Task Scheduling for Decentralized LLM Serving in Heterogeneous Networks

Elden Ren

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-111
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-111.html

May 16, 2024

Acknowledgement

I express my sincere gratitude to my advisor Dawn Song for her support and mentorship over the past three years and to Professor Jacob Steinhardt for being my second reader and providing insightful feedback.

I would like to thank my mentor Xiaoyuan Liu for his guidance and feedback through this process. I would also like to thank my collaborators Tianneng Shi and Zhongjing Wei.

Finally, I express my heartfelt gratitude to my family and friends for their unwavering encouragement and support, which has been indispensable in every aspect of my life.

Task Scheduling for Decentralized LLM Serving in Heterogeneous Networks

by Elden Ren

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Dawn Song

Professor Dawn Song
Research Advisor

5/6/24

(Date)

* * * * * * *

Jacob Steinhardt

Professor Jacob Steinhardt
Second Reader

5/15/24

(Date)

Task Scheduling for Decentralized LLM Serving in Heterogeneous Networks

by

Elden Ren

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor Jacob Steinhardt

Spring 2024

TASK SCHEDULING FOR DECENTRALIZED LLM SERVING IN HETEROGENEOUS NETWORKS

Elden Ren
UC Berkeley
elden.ren@berkeley.edu

ABSTRACT

The rapid expansion of generative AI and its integration into daily workflows has magnified the demand for large language model (LLM) inference services. However, the deployment of LLM models is often burdened by the high cost and limited availability of GPU resources. A Decentralized Physical Infrastructure Network (DePIN) tailored for LLM inference could leverage idle GPU resources globally to enable decentralized serving of LLMs. In this paper, we discuss the challenge of scheduling model pipelines in a heterogeneous network to enable fast decentralized serving. We focus on the metric of time per output token (TPOT) [1], which measures the latency to generate each token after the first token. We present a novel heuristic scheduling algorithm that enables fast inference in decentralized LLM serving systems and is practical for large, heterogeneous networks. The experimental results show the feasibility of using consumer-grade GPUs for low-latency LLM inference and validate the effectiveness of our algorithm. The proposed algorithm achieves uniformly lower TPOT than the integer programming baseline and does so with a shorter execution time.

1 Introduction

Interest in generative AI, especially text-generating chatbots, has significantly increased in recent years. Millions of individuals [2] and many companies [3] are now using AI chatbots in their daily workflows, boosting the demand for large language model (LLM) services. Closed-source models are popular, but open-source models have also been quickly improving and now show competitive performance in chatbot evaluations [4]. This has drawn industry attention, leading to the emergence of companies providing paid services for running open-source LLM inference.

Providing sufficient GPU computing resources is a major challenge for LLM-hosting companies. They can either use cloud services or build their own computing clusters. Cloud services are costly and often require negotiations for availability. The alternative of buying high-end GPUs to perform LLM inference also leads to significant expenses. However, there is a large source of untapped GPU compute in consumer-owned GPUs–the PC and AIB GPU market shipped nearly 10 million units in Q4 2023 alone [5], and consumer-owned GPUs, which are commonly used for purposes like gaming and photo/video editing, are often not in continuous use.

A recent Decentralized Physical Infrastructure Networks (DePIN) project [6] uses decentralized GPU nodes worldwide to provide rendering services. Inspired by this practice, we consider utilizing idle GPU resources worldwide for a decentralized LLM serving system which potentially provides better availability, achieves better cost-efficiency, and rewards GPU owners.

Compared to rendering tasks, the decentralized serving of LLMs presents unique system design challenges. First, many interactive applications such as chatbots rely on timely responses from the serving endpoint. A practical decentralized LLM serving system system must achieve low latency with efficient model inference. Second, as indicated by the name, LLM inference workloads are memory-intensive. One serving task using a larger model may not be feasible to compute on a single GPU, as the memory requirement of storing all the weights may require multiple GPUs to fit.

With model pipelining [7], it is possible to serve an LLM request using a circuit of multiple machines, with each machine computing some layers and passing the result on to successive machines. Pipelining allows for machines that

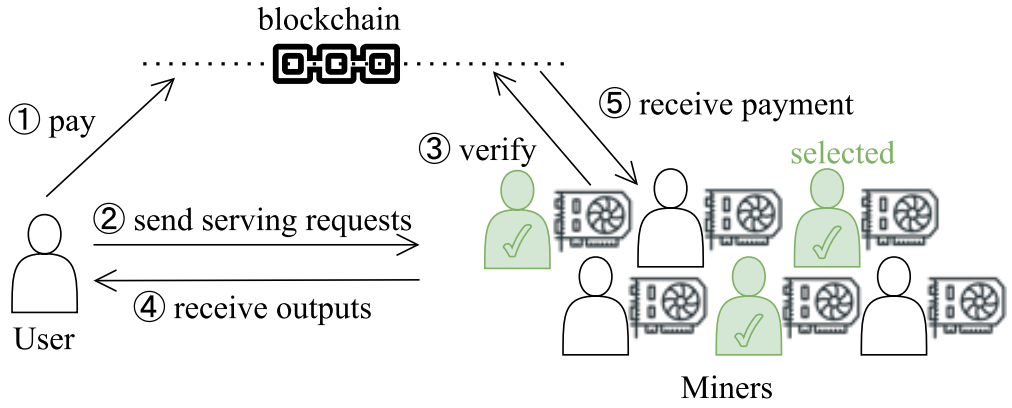

Figure 1: An illustration of decentralized LLM serving.

otherwise could not fit an entire LLM in GPU memory to contribute to a decentralized LLM serving network. This enables more machines to provide GPU computing power to those who require it.

