Practical Performance Guarantees for Pipelined DNN Inference

Aaron Archer [*1] **Matthew Fahrbach** [*1] **Kuikui Liu** [2] **Prakash Prabhu** [1]

Abstract

We optimize pipeline parallelism for deep neural network (DNN) inference by partitioning model graphs into k stages and minimizing the running time of the bottleneck stage, including communication. We give practical and effective algorithms for this NP-hard problem, but our emphasis is on tackling the practitioner's dilemma of deciding when a solution is good enough. To this end, we design novel mixed-integer programming (MIP) relaxations for proving lower bounds. Applying these methods to a diverse testbed of 369 production models, for $k \in \{2, 4, 8, 16, 32, 64\}$, we empirically show that these lower bounds are strong enough to be useful in practice. Our lower bounds are substantially stronger than standard combinatorial bounds. For example, evaluated via geometric means across a production testbed with $k = 16$ pipeline stages, our MIP formulations raise the lower bound from 0.4598 to 0.9452, expressed as a fraction of the best partition found. In other words, our improved lower bounds close the optimality gap by a factor of 9.855x.

1. Introduction

Large-scale machine learning (ML) workloads rely on distributed systems and specialized hardware accelerators, e.g., graphics processing units (GPUs) and tensor processing units (TPUs). Fully utilizing this hardware, however, remains an increasingly important challenge. ML accelerators have a small amount of *fast memory* co-located with each computational unit (CU), and a much larger amount of *slow memory* that is accessed via an interconnect shared among the CUs. Achieving peak performance for deep neural network (DNN) training and inference requires the ML compiler and/or practitioner to pay significant attention to where intermediate data is stored and how it flows between CUs. This work addresses pipeline partitioning for DNNs to maximize *inference throughput*, with a particular focus on lower bound methods for proving per-instance approximation ratios.

[*]Equal contribution [1]Google [2]MIT. Correspondence to: Matthew Fahrbach <fahrbach@google.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

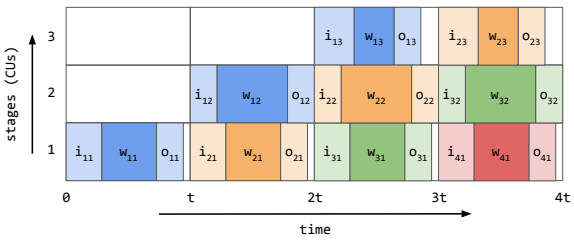

Figure 1. Inference pipeline from startup to steady state with $k = 3$ stages. Each inference batch is represented with the same color as it advances through the pipeline. Values $\mathtt{i}_{b\ell}$, $\mathtt{w}_{b\ell}$, $\mathtt{o}_{b\ell}$ are the times needed for stage ℓ to get its input for batch b, process it, and flush its output. Stage 2 is the bottleneck, i.e., $t = \mathtt{i}_{*2} + \mathtt{w}_{*2} + \mathtt{o}_{*2}$, and limits system throughput. Empty space (white) denotes idle time.

ML inference handles two main types of data: *model parameters* (weights learned during training) and *activations* (intermediate outputs of the model, e.g., from hidden layers). Keeping activations in fast memory (e.g., SRAM) is critical, so ML compilers often treat it as a hard constraint. Model parameters can be streamed from slow memory, but caching them in fast memory greatly boosts performance. When we partition an end-to-end inference computation into a linear pipeline with k stages and process each stage on a different CU, we *increase the amount of fast memory* at our disposal by a factor of k, allowing us to cache more parameters and support larger activations (and hence larger models and/or batch sizes). However, this introduces two main challenges: (1) CUs must send their outputs downstream, often via a slow and contended data channel, so we need to minimize communication overhead; and (2) we must balance the running time across all stages of the partition since the *overall throughput is governed by the bottleneck stage*.

1.1. Practioner's dilemma

Suppose you are an ML engineer who has been tasked with partitioning a model graph for pipelined inference as illustrated in Figure 1. We explain the practitioner's dilemma with a toy example. Assume you are searching for the best way to partition a model among 8 processors, and you discover a solution where the bottleneck processor takes 10ms to finish. In this case, the pipeline finishes one inference every 10ms, so the throughput is 1 inference / 10ms = 100 inferences per second, and the latency of a single inference

is 8 x 10ms = 80ms. Is this a good solution? How do you know there isn't one that is ten times better?

Suppose now that you devise an approximation algorithm and prove it has a worst-case *approximation ratio* of 2. Congratulations, proving a worst-case bound is often no easy feat! You run your algorithm on the same instance as before, generating a partition with a bottleneck stage of 12ms, and a *lower bound* of 6ms. Is this a good result? What if your boss tells you that your company is about to spend millions of dollars on hardware to run your inference pipeline. If you can improve your 12ms solution all the way down to 6ms, then you can save half of this hardware or run twice as many more inferences. This is great motivation to improve your solution, but how do you know when to stop? *It could be that the lower bound is weak, not your solution.*

Two things went wrong here. Your algorithm and your lower bound are robust to all inputs, but you have a particular instance in front of you, and that is all you care about. If you could run a different heuristic to output a solution with a bottleneck stage of 10ms, along with a lower bound certificate for this instance of 9.5ms, that means it is impossible to improve the solution by more than 5%. Presumably, this will make both you and your boss happier, and suggest that you can now spend your time on something else.

In practice, we care about good performance on a whole family of instances. Partitioning algorithms run inside ML compilers, often with tight time constraints that preclude computing strong lower bounds in the compiler itself. In this case, one way to gain confidence in the quality of partitioning algorithms is to create a testbed of instances that are representative of the ones we solve in practice, run our algorithms to generate solutions and lower bounds offline, and examine the approximation ratios we were able to prove for these instances. If the average per-instance approximation ratio is 1.05 on the testbed, we argue this should give us more confidence in the partitioning algorithm than would the proof that a different algorithm has worst-case ratio 2. If our lower bounds are fast enough to run within the compiler's time limits, that is even better, as we can then bound the suboptimality of each instance, rather than trusting that the testbed results generalize to the instance at hand.

This ethos motivates the focus of our paper: we describe a sequence of successively stronger (but costlier to compute) lower bound methods that can be used to prove per-instance lower bounds for the pipeline partitioning problem. We also give algorithms for constructing partitions for the original problem, and show that the cost of these partitions is close to the lower bounds across a testbed of hundreds of production models with a variety of architectures.

1.2. Our contributions and techniques

The main contributions of this work are as follows:

1. We formalize the *max-throughput partitioning problem* (MTPP) for pipelined inference, and we prove that it is NP-hard. We then formulate a novel mixed-integer program (MIP) for MTPP, and study sparse relaxations to obtain strong lower bounds (Section 3).
2. We give a fast and practical pipeline partitioning algorithm called `SliceGraph` that combines dynamic programming with a biased random-key genetic algorithm (Section 4).
3. We present extensive experiments across real and synthetic model graphs for a wide variety of ML architectures and workloads (Section 5). Using our (*a posteriori*) MIP lower bounds, we demonstrate that `SliceGraph` is highly effective in practice, e.g., for $k \leq 16$, our strongest lower bound is (on average) 95.5% of the `SliceGraph` solution, whereas the standard combinatorial lower bound is only 46.0%.

2. Preliminaries

To further build intuition, it is helpful to think of pipelined inference as an assembly line where the model is split into k stages and the inputs for each stage are produced earlier in the assembly line. If t is the running time of the longest stage (i.e., the *bottleneck*), each stage can finish its local computation in parallel in time t. Every t units of time we advance each batch one stage forward in the assembly line (see Figure 1), so the end-to-end latency for a batch of inferences is kt and the throughput of the system is $1/t$ (i.e., one batch per t units of time). To maximize system throughput, it is critical to the partition work in a way that minimizes the bottleneck time t.

2.1. Computation graphs

An ML model is commonly represented as a computation graph $G = (V, E)$, where V is the set of node operations (called *ops*) and $E \subseteq V \times V$ are the data flow edges. Let $n = |V|$ and $m = |E|$. For simplicity, assume each op u outputs one tensor consumed by (possibly many) downstream ops v, which we represent by the edges (u, v). This corresponds to a `tensorflow.Graph` (Abadi et al., 2016), and has analogs in MLIR (Lattner et al., 2021), MXNet (Chen et al., 2015), and PyTorch (Paszke et al., 2019).

We introduce a few node weights to help model the cost of inference:

- $\texttt{work}(v)$ is the running time of $v \in V$. This is typically estimated with an analytic or learned cost model (Kaufman et al., 2021).
- $\texttt{size}_{\texttt{param}}(v)$ is the memory footprint of the model parameters that $v \in V$ uses. For example, if v is a matrix multiplication op, $\texttt{size}_{\texttt{param}}(v)$ is the storage cost for the entries of the matrix.

