning on the Pareto optimal frontier. This strategy aims to introduce massive memory savings while incurring negligible incremental computational costs.

5.2.1 Overview

Several works [1, 2, 11, 20] have been devoted to designing strategies to minimize the memory consumption of the entire model or to optimize the computational cost within a defined memory budget. In pursuit of high accuracy, these studies carefully evaluate the cost associated with temporary tensors per operator, which not only increases the cost of individual computations but also renders previous solutions obsolete due to changes in parallel configurations. Most critically, in hybrid parallel scenarios, determining the optimal parallel configuration requires a large number of trials, resulting in unacceptable solution overhead for existing methods.

However, in the field of pipeline parallelism, the scope of checkpoints is often limited to a maximum of l layers rather than encompassing the entire model. Furthermore, the size of **stored** activation size should be focused, as on-the-flying temporary memory is generated by at most l layers, while stored activations are generated by at most $(vp+p-2)l$ layers.

We propose a compute-memory balanced solution on the Pareto Frontier that is insensitive to parallel configurations given a specified memory budget.

5.2.2 Pareto Frontier

In a transformer layer, the size of activation stored by each sublayer is shown in Table 5. First, for each activation tensor, we find the minimum computation cost to reconstruct it. As is different from previous works, temporary memory can be ignored in pipeline parallelism, thus we reconstruct activations layer by layer. During the construction, all activations of previous layers can be used, no matter whether the previous activation is stored or reconstructed.

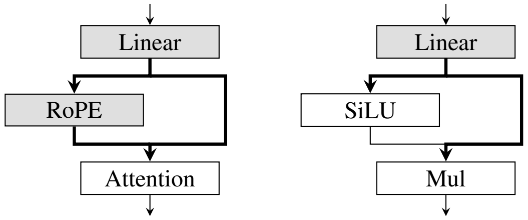
Figure 4(a) shows an example of reconstructing the input of Attention. Two layers are required to recompute. Figure 4(b) shows a slightly more complex case. Linear is required to be

Table 5: Stored activation size of each sublayer in a transformer layer for Llama/Llama2 models. The size that is two or more orders of magnitude smaller than the others is omitted.

#	Operation	Input sublayer ID	Stored activation size (in bytes)
1	`RMSNorm`	0[†]	$2bsh/(tc)$
2	`Linear`	1	$2bsh/(tc)$
3	`RoPE`	2	
4	`Attention`	2, 3	$(4+4g/a)bsh/(tc)$
5	`Linear`	4	$2bsh/(tc)$
6	`Add`	0[†], 5	
7	`RMSNorm`	6	$2bsh/(tc)$
8	`Linear`	7	$2bsh/(tc)$
9	`SiLU`	8	$2bsH/(tc)$
10	`Mul`	8, 9	$4bsH/(tc)$
11	`Linear`	10	$2bsH/(tc)$
12	`Add`	6, 11	
	Total		$(12+\frac{4g}{a}+\frac{8H}{h})bsh/(tc)$[‡]

[†] Sublayer ID 0 refers to the input of the transformer layer.

[‡] `Attention` (#4) and `Linear` (#5) share one stored activation tensor of size $2bsh/(tc)$.



(a) Reconstruct the input of Attention. Two layers are required to recompute.

(b) Reconstruct the input of SiLU. The second operand of Mul is also reconstructed.

Figure 4: Reconstruction cost. Operators recomputed are marked in gray. Tensors reconstructed are marked in bold.

Table 6: Stored activation and corresponding saved recomputing. Activation ID is the index of the sublayer that stores the activation. Recomputing ID is the index of the sublayer. A check mark indicates the activation is stored in our checkpointing and the reconstruction time is saved. Activation size and reconstruction time are measured on Llama-175B, $b=1$, $s=4096$, $t=4$, $c=1$.

Activation		Recomputing		Balanced
ID	size	ID	time (ms)	checkpointing
#1	$2bsh/(tc)$	-	-	✓[‡]
#2	$2bsh/(tc)$	#1	0.061	
#4a[†]	$6bsh/(tc)$	#2 #3	1.432	✓
#5	$2bsh/(tc)$	#4	0.454	✓
#7	$2bsh/(tc)$	#5 #6	1.018	✓
#8	$2bsh/(tc)$	#7	0.061	
#9 #10b[†]	$10.7bsh/(tc)$	#8	2.287	✓
#10a[†]	$5.3bsh/(tc)$	#9	0.105	
#11	$5.3bsh/(tc)$	#10	0.107	
-	-	#11 #12	1.684	✓[‡]
Total	$37.3bsh/(tc)$		7.209	

[†] #4a is QKV while the other part (#4b) is the output of `Attention` (alias to #5). #10a is the first operand of `Mul` that alias to the output of `SiLU`, #10b is the second operand.

[‡] The input must be stored. The last two layers do not need to be recomputed, so no activation is stored and the recomputing time is always saved.

recomputed when reconstructing the input of SiLU, and the output of Linear is used by both SiLU and Mul, so recomputing the Linear layer reconstructs both activations.

The computation cost to reconstruct each activation tensor is shown in Table 6. For example, reconstructing activation#2 (i.e., the activation stored by the second layer in Table 5) is compute-memory efficient because it saves $2bsh/(tc)$ memory by introducing only recomputing of RMSNorm#1, where the computation cost is only 0.061 ms. In contrast, reconstructing activation#7 is not efficient because it saves the same memory size but introduces 1.018 ms recomputing.

By enumerating the set of stored activations, we can get the Pareto frontier of memory cost and computation cost, as shown in Figure 5. The granularity of our checkpointing method is each transformer layer, so reconstructing the input of a transformer layer by recomputing the last two layers of the previous transformer layer is not considered. Splitting one layer into multiple layers (for example, recomputing half of Linear to reconstruct half activation size) may generate more solutions but is not considered in our method.

5.2.3 Compute-Memory Balanced Solution

Our compute-memory balanced checkpointing method uses the inflection point of the Pareto frontier. Stored activations are shown in Table 6. All computing-intensive layers (Linear and Attention) are not recomputed in our method. The set of layers that are recomputed includes RMSNorm#1, RMSNorm#7, SiLU#9, and Mul#10. Total recomputing time is only 1.5% times forward and backward time.