In this paper, we specifically consider the problem of scheduling LLM inference workloads to optimize inference speed in a decentralized network. The inference speed for a serving system has a considerable impact on its user experience. Thus, we focus on the metric of time per output token (TPOT) [1], which measures the latency to generate each token after the first token. As most requests generate dozens of tokens, TPOT has the most significant impact on the end-to-end request finishing time. Unlike in centralized serving, where the devices are homogeneous and closely clustered, in the decentralized setting, the heterogeneity in GPU devices and network conditions means different node selections and layer arrangements can result in significant differences in the TPOT for a given request.

The challenge of building a decentralized LLM serving system that achieves tolerable latency is multiple. First, most API services are only affected by the network latency once, but this is not the case for decentralized LLM serving. With model-pipelined auto-regressive decoding, generating each token adds network latency to the request finishing time. Also, different nodes providing compute to the system may have different and dynamic available GPU memories. To efficiently run the model, the layer parameters need to stay in the GPU memory during the complete inference procedure. But given the GPU memory constraints of most available GPUs, for larger models (e.g. llama-2-70b), it is impossible to fit the weights whole model into one node (and the number of layers that can fit within each node is often dynamic). Moreover, different GPU devices have different performance and take different times to run the same layer. Accounting for all these factors to achieve a minimal TPOT becomes the scheduling problem.

2 Background & Related Work

This section covers the basics of LLM serving and model pipelining, with a focus on the network architecture of the widely-used llama-2 model [8] as an example. We explain why we distribute the inference task across multiple GPU nodes and the importance of scheduling these tasks on the proper nodes in a proper order. We also review some related works on distributed and decentralized ML inference.

2.1 Transformer Layers and Auto-regressive Decoding

As a text generation model, the llama-2 model takes the input text and continuously generates text tokens, which represent sequences of characters in a pre-defined dictionary. The model follows an auto-regressive decoding procedure, generating one token in each round of inference until reaching a special termination token. In each round, the previously generated tokens are appended to the input. During the inference procedure, the input is processed through different network layers to generate the probability distribution of the next token. Then, a random sample is taken from the distribution to choose the next token.

The llama-2 model consists of three types of layers: one embedding layer, multiple transformer layers, and one output layer (for the sake of simplicity, we consider the final normalization operation part of the output layer). The number of transformer layers varies depending on the size of the model. In practice, the processing of these transformer layers takes most of the time during the inference procedure. To run inference efficiently, the model parameters for these layers are loaded into GPU memory when a serving system starts.

2.2 Model Parallelism and Partition

Limitation in GPU memory is the major challenge when serving a large model such as the llama-2. For example, the largest llama-2 model with 70 billion parameters needs approximately 130 GB of GPU memory. However, widely-used GPUs, like the A100, only have less than 80 GB of GPU memory, not to mention that consumer-grade GPUs have even less memory. As a result, it is often unfeasible to accommodate the entire model within a single device.

To address these challenges, the established approach is model parallelism, a technique commonly employed in both distributed training [9] [10] [7], and serving [11] [12] phases. For transformer-based LLMs, there exist two primary categories of model parallelism, differentiated by their methods of partitioning the models: intra-layer and inter-layer parallelism.

2.2.1 Intra-layer Parallelism

Transformer models perform high-dimensional matrix multiplications on tensors. Intra-layer parallelism splits these operations and their parameters over multiple devices. This approach allows faster computation on large models that wouldn't fit on a single GPU, making it particularly suitable for models with massive parameter sizes. However, this can lead to large communication overhead due to data exchange between GPUs [12], and this cost can be significantly higher in decentralized settings.

2.2.2 Inter-layer Parallelism

Since a Transformer model is executed layer by layer, the inter-layer parallelism divides the model into smaller stages with a layer as the smallest unit and then distributes them to different devices [7]. The stages are executed in a pipeline way [13], [14] to avoid waiting for the completion of the previous stage and reduce the idle time of devices during multiple rounds of execution. So it can also be referred to as pipeline parallelism. This strategy substantially reduces communication overhead because only a small amount of data needs to be forwarded from the output of one stage to the input of the next stage.

A decentralized serving system can derive significant benefits from this strategy, especially under slow and unreliable network conditions. Thus, when serving models that require GPU memory exceeding the maximum available GPU memory in any node in the network, we select a proper set of nodes and use model pipelining to distribute the layers as shown in Figure 2. Given the difference in the computation resource on each node and the pairwise network latency, the selection of nodes and layer arrangements can have a considerable impact on end-to-end latency.

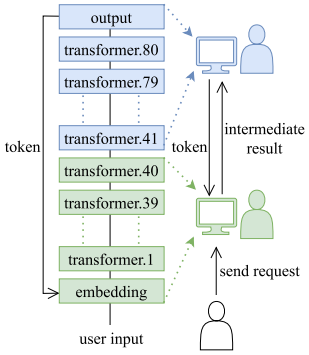

Figure 2: Model Pipelining: inter-layer parallelism is used to split the model over multiple machines.

2.3 Related Work

ML inference using heterogeneous hardware. Li et al. show in [15] that an optimized heterogeneous system for serving ML inference is able to achieve higher throughput with the same power and latency constraints or saving power while achieving the same latency and throughput compared to a homogeneous system. [15] uses an classical optimization approach to allocating a pool of machines (rather than scheduling) to an ML inference task to achieve better performance than a comparable homogeneous system. This work shows that it is not only feasible to use heterogeneous infrastructure to serve ML inference but it may in fact be more cost-efficient to do so. [16] considers the problem of serving LLMs in a centralized, heterogeneous cluster of GPUs. The proposed LLM-PQ system exploits model pipelining and uses an ILP formulation to reduce the inference latency (among other objectives) of the model pipeline. [16] shows that quantization and optimized model partitioning can improve the throughput of LLM serving in a heterogeneous cluster, though this system does not closely consider communication latency, which is a more significant factor in a decentralized setting.

