Mind Bubbles and Memory: Bounds on Scheduling Pipeline Parallelism with Rematerialization

Adrien Aguila–Multner, Olivier Beaumont, Lionel Eyraud-Dubois, Julia Gusak

Mind Bubbles and Memory: Bounds on Scheduling Pipeline Parallelism with Rematerialization

Adrien Aguila-Multner
Inria Center of the University of Bordeaux
Bordeaux, France
ORCID:0009-0002-1004-339X

Olivier Beaumont
Inria Center of the University of Bordeaux
Bordeaux, France
ORCID:0000-0003-2741-6228

Lionel Eyraud-Dubois
Inria Center of the University of Bordeaux
Bordeaux, France
ORCID:0000-0003-2475-3309

Julia Gusak
Inria Center of the University of Bordeaux
Bordeaux, France
ORCID:0000-0002-0443-6333

***Abstract*—Training large neural networks, especially Transformer-based Large Language Models (LLMs), requires massive high-performance computing (HPC) resources. Within each microbatch, computations follow a strictly sequential flow through a stack of transformer blocks: a forward pass to compute the loss, and a backward pass to propagate gradients. This sequential structure limits intrinsic parallelism. To improve performance, several complementary strategies have been developed: data, tensor, sequence, and pipeline parallelism, typically combined to achieve scalability over tens of thousands of GPUs.**

This paper presents a formal analysis of *pipeline parallelism* (PP) for large-scale training. In PP, the model is partitioned into multiple stages, and microbatches are injected into the pipeline to overlap computation. The main challenge is to minimize idle periods (pipeline *bubbles*) while managing memory usage, since each GPU must store intermediate activations from multiple in-flight microbatches. Existing scheduling algorithms such as GPipe, 1F1B, Hanayo, and Megatron reduce idle time but lack formal lower bounds or explicit modeling of memory constraints.

We develop a unified analytical approach for PP scheduling, deriving lower bounds on completion time for both single-wave and multi-wave regimes. Our analysis explicitly incorporates a memory constraint K, denoting the number of activations that can be stored per GPU. Exact results are provided for two extreme cases (minimal memory ($K = 1$) and large memory ($K \geq m$)), while general lower bounds are established for intermediate configurations. Our analysis highlights the intrinsic coupling between pipeline utilization and memory footprint, providing a foundation for evaluating and comparing pipeline scheduling algorithms under realistic memory constraints.

I. Introduction

The training of large Transformer-based neural networks, including today's foundation and language models, has become one of the dominant workloads in high-performance computing (HPC). The underlying computation graph exhibits a fundamentally sequential structure: each layer (a transformer block) processes activations in a strict forward order, followed by a backward traversal for gradient computation. As a result, the critical path spans the entire model depth, yielding poor inherent parallelism.

To overcome this limitation, several complementary parallelization strategies have been proposed. **Tensor and sequence parallelisms** [1]–[3] split operations within each transformer across multiple GPUs, reducing the critical path but introducing significant communication overhead between nodes. **Data parallelism** [4], [5] replicates model weights across devices and processes multiple data samples simultaneously; it scales well but incurs expensive global gradient synchronization through AllReduce collective communications. **Pipeline parallelism (PP)** [1], [6]–[9] partitions the layers of the model into contiguous stages, assigned to the different GPUs. Several microbatches are injected into the pipeline, allowing multiple stages and GPUs to operate concurrently.

In the present paper, we focus on pipeline parallelism (PP). PP distributes both computation and memory usage due to weights and activations per microbatch, but it introduces two major challenges. First, the pipeline cannot be fully occupied: during each iteration, some GPUs remain idle, creating *bubbles* that degrade performance efficiency. Second, limiting the bubbles requires in general to increase the number of in-flight microbatches, what in turn increases the number of stored activations and amplifies memory pressure on each GPU.

A wide range of scheduling algorithms have been proposed to mitigate idle time, such as GPipe, 1F1B, Hanayo, and Megatron, based on different strategies for interleaving forward and backward operations. However, these works rely on clever combinatorial reasoning and generally lack a formal characterization of the minimal achievable time, or *lower bounds*, especially when memory constraints are considered.

When GPU memory is insufficient to keep all intermediate activations, a complementary strategy is to use **rematerialization** (also known as activation checkpointing). During the forward phase, some activations can be discarded from GPU memory and then later recomputed on demand during the backward phase. This approach allows the training process to fit within limited memory capacity at the cost of additional computation. Importantly, in a PP context, these recomputations can be advantageously scheduled within idle periods

(*bubbles*), exploiting otherwise unused compute cycles. Thus, rematerialization offers a practical way to trade memory for computation while maintaining high utilization.

This paper aims to fill that gap through a formal, memory-aware analysis of pipeline parallel training. Our main contributions are: (i) a unified analytical model of pipeline parallel training, integrating both single-wave and multi-wave scheduling strategies (Section II); (ii) the derivation of lower bounds on the completion time for processing m microbatches on N GPUs, even in presence of memory limitations (Section IV). To establish these bounds, we rely on new techniques and we explicitly incorporate a memory constraint through a parameter K, representing the number of forward activations that can be stored simultaneously on each GPU; (iii) exact results and optimal schedules for the two extreme cases – **minimal memory** ($K = 1$) and **large memory** ($K \geq m$) – corresponding respectively to constrained and unconstrained pipelines (Section III); (iv) a detailed discussion of representative scheduling strategies, including GPIPE, 1F1B, HANAYO, and MEGATRON, within this unified modeling framework (Section V). Finally, Sections VI and VII discuss related works and limitations of this work.

Our analysis shows that pipeline idle time and memory footprint are inherently coupled: reducing bubbles typically requires storing more activations, while reducing memory usage increases idle time. Establishing formal limits under this trade-off is crucial to designing efficient pipeline schedules for large-scale model training on exascale HPC systems.

II. MODELING FRAMEWORK

A. Overview and Scope

This section formalizes the pipeline parallel training model and introduces the notations used throughout the paper. We start by explaining why we can restrict the focus to the pipeline parallelism dimension.

Following [10], let the total number N of GPUs be denoted as:

$$N = DP \times PP \times CP \times TP,$$

where DP denotes data parallelism, PP pipeline parallelism, CP context parallelism, and TP tensor parallelism. In the rest of the paper, we will assume $DP = CP = TP = 1$ and focus on $PP = N$.

We argue that our proposed analysis of pipeline parallelism remains valid when combined with other forms of parallelism. Tensor and context parallelisms (*TP* and *CP*) effectively reduce the time required to execute each forward (F) and each backward (B) operation by distributing the computation of a single operation over multiple GPUs. However, they do not alter the logical sequencing of tasks across pipeline stages and just decrease the time necessary for performing any operation. In what follows, GPU_i performs tasks F (resp. B) should be translated as the group of $CP \times TP$ GPUs perform tasks F (resp. B) if tensor and context are used. Consequently, the analysis of the pipeline itself is orthogonal to TP and CP: it can be carried out independently and its conclusions remain valid under any intra-layer parallelization efficiency.

