HAP^5: Heterogeneity-Aware Pipeline Parallelism Planning and Performance Prediction

Anurag Dutt* Stony Brook University
Buvana Ramanan Nokia Bell Labs
Brian Friedman Nokia Bell Labs
Anshul Gandhi† Stony Brook University
Manzoor Khan Nokia Bell Labs

***Abstract*—We present HAP^5, a hardware-agnostic framework for performance prediction and planning of heterogeneous pipeline-parallel training and inference. HAP^5 combines symbolic tracing, a tile-aware compute model, explicit inter-stage communication prediction, and stage-aware composition to estimate end-to-end latency without profiling on the target hardware. It further uses an MILP-based planner to recommend heterogeneous stage partitions and GPU placements. Across heterogeneous deployments of GPT2-Medium, GPT2-Large, and Llama2, HAP^5 achieves average prediction errors of 12.3% for training and 7.9% for inference, significantly outperforming existing baselines. Further, our MILP-selected, non-uniform pipeline stage partitions reduce latency/token by 15.0% on average for training and 14.8% for inference relative to the default, uniform partitioning.**

***Index Terms*—LLMs, latency prediction, pipeline parallelism**

I. Introduction

Large language models (LLMs) have transformed applications in search, code generation, and scientific analysis, but their growing model sizes and sequence lengths have substantially increased the computational and memory demands of training and inference. This has intensified pressure on GPU availability and cost, motivating better use of existing GPU resources. In many settings, individuals and organizations have temporarily idle GPUs that can be contributed to a ***shared compute pool***, where users can discover available devices and provision execution environments. However, many such GPUs are heterogeneous edge devices, workstations, or laptops that cannot independently host an entire LLM. **Pipeline parallelism (PP)** makes these fragmented resources usable by partitioning the model across devices, so each GPU stores and executes only a subset of layers. This setting is increasingly relevant in decentralized and federated compute platforms such as OurAI [1] and PrimeIntellect [2], which expose ***geographically distributed, and heterogeneous GPUs*** on demand.

Given a pool of available and distributed GPUs, deciding which subset of GPUs to use to run a model is a non-trivial problem. Performance depends ***jointly*** on the compute capabilities of the GPUs (e.g., throughput, memory capacity) and on the communication fabric connecting them (e.g., PCIe, NVLink, or wide-area networks). Further, empirically benchmarking various combinations and subsets of available GPUs to evaluate end-to-end performance is often prohibitively expensive in terms of time and resources. To make informed and scalable decisions, users need the ability to ***predict*** how their model will perform on a given combination of GPUs and interconnects. However, accurately predicting model performance in this distributed setting is challenging for several key reasons:

- **Lack of hardware access for profiling.** Target GPUs may not be accessible prior to deployment, requiring a prediction framework that operates without direct profiling of the hardware. Moreover, compute capabilities of the available GPUs (e.g., throughput, memory bandwidth, architecture) themselves may vary significantly, complicating accurate performance estimation in the absence of direct measurements.
- **Heterogeneous communication.** In distributed inference and training, end-to-end performance often critically depends on the communication infrastructure. Given that GPUs in such shared pools can be connected via diverse interconnects (e.g., NVLink, PCIe, or wide-area networks), predicting communication latency reliably for an arbitrary model deployment is a difficult problem.
- **Pipeline scheduling effects.** Under pipeline parallelism, scheduling effects such as pipeline bubbles can arise during training and significantly inflate latency when pipeline stages are imbalanced. Such scheduling effects complicate prediction because end-to-end latency is no longer a simple sum of per-stage compute times; instead, it depends on stage imbalance and the non-uniform overlap between computation and communication introduced by pipeline bubbles [3].
- **Optimal pipeline partitioning across heterogeneous GPUs.** Even with accurate performance predictions, determining how to partition the model into pipeline stages across distributed and heterogeneous GPUs for best performance is a combinatorial optimization problem.

There has been some prior work on auto-parallelization and performance modeling for large-scale deep learning systems, including frameworks such as Colossal-AI [4], Alpa [5], and related auto-parallel planners [6]. However, they ***require access to the target hardware*** for profiling, calibration, or direct execution to guide their cost models and search procedures. This assumption does ***not*** hold in decentralized or pre-provisioning environments where GPUs are distributed, heterogeneous, and unavailable for direct profiling *before* deployment.