Scheduling model pipelines in homogeneous data centers. It is an ongoing challenge to optimally parallelize model training and inference in centralized computing clusters with homogeneous hardware. [17] uses dynamic programming to partition model pipelines to minimize training latency in a cluster setting. [18] proposes an integer programming algorithm for minimizing inference latency in a system that consists of both CPUs and GPUs but with all of the the GPUs being the same type and having the same memory limitations. In homogeneous settings, the the search space is much simpler than in the heterogeneous setting, as the communication costs, computation latencies, and memory

limits for machines are constant, so the search space consists of many equivalent classes. In contrast, the dynamic memory limits and varying layerwise perfomance for each GPU specification means there are exponentially more possible schedules in the heterogeneous setting.

Decentralized inference of large language models. PETALS is an open-source system for collaborative inference and fine-tuning of large language models [19]. In experimental and production settings, [19] finds that the inference time of a decentralized LLM serving request, is most sensitive to network latency, and it is suggested that clients ping nearby servers to measure latency and run a beam search to find a low latency path for computing pipeline LLMs. Most of the efforts made for reducing inference latency in [19] are regarding quantization of communication buffers and model weights to enable model inference of memory-intensive LLMs on consumer GPUs. [20] uses a similar approach of quantization and beam search for worker assignment to optimize decentralized LLM serving.

Training and inference on heterogenous networks. There has also been prior work on decentralized training of foundation models like open-source LLMs. [21] propose a scheme for of decentralizing the training of foundation models (such as llama-2) to support data parallelism and model-pipeline parallelism. This system treats task scheduling as a balanced graph partition problem, and uses a hybrid genetic algorithm to optimize the vertical pipelining and horizontal communication. [21] considers 5 possible decentralized serving environments: 1) Data center on demand 2) Data center spot instances 3) Multiple data centers 4) Regional geo-distributed 5) World-wide geo-distributed. The experimental results reflect the fact that the geographic distribution of machines in a decentralized system significantly affects the latency of pipelined computing in the system. Another work [16] considers the problem of serving LLMs in a centralized, heterogeneous cluster of GPUs.

3 Scheduling Problem Definition

Given a network of machines and their specifications, we want to schedule a circuit that runs an LLM inference request with minimal latency, as this will maximize the token rate observed by the user. Task scheduling in decentralized serving can be approximated as a latency optimization problem with memory constraints. Here we provide a problem definition considering all major factors. Given N machines in a decentralized network $M_1...M_N$, to run a model with K layers $L_1...L_K$, we assume the following metrics are measured in advance. We also assume these measurements are stable during the inference for a given inference request.

1. Layer computing latency, denoted as $T_l(M_i, L_j)$, reflects the time required for machine M_i to compute layer L_j.
2. Network communication latency, denoted as $T_n(M_i, M_j)$, reflects the time required to send intermediate results from machine M_i to machine M_j. Note that sequentially running two or more layers on the same machine introduces nearly no latency, $T_n(M_i, M_i) = 0$.
3. Remaining GPU memory, denoted as $R_m(M_i)$, reflects the available GPU memory on machine M_i for running the current request.
4. Layer memory requirement, denoted as $R_l(L_i)$, reflects how much GPU memory the layer takes despite the differences in machine GPU types.

Now, we can define the problem more formally:

Problem 1. *Find an assignment sequence $A_1...A_K \in \{M_1...M_N\}$ to minimize*

$$\text{TPOT} = \sum_{i=1}^{K} (T_l(M_{A_i}, L_i) + T_n(M_{A_i}, M_{A_{i+1}}))$$

where A_{K+1} is set to A_1 for notation simplicity, such that

$$\forall M_i \in \{M_1...M_N\}, R_m(M_i) \geq \sum_{j \in \{A_j = M_i\}} R_l(L_j).$$

With variable layer memory requirements and machines with finite memory availability, the problem of finding a valid assignment of the model layers is equivalent to the the bin packing problem, which is an NP-hard combinatorial problem. However, in practice, with reasonably sufficient compute resources in the system, it is feasible to find a valid assignment with relative ease.

4 Algorithm Design

We propose two possible approaches for solving the task scheduling problem. We note that the formal definition of the problem has the form of a constrained optimization problem. If we approach it directly as an optimization problem, we can use integer linear programming to find promising assignments. Alternatively, we consider that that the global optimal assignment often contains a local optimal assignment subsequence. We develop a more efficient heuristic greedy shortest-path based algorithm to find solutions within a limited time window for scheduling.

4.1 Conversion to Shortest Cycle Problem

Consider a simplified version of the scheduling problem where there is no GPU memory constraint, we can generate a directed graph, with vertices denoting the computation of a layer of the LLM on a machine, and the edges between the vertices representing the latency of computing the layer on the assigned machine and communicating to the machine computing the next layer. Then we can apply the shortest cycle algorithm to minimize the TPOT.

