TU WIEN Informatics

Optimizing Distributed LLM Inference for Heterogeneous Workers through Dynamic Graph Partitioning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Gabriel Kitzberger, BSc
Matrikelnummer 12024014

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Alireza Furutanpey

Wien, 15. März 2026

Gabriel Kitzberger	Schahram Dustdar

TU WIEN Informatics

Optimizing Distributed LLM Inference for Heterogeneous Workers through Dynamic Graph Partitioning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Gabriel Kitzberger, BSc
Registration Number 12024014

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Assistance: Dr. Alireza Furutanpey

Vienna, March 15, 2026

Gabriel Kitzberger	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Gabriel Kitzberger, BSc

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, March 15, 2026

Gabriel Kitzberger

Acknowledgements

First, I would like to thank Alireza Furutanpey for supporting me throughout this thesis. Your insights into the scientific process have been invaluable; you always pushed me in the right direction while still letting me come to the correct conclusions myself. I can remember how little I knew about scientific work before starting this thesis. Thank you, I found in you a person that I can strive to be.

I also thank Pantelis Frangoudis for providing feedback during the initial stages and Alexander Knoll for supporting me in setting up the test bed for my evaluation.

I am grateful to Schahram Dustar for providing the framework in which this thesis was built. The DSG team is a wonderful collection of people with the right amount of resources to produce great results. The atmosphere in the team was fantastic and I believe the positive culture is a reflection of your leadership.

Lastly, I am deeply grateful to my family and friends. Thank you for always being there and listening when stress got high or something did not go as planned.

Kurzfassung

Hohe Parameteranzahlen moderner großer Sprachmodelle (Large Language Models, LLMs) beschränken die Inferenz auf ressourcenstarke Einrichtungen mit Server-Hardware. Die verteilte Inferenz mittels heterogener Endverbrauchergeräte bietet einen vielversprechenden Ausweg; bestehende Systeme erfordern jedoch manuelle Konfiguration und Einrichtung, was die Zugänglichkeit auf erfahrene Nutzer begrenzt.

Wir schlagen einen dynamischen Graphpartitionierungsalgorithmus für ONNX-basierte LLMs vor, der die Modellpartitionierung auf eine Variante des *Ordered Partition Problem* reduziert, welches für n Schichten und m Worker in $\mathcal{O}(n^2m)$ lösbar ist. Der Algorithmus optimiert gemeinsam den Speicher der Worker, die Ausführungsgeschwindigkeit, die Netzwerkbedingungen sowie bereits heruntergeladene Modellgewichte, um die End-to-End-Inferenzlatenz zu minimieren. Wir integrieren diesen Algorithmus in einen verteilten Inferenzserver, implementiert in über 5.500 Zeilen Rust. Der Server partitioniert das Modell zur Laufzeit dynamisch neu, wenn Worker dem System beitreten oder es verlassen. Durch die Nutzung des Browsers als Verteilungsmechanismus wird eine konfigurationsfreie Teilnahme ermöglicht, wodurch die manuelle Einrichtung der Worker entfällt.

Unsere Experimente zeigen, dass unser Kostenmodell einen mittleren absoluten prozentualen Fehler (MAPE) von 8,4% insgesamt und 4,4% für große Modelle erzielt. Die dynamische Partitionierung übertrifft durchgehend die statische gleichmäßige Schichtenaufteilung, und eine Ablationsstudie der Metriken bestätigt, dass jede Worker-Metrik einen bedeutsamen Beitrag zur Zuweisungsqualität leistet. Wir demonstrieren, dass das System unerwartete Verbindungsabbrüche überbrückt und ein Modell mit insgesamt 60 GB Gewichten auf heterogene Geräte verteilt.

Abstract

High parameter counts of frontier large language models (LLMs) restrict inference to well-resourced institutions with server-grade hardware. Distributed inference using heterogeneous consumer devices offers a path forward; however, existing systems require manual configuration and setup, limiting accessibility to expert users.

We propose a dynamic worker-aware graph partitioning algorithm for ONNX-based LLMs that reduces model partitioning to a variant of the *Ordered Partition Problem*, solvable in $\mathcal{O}(n^2m)$ for n layers and m workers. The algorithm jointly optimizes worker memory, execution speed, network conditions, and cached model weights to minimize end-to-end inference latency. We integrate this algorithm into a distributed inference server implemented in over 5,500 lines of Rust. The server dynamically repartitions the model at runtime in response to workers joining or leaving the system. Using the browser as a distribution mechanism enables zero-setup participation, eliminating the need for manual worker configuration.

Empirical evaluation shows that our cost model achieves a mean absolute percentage error (MAPE) of 8.4% overall and 4.4% for large models. Dynamic partitioning consistently outperforms static equal-layer splitting, and an ablation study confirms that each worker metric contributes meaningfully to assignment quality. We demonstrate the system recovering from unexpected worker disconnects and distributing a model totaling 60 GB of weights across heterogeneous devices.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Questions . . . 2
1.3 Contributions . . . 2
1.4 Thesis Organization . . . 3

2 Background **5**
2.1 Large Language Models (LLMs) . . . 5
2.2 Distributed Deep Learning . . . 11
2.3 ONNX & Execution Engine . . . 12
2.4 Graph Theory & Partitioning . . . 14

3 Related Work **17**
3.1 Distributed LLM Inference . . . 17
3.2 Web-Based LLM Inference . . . 18
3.3 Model Partitioning . . . 18
3.4 Research Gap . . . 18

4 System Design **21**
4.1 Design Methodology . . . 21
4.2 System Overview . . . 23
4.3 Metric Collection . . . 26
4.4 Server State Changes . . . 28

5 Partitioning **33**
5.1 Problem Formulation . . . 33
5.2 Cost Modeling . . . 35
5.3 Complexity Analysis . . . 38

5.4 The Dynamic Programming Algorithm 40
5.5 Worker Ordering Optimization 41

6 Implementation Details **45**
6.1 Internal Representations 45
6.2 Graph Processing Pipeline 46
6.3 Partitioning Algorithm 48

7 Evaluation **49**
7.1 Experimental Setup 49
7.2 Baseline 52
7.3 RQ3: Cost Model Validation 56
7.4 RQ4: Static vs. Dynamic Partitioning 58
7.5 RQ5: Metric Ablation Study 61
7.6 Case Studies 63

8 Discussion **65**
8.1 Research Questions 65
8.2 Limitations 67
8.3 Practical Viability 69
8.4 Future Work 69

9 Conclusion **71**

Overview of Generative AI Tools Used **73**

List of Figures **75**

List of Tables **77**

List of Algorithms **79**

Bibliography **81**

CHAPTER 1

Introduction

1.1 Motivation

The computational demands of frontier models have grown dramatically in recent years. The Llama family was launched in 2023 with the largest model having 65 billion parameters. Just two years later, we have open-weight models such as DeepSeek-V3 [DLM+25] with 671 billion parameters and Kimi-K2 [TBB+25] with 1 trillion parameters [Epo25]. Although smaller models can run on consumer hardware, large frontier models require hundreds of gigabytes of VRAM, restricting access to well-resourced institutions. Furthermore, inference engines such as vLLM [vll25] are optimized for server-grade hardware and throughput, rather than end-to-end latency on consumer devices.

