Joint Optimization of Gradient Checkpointing and Tensor Parallelism for Model Training

Xiaoxuan Liu
UC Berkeley
xiaoxuan_liu@berkeley.edu

Siddharth Jha
UC Berkeley
sidjha@berkeley.edu

Alvin Cheung
UC Berkeley
akcheung@cs.berkeley.edu

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

Abstract

As the trend towards larger and more complex models continues, efficient training methods become critical. Two potential solutions to memory issues include model-parallel training and memory optimization techniques. Previous work optimizes those techniques in isolation, leading to suboptimal performance. A joint optimization approach can lead to more efficient and effective training, but the large solution space makes it challenging to find an optimal solution. In this paper, we propose KIWI, an optimizer that *jointly* considers both tensor parallelism and memory optimization techniques (e.g., gradient checkpointing) at the operator level. Using effective search space pruning methods, we manage computational feasibility while retaining solver optimality. Importantly, KIWI maintains training accuracy and does not disrupt the training process. We demonstrate the effectiveness of our approach through experiments on transformer models, showing that KIWI can reduce training overhead by up to 2.2× compared to optimizing each aspect independently.

1 Introduction

Recent years have witnessed a rapid growth in the size of deep learning models, resulting in improved performance across various domains such as natural language tasks [12, 9, 31] and vision tasks [13, 26]. However, training these models requires a considerable amount of on-device GPU memory. Unfortunately, the increase of GPU memory capacity has been relatively slow [3–5, 1], leading to a fundamental barrier to the development of large neural network (NN) models.

To mitigate memory constraints in training large models, two main strategies have been developed: memory optimization methods and distributed training. Among memory optimization methods, gradient checkpointing is a prominent technique that discards intermediate results during the forward pass and recomputes them during the backward pass. On the other hand, distributed training segments the network into subgraphs, often sequential layers (pipeline parallelism), with computations within each subgraph further divided across multiple devices (tensor parallelism). Modern practices [37, 17, 29] typically employ gradient checkpointing in conjunction with different parallelism forms to train large-scale models efficiently. Other memory optimization methods such as quantization [10, 24] and low-rank approximation [15] exist but may hinder the model's performance, unlike gradient checkpointing and forms of parallelism.

This paper explores the integration of gradient checkpointing (GC) and tensor parallelism (TP), wherein the various applications of TP to an operator are termed as 'sharding strategies'. Each layer

Preprint. Under review.

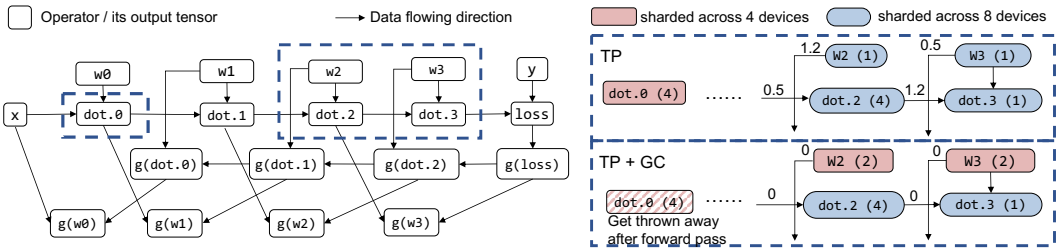


Figure 1: The graph gives a example of different decisions made by TP and TP+GC. The left graph illustrates the computation graph of a four-layer MLP, with nodes featuring different execution costs highlighted within dashed boxes. The right graph compares the strategies of TP and TP+GC. The numbers within brackets indicate the memory usage of each operator in megabytes (MB). Numbers on the arrows indicate the resharding cost (communication between devices to redistribute the output of an operator into the format required by the subsequent operator).

offers the flexibility to select a combination of recomputation and sharding strategies. However, current practices for checkpointing activations often overlook the TP strategy. For instance, a common heuristic in transformer-based models only checkpoints activations of matrix multiplications [2] as they are expensive to compute, neglecting system-specific settings and TP strategy.

However, we find that the GC strategy should be closely related to the TP strategy. While both techniques aim to reduce memory usage, TP introduces additional communication overhead, while GC incurs extra computational overhead. Therefore, the optimal decisions may vary depending on the specific hardware settings. For example, suppose the training system contains fast communication links between GPUs. In such a setting, since communication is relatively cheap, an optimal plan would consist of aggressive TP with minimal GC. Conversely, given the same number of available GPUs, if the training system uses slow communication links between GPUs, then it would be better to use a minimal TP strategy with aggressive GC.

Figure 1 provides an example that compares the impact of TP alone versus TP+GC on a four-layer MLP running on 8 GPUs with a slow inter-GPU connectionWith TP alone, weights w_2 and w_3 are distributed across 8 devices, each consuming 1 MB of memory. This approach incurs a significant resharding cost because the backward operators rely on weights (w_2, w_3) in different formats. In contrast, when employing GC, we can discard the intermediate result $dot.0$ before computing $dot.2$, effectively saving memory for w_2 and w_3. Consequently, w_2 and w_3 can be sharded across 4 devices (with two devices replicating the same shard). This optimization reduces the resharding cost for w_2 and w_3. Moreover, it mitigates resharding costs for $dot.2$ and $dot.3$, allowing the adoption of different sharding strategies that bypass the need for resharding. By incorporating gradient checkpointing alongside tensor parallelism, we achieve more efficient training that minimizes the resharding cost and enhances overall performance (reducing total execution time from 11.4ms to 8.6ms for the example).

In summary, the paper makes the following contributions:

1. We propose the idea of jointly optimizing checkpointing and sharding, and formulate the problem as a quadratically constrained quadratic program (QCQP).
2. We demonstrate how to reduce the complex search space of joint optimization to make the problem feasible to solve on large models such as transformers.
3. We implement KIWI and show its effectiveness in reducing training overhead by up to 2.2× compared to the optimal tensor parallelism strategy without considering GC. Furthermore, KIWI surpasses expert-designed GC heuristics when evaluated on transformer models.