In this paper, we remedy this gap by ***designing HAP^5, a hardware-agnostic performance prediction framework for LLMs*** that applies to both training and inference. HAP^5 predicts end-to-end latency for a given model under pipeline parallelism across arbitrary combinations of GPUs and communication fabrics. HAP^5 also supports different batch sizes, sequence lengths, and precision configurations, produces fast latency estimates, and can recommend optimal pipeline stage partitions across available GPUs. Importantly, HAP^5 only assumes

*This work was developed at Nokia Bell Labs during the first author's internship, and †was supported in part by NSF grants CCF-2324859, CNS-2214980, and CNS-2106434.

knowledge of high-level hardware characteristics (e.g., peak performance, link bandwidth) rather than requiring access to the target hardware for profiling. This enables users to assess candidate GPU clusters efficiently before provisioning.

To address the challenge of predicting device-specific compute latency in the absence of hardware access, we decompose the model into fine-grained computational primitives using ***symbolic tracing***. Rather than treating layers as monolithic units, this decomposition exposes tensor shapes and operator structure, enabling hardware-agnostic performance estimation based on architectural characteristics such as peak throughput and memory bandwidth. To address heterogeneous communication and pipeline scheduling, we infer adjacent stage-boundary tensor sizes from the traced graph and combine them with available interconnect bandwidth information to estimate communication costs without execution. We then integrate these to find the ***stage-aware critical-path*** to predict end-to-end performance. To address optimal stage partitioning across heterogeneous and distributed GPUs, we formulate pipeline partitioning and stage placement as a mixed integer linear program (MILP) that minimizes the predicted latency subject to feasibility constraints (e.g., device memory and interconnect constraints). This enables selecting splits and GPU assignments that balance per-stage processing times so that stages progress at similar rates without synchronization issues. By contrast, existing approaches employ uniform partitioning, which we show to be suboptimal under heterogeneous environments.

Across GPT2-Medium, GPT2-Large, and Llama2, HAP[5] achieves average latency prediction error of 12.3% for training and 7.9% for inference, while its MILP-selected non-uniform partitions reduce latency/token by 15% and 14.8% for training and inference, respectively. Overall, HAP[5] achieves the lowest prediction error compared to prevalent baselines, across models, hardware, and PP-configurations, indicating that its heterogeneous PP planner accurately predicts end-to-end latency.

II. Background

Decentralized training and inference (DeTI) platforms aggregate available GPU capacity across independently operated providers, offering an alternative to traditional cloud clusters. OurAI, PrimeIntellect, and Akash Networks are representative real-world examples: decentralized, identity- and contract-driven GPU sharing marketplaces that match compute providers and consumers under multi-stakeholder incentives [1], [2], [7]. Across these platforms, the consumer-facing promise is access to GPUs beyond hyperscalers, often with more flexible pricing and supply. The systems reality is that the resulting pool is heterogeneous in compute/memory capability (e.g., server-class GPU vs. edge GPU) and network connectivity (e.g., NVLink vs. Ethernet). These properties make distributed execution planning substantially harder than in centralized clusters, where hardware and network topologies are typically known in advance.

In such DeTI settings, deep learning workloads require distributed execution mechanisms to efficiently leverage fragmented resources. **Parallelism** provides this mechanism by partitioning the workload across multiple GPUs so that compute and memory demands can be distributed. There are three primary parallelism strategies for distributed deep learning: data parallelism (DP), tensor parallelism (TP), and pipeline parallelism (PP) [8]. DP replicates the entire model on all devices (GPUs) and splits the input across replicas in each iteration. TP partitions the computation within large layers across GPUs to execute a single layer collaboratively [9], [10]. PP partitions the model by layers into sequential stages placed on different GPUs, executing micro-batches through these stages via a pipelined schedule [3], [11].

In heterogeneous DeTI settings, DP is often impractical because many contributed GPUs cannot host the full model at useful batch/output sizes, while TP incurs frequent collective communication whose performance is bottlenecked by the slowest interconnects in a heterogeneous cluster [9], [10]. As a result, PP is often the most practical mechanism for leveraging distributed GPUs in DeTI platforms. It partitions the model into sequential stages placed on different GPUs, and processes each batch as a sequence of micro-batches that flow through the stages in a pipelined manner so multiple stages can be active concurrently [3], [11]. This translates to higher performance, as confirmed by our experiments (Table I, see Section V-A for setup). We thus employ PP to leverage the distributed GPUs in our work.