Distributed inference offers a path forward by pooling resources of multiple devices to run models that fit on no single device. vLLM supports multi-device scaling, but only across homogeneous hardware. Systems such as Petals [BRC+23] and EdgeShard [ZSC+25] enable inference on heterogeneous consumer hardware and edge devices. However, they require users to manually configure workers and download multi-gigabyte model weights; a barrier that excludes non-expert users.

A solution to this usability problem is the web browser. Zero-setup participation, where users can contribute just by opening a website, would dramatically lower the entrance barrier for distributed inference. Realizing this vision introduces technical challenges that are absent in native distributed systems. Browsers are sandboxed environments that prohibit high-speed memory connections such as RDMA, restrict GPU access to the WebGPU API rather than native CUDA, and impose security constraints that inference engines like vLLM are not designed for. Beyond the browser, heterogeneous participants bring different capabilities: varying memory budgets, execution speeds, network conditions, and cached model weights. No existing system adapts model partitioning at runtime to heterogeneous workers in a web environment.

This thesis addresses this gap. We propose a dynamic graph partitioning algorithm for ONNX-based LLMs that considers memory, execution speed, network conditions, and cached model weights. Workers may run on diverse technology stacks, including the browser via WebGPU or Python via CUDA, and may join or leave the system at any time. The algorithm responds to such events by repartitioning the model, enabling the distributed inference server to maintain high throughput under dynamic conditions.

1.2 Research Questions

This thesis investigates the following research questions:

- **LLM Partitioning**: What methods and tools exist for partitioning ONNX-based LLMs for distributed inference?
- **Algorithmic Complexity**: What computational complexity trade-offs arise in the proposed worker-aware dynamic partitioning algorithm, and under what conditions is it tractable?
- **Cost Model Validation**: How closely does the end-to-end inference latency estimated by the worker-aware dynamic partitioning algorithm match real-world measurements?
- **Static vs. Dynamic Partitioning**: How does dynamic worker-aware partitioning compare to static equal-layer splitting in terms of inference throughput?
- **Metrics**: How does removing individual worker metrics from the partitioning algorithm affect inference throughput?

1.3 Contributions

This thesis makes the following contributions:

- **Dynamic partitioning algorithm**: A worker-aware graph partitioning algorithm for ONNX models that reduces downtime during initialization and minimizes end-to-end inference latency during runtime.
- **Distributed inference system**: A distributed LLM inference server that supports heterogeneous workers including Python (CUDA) and browser (WebGPU) environments. Using the web for distribution, we facilitate zero-setup participation for web-based workers.
- **Empirical evaluation**: We establish baselines that characterize the behavior of the system under varying model sizes, worker counts, prompt lengths, and device types. We validate the algorithm's cost model against real-world measurements, quantify the throughput gains of dynamic over static partitioning, and assess the contribution

of individual worker metrics. Finally, demonstrate dynamic repartitioning and deploy a model totaling 60 GB of weights across devices.

1.4 Thesis Organization

In Chapter 2, we provide the required background. In Chapter 3, we list related work. In Chapters 4, 5, and 6, we explore the system's design, the partitioning algorithm, and implementation details. Chapters 7 and 8 focus on our results. Finally, Chapter 9 concludes our work.

CHAPTER 2

Background

In this chapter, we provide background on dynamic graph partitioning for LLMs. We start with an overview of an LLM's architecture and how to run inference. Next, we explore parallelism in deep learning and graph partitioning. Lastly, we give an introduction to the ONNX ecosystem.

2.1 Large Language Models (LLMs)

2.1.1 LLM Architecture

The architecture of LLMs is typically categorized as one of: *Encoder-Decoder*, *Causal Decoder*, *Prefix Decoder*, or *MoE* [ZZL+23, NKQ+25, RMF+24]. In this section, we focus on the *Causal Decoder*, as it is the basis for most generative LLMs. The *MoE* architecture is widely used for newer models; however, since it works similarly to the *Causal Decoder* during inference with ONNX Runtime, we will not explain it in detail. For example, `gpt-oss-20B`, an *MoE* model, accepts the same inputs as `Qwen3-0.6B`, a *Causal Decoder*.

High-Level Data Flow

Tokenization Tokenization is the process of converting text into discrete tokens. This is typically done using the Hugging Face `tokenizers` library [MP23]. For example:

Tokens are a text representation that LLMs understand!

is tokenized as

$$[29300, 525, 264, 1467, 13042, 429, 444, 10994, 82, 3535, 0]$$

Using a vocabulary, which maps subsequences of the input text to discrete tokens, the text is converted to a tensor of integers called `input_ids`. LLM inputs typically include two additional tensors: the `attention_mask` is a tensor containing zeros and ones. For generative LLMs it typically only contains ones, meaning that all past tokens should be used as context to compute the next token during a forward pass. Secondly, the `position_ids` are used for positional encoding. For some models, they can be omitted from the input, as they can be computed directly from the other inputs.

Embedding The first part of an LLM model is the input or embedding layer. It computes word embeddings for each token from the 3 input tensors. These embeddings are learned as higher-dimensional vector representations of individual tokens. They capture the basic meaning of individual tokens; for example, *apple* and *tree* might be closer together in this vector space than *apple* and *window*.

Transformer Layers The embeddings are passed through a sequence of transformer layers. The output of each previous layer is used as the next layer's input, resulting in an entirely linear data flow through the model. During this process, new tokens are enriched with the context of previous tokens through the attention mechanism.

Output Head The output of the last transformer layer is converted to a tensor called *logits*. Each element of this tensor has the same length as the token vocabulary. We can use the logits in a decoding process to find the output token of the model.