By using the compute-memory balanced checkpointing, the stored activation size of each transformer layer becomes

$$M'_b = \frac{1}{tc}\left(8+\frac{4g}{a}+\frac{4H}{h}\right)lbsh \tag{10}$$

Compared to no checkpointing, it saves 39% activation memory usage for Llama-175B and Llama-65B, and it saves 44% for Llama2-70B. Although the time is measured on one model, our checkpointing method is efficient for models of all scales. Because the executed recomputing cost is proportional to bsh, while each of the saved recomputing cost is proportional to bsh^2 or bs^2h. Interestingly, full checkpointing is not on the Pareto frontier. The reason is that full checkpointing always recomputes all layers including the last two layers.

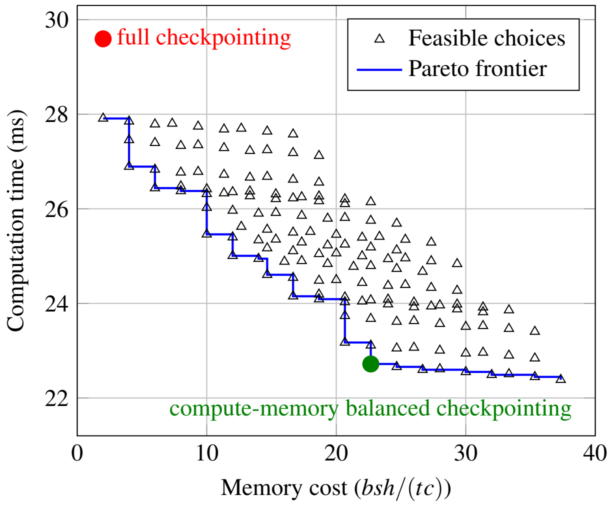
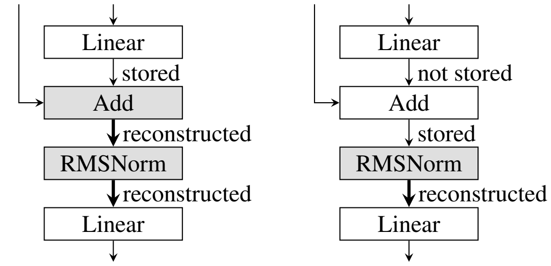


Figure 5: Memory cost and computation cost of different checkpointing methods of Llama-175B. Computation time is the sum of forward, recomputing, and backward. Triangle marks: all feasible choices. Blue line: Pareto frontier. Green point: compute-memory balanced checkpointing that is selected from the Pareto frontier. Red point: full checkpointing.



(a) SuperNeurons [31]

(b) Ours

Figure 6: Difference between two checkpointing methods. SuperNeurons [31] uses more recomputing than our method when reconstructing the input of Linear.

Both SuperNeurons [31] and our approach avoid recalculating Linear/Attention/Conv layers. However, Figure 6 shows an example where these two checkpointing methods are different indeed. SuperNeurons store the output of Linear, so they recompute more layers than ours.

5.3 Hybrid Parallel Parameters Tuning

The search space for hybrid parallelism parameters is vast, and the addition of offloading and checkpointing mechanisms further amplifies this complexity. This section treats parameter tuning as a constrained optimization problem and proposes an efficient method to solve this problem. In pursuit of modeling accuracy, we incorporate factors that are difficult to model, such as hardware characteristics and resource contention caused by concurrent computation and communication, into empirical measurements. By utilizing these empirically determined parameters, the model's theoretical predictions are in good agreement with its actual performance, with a difference of about two percent. Therefore, the optimal parallel configuration can be quickly determined through an exhaustive search, greatly improving the efficiency of the search process.

5.3.1 Tuning Algorithm

To reduce the total number of experiments, we measure some primitive information and subsequently construct a performance model based on those. Before explaining the performance model, we give the formulation of the optimization problem. The time of each iteration is modeled as T_{model}. Then we solve the minimizing problem:

$$\begin{gathered}\min_{t,c,p,l,\text{ckpt}} \quad T_{\text{model}} \\ \text{s.t. } M_{\text{gpu}} \leq M_{\text{gpu}}^{\text{thresh}} \wedge M_{\text{host}} \leq M_{\text{host}}^{\text{thresh}}\end{gathered} \tag{11}$$

where $M_{\text{gpu}}^{\text{thresh}}$ and $M_{\text{host}}^{\text{thresh}}$ are the GPU memory size threshold and host memory threshold for each device, respectively.

5.3.2 Performance Modeling

In pursuit of accurate modeling, key performance benchmarks as shown in Table 7 must be identified. The motivation for tracking computation and communication durations stems from their nonlinear relationship with theoretical computational complexity and message size. Specifically, the forward propagation time of the embedding layer, transformer layer, and header layer must be evaluated on all permutations of (t,c). These timings may vary due to factors such as GPU type, CUDA version, and specific operator implementation. Concurrently, this methodology offers the advantage of implicitly accounting for factors that are challenging to model explicitly, such as resource contention arising from variations in data shapes, concurrent communication and computation tasks, and other related complexities.

Typically, for a specific model configuration, the measurement process can be completed within a few minutes. The time for each iteration can be divided into the following parts:

1) Pipeline warm-up phase starts from the beginning of the first micro-batch forward on the first pipeline stage (on the first pipeline parallel rank), and ends at the beginning of the first micro-batch forward on the last pipeline stage (on the last pipeline parallel rank).

$$T_{\text{warmup}} = \underbrace{p(T_{\text{embF}} + lT_{\text{F}} + T_{\text{p2p}})}_{\text{first } p \text{ forwards on first rank}} + \underbrace{(vp - p - 1)(lT_{\text{F}} + T_{\text{p2p}})}_{\text{next stages of first micro-batch}} \tag{12}$$

where T_{p2p} is the time of the pipeline parallel peer-to-peer communication. T_{p2p} should be measured for each communication message size $2bsh/(tc)$.