2 Related Work

2.1 Memory Optimization Methods (MOM)

Memory optimization methods play a crucial role in training large models. They enable the training of models that would otherwise exceed the available memory. Moreover, these methods facilitate training with larger batch sizes, resulting in improved throughput and preventing GPU underutilization.

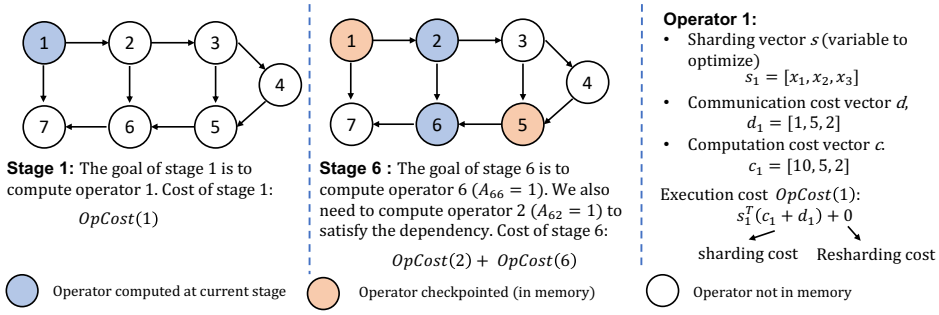


Figure 2: Example of calculating objective values.

Prominent memory optimization methods incude offloading [16, 34, 28, 30], quantization [10, 24], and gradient checkpointing [11, 21, 18, 19]. Offloading transfers data from the constrained GPU memory to the more plentiful CPU memory. Quantization, on the other hand, compresses activations to reduce their memory footprint. Lastly, gradient checkpointing discards activations during the forward pass and recomputes them during the backward pass as needed. More recent work [8, 27] looks into ways of combining different memory optimization techniques. Although KIWI optimizes gradient checkpointing, it can also be applied with orthogonal memory optimization methods such as quantization and offloading.

2.2 Gradient Checkpointing

Gradient checkpointing [11, 21, 18, 19] is a technique that discards activations in the forward pass and recomputes them in the backward pass as needed. This approach involves a trade-off between memory usage and computation cost. The challenge lies in finding the optimal balance – avoiding unnecessary re-computation while ensuring enough re-computation to prevent memory overflows. Checkmate [18] automatically determines the optimal strategies by solving an ILP that minimizes execution time while ensuring that the peak memory does not exceed the device limit.

2.3 Model Parallelism

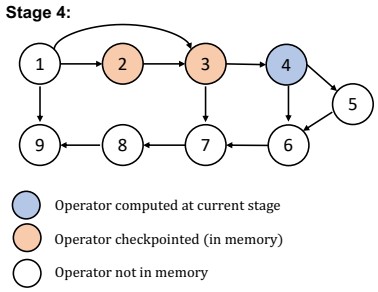
Model parallelism splits computation among multiple GPUs. The two classes of model parallelism are tensor parallelism [22, 35, 36] and pipeline parallelism [17, 14, 23]. Pipeline parallelism partitions the computation graph into consecutive layers and runs each partition on separate GPUs. Tensor parallelism partitions the input to an operator and runs the operator in parallel on multiple GPUs. Pipeline parallelism and tensor parallelism are easy to combine and are often applied together to train large models.

2.4 Combining Gradient Checkpointing With Parallelism

The automated joint application of gradient checkpointing with parallelism is still an open problem. Prior work such as Checkmate [18] and Alpa [37] independently solve for optimal gradient checkpointing and sharding, respectively. Colossal-Auto [25] attempts to solve the joint optimization problem but still runs the gradient checkpointing and the sharding pass separately due to the large search space. Piper [32] solves a dynamic programming problem to determine an optimal method of combining parallelization with gradient checkpointing. However, Piper solves at the granularity of entire layers and does not consider individual operators. Additionally, Piper is not fully automated since it requires that combinations of tensor parallelization strategies be manually enumerated and passed as input.

3 Problem Formulation

A computation graph, denoted as $G = (V, E)$, is a directed acyclic graph where nodes ($|V|$) represent operations that produce values (e.g., tensors), and edges symbolize dependencies between operators. Given G, the objective is to find an optimal gradient checkpointing and sharding strategy that



Memory Freed in Stage 4:
We free operators 1 and 2.
$L_4 = \{1, 2\}$: operators 1 and 2 are not needed in the forward pass once stage 4 begins.
$F_{2,3} = 1$: operator 2 is no longer needed after stage 3.
$z_1 = 1$: operator 1 is needed by backward operator 9.
$z_2 = 0$: operator 2 does not have backward dependency.
Given $m_1 = 0$, and $m_2 = 1$, we have
$memFreed_4 = s_2^T mem_2 + (1 * s_1^T mem_1 + 0 * s_2^T mem_2) = s_1^T mem_1 + s_2^T mem_2$

Memory Recomputed in Stage 4:
We never recompute in the forward pass, $memRecomputed_4 = 0$.

Total Memory Consumption in Stage 4:
$U_4 = U_3 + s_4^T mem_4 + memRecomputed_4 - memFreed_4$
$= U_3 + s_4^T mem_4 - s_1^T mem_1 - s_2^T mem_2$

Figure 3: Example of calculating memory in the forward stage. In stage 4, we can free operator 1 as it is not checkpointed and no longer needed to execute the forward pass. We also free operator 2, although it is checkpointed, because it is not needed in the remainder of the computation.

minimizes total execution time C while satisfying a memory limit given by the user. Therefore, for each operator in the graph, KIWI must determine a parallelization strategy for its computation, alongside a schedule dictating when to release a given node's memory and when to recompute it.