Decoding Strategies There are many LLM decoding strategies [SYC+24], four common ones being [Dec25]:

- **Greedy Search** always chooses the most likely next token by using `ArgMax` on the logits.
- **Beam Search** [MY17] is a strategy in which we generate our logits, choose the top-k next tokens, and re-run inference on all of them. From our resulting k^2 outputs, we choose the top-k, continuing the generation process. This allows us to explore different sequences in parallel while keeping the search space small.
- **Top-k Sampling** [AMY18] uses the logits as a probability distribution and randomly samples the next token from the top-k tokens.
- In **Nucleus Sampling** [HBD+20] we choose a value $p \in [0, 1]$. The top-k are chosen so that the sum of their probabilities exceeds p. We then randomly sample the output token from the top-k tokens.

Global Dependencies Although the data flow through the model is entirely linear, there are tensors that are broadcast to every transformer layer. Qwen3 [YLY+25], which we are using for our evaluation, includes two such tensors. We need to consider these connections when partitioning the model and distributing tensors during inference.

The Transformer Architecture

The core of an LLM is the transformer [VSP+17]. Through GPT-2 [RWC+19] and GPT-3 [BMR+20], the transformer emerged as a great choice for text generation. The transformer typically includes 3 main components: positional encoding, normalization, and attention. Since all elements in a transformers sequence can be computed in parallel, the positional embeddings inject information about the order of input elements. Normalization is required for stable model training.

The Attention Mechanism The input of an attention block is constructed as QKV-triples of tensors. By computing the dot product between the query Q and the key K, the model captures the relationships between pairs of elements in the input sequence. For example, in the sentence *The cat sat on the mat because it was tired.* the dot product of *cat* and *it* might be particularly high; the model learned that *it* refers to *cat*. We multiply this importance by the value V to get a new tensor as the output. Formally:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_K}})V$$

The required computation in a transformer scales quadratically with the sequence length. For every new token, we need to calculate its relation to all previous tokens in the sequence. Improving the transformer and its attention mechanism has seen extensive coverage in the literature [DFE+22, KLZ+23, ALd+23, DLF+24, DLM+25].

Qwen3 models use Grouped-Query Attention (GQA) [ALd+23], Rotary Positional Embeddings (RoPE) [SAL+24], and RMSNorm [ZS19].

Tensor Shapes

To partition a model, we need to consider the model's data flow and tensor shapes. In this section, we investigate the shapes of Qwen3-0.6B in detail. Since LLMs are structurally similar, this analysis also applies to other models. Figure 2.1 shows a single layer of the model.

Most models have the following inputs:

- `input_ids: [batch_size, sequence_length]`
- `attention_mask: [batch_size, total_sequence_length]`
- `position_ids: [batch_size, sequence_length]`

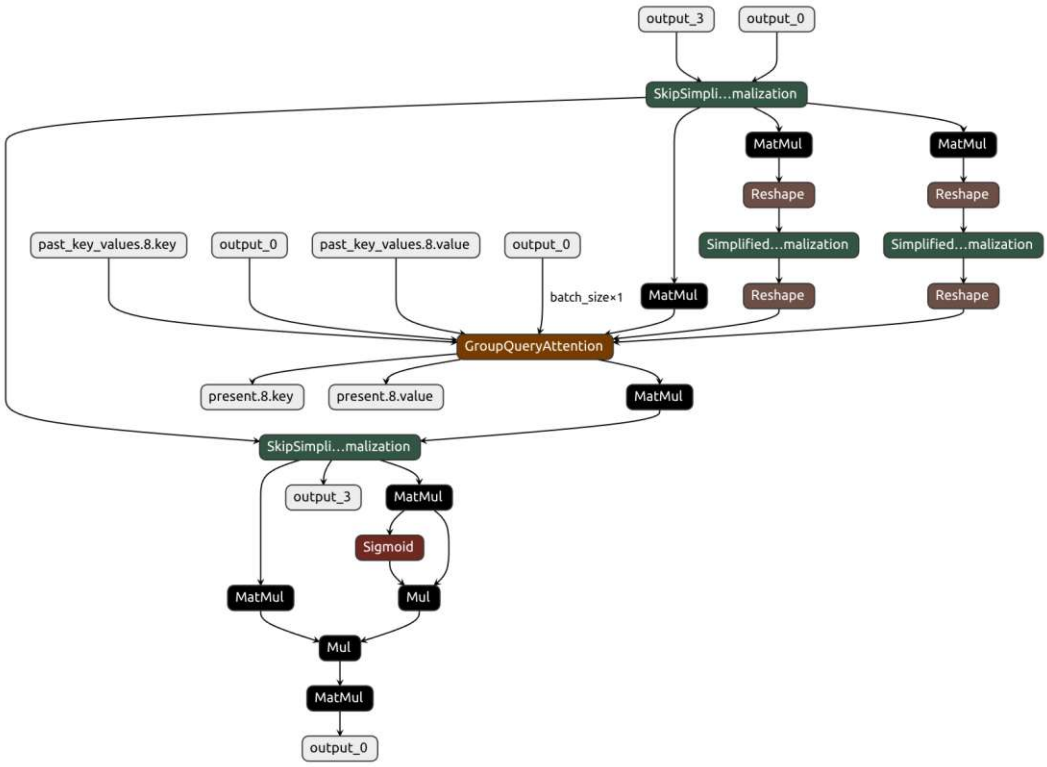


Figure 2.1: A single layer of Qwen3-0.6B stored as an ONNX file and visualized using Netron.

and outputs:

- `logits: [batch_size,sequence_length,151936]`

Machine learning models are typically designed to compute multiple outputs in a single forward pass. The batch size refers to the number of parallel forward passes. For large-scale inference with many requests, batching requests is a complex problem. Using a batch size of 1 is simpler and enough for systems that do not focus on scalability.

There are three different variants of the sequence length. This distinction becomes important when using a KV cache during inference (see Section 2.1.2). The *total sequence length* is the count of all tokens processed so far. The past sequence length is the number of tokens that have already been passed through the model. The *sequence length* is their difference, i.e., the number of tokens that will be passed through the model in the next forward pass. In autoregressive token generation, the sequence length is typically 1, since we generate one token at a time.

For Qwen3-0.6B, transformer layers have the following inputs:

- `[batch_size,sequence_length,1024]`
- `[batch_size,sequence_length,1024]`
- `[batch_size,8,past_sequence_length,128]` (KV Cache)
- `[batch_size,8,past_sequence_length,128]` (KV Cache)
- `[batch_size]` (Global Dependency)
- `int` (Global Dependency)

and outputs:

- `[batch_size,sequence_length,1024]`
- `[batch_size,sequence_length,1024]`
- `[batch_size,8,total_sequence_length,128]` (KV Cache)
- `[batch_size,8,total_sequence_length,128]` (KV Cache)

As shown in Figure 2.1, it is not possible to cut the transformer layer into roughly equal subgraphs and get a k-connectivity of less than 2. Regarding tensor size, the boundaries between layers minimize the size of tensors that need to be transferred. Since the input and output of every `SkipSimplifiedLayerNormalization` node has the same shape, there are 4 natural cuts per layer, as each layer has two such nodes. We will use this information when discussing graph partitioning in Section 2.4 and Chapter 5.