Data parallelism (*DP*), on the other hand, acts at a higher level. It replicates the entire pipeline group across multiple sets of GPUs, each processing a distinct subset of the input data. Therefore, each data parallel group contains $PP \times CP \times TP$ GPUs. The synchronization required for gradient aggregation occurs after all pipeline steps are completed. Therefore, the pipeline analysis presented here focuses on a single DP group and applies identically to all replicas.

B. Decomposition of the Pipeline Optimization Problem

Optimizing pipeline parallelism involves solving three interdependent sub-problems that jointly determine the overall training efficiency.

(i) Number of microbatches: The first problem concerns the choice of the number m of microbatches injected into the pipeline.

(ii) Stage allocation of model layers: The second problem concerns how the L layers of the neural network are partitioned and mapped onto the N available GPUs. Each GPU is assigned one (single wave allocation) or several (multi-wave allocation) contiguous subsets of layers, defining one or more *pipeline stages*. This allocation determines the effective pipeline depth and the computational load per GPU.

(iii) Global scheduling of forward and backward tasks: Given a fixed stage allocation, the second problem is to determine the global order in which all forward and backward operations are executed across the N GPUs. This is a constrained scheduling problem whose objective is to minimize the **completion time** (makespan) required to process m microbatches. The schedule must satisfy:

- intra-microbatch **dependencies**, ensuring that for each microbatch, backward operations occur in reverse order after the corresponding forward computations;
- **resource constraints**, ensuring that each GPU executes at most one operation at a time and respects its memory limit K.

C. Performace Tradeoffs

Optimizing pipeline performance depends on three main factors: the number m of injected microbatches, the number w of concurrent waves, and the scheduling policy used to interleave forward and backward tasks. Each factor improves pipeline utilization but introduces specific limitations.

(i) Number of microbatches: Increasing the number of microbatches injected into the pipeline generally improves GPU utilization by reducing the impact of idle periods (*bubbles*). However, this strategy has two main drawbacks. First, it increases memory consumption: each forward task F produces activations that must be stored until the corresponding backward task B is executed. As a result, the total memory footprint grows approximately linearly with the number of in-flight microbatches, although optimized schedules may mitigate this by overlapping F and B operations.

Second, the number of microbatches is constrained by the training process itself. For instance, in the final stages of large-scale LLM training, models are trained with very long sequences (typically up to 132,000 tokens according to [10]). To maintain model quality, the number of tokens processed between two weight updates should not exceed roughly 16 million, corresponding to about 128 such sequences. Dividing this number by the degree of data parallelism DP provides an upper bound on the number of microbatches that can be injected into each pipeline concurrently.

(ii) Number of waves: Increasing the number of waves (i.e., virtual pipeline stages per GPU) provides additional opportunities for overlapping computation and reducing bubbles. However, two practical limitations arise. First, the achievable number of waves is bounded by the model's granularity: unless transformer blocks are further subdivided, the number of stages cannot exceed the number of transformer layers (typically around 128 in very large models). Second, communication volume increases linearly with the number of waves, since each additional wave introduces one more set of peer-to-peer data exchanges between consecutive partitions. The data transferred between two partitions corresponds to the input of a transformer block and is therefore independent of the number of blocks per stage. Nevertheless, this additional communication cost remains small compared to the data traffic induced by tensor, context, and data parallelism.

(iii) Scheduling and rematerialization: The scheduling policy directly influences both the overall completion time (makespan) and the memory footprint. On each GPU, executing the forward and backward tasks of the same microbatch close to each other reduces memory pressure by freeing stored activations earlier. Conversely, injecting more forward tasks into the pipeline helps minimize idle periods. When rematerialization is allowed – i.e., when selected activations produced by F tasks can be discarded and recomputed later before their corresponding B tasks – the scheduling problem becomes more complex: it combines task ordering with a decision problem specifying which activations to recompute and when.

In this paper, we focus on the third problem (scheduling under memory constraints and rematerialization). The number m of microbatches and the allocation of model stages to GPUs are assumed to be fixed. In practice, the maximum number of microbatches is constrained by the training procedure itself, while the allocation problem can be explored exhaustively: for a two-wave configuration and N GPUs, there are $N!$ possible ways to assign stages to GPUs.

Different scheduling algorithms (GPIPE [6], 1F1B [7], HANAYO [8], MEGATRON [1], Zero-Bubble [9]) can be interpreted as heuristic or partially optimal solutions to this global scheduling problem under specific assumptions.

The following sections formalize this problem and derive lower bounds on the minimal achievable completion time given an allocation and a memory constraint.

D. Pipeline Structure

A training step involves processing m microbatches. For each microbatch, the computation consists of:

- a **forward pass (F)** that produces intermediate activations, and
- a **backward pass (B)** that consumes these activations to compute gradients.

The pipeline can operate in different regimes, depending on how the model is partitioned and how many *waves* of microbatches are injected concurrently.

Single-wave pipeline: In the simplest configuration, the model is split into N stages, each mapped to one GPU. The single-wave model therefore corresponds to the case where the number of pipeline partitions equals the number of GPUs. Well-known scheduling variants in this regime include:

- the **fill–drain schedule** (GPIPE [6]), where all forwards are executed before any backward begins;
- the **1F1B steady-state schedule** [7], where forward and backward phases alternate after the pipeline is filled.

Both schedules achieve partial overlap between GPUs but differ in idle time and memory usage.

Two-wave pipeline: The pipeline can be refined by further partitioning the model into $2N$ (or more) sub-stages, still executed on N GPUs. Each GPU alternates between two independent sub-pipelines, effectively handling two interleaved *waves* of microbatches. This strategy increases concurrency and reduces bubbles by allowing partial overlap of the forward and backward phases across the two waves. Two-wave configurations appear in practical systems such as HANAYO [8] and MEGATRON [1], where each GPU hosts two "virtual stages". In Hanayo, layers i and $2N+1-i$ are allocated to GPU_i whereas in Megatron, layers i and $N+i$ are allocated to GPU_i. Because each sub-stage performs only half the work of a full stage, the duration of forward and backward computations is reduced accordingly, while the total pipeline depth effectively doubles.

Two-wave configurations can be extended to w-wave configurations by increasing the number of contiguous sub-stages in the partition of the model.

In the following, we denote by t_F and t_B the average times for the forward and backward computations of one full stage, typically with $t_B \approx 2t_F$. When considering two-wave strategies, the duration of one forward or backward computation will be reduced to $t_F/2$ or $t_B/2$ respectively. In all our examples, we use $t_F = 2$ and $t_B = 4$ to ensure that all computation times remain integer in the two-wave cases. When counting memory usage, we consider that one full activation uses one "memory slot": if storing a full activation requires M_A bytes, and the GPU has M_{GPU} available memory, then the memory constraint means that we can store at most $K = \left\lfloor \frac{M_{GPU}}{M_A} \right\rfloor$ activations in memory at any given time. For two-wave strategies, the result of a forward computation for one virtual stage only uses half a slot.