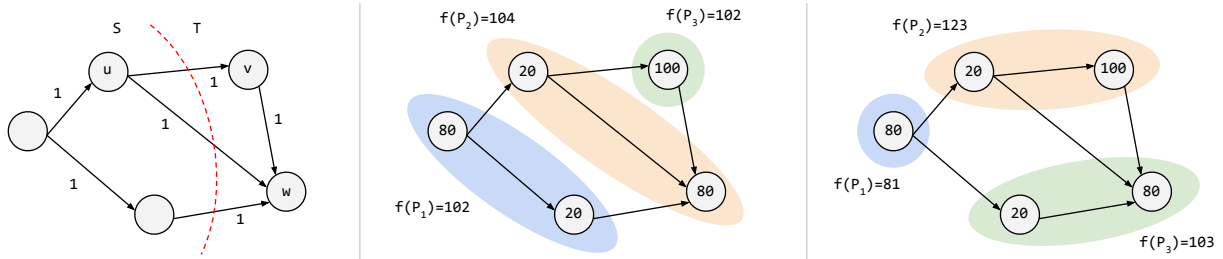

Figure 2. Partitioning computation graphs: (left) tensor cut property where $\texttt{io}(S,T) = 2$ because v and w consume the same tensor; (middle) invalid partition because blocks P_2 and P_3 form a cycle in the quotient graph; (right) valid partition with block costs for $k = 3$.

- $\texttt{size}_{\texttt{out}}(v)$ is the size of the output of v (e.g., in bytes).

We use standard graph theory notation to denote dependencies between nodes:

- $N^-(v) = \{u \in V : (u,v) \in E\}$ is the set of nodes whose output is consumed by v.
- $N^+(v) = \{w \in V : (v,w) \in E\}$ is the set of nodes that consume the output of v.
- $N^-(S) = \bigcup_{v \in S} N^-(v) \setminus S$ and $N^+(S) = \bigcup_{v \in S} N^+(v) \setminus S$ extend the neighborhood notation to sets of nodes.

The reason for excluding S from the neighborhoods will become clear when we consider the inter-block communication costs for a partition of G.

2.2. Problem statement

Acyclic quotient graph constraint Let $\mathcal{P}_k(G)$ be the set of partitions of V into k blocks (possibly empty) such that the induced *quotient graph* of G is acyclic. Formally, let $P = \{P_1, P_2, \ldots, P_k\}$ be a partition of V, i.e., $P_1 \cup P_2 \cup \cdots \cup P_k = V$ and $P_i \cap P_j = \varnothing$ for all $i \neq j$. Since P_i can be empty, we can think of partitioning V into at most k blocks. For each $v \in V$, let $[v]_P$ denote the block in P containing v. The quotient graph $Q = (P, E')$ for partition P has blocks of P as its nodes and reduced edge set $E' = \{([u]_P, [v]_P) : (u,v) \in E \text{ and } [u]_P \neq [v]_P\}$. We require Q to be acyclic so that there is valid data flow when G is partitioned across different processors.

Inter-block communication Let B be the bandwidth of the interconnect between blocks. For disjoint sets $S, T \subseteq V$, the IO cost (in units of time) from S to T is

$$\texttt{io}(S,T) = \frac{1}{B} \sum_{v \in N^-(T) \cap S} \texttt{size}_{\texttt{out}}(v). \tag{1}$$

We overload singleton notation: $\texttt{io}(u,v) = \texttt{io}(\{u\},\{v\})$.

By summing over the set of producer ops $v \in N^-(T) \cap S$, each tensor going from S to T is counted once, *even if it has many consumers in* T. This is different from the cost of a traditional edge-cut set since it considers only one edge in each tensor edge equivalence class (see Figure 2). Refining how the cost of communication is modeled is where computation graph partitioning begins to deviate from more familiar cut-based graph partitioning problems.

Streaming model parameters Each block is a computational unit with a fixed amount of fast memory, e.g., SRAM for GPUs and multi-chip packages (Mei & Chu, 2016; Gao et al., 2020; Dasari et al., 2021) and high-bandwidth memory for TPUs (Jouppi et al., 2017; 2023). To achieve peak performance, it is essential that all model parameters assigned to a block be fully cached. Otherwise, some of these parameters must be streamed to the block during each inference batch from slow memory, e.g., shared DRAM. Inter-block bandwidth is typically at least an order of magnitude slower than intra-block bandwidth (Dao et al., 2022), so we ignore the communication between ops within a block and refer to the time needed to stream parameters that spill over as the *overflow cost* of a block.

There are two key factors for deciding if all model parameters can be fully cached on a block: (1) the size of its fast memory, and (2) the peak activation memory. We start by describing the peak memory scheduling problem (Marchal et al., 2019; Paliwal et al., 2020; Ahn et al., 2020; Lin et al., 2021; Vee et al., 2021; Zhang et al., 2022; Fradet et al., 2023; Jin et al., 2023). For a set of ops P_i, the peak memory scheduling problem is to find a linear execution order of $v \in P_i$ minimizing the amount of working memory needed for all intermediate computations. Once we know how much fast memory to reserve for the activations, we allocate the rest for caching parameters.

The overflow cost for a set of ops $S \subseteq V$ on a block with M fast memory, peak memory $\texttt{peak}(S)$, and inter-block bandwidth B is

$$\texttt{overflow}(S) = \frac{(\texttt{size}_{\texttt{param}}(S) + \texttt{peak}(S) - M)^+}{B}, \tag{2}$$

where we use the notation $x^+ = \max(x, 0)$.

Total block cost The total cost of a block with ops $S \subseteq V$ (i.e., its running time) in a pipeline partition is

$$f(S) = \overbrace{\texttt{io}(V \setminus S, S)}^{\text{input tensors}} + \sum_{v \in S} \texttt{work}(v) + \texttt{overflow}(S) + \overbrace{\texttt{io}(S, V \setminus S)}^{\text{output tensors}}. \quad (3)$$

Putting everything together, we arrive at the following min-max objective function.

Definition 2.1. For a computation graph G and number of blocks k, the *max-throughput partitioning problem* (MTPP) is

$$P^* = \arg\min_{P \in \mathcal{P}_k(G)} \left\{ \max_{i \in [k]} f(P_i) \right\}, \quad (4)$$

where $[k] = \{1, \ldots, k\}$. Let $\text{OPT} = \max_{i \in [k]} f(P_i^*)$ denote the minimum bottleneck cost.

Remark 2.2. There are *many more* moving parts to ML performance than partitioning model graphs—it is just one of many ML compiler passes (e.g., op fusion, tensor sharding, fine-grained subgraph partitioning, peak memory scheduling). However, it is one of the highest-order components with a significant impact on overall system efficiency.

3. Mixed-integer programming lower bounds

We first prove that MTPP is NP-hard and cannot have a fully polynomial-time approximation scheme (FPTAS), unless P = NP, by giving a reduction from the minimum makespan scheduling problem on k identical parallel processors, which is strongly NP-hard (Hochbaum & Shmoys, 1987). We defer all proofs in this section to Appendix A.

Theorem 3.1. *For $k = 2$,* MTPP *is* NP*-hard. Furthermore, there does not exist a fully polynomial-time approximation scheme for* MTPP*, unless* P = NP.

In light of this hardness, we formulate a novel mixed-integer program to solve MTPP and focus on deriving strong lower bounds. Even for medium-sized models and moderate values of k, the exact program pushes the limits of MIP solvers, so we relax the formulation to give strong lower bounds while using fewer variables, constraints, and non-zeros. The exact MIP in (5) and its relaxations are the main theoretical contribution of our work, allowing us to provide strong per-instance approximation guarantees.

To simplify the presentation, we ignore the `overflow` terms in (3) since $\texttt{peak}(S)$ depends on how the ops in a block are scheduled (Paliwal et al., 2020). This is equivalent to reserving a buffer for activations in each block and treating the remaining amount of fast memory as the new budget.

3.1. Exact MIP formulation

We now present the MIP for solving MTPP in Figure 3. The main idea is to number the blocks from 1 to k in DAG order (i.e., a topological order of the induced quotient graph) and use binary decision variables to assign nodes to blocks.

Decision variables There are $O(nk)$ binary variables:

- x_{vb} indicates whether node $v \in V$ is assigned to block $b \in [k]$.
- y_{vb} indicates whether node $v \in V$ is assigned to some block at or before b. For any feasible assignment, this means $y_{vk} = 1$, for all $v \in V$, and $y_{v(b-1)} \leq y_{vb}$, for all $v \in V, b \in [k]$. We let $y_{v0} = 0$ for notational convenience.
- c_{ub} indicates whether any edge $(u, v) \in E$, corresponding to the tensor that u produces, flows *into or out of* block b.

All decision variables are nominally binary in (12), but we can relax the x and c variables to $[0, \infty)$ since they naturally lie in $\{0, 1\}$ whenever the y variables do.

The x and y variables represent the same information in two ways, and hence are redundant. Using both, however, allows us to express some constraints more naturally. In our code, we use (11) to eliminate each occurrence of x_{vb}. Doing so offers two advantages relative to eliminating the y variables. First, the tensor cut constraints require $O(mk^2)$ non-zeros if expressed purely in terms of the x variables. Second, and more crucial, branching on the y variables works *in tandem* with the acyclicity constraints to create more asymmetry in the branch-and-bound process, allowing the MIPs to solve faster compared to branching on the x variables.

Objective value The auxiliary `bottleneck` variable allows us to minimize the max block cost via constraint (6). The $\texttt{block}_b$ variables defined in (7) capture the total node cost assigned to block b plus the induced cut costs, counting each tensor edge in the cut-set exactly once.

DAG constraints Given the "completed-by-block b" variables y_{vb}, we use constraint (8) to force the quotient graph to be acyclic. If $(u, v) \in E$ and v is assigned to block b or earlier, the DAG constraints guarantee that u is also assigned to block b or earlier, so (8) holds, and conversely.