2.1.2 LLM Inference

LLM inference consists of prefill and decoding. During prefill, the prompt is passed through the model. During decoding, one token at a time is generated.

Phase 1: Prefill (The Prompt)

The prefill phase is compute-bound. In most cases, the sequence length during prefill is the same as the number of input tokens. The past sequence length is 0. No KV cache is present, and intermediate tensors are large. This results in significant computation during the model's forward pass, as all prompt tokens must be processed.

Phase 2: Decoding (Token Generation)

After the prefill phase, we can use the KV cache. During decoding, the sequence length is typically 1; only a single token is passed through the model, and information about past tokens is stored in the KV cache. This results in lower computational intensity; the phase is memory bound since the gradually growing KV cache needs to be stored in memory.

State Management: The KV Cache

The key-value cache (KV cache) is an optimization of the attention mechanism. Since new tokens only attend to previous tokens, we can cache the keys and values of previous tokens. For a total sequence length of n, this reduces the computations from $O(n^2)$ to $O(n)$ for each forward pass. As a trade-off, we need to store two tensors for each layer with shape `[batch_size, num_kv_heads, past_sequence_length, head_dim]`. For L layers, this results in $O(Ln)$ memory. For each subsequent forward pass, we provide the previously generated token and the KV cache to compute the next token.

2.1.3 Computational Constraints

Memory Constraints

Model	Full Precision	Quantized
Qwen3-0.6B	1.5 GB	500 MB
Qwen3-8B	15.2 GB	5.2 GB
Qwen3-32B	61 GB	20 GB
Qwen3-235B-A22B	438 GB	142 GB
DeepSeek-V3 (671B)	1.27 TB	404 GB

Table 2.1: The model size in bytes for different models. The model size for full precision was taken from Hugging Face [Hug25a], for the quantized model size we used `Q4_K_M` GGUF from the Ollama model library [Oll].

Table 2.1 relates the number of model parameters to their model size in bytes. The memory required to run an LLM roughly equals the sum of the model weights, activation memory, and KV cache. The model's weights are loaded into memory once and remain static in size. Memory for activation is ephemeral and small; it can be ignored for calculations. The KV cache can grow substantially when many parallel inference requests with long sequences are issued.

Optimizing the KV cache has been extensively covered in the literature. FlashAttention [DFE+22] is an IO-aware algorithm that optimizes GPU memory operations. PagedAttention [KLZ+23] is an attention algorithm that stores keys and values in non-contiguous paged memory, reducing the memory footprint during LLM inference. Optimizations such as grouped-query attention (GQA) [ALd+23], multi-head latent attention (MLA) [DLF+24], or DeepSeek Sparse Attention (DSA) [DLM+25] optimize the KV cache for inference with high sequence lengths.

Quantization

During model training, a model's weights must have high precision to perform gradient descent; they must be updated in small steps during the backward pass. For inference, this is not necessary. Quantization [DLBZ22, FAHA23, LTT+24] is a technique in which

the model's weights are transformed to lower precision, saving space and optimizing inference.

2.2 Distributed Deep Learning

Parallelism in deep learning is typically categorized into data parallelism, pipeline parallelism, or tensor parallelism [ZZL+23, ZLZ+22]. These techniques are typically used during model training; however, they can be adapted for inference.

2.2.1 Data Parallelism

Data parallelism is used for small model's, processing large amounts of data. The model is replicated across multiple devices and the data is split. This increases throughput since all models can run inference in parallel. For LLM inference, this approach is useful for scenarios focusing on high throughput and scalability.

2.2.2 Tensor Parallelism

In tensor parallelism, also referred to as operator or intra-operator parallelism, individual operators are computed on different devices in parallel. For example, a matrix multiplication $Y = XA$ might be computed on two devices as $Y = [XA_1, XA_2]$, where the matrix A is divided by column [ZZL+23]. This approach is useful for large models with small data and high speed interconnects. Operators can be computed in parallel, speeding up inference. High-speed interconnects are required, as the outputs of the operators need to be synchronized before computing the next operator.

Megatron-LM [SPP+20] uses 512 NVIDIA V100 GPUs to speed up LLM training using tensor parallelism. Alpa [ZLZ+22], Orca [YJK+22], and PipeDream [NHP+19] combine parallelism approaches, including tensor parallelism, for LLM training and inference.

2.2.3 Pipeline Parallelism

In pipeline parallelism, also known as inter-operator parallelism, operators are grouped and placed on different devices. During a forward pass, all operators on a device are computed, and the intermediate tensors are passed to the next device. For transformer models, consecutive layers are typically placed onto the same GPU [ZZL+23]. This type of parallelism is useful for large models with slower GPU interconnects, as communication between devices is only required when transferring intermediate tensors. As a downside, pipeline parallelism results in more idle time and slower end-to-end inference, as computation is not parallelized and devices have to wait for inputs.

GPipe [HCB+19] pioneered pipeline parallelism for deep neural networks, with others following shortly [NHP+19, YWN+25].

2.3 ONNX & Execution Engine

In this thesis, we use ONNX [onn25] as the model format. During runtime, the server partitions the ONNX model into sub-graphs, which are distributed to workers. In Chapters 5 and 6, we explore partitioning and ONNX graph manipulation in detail.

2.3.1 Computational Graph Representation

ONNX represents machine learning models as directed acyclic graphs (DAGs). The graph along with the tensor information and metadata is encoded using Protocol Buffers [pro25]. Figure 2.1 shows the graph of a single layer of Qwen3-0.6B visualized using Netron.

The inputs and outputs to the DAG are identified using strings and require shape information. The nodes in the graph are computational operators, represented as a topologically sorted list. The edges are data, tensors in most cases, that serve as inputs and outputs to operators. An edge represents data flowing from a named output to a named input. A single output can be used for multiple inputs. In ONNX, edges are stored as strings within nodes.

Constant tensors, such as model weights, are stored as initializers. Tensor data can be stored either in the ONNX file or as external data. For external data, the file location is specified relative to the ONNX file, and the data is encoded as raw bytes.

For an LLM, the inputs to the ONNX model are `input_ids`, `attention_mask`, `position_ids`, and the KV cache. The outputs are `logits` and the updated KV cache.