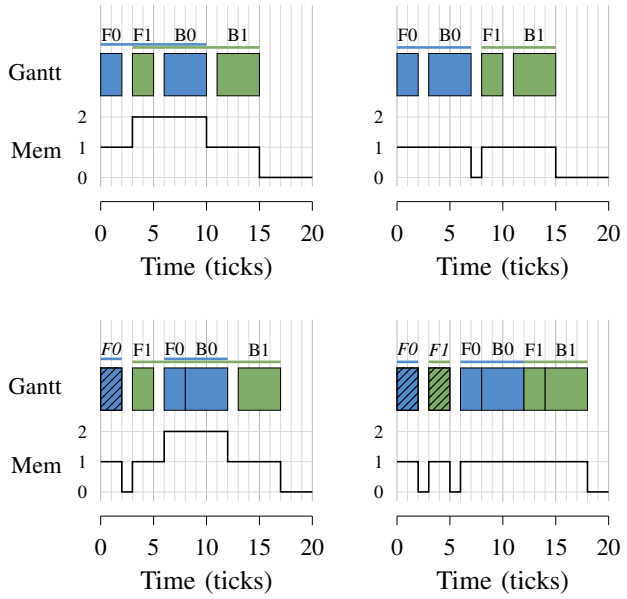


Fig. 1. Illustration of memory usage for two microbatches (blue and green) on a single GPU. Non-hashed blocks correspond to standard forward passes that retain activations until their backward phase, while hashed blocks denote rematerialized forwards that discard and later recompute activations. The lower plot shows the resulting memory footprint over time.

E. Memory Usage under Different Scheduling Strategies

To illustrate the memory management challenges and introduce the visual representations used throughout this paper, we consider in Figure 1 a single GPU executing two microbatches, shown in blue and green. For simplicity, dependencies originating from other GPUs and additional microbatches on the same GPU are not represented. By construction, for a given microbatch, the forward task (F) must precede its corresponding backward task (B), but the F and B tasks of different microbatches can be interleaved in various ways.

Moreover, each forward task can be executed in two distinct modes. In the first mode (non-hashed blocks in the figure), the activations computed during the F task are preserved in GPU memory until their associated B task consumes them. In the second mode (hashed blocks), the activations are immediately discarded once the F computation and the transfer of its output to the next GPU are complete; they need to be recomputed later, typically right before executing the corresponding B task. This latter case corresponds to the use of **rematerialization**.

Below each timeline, we depict the corresponding memory footprint associated with the two microbatches. The height of the curve represents the number of stored activations as a function of time, highlighting how task interleaving and rematerialization decisions directly affect the instantaneous memory usage.

F. Existing Pipeline Scheduling Strategies

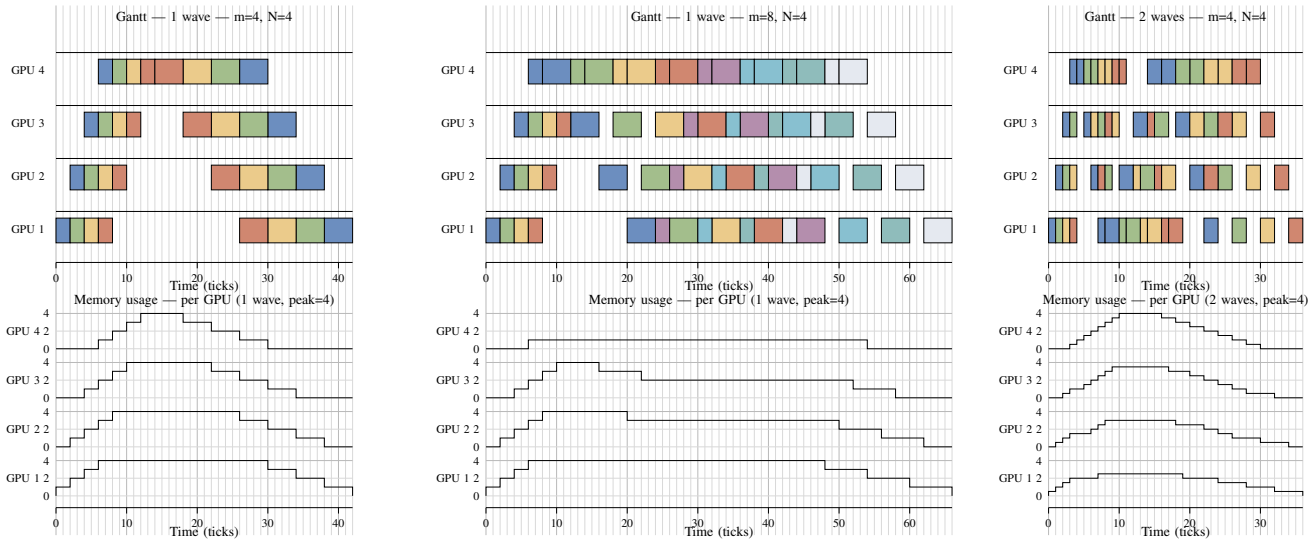
We now introduce several pipeline parallelism algorithms together with their associated memory consumption. All examples are illustrated for a configuration with four GPUs. These algorithms differ along three dimensions: (i) the number of injected microbatches, (ii) the number of concurrent waves (with possible generalizations beyond two waves for HANAYO and MEGATRON), and (iii) the strategy used to interleave forward (F) and backward (B) tasks. In all figures, each microbatch is represented by a distinct color, and the B tasks are twice as long as the corresponding F tasks. Since our study focuses on memory behavior, we systematically report the number of occupied memory slots over time for each algorithm.

The first method, GPIPE [6], is illustrated in Figure 2a for the case $m = 4$ and a single wave, though it trivially generalizes to any number of microbatches m. As m increases, pipeline efficiency improves because the bubble size remains constant, but memory requirement grows linearly with m. A first improvement in the single-wave regime was introduced by the 1F1B [7] algorithm (shown in Figure 2b), typically used with $m = 2N$. In 1F1B, each GPU starts with a few F tasks and then alternates between F and B tasks as soon as backward computations become available. Executing B tasks as early as possible allows intermediate activations to be released sooner, reducing the peak memory usage to four slots (for all GPUs except the last one). Equation 1 below shows that both GPIPE and 1F1B are optimal, in terms of makespan, for their respective values of N and m.