3.1 Joint Optimization

At a high level, we divide a computation graph consisting of $|V|$ operators into $|V|$ stages, with the goal of stage t to compute operator t. As shown in Figure 2 stage 6, it is possible to compute more than one operator in a single stage to satisfy dependencies. In each stage, we may choose to recompute an operator, checkpoint an operator, or free an operator's memory. The objective is to minimize the total cost of computation across stages:

$$\underset{A,B,U,F,s,e}{\arg\min} \sum_{t=1}^{|V|} \sum_{i=1}^{t} A_{ti} * [s_i^T (c_i + d_i) + \sum_{(v_j, v_i) \in E} e_{ji}^T R_{ji}] \tag{1}$$

Objective: The total cost comprises two components: sharding cost and resharding cost. As depicted in Equation 1 and Figure 2, $s_i^T (c_i + d_i)$ signifies the sharding cost, with s_i representing a one-hot encoding of operator i's sharding strategies. Since the operator can be sharded in varied manners, each strategy carries a corresponding computation (c) and communication (d) cost. Resharding costs arise when operator i, using operator j as an input, requires a different sharding strategy for operator j's output. As a result, communication between devices might be necessary to reshard operator j into the format required by operator i. This cost is denoted by $\sum_{(v_j, v_i) \in E} e_{ji}^T R_{ji}$ in our model, where e_{ij} is a one-hot encoding representing the resharding strategy between operators i and j. R_{ji}, a constant vector, encompasses various resharding costs between operators j and i.

The checkpoint schedule is determined by the variable A_{ti}, which represents whether operator i is (re)computed in stage t. If we choose to compute v_i in stage i and recompute v_i in stage t_1, both A_{ii} and $A_{t_1 i}$ will be set to 1. This implies that the schedule incurs the costs of sharding and resharding for v_i twice in different stages.

Constraints: Our optimization problem hinges on two types of constraints: dependency and memory constraints. The former ensures an operation is computed only once all its inputs are ready, thereby satisfying all dependencies for a given operator. The latter ensures peak memory usage at any stage doesn't exceed the device's memory limit. We employ the auxiliary variables B, U, and F in Equation 1 to articulate these constraints. Despite not appearing directly in the objective function, they're pivotal in the problem's formulation. As these constraints echo those used in Checkmate, we detail the functionality of these variables and constraints in the Appendix.

3.2 Reduce Problem Complexity

In preliminary experiments, we discovered that resolving the optimization problem outlined in Equation 1 for large models (e.g., transformers) is time-consuming. As noted in Section 4.5, it took 26

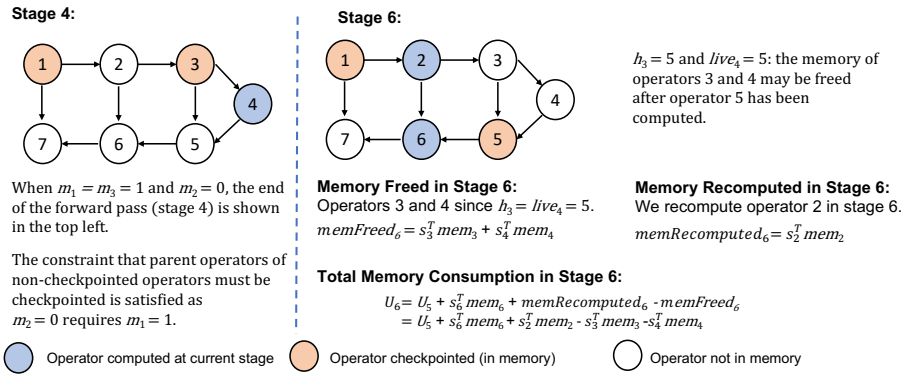


Figure 4: Example of calculating memory in the backward stage.

minutes to identify the optimal solution for a 7-layer MLP comprising merely 26 operators. To expedite this process, we proposed a new formulation designed to navigate a reduced search space and facilitate resolution on large networks. KIWI employs three simplifications to pare down the search space. (1) if an operator is recomputed, none of its inputs are recomputed; (2) once an operator's output is recomputed, it remains in memory; (3) the memory for forward operators not checkpointed is instantly freed once they're no longer required for forward pass computations.

3.2.1 Objective and Optimization Variables

$$\arg\min_{m,s,e,U,F} \sum_{i=1}^{|V|} (1 + (1 - m_i)z_i)[s_i^T(c_i + d_i) + \sum_{(v_j,v_i)\in E} e_{ji}^T R_{ji}] \tag{2}$$

We initiate by simplifying the objective through eliminating the concept of stages.We revise the objective from Equation 2 by incorporating $(1 - m_i)z_i$ into the objective function as a replacement for A_{ti}. This alteration eradicates the necessity for summing over stages t, thereby reducing the quadratic terms in the objective from $O(|V|^2 + |V||E|)$ to a more manageable $O(|V| + |E|)$.

As we no longer consider the concept of stages, we utilize a binary variable, m_i, to indicate whether we should free the memory of node i during the forward pass (i.e., m_i=0 implies freeing i's memory). Additionally, we introduce the constant z_i, which equals 1 if and only if operator i serves as an input to any operator in the backward pass. The inclusion of z_i ensures that we only recompute a forward node when it is required by the backward pass.

Similarly, $s_i^T(c_i + d_i)$ represents the sharding cost, while $\sum_{(v_j,v_i)\in E} e_{ji}^T R_{ji}$ denotes the resharding cost. The variables U and F are auxiliary variables utilized in the constraints, and their specific roles will be elaborated on in Section 3.2.2.

3.2.2 Constraints

We first describe the modified dependency constraints and memory constraints. To remove the concept of stages, we integrate the dependency information into the computation of peak memory. This allows us to ensure that an operation is only computed if all its inputs are available, while also ensuring that the peak memory at any point in the computation does not exceed the device memory. More concretely, the general form to compute the memory consumption U_{i+1} after computing node v_{i+1} can be represented:

$$U_{i+1} = U_i + s_{i+1}^T mem_{i+1} + memRecomputed_{i+1} - memFreed_{i+1} \tag{3}$$

$$U_0 = s_0^T mem_0 \tag{4}$$

$memRecomputed_{i+1}$ is the memory cost for recomputing forward pass operators needed as inputs for backward operator v_{i+1}, and $memFreed_{i+1}$ is the amount of memory we can free when computing operator v_{i+1}. $s_{i+1}^T mem_{i+1}$ is the memory consumption for computing operator v_{i+1} using the sharding strategy specified in s_{i+1}.