Table 7: Notations of primitive information.

Symbol	Explanation	Measured times	Time to measure
$T_{\text{embF}}, T_{\text{embB}}$	Fwd/bwd time of the embedding layer	each model, each (b,s,t,c)	$2 \sim 15$ min for each model each (b,s)
$T_{\text{F}}, T_{\text{B}}$	Fwd/bwd time of each transformer layer		
$T_{\text{headF}}, T_{\text{headB}}$	Fwd/bwd time of the head layer		
T_{ckpt}	Recompute time of the balanced checkpointing	each $(b,s/(tc),h,s/c,H/t)$	total <15 min for all models (shared among models)
T_{p2p}	Time of pipeline parallel peer-to-peer comm.	each $(2bsh/(tc))$	
BW_{opt}	Algorithm bandwidth of optimizer comm.	each (t,cd)	
BW_{DtoH}	DtoH memory copy bandwidth per GPU	once	total <10 min for all models (shared among models)
BW_{HtoD}	HtoD memory copy bandwidth per GPU		
BW_{bidir}	Bidirectional memory copy bandwidth per GPU		
ω_{adam}	#Param. to time ratio of Adam optimizer		
β_{p2p}	Computation slow down ratio if overlaps p2p		
β_{offload}	Computation slow down ratio if overlaps offloading		

2) Pipeline steady phase follows the warm-up phase and ends to the tail of the last micro batch backward on the last pipeline stage.

$$T_{\text{steady}} = \underbrace{p(lT_{\text{F}} + T_{\text{headF}} + T_{\text{headB}} + lT_{\text{B}})}_{\text{first } p \text{ micro-batches on the last rank}} + \underbrace{(m-p)(vlT_{\text{F}} + T_{\text{headF}} + T_{\text{headB}} + vlT_{\text{B}})}_{\text{other } m-p \text{ micro-batches on the last rank}} \tag{13}$$

3) Pipeline cooldown phase follows the steady phase and ends to the tail of the last micro batch backward on the first pipeline stage.

$$T_{\text{cooldown}} = \underbrace{p(T_{\text{p2p}} + lT_{\text{B}} + T_{\text{embB}})}_{\text{last } p \text{ backwards on last rank}} + \underbrace{(vp-p-1)(T_{\text{p2p}} + lT_{\text{B}})}_{\text{other stages of last micro-batch}} \tag{14}$$

4) Optimizer communication and computation includes reduce-scatter of gradients, optimizer step, and all-gather of model weights.

Gradients and weights are communicated in the Cartesian product of context parallel group and data parallel group. The message size of optimizer communication is large enough, thus transmission time is decided by bandwidth, while latency is ignored. Denote BW_{opt} as the algorithm bandwidth of the optimizer communication group. Denote ω_{adam} as the ratio of the number parameters to Adam optimizer execution time. Then optimizer communication and computation time is modeled as

$$T_{\text{opt}} = \underbrace{M_m/\text{BW}_{\text{opt}}}_{\text{communication}} + \underbrace{(vlP+Vh)/(tcd)/\omega_{\text{adam}}}_{\text{Adam computation}} \tag{15}$$

where BW_{opt} should be measured for each combination of (t,cd), and $\omega_{\text{adam}} = 53.4$ GHz for Megatron-LM implementation on NVIDIA H800 80GB.

5) Offloading overhead and checkpointing overhead. Offloading overhead is computed according to Section 5.1.3, and the total offloading overhead is denoted to T_{offload}. The time to recompute selected sublayers in one transformer layer is denoted to T_{ckpt}. If compute-memory balanced checkpointing is used, T_{ckpt} is added to T_{B}. Our performance model also supports full checkpointing with the replacement of T_{ckpt} to T_{F}.

6) Computation slow down caused by overlap. The number of pipeline parallel peer-to-peer communications that are overlapped with computation on the critical execution path is $(4mv-2m+2p-2)$, where bidirectional communications are counted twice because both compete with computation. The additional time caused by computation slowdown is considered proportional to overlapped communication time with a ratio β_{p2p}. The number of offloading that overlaps with computation on the critical execution path is $(mv+p-2)$ and so is reloading. The additional time caused by computation slow down is considered proportional to offload size with a ratio β_{offload}.

$$\begin{aligned} T_{\text{slowdown}} = (4mv-2m+2p-2)\beta_{\text{p2p}}T_{\text{p2p}} + \\ \beta_{\text{offload}}(mv+p-2)\alpha M_b \end{aligned} \tag{16}$$

where β_{p2p} is measured by comparing the origin timeline and the timeline without peer-to-peer communication (replace peer-to-peer communication with a null operation), and β_{offload} is measured by comparing the timelines with different offload ratio α. On our hardware, we have $\beta_{\text{p2p}} = 0.05$ and $\beta_{\text{offload}} = 0.0016$ sec/GB.

As a result, we can estimate the total time as

$$\begin{aligned} T_{\text{model}} = T_{\text{warmup}} + T_{\text{steady}} + T_{\text{cooldown}} + \\ T_{\text{opt}} + T_{\text{offload}} + T_{\text{slowdown}} \end{aligned} \tag{17}$$

5.3.3 Parameter Searching

Before searching for the optimal parameter, we make some primitive measurements:

(1) For each combination of (t,c), we record the model layers computation time. Since the computation time is independent of data parallelism and pipeline parallelism, we can perform these measurements on a model with a reduced number of layers for time-saving. Depending on the size of

the model and the length of the sequence, the measurement duration varies from a few minutes to tens of minutes.

(2) The recomputation time of the memory balancing checkpoint is evaluated for each input shape of the selected sublayer. Models with different sequence lengths can share these measurements, which typically take a minute to complete.

(3) Measure point-to-point communication time for each $2bsh/(tc)$ configuration, which also takes approximately one minute.

(4) Evaluate the communication speed of the optimizer for each combination of (t, cd), which takes less than 10 minutes.

(5) Memory copy bandwidth and the parameters ω_{adam}, β_{opt}, and β_{offload} are measured only once and then applied to all models and sequence lengths.