2.3.2 Tensors and Data Types

In ONNX, tensors are stored as a combination of data, shape, and datatype. The shape is represented as a list of integers; the datatype is stored as an enum. Shape information also supports string names for dynamic shapes; for example, the KV cache has the shape `float32[batch_size,sequence_length,1024]` for Qwen3-0.6B.

2.3.3 Operators

The operators, such as `MatMul` or `ArgMax`, are stored in nodes as strings. The execution engine parses the nodes and computes the operator once all node inputs are present. Since the node list is topologically sorted, the simplest way to obtain the final output is to compute the nodes in order. Inference engines typically parallelize execution when possible.

ONNX allows specifying custom operators with `FunctionProto`. The functions defined this way can be made up of more primitive ONNX operators. For example, a new attention operator could be composed in such a way. The execution engine can choose

between custom implementations for these operators if available, or fall back to the subgraph defined in `FunctionProto`.

2.3.4 ONNX Runtime (ORT)

ONNX Runtime [ONN18] is an execution engine for ONNX models. The engine loads the ONNX file and computes the model's output given a map of named inputs. One of its biggest advantages is flexibility; ONNX Runtime can be used on most major platforms, including Windows, Linux, Android, and the Web. It supports arbitrary inputs and outputs and offers an extensive selection of operators.

Execution Providers (EPs)

ONNX Runtime can be built for execution in different environments. For example, `CUDAExecutionProvider` uses the GPU with CUDA to execute the model, and `CPUExecutionProvider` executes using the CPU.

ONNX Runtime Web is a variant built for execution in web environments. The engine is compiled into WebAssembly and can be used from JavaScript as a WebAssembly module. Using WebGPU, it can execute ONNX models on the GPU.

2.3.5 Converting LLMs to ONNX

Most open-weight models are available on Hugging Face [Hug25a]. Since few LLMs are available as ONNX files, we need to convert them. The Python library `onnx` has an export function; however, it is impractical to use and produces unoptimized ONNX models. Optimum [hug25b] can be used to produce ONNX models using a simple command. Similarly, Olive [mic25] can produce optimized ONNX models directly from LLMs stored on Hugging Face.

2.3.6 Partitioning

To partition an ONNX model, we have to find node outputs along which to split the model. For an LLM, a natural choice is to split between two transformer layers. After identifying all edges/outputs that separate the two subgraphs, two new ONNX models have to be created. The first model now contains additional graph outputs: the inputs required for the other model. The second model contains new graph inputs: the outputs of the previous model. Additionally, both graphs need to be cleaned, which involves removing nodes of the other graph and ensuring that graph inputs and outputs match the model's nodes.

The Python library `onnx` supports partitioning through the use of the `extract_model` function. This function creates a new model with the specified inputs and outputs from an existing ONNX model. Microsoft Olive supports static equal-layer partitioning of LLMs using the `–num_split` flag.

2.4 Graph Theory & Partitioning

In this section, we establish the theoretical foundations for partitioning neural networks. We define the graph representation of deep learning models and explore general graph partitioning and its computational complexity. Finally, we demonstrate how the specific architecture of LLMs allows us to reduce the problem to the *Ordered Partition Problem*, enabling simpler and more efficient algorithms.

2.4.1 Graph Representation of Neural Networks

Machine learning models are typically represented as dataflow graphs. The nodes are computational operators such as `MatMul` or `SoftMax`; the edges represent input and output tensors. Formally, we can define models as computational graphs $G = (V, E)$, where the vertices V are computational operators, and the edges E represent their input and output data. Operators are functions that take multiple inputs and produce an output [GBC16, ABC+16, ZLZ+22].

To ensure valid execution schedules and enable automatic differentiation, these graphs are defined as Directed Acyclic Graphs (DAGs) [ABC+16, PGM+19]. Even architectures with loops, such as Recurrent Neural Networks (RNNs), are computationally represented as unrolled DAGs during training to permit backpropagation through time [Wer90, GBC16]. This acyclic constraint ensures that there is a topological ordering of operations, allowing gradients to propagate via the chain rule.

Topological Sort

> A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. [CLRS09]

Since the execution of machine learning models during inference can generally be represented as a DAG, a topological ordering exists [Ski20]. Finding this topological sort allows the execution engine to compute operators in order while ensuring that all inputs are present. For example, the ONNX model format represents operators as a topologically sorted list of nodes.

When partitioning a machine learning model, using a topologically sorted list of operators allows the model to be cut at any point and obtaining two valid subgraphs. However, this does not guarantee minimal communication, as a single cut in the sequence might sever multiple data dependencies (e.g., residual connections) that jump over the cut. Optimal partitioning requires finding points in this order where the cut minimizes the total data volume of the severed edges. This problem is non-trivial as the topological sort of DAGs is not unique.

When exporting LLMs to ONNX, the transformer layers are similar in structure and are separated in the topological sort. Since the residual streams between the transformer

layers are natural cuts (see Section 2.1.1), splitting an ONNX model at layer boundaries always results in valid subgraphs with small intermediate tensors.

2.4.2 The Graph Partitioning Problem (GPP)

The primary motivation for partitioning large language models (LLMs) is hardware constraints: model parameters, intermediate activations, and state often exceed the memory capacity of a single accelerator. Consequently, the model must be split across multiple devices. Formally, this can be modeled as the *Graph Partitioning Problem* (GPP):

> Given a number $k \in \mathbb{N}_{>1}$ and an undirected graph $G = (V, E)$ with *non-negative* edge weights, $\omega : E \to \mathbb{R}_{>0}$, the *graph partitioning problem* (GPP) asks for a partition Π of V with *blocks* of nodes $\Pi = (V_1, ..., V_k)$:
>
> 1. $V_1 \cup \cdots \cup V_k = V$
> 2. $V_i \cap Vj = \emptyset \quad \forall i \neq j$
>
> A balance constraint demands that all blocks have about equal weights. [BMS+16]

The balanced graph partitioning problem is known to be NP-complete [GJ90, Ski20]. Due to this complexity, exact solutions are computationally intractable for the massive graphs typical in deep learning.

Standard approaches rely on heuristics. *Spectral partitioning* utilizes the eigenvectors of the graph's Laplacian matrix to find approximations [BMS+16]. *Iterative refinement* algorithms, such as the Kernighan-Lin method, swap vertices between partitions to improve a local objective [KL70]. *Multilevel algorithms*, exemplified by METIS [KK98], recursively coarsen the graph to a manageable size, partition the coarsened graph, and then uncoarsen while refining the cuts.

However, applying general GPP solvers to LLMs presents challenges. First, general solvers optimize for balanced subgraphs, whereas LLM inference is often constrained by hard memory limits that cannot be violated. Second, these heuristics typically work on undirected graphs and do not respect the data flow of the model, potentially introducing inefficient communication patterns.