Tensor edge-cut constraints Each tensor τ can be represented by multiple edges in G, each with the same source node $u(\tau)$. If *any* of these edges is cut by block b, we must set $c_{ub} = 1$. Constraint (9) captures the case where an edge (u, v) flows into block b from the left—namely that when v is assigned to block b (i.e., $x_{vb} = 1$) and u is assigned to an earlier block (i.e., $y_{u(b-1)} = 1$), then c_{ub} is forced to be 1,

$$\text{minimize } \texttt{bottleneck} \tag{5}$$

$$\text{such that } \texttt{bottleneck} \geq \texttt{block}_b \quad \forall b \in [k] \tag{6}$$

$$\texttt{block}_b = \sum_{v \in V} \texttt{work}(v) \cdot x_{vb} + \frac{1}{B} \sum_{u \in V} \texttt{size}_{\texttt{out}}(u) \cdot c_{ub} \quad \forall b \in [k] \tag{7}$$

$$y_{ub} \geq y_{vb} \quad \forall (u,v) \in E, b \in [k] \quad \texttt{// DAG constraints} \tag{8}$$

$$c_{ub} \geq y_{u(b-1)} + x_{vb} - 1 \quad \forall (u,v) \in E, b \in [k] \quad \texttt{// cut input tensors} \tag{9}$$

$$c_{ub} \geq x_{ub} - y_{vb} \quad \forall (u,v) \in E, b \in [k] \quad \texttt{// cut output tensors} \tag{10}$$

$$c_{ub} \geq 0 \quad \forall (u,v) \in E, b \in [k]$$

$$y_{v(b-1)} \leq y_{vb} \quad \forall v \in V, b \in [k]$$

$$y_{v0} = 0 \quad \forall v \in V \quad \texttt{// convenience variable}$$

$$y_{vk} = 1 \quad \forall v \in V \quad \texttt{// boundary condition}$$

$$x_{vb} = y_{vb} - y_{vb-1} \quad \forall v \in V, b \in [k] \tag{11}$$

$$y_{vb}, x_{vb}, c_{vb} \in \{0,1\} \quad \forall v \in V, b \in [k] \tag{12}$$

Figure 3. Exact MIP for solving MTPP, where variables $x_{vb} \in \{0,1\}$ indicate whether node $v \in V$ is assigned to block $b \in [k]$.

and otherwise it is not. For outgoing edges, if u is assigned to block b (i.e., $x_{ub} = 1$) and v is assigned to a later block (i.e., $y_{vb} = 0$), then constraint (10) forces $c_{ub} = 1$.

Theorem 3.2. *The mixed-integer program in Eq. (5) solves the max-throughput partitioning problem using $O(nk)$ variables, $O(mk)$ constraints, and $O(mk)$ non-zeros.*

3.2. Relaxing to a three-superblock formulation

If the exact MIP is too difficult to solve, then we can use a relaxed "three-superblock" formulation whose size does not depend on k to compute a lower bound for OPT. The idea is to imagine the bottleneck block, consolidate all earlier blocks into one superblock, and all later blocks into a third superblock. Then, we use a combinatorial lower bound L to ensure that the middle block is sufficiently expensive.

Lemma 3.3 (Simple lower bound)**.** *For any computation graph $G = (V, E)$, cost function $\texttt{work} : V \to \mathbb{R}_{\geq 0}$, and partition of V into $k \geq 1$ blocks, there exists a block with at least L units of* work*, where*

$$L = \max\left(\max_{v \in V} \texttt{work}(v), \frac{1}{k} \sum_{v \in V} \texttt{work}(v)\right) \leq \text{OPT}. \tag{13}$$

This formulation is the same as the exact MIP in Figure 3, for $k = 3$, except for two small changes:

1. Add a constraint that forces the node cost of block 2 to be at least the simple lower bound in Lemma 3.3:

$$\sum_{v \in V} \texttt{work}(v) \cdot x_{v2} \geq L. \tag{14}$$

2. Remove $\texttt{block}_1$, $\texttt{block}_3$, and all constraints involving them from the MIP. This simplifies the objective to

$$\text{minimize } \texttt{block}_2,$$

as the middle block aims to model the bottleneck cost.

Observe that the true bottleneck block can hide inside of superblocks 1 or 3, and hence would not contribute to the objective. Therefore, this relaxation can give strictly weaker lower bounds than the exact MIP.

Corollary 3.4. *For any computation graph G and number of blocks $k \geq 1$, the three-superblock MIP uses $O(n)$ variables, $O(m)$ constraints, and $O(m)$ non-zeros, and gives a lower bound for the* MTPP *objective.*

3.3. "Guess the bottleneck block" formulation

The three-superblock MIP is agnostic about which block in the original instance (represented by block 2 in the relaxation) is the one with $\texttt{work} \geq L$. Building on this, another approach is to guess that the bottleneck is block $j \in [k]$, and get a stronger lower bound LB_j under this assumption. Since the guess could be wrong, we must compute LB_j for all $j \in [k]$ and take $\min_{j \in [k]} \text{LB}_j$ as the valid lower bound for OPT.

The formulation for LB_j is the same as the exact MIP with $k = 3$ in Figure 3, except we add constraint (14) and make one other change. For $b \in \{1, 3\}$, the right-hand side of (7) defining $\texttt{block}_b$ is the combined node cost and cut cost for superblock b, excluding the tensors that are cut by blocks in the same superblock and counting tensors that enter superblock 3 only once, even if their edges terminate in dif-

ferent blocks within superblock 3. The worst block in the superblock is at least as expensive as the average block, so we can replace the constraints in (6) with the following lower bounds:

$$\texttt{bottleneck} \geq \frac{1}{j-1} \cdot \texttt{block}_1$$
$$\texttt{bottleneck} \geq \frac{1}{k-j} \cdot \texttt{block}_3.$$

If $j = 1$, this forces $x_{v1} = 0$ for all $v \in V$; and if $j = k$, this forces $x_{v3} = 0$ for all $v \in V$. Said differently, nodes cannot be assigned before block 1 or after block k. If $k = 3$, then there is no reason to prefer one relaxation over the other (i.e., Section 3.2 and Section 3.3), but for $k \gg 3$, the two relaxations use substantially fewer variables and constraints than the exact formulation in Section 3.1.

4. Algorithm

We now present our approach to pipeline partitioning. This algorithm is simple by design and runs inside ML compilers with tight latency requirements (e.g., XLA for TensorFlow). In Section 5, we prove that it is near-optimal across a production testbed by computing per-instance approximation guarantees using our new MIP formulations.

4.1. Reducing to a search over topological orderings

We first reduce MTPP to a search over topological orderings as follows:

1. An optimal partition P^* in Eq. (4) has a corresponding topological order π^*.
2. There exist node weights $\mathbf{x}^* \in [0,1]^n$ such that Kahn's topological sorting algorithm with tiebreaking on $\mathbf{x}^*$ recovers π^* (see Appendix B.1).
3. For any topological order $\pi \in \mathfrak{S}_V$, we can efficiently compute an optimal segmentation of π via dynamic programming. By slicing a topological order this way, we easily *satisfy the acyclicity constraint*.

One method for searching over topological orders in Item 2 is to sample random node weights. Another is to learn the weights using a genetic algorithm or reinforcement learning. To implement the dynamic program in Item 3 efficiently, we use a fast data structure for segment cost queries.

Lemma 4.1. *There is a* `SegmentCostDataStructure` *that takes computation graph $G = (V, E)$ and topological order $\pi \in \mathfrak{S}_V$ as input, and supports the following operations:*

- `Initialize`(G, π)*: Preprocesses the graph in $O(n^2 + m \log^2(n))$ time.*
- `Query`(ℓ, r)*: Returns $f(\{v_{\pi(\ell)}, \ldots, v_{\pi(r)}\})$ in Eq. (3) in constant time, after initialization.*

Algorithm 1 Optimal MTPP slicing of topological order π into at most k blocks.

```
function SliceGraph(G, k, π)
// Partitions the full topological order π
    Initialize segment_cost data structure for (G, π)
    return DP(segment_cost, n, k)
function DP(segment_cost, r, k')
// Recursively partitions the first r nodes optimally into k' blocks
    if k' = 1 then
        return segment_cost.Query(1, r)
    ans ← ∞
    for ℓ = 1 to r do
        a ← DP(segment_cost, ℓ, k' − 1)
        b ← segment_cost.Query(ℓ + 1, r)
        ans ← min(ans, max(a, b))
    return ans
```

All proofs for this section are deferred to Appendix B, but at a high level, `SegmentCostDataStructure` computes the cost of each $[\ell, r]$ slice of the topological order π (counting each tensor in a cut once) using a sliding window algorithm and two-dimensional Fenwick tree (Mishra, 2013).

Lemma 4.2. `SliceGraph` *runs in time $O(n^2 k + m \log^2 n)$ and finds an optimal segmentation of topological order π into at most k blocks for* MTPP.

4.2. Searching over topological orders

Any topological order $\pi \in \mathfrak{S}_V$ can be realized using Kahn's algorithm with the right node priorities $\mathbf{x} \in [0,1]^n$. Thus, we now focus on methods for finding good vectors of node weights. Note that we only recast MTPP into a search over topological orders to bypass the acyclic quotient graph constraint—it is still *provably hard* to find an optimal topological order. Our next result formalizes this observation.

Theorem 4.3. *There exist node priorities $\mathbf{x}^* \in [0,1]^n$ such that Kahn's algorithm with tie-breaking outputs a topological order π^* for which* `SliceGraph`$(G, k, \pi^*) = \text{OPT}$.

Since it is NP-hard to find an optimal topological order π^*, we explore heuristics that are fast and work well in practice.