After obtaining the primitive information mentioned above, we exhaustively enumerate all combinations of $(t, c, p, l, \text{ckpt})$, ensure it is a valid group of hybrid parallel parameters, affirm the memory size matches the constraint, and estimate T_{model} using Equation 17.

Based on our record, Problem 1 introduced in Section 4.2 is solved in 0.001 seconds with an exhaustive searching algorithm implemented in Python. Problem 2 can also be resolved in a short time since no additional primitive information is needed.

6 Evaluation

6.1 Experimental Settings

Experiments are conducted on a cluster where each node is equipped with eight NVIDIA H800 80GB GPUs interconnected via NVLink, with a bandwidth of 400 GB/s per GPU. For inter-node communication, each node is outfitted with eight 100 Gbps NICs. The nodes are configured with two Intel Xeon 8468V CPUs and 1TB of host memory. The cluster consists of 32 nodes. Training precision is BF16 with FP32 gradients accumulation. The optimizer is Adam with FP32 optimizer states. Interleaved pipeline parallel is always enabled. The global batch size is 256 if not specified otherwise. Micro batch size is fixed to 1. Our baseline is forked from Megatron-LM and we implement some industry level improvement on it. Further implementation details refer to Appendix B.1.

6.2 Offloading and Checkpointing

In this section, we measure GPU memory size to examine whether offloading and checkpointing work as expected.

Figure 7 shows the relationship between GPU memory usage and offload ratio. Without the help of offloading, running Llama-65B with context windows size of 8192 using parallel parameter $(t, c, p, l) = (2, 2, 8, 2)$ results in out-of-memory. The model can be trained using the offloading technique with $\alpha \geq 0.3$. The model can also be trained using the checkpointing technique. GPU memory size decreases as long as the offload ratio increases. Extra tensors not included in our memory model are small. The size of extra tensors is "max memory allocated" minus M_{gpu}, which is 1,188 MB on average.

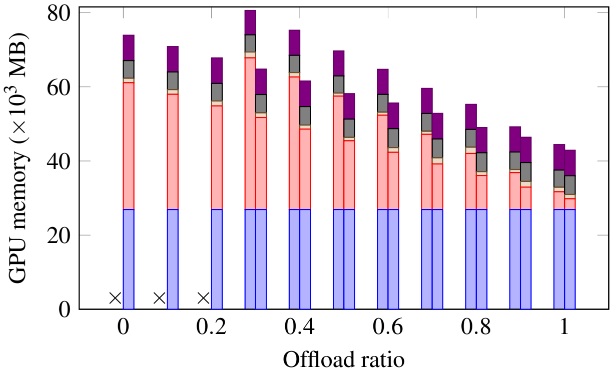


Figure 7: GPU memory with the use of offloading and checkpointing. Left bar: w/o checkpointing. Right bar: w/ checkpointing. From bottom to top: $M_m + M_o$, M_{gpu}, peak of all tensors (`max_memory_allocated`), PyTorch reserved memory (`max_memory_reserved`), total GPU memory usage. The cross mark indicates OOM. The model is Llama-65B, $s = 8192$, $t = 2$, $c = 2$, $p = 8$, $l = 2$.

In this example, the extra memory size reserved by PyTorch is 4,864 MB on average, extra memory size out of PyTorch memory management is 6,814 MB on average. The total memory size of NVIDIA H800 is 81,559 MB. There are 69,881 MB left on average for M_{gpu} and other tensors.

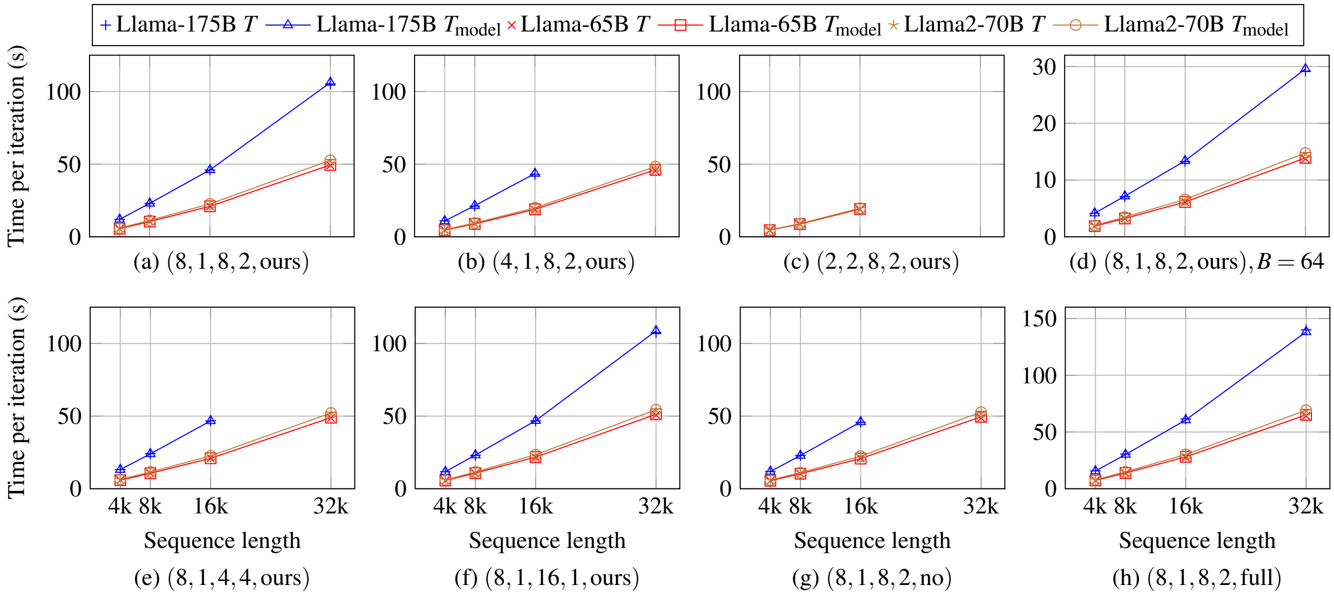
In the next sections, we set $M_{\text{gpu}}^{\text{thresh}} = 65,000$ MB if not otherwise specified. As for host memory, the total host memory size is 1 TB, so we allocate at most $M_{\text{host}}^{\text{thresh}} = 100,000$ MB for each device.