Next, we explain how to calculate the value of $memRecomputed_{i+1}$. When considering operator v_{i+1}, there are two cases to consider: if it is an operator in the forward pass or in the backward pass. (1) If v_{i+1} is an operator in the forward pass then $memRecomputed_{i+1} = 0$ since we do not do any recomputation during the forward pass. (2) If operator v_{i+1} is in the backward pass, $memRecomputed_{i+1}$ may be non-zero since we may need to recompute operators in the forward pass to use as inputs to operator v_{i+1}. Let's define X_{i+1} as the set of forward pass operators v_j where operator v_{i+1} is the first backward pass operator utilizing v_j. Then:

$$memRecomputed_{i+1} = \sum_{j \in X_{i+1}} (1 - m_j) s_j^T mem_j \tag{5}$$

By considering the set X_{i+1}, we can calculate the value of $memRecomputed_{i+1}$ by summing the memory consumption of the operators in X_{i+1} that we did not checkpoint (i.e. where $1 - m_j = 1$). This captures the additional memory needed due to recomputation for the backward pass operator v_{i+1}.

Next, we discuss the computation of $memFreed_{i+1}$, representing the memory released during the calculation of operator v_{i+1}. Before delving into the details of forward and backward operators, let's introduce a new variable, F_{ij}. This variable indicates whether memory from a forward operator v_i can be released upon the computation of operator v_j. Specifically, if $F_{ij} = 1$, it means the memory utilized by forward operator v_i can be freed after computing operator v_j. With F_{ij}, we can calculate $memFreed_{i+1}$. The detailed calculation of F_{ij} will be discussed in Section 3.2.3. (1) For a forward operator, we have:

$$memFreed_{i+1} = \sum_j F_{ji} s_j^T mem_j + \sum_{j \in L_{i+1}} z_j (1 - m_j) s_j^T mem_j \tag{6}$$

where L_i is the set of forward pass operators that cease to be live in the forward pass for the first time at time i. The first summation frees the memory of operators that are not needed in the backward pass, and the second summation frees the memory of operators that are recomputed in the backward pass but no longer needed in the forward pass. We gave a example of in Figure 3. (2) For a backward operator, we should free the memory of operators in the forward pass for which F_{ji} is true and we should free the memory of operators in the backward pass which are no longer live (Figure 4 gives an example). Let Y_i be the set of backward operators j for which $live_j = i$. Thus we have:

$$memFreed_{i+1} = \sum_j F_{ji} s_j^T mem_j + \sum_{j \in Y_i} s_j^T mem_j \tag{7}$$

Once we have the formulation for memory consumption, we can add the constraint that it can not exceed the device limit as below:

$$U_i \leq deviceMemory \tag{8}$$

Lastly, we also need to introduce additional constraints to model the assumption that all inputs of a recomputed node are never recomputed. Suppose p_j is the number of inputs for operator j, then to ensure that all inputs of a recomputed node are never recomputed, we have the constraint:

$$m_j = 0 \implies \sum_{(v_i, v_j) \in E} m_i \geq p_j \tag{9}$$

3.2.3 Calculation of F (Time to Free Operators)

Since we never recompute backward operators, we can accurately determine when to free the memory associated with the backward operators through standard liveness analysis. Specifically, we can release a backward operator's memory once all operators that depend on it as input have been computed. Thus we only define F_{ij} for v_i being an operator in the forward pass. For a forward operator, we may free it at most twice: if the operator is not recomputed, then we may free it once it is no longer necessary in the backward pass. If the operator is recomputed in the backward pass, then we first free it when it is no longer needed during the forward pass, and free it again when it is no longer needed in the backward pass. Let H_i be the time when forward operator v_i is recomputed in the backward pass and let $live_i$ be the latest operator in the graph that uses v_i as an input. To be

precise, $live_i = \max(j)\ \forall(v_i, v_j) \in E$. If operator v_i is recomputed then the earliest we may free its memory in the backward pass is at h_i where:

$$h_i = \max(H_j(1 - m_j), live_i)\ \forall(v_i, v_j) \in E \tag{10}$$

Then we can use h_i to represent F_{ij}

$$F_{ij} = 1 \iff h_i = j \tag{11}$$

To linearize these constraints, we introduce auxiliary variables, whose details we list in the Appendix.

4 Evaluation

In this section, we explore the impact of joint optimization on the cost and memory usage of DNN training. To this end, we address the following research questions: (1) What is the tradeoff between memory usage and training latency when using sharding and gradient checkpointing under different memory limits? (2) What is the impact of communication speed on the sharding strategy? (3) To what extent can KIWI with reduced complexity approximate the optimal joint optimization policy?

We evaluate the effectiveness of KIWI by comparing it against baselines on Transformer encoders and decoders [33], across different hardware configurations. Our results show that joint optimization using KIWI enables lower computational overhead than the baselines at all memory budgets and across all hardware setups. Additionally, we compare the sharding strategies with and without KIWI, and our findings show that joint optimization can significantly improve performance, resulting in an up to 2.2× reduction in training latency. Finally, we demonstrate that KIWI with reduced complexity achieves near-optimal solutions in a significantly shorter time compared to the original version of KIWI. Unless specifically indicated, all references to KIWI mentioned thereafter pertain to the version that has undergone complexity reduction, as described in Section 3.2.

4.1 Baselines and experimental setup

Baselines: We compare KIWI's performance against two heuristics commonly used for Transformers.