Algorithm 1: Graph generation

Result: Graph $G = (V, E)$ representing decentralized LLM serving network
Initialize $V = \{\}$ # vertices;
Initialize $E = \{\}$ # edges;
Set $d(s) = 0$;
Create a priority queue Q with elements $(v, d(v), r(v))$ ordered by the distance;
for $l \in \{1, \ldots, K\}$ **do**
    $next\_l = (l == K)\ ?\ 1 : l + 1$;
    for $i \in \{1, \ldots, N\}$ **do**
        # add vertices;
        $new\_vertex = (l, i)$;
        $V.add(new\_vertex)$;
        # add edges;
        for $j \in \{1, \ldots, N\}$ **do**
            $next\_vertex = (next\_l, j)$;
            $distance = T_n(i, j) + T_l(j, next\_l)$;
            $new\_edge = (new\_vertex, next\_vertex, distance)$;
            $E.add(new\_edge)$;

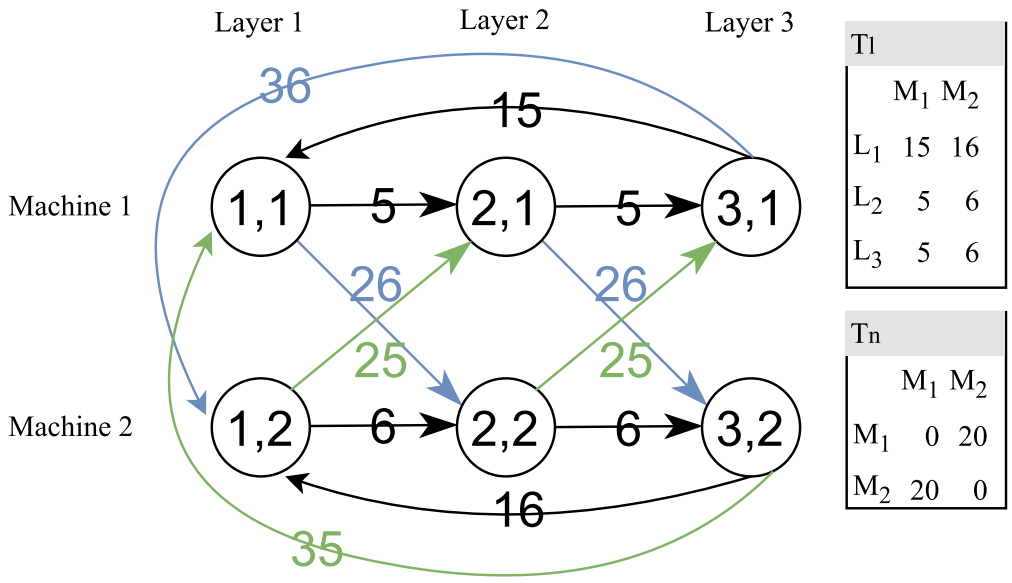

Figure 3: An example of graph generation.

Algorithm 1 describes our graph conversion algorithm. We represent the assignment to run L_l on M_i as a vertex (l, i). The edge from (l, i) to $(\text{next\_}l, j)$ represents moving the task from M_i to M_j. The distance of the edge captures both the communication latency and the computing latency to run $L(\text{next\_}l)$ on M_j. Any cycle in the graph has a length equal to N and maps to an assignment, with the shortest cycle representing the optimal solution.

However, with the memory constraint, well-known shortest-cycle algorithms such as multi-source Dijkstra are not guaranteed to find a valid assignment. The scheduling algorithm would have to ensure that the selected task schedule is a valid assignment of layers to machines that would not exceed the memory limitations of any machine.

4.2 Integer Linear Programming

We can define a set of indicator variables that determine the assignments of layers to machines. Then, the latency of a single walk through the circuit is be defined as a linear combination of the indicator variables. The problem of optimizing the latency reduces to minimizing the latency objective function subject to the constraint that each layer is assigned to exactly one machine which has sufficient capacity for the layer, which can be solved via integer linear programming.

$$\min \sum_{l=1}^{K} \sum_{i=1}^{N} \sum_{j=1}^{N} x_{i,j,l}(T_l(M_i, L_l) + T_n(M_i, M_j))$$

subject to

$$\sum_{i=1}^{N} x_{i,m,l} - \sum_{i=1}^{N} x_{m,i,l+1} = 0 \;\; \forall (m, l) \in [N] \times [K-1] \tag{1}$$

$$\sum_{i=1}^{N} x_{i,m,K} - \sum_{i=1}^{N} x_{m,i,1} = 0 \tag{2}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{N} x_{i,j,l} = 1 \;\; \forall l \in [K] \tag{3}$$

$$\sum_{j=1}^{N} \sum_{l=1}^{K} x_{i,j,l} R_l(L_l) \leq R_m(M_i) \;\; \forall i \in [N] \tag{4}$$

Table 1: Integer programming form for decentralized scheduling

To see why a solution in Table 1 represents a valid assignment in sec_def, consider $x_{i,j,l}$ as a binary value that indicates whether machine M_i is selected to run layer L_l and send the intermediate result to machine M_j. (3) guarantees that each layer is executed by exactly one machine. Then, (1) and (2) guarantee that each machine will have the same number of input and output edges selected, ensuring that the machine receiving input will send the next intermediate result. Last, describes the memory constraint specified in the original problem. By solving the above integer linear programming problem with a commercial optimizer, we can acquire valid and optimized assignments for the original scheduling problem.

4.3 Heuristic Greedy Search

The conversion of the scheduling problem into a shortest cycle problem motivates a graph algorithm-based solution to scheduling the optimal circuit. We can try to use a well-known shortest cycle algorithm like Dijkstra's and make use of additional state variables to track what edges can be traversed after reaching a certain point in the path.

To minimize the communication latency between machines, it is often preferable to run as many consecutive layers as possible on a given machine until no more layers will fit in memory. Because transformer layers are the same, it rarely makes sense to assign non-consecutive transformer layers to the same machine, as this results in unnecessary communication. Following this greedy policy, we present an heuristic algorithm that always use up the memory on one machine before selecting the next. Such a heuristic method actively avoids large network latencies while achieving a more controllable execution time.