6.3 Performance Model

This section evaluates the accuracy of the performance model. We utilize a controlled variable methodology to measure the impact of various distributed parameters on the performance model. Figure 8(a)(b)(c) reveals our performance model is accurate for various t and c. Figure 8(a)(e)(f) suggests it is robust for different p and l. Figure 8(a)(g)(h) indicates our performance model maintains correctness for all three checkpointing methods. Figure 8(a)(d) illustrates the model can fit different global batch sizes. In all the cases, the difference between measured time and T_{model} is no larger than 2.0%.

6.4 End-to-End Performance Tuning

In this section, we compare the optimal performance of the baseline system and our system on three models and a variety of sequence lengths. For each system, optimal hybrid parallel



(a) $(8, 1, 8, 2, \text{ours})$ (b) $(4, 1, 8, 2, \text{ours})$ (c) $(2, 2, 8, 2, \text{ours})$ (d) $(8, 1, 8, 2, \text{ours}), B = 64$

(e) $(8, 1, 4, 4, \text{ours})$ (f) $(8, 1, 16, 1, \text{ours})$ (g) $(8, 1, 8, 2, \text{no})$ (h) $(8, 1, 8, 2, \text{full})$

Figure 8: Verify the accuracy of our performance model with different parameters. Items in a parenthesis are t, c, p, l, and checkpointing method, respectively.

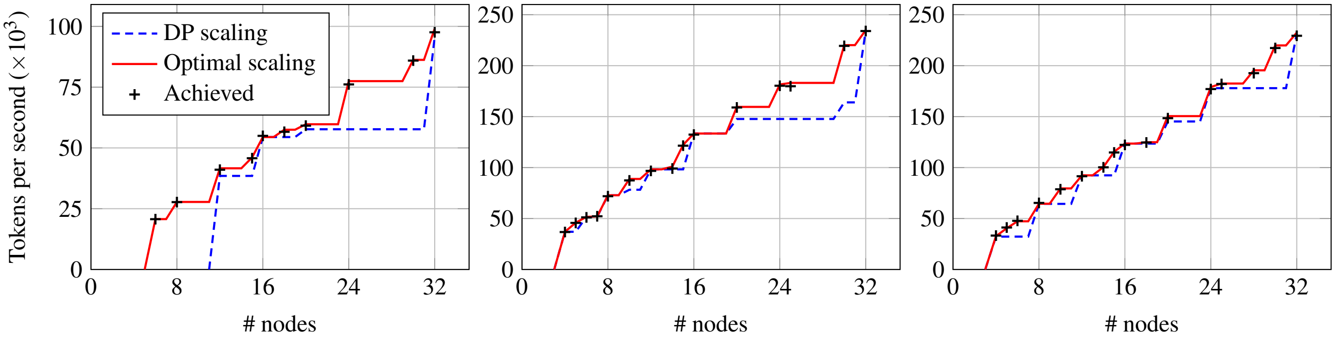
Table 8: End-to-end performance tuning. The global batch size is 256, number of GPUs is 256, throughput metric is normalized to tokens per second per GPU.

		w/o offloading and balanced checkpointing							w/ offloading and balanced checkpointing						
Model	s	t	c	p	l	ckpt	Throughput / MFU	Latest Megatron	t	c	p	l	ckpt	α	Throughput / MFU
Llama-175B	4096	4	2	16	2	no	367 / 39.8%	340 / 37.0%	2	2	16	1	no	53%	**381 / 41.4%**
Llama-175B	8192	2	2	16	1	full	299 / 33.4%	278 / 31.0%	4	1	8	2	ours	63%	**387 / 43.2%**
Llama-175B	16384	4	1	8	2	full	289 / 34.0%	284 / 33.3%	4	1	8	2	ours	85%	**382 / 44.8%**
Llama-175B	32768	4	2	8	2	full	250 / 32.3%	234 / 30.2%	4	2	8	2	ours	85%	**330 / 42.7%**
Llama-65B	4096	4	1	8	2	no	897 / 36.8%	868 / 35.6%	2	1	8	2	no	36%	**914 / 37.5%**
Llama-65B	8192	2	4	8	2	no	884 / 37.7%	802 / 34.2%	2	2	8	2	no	36%	**929 / 39.6%**
Llama-65B	16384	8	1	8	1	no	785 / 36.1%	753 / 34.6%	4	1	4	2	ours	43%	**879 / 40.4%**
Llama-65B	32768	2	2	4	2	full	590 / 31.0%	551 / 28.9%†	4	2	4	2	ours	43%	**734 / 38.5%**
Llama-65B	65536	2	4	4	2	full	433 / 28.3%	335 / 21.9%†	4	2	4	2	ours	77%	**548 / 35.9%**
Llama2-70B	4096	2	2	8	2	no	**875 / 37.9%**	804 / 34.8%	2	2	8	2	no	0%	**875 / 37.9%**
Llama2-70B	8192	2	4	8	2	no	**896 / 40.2%**	807 / 36.3%	2	4	8	2	no	0%	**896 / 40.2%**
Llama2-70B	16384	2	8	8	2	no	771 / 37.2%	612 / 29.5%	2	4	8	2	no	44%	**846 / 40.8%**
Llama2-70B	32768	2	16	8	2	no	612 / 33.5%	424 / 23.2%	2	4	4	2	ours	89%	**724 / 39.6%**
Llama2-70B	65536	4	16	4	5	no	438 / 29.7%	402 / 27.2%	2	4	8	1	ours	75%	**544 / 36.9%**
Llama2-70B	131072	2	4	8	1	full	285 / 26.7%	OOM	2	8	8	1	ours	77%‡	**352 / 33.0%**

† We configure `torch.cuda.set_per_process_memory_fraction` to run the case, otherwise, the latest Megatron-LM runs out of memory.