- Checkpoint dots [2]: checkpointing only activations resulting from matrix multiplication operations. This heuristic is effective since re-computing element-wise operators is significantly less computationally expensive than re-computing matrix multiplications that have higher computation intensity.
- Megatron's selective recompute [20]: checkpointing activations other than Transformer's attention layer. This heuristic is specific to Transformer models and takes advantage of the fact that while the attention layer requires a lot of memory, recomputing it incurs much less overhead than recomputing the entire transformer layer.

For both baselines, we apply the heuristics first to rewrite the computation graph. We then use Alpa to find the optimal sharding strategies for all the operators in the rewritten graph. Additionally, we compare KIWI's performance with that of baseline Alpa [37], which only performs sharding without any recomputation.

Implementation: KIWI is implemented in Python and accepts low-level HLO operators [6]. We chose to solve on the HLO level as it provides more flexibility, allowing operators within the same layer to have different sharding and recomputation strategies. However, this decision greatly increases the problem's complexity. For instance, a single encoder layer can translate to 308 low-level operators. To obtain the HLO computation graph, we translate the network definition, defined with JAX, to a lower-level HLO computation. We then construct and solve the optimization problem, Equation 2, using the Gurobi mathematical programming library as a QCQP program.

Experiment setup: We chose to evaluate a 4-layer transformer for the encoders and decoders, as it is roughly equivalent in size to a typical pipeline stage [17]. The integration of KIWI into pipeline parallel schedules is straightforward by running it on each partition's graph subset. To maintain consistency, we adopted the same configuration as the GPT 13B model [9] for each transformer layer, which includes 5120 hidden units and 40 attention heads. We use batch size = 16 for all experiments.

Furthermore, we vary the shape of device meshes to simulate various levels of tensor parallelism and different hardware settings, enabling a comprehensive evaluation of the system's scalability and

efficiency. A device mesh is a 2-D array representing a cluster's logical view. For instance, the device mesh of shape (2, 3) with values [[0, 1, 2], [3, 4, 5]] can represent two nodes where each node has three GPUs. Communication can occur both inter-node (along device mesh dimension 0) and intra-node (along device mesh dimension 1) with varying communication bandwidth Lastly, to obtain the computation and communication cost of each operator, we profile the execution cost of all operators across all potential operator shapes on the GCP n1-standard instance with 8 V100 GPUs, 16 vCPUs, and 60 GB CPU memory [7]. The 8 GPUs in a node are connected via NVLink. For inter-node communication, we profile n1-standard instances located in the same region. The batch latency results are simulated by aggregating the communication and computation costs of all operators in the network.

4.2 Evaluation Results

4.2.1 Performance

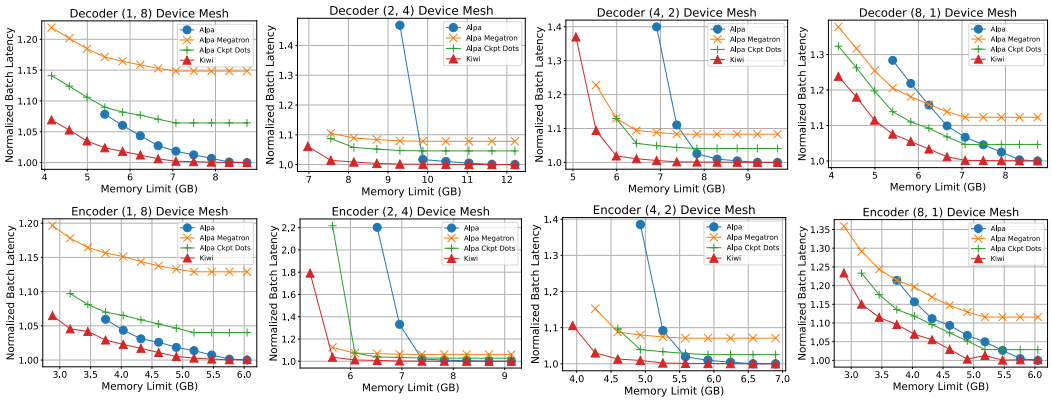


Figure 5: Performance across different device meshes.

Figure 5 compares KIWI and the baselines on transformer encoders and decoders across different device meshes. The y-axis represents the computation and communication overhead in terms of time, relative to the Alpa baseline without any memory limit. The x-axis indicates the total memory budget given to execute each model with the specified batch size, considering single precision training. Each algorithm offers the memory budget parameter that allows adjusting the trade-off between recomputation or communication and memory usage, where smaller memory budgets result in higher overhead.

Takeaway: KIWI consistently outperforms the other configurations in all scenarios—up to 2.2× faster execution compared to Alpa, up to 2.2× faster compared to Alpa combined with the checkpoint dots heuristics, and up to 1.2× faster compared to Alpa combined with the Megatron's selective recompute heuristics. It is worth highlighting that both the Megatron heuristics and the checkpoint dots heuristics are expert-designed manual optimizations specifically tailored for Transformers. Despite this, KIWI manages to surpass their performance under different device mesh settings. Furthermore, KIWI excels in solving under highly constrained memory settings. For example, when considering device meshes of dimensions (2, 4) and (4, 2), KIWI successfully solves the optimization problem with memory limits of 5.1 GB and 3.8 GB respectively for the encoder model, and 6.8 GB and 4.9 GB respectively for the decoder model. In contrast, none of the other algorithms can achieve the same level of optimization under such stringent memory limits. Another important observation is that KIWI never performs worse than Alpa. However, checkpointing heuristics + Alpa may exhibit suboptimal performance in certain configurations. This is because checkpointing heuristics + Alpa disregard the possibility that sharding can be more efficient than recomputation in certain cases, as the heuristics always performs recomputation based on predefined rules. In contrast, KIWI takes into account the specific characteristics of the problem and can dynamically adapt its optimization strategy to achieve better results. We will compare the strategies picked by different algorithms in Section 4.3.