As we will see and formally demonstrate in Section III-B, increasing the number of waves reduces the makespan by further diminishing pipeline bubbles. Once multiple waves are introduced, one must decide how model stages are allocated across GPUs. Two main allocation strategies have been proposed in the literature. In MEGATRON [1], GPU i receives stages i and $i + N$; we refer to this configuration as a CYCLIC allocation. The standard MEGATRON schedule additionally defines the ordering of the forward and backward computations. It is typically defined for $m = 8$ microbatches, as depicted in Figure 3. The peak memory consumption occurs on GPU 1, with 11 simultaneous activations stored in memory, which requires 5.5 memory slots (each activation being half the size of those used in the single-wave setting). From a scheduling perspective, MEGATRON relies on the same "backward-as-soon-as-possible" policy as 1F1B.

An alternative stage allocation strategy was introduced by HANAYO [8], in which GPU i receives stages i and $2N+1-i$, improving upon the mirrored stage mapping proposed in CHIMERA [11]. We refer to this configuration as a MIRROR allocation. HANAYO is presented for two waves in Figure 2c with $m = 4$, but it can easily be generalized to a larger number of waves. Its peak memory consumption reaches 4 slots with 8 half-activations in memory on GPU N, which hosts two consecutive stages. We show in Section V that among all possible allocation schemes (CYCLIC, MIRROR, or arbitrary), CYCLIC is obtains the lowest possible bounds for the considered values of N and m, whereas MIRROR does not.

In the rest of this paper, we consider a fixed setting for



(a) GPipe, N=4, M=4 (b) 1F1B, N=4, M=8 (c) Hanayo (mirror), N=4, M=4

Fig. 2. Comparison of pipeline scheduling strategies with N=4 GPUs. Each subfigure shows the timeline of F/B tasks (colors = microbatches; B twice as long as F) and the corresponding memory footprint over time. Two-wave methods (Megatron, Hanayo) use half-size activation slots compared to single-wave (GPipe, 1F1B). Megatron is shown on Figure 3.

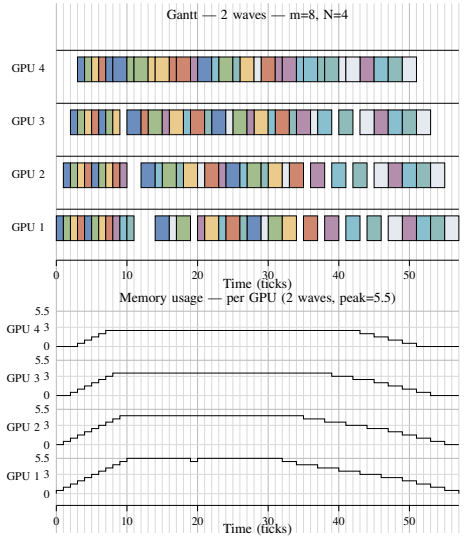


Fig. 3. Pipeline schedule for Megatron (cyclic), N=4, M=8

given values of N, m, t_F and t_B. Our goal is to obtain valid bounds on the *best* achievable makespan $T(K, w)$ for pipeline scheduling within this setting, depending on the number of waves w and the number of available memory slots K.

We can start with a remark that when K is large enough, the pipeline dependency constraints provide a lower bound on the makespan. Indeed, the GPU in charge of stage N starts working no sooner than time $(N-1)\frac{t_F}{w}$, and must stop no later than $(N-1)\frac{t_B}{w}$ before the end of the schedule. Since its workload is $m(t_F + t_B)$, we obtain:

$$T(\infty, w) \geq (N-1)\frac{t_F + t_B}{w} + m(t_F + t_B) \qquad (1)$$

We can see that this bound is reached by GPipe (Figure 2a), 1F1B (Figure 2b), and Megatron (Figure 3): the last GPU experiences no idle time in its compute interval.

III. Lower Bounds with very limited memory

In this Section we consider the case of the lowest possible available memory, which means that we can only store in memory the result of *one* forward operation. This restricted setting helps explain the arguments used to obtain our lower bounds, and allows for much stronger bounds for which we are actually able to obtain matching schedules.

Note however that the memory requirement is not the same depending on the number of waves: since with w waves, each forward operation is further divided into w sub-operations, storing one result only requires $\frac{1}{w}$ slots. Despite this inconsistency, the results are interesting and using more waves still allows for a lower makespan.

A. Single-wave case

With only one available memory slot, when computing a backward operation only the activation produced by the corresponding forward can be present in the memory. On the very last stage this does not necessarily provide a stronger constraint, since the 1F1B schedule allows the last stage to compute each backward operation immediately after the corresponding forward (as seen on Figure 2b).

However, let us consider stage $N-1$ and some arbitrary micro-batch j. If the corresponding activations are stored in memory after F_j, then because of the memory constraint no other operation can happen until B_j is computed. Because of the pipeline dependencies, the operations F_j^N and B_j^N need to be performed before B_j can be computed. This incurs an

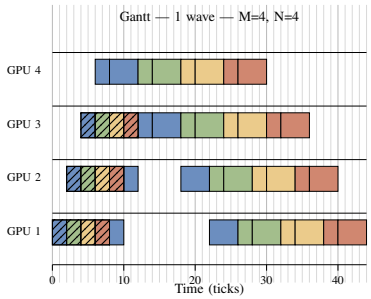


Fig. 4. Single-wave low-memory schedule for N=4, M=4: at most one activation is in memory at any time on each GPU.

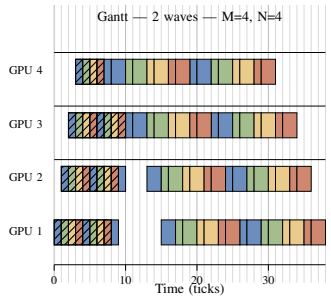


Fig. 5. Two-waves low-memory schedule for N=4, M=4: at most one half-activation is in memory at any time on each GPU.

idle time on GPU $N-1$ of duration at least $t_F + t_B$, during which no work can be performed.

On the other hand, if the activations for micro-batch j are *not* stored in memory, this only requires to occupy GPU $N-1$ for a duration t_F when recomputing the necessary activations right before the B_j operation. Since memry is not used, other useful work can be performed between these two steps, so that recomputing leaves more available time on the GPU than storing the activations.

This shows that there exists an optimal schedule in which GPU $N-1$ performs recomputations on *all* micro-batches. This incurs a computation time $m \cdot (2t_F + t_B)$. Because of pipeline dependencies, GPU $N-1$ starts computing no sooner than $(N-2)t_F$, and ends no later than $(N-2)t_B$ before the end of the schedule. This provides a lower bound on the makespan of the optimal schedule:

$$T(1,1) \geq (N-2)(t_F + t_B) + m \cdot (2t_F + t_B) \qquad (2)$$

This bound can actually be reached with a schedule where all GPUs except for GPU N recompute all activations, and where GPU N performs all operations in 1F1B order. The only requirement is that $t_B \leq m \cdot t_F$ so that GPU $N-1$ can be kept busy (with other micro-batches) after computing F_1 and before recomputing F_1 before B_1. Such a schedule is depicted on Figure 4.