Beyond general partitioning, other graph-theoretic formulations offer partial solutions but face practical limitations in this context. The *max-flow min-cut theorem* allows for finding a partition that minimizes edge cut weights (communication volume) in polynomial time using network flow algorithms [Ski20]. However, this formulation minimizes communication while not adhering to memory constraints. Similarly, automatic detection of repeated structures, such as Transformer layers, could be formulated as the *Subgraph Isomorphism* problem, which is NP-complete [GJ90]. In practice, such

computational complexity is unnecessary for LLMs, where layers can be identified reliably through simpler methods, such as using naming conventions within the ONNX definition.

2.4.3 Reduction to the Ordered Partition Problem

Although the GPP is NP-complete, the specific architecture of LLMs allows for a simplification. By coarsening the computational graph $G = (V, E)$ so that a single node v_i represents an entire layer, the complex DAG reduces to a *linear chain graph*:

$$v_1 \to v_2 \to \cdots \to v_n$$

where n is the number of layers. This coarsening is natural because the boundaries between layers represent cuts that minimize communication.

This reduces the GPP to the *Ordered Partition Problem* (also known as the *Integer Partition without Rearrangement* problem).

> **Input:** An ordered arrangement S of non-negative numbers $s_1, \ldots, s_n$ and an integer m.
> **Output:** A partition of S into m or fewer consecutive ranges to minimize the maximum sum over all ranges, without reordering the elements. [Ski20]

In the context of LLMs, the numbers s_i represent the cost (e.g., memory usage or compute time) of layer i. Unlike GPP, this problem can be solved exactly using dynamic programming in $O(n^2 m)$ time [Ski20].

To accommodate the specific constraints of deep learning, we can define a custom cost function $C(range)$. For example, if a range of layers exceeds the available memory of a device, we assign $C = \infty$, effectively pruning invalid partitions. This allows us to find the optimal layer assignment for a fixed worker ordering, rendering approximate heuristics like METIS unnecessary for this subproblem.

It is important to note that this reduction assumes that the order of workers is fixed. As we will discuss in Chapter 5, if the worker ordering is flexible (heterogeneous devices), the problem re-introduces complexity.

CHAPTER 3

Related Work

In this chapter, we detail the latest work on model partitioning and LLM inference. We start with distributed LLM inference and browser-based inference. We explore partitioning strategies for distributed model training and inference. Lastly, we show that the literature on distributed inference using the browser is limited, necessitating a partitioning algorithm that accounts for web-specific challenges.

3.1 Distributed LLM Inference

Many centralized inference engines support distributed inference. vLLM [vll25] supports pipeline and tensor parallelism; SGLang [sgl25] supports all parallelism modes. However, these engines typically require homogeneous hardware, focusing on throughput and scalability.

Petals [BRC+23] runs LLMs on heterogeneous geo-distributed hardware. They allow for reliable inference even if some devices randomly disconnect. Newly connected workers choose a subset of the LLM to serve. Users sending inference requests use a shortest-path algorithm to find a subset of workers for inference.

EdgeShard [ZSC+25] runs collaborative inference using edge devices and cloud servers. They use dynamic programming to find model partitions that optimize inference latency and throughput, taking into account heterogeneous hardware and network conditions.

Others cover aspects such as deploying LLMs on mobile devices [ZSL+24], running MoE models on edge devices [YGW+25], or providing an open-source framework for inference on heterogeneous consumer hardware [exo25].

3.2 Web-Based LLM Inference

Regarding web-based LLM inference, the literature is scarce. WebLLM [RQZ+24] compiles LLMs for WebGPU using TVM [CMJ+18], retaining up to 80% native performance. WeInfer [CMSL25] improves WebLLM through more efficient buffer-reuse strategies and switching to an asynchronous approach that does not block GPU execution. Wllama [Ngu25] is an experimental project that provides WebAssembly bindings for llama.cpp [ggm25]. MediaPipe LLM by Google [LLM] offers an LLM Inference API in the browser for Gemma models. Transformers.js [Tra] offers a functional equivalent of the Python transformers library for the web. They use ONNX Runtime Web to run LLMs in the browser.

3.3 Model Partitioning

Partitioning of machine learning models has been extensively studied. Foundational works such as GPipe [HCB+19] established the principles of pipeline parallelism, while Megatron-LM [SPP+20] popularized tensor parallelism for training massive transformer models.

3.3.1 Dynamic Partitioning

Early approaches often assume static hardware environments; recent literature focuses on dynamic partitioning to handle heterogeneity and runtime fluctuations.

Deng et al. [DLZ+24] propose a method that adjusts partitions in response to resource variations. They derive models for execution duration and redistribution costs, jointly optimizing to minimize overall training time. DynPipe [YWN+25] uses a random forest model to predict the impact of environmental changes, such as network jitter or task interference, and adjusts the partition plan to restore balance.

In the context of IoT and edge computing, Zhou et al. [ZSB+19] present a runtime acceleration framework that dynamically selects the optimal degree of parallelism based on resource availability and network conditions.

3.3.2 Algorithmic Approaches

To solve the partitioning problem, various algorithmic strategies have been employed. The most common approach is dynamic programming, used by systems such as Alpa [ZLZ+22], PipeDream [NHP+19] and others [ZSB+19, TTHT21, ZSC+25, HIZ+22, LZG+21]. Others formulate the problem as Linear Programming (LP) to handle complex constraints [ZCZ+21, HL22].

3.4 Research Gap

Table 3.1 compares existing systems in key dimensions for accessible distributed inference.

System	Distributed	Heterogeneous	Zero-Setup
vLLM	✓	✗	✗
Petals	✓	✓	✗
WebLLM	✗	✗	✓
Ours	✓	✓	✓

Table 3.1: Comparison of LLM inference systems

Existing systems make trade-offs that limit accessibility. vLLM and SGLang require homogeneous hardware and a complex setup. Petals supports heterogeneous distribution, but requires manual model downloads and installation of Python dependencies. WebLLM runs in the browser, but cannot distribute computation.

Our system combines web-based execution with distributed inference for heterogeneous workers. This enables deployment scenarios that are inaccessible with existing systems: users can pool their resources to run models too large for any single device, without requiring software installation or homogeneous hardware.

This requires a specialized dynamic partitioning algorithm that adapts to worker conditions, considers the cost of downloading model weights, and jointly optimizes for end-to-end inference latency and redistribution cost.