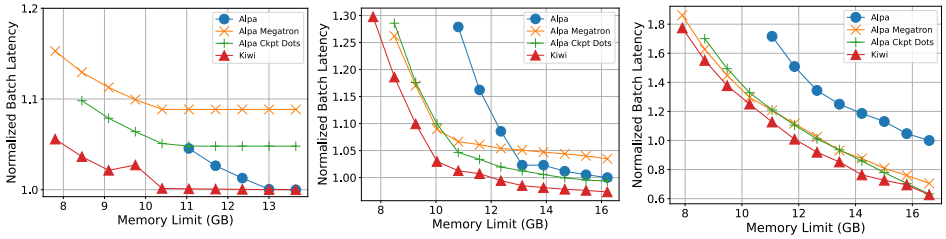


Figure 6: Performance on device mesh (1, 4) when varying communication bandwidth. The first picture represents the same bandwidth as NVLink, the second picture simulates a 10x slowdown in communication, and the third simulates a 100x slowdown in communication.

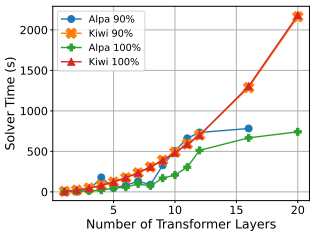


Figure 7: Solver execution time with various transformer layers

Memory Limit Percentage	Before Reducing Complexity Normalized Latency / Solver Time (s)	After Reducing Complexity Normalized Latency / Solver Time (s)
20%	8.35 / 1717.18 $\pm$ 78.4	8.35 / 0.79 $\pm$ 0.02
40%	4.88 / 660.41 $\pm$ 24.5	5.09 / 0.91 $\pm$ 0.03
60%	2.71 / 718.49 $\pm$ 24.1	3.13 / 0.80 $\pm$ 0.04
80%	1.54 / 39.71 $\pm$ 0.85	1.72 / 0.67 $\pm$ 0.02

Table 1: Compare the MLP batch latency and the 95% confidence interval of KIWI solver time before and after reducing complexity. The batch latency is normalized with the Alpa MLP execution time without any memory limit.

4.3 Joint and Automatic Optimization

In this section, we seek to understand why KIWI outperforms the baselines. Specifically, we investigate the behavior of different algorithms when varying the communication bandwidth. Figure 6 depicts the execution time of various algorithms on a 4-layer transformer decoder as we manipulate the communication bandwidth between GPUs. As the communication links become slower, KIWI exhibits a significantly improved execution time relative to Alpa. This observation holds true for the different gradient checkpointing heuristics as well.

Furthermore, in contrast to previous gradient checkpointing heuristics, KIWI demonstrates dynamic adaptability by adjusting the degree of recomputation based on the system settings. As illustrated in the first plot in Figure 6, the heuristics perform worse than Alpa when the communication speeds remain unmodified, whereas KIWI consistently outperforms them regardless of the communication speed. This is attributed to KIWI's ability to increase the amount of recomputation when the communication links are slow, thereby minimizing communication overhead. Conversely, when the communication links are fast, KIWI reduces the reliance on recomputation and instead prioritizes sharding as a strategy to alleviate memory pressure.

4.4 Scability Test

We next investigate the scalability of KIWI with the complexity reduction by measuring its execution time compared to Alpa. We vary the number of transformer layers and record the execution time for both methods. As mentioned, a single encoder layer can be translated into 308 low-level HLO primitives, and we provide the number of operators in Table 2 for reference. We assess the stability of KIWI and Alpa under 90% and 100% memory limits. The results, shown in Figure 7, indicate that despite can KIWI handle a more complex problem than Alpa, their execution times are comparable when the number of transformer layers is below 12. However, as the number of layers exceeds 12, KIWI experiences slower execution due to the increased complexity, scaling with the number of

Layers	Operators	Layers	Operators	Layers	Operators	Layers	Operators	Layers	Operators	Layers	Operators
1	308	4	1192	8	2254	12	3366	16	4478	20	5590

Table 2: Number of operators when varying the number of encoder layers.

operators. Nevertheless, in practical applications, tensor parallelism is employed within each pipeline stage, with each stage typically comprising fewer than 20 layers [17] and we consider this to be a minor concern, as KIWI can be run independently within each pipeline stage.

4.5 Optimality Analysis

We also empirically evaluate the performance of KIWI before and after reducing the problem complexity (Section 3.2), aiming to assess how closely the simplified solution aligns with the optimal solution. Due to the prohibitively long runtime of the optimal solution, we conduct our tests on a simpler model, specifically a 7-layer MLP. This model contains a smaller number of operators (26 operators) compared to the transformer model, making it more manageable for experimentation purposes. As shown in Table 1, KIWI demonstrates comparable performance even after the simplification process. In the worst-case scenario, with an 80% memory budget, KIWI increases the normalized batch latency from 1.54 to 1.72, resulting in an 11.6% increase. However, KIWI significantly reduces the solver time under more stringent memory constraints. In fact, when operating with a 20% memory budget, KIWI achieves over 2000$\times$ reduction in solver time while still finding the optimal solution.

5 Limitations and Future Work

As demonstrated, solving the optimal joint optimization problem in practice is extremely challenging. Therefore, KIWI introduces certain restrictions on the search space to find a feasible solution. While KIWI achieves similar performance compared to the completely optimal problem on smaller models, its performance on larger models, such as transformers, remains uncertain since running the optimal problem on such a large graph is not feasible. For future work, it will be an interesting direction to explore modifications to KIWI that allow for larger search spaces while still being computationally feasible to solve. Currently, KIWI already outperforms expert-designed heuristics. By exploring larger search spaces, we can potentially achieve even better optimization results for large models.

6 Conclusion

This paper highlights the importance of performing joint optimization on GC + TP for training large models. We present a formulation of the joint optimization problem using quadratically constrained quadratic programming. We also introduce ways to greatly reduce computational complexity while maintaining optimization effectiveness. Importantly, unlike quantization and low-rank approximation, KIWI does not change the model's training behavior. Experiments show that KIWI can achieve better performance compared to expert-designed heuristics automatically.