This algorithm is not guaranteed to find the optimal solution, as the greedy approach could lead to memory on certain machines being "used up" by earlier layers when it is optimal to instead use it for later layers in the pipeline. However,

in most practical scenarios, the greedy approach to this constrained optimization problem is fairly close to the true optimal solution. The algorithm is efficient at scale, with a time complexity of $O(N^3 K \log N^2 K)$.

Algorithm 2: Greedy Heuristic Shortest Path Algorithm.

Data: Graph $G = (V, E)$
Result: Length of shortest circuit completing single inference task
$ans = \infty$;
for $i \in \{1, \ldots, N\}$ *# as the starting machine* **do**
  for $v \in V$ **do**
    $d[v] = \infty$
  PriorityQueue $Q = \{\}$;
  $current\_v = starting\_v = (1, i)$;
  $current\_l = 1$;
  $current\_d = d[starting\_v] = 0$;
  $current\_mem = R_l(l)$;
  # use up the memory from the starting machine;
  while $current\_mem + R_l(current\_l + 1) <= R_m(i)$ & $current\_l < K$ **do**
    $next\_e = edge(current\_l, i) to (current\_l + 1, i)$;
    $(\_, \_, distance) = next\_e$;
    $current\_mem \mathrel{+}= R_l(current\_l + 1)$;
    $current\_l \mathrel{+}= 1$;
    $current\_v = (current\_l, i)$;
    $d[current\_v] = current\_d \mathrel{+}= distance$
  if $current\_l == K$ **then**
    $ans = min(ans, current\_d + find\_distance((K, i), (1, i)))$;
    continue;
  $used\_machines = i$ # memory from machine i is used up ;
  $Q.add((current\_d, current\_v, used\_machines))$;
  while Q *not empty* **do**
    $(last\_d, last\_v, used\_machines) = Q.pop()$;
    for $e \in E$ *where source is* $current\_v$ **do**
      $(\_, current\_v, distance) = e$; $(current\_l, current\_m) = current\_v$;
      if $current\_m$ *in* $used\_machines$ **then**
        continue;
      $current\_mem = R_l(current\_l)$;
      $current\_d = last\_d + distance$;
      if $current\_mem > R_m(current\_m)$ **then**
        continue;
      while $current\_mem + R_l(current\_l + 1) <= R_m(current\_m)$ & $current\_l < K$ **do**
        $next\_e = edge(current\_l, current\_m)$ to $(current\_l + 1, current\_m)$;
        $(\_, \_, distance) = next\_e$;
        $current\_mem \mathrel{+}= R_l(current\_l + 1)$; $current\_l \mathrel{+}= 1$;
        $current\_v = (current\_l, i)$;
        $current\_d \mathrel{+}= distance$;
      if $current\_l == K$ **then**
        $ans = min(ans, current\_d + find\_distance((K, current\_m), (1, i)))$;
      if $current\_d < d[current\_v]$ **then**
        $d[current\_v] = current\_d$;
        $Q.add((current\_d, current\_v, used\_machines + current\_m))$;

4.3.1 Edge Augmentation

The fact that the problem requires finding the optimal loop in the generated graph means that the algorithm must consider the starting node in the circuit. The way this is done in Algorithm 2 is to greedily compute shortest paths starting from each machine and accounting for the cost of completing the loop, then choosing the best loop.

However, by augmenting the edges in the generated graph, we can convert the shortest cycle problem to a shortest path problem. We can modify the network latencies as follows:

$$T_n(M_i, M_j) \leftarrow T_n(M_i, M_j) + T_n(M_{A_1}, M_j) - T_n(M_{A_1}, M_i)$$

where M_{A_1} is the first machine in the circuit being explored in a step of the algorithm. In other words, we could apply a penalty term to cause the algorithm to select machines closer to the starting machine, which gives consideration to the fact that the circuit must eventually return to the first machine.

In this case, the penalty term being applied is telescoping, so the sum of the the augmented network latencies on the path from M_{A_1} to M_{A_K} would be equal to the sum of the original network latencies of the circuit $M_{A_1}, \ldots, M_{A_K}$. This edge augmentation would enable us to only use one pass of the Dijktra-based heuristic shortest path algorithm if we track the starting nodes of the paths being explored, reducing the time complexity of the algorithm by a factor of N. If the decentralized serving system were to have a large number of machines participating, this edge augmentation strategy could have a considerable effect on the runtime of the task scheduling.

4.4 Practical Considerations

The scheduling algorithms optimize TPOT for a given single request in a stable environment. However, in practice, many factors are not covered by our definition but need to be considered.

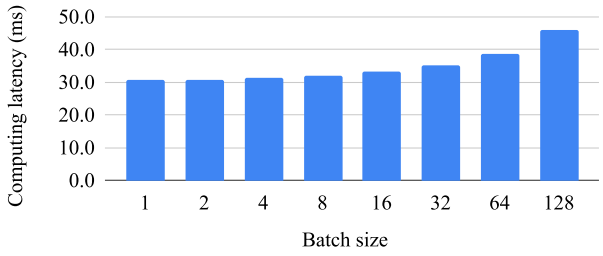

Figure 4: Computing latency for running the first 40 layers of llama-2-70b on a single H100.

Handling multiple requests. In practice, the serving system processes multiple requests to achieve larger throughput. To check the possibility of applying the same scheduling algorithm to optimize the TPOT of multiple requests, we evaluated how batch size influences computing latency. As shown in Figure 4, we find that when the batch size is smaller than a threshold, the computing latency remains close (e.g. 30.6 ms for a batch size of 1 and 33.1 ms for a batch size of 16). Combined with techniques such as iteration-level scheduling [11], the decentralized scheduling algorithm can also be used to optimize the TPOT of multiple requests when the workload is not heavy.