CHAPTER 4

System Design

In this chapter, we focus on system design and its relation to model partitioning. We start with our design philosophy where we explain the goals and non-goals for dynamic model partitioning. Next, we provide an overview of each component of the distributed LLM inference system: the topology, the server, the workers, and the communication protocol. We explain the metrics that the server collects during runtime. Lastly, we show the server's dynamic repartitioning loop.

4.1 Design Methodology

4.1.1 Goals

Zero-Setup LLM Inference

The core idea of our system is to facilitate zero-setup distributed LLM inference, using the web for distribution and ONNX Runtime Web as the execution engine. As such, all the technologies involved must support this environment.

This decision represents a trade-off between user-facing complexity and inference speed. Web technologies such as WebSockets and WebGPU, although highly optimized, incur efficiency penalties. For example, a native peer-to-peer solution could use Remote Direct Memory Access (RDMA), reducing serialization and network overhead. Similarly, browser-based WebGPU execution is inherently less efficient than native execution.

Worker Heterogeneity

The system and partitioning algorithm can handle a wide variety of workers. Workers can be heterogeneous in memory, bandwidth, latency, and execution speed. The inference server collects information about workers, and the dynamic partitioning algorithm finds

assignments. Essentially, as long as a worker implements the communication protocol correctly, the server should be able to use it effectively.

Fault Tolerance

We assume that the workers are stable and connect for longer periods of time. However, given a web-based environment with varying network conditions, the server must handle arbitrary disconnects. We optimize model partitioning to find the most efficient configuration in the long term; however, if a worker arbitrarily disconnects, the main goal is to resume token generation as quickly as possible.

Minimizing End-to-End Latency

We design our cost function and algorithm to minimize end-to-end latency for single-inference requests.

Interpretable Partitioning Decisions

To ensure that the partitioning algorithm delivers near-optimal solutions, it should be interpretable so that we can verify the correctness of the assignments. We design our partitioning algorithm to return the Time Per Output Token (TPOT). We use metrics during inference to verify that the algorithm's estimates match real-world inference requests. In Chapter 5 we discuss the algorithm in more detail, and in Chapter 7 we empirically validate the cost model.

4.1.2 Non-Goals

Efficiency & Scalability

Distributed inference is inherently less efficient than centralized inference due to communication overhead. Similarly, web-based approaches using consumer hardware are not as efficient as native execution on server-hardware. Our goal is to demonstrate that web-based distributed LLM inference is viable; we do not aim to compete with highly optimized inference solutions such as Ollama or vLLM. Similarly, we do not consider optimizations such as batch processing or data parallelism.

Security

We assume that all components of the system are trusted; otherwise, we would need to verify the workers' outputs, which would require additional computation. This would either require more connected workers or slow down inference, neither of which is a trade-off we believe necessary at this point.

Privacy

Privacy-preserving approaches are not within the scope of this thesis. They would likely require a different architecture and setup. Using pipeline parallelism with layer-wise partitioning, a worker requires full information about its inputs and outputs. Therefore, any worker can discover all generated tokens by copying and running the rest of the model.

Partitioning of Arbitrary DAGs

LLMs have many transformer layers; for example, Qwen3-0.6B has 28 layers, Gemma 2 2B has 26 layers, and Phi-4 has 32 layers. Larger models typically have more transformer layers. We assume that partitioning at transformer layers is coarse enough for most workers, even with limited memory. We do not aim to partition arbitrary DAGs, even if this yields better assignments in edge cases.

4.2 System Overview

4.2.1 Topology

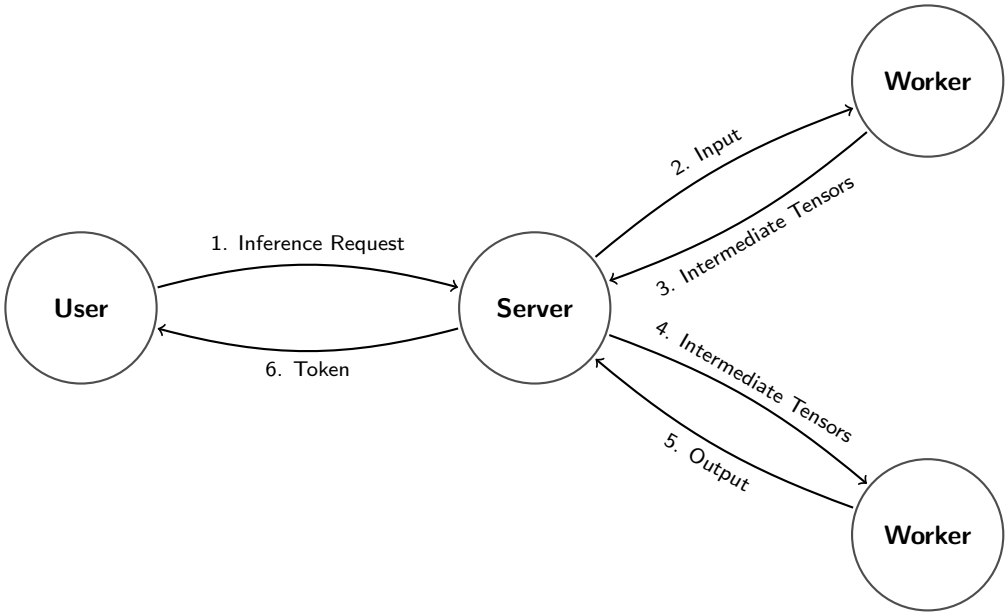


Figure 4.1: The system's topology with the server as the central orchestrator. Arrows represent the data flow during inference.

As shown in Figure 4.1, the system adopts a star topology, with the inference server acting as the central orchestrator. In a web-based environment, a central server is a prerequisite to deliver the web application.

We reuse this existing infrastructure to route all data, eliminating the need for direct worker-to-worker communication. Although this centralization simplifies coordination, establishing a single source of truth for metric collection and state management, it introduces network latency compared to peer-to-peer architectures. We defer the exploration of peer-to-peer networks to future work (see Section 8.4).

Distributed State Management

To optimize for end-to-end latency, we implement a distributed, non-replicated KV cache. As detailed in Section 2.1.2, the KV cache is critical for efficient decoding. In our design, each worker maintains its partition of the KV cache in local memory without synchronization to the central server. This approach eliminates the bandwidth cost of a shared cache.

The trade-off is that workers become stateful. A worker disconnecting results in the loss of its cache partition, requiring re-calculation of the context window (i.e. rerunning the prefill phase using all tokens as the input). However, this design decision aligns with our core philosophy: we optimize for efficient inference while assuming stable workers, accepting a higher recovery cost for the infrequent event of worker failure.