‡ The offload ratio is set to 2% higher than that is calculated by the memory model because memory overhead is large when the context window is long.

parameters are solved by our performance model. Results are shown in Table 8. The performance of the latest Megatron-LM is also shown for comparison. Our baseline slightly outperforms the latest Megatron-LM using the same parameters. Note that our performance model is not tuned for the latest Megatron-LM, we just use the same parameters to run it. With the help of offloading and balanced checkpointing, our method has more room to trade-off parallel configurations, resulting in significant performance gain compared to the baseline. For example, our method significantly increases MFU from 32.3% to 42.7% for Llama-175B with a context window size of 32,768 on 256 NVIDIA H800 GPUs.



(a) Llama-175B, $s = 4096$ (b) Llama-65B, $s = 4096$ (c) Llama2-70B, $s = 8192$

Figure 9: Data parallel scaling and optimal scaling. Lines are modeled throughput, marks are achieved throughput. The range of global batch size is 256 ± 16.

Here are some interesting observations from the result:

(1) None of the factors in the hybrid parallel search space are trivial to achieve optimum performance. For a single model, the solution evolves with varying context window sizes. Models using the same context window size may also demand distinct parallel parameters to attain maximum throughput.

(2) For the GQA model, context parallelism is preferred over tensor parallelism. This reason is that context parallelism communication size $12lbsg(h/a)/t$ is only $g/a = 1/8$ times that of the model without GQA (which is $12lbsh/t$).

(3) For the GQA model, inter-node context parallelism is preferred over full checkpointing. However inter-node context parallelism has a significant overhead. For example, when s increases from 8192 to 16384, MFU drops 3.0% using our implementation, and MFU drops 6.8% using the latest Megatron-LM implementation; the overhead of offloading is much smaller than inter-node context parallelism, and no MFU drop is observed.

6.5 Optimal Scaling

It is a practical requirement to find the optimal training configuration for each scale of a cluster. The example is to train three models with different context window lengths, global batch size can be chosen from the range 256 ± 16. The number of nodes scales from 4 to 32. The data parallel scaling method is to tune the optimal parameters for $B = 256$ on half nodes, and scale data parallel size as long as the number of nodes changes.

The optimal scaling method is to tune the parameters for every global batch size and every number of nodes. The output of the optimal scaling algorithm is $(B, t, c, p, l, \text{ckpt})$ for each number of nodes. We implement the optimal scaling algorithm by exhaustive searching using Python. The algorithm finishes in 1 second. The result is shown in Figure 9.

Then we run experiments on each output of the optimal scaling algorithm. The result shows that each modeled throughput is achieved with an error of at most 1.9%. The optimal scaling method outperforms data parallel scaling at many numbers of nodes. For example, given 24 nodes, the optimal scaling achieves 1.80×10^5 tokens per second (TPS) on Llama-65B $s = 4096$, and the optimal parameter is $(B, t, c, p, l, \text{ckpt}) = (240, 4, 1, 8, 2, \text{no})$. Data parallel scaling can utilize only 20 nodes and is expected to achieve 1.48×10^5 TPS, the parameter is $(240, 2, 1, 8, 2, \text{ours})$. Besides, it's also worse than the optimal parameter on 20 nodes, which is $(240, 2, 1, 10, 2, \text{no})$ and achieves 1.59×10^5 TPS.

7 Conclusion

This paper proposed two activation rematerialization methods including Pipeline-Parallel-Aware Offloading, which maximizes the utilization of host memory for storing activations, and Compute-Memory Balanced Checkpointing, which seeks a practical equilibrium between activation memory and computational efficiency. To optimize the vast search space of parallelism parameters, an efficient method is proposed to exhaustively search for the optimal combination of parameters by building the performance model with few-shot measurements on the primitive information.

Limitation and Future Work Our proposed compute-memory balanced checkpointing method supports a smaller maximum sequence length than the full checkpointing method. In the future, we may explore more optimization strategies to solve this problem. Furthermore, the temperature of the GPUs may affect the accuracy of the time measurement and thus the performance model, it is interesting to take a thorough consideration of these aspects in the future.

References

[1] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *arXiv preprint arXiv:1911.13214*, 2019.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient combination of rematerialization and offloading for training DNNs. In *Advances in Neural Information Processing Systems*, volume 34, pages 23844–23857, 2021.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, 2022.

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[8] Johannes Hagemann, Samuel Weinbach, Konstantin Dobler, Maximilian Schall, and Gerard de Melo. Efficient parallelization layouts for large-scale distributed model training. In *Workshop on Advancing Neural Network Training at 37th Conference on Neural Information Processing Systems*, 2023.

[9] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. PipeDream: Fast and efficient pipeline parallel DNN training. *arXiv preprint arXiv:1806.03377*, 2018.

[10] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32, pages 103–112, 2019.

[11] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020.

[12] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems*, volume 1, 2019.

[13] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

[14] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems*, volume 5, pages 341–353, 2023.

[15] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined backpropagation at scale: Training large models without batches. In *Proceedings of Machine Learning and Systems*, volume 3, pages 479–501, 2021.

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. In *Proceedings of the VLDB Endowment*, volume 13, pages 3005–3018, 2020.

[17] Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. Sequence parallelism: Making 4D parallelism possible. *arXiv preprint arXiv:2105.13120*, 2021.

[18] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 7937–7947, 2021.

[19] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, number 58, pages 1–15, 2021.

[20] Shishir G. Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph E. Gonzalez. POET: Training neural networks on tiny devices with integrated rematerialization and paging. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pages 17573–17583, 2022.

[21] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[22] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong 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*, number 59, pages 1–14, 2021.

[23] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 540–553, 2021.

[24] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 551–564, 2021.

[25] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[26] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[28] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In

Advances in Neural Information Processing Systems, volume 30, 2017.

[30] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. ZeRO++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

[31] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53, 2018.

[32] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, number 26, pages 1–17, 2019.

[33] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake A. Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 1, pages 93–106, 2023.