B. Multi-wave case

The previous argument can be generalized to the case with $w > 1$ waves. It is indeed still possible to argue that on all GPUs except the one that processes the last stage (stage wN), the activations for all micro-batches need to be recomputed. Otherwise, the idle time incurred by the memory constraint is larger than the recomputation time required to rematerialize the activations.

We now consider the GPU k that processes stage N, and differentiate two cases. Either GPU k also processes stage wN, or it does not.

If it does not, then by the above argument all its forward operations must be recomputed. There are w stages and m micro-batches, and for each of them the computing requirement is $\frac{2t_F+t_B}{w}$. Because of pipeline dependencies with the $N-1$ stages before it, this provides a lower bound

$$\begin{aligned} T &\geq (N-1)\frac{t_F + t_B}{w} + w \cdot m\frac{2t_F + t_B}{w} \\ &\geq (N-1)\frac{t_F + t_B}{w} + m(2t_F + t_B). \end{aligned}$$

On the other hand, if GPU k *does* process stage wN, we instead consider the GPU k' that processes stage $N-1$. Since k' does not process stage wN, all its forward operations must be recomputed, and the same reasoning provides a lower bound

$$T \geq (N-2)\frac{t_F + t_B}{w} + m(2t_F + t_B).$$

In both cases, we can obtain a similar lower bound as in the single-wave case:

$$T(1,w) \geq (N-2)\frac{t_F + t_B}{w} + m \cdot (2t_F + t_B) \qquad (3)$$

As above, it is possible to construct a schedule that achieves a makespan equal to this bound. We use a cyclic Megatron-like assignment of stages, where GPU k processes stages $k, k+N, k+2N, ..., k+wN$. Each GPU processes its stages in order, ie starts with the m forward operations for the first stage, then the next m operations for the second stage, and so on. All forward operations are recomputed except for the operations of stage wN. Such a schedule (for $w = 2$) is shown on Figure 5.

If $m \geq N$, in addition to the $t_B \leq mt_F$ condition above, we can see that GPU $N-1$ is busy with forward operations until it starts processing backward operations. This shows that the schedule achieves the lower bound given on Equation (3).

IV. Lower bounds for a fixed memory limit

A. Single wave case

Let us consider a single-wave schedule, with K available memory slots. We study the memory usage of a specific GPU k. As before, because of pipeline dependencies, GPU k starts computing no sooner than time $(k-1) \cdot t_F$, and stops no later than $(k-1) \cdot t_B$ before the end of the schedule. We denote by C_k the available computing time for GPU k, and T the length of the schedule, so that

$$C_k + (k-1)(t_F + t_B) \leq T. \qquad (4)$$

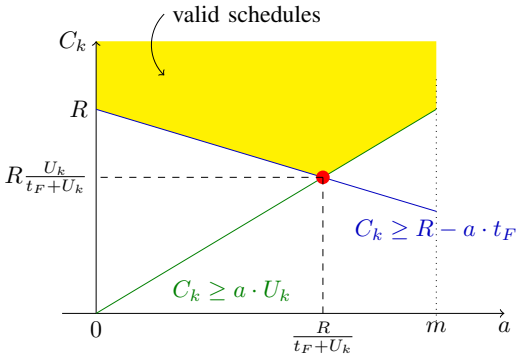


Fig. 6. Depiction of bounds for the single-wave case, as given by equations (5) and (6), with the zone of valid schedules containing all values (a, C_k) that satisfy both bounds. The lower bound (7) is obtained at the red spot: all valid schedules have a higher value of C_k.

During this available compute time on GPU k, let us look at the *cumulative* memory usage on GPU k: if some activation is stored in memory for τ time units, we will say that it occupies τ cumulative memory. Since there are K slots for each time unit, the total cumulative memory occupation during time C_k is bounded by $K \cdot C_k$.

Let us denote as a the number of activations of GPU k that are *kept* in memory in the schedule between their forward and backward operations. Because of pipeline dependencies, the time between the forward and the backward operations is at least $(N - k + 1) \cdot (t_F + t_B)$, since we have to wait for the results of the computations on the $N - k + 1$ stages after stage k. Each activation stored in memory uses at least $(N - k + 1) \cdot (t_F + t_B)$ cumulative memory, so that we can write the following bound on total cumulative memory usage:

$$a \cdot (N - k + 1)(t_F + t_B) \leq K \cdot C_k \tag{5}$$

In addition, each activation which is *not* kept in memory induces a recomputation of the forward operation, and there are $m - a$ such activations, so that we can also express a bound on the total computational load:

$$m(t_F + t_B) + (m - a)t_F \leq C_k \tag{6}$$

For simplicity, let us denote $U_k = (N - k + 1)\frac{t_F+t_B}{K}$, and $R = m(2t_F + t_B)$. For a schedule which stores a activations in memory, the two above bounds can be written as $C \geq \max(a \cdot U_k, R - a \cdot F)$. These bounds with the corresponding valid zone are depicted in Figure 6. The best possible choice for a is when both values in the max are equal, which leads to the following bound for the compute time of GPU_k:

$$C_k \geq R \cdot \frac{U_k}{U_k + t_F} \tag{7}$$

We obtain such a bound for all GPUs, and combined with the pipeline dependencies before and after the available computing time as written in equation (4), we obtain:

$$T(K, 1) \geq \max_k \left((k-1)(t_F + t_B) + R \cdot \frac{U_k}{U_k + t_F} \right) \tag{8}$$
$$\text{where } U_k = (N - k + 1) \cdot \frac{t_F + t_B}{K}$$
$$\text{and } R = m \cdot (2t_F + t_B)$$

B. Multi-wave lower bound

We now show how this analysis can be generalized to a multi-wave schedule, with K memory slots. For simplicity, we perform the analysis for a 2-wave schedule, but all the arguments can be easily generalized to any value of w.

We consider a given GPU k, responsible for stages f_k and l_k. For notational simplicity, we will drop the index k when it is clear from context, and use $f = f_k$ and $l = l_k$. It is the time between computing the forward of stage f on microbatch 1, and computing the backward of stage f on microbatch m. As before we can write the pipeline dependency constraint, but now each operation is twice shorter:

$$C_k + (f - 1)\frac{t_F + t_B}{2} \leq T. \tag{9}$$

We again analyze the cumulative memory usage for GPU k, however we now have two different kinds of activations, either produced by F_f or by F_l. As before, an activation produced by F_f has to stay in memory for at least $\tau_f = (2N - f + 1) \cdot \frac{t_F+t_B}{2}$ time units. Similarly, an activation produced by F_l stays in memory for at least $\tau_l = (2N - l + 1) \cdot \frac{t_F+t_B}{2}$ time units.