Reserving GPU memory for ephemeral intermediate results. During model inference, the intermediate results and the previous state caches both take up GPU memory. To avoid out-of-memory errors, before estimating the remaining memory value R_m, the machine should reserve enough space for the dynamic content.

In addition, the system design and deployment environment also affect the accuracy of the estimation given by the scheduling algorithm.

5 Evaluation

In this section, we evaluate the feasibility of serving LLMs with consumer-grade GPUs and collect microbenchmarks. We compare the performance of our scheduling algorithm in different decentralized environments. The experiment results show that our heuristic algorithm achieves uniformly better TPOT than all other algorithms with less scheduling time.

5.1 Microbenchmarks

We conducted layerwise performance evaluations for llama-2-70b on three different GPUs: A10g, A100, and RTX 3090. In Table 2, we can see that the transformer layers introduce the highest latency. In an ideal environment with no network latency, we can achieve a TPOT of 175 ms with model pipelining on only RTX 3090s.

	embedding (ms)	transformer x1 (80 in total) (ms)	output (ms)	total (ms)
A10g	0.098	3.748	1.293	301.231
A100	0.074	1.211	0.471	97.425
RTX 3090	0.062	2.177	0.785	175.007

Table 2: Microbenchmark on layerwise computing latency for llama-2-70b on different GPUs.

5.2 Algorithm Comparison

With the above microbenchmarks, we test our candidate algorithms and measure their performance in finding a minimal TPOT. Specifically, to simulate different decentralized environments, we use latency data between 21 regions from an existing cloud platform and add Gaussian noise with a 20% standard deviation to simulate communication latencies. We introduce 4 different testbeds:

- Testbed 1: 21 machines in 21 regions with the following GPU types: 1xA100, 4xRTX3090, 16xA10g.
- Testbed 2: 42 machines in 21 regions with the following GPU types: 2xA100, 8xRTX3090, 32xA10g. This testbed simulates a decentralized environment with more abundant computing resources.
- Testbed 3: 21 machines in 5 regions with the following GPU types: 1xA100, 4xRTX3090, 16xA10g. This testbed simulates the private cloud and small cluster settings.
- Testbed 4: 21 machines in 21 regions with the following GPU types: 1xA100, 4xRTX3090, 16xA10g. Additionally, the percentage of available memory for each machine is drawn from Uniform(0.25, 0.75). This testbed simulates the scenario where the GPU owners are also running other GPU-related programs.

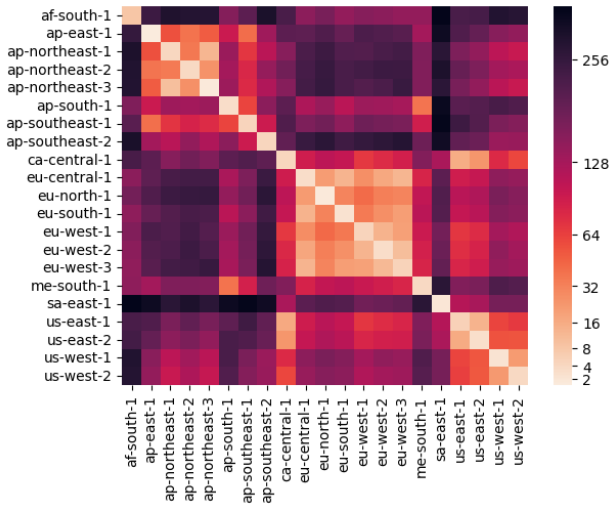

Figure 5: Network latencies between different AWS regions. [22]

For each testbed design, we generate configurations (assigning the locations of the machines and generating latencies) from 16 random seeds, and we average the latency results across the configurations. The algorithms we compared are our heuristic greedy search, random search, and integer linear programming (ILP). The random search works by testing a fixed number (1, 64, 4096) of random permutations of the K machines and greedily assigning as many layers as possible to each machine in order, and choosing the ordering that has the lowest estimated latency. The ILP algorithm

solves the ILP specified in Table 1 using a commercial solver [23], with the algorithm terminating after an allotted time (4, 16 seconds).

	ours	ours (augmented)	ilp4	ilp16	random1	random64	random4096
testbed1	224.12	225.41	277.38	225.44	1,142.98	418.34	266.29
testbed2	172.71	173.31	259.01	177.40	1,166.57	407.67	204.10
testbed3	185.47	185.74	202.69	187.59	1,052.52	418.44	217.09
testbed4	416.03	419.84	873.06	495.93	1,696.72	885.53	608.59

Table 3: Average TPOT (ms) in testbed environments.

Table 3 shows a comparison of the latencies of the cycle scheduled by each algorithm in the four testbed environments we used. We find that the heuristic greedy search algorithm outperforms the random and ILP algorithms on average in all four testbed environments, with the largest difference occurring in the case of partial memory with constrained resources (testbed 4). The partial memory makes it necessary for the pipeline to use many different machines, so communication between machines has a greater effect on the overall latency of the cycle. As a result, the intuition of the shortest-cycle-based search makes it more effective than the ILP, which considers a broader search space.

We also find that the edge-augmented version of our algorithm finds schedules almost as efficient as non-augmented version, but does so with just a fraction of the compute. We see that the difference between our algorithm and its augmented version is most negligible in testbed 3, which simulates small cluster settings. This makes sense, as in small cluster settings, the algorithm should choose closely clustered machines and schedule the circuit on only those machines, and the edge augmentation would likely do so as it increases the tendency to select machines close to the starting machines.