Random weights. Each weight $x_i \sim U(0,1)$ is drawn i.i.d. from the uniform distribution. Note that this is not the same as sampling topological orders uniformly at random, which has its own rich history (Matthews, 1991; Bubley & Dyer, 1999; Huber, 2006; García-Segador & Miranda, 2019).

Biased random-key genetic algorithm (BRKGA). BRKGA is a problem-agnostic metaheuristic that evolves real-valued vectors $\mathbf{x} \in [0,1]^N$ (called *chromosomes*) using a *decoder* function that links BRKGA's evolutionary rules to the problem at hand (Gonçalves & Resende, 2011). We use BRKGA

to optimize the node priorities $\mathbf{x}$ by evaluating the quality of its induced optimal segmentation $P(\mathbf{x})$. In the language of genetic algorithms, the *fitness* of $\mathbf{x}$ is the MTPP objective $\max_{i \in [k]} f_{P(\mathbf{x})}(i)$. We present this decoder in Algorithm 2.

Algorithm 2 BRKGA decoder using Kahn's algorithm and `SliceGraph` to partition G.

function `BrkgaSortAndSliceDecoder`(G, k, chromosome $\mathbf{x} \in [0,1]^n$ of node priorities)
  $\pi \leftarrow$ `KahnWithNodePriorities`$(G, \mathbf{x})$
  return `SliceGraph`(G, k, π)

5. Experiments

We now present an empirical study of our MIP lower bound formulations. To motivate this, we revisit the practitioner's dilemma (Section 1.1) with a case study for a specific model in our production dataset called `net10_new_ckpt-4000009` for $k = 4$ blocks. Suppose you want to optimize this model to run efficiently in production on millions of dollars worth of hardware. You use `BrkgaSortAndSliceDecoder` to partition the model, and the best solution you find has a bottleneck time of 3454.79. Is this good or bad? How hard should you work to improve it?

Since the units are arbitrary, we rescale the solution to have value 1. Applying Lemma 3.3, you prove the `simple` combinatorial lower bound of 0.2775 for this instance (in the rescaled units). This is not enough to satisfy your boss since you could hypothetically reduce your costs by 72%. Next you turn to the `bottleneck` lower bound in Section 3.2, which computes to 0.6570 (i.e., 2.37x stronger than `simple`). This makes your boss less cranky, but there is still an uncomfortable gap, so you compute `bottleneck-guess` in Section 3.3, yielding another 1.39x increase in your lower bound to 0.9152. This is enough to satisfy your boss, but for your own curiosity you run Gurobi on the `exact` MIP, and arrive at a lower bound of 0.9913. As it turns out, your original solution was nearly optimal the whole time, and you just didn't know it! However, this increasing sequence of certificates—obtained with increasing computational effort—steadily increased your confidence.

Our experiments repeat this exercise for hundreds of computation graphs across our production testbed, for all $k \in \{2, 4, 8, 16, 32, 64\}$. We rescale each lower bound to represent it as a fraction of the best solution value found, and then we summarize the quality of the lower bounds in Table 1 by taking their geometric mean across all graphs.

Datasets Our production testbed is a superset of the models in the experiments of Xie et al. (2022). This includes 369 computation graphs from many application domains. (e.g., BERT, ResNet, MobileNet, vision models, LSTMs, speech encoders, and WaveRNN). Most of these graphs are publicly available, but we cannot publish the node and edge weights as they come from an internal proprietary cost model.

We also run the same experiment on 1000 publicly available synthetic model graphs from REGAL (Paliwal et al., 2020). See Appendix C.2 for the results and more details.

Setup We solve the MIPs using a combination of Gurobi v9.0.2 (Gurobi Optimization, LLC, 2023) and SCIP v7.0.1 (Bestuzheva et al., 2021). Each instance is run on a heterogeneous cluster containing, e.g., Intel Xeon Platinum 8173M @ 2.00GHz processors, and the best lower bound proven in a fixed time limit is reported.

Partitioning algorithms We compare several topological sort heuristics for `SliceGraph`: random weights, BRKGA, and minimum linear arrangement. Overall, BRKGA worked best, so we use its solutions to normalize the lower bounds in Table 1. We provide more details in Appendix C.1.

Results We discuss several interesting properties about the results in Table 1. First, `bottleneck-guess` and `exact` produce the same lower bounds for $k = 2$. This is not a coincidence since the MIPs are equivalent for $k = 2$. For larger values of k, `bottleneck-guess` MIPs can "cheat" in two ways: (1) by ignoring communication costs within the superblocks, and (2) effectively counting the total work of each superblock as if it is smeared uniformly across its blocks. As the value of k increases, so does the opportunity to cheat since a larger fraction of the communication cost is ignored and the smearing effect is more pronounced. Hence, `bottleneck-guess` loses ground relative to `exact`. The advantage of `bottleneck-guess` over `bottleneck` is that it considers edges crossing the superblock boundary. However, larger values of k dilute this advantage since the communication costs get amortized over a larger number of blocks im-

Table 1. Geometric means of the best available lower bound from the MIP hierarchy, normalized by the best solution found using BKRGA, across the production dataset.

Lower bound	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$
`simple` (Lemma 3.3)	0.8340	0.6627	0.5236	0.4598	0.4435	0.4401
`bottleneck`	0.9597	0.7911	0.6481	0.5770	0.5590	0.5543
`bottleneck-guess`	0.9901	0.8446	0.6601	0.5780	0.5593	0.5543
`exact`	0.9901	0.9737	0.9588	0.9452	0.8749	0.7874

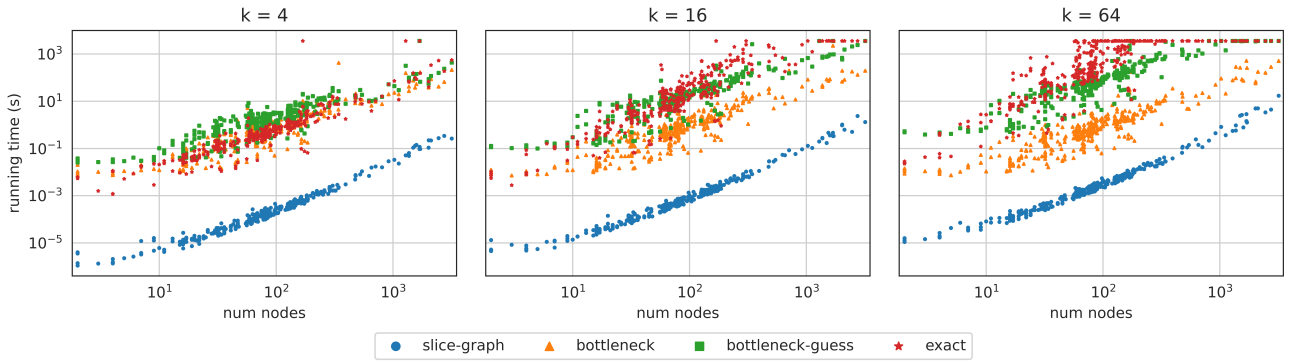

Figure 4. Running times of `SliceGraph` and different MIP lower bound computations across the production models. Each point denotes a run for one graph, color-coded to denote `SliceGraph` partitioning vs. `bottleneck`, `bottleneck-guess`, and `exact` lower bounds. The `bottleneck-guess` times are summed across all k MIP instances involved. Each plot is for a different value of k. In order to facilitate visual comparisons across the plots, all three employ the same y-axis. Some of the data tops out at 3600 seconds since that is where we set the MIP time limit.

plicitly contained in each superblock. By $k = 64$, the lower bounds produced by `bottleneck-guess` and `bottleneck` are the same.

Interestingly, the additive gaps between `bottleneck` and `simple` in Table 1 hold for each k, hovering near 0.12. This makes sense because the advantage of `bottleneck` is that it considers communication costs while `simple` does not, and its treatment of these costs does not depend on k.

The running times show that `bottleneck-guess` and `exact` are about four orders of magnitude slower than `SliceGraph` in Figure 4. Therefore, this MIP-based analysis is most valuable for offline analysis, rather than running as part of the compiler. For $k \leq 16$, `bottleneck-guess` running times are roughly on par with `exact`. We note that `bottleneck-guess` in Figure 4 depicts the sum of the running times for all k sub-MIP solves when guessing the k bottlenecks to compute the lower bound. Since these solves are independent, they could be run in parallel to achieve a faster wall-clock time. Further, MIP solve times tend to be strongly superlinear in the problem size (observe the log-log scale), so we would expect `bottleneck-guess` to show an advantage even with total running time for larger k. Indeed, this behavior clearly emerges by $k = 64$.

6. Related work

Model parallelism Two of the seminal works on pipeline parallelism for machine learning are GPipe (Huang et al., 2019) and PipeDream (Narayanan et al., 2019; 2021). These works focus on scaling up *DNN training* and have spawned a long list of related work: `torchgpipe` (Kim et al., 2020), FPDeep (Wang et al., 2020), Pipemare (Yang et al., 2021), TeraPipe (Li et al., 2021), BaPipe (Zhao et al., 2022), SAPipe (Chen et al., 2022), BPipe (Kim et al., 2023), synchronous pipeline planning (Luo et al., 2022), scheduling in heterogenous settings (Park et al., 2020; Yuan et al., 2022), breadth-first pipeline parallelism (Lamy-Poirier, 2023), and SWARM parallelism (Ryabinin et al., 2023).