Let us note a^f and a^l the number of activations (of stage f and l respectively) which are kept in memory between the forward and backward computations, and denote $a = a^t + a^l$ the total number of activations kept in memory at some point. We can write a similar bound as before based on the total cumulative memory usage for these activations, which only use half a memory slot:

$$\begin{aligned} K \cdot C_k &\geq \frac{1}{2}(a^f \tau_f + a^l \tau_l) \\ 2K \cdot C_k &\geq \frac{t_F + t_B}{2}\left((a^l + a^f)(2N + 1) - (a^l l + a^f f)\right) \\ C_k &\geq \frac{t_F + t_B}{4K}\left(a \cdot (2N + 1 - f) - a^l \cdot (l - f)\right) \end{aligned} \tag{10}$$

Since the available compute time is C_k, the total memory usage is bounded by $S \leq K \cdot C_k$, which provides an upper bound on C_k. In addition, each activation which is *not* kept in memory induces a recomputation, so that:

$$\begin{aligned} C_k &\geq m(t_F + t_B) + (2m - a) \cdot \frac{t_F}{2} \\ &\geq m(2t_F + t_B) - a \cdot \frac{t_F}{2} \end{aligned} \tag{11}$$

The schedule may use any value for a^l and a^f, however we can see in (10) that for a fixed value of a, higher values of a^l result in lower total memory usage, with no change to the amount of recomputation required. We will thus consider

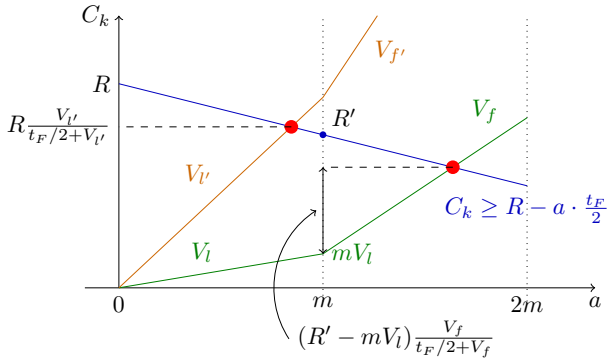


Fig. 7. Depiction of bounds for the multi-wave setting, showing two possible cases: when $mV_l < R'$ (green line), the intersection happens for $a > m$, and when $mV_{l'} > R'$ (orange line), the intersection happens for $a < m$.

schedules in which $a^l = a$ if $a \leq m$, and $a^l = m$ and $a^f = a - m$ if $a > m$. For such a schedule, we can write (10) as:

$$C_k \geq \frac{t_F + t_B}{4K} \cdot \begin{cases} a(2N + 1 - l) & \text{if } a \leq m \\ a(2N + 1 - f) - m(l - f) & \text{otherwise} \end{cases} \tag{12}$$

As before, we can combine the compute bound (equation (11)) and the memory bound (equation (12)) by using the value of a for which both bounds are equal. This time however, the bound given by equation (12) is piecewise affine, with slopes $V_l = (2N - l + 1) \cdot \frac{t_F+t_B}{4K}$ for $0 \leq a \leq m$ and $V_f = (2N - f + 1) \cdot \frac{t_F+t_B}{4K}$ for $m \leq a \leq 2m$. We can thus distinguish two cases: either both bounds are equal for $a < m$ or for $a > m$. We denote with $R' = m(3F/2 + B)$ the value of the compute bound when $a = m$; the value of the memory bound for $a = m$ is simply $m \cdot V_l$. Figure 7 shows the notations, with the two cases depicted.

We can thus claim that $C_k \geq X_k$ for any schedule, where X_k is the value of both bounds when they are equal and can be written as:

$$X_k = \begin{cases} R \cdot \frac{V_l}{V_l + t_F/2} & \text{if } m \cdot V_l > R' \\ mV_l + (R' - mV_l) \cdot \frac{V_f}{t_F/2+V_f} & \text{otherwise} \end{cases} \tag{13}$$

If $X_k < m(t_F + t_B)$ (this happens if the bounds are equal for $a > 2m$), we can instead use the total computation bound that does not depend on memory usage: $C_k \geq m(t_F + t_B)$. As before, we obtain these bounds for all GPUs, and we can combine them with equation (9) to get the final bound:

$$T(K, 2) \geq \max_k \left((f_k - 1)\frac{t_F + t_B}{2} + \max(X_k, m(t_F + t_B)) \right) \tag{14}$$

Like the previous ones, this bound makes no assumption about the ordering and the rematerialization decisions of the schedule. However, it is expressed for a fixed assignment of stages to GPUs, since it is based on the values of f_k and l_k for each GPU k. We will analyze in the next Section how the value of this bound changes for different stage assignments.

All of the arguments above can be generalized to obtain a similar bound for a w-wave schedule: the expression of the memory bound (12) is piecewise affine with w pieces, and the value of B_k in equation (13) is expressed with w cases instead of just 2. The rest of the proof is unchanged.

V. Comparison of lower bounds

In this section, we empirically evaluate and compare the lower bounds derived in Sections III and IV. We adopt the same experimental setting as before, with $t_F = 2$ and $t_B = 4$ time units. We consider configurations with $N = 8$ and $N = 16$ GPUs, and vary the number of microbatches from $m = N$ up to $m = 20N$. The following bounds are evaluated:

- The RecomputeAll bounds derived in Section III, corresponding to the case of a single available memory slot. Each value of the number of waves w yields both a lower bound and an associated schedule.
- The NoRemat bound, which represents the critical-path time of an ideal schedule assuming infinite memory. As for the previous case, a bound and a corresponding schedule are defined for each number of waves w.
- The Single bound, derived in Section IV-A (Equation (8)), which provides a lower bound on the makespan of any single-wave schedule for an arbitrary number of memory slots K.
- Bounds for any schedule using the Cyclic stage assignment, where GPU k processes stages $(k, N + k, 2N + k, \dots)$, inspired by the Megatron schedule. This bound (Equation (14)) can be computed for any number of memory slots K and any number of waves $w > 1$.
- Similarly, bounds for schedules with the Mirror stage assignment, inspired by Hanayo, in which GPU k processes stages $(k, 2N-k+1, 2N+k, 4N-k+1, \dots)$.
- Finally, for $N = 8$ and $w = 2$, we compute the two-wave bound for all possible stage-to-GPU assignments using Equation (14). The resulting bound is denoted as Exhaustive.

The results are displayed in Figure 8, with one plot for each combination of N and m. The RecomputeAll and NoRemat bounds appear as single points, as they do not depend on the number of memory slots K. All other bounds are computed for each value of K, ranging from 1 to N.

The first observation highlights the well-known effect of the number of waves: when m is sufficiently small, increasing the number of waves provides a substantial improvement in makespan, as clearly visible for the NoRemat and RecomputeAll schedules. This benefit, however, is much more pronounced when increasing the number of waves from one to two than when going from two to three, indicating diminishing returns beyond two waves.