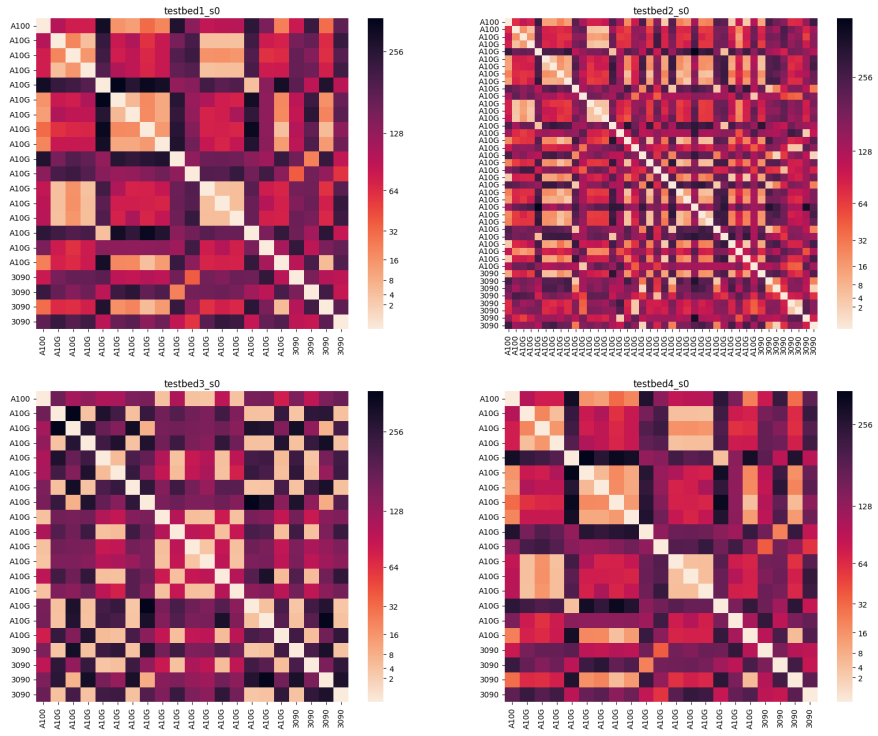

Figure 6: Latencies between machines in example testbed settings.

We note that in scenarios that have more machines or require more complex scheduling, (testbed 2 and testbed 4), the random algorithm's performance relative to the other two proposed algorithms is worse than in smaller and simpler testbeds. In testbed 3, we observe that our algorithm has very similar (though slightly better) performance to ilp4 and ilp16. This is because, in clustered environments, the optimal solution often uses machines in the same region. In this

	ours	ours (augmented)	ilp4	ilp16	random64	random4096
testbed1	0.098	0.005	4.465	13.750	0.009	0.560
testbed2	0.506	0.017	6.153	17.123	0.009	0.579
testbed3	0.093	0.005	4.463	11.270	0.009	0.565
testbed4	0.856	0.034	4.490	16.525	0.010	0.584

Table 4: Average execution time (s) of proposed algorithms in testbed environments.

case, the ILP solver might quickly converge to some local minimum but not find a better solution with more iterations. We also note that in the small cluster setting, the ILP often finds the true optimal solution. This is evident by the fact that in Table 4 we observe the average execution time of the ILP with the 16 second time limit is well below 16 seconds, meaning there were some cases where the solver terminated early after achieving the optimal solution.

We also considered the case where there is no time limit imposed on the scheduling algorithm. We experimented with running the random path and ILP algorithms for 60 seconds, and recording iterative improvements in the circuits searched by these algorithms. Note that the commercial solver takes the time limit as a parameter and this affects the algorithmic behavior of the optimizer, which is why the latencies for the ILP algorithm in Figure 7 do not match the results from Table 3.