References

[1] A100 manual. `https://www.nvidia.com/en-us/data-center/a100/`.

[2] Jax checkpoint policy. `https://jax.readthedocs.io/en/latest/notebooks/autodiff_remat.html#custom-policies-for-what-s-saveable`.

[3] K80 manual. `https://www.nvidia.com/en-gb/data-center/tesla-k80/`.

[4] T4 manual. `https://www.nvidia.com/en-gb/data-center/tesla-t4/`.

[5] V100 manual. `https://www.nvidia.com/en-us/data-center/v100/`.

[6] Hlo semantics. `https://www.tensorflow.org/xla/operation_semantics`.

[7] N1 instance configuration. `https://cloud.google.com/compute/docs/general-purpose-machines#n1_machines`. Accessed: 2023-05-08.

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

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`.

[10] J. Chen, L. Zheng, Z. Yao, D. Wang, I. Stoica, M. Mahoney, and J. Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, pages 1803–1813. PMLR, 2021.

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

[12] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

[13] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[14] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

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

[16] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[17] 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*, 32, 2019.

[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*, 2:497–511, 2020.

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

[20] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

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

[22] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[23] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[24] X. Liu, L. Zheng, D. Wang, Y. Cen, W. Chen, X. Han, J. Chen, Z. Liu, J. Tang, J. Gonzalez, et al. Gact: Activation compressed training for generic network architectures. In *International Conference on Machine Learning*, pages 14139–14152. PMLR, 2022.

[25] Y. Liu, S. Li, J. Fang, Y. Shao, B. Yao, and Y. You. Colossal-auto: Unified automation of parallelization and activation checkpoint for large-scale models, 2023.

[26] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

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

[28] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

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

[30] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. Zero-offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference*, pages 551–564, 2021.

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

[32] J. M. Tarnawski, D. Narayanan, and A. Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021.

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

[34] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. 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.

[35] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[36] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu, et al. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*, 2021.

[37] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

7 Appendix

7.1 Optimal Joint Optimization

In this section, we provide additional details for the KIWI's optimal joint optimization problem that were omitted in the main text.

7.1.1 Additional Variable Definitions

$B_{t,i}$ is a binary variable that indicates whether we choose to checkpoint operator i from stage $t-1$ to stage t. $U_{t,i}$ represents the total memory used after computing operator i in stage t. $F_{t,i,j}$ is a binary variable that is true if we free the memory of operator i after computing operator j in stage t.

7.1.2 Dependency Constraints

An operation is computed in stage t only if all dependencies are available. Additionally, an operator may only be checkpointed in stage t if it was computed in stage $t-1$ or checkpointed in stage $t-1$. This is equivalent to writing

$$A_{t,j} \leq A_{t,i} + B_{t,i}, \forall t, \forall (v_i, v_j) \in E \tag{12}$$

$$B_{t,i} \leq A_{t-1,i} + B_{t-1,i}, \forall t \geq 2, \forall i \tag{13}$$

Moreover, we force progress by requiring node i to be computed at stage i. At stage i, we require that nodes $i+1$, $i+2$, ..., $|V|$ are not yet computed. This constraint can be expressed as

$$A_{i,i} = 1, \sum_{j>i} A_{i,j} = 0, \sum_{j \geq i} B_{i,j} = 0 \tag{14}$$

In the original checkmate paper, since parameters are always kept in memory, they are not part of the formula. However, in the distributed setting, we also need to shard the parameters and therefore the nodes in all formulas above also include the parameter nodes. To ensure that parameters are always in memory, we add the below constraints:

$$B_{t,i} = 1, \forall i \in Param, t \geq i \tag{15}$$

7.1.3 Memory Constraints

We adopt the same approach as Checkmate [18] for defining memory constraints. We introduce the memory counting variable $U_{t,k}$, which denotes the memory used after computing node v_k in stage t. We want the memory usage after computing each node at every stage to be smaller than the device memory, that is

$$U_{t,k} < deviceMemory, \forall t, \forall k \tag{16}$$

The binary variable $F_{t,i,k}$, represents the deallocation of node v_i in stage t after evaluating node v_k, is used to recursively define $U_{t,k}$ for $(v_i, v_k) \in E$. Additionally, all checkpointed values are resident in memory at the beginning of a stage. Hence, we initialize the recurrence,

$$U_{t,0} = \sum_{i=1}^{|V|} s_i^T mem_i * B_{t,i} \tag{17}$$

Before evaluating v_{k+1}, v_k and dependencies (parents) of v_k may be deallocated if there are no future uses. Then, the memory usage after computing v_{k+1} is

$$U_{t,k+1} = U_{t,k} - memFreed_k(t) + A_{t,k+1} * (s_{k+1}^T mem_{k+1}) \tag{18}$$

where $memFreed_k(t)$ is the amount of memory freed by deallocating v_k and its parents (if there are no future uses) at stage t.

$$memFreed_k(t) = \sum_{(v_i, v_k) \in E} (s_i^T mem_i) * F_{t,i,k} \tag{19}$$

$$F_{t,i,k} = A_{t,k} * (1 - B_{t+1,i}) \prod_{(v_i, v_j) \in E, j>k} (1 - A_{t,j}) \tag{20}$$

$(1 - B_{t+1,i})$ ensures that node v_i is only freed if it is not checkpointed for the next stage. The $\prod_{(v_i,v_j)\in E, j>k}(1 - A_{t,j})$ ensures that $F_{t,i,k} = 0$ if any child of v_i is computed in the current stage, since then v_i needs to be retained for later use. Multiplying by $A_{t,k}$ ensures that values are only freed at most once per stage. The constraints mentioned above involve several non-linear terms, such as the recursive definition of U and the product of multiple items in F. To linearize these non-linear terms, we apply the same method as Checkmate.