Another important observation is that the cost of rematerialization remains relatively small when m is low. For instance, with $N = m = 8$, the makespan increases only from 90 to 100 between the NoRemat and RecomputeAll schedules for a single wave. When m becomes large, however, the

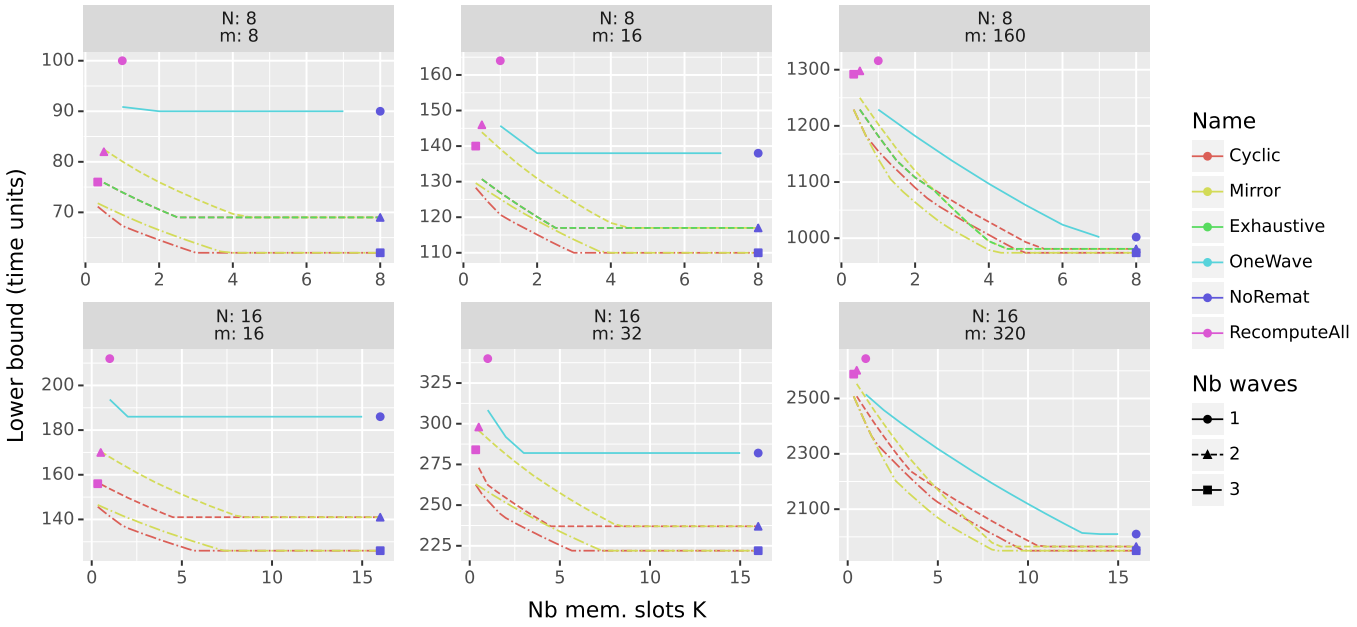


Fig. 8. Empirical comparison of the bounds of Sections III and IV.

recomputation overhead can reach up to 33%, as predicted by theory and confirmed by the two rightmost plots.

For smaller values of m, we also observe that single-wave schedules can reach the critical path with a modest amount of memory slots. Starting from $K = 2$ or 3 slots, the ONEWAVE bound already matches the NOREMAT value. Our results therefore do not preclude the existence of low-memory, makespan-optimal single-wave schedules, and identifying algorithms capable of achieving such schedules would be a promising direction for future work. It should be noted, however, that such optimal schedules are not necessarily rematerialization-free: the critical-path bound is defined for the last GPU, so achieving this bound requires the last GPU to avoid recomputation, while the other GPUs can use some of their idle time to perform rematerialization and thereby reduce memory pressure.

This property, however, does not hold when m is large. In this case, single-wave schedules require a significantly larger amount of memory to reach the critical path, which is consistent with the widening gap between the NOREMAT and RECOMPUTEALL bounds as m increases.

When considering multi-wave approaches, we observe that the number of memory slots required to reach the critical path remains relatively small. In most cases, it is possible to consistently achieve the optimal makespan (as limited by the critical path) with only $K = N/2$ memory slots. This property holds for both small and large values of m. Designing practical rematerialization schedules capable of reaching this bound would therefore be a highly interesting direction for future research.

When comparing the different multi-wave approaches, we note that the CYCLIC assignment is significantly more efficient than the MIRROR assignment for small values of m. According to our exhaustive enumeration for $N = 8$, when $m = 8$ or $m = 16$, CYCLIC achieves the lowest possible lower bound among all possible stage-to-GPU assignments. For larger values of m, however, no single stage assignment consistently dominates. In the two-wave case, the lowest bound for smaller values of K is obtained with the CYCLIC assignment, while for larger K the MIRROR assignment becomes superior. For three waves, MIRROR achieves the lowest bound for all values of K. Interestingly, our exhaustive search confirms that there is no intermediate configuration for two waves: no stage assignment yields a lower bound than those achieved by either CYCLIC or MIRROR. The advantage of the MIRROR assignment can even cancel the benefits of additional waves. For instance, when $m = 20N$, there exist values of K for which the lower bound of the two-wave MIRROR schedule is lower than that of the three-wave CYCLIC schedule. For $N = 16$, this occurs when $K = 7$ or 8.

VI. RELATED WORKS

Several studies have analyzed the efficiency limits of pipelined training from different perspectives.

From a systems viewpoint, the PIPEDREAM framework [7] addresses pipeline inefficiency by introducing asynchronous execution, where each GPU continuously processes forward and backward tasks using its local copy of the model weights. This design achieves high hardware utilization but causes gradient staleness, since backward passes may use outdated weights. PIPEDREAM-2BW [12] mitigates this issue through

double-buffered weights, partially restoring synchronous semantics at the expense of additional memory. Colin et al. [13] investigated the asymptotic convergence rate of distributed optimization algorithms executed under such an asychronous pipeline model. Their analysis shows that overlapping multiple microbatches in the pipeline introduces a delay in gradient updates that fundamentally slows convergence, establishing lower bounds that depend on both the pipeline depth and the number of in-flight microbatches. Although their framework focuses on optimization dynamics rather than execution time, it provides a complementary theoretical foundation: even with perfect resource utilization, pipeline delays impose an intrinsic inefficiency. In contrast, our work focuses on a fully synchronous regime in which all gradients are computed using the current model parameters. In this setting, pipeline scheduling affects the execution time and memory footprint, but not the convergence rate, allowing us to derive formal lower bounds on achievable efficiency under strict memory constraints.