TABLE I: Throughput achieved by different parallelism strategies.

Model	Parallelism	Tokens/s
GPT2 M.	DP	4.8
GPT2 M.	TP	1.4
GPT2 M.	PP	10.9
Llama2	DP	2.1
Llama2	TP	0.8
Llama2	PP	6.4

III. Related Work

To our knowledge, there is no prior work on *plan-specific performance prediction for distributed parallel executions*, especially under heterogeneous devices and interconnects, and without access to the target hardware. We now summarize the most relevant works below.

Auto-parallel planners and parallelism frameworks. Several systems support DP/TP/PP and automate configuration selection through search or optimization. Colossal-AI and Colossal-Auto provide a unified interface for large-scale training and automate parallelism and memory trade-offs [4], [6]. Alpa searches over inter-operator partitioning and intra-operator sharding to generate distributed execution plans [5], Slapo offers a schedule language for progressive graph transformations and optimization [12], and Mist co-optimizes memory and parallelism over a large configuration space [13]. While effective in controlled environments, these systems typically rely on target-hardware availability for profiling, calibration, or online cost estimation, and are commonly evaluated on homogeneous or tightly managed clusters.

Communication prediction. For communication prediction, many systems use simple analytical models that estimate transfer time from message size and link characteristics (e.g.,

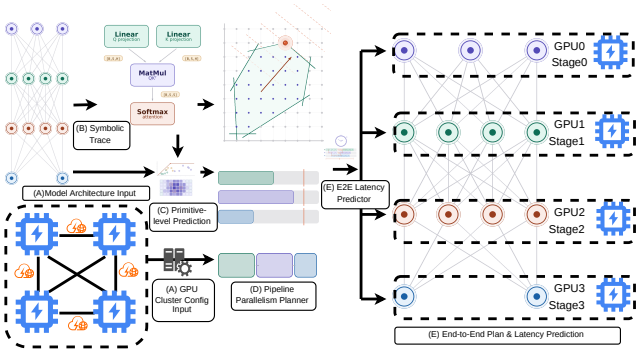

Fig. 1: HAP[5] architecture showing (A) model architecture and GPU cluster configuration inputs, (B) symbolic trace (C) primitive-level latency prediction, (D) pipeline-parallel planning, and end-to-end latency prediction stages.

the α–β / Hockney model and the LogP/LogGP family) [14]–[16]. Similar to modern deep-learning planners such as Alpa [5], we model communication cost from message size and link bandwidth, but derive stage-boundary tensor sizes via symbolic tracing and compose these costs with the pipeline schedule to estimate end-to-end latency under heterogeneous PP.

Performance prediction models. This line of work predicts latency using architecture and hardware-aware predictors that can be used for model-level prediction. NeuSight and Habitat are especially relevant, as they decompose the model to map tensor shapes to device execution behavior and improve fidelity beyond FLOP-based heuristics [17], [18]. However, these predictors focus on single-GPU execution and do not directly solve distributed planning. HAP[5] builds on this direction by using primitive-level prediction as input while explicitly modeling stage boundaries, heterogeneous fabrics, and stage-aware composition for a *given* pipeline configuration.

IV. Design

To enable hardware-agnostic PP planning in DeTI settings, HAP[5] scores candidate GPU sets and interconnects without target-hardware benchmarking. Figure 1 summarizes its workflow: structure extraction, cost prediction, PP planning, and end-to-end latency estimation.

A. Inputs and outputs for HAP[5]

The planner in HAP[5] takes as input: (i) the architecture of the LLM to be executed, (ii) the runtime configuration parameters of the LLM (batch size, sequence length, dtype/quantization, micro-batch count, training vs. inference), and (iii) a heterogeneous cluster description (Flops, per-device memory, and an interconnect topology capturing bandwidth between endpoints). The output is a *pipeline configuration and GPU placement*: an ordered set of stages where each stage is defined by a contiguous layer range and a GPU assignment, together with predicted end-to-end latency (including a breakdown into per-stage compute and inter-stage communication components).

B. Symbolic tracing of the model's computation graph

The first step in HAP[5] is to use *symbolic tracing* to extract, *without execution on the target hardware*, the model's constituent compute primitives and tensors that drive PP execution and communication.