Of these works, PipeDream considers the most similar mathematical model (Narayanan et al., 2019, Section 3.1). There is some overlap with MTPP (e.g., a min-max running time objective), but there are also major differences:

1. PipeDream assumes a single topological order and that the induced quotient graph is a *path graph*. This means IO can only flow between adjacent blocks instead of to shared memory for downstream consumers.
2. PipeDream uses communication-work concurrency, so the running time of a processor executing ops $S \subseteq V$ is $\max(\texttt{work}(S), \texttt{io}(V \setminus S, S) + \texttt{io}(S, V \setminus S))$.
3. PipeDream supports data parallelism and is designed to use replicated workers for training. The model weight updates for replicated workers use *weight stashing* and a syncing technique that is not necessary for inference.

Another technique for going beyond data parallelism is *tensor sharding*, e.g., Mesh TensorFlow (Shazeer et al., 2018), Megatron-LM (Shoeybi et al., 2019), GShard (Lepikhin et al., 2021), and GSPMD (Xu et al., 2021).

Acyclic graph partitioning The origins of acyclic graph partitioning are in multiprocessor scheduling (Garey & Johnson, 1979). There have since been several key applications for pipeline parallelism (Cong et al., 1994; Gordon et al., 2006; Sanchez et al., 2011). The computational hardness and inapproximability of balanced acyclic partitioning has recently been revisited in Moreira et al. (2017); Papp et al. (2023). Some practical methods for acyclic graph partitioning use graph coarsening (Moreira et al., 2020; Herrmann et al., 2019; Popp et al., 2021) and MIP with branch-and-bound solvers (Nossack & Pesch, 2014; Albareda-Sambola et al., 2019; Özkaya & Çatalyürek, 2022).

Conclusion

This work formalizes MTPP for pipelined DNN inference, proposes novel mixed-integer programs for computing lower bounds for this partitioning objective, and presents fast and effective partitioning algorithms for maximizing inference throughput. Our lower bounds allow us to prove strong a posteriori approximation guarantees, which can be invaluable in practice since countless software engineering hours are spent partitioning mission-critical ML models across accelerators to improve system efficiency. Without good lower bounds, it is often unclear if practitioners should continue searching for better partitions, or if they are near optimality and just don't know it. Our MIP formulations allow us to compute certificates that act as a stopping condition on further investment of software engineering time.

Impact statement

We present work that advances the field of ML efficiency. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

Acknowledgements

We thank Dong Hyuk Woo for encouraging us to research pipeline partitioning algorithms. Part of this work was done while Kuikui Liu was an intern at Google Research.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016.

Ahn, B. H., Lee, J., Lin, J. M., Cheng, H.-P., Hou, J., and Esmaeilzadeh, H. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *Proceedings of Machine Learning and Systems*, 2:44–57, 2020.

Albareda-Sambola, M., Marín, A., and Rodríguez-Chía, A. M. Reformulated acyclic partitioning for rail-rail containers transshipment. *European Journal of Operational Research*, 277(1):153–165, 2019.

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., van der Hulst, R., Koch, T., Lübbecke, M., Maher, S. J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M. E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D., and Witzig, J. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

Bubley, R. and Dyer, M. Faster random generation of linear extensions. *Discrete Mathematics*, 201(1):81–88, 1999.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, Y., Xie, C., Ma, M., Gu, J., Peng, Y., Lin, H., Wu, C., and Zhu, Y. SAPipe: Staleness-aware pipeline for data parallel DNN training. In *Advances in Neural Information Processing Systems*, 2022.

Cong, J., Li, Z., and Bagrodia, R. Acyclic multi-way partitioning of boolean networks. In *Proceedings of the 31st Annual Design Automation Conference*, pp. 670–675, 1994.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Dasari, U. K., Temam, O., Narayanaswami, R., and Woo, D. H. Apparatus and mechanism for processing neural network tasks using a single chip package with multiple identical dies, March 2 2021. US Patent 10,936,942.

Fradet, P., Girault, A., and Honorat, A. Sequential scheduling of dataflow graphs for memory peak minimization. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 76–86, 2023.

Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating GPU memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1342–1352, 2020.

García-Segador, P. and Miranda, P. Bottom-up: A new algorithm to generate random linear extensions of a poset. *Order*, 36(3):437–462, 2019.

Garey, M. R. and Johnson, D. S. *Computers and Intractability*. W. H. Freeman, 1979.

Gonçalves, J. F. and Resende, M. G. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, 2011.

Gordon, M. I., Thies, W., and Amarasinghe, S. Exploiting coarse-grained task, data, and pipeline parallelism

in stream programs. *ACM SIGPLAN Notices*, 41(11): 151–162, 2006.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

Herrmann, J., Ozkaya, M. Y., Uçar, B., Kaya, K., and Çatalyürek, Ü. V. V. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 41(4):A2117–A2145, 2019.

Hochbaum, D. S. and Shmoys, D. B. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM*, 34(1):144–162, 1987.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems*, 32, 2019.

Huber, M. Fast perfect sampling from linear extensions. *Discrete Mathematics*, 306(4):420–428, 2006.

Jin, C., Purohit, M., Svitkina, Z., Vee, E., and Wang, J. R. New tools for peak memory scheduling. *arXiv preprint arXiv:2312.13526*, 2023.

Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., et al. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–14, 2023.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

Kahn, A. B. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

Kaufman, S., Phothilimthana, P., Zhou, Y., Mendis, C., Roy, S., Sabne, A., and Burrows, M. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems*, 3:387–400, 2021.

Kim, C., Lee, H., Jeong, M., Baek, W., Yoon, B., Kim, I., Lim, S., and Kim, S. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910*, 2020.

Kim, T., Kim, H., Yu, G.-I., and Chun, B.-G. BPipe: Memory-balanced pipeline parallelism for training large language models. *International Conference on Machine Learning*, pp. 16639–16653, 2023.

Lamy-Poirier, J. Breadth-first pipeline parallelism. *Proceedings of Machine Learning and Systems*, 5, 2023.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. GShard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations*, 2021.

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

Lin, J., Chen, W.-M., Cai, H., Gan, C., and Han, S. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352*, 2021.

Luo, Z., Yi, X., Long, G., Fan, S., Wu, C., Yang, J., and Lin, W. Efficient pipeline planning for expedited distributed DNN training. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pp. 340–349. IEEE, 2022.

Marchal, L., Simon, B., and Vivien, F. Limiting the memory footprint when dynamically scheduling DAGs on shared-memory platforms. *Journal of Parallel and Distributed Computing*, 128:30–42, 2019.

Matthews, P. Generating a random linear extension of a partial order. *The Annals of Probability*, 19(3):1367–1392, 1991.

Mei, X. and Chu, X. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

Mishra, P. A new algorithm for updating and querying sub-arrays of multidimensional arrays. *arXiv preprint arXiv:1311.6093*, 2013.

Moreira, O., Popp, M., and Schulz, C. Graph partitioning with acyclicity constraints. In *16th International Symposium on Experimental Algorithms (SEA)*, volume 75, pp. 30:1–30:15, 2017.

Moreira, O., Popp, M., and Schulz, C. Evolutionary multilevel acyclic graph partitioning. *Journal of Heuristics*, 26 (5):771–799, 2020.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel DNN training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021.

Nossack, J. and Pesch, E. A branch-and-bound algorithm for the acyclic partitioning problem. *Computers & Operations Research*, 41:174–184, 2014.

Özkaya, M. Y. and Çatalyürek, Ü. V. A simple and elegant mathematical formulation for the acyclic DAG partitioning problem. *arXiv preprint arXiv:2207.13638*, 2022.

Paliwal, A., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. Reinforced genetic algorithm learning for optimizing computation graphs. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Papp, P. A., Anegg, G., and Yzelman, A.-J. N. Partitioning hypergraphs is hard: Models, inapproximability, and applications. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 415–425, 2023.

Park, J. H., Yun, G., Chang, M. Y., Nguyen, N. T., Lee, S., Choi, J., Noh, S. H., and Choi, Y.-r. HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference*, pp. 307–321, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

Popp, M., Schlag, S., Schulz, C., and Seemaier, D. Multilevel acyclic hypergraph partitioning. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pp. 1–15. SIAM, 2021.

Ryabinin, M., Dettmers, T., Diskin, M., and Borzunov, A. SWARM parallelism: Training large models can be surprisingly communication-efficient. In *International Conference on Machine Learning*, pp. 29416–29440. PMLR, 2023.

Sanchez, D., Lo, D., Yoo, R. M., Sugerman, J., and Kozyrakis, C. Dynamic fine-grain scheduling of pipeline parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 22–32. IEEE, 2011.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-TensorFlow: Deep learning for supercomputers. *Advances in Neural Information Processing Systems*, 31, 2018.

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

Vee, E. N., Purohit, M. D., Wang, J. R., Ravikumar, S., and Svitkina, Z. Scheduling operations on a computation graph, March 30 2021. US Patent 10,963,301.

Wang, T., Geng, T., Li, A., Jin, X., and Herbordt, M. FPDeep: Scalable acceleration of cnn training on deeply-pipelined FPGA clusters. *IEEE Transactions on Computers*, 69(8):1143–1158, 2020.