In addition to the classical pipeline schedules discussed in Section II-F, recent work has proposed more aggressive strategies to further reduce pipeline bubbles. Among them, the ZEROBUBBLE approach [9] decouples the backward phase into two sub-stages: one computing gradients with respect to activations and another updating the model weights. This decoupling allows for improved overlap between forward and backward computations and can almost eliminate idle time in the pipeline. However, these methods require retaining intermediate activations and partial gradients for longer periods, significantly increasing memory consumption. Using rematerialization for these partial gradients has been proposed in [14], but this results in increased computational overhead. Consequently, these decoupled strategies are not well suited to memory-constrained regimes, which are the focus of our present analysis.

Another closely related body of work studies the partitioning and steady-state scheduling of linear pipeline skeletons, where a sequence of dependent tasks is repeatedly executed over a stream of data items. In this context, the main objective is to partition the pipeline stages among processing elements so as to maximize the steady-state throughput or minimize the makespan under communication costs [15]–[17]. These studies assume unidirectional data flow and focus on load balance and communication–computation overlap, but do not account for the bidirectional dependencies and memory retention induced by backward propagation in DNN training. Our work differs by explicitly modeling both forward and backward passes and by deriving formal lower bounds that incorporate memory constraints and rematerialization decisions.

Complementary to pipeline-based analyses, a rich line of research has addressed activation rematerialization for general computation graphs. Methods such as CHECKMATE [18], MOCCASIN [19], ROCKMATE [20] and HIREMATE [21] formulate activation checkpointing as an optimization problem over a directed acyclic graph (DAG), aiming for a bounded peak memory usage while minimizing the total recomputation cost. These approaches focus on single-pass execution of a static DAG, typically a single forward–backward iteration of a neural network, without considering steady-state reuse or pipeline overlap across multiple microbatches. In contrast, our work studies rematerialization in the context of repetitive pipelined execution, where memory reuse and interleaved scheduling fundamentally alter both the attainable makespan and the structure of optimal recomputation strategies.

VII. LIMITATIONS

To the best of our knowledge, this work provides the first rigorous derivation of lower bounds for pipeline parallel training under explicit memory constraints. Our analysis deliberately relies on a simple and tractable model that captures only two architectural parameters: the computation time of each task—allowing the inclusion of tensor, sequence, or context parallelism—and the available GPU memory. This abstraction highlights the fundamental trade-offs between compute efficiency and memory usage. More detailed architectural features, such as the explicit modeling of input activations, inter-GPU communication, or offloading mechanisms, could naturally be incorporated into the analysis to improve realism. However, doing so would considerably increase the analytical complexity and risk obscuring the main conceptual insights provided by the lower bounds, which are intended to serve as interpretable guides for designing efficient scheduling strategies.

A first limitation concerns communication costs and intermediate activation transfers. Our current model assumes negligible communication times between GPUs. In practice, inter-GPU links such as NVLink or NVSwitch introduce latency and bandwidth constraints that may slightly affect the critical path. While integrating such effects into the analysis is possible, it would significantly increase the complexity of the lower bound derivations without fundamentally altering the qualitative conclusions regarding pipeline efficiency. Extending the model to account for communication overheads would require an explicit representation of the interconnect topology and link bandwidths, which we leave for future work.

A second limitation concerns alternative memory-saving strategies such as offloading [22]. Offloading transfers activations to host memory or other storage devices instead of discarding or recomputing them, and is therefore orthogonal to rematerialization. However, it places additional pressure on the PCIe or NVLink interconnects and its benefit depends strongly on architectural parameters such as the hidden dimension, sequence length, and available bus bandwidth. When spare bandwidth is available, offloading could indeed complement rematerialization, but analyzing its impact would require a significantly more complex architectural model involving CPU memory and interconnect contention, which falls outside the scope of this study.

CONCLUSION

This work provides, to the best of our knowledge, the first rigorous analytical study of pipeline parallel training under explicit memory constraints. By modeling only the

computation time of each task and the available GPU memory, we derive tight lower bounds that capture the fundamental trade-offs between efficiency and memory usage. Despite the simplicity of this model, ignoring communication latency and architectural details, it reveals non-trivial structural properties of optimal schedules and provides usable insights for the design of future training algorithms.

The derived lower bounds establish theoretical limits that any pipeline-parallel training strategy must satisfy in memory-constrained regimes. They thus form a basic reference for evaluating new scheduling algorithms, independently of implementation or hardware-specific optimizations. Beyond their evaluative role, these bounds also provide intuition on the lifetime of activations and the role of recomputation in balancing throughput and memory pressure. In particular, our analysis shows that systematic recomputation is often inexpensive: even in extremely low-memory regimes ($K = 1$), the additional computational overhead remains bounded to a ratio of $\frac{4}{3}$. As a result, a two-wave pipeline with massive rematerialization outperforms a single-wave schedule that benefits from infinite memory.

Overall, this theoretical framework, despite its minimal assumptions, offers valuable qualitative guidance for both system designers and algorithmic researchers. It demonstrates that carefully combining memory-aware scheduling and controlled rematerialization can achieve near-optimal efficiency, even in the most constrained hardware environments.

REFERENCES

[1] 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.

[2] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," *arXiv preprint arXiv:2105.13120*, 2021.

[3] J. Fang and S. Zhao, "Usp: A unified sequence parallelism approach for long context generative ai," *arXiv preprint arXiv:2405.07719*, 2024.

[4] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, no. 112, pp. 1–49, 2019.

[5] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[6] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[7] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[8] Z. Liu, S. Cheng, H. Zhou, and Y. You, "Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[9] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble (almost) pipeline parallelism," in *The Twelfth International Conference on Learning Representations*, 2024.

[10] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[11] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[12] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[13] I. Colin, L. Dos Santos, and K. Scaman, "Theoretical limits of pipeline parallel optimization and application to distributed deep learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[14] A. Aguila-Multner, O. Beaumont, L. Eyraud-Dubois, and J. Gusak, "Optimized Forward-Backward Rematerialization for Memory-Efficient Pipeline Parallel Training," Jul. 2025, working paper or preprint. [Online]. Available: https://inria.hal.science/hal-05151601

[15] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 151–162, 2006.

[16] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling: Models and algorithms," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, pp. 1–36, 2013.

[17] D. Orhan, Y. Idouar, L. L. Pilla, A. Cassagne, D. Barthou, and C. Jégo, "Scheduling strategies for partially-replicable task chains on two types of resources," in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2025, pp. 896–905.

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

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

[20] 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 *International Conference on Machine Learning*. PMLR, 2023, pp. 42 018–42 045.

[21] J. Gusak, X. Zhao, T. Le Hellard, Z. Li, L. Eyraud-Dubois, and O. Beaumont, "Hiremate: Hierarchical approach for efficient re-materialization of neural networks," in *Forty-second International Conference on Machine Learning*.

[22] X. Wan, P. Qi, G. Huang, M. Lin, and J. Li, "Pipeoffload: Improving scalability of pipeline parallelism with memory optimization," *arXiv preprint arXiv:2503.01328*, 2025.