For the transformer-style LLMs that we target (models composed of a stack of repeated blocks/layers), symbolic tracing yields a stable sequence of primitives per block (e.g., linear projections, attention matmuls, MLP projections, and normalizations) together with their tensor shapes. These shapes are determined by the execution setting: micro-batch size B_μ, sequence length S, and hidden dimension H (from the model configuration). The traced shapes allow us to derive hardware-agnostic per-layer activation memory footprints (bytes) and compute volumes (e.g., FLOPs), which allow us to formulate the PP optimization and aid in performance prediction.

For any candidate split between layers i and $i{+}1$, the key communication is the boundary activation tensor transferred from stage i to $i{+}1$ (and the corresponding backward-direction tensor in training). We derive these tensor sizes directly from traced shapes (typically proportional to $B_\mu \times S \times H$), convert to bytes using the configured dtype/quantization, then feed it to the communication model in the prediction engine.

C. Prediction engine

The prediction engine serves two roles in HAP[5]. First, it produces the cost coefficients needed by the MILP: layer-wise compute latency estimates, and boundary communication latency estimates derived from payload sizes. Since the planner evaluates many candidate plans, obtaining these layer-wise costs once (for a fixed model architecture and execution setting) enables fast scoring of alternative stage boundaries and placements. Second, given a concrete PP plan (stage ranges and GPU assignments), it composes per-stage compute and per-boundary communication costs to estimate end-to-end latency.

1) Compute prediction: For each primitive p (e.g. BMM, GEMM) extracted from symbolic tracing, we estimate its execution time on GPU type g as $\hat{t}_{p,g}$ using a shape-aware, tiling-based model inspired by NeuSight [17]. This hardware-agnostic predictor uses (i) primitive shapes from symbolic tracing and (ii) a per-GPU capability profile, including peak tensor/FP throughput, memory bandwidth, SM count/clock, and related architectural parameters, obtained from specifications or one-time offline calibration. Following NeuSight, we compute $\hat{t}_{p,g}$ by decomposing the primitive into tiles, estimating the number of tile "waves" from the tile count and SM count of GPU type g, and deriving per-tile latency from a roofline-bounded achieved throughput (roofline $\times$ predicted utilization) before aggregating across waves.

We then aggregate primitive-level predictions into a per-layer compute latency coefficient. Let $\mathcal{P}_l$ denote the set of traced primitives in layer l; then, for GPU type g, the per-layer latency coefficient is $\hat{c}_{l,g} = \sum_{p \in \mathcal{P}_l} \hat{t}_{p,g}$. This latency coefficient allows the MILP to quantify the cost of placing layer l on GPU type g. For any candidate plan, the compute time of a stage is obtained by summing $\hat{c}_{l,g}$ over the layers assigned to that stage, since layers within a stage execute sequentially for each micro-batch. Cross-stage concurrency (e.g., 1F1B) is handled separately by the stage-aware composition step (Section IV-E), rather than by altering the per-stage compute aggregation.

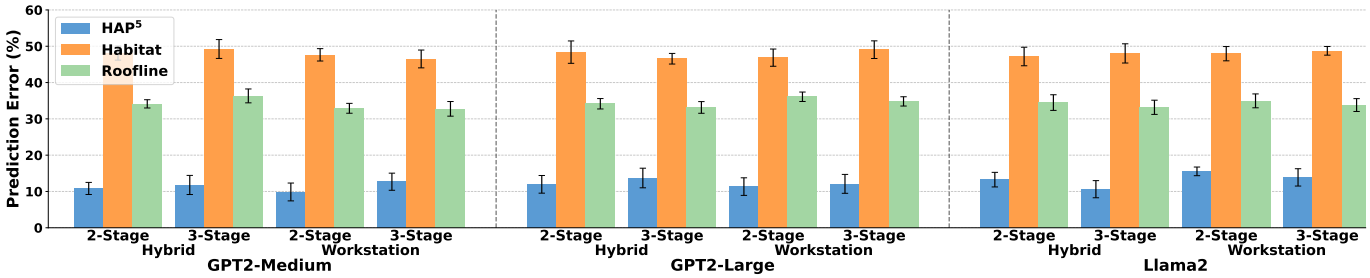

Fig. 2: Prediction Errors for HAP^5 across all models, pipeline depths, and experimental setups - Training.