7.1.4 Sharding Constraints

For each operator, only a single sharding strategy is picked at each time:

$$\sum s_i = 1, \forall i \tag{21}$$

Furthermore, similar to Alpa [37], e_{ij} represents the resharding decision between node i and j. If $s_i \in \{0,1\}^{k_i}$ and $s_j \in \{0,1\}^{k_j}$, then $e_{ij} \in \{0,1\}^{k_i \times k_j}$. To make e_{ij} equal to $s_j^T s_i$, we introduce the following constraints for each e_{ij}:

$$\sum_{m=1}^{k_i} e_{ijmn} \leq s_{jn}, \forall 1 \leq n \leq k_j \tag{22}$$

$$\sum_{n=1}^{k_j} e_{ijmn} \leq s_{im}, \forall 1 \leq m \leq k_i \tag{23}$$

7.2 Joint Optimization With Reduced Complexity

In this section, we describe how we linearize the constraints in KIWI's reduced complexity optimization problem. Additionally, we provide proofs of correctness for the optimization problem.

7.2.1 Linearization

To linearize Equation 10, we introduce helper variables hz. Let hz_i be a vector of binary decision variables. The length of hz_i is equal to one more than the number of forward operators that use operator i as an input. Suppose there are k such operators. The constraint then becomes

$$\sum_{j=0}^{k} hz_{ij} = 1 \tag{24}$$

$$h_i \leq live_i + |V|(1 - hz_{i0}) \tag{25}$$

$$h_i \geq live_i \tag{26}$$

$$h_i \leq H_j(1 - m_j) + |V|(1 - hz_{ij})\ \forall j \geq 1 \tag{27}$$

$$h_i \geq H_j(1 - m_j)\ \forall j \geq 1 \tag{28}$$

To linearize Equation 11, we introduce binary variables F_- and F_+ to constrain F. If we let δ be some small positive number, then $F_{ij} = 1 \iff h_i = j$ is equivalent to the following constraints:

$$jF_{ij} + (j+\delta)F_{+_{ij}} \leq h_i \leq (j-\delta)F_{-_{ij}} + jF_{ij} + |V|F_{+_{ij}} \tag{29}$$

$$F_{-_{ij}} + F_{ij} + F_{+_{ij}} = 1 \tag{30}$$

To Linearize Equation 9, we introduce a constant big M and simplify the constraint as:

$$Mm_j + \sum_{(v_i,v_j)\in E} m_i \geq p_j \tag{31}$$

7.2.2 Correctness Proofs

In this section, we give proof outlining the correctness of certain parts of the optimization problem. First, we show that we do not free an operator's memory twice in either the forward pass or the backward pass. In the forward pass, as described in Equation 6 we free operator i's memory after computing operator j if either F_{ij} is true or $z_i(1-m_i)$ is true and $i \in L_{j+1}$. Towards a contradiction, suppose we free operator i's memory both after computing operator j and after computing operator k, with both operators j and k being distinct operators in the forward pass. Suppose this is because both $F_{ij} = 1$ and $F_{ik} = 1$. However, $F_{ij} = 1 \iff h_i = j$ and $F_{ik} = 1 \iff h_i = k$. Thus F_{ij} and F_{ik} cannot both be true. The other case in which we free operator i's memory at both timesteps is when $z_i(1-m_i) = 1$ and $i \in L_{j+1} \cap L_{k+1}$. However, by construction of L, $L_{j+1} \cap L_{k+1} = \emptyset$.

The last case in which operator i's memory could be deallocated twice in the forward pass is, without loss of generality, when $F_{ij} = 1$ and $z_i(1 - m_i)$ and $i \in L_{k+1}$. However, if $z_i = 1$, then operator i is used as input to an operator in the backward pass. This implies $live_i$ corresponds to an operator in the backward pass. Since $F_{ij} = 1$ implies $live_i \leq j$, this case is not possible since j is an operator in the forward pass.

It is a similar process to show that we do not free an operator's memory twice in the backward pass. In the backward pass, as described in Equation 7, we free operator i's memory after computing operator j if either $F_{ij} = 1$ or $i \in Y_j$. Suppose i is an operator in the forward pass and j, k are distinct operators in the backward pass so that we free operator i's memory both after computing operators j and k. This can only be achieved if $F_{ij} = 1$ and $F_{ik} = 1$. As we showed above, this is not possible. Now suppose i is an operator in the backward pass. Then this can only be achieved when $i \in Y_j \cap Y_k$. However, similar to the construction of L, $Y_j \cap Y_k = \emptyset$, so this is not possible.

Next we show that we linearize constraints correctly. First we show that Equation 10 is linearized correctly. Equation 26 and Equation 28 imply that $h_i \geq \max(H_j(1 - m_j), live_i) \; \forall (v_i, v_j) \in E$. Equation 24 implies that only one hz_{ij} will equal one and the rest zero. For that hz_{ij} we will have $h_i \leq live_i$ if $j = 0$, or we will have $h_i \leq H_j(1 - m_j)$ otherwise. So it follows that $h_i = \max(H_j(1 - m_j), live_i) \; \forall (v_i, v_j) \in E$

Now we show that Equation 11 is linearized correctly. Suppose $h_i = j$, so Equation 29 becomes

$$jF_{ij} + (j + \delta)F_{+_{ij}} \leq j \leq (j - \delta)F_{-_{ij}} + jF_{ij} + |V|F_{+_{ij}} \tag{32}$$

If $F_{ij} = 0$, then $F_{+_{ij}} = 0$ to satisfy the first inequality. But, then the right inequality becomes $j \leq (j - \delta)F_{-_{ij}}$ which can never be satisfied. So $F_{ij} = 1$ must be true. Now suppose $F_{ij} = 1$. Then, from Equation 30, we know that $F_{+_{ij}} = F_{-_{ij}} = 0$, since $F_{-_{ij}} + F_{ij} + F_{+_{ij}} = 1$. The constraint then becomes $j \leq h_i \leq j$, which implies that $h_i = j$.