Xie, X., Prabhu, P., Beaugnon, U., Phothilimthana, P., Roy, S., Mirhoseini, A., Brevdo, E., Laudon, J., and Zhou, Y. A transferable approach for partitioning machine learning models on multi-chip-modules. *Proceedings of Machine Learning and Systems*, 4:370–381, 2022.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. GSPMD: General and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C., and De Sa, C. Pipemare: Asynchronous pipeline parallel DNN training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

Yuan, B., He, Y., Davis, J., Zhang, T., Dao, T., Chen, B., Liang, P. S., Re, C., and Zhang, C. Decentralized training of foundation models in heterogeneous environments. *Advances in Neural Information Processing Systems*, pp. 25464–25477, 2022.

Zhang, K., Wang, H., Hu, H., Zou, S., Qiu, J., Li, T., and Wang, Z. TENSILE: A tensor granularity dynamic GPU memory scheduling method toward multiple dynamic workloads system. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

Zhao, L., Xu, R., Wang, T., Tian, T., Wang, X., Wu, W., Ieong, C.-I., and Jin, X. BaPipe: Balanced pipeline parallelism for DNN training. *Parallel Processing Letters*, 32(3&4):2250005:1–2250005:17, 2022.

A. Missing analysis for Section 3

Theorem 3.1. *For $k = 2$,* MTPP *is* NP*-hard. Furthermore, there does not exist a fully polynomial-time approximation scheme for* MTPP*, unless* P = NP.

Proof. The minimum makespan scheduling problem on k identical parallel processors is as follows. We are given processing times for n jobs $(p_1, p_2, \ldots, p_n)$ and asked to find an assignment of jobs to processors so that the completion time (i.e., *makespan*) is minimized. We reduce to MTPP by constructing a graph G with n vertices and no edges, setting $\texttt{work}(v_i) = p_i$, and computing a max-throughput partition of G to solve the original makespan instance.

For $k = 2$, this is the NP-hard *partition problem*. More generally, minimum makespan scheduling is strongly NP-hard, so there cannot exist an FPTAS, unless P = NP (Hochbaum & Shmoys, 1987). □

Theorem 3.2. *The mixed-integer program in Eq. (5) solves the max-throughput partitioning problem using $O(nk)$ variables, $O(mk)$ constraints, and $O(mk)$ non-zeros.*

Proof. The x, y, and c variables are each indexed over all nodes and blocks, so there are $O(nk)$ variables total. Constraints Eqs. (8) to (10) are each indexed over all edges and blocks, so there are $O(mk)$ of those. Each constraint except for (7) has a constant number of non-zeros, so they contribute $O(mk)$ non-zeros. Each of the k constraints of type (7) has n each of the x and c variables, so nk non-zeros overall. Thus, there are $O(mk)$ non-zeros in total.

To prove this MIP correctly models MTPP, we must prove that (a) every solution to the problem corresponds to a solution of the MIP (with the same objective value), and (b) every solution to the MIP can be transformed into a solution with the same or better cost that corresponds to a solution of the problem (with the same objective value).

To prove (a), start with any MTPP solution. Each node v is assigned to exactly one block b, so set $x_{vb} = 1$ and $x_{vb'} = 0$, for all $b' \neq b$. Set y to match, i.e., $y_{v0} = \cdots = y_{v(b-1)} = 0$ and $y_{vb} = \cdots = y_{vk} = 1$. For each edge (u, v) and block b, set $c_{ub} = 1$ if the edge crosses into or out of the block, and 0 otherwise. Finally, set $\texttt{block}_b$ to satisfy (7) and set `bottleneck` to be the maximum of the block costs.

We must now check that all of the constraints are satisfied. Constraints (6) and (7) are satisfied by construction. Since the original solution satisfies the DAG constraints, each edge goes from some block to the same or a later block, so constraint (8) is satisfied. Constraint (9) cannot be violated unless $y_{u(b-1)} = x_{ub} = 1$, because otherwise the RHS is zero or negative. But in this case, edge (u, v) originates before block b and terminates inside block b, so the edge is cut as an input tensor, so we set $c_{ub} = 1$, satisfying constraint (9). Similarly, the only way constraint (10) can be violated is if $x_{ub} = 1$ and $y_{vb} = 0$. In this case, edge (u, v) originates in block b and terminates after block b, so the edge is cut as an output tensor, so we have set $c_{ub} = 1$, satisfying constraint (10). Constraint (11) and the y monotone ordering constraints are satisfied by construction. Finally, `bottleneck` really does capture the objective value since it equals the cost of the most expensive block, and the block costs are defined in (7). Therefore, every solution to MTPP corresponds to a solution of the MIP with the same cost.

Now we prove property (b). First, note that constraints (9) and (10) each place one lower bound on c_{ub} for each edge $(u, v) \in E$. Since the only other place c_{ub} appears is in the objective function (implicitly via constraints (7) and (6)), setting c_{ub} to the maximum of those lower bounds can only improve the objective function without harming feasibility. Similarly, `bottleneck` should be set to the maximum of the lower bounds in (6). For a fixed $v \in V$, the y_{vb} variables start at 0 when $b = 0$ and end at 1 when $b = k$, and by (11) we have $x_{vb} = 1$ for the value of b when y_{vb} first jumps up to 1. Thus, the set $\{v \in V : x_{vb} = 1\}$ defines the b-th block of the partition, and these blocks disjointly cover all nodes $v \in V$. Moreover, by a similar argument as above, if any of the edges $(u, v) \in E$ forces $c_{ub} = 1$ via constraints (9) or (10) then edge (u, v) really is cut by block b in this partition, and otherwise none of the edges out of u is cut by block b. Thus, (7) captures the cost of each block in this solution, and `bottleneck` captures the cost of the bottleneck block. □

Corollary 3.4. *For any computation graph G and number of blocks $k \geq 1$, the three-superblock MIP uses $O(n)$ variables, $O(m)$ constraints, and $O(m)$ non-zeros, and gives a lower bound for the* MTPP *objective.*

Proof. The three-superblock MIP is essentially the same as the formulation in Figure 3 for $k = 3$, which means k gets absorbed in the big-O notation and the sizes become $O(n)$ variables, $O(m)$ constraints, and $O(m)$ non-zeros.

To prove this MIP gives a valid lower bound, we start with any solution P^* to MTPP and generate a MIP solution whose value is the same or lower. Lemma 3.3 shows that some block must be assigned at least L units of work, so find one such

block b in P^*. For each node v, set $x_{v2} = 1$ if v is in block b, $x_{v1} = 1$ if v is in an earlier block, and $x_{v3} = 1$ if v is in a later block. Set all other x variables to 0, the y variables as implied by constraints (11), and the c variables to the minimum value that satisfies constraints (9) and (10). We have satisfied the work constraint by construction, and by the same reasoning as in the proof of Theorem 3.2, the value of the MIP solution we constructed equals the cost of block b, which is at most the cost of the bottleneck block in the partition. Since this construction works for all solutions to MTPP, we have proven that the optimal solution of the three-superblock MIP is a lower bound for the true optimal solution. □

B. Missing algorithms and analysis for Section 4

B.1. Kahn's algorithm with node priorities

Kahn's algorithm is a topological sort algorithm that repeatedly peels off the leaves of a DAG (Kahn, 1962). It is particularly useful because it can output different orderings—if there are multiple leaves, different tie-breaking rules produce different topological orders. We give pseudocode for `KahnWithNodePriorities` in Algorithm 3, which takes a vector $\mathbf{x} \in [0,1]^n$ of node priorities as input, and runs in time $O(n \log n + m)$ if implemented with a heap-based priority queue for the active set of leaves.

Algorithm 3 Kahn's topological sorting algorithm with tie-breaking by node priorities.

```
function KahnWithNodePriorities(graph G = (V, E), node priorities x ∈ [0,1]^n)
    Initialize π ← 0_n, indegree ← 0_n, and i ← 1
    for each v ∈ V do
        indegree[v] ← |N⁻(v)|
    Initialize priority queue q        // Max heap implementation
    for each leaf node v ∈ V do
        Insert priority-node pair (x_v, v) into q
    while q is not empty do
        u ← top(q); pop(q)
        π[i] ← u
        for each v ∈ N⁺(u) do
            indegree[v] ← indegree[v] − 1
            if indegree[v] = 0 then
                Insert priority-node pair (x_v, v) into q
        i ← i + 1
    return π
```

B.2. Segment cost data structure

We start with a simpler version of the segment cost data structure that makes entrywise updates to the `io_struct[][]` array during initialization. We give a proof of its correctness, and then we explain how to speed up this preprocessing step with a two-dimensional Fenwick tree to achieve faster $O(\log^2(n))$ subrectangle updates (Mishra, 2013).

Warmup B.1. *There is a* `SegmentCostDataStructure` *that takes a computation graph $G = (V, E)$ and topological order $\pi \in \mathfrak{S}_V$ as input, and supports the following operations:*

- `Initialize`(G, π)*: Preprocesses the graph in $O(n^3)$ time.*
- `Query`(ℓ, r)*: Returns $f(\{v_{\pi(\ell)}, \ldots, v_{\pi(r)}\})$ in Eq. (3) in constant time, after initialization.*

Proof. The correctness is clear by inspection since we are memoizing the contributions of `work`, `io`, and $\texttt{size}_{\texttt{param}}$ to the overall cost of each segment. Calls to `Query`(ℓ, r) take constant time since we are only performing $O(1)$ array look-ups and arithmetic operations. It remains to show that the memoization data structures can be built in $O(mn^2)$ time.