2) Communication prediction: Inter-stage communication under PP is dominated by transferring boundary tensors between adjacent stages. Using (i) the boundary payload b_k (bytes) obtained from symbolic tracing, and (ii) the interconnect characteristics between the selected endpoints, we estimate transfer time with a standard α-β model [14] as:

$$\mathrm{Comm}(b_k; g, g') \;=\; \alpha_{g,g'} + \beta_{g,g'}\, b_k \tag{1}$$

where g and g' denote the source and destination GPU endpoints for the boundary transfer between consecutive stages. In the α-β model, $\alpha_{g,g'}$ captures the fixed per-message latency on the link from g to g', and $\beta_{g,g'}$ captures the per-byte transmission cost (i.e., the inverse of the effective bandwidth) on that link. Since DeTI deployments can mix fabrics (intra-node NVLink/PCIe, inter-node Ethernet), $\{\alpha_{g,g'}, \beta_{g,g'}\}$ is specified per endpoint pair in the cluster description. HAP^5 models communication using the pre-execution cluster description and assumes that stage placements remain fixed during a planned run. Transient network congestion or runtime parameter migration would require online monitoring and adaptive re-optimization, which is beyond our current scope.

D. MILP formulation for PP partitioning and GPU placement

Given a heterogeneous GPU pool and the traced model structure, the planner must choose (i) where to cut the model into stages (contiguous layer ranges), and (ii) which GPU runs each stage. In heterogeneous settings, uniform splits can create imbalance (pipeline bubbles) and expensive boundaries over slow links (see Section V). To address these requirements, we formulate and solve an MILP that jointly selects layer-to-stage assignment and stage-to-GPU placement, while minimizing makespan proxy and enforcing feasibility (e.g., memory capacity).

Let $l \in \{1, \ldots, L\}$ index the layers of the model, $k \in \{1, \ldots, K\}$ index the stages, and $g \in \mathcal{G}$ index the available devices/GPUs. We use binary variables $a_{l,k}$ (assign layer l to stage k) and $y_{k,g}$ (place stage k on device g), and a scalar $T \geq 0$ (makespan proxy). We minimize the maximum per-stage "work" (compute plus boundary communication):

$$\begin{aligned}
\min \quad & T \\
\text{s.t.} \quad & \textstyle\sum_{k=1}^{K} a_{l,k} = 1 \quad \forall l \\
& \textstyle\sum_{g \in \mathcal{G}} y_{k,g} = 1 \quad \forall k \\
& T \geq \textstyle\sum_{l,g} \hat{c}_{l,g}\, a_{l,k}\, y_{k,g} + \mathrm{Comm}(b_k; g, g') \\
& \qquad \forall k \in \{1, \ldots, K-1\} \\
& \textstyle\sum_{l} m_l\, a_{l,k} + r_k \leq \textstyle\sum_{g} \mathrm{Mem}_g\, y_{k,g} \quad \forall k
\end{aligned} \tag{2}$$

Here, m_l is parameter memory requirement for layer l, r_k for stage k, and Mem_g is available memory on GPU g. For each boundary k, g and g' denote the GPUs selected for stages k and $k{+}1$ by $y_{k,g}$ and $y_{k+1,g'}$. $\mathrm{Comm}(b_k; g, g')$ is computed using the α-β model in Eq. (1) (implemented via standard MILP linearization for device-pair selection). To instantiate the MILP, we derive both compute and communication coefficients ($\hat{c}_{l,g}$ and b_k) from the traced shapes, as discussed in prior subsections. We also enforce that stages correspond to contiguous layer ranges and preserve order using contiguity constraints (omitted here for brevity). Essentially, for a fixed candidate GPU subset and a fixed pipeline stage count K, our MILP jointly optimizes the layer-to-stage assignment and stage-to-GPU placement by minimizing a bottleneck-stage proxy that accounts for per-stage compute and boundary communication costs, subject to memory-capacity and contiguity constraints. The choice of K and the candidate GPUs is handled outside the MILP.

While MILPs are NP-hard in the worst case, the instances induced by PP planning have modest dimensionality (contiguous stage ranges and a small number of candidate GPUs), and we solve them in under 10 seconds using PuLP [19], an off-the-shelf MILP solver backend. This solve time is short enough to support re-planning when the candidate GPU pool changes, allowing HAP^5 to quickly recalibrate stage partitions and placements if GPUs are added to or removed from the cluster. Feasibility is explicitly checked via the memory constraints; if a candidate GPU set admits no feasible partition, the solver reports infeasibility and the planner can fall back to a different subset or, a different stage count K.