[34] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[35] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. PipeMare: Asynchronous pipeline parallel DNN training. In *Proceedings of Machine Learning and Systems*, volume 3, pages 269–296, 2021.

[36] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pages 559–578, 2022.

A Artifact Appendix

A.1 Abstract

This artifact offers resources to reproduce the accelerating method. The minimum demo can be run on 4 GPUs, and all the experiment results can be verified on a cluster that hardware is specified in Section 6.1. This artifact also includes scripts and a Docker image, ensuring reproducibility of the experiment results. The artifact is publicly accessible on GitHub to promote further research.

A.2 Scope

1. Getting Started Instructions: To reproduce the accelerating method on a minimum demo using 4 GPU cards. It verifies that our activation rematerialization mechanism reduces GPU memory consumption, and achieves a significant performance improvement over the baseline by utilizing offloading and balanced checkpointing. Besides, it verifies that our baseline outperforms the latest Megatron-LM.

2. Detailed Instructions: To reproduce the exact performance reported in this work. By following these instructions, researchers should be able to reproduce all the memory usage values in Figure 7, all the “Throughput / MFU” values listed in Table 8, and all the “achieved throughput” values in Figure 9.

A.3 Contents

This artifact includes source code, a readme document, scripts, a Docker image, and a sample dataset.

A.4 Hosting

This artifact can be downloaded from the `atc24ae` branch of https://github.com/kwai/Megatron-Kwai. The readme document and scripts are located at `examples/atc24`.

Table 9: The main differences between the latest Megatron-LM, codebase, and our code may influence performance. Whether our implementation is the same as Megatron-LM is shown in the last column.

Feature	Latest Megatron	Codebase	Ours	Impl.
Context parallelism	✓		✓	Diff.
Overlap TP comm.	✓		✓	Diff.
Manual Python GC	✓		✓	Diff.
Embedding layer	✓	✓	✓	Diff.
Grouped query attn.	✓		✓	Diff.
FlashAttention-2	✓		✓	Same
Fused RMSNorm	✓		✓	Same
Fused RoPE			✓	-
Our offloading			✓	
Our checkpointing			✓	

Table 10: Schedule of offloading and reloading on the first pipeline parallel rank. In the example, the number of micro-batches is 8, and the number of pipeline stages is 2. Each activation block is represented in two numbers in parenthesis: micro batch index and pipeline stage index. An overview of the pipeline of all ranks can be found in Figure 3.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
forward	(1,1)	(2,1)	(3,1)	(4,1)	(1,2)	(2,2)	(3,2)	(4,2)	(5,1)	(6,1)	(7,1)		(8,1)		(5,2)		(6,2)
backward												(1,2)		(2,2)		(3,2)	
# living act.	1	2	3	4	5	6	7	8	9	10	11	11	11	11	11	11	11
offload		(1,1)	(2,1)	(3,1)	(4,1)	(1,2)	(2,2)	(3,2)	(4,2)	(5,1)	(6,1)	(7,1)		(8,1)		(5,2)	
reload											(1,2)	(2,2)		(3,2)		(4,2)	
# living @GPU	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3
# living @host	0	1	2	3	4	5	6	7	8	9	10	10	10	10	10	10	10

step	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	
forward		(7,2)		(8,2)												
backward	(4,2)		(1,1)		(2,1)	(3,1)	(4,1)	(5,2)	(6,2)	(7,2)	(8,2)	(5,1)	(6,1)	(7,1)	(8,1)	
# living act.	11	11	11	11	11	10	9	8	7	6	5	4	3	2	1	
offload	(6,2)		(7,2)		(8,2)											
reload	(1,1)		(2,1)		(3,1)	(4,1)	(5,2)	(6,2)	(7,2)	(8,2)	(5,1)	(6,1)	(7,1)	(8,1)		
# living @GPU	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	
# living @host	10	10	10	10	10	9	8	7	6	5	4	3	2	1	0	

B Implementation Details

B.1 Software Version

CUDA toolkit version is 12.3.0, PyTorch version is 2.1.0-rc2, NCCL version is 2.18.3-1, FlashAttention version is 2.2.5, and TransformerEngine version is 1.1.0.

"Latest Megatron" refers to the publicly available Megatron-LM. We checked out the main branch on Jan 1, 2024, when the latest commit was `2bc6cd3`. We follow their example to set experiments. These options are additionally used to enhance performance: 1) The implementation from Megatron core (which uses TransformerEngine) is used; 2) Distributed optimizer is used; 3) FlashAttention-2 is used; 4) Overlap of tensor parallel communication and `Gemm` kernels [33] is enabled when $c = 1$[1]; 5) Manual Python garbage collection is enabled and the interval is set to infinity[2].

Our codebase is forked from Megatron-LM on Jun 9, 2023, with commit `db71a33`. Table 9 shows the main difference between our code and Megatron-LM. We implement offloading and checkpointing on the top of the codebase rather than the latest Megatron-LM because we have made a lot of industrial modifications to the codebase, and our code is more robust and faster than the latest Megatron-LM before offloading and checkpointing are added.

B.2 Detailed Counting of Activations

A schedule of $p = 4$, $v = 2$ on the first pipeline parallel rank is shown in Table 10. An activation block is represented by a pair of micro-batch ID and pipeline stage ID.

Let's explain the example through a case. (3,2) is the activation block of the third micro batch generated by the second pipeline stage. (3,2) is generated at step 7. Offloading of (3,2) starts at the beginning of step 8 and is serialized to finish before the beginning of step 9. Reloading of (3,2) starts at the beginning of step 14 and is serialized to finish before the beginning of step 16. At last, (3,2) is used in the backward stage at step 16.

At each step, the statistics "# living act." is the number of unique stored activation blocks. With the offloaded schema, living activation blocks are divided into activation blocks on the GPU and activation blocks on the host. Activation blocks that are under offloading or reloading are counted to both "# living @GPU" and "# living @host".