The prefix-sum data structures `work_struct` and `mem_struct` can all be built in $O(n)$ time since we take one pass over each vertex of the computation graph, and updating each entry requires $O(1)$ time (assuming $O(1)$-time queries to `work` and $\texttt{size}_{\texttt{param}}$). Furthermore, `io_struct` can be constructed in $O(n^3)$ time. To see this, observe that we take a single pass over all vertices $u \in V$, and for each vertex, we perform at most $O(n^2)$ arithmetic operations and calls to $\texttt{size}_{\texttt{out}}$ since each update is for a distinct $[\ell, r]$ interval. □

Algorithm 4 Segment cost data structure for blocks of the form $P = \{v_{\pi(\ell)}, v_{\pi(\ell+1)}, \ldots, v_{\pi(r)}\}$.

function `Initialize`(G, π)
  `work_struct` $\leftarrow$ `InitWorkStruct`(G, π)
  `mem_struct` $\leftarrow$ `InitMemStruct`(G, π)
  `io_struct` $\leftarrow$ `InitIOStruct`(G, π)
function `Query`(ℓ, r)
  Query `work` $\leftarrow$ `work_struct`$[r]$ $-$ `work_struct`$[\ell - 1]$
  Query $\texttt{size}_{\texttt{param}}$ $\leftarrow$ `mem_struct`$[r]$ $-$ `mem_struct`$[\ell - 1]$
  Set `overflow` $\leftarrow$ $\texttt{size}_{\texttt{param}} + \texttt{peak}([v_{\pi(\ell)}, v_{\pi(\ell+1)}, \ldots, v_{\pi(r)}]) - M$
  Update `overflow` $\leftarrow \frac{1}{B} \max\{\texttt{overflow}, 0\}$
  Query `io` $\leftarrow$ `io_struct`$[\ell][r]$
  return `work` $+$ `overflow` $+$ `io`

In Algorithm 5, we use the fact that $\pi \in \mathfrak{S}_V$ is a permutation of the vertices and that $\pi^{-1} : V \to [n]$ tells us the index at which a given node appears in the topological order.

Now we demonstrate how the preprocessing time can be reduced using subrectangle range updates to subtract $\texttt{size}_{\texttt{out}}(u)$ from disjoint regions of the two-dimensional `io_struct[][]` array.

Algorithm 5 Segment cost data structure helper functions.

function `InitWorkStruct`(G, π)
`// Builds 1D array of work prefix sums`
  Initialize `work_struct` $\leftarrow \mathbf{0}_n$
  `work_struct`$[1]$ $\leftarrow$ `work`$(\pi(1))$
  for $i = 2$ to n **do**
    `work_struct`$[i]$ $\leftarrow$ `work_struct`$[i-1]$ $+$ `work`$(\pi(i))$
  return `work_struct`
function `InitMemStruct`(G, π)
`// Builds 1D array of` $\texttt{size}_{\texttt{param}}$ `prefix sums`
  Initialize `mem_struct` $\leftarrow \mathbf{0}_n$
  `mem_struct`$[1]$ $\leftarrow$ $\texttt{size}_{\texttt{param}}(\pi(1))$
  for $i = 2$ to n **do**
    `mem_struct`$[i]$ $\leftarrow$ `mem_struct`$[i-1]$ $+$ $\texttt{size}_{\texttt{param}}(\pi(i))$
  return `mem_struct`
function `InitIOStruct`(G, π)
`// Builds 2D array of io segment costs`
  Compute `total` $\leftarrow \sum_{v \in V} \texttt{size}_{\texttt{out}}(v)$
  Initialize $n \times n$ array `io_struct` with `total`
  for $u \in V$ **do** `// Remove` $\texttt{size}_{\texttt{out}}(u)$ `from eligible segments`
    Let $S = (v_1, v_2, \ldots, v_d)$ be the nodes in $\{u\} \cup N^+(u)$ sorted s.t. $\pi^{-1}(v_i) < \pi^{-1}(v_{i+1})$
    for $1 \le \ell \le r < \pi^{-1}(v_1)$ **do**
      `io_struct`$[\ell][r]$ $\leftarrow$ `io_struct`$[\ell][r]$ $-$ $\texttt{size}_{\texttt{out}}(u)$
    for $i = 1$ to $d - 1$ **do**
      for $\pi^{-1}(v_i) < \ell \le r < \pi^{-1}(v_{i+1})$ **do**
        `io_struct`$[\ell][r]$ $\leftarrow$ `io_struct`$[\ell][r]$ $-$ $\texttt{size}_{\texttt{out}}(u)$
    for $\pi^{-1}(v_d) < \ell \le r \le n$ **do**
      `io_struct`$[\ell][r]$ $\leftarrow$ `io_struct`$[\ell][r]$ $-$ $\texttt{size}_{\texttt{out}}(u)$
    for $1 \le \ell \le \pi^{-1}(v_1)$ **do** `// Segments fully containing` $\{v_1, v_d\}$
      for $\pi^{-1}(v_d) \le r \le n$ **do**
        `io_struct`$[\ell][r]$ $\leftarrow$ `io_struct`$[\ell][r]$ $-$ $\texttt{size}_{\texttt{out}}(u)$
  return `io_struct`

Lemma 4.1. *There is a* `SegmentCostDataStructure` *that takes computation graph $G = (V, E)$ and topological order $\pi \in \mathfrak{S}_V$ as input, and supports the following operations:*

- `Initialize`(G, π)*: Preprocesses the graph in $O(n^2 + m \log^2(n))$ time.*
- `Query`(ℓ, r)*: Returns $f(\{v_{\pi(\ell)}, \ldots, v_{\pi(r)}\})$ in Eq. (3) in constant time, after initialization.*

Proof. We use a two-dimensional Fenwick tree (Mishra, 2013) to implement the `io_struct` array. This data structure needs $O(n^2)$ time and space to initialize as $\mathbf{0}_{n\times n}$. It also supports the operation `Update`(p, q, x) where $p = (i_1, j_1)$ and $q = (i_2, j_2)$ define two corners of a rectangular sub-array, and adds x to all entries `io_struct`$[i][j]$ for all $(i, j) \in [i_1, i_2] \times [j_1, j_2]$ in time $O(\log^2(n))$. Therefore, we can first initialize all entries to `total` in $O(\log^2(n))$ time.

For each $u \in V$, we describe how to update `io_struct` in $O(\deg^+(u) \log^2(n))$ time. First, observe that there are $\deg^+(u) + 2$ updates in Lines 17–24 of the form $i \le \ell \le r \le j$. It follows that we can call `Update`$((\ell, \ell), (r, r), -\texttt{size}_{\texttt{out}}(u))$ to correctly update the Fenwick tree. Note that this updates entries that will never be queried (i.e., when $\ell > r$), but this is not a problem. Finally, we call `Update`$((1, \pi^{-1}(u)), (\pi^{-1}(v_d), n), -\texttt{size}_{\texttt{out}}(u))$ to update segments that fully contain $\{u, v_d\}$. Putting everything together, the total running time to maintain `io_struct` as a two-dimensional Fenwick tree is

$$O(n^2) + \sum_{u \in V} O\big(\deg^+(u) \log^2(n)\big) = O\big(n^2 + m \log^2(n)\big).$$

Since each element-wise query takes $O(\log^2(n))$ time in isolation, we use the fact that we can iterate over all $O(n^2)$ entries of the Fenwick tree and write the values of `io_struct`$[\ell][r]$ in a separate two-dimensional array in amortized $O(n^2)$ time. This allows us to achieve $O(1)$ time queries for the segment cost data structure after initialization. Correctness follows from Mishra (2013) and Warmup B.1. □

B.3. Analysis of the `SliceGraph` algorithm

Lemma 4.2. `SliceGraph` *runs in time $O(n^2 k + m \log^2 n)$ and finds an optimal segmentation of topological order π into at most k blocks for* MTPP.

Proof. For any topological order $\pi \in \mathfrak{S}_V$, initialize `segment_cost` for $G(\pi)$ in $O(n^2 + m \log^2(n))$ time and $O(n^2)$ space by Lemma 4.1. Then, run the dynamic programming algorithm `DP`$(\texttt{segment_cost}, n, k)$ on the full topological order π, which can be done in $O(n^2 k)$ time and $O(n^2)$ space since there are $O(nk)$ states and each state can be computed recursively in $O(n)$ time after preprocessing all $[\ell, r]$ segment costs. This gives an optimal stars-and-bars partition of π into k (possibly empty) blocks for the MTPP objective in Eq. (4), which proves the result. □

Theorem 4.3. *There exist node priorities $\mathbf{x}^* \in [0, 1]^n$ such that Kahn's algorithm with tie-breaking outputs a topological order π^* for which* `SliceGraph`$(G, k, \pi^*) = \mathrm{OPT}$.

Proof. Let P^* be an optimal MTPP partition of the nodes, and let $Q = (P^*, E')$ be the induced acyclic quotient graph on the blocks. Let σ be a topological ordering of the blocks P^* in Q. Then, set $x_v^* \leftarrow k - \sigma^{-1}[[v]_{P^*}]$ for every $v \in V$, where $[v]_P$ denotes the block index $i \in [k]$ of the partition $P = \{P_1, P_2, \ldots, P_k\}$ and σ^{-1} is the inverse permutation of $\sigma \in \mathfrak{S}_{P^*}$. This means nodes appearing in the first block of P^* according to σ have highest priority. Running Kahn's algorithm with $\mathbf{x}^*$ recovers a topological order $\pi^* \in \mathfrak{S}_V$ such that when optimally segmented into k blocks, has an objective value that is equal to partition P^*. □

Even with optimal topological order slicing, there exist worst-case instances (k, G, π), for any $k \ge 2$, that can be as bad as the trivial MTPP algorithm that puts all nodes into the same block.