E. Stage-aware critical-path latency prediction

Using this solution, we instantiate the per-stage compute latency estimate, $\sum_{l,g} \hat{c}_{l,g} a_{l,k} y_{k,g}$, and the boundary communication latency estimate, $\mathrm{Comm}(b_k; g, g')$. We then find the stage-aware critical-path to compose these estimates into an end-to-end latency prediction.

Inference. We estimate end-to-end inference latency by summing per-stage forward compute costs and forward boundary transfer costs as each micro-batch traverses the stages in order.

Training. Here, the end-to-end iteration time is not a simple sum because stages process different micro-batches concurrently under the 1F1B schedule [3]. We therefore perform a stage-aware critical-path analysis by reconstructing the 1F1B execution over micro-batches and stages: for each {stage, micro-batch}, we compute the earliest start time of its forward/backward step from the completion of its predecessor steps on the same stage and adjacent stages, and then propagate these dependencies to identify the critical path. This captures pipeline bubbles and backpressure induced by heterogeneous stage and link speeds, yielding a schedule-consistent end-to-end latency estimate.

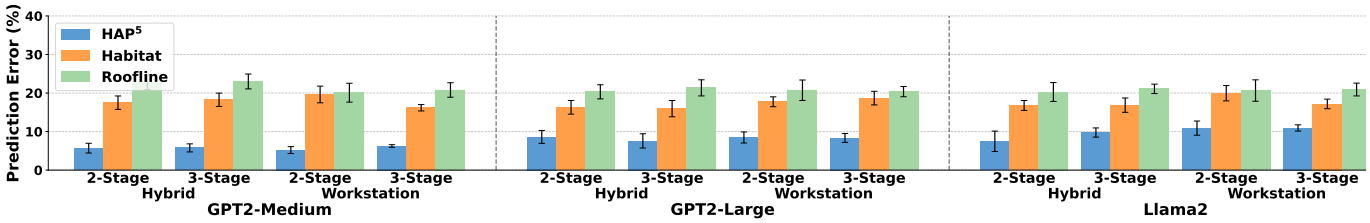

Fig. 3: Prediction Errors for HAP[5] across all models, pipeline depths, and experimental setups - Inference.

V. Evaluation

This section evaluates the effectiveness of HAP[5] from two complementary perspectives: (i) our latency prediction accuracy, including comparison against two adapted baselines, and (ii) the performance gains afforded by our MILP-based PP stage partitioning relative to the default, uniform PP stage split.

A. Experimental Setup

We evaluate heterogeneous PP execution under a 1F1B schedule for two pipeline depths (referring to number of partitions/GPUs), PP$=2$ and PP$=3$, using two setups that capture practical heterogeneous nodes with differences in compute throughput and memory capacity.

Hybrid Setup uses an RTX A6000 and RTX A5000 for PP$=2$, and an RTX A6000, RTX A5000, and RTX A3500 for PP$=3$. It represents a hybrid workstation/laptop (edge) setting, where the A6000 is hosted on a workstation with 10 GbE connectivity, while the A5000 and A3500 communicate over 1 GbE links.

Workstation Setup uses an RTX A4000 and RTX A5000 for PP$=2$, and an RTX A4000 with two RTX A5000 GPUs for PP$=3$. It represents a workstation-only setting in which all three GPUs are hosted on separate workstations connected via 1 GbE. For both setups, we evaluate (i) GPT2-Medium, (ii) GPT2-Large, and (iii) Llama2 in both training and inference. For PP$=2$, we sweep batch sizes $\{4, 8, 16\}$ and sequence lengths $\{512, 1024\}$; for PP$=3$, we sweep batch sizes $\{6, 12, 18\}$ with the same sequence lengths.

B. Baselines

We compare HAP[5] against two *adapted* baselines: *Habitat* and *Roofline*. Since neither baseline is originally designed as a full heterogeneous PP planner, we adapt both into a common evaluation harness that isolates the quality of the *device-side compute model*: all methods use the same traced model graph, primitive decomposition, candidate PP stage boundaries, inter-stage communication sizes, and schedule-aware end-to-end composition for training and inference, and differ only in how per-primitive compute latency is estimated.