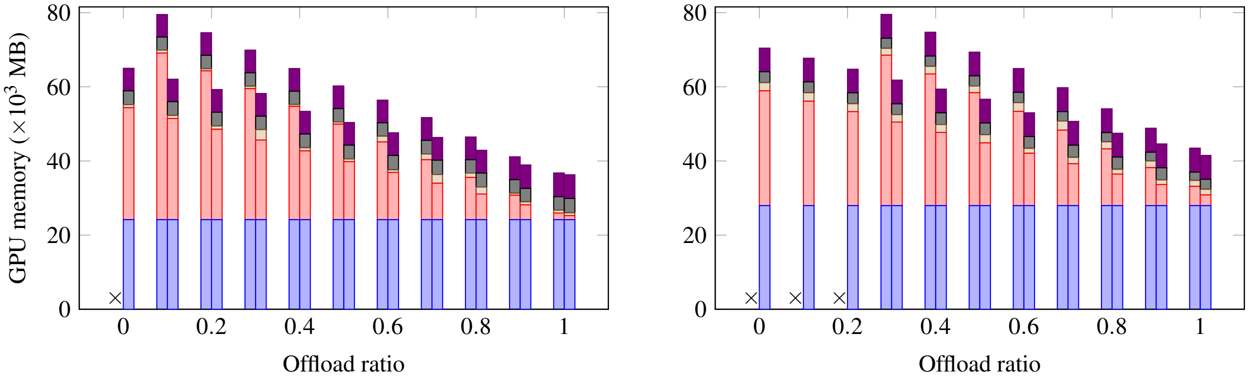
B.3 Not Overlapped Offloading

The overhead caused by not overlapped offloading at the warm-up phase is given in Section 5.1.3.

In the steady phase, on the device whose pipeline parallel rank is $p-2$, the first forward stage is followed by the pipeline bubble. The each of the following $p-2$ backward and forward stages overlaps $lT_{\mathrm{F+B}} + T_{\mathrm{head}}$, where $T_{\mathrm{F+B}} = T_{\mathrm{F}} + T_{\mathrm{B}}$, $T_{\mathrm{head}} = T_{\mathrm{headF}} + T_{\mathrm{headB}}$. Then there are m/p groups of paired backward and forward stages. In each group, there are $(v-1)p$ backward and forward stages that each overlaps

[1] In the latest Megatron-LM, a runtime error is raised if context parallelism works together with an overlap of tensor parallel communication.

[2] Python garbage collection only affects host memory size, does not affect whether OOM occurs on GPU.



(a) Llama-175B, $s = 4096$, $t = 4$, $c = 1$, $p = 16$, $l = 1$ (b) Llama2-70B, $s = 8192$, $t = 4$, $c = 1$, $p = 4$, $l = 2$

Figure 10: GPU memory with the use of offloading and checkpointing. Left bar: w/o checkpointing. Right bar: w/ checkpointing. From bottom to top: $M_m + M_o$, M_{gpu}, peak of all tensors (`max_memory_allocated`), PyTorch reserved memory (`max_memory_reserved`), total GPU memory usage. The cross mark means the OOM.

$lT_{\text{F+B}}$ and each of the other p groups overlaps $lT_{\text{F+B}} + T_{\text{head}}$. An exception is the last backward stage in the last group is followed by the pipeline bubble. In summary, the overhead at the steady phase is

$$\begin{aligned} &(m-3)\max(0, 2\alpha M_b/\text{BW}_{\text{bidir}} - lT_{\text{F+B}} - T_{\text{head}}) + \\ &(m-p)(v-1)\max(0, 2\alpha M_b/\text{BW}_{\text{bidir}} - lT_{\text{F+B}}) \end{aligned} \tag{18}$$

The overhead at the cooldown phase is

$$\begin{aligned} &(vp-p-1)\max(0, \alpha M_b/\text{BW}_{\text{HtoD}} - lT_{\text{B}}) + \\ &(p-1)\max(0, \alpha M_b/\text{BW}_{\text{HtoD}} - lT_{\text{B}} - T_{\text{embB}}) \end{aligned} \tag{19}$$

C Additional Evaluation

C.1 Additional Offloading and Checkpointing Results

We further verify our proposed methods on two additional benchmark models, as shown in Figure 10. We observe a similar memory overhead pattern as in Figure 7.

C.2 Training Loss

We train a Llama2-70B model from scratch to plot the training loss pattern. The context window size is 4096, the global batch size is 1024. Learning rate and optimizer parameters are set the same as Meta's Llama 2 [28]. The learning rate schedule strategy is cosine, the peak learning rate is 1.5×10^{-4}, the number of learning rate decay iterations is 500,000, the number of warm-up iterations is 2000, the decay final learning rate is 1.5×10^{-5}, weight decay is 0.1, gradient clipping is 1.0. Adam parameters are $\beta_1 = 0.9$, $\beta_2 = 0.95$, eps $= 1 \times 10^{-5}$. To verify that our techniques correctly work together with all hybrid parallel methods, we set the parameters as follows: $t = 2$, $c = 2$, $p = 8$, $l = 2$, ckpt $=$ ours, $\alpha = 0.5$.

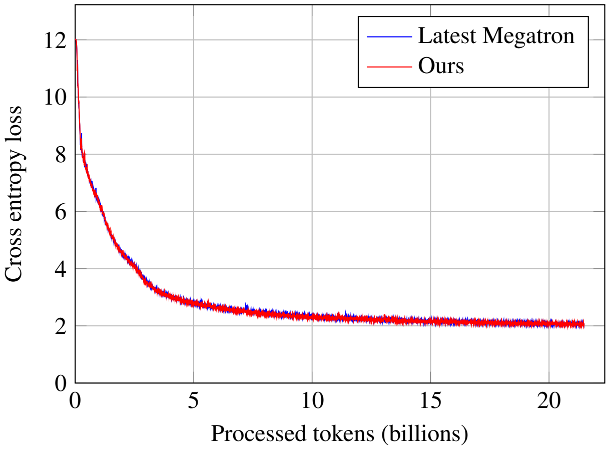


Figure 11: Training loss.

We compare the training loss with the latest Megatron-LM, as shown in Figure 11. The result shows that our accelerating techniques do not harm the original model's performance.