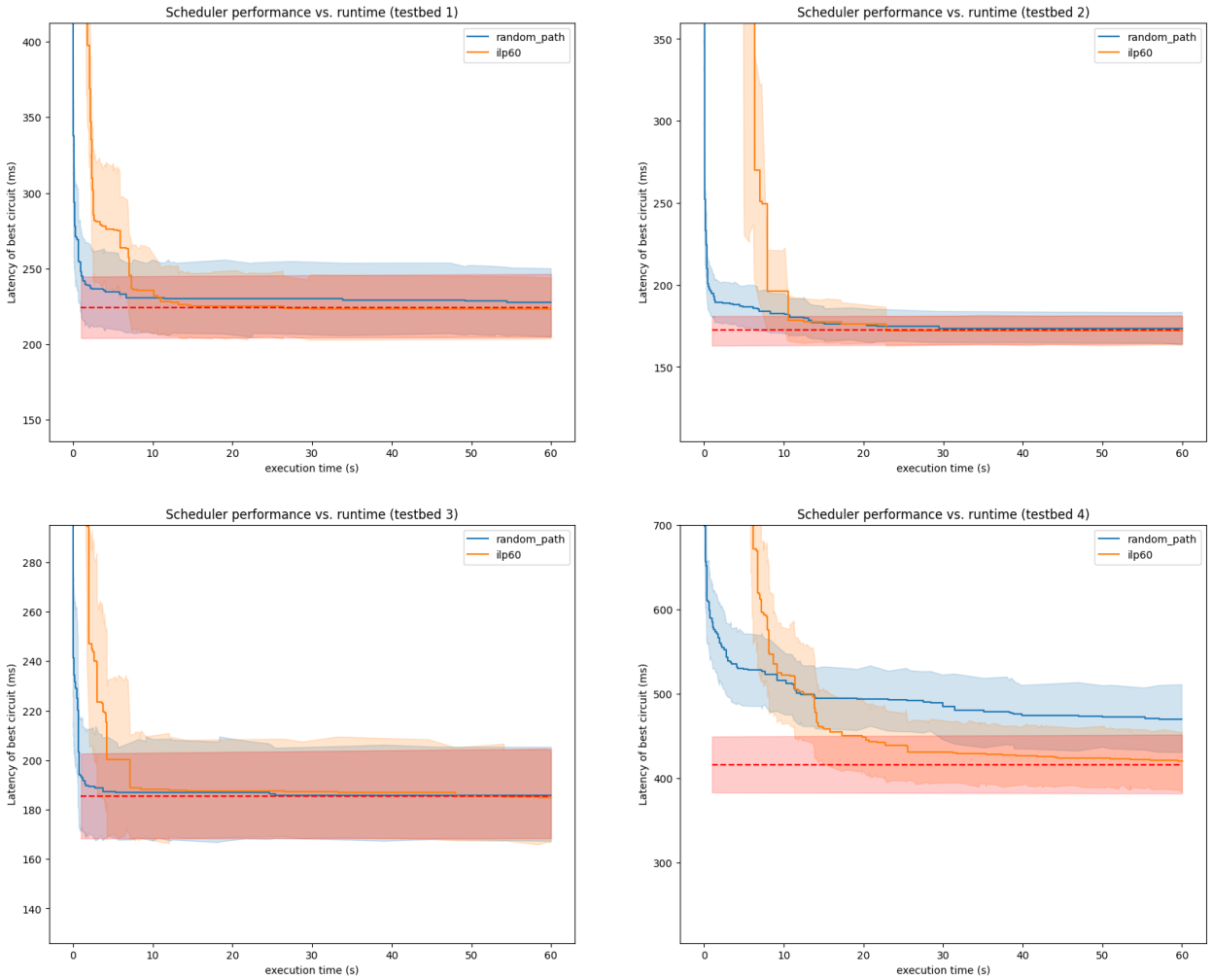

Figure 7: Comparison of scheduling algorithms. Line shows average TPOT across 16 seeds. Colored regions represent 95% confidence intervals.

The plots in Figure 7 show that our heuristic greedy algorithm performs well against the other search algorithms even when the other algorithms are allowed many iterations to find the best circuit. In all 4 testbeds we observed that that the ILP solver converges toward the TPOT achieved by our greedy heuristic algorithm. In testbeds 1-3, the ILP achieves a lower average TPOT than our greedy heuristic algorithm, but this difference is nearly negligible. In testbed 4, the ILP solver converges to close to the average TPOT achieved by Algorithm 2, but has slightly higher average TPOT. In testbeds 1-3, we observe that the random path reduces the TPOT and eventually achieves performance that is almost as good as the heuristic greedy algorithm and the ILP solver. But in testbed 4, the random search converges to a significantly higher average TPOT than our algorithm. This is due to the fact that when the resources are more limited and the system is more heterogeneous (non-homogeneous memory availabilities).

Overall, the testbed experiments underscore the robustness of our scheduling algorithm. Our heuristic algorithm can finish scheduling the testbeds within 1 second, while the ILP method would require a significantly longer runtime. In a practical setting, the scheduler needs to finish within 1 second so the decentralized LLM serving system can respond quickly to user requests and update when the network changes. Our algorithm achieves better or similar results to the ILP method, which uses significantly more computation. We find the heuristic algorithm is efficient and near-optimal in realistic testbed settings.

6 Discussion and Future Work

In this paper, we detail the challenge of task scheduling for decentralized LLM serving and demonstrate the feasibility of using model pipelining to perform LLM inference on a decentralized network of consumer-grade GPUs. We propose an effective scheduling algorithm that can achieve a near-optimal TPOT in realistic settings, with practical execution time for a real-time service.

Our empirical studies show that our scheduling algorithm can effectively reduce the serving TPOT, with significantly better results in more heterogeneous and constrained environments. In a test deployment, we find it possible to achieve an inference speed of more than 3 tokens per second with 8 machines with RTX 3090s, which is close to human typing speed. Having attained these findings for the set of machines (A100, A10G, RTX 3090) and model (llama-2) we studied, an immediate next direction can be extending these results to environments with other types of machines (e.g. RTX 4090, H100) and with microbenchmarking on other models such as the newly released llama-3.

We note that our latency-oriented design may potentially reduce the overall throughput for a decentralized LLM serving system. For inference tasks that are insensitive to latency requirements, a future direction is to design decentralized serving systems that optimize the overall throughput. Such systems may benefit more from the low-cost computing infrastructure, as most centralized serving systems charge API usage based on token counts. Our early experiments show that is possible to achieve close to 50 tokens per second throughput when making multiple parallel requests to the system.

Overall, this work shows that decentralization of open-source LLM serving is a promising approach for enabling LLM hosting. With our scheduling algorithm, it is feasible to efficiently schedule low-latency circuits for LLM inference in decentralized, heterogeneous environments. With the demand for LLM serving and GPU compute continually increasing, decentralized LLM serving could play an important role in providing the resources to handle large computational loads.

References

[1] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

[2] Taylor Orth. What americans think about chatgpt and ai-generated text, 2023.

[3] Bernard Marr. 10 amazing real-world examples of how companies are using chatgpt in 2023, 2023.

[4] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

[5] Shipments of graphics add-in boards increase for third quarter in a row, 2024.

[6] Render Network Foundation. Render network whitepaper, 2017.

[7] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022.

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

[9] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

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

[11] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[12] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[14] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[15] Baolin Li, Vijay Gadepally, Siddharth Samsi, Mark Veillette, and Devesh Tiwari. Serving machine learning inference using heterogeneous hardware. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2021.

[16] Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, and Chuan Wu. Llm-pq: Serving llm on heterogeneous clusters with phase-aware partition and adaptive quantization, 2024.

[17] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[18] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators, 2020.

[19] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models, 2023.

[20] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures, 2022.

[21] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments, 2023.

[22] Aws latency monitoring.

[23] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.