Adapted Habitat. Habitat serves as an analytical-model baseline. We replace HAP[5]'s primitive-level compute predictor with a Habitat-style analytical latency model that estimates each primitive's execution time from its tensor/operator characteristics and hardware properties. These primitive estimates are then summed to obtain per-stage compute times, which are combined with the same communication model and PP schedule used by HAP[5] to produce end-to-end latency predictions [18].

Adapted Roofline. Roofline serves as a hardware-bound baseline. For each primitive, we estimate latency using a Roofline-style bound based on its arithmetic work and memory traffic, together with the target device's peak FLOPs and memory bandwidth. This yields a per-primitive lower-bound latency estimate, which is then aggregated into per-stage compute times and composed through the same communication and PP execution model as HAP[5] [17].

C. Evaluating Prediction Accuracy

Figures 2 and 3 summarize our prediction results across all configurations formed by 3 models, 2 hardware setups, and 2 PP depths. Each grouped cluster corresponds to one fixed $\{\text{model}, \text{setup}, \text{PP depth}\}$ configuration and contains three bars, denoting the end-to-end latency MAPE achieved by HAP[5], Habitat, and Roofline. The x-axis is organized hierarchically: the outer grouping corresponds to model, within each model block the next grouping corresponds to Setup-1 and Setup-2, and within each setup block the two clusters correspond to 2-stage and 3-stage PP. Ground truth is the latency measured from the actual Colossal-AI/vLLM execution of that parallel deployment configuration. Thus, lower prediction error means the predicted latency is closer to the observed runtime. For brevity, we aggregate results across the batch-size and sequence-length sweep for each fixed configuration for uniform stage splits. Specifically, each bar reports the *median* MAPE across all batch-size/sequence-length runs for that configuration, while the error bar denotes the corresponding standard deviation.

Training. Figure 2 reports our prediction results for training. Across all 12 $\{\text{model}, \text{setup}, \text{PP depth}\}$ configurations, ***HAP[5] is consistently the most accurate predictor***, with median error ranging from 9.9% to 15.5%, with an average error of 12.3%. In comparison, Roofline ranges from 32.8% to 36.3% with an average of 34.2%, while Habitat ranges from 46.5% to 49.2% with an average of 47.9%. Thus, HAP[5] lowers training prediction error by 21.9–26.4 percentage points relative to Roofline and by 31.0–37.0 percentage points relative to Habitat across the plotted configurations. This corresponds to a 64.0% average reduction in error relative to Roofline and a 74.3% average reduction relative to Habitat.

Inference. Figure 3 shows the same comparison under inference. For inference as well, ***HAP[5] is always the most accurate method*** in every cluster. Its median inference error ranges from 5.2% to 10.9%, with an average of 7.9%. Habitat ranges from 15.9% to 19.9% (average 17.6%), while Roofline ranges from 20.1% to 23.0% (average 21.0%). Averaged across all configurations, HAP[5] lowers inference error by 54.8% relative to Habitat and by 62.2% relative to Roofline. Overall, the inference results confirm the same trend as training: HAP[5] generalizes consistently across models, setups, and PP depths while achieving much lower error than the baselines.

We also tested HAP[5] under FP16 precision and find that the prediction error remains within 8%–17%, indicating that the predictor remains stable across training and inference under both single and half precision.

Insights from results. The key advantage of HAP[5] is its decomposition: primitive compute is predicted with a workload-

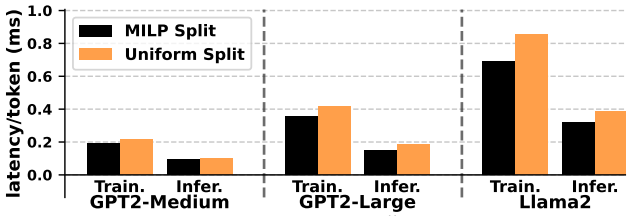

Fig. 4: Throughput gains via HAP[5]'s heterogenous split.

aware model, communication is predicted explicitly from tensor transfer cost, and both are composed under the target PP schedule. HAP[5] further improves compute prediction through a tile-aware model that estimates execution at the granularity actually used by modern GPU kernels, rather than treating an operator as a single monolithic cost. By modeling tiling-induced effects such as kernel decomposition, wave/occupancy behavior, and memory movement more faithfully, HAP[5] produces more accurate stage-level compute estimates, while also explicitly modeling communication and schedule composition. By contrast, Habitat predicts kernel latency from coarse, high-level operator and GPU features, which misses hardware/software execution details and generalizes poorly across unseen dimensions and GPUs. Similarly, Roofline provides only an idealized bound from arithmetic intensity, peak FLOPs, and memory bandwidth, thus missing realized utilization and latency-hiding effects. On top of these single-device compute-model limitations, neither baseline explicitly models PP parallelism, i.e., stage imbalance, inter-stage communication, and pipeline bubbles.