Lemma B.2. *For any $k \ge 2$, there is a computation graph G and topological order π such that when optimally sliced,* `SliceGraph`$(G, k, \pi) = k \cdot \mathrm{OPT}$.

Proof. Consider a graph with $n = 2k$ nodes of two types: nodes $\{1, 2, \ldots, k\}$ are *heavy* with weight $\texttt{work}(v_i) = 1 - \varepsilon$, and nodes $\{k+1, k+2, \ldots, 2k\}$ are *light* with weight $\texttt{work}(v_i) = \varepsilon$. Add one directed edge $(1, k+1)$ with weight $10k$. Clearly

OPT $= 1$ since G can be partitioned as $P^* = \{\{1, k+1\}, \{2, k+2\}, \ldots, \{k, 2k\}\}$, which means $f(P_i^*) = (1-\varepsilon)+\varepsilon = 1$ for all $i \in [k]$ since each block has no incoming or outgoing edges.

Now consider the topological order $\pi = (1, 2, \ldots, k, 2k, 2k-1, \ldots, k+1)$. Any stars-and-bars partition P with an internal separator must cut the edge $(1, k+1)$, which means $\max_{i\in[k]}\{f(P_i)\} \geq 10k$. Therefore, the optimal slicing of π groups all nodes into the same block and has objective value $k(1-\varepsilon) + k\varepsilon = k$. □

Remark B.3. Going a step further, consider graph G above without edge $(1, k+1)$. Any topological sort-based algorithm for k blocks can be as bad as $\frac{2k}{k+1} \cdot$ OPT since any stars-and-bars partition must have an MTPP objective value of at least $\max\{2(1-\varepsilon), (1-\varepsilon) + k\varepsilon\}$. To see this, observe that either two heavy nodes must be in the same block or there is a divider between each pair of adjacent heavy nodes. Setting $\varepsilon = \frac{1}{k+1}$ proves the claim.

C. Missing details and experiments for Section 5

C.1. Partitioning algorithms

We now describe the partitioning algorithms used in our experiments to compute graph partitions for the MTPP problem:

- **Random** This random topological sort algorithm samples T i.i.d. weight vectors $\mathbf{x}^{(t)} \in [0,1]^n$ where $x_i^{(t)} \sim U(0,1)$, maps them to topological orders $\pi^{(t)}$ as described in Section 4.2, and returns $\min_{t\in[T]} \texttt{SliceGraph}(G, \pi^{(t)}, k)$.
- **BRKGA** We run BRKGA with `BrkgaSortAndSliceDecoder` (Algorithm 2), and we set the population to size 100 and the number of generations to 100, for 10^4 total candidate evaluations, which we denote as `brkga-10000` in Table 2. `brkga-100` sets the population size to 10 and the number of generations to 10, for 100 total candidate evaluations.
- **MLA** A *minimum linear arrangement* (MLA) of an undirected graph $G = (V, E, w)$ is a node permutation $\pi \in \mathfrak{S}_V$ minimizing the objective

$$h(\pi) = \sum_{\{u,v\}\in E} w(u,v) \cdot \left|\pi^{-1}(u) - \pi^{-1}(v)\right|.$$

  To generalize this idea to computation graphs, we restrict the search by setting $h(\pi) = \infty$ if π is not a topological order, and setting $w(u,v) = 1$ for the (unweighted) `mla` objective or $w(u,v) = \texttt{io}(u,v)$ for the `mla-weighted` objective. MLA is also an NP-hard problem, so we use BRKGA to optimize this objective (similar to Algorithm 2), but now the fitness is $h(\pi(\mathbf{x}))$. We set the population to size 100 and the number of generations to 100, for a total of 10^4 candidate evaluations. We observed that there is a clear benefit to using `mla-weighted` over `mla` for MTPP, which agrees with intuition. Even though MLA orderings optimize for a different objective, they are still competitive, especially for being a zero-shot heuristic.

We compare the quality of these linear ordering heuristics for `SliceGraph` on the production data in Table 2.

C.2. REGAL

Dataset The synthetic computation graphs from Paliwal et al. (2020, Appendix A.1.4) are constructed as follows. The base graphs are sampled from a set of classic random graph model (see Table 2 therein for the parameters of each random

Table 2. Geometric means of approximation ratio upper bounds $\texttt{SliceGraph}(G, k, \pi)/L(G, k)$ for the production models. Compares the MTPP objective value induced by different linear ordering algorithms relative to the simple lower bound in Lemma 3.3 (lower is better). The `random-`T heuristic generates T i.i.d. node-weight vectors $\mathbf{x}^{(t)}$ and returns the best solution found.

Algorithm	$k=2$	$k=4$	$k=8$	$k=16$	$k=32$	$k=64$
`mla`	1.216	1.538	1.950	2.224	2.304	2.326
`mla-weighted`	1.206	1.516	1.920	2.182	2.258	2.283
`random-1`	1.220	1.551	1.959	2.230	2.309	2.333
`random-100`	1.201	1.512	1.916	2.179	2.258	2.282
`random-10000`	**1.199**	**1.509**	1.911	2.176	2.256	2.281
`brkga-100`	1.201	1.516	1.919	2.184	2.263	2.288
`brkga-10000`	**1.199**	**1.509**	**1.910**	**2.175**	**2.255**	**2.272**

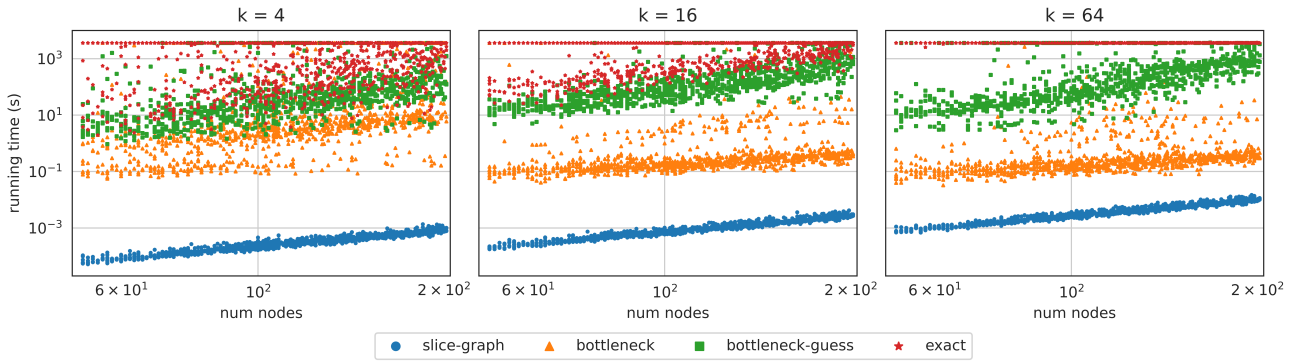

Figure 5. Running times of `SliceGraph` and different MIP lower bound computations across the REGAL models. Each point denotes a run for one graph, color-coded to denote `SliceGraph` partitioning vs. `bottleneck`, `bottleneck-guess`, and `exact` lower bounds. The `bottleneck-guess` times are summed across all k MIP instances involved. Each plot is for a different value of k. In order to facilitate visual comparisons across the plots, all three employ the same y-axis. Some of the data tops out at 3600 seconds since that is where we set the MIP time limit.

graph model). The graphs have $50 \leq n \leq 200$ nodes, and are converted to directed acyclic graphs via a random topological order. The size of each tensor is sampled from the normal distribution $\mathcal{N}(50, 10)$. Each node cost is the sum of its input and output tensor costs plus a random fraction r of the total memory cost (i.e., the sum of all tensor sizes), where $r \sim \mathcal{N}(0, 0.1)$. If a node has more than one output tensor, we use the lexicographically least according to the tensor index. Finally, Paliwal et al. (2020) filter these graphs and only keep those that are sufficiently hard for their min-peak scheduling objective.

Results We report a parallel set of results on the REGAL dataset (see Figure 5 and Table 3) to accompany our running time plots and table of lower bound ratios for the production models in Section 5.

Table 3. Geometric means of the best available lower bound from the MIP hierarchy, normalized by the best solution found using BKRGA, across the REGAL dataset.

Lower bound	$k = 2$	$k = 4$	$k = 8$	$k = 16$	$k = 32$	$k = 64$
`simple` (Lemma 3.3)	0.9579	0.8795	0.7911	0.5821	0.4087	0.3336
`bottleneck`	0.9794	0.8818	0.7918	0.5824	0.4090	0.3338
`bottleneck-guess`	0.9804	0.8826	0.7918	0.5824	0.4090	0.3338
`exact`	0.9804	0.9579	0.9407	0.8929	0.5910	0.3810

C.3. Solving the MIPs

To solve the MIPs that underpin Table 1 and Table 3, we used a combination of the Gurobi (Gurobi Optimization, LLC, 2023) and SCIP (Bestuzheva et al., 2021) solvers. For `bottleneck`, we used Gurobi with a 15-minute time limit, and for `exact`, we relaxed this to 60 minutes because these MIPs are tougher to solve. For technical reasons having to do with our computing setup, we instead used SCIP for `bottleneck-guess`, with a 60-minute time budget that was shared across all values of k of the MIPs that compose a single `bottleneck-guess` instance (one MIP per guess). A non-trivial fraction of the instances failed to solve to provable optimality within the time limit, especially for `exact` with $k = 64$. In these cases, the solver still returns a valid lower bound, and we use that.