D. MILP-Guided Non-Uniform Stage Partitioning

We next evaluate whether HAP[5]'s MILP-based PP stage partitioner improves execution over the default, uniform layer split [4], [10] on a heterogeneous 3-GPU pool. Figure 4 is organized by model, with two groups per model corresponding to training and inference; within each group, the two bars compare the MILP-guided split on our workstation setup against the uniform split, and lower latency/token is better. The experiments were run on the Workstation Setup, with 2×A5000 GPUs and 1×A4000 GPU. The MILP selects non-uniform 3-stage layer partitions of $(9, 9, 6)$, $(13, 13, 10)$, and $(12, 12, 8)$ for GPT2-Medium, GPT2-Large, and Llama2, respectively, assigning fewer layers to the slower, final stage (on the A4000). As shown in Figure 4, our ***MILP-guided splits consistently reduce per-token latency in both training and inference***: for GPT2-Medium by 12.1% and 10.7%, for GPT2-Large by 14.3% and 17.3%, and for Llama2 by 18.7% and 16.5%, corresponding to speedups of up to $1.23\times$ in training and $1.21\times$ in inference. In terms of latency prediction accuracy, even for the non-homogeneous PP stage split experiments in Figure 4, HAP[5] achieves low prediction error (9.85% to 13.51%), indicating its robustness to stage splits.

VI. Conclusion

As democratized and decentralized GPU markets continue to emerge, effectively leveraging contributed and heterogeneous resources requires accurate performance prediction prior to deployment. In this paper, we presented HAP[5], a hardware-agnostic framework for heterogeneous pipeline-parallel planning and latency prediction that combines symbolic tracing, communication modeling, and schedule-aware composition. Our results show that HAP[5] can accurately estimate end-to-end performance and help identify effective PP deployments without requiring target-hardware benchmarking.

References

[1] S. Bidkar, A. Gudal, L. Drabeck, J. E. Simsarian, B. Theeten, and M. Khan, "Ai agent-driven network-aware decentralized compute resource brokering," in *Proc. Optical Fiber Communication Conference (OFC)*.

[2] Prime Intellect, "Prime intellect: The decentralized ai development platform," 2026.

[3] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[4] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You, "Colossal-ai: A unified deep learning system for large-scale parallel training," in *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*. Association for Computing Machinery, 2023.

[5] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[6] 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.

[7] Overclock Labs, "Akash network: The decentralized cloud computing marketplace," https://akash.network, 2026.

[8] J. Zhou, Y. Chen, Z. Hong, W. Chen, Y. Yu, T. Zhang, H. Wang, C. Zhang, and Z. Zheng, "Training and serving system of foundation models: A comprehensive survey," *IEEE Open Journal of the Computer Society*.

[9] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, O. Kuchaiev, D. Vainbrand, S. Mishra, R. Mathur, M. Jiang, X. Yang, Y. Zhu, Y. Wang *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[10] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020.

[11] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, 2019.

[12] H. Chen, C. H. Yu, S. Zheng, Z. Zhang, Z. Zhang, and Y. Wang, "Slapo: A schedule language for progressive optimization of large deep learning model training," 2023.

[13] Z. Zhu, C. Giannoula, M. Andoorveedu, Q. Su, K. Mangalam, B. Zheng, and G. Pekhimenko, "Mist: Efficient distributed training of large language models via memory-parallelism co-optimization," 2025.

[14] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2: Architecture, Programming, and Algorithms*. A. Hilger, 1988.

[15] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," in *4th Symposium on Principles and Practice of Parallel Programming*, 1993.

[16] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model for parallel computation," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1.

[17] S. Lee, A. Phanishayee, and D. Mahajan, "Forecasting gpu performance for deep learning training and inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery.

[18] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based computational performance predictor for deep neural network training," in *2021 USENIX Annual Technical Conference*, Jul. 2021.

[19] C. C. N. Kuhn, G. Calbert, I. Garanovich, and T. Weir, "Integer linear programming supporting portfolio design," 2023. [Online]. Available: https://arxiv.org/abs/2303.14364