4.2.2 Server

The inference server is written in the Rust programming language. Its main goal is to facilitate efficient token generation. As such, it has the following responsibilities:

- **User API**: The server provides the API for users to connect and send inference requests. The server returns a stream of tokens.
- **Worker API**: Workers connect using a separate API using both HTTP and WebSockets for communication. HTTP is used to download large amounts of data such as model weights and to testing the worker's bandwidth. WebSockets are used for bi-directional communication; for example, to transfer control signals or tensors during inference. In Section 4.2.4, we explain the communication protocol in detail.
- **Inference Orchestration**: Upon receiving a prompt, the server checks if enough workers are connected so that the model is fully loaded on a set of distributed workers. If this is true, the server passes the tensors from worker to worker through the chain until the final output is returned. It sends the generated token to the user and continues generation until the EOS token is reached. If a worker disconnects or network conditions degrade, the server pauses active generation, triggers a repartitioning event, and resumes generation once the new assignment is ready.
- **Metric Collection**: To inform scheduling decisions, the server aggregates metrics. This includes hardware capabilities and runtime statistics from connected workers, as well as static information about individual layers.

- **Partitioning**: Using the collected metrics, the server executes the partitioning algorithm to distribute the ONNX model graph across the available worker pool. We ensure that the memory requirements are within the range of each participating device.

4.2.3 Workers

We use two types of worker for our evaluation: a web-based worker running in the browser using WebGPU and ONNX Runtime Web, and a native worker implemented in Python using CUDA and ONNX Runtime. Workers have the following responsibilities:

- **Computation**: A worker's main responsibility is to load a partition of the ONNX graph and execute it using the input provided by the server. There is no specification about how the worker executes the ONNX model provided by the server; in theory, the worker can parse the model to a more efficient format such as TensorRT or custom kernels for LLM inference. We leave the implementation of such a worker for future work 8.4.
- **Metrics**: The worker sends metrics to the server to be used for partitioning decisions.
- **State Management**: Given an ONNX model, workers manage their own inference sessions and KV cache.

4.2.4 Communication

Communication with users is done with a WebSocket per inference request. Upon WebSocket upgrade, the user sends the prompt. The server sends back decoded tokens as text messages.

Communication with the worker follows a request-response pattern initiated by the server. The worker never initiates communication. The following message pairs exist:

- **Init-Connect**: When a new worker connects, the server initializes communication by sending shared state. The worker responds with initial metrics such as available memory, bandwidth, and already downloaded layers.
- **DownloadLayers-LayerDownload**: The server might instruct a worker to download some model layers to prepare for a potential future assignment. For each downloaded layer, the worker responds with a **LayerDownload** message.
- **PrepareModel-PrepareDone**: After model partitioning, the server instructs all workers who are part of the assignment to prepare their partition. Upon completion, the workers acknowledge.

- **CommitModel-CommitDone**: After all workers have finished preparing, the server commits the new model partition.
- **Computation-ComputationResult**: During token generation, the server sends all input tensors except the KV cache to workers, who respond with their partition's output.

Additionally, the server can send the messages **InvalidateCache** to finish an inference request, and **ReleaseSession** to free all resources from the worker until a new model is assigned.

The server and the workers communicate using binary WebSocket messages. The messages are encoded using Protocol Buffers to facilitate efficient transfer of floating point tensors.

4.3 Metric Collection

To partition the ONNX model, the server uses metrics collected during runtime. In this section, we list all metrics that the server uses for partitioning decisions and how they are collected. In Chapter 5, we show how the metrics are used.

4.3.1 Layer Metrics

At startup, the server loads and parses the ONNX models so that it can be used for distributed LLM inference. Parsing involves moving initializers (see Section 2.3) to external data using the initializer's SHA256 hash as the filename. This reduces the memory footprint of the model while the server operates on the graph. The server then partitions the model at the layer boundaries, resulting in $\#\text{layers} + 2$ subgraphs as the embedding and output layers are considered their own subgraphs. The server collects metrics about each layer:

- **Weights in Bytes**: When splitting the LLM into layers, the server sums up the layer's weights in bytes.
- **Required Memory**: This metrics is an estimate of how much memory is required to run the layer. We compute it as $1.5 * \texttt{weights\_in\_bytes}$. A more sophisticated metric collection system could determine the amount of bytes required during runtime; for example, by changing estimates depending on the number of parallel inference requests or the sequence lengths. We leave this for future work 8.4.
- **Input/Output Size**: For each layer, we compute the size of all inputs and outputs in bytes. Combined with bandwidth and latency, we can estimate the network time during distributed inference.
- **Computational Cost**: We estimate the computational cost C_i of each layer by running them on the server. Given our assumption that the model is too large

to run on a single device, we create a local ONNX Runtime session for each layer separately, run each layer 5 times, and take the mean time. Next, we calculate the proportional difficulty of each layer. For LLMs, all layers except the embedding and output layers have a similar proportional difficulty. We multiply the percentage by an arbitrary magic number of 10_000_000. We will explain how this metric is used in the next section.

4.3.2 Worker Metrics

The collection of metrics for workers is split into two phases: First, when a new worker connects, the server and worker jointly determine the workers capabilities so that initial partitioning decisions can be made. Second, while a worker is connected, the metrics are updated continuously.

The following metrics are collected about workers:

- **Available Memory**: After a worker connects, it sends the amount of memory it has available for inference sessions as part of the `Connect` message. The worker can choose any arbitrary amount; however, it must be able to run a model of this size. For example, in our evaluation, we set the memory to lower values to force distribution to more workers. A worker could also decide to use both the GPU and CPU for inference; it would combine VRAM and RAM. This metric is determined when a worker connects and cannot be changed unless the worker reconnects.

- **Computational Capabilities**: The execution speed of a worker consists of the *Session Overhead* and the *Computational Speed*. During the worker's initialization phase, the server assigns two layer ranges $[i, j)$ and $[i, k)$ with $i < j < k$ and $\lceil \frac{k-i}{2} \rceil = j - i$ (the first range is half of the second) to the worker. For each range, we run 7 computations and measure the time taken. Since the time varies after creating a new ONNX Runtime session, we discard the first 4 computations and compute the mean of the last 3 computations. Let $t_{i,j}$ and $t_{i,k}$ denote the mean time to execute the layer ranges. We determine the session overhead SO_w and the computational speed CS_w of the worker as follows:

$$
\begin{aligned}
\Delta_t &= t_{i,k} - t_{i,j} \\
t_{\text{per\_layer}} &= \frac{\Delta_t}{k - j} \\
SO_w &= t_{i,j} - (j - i) \cdot t_{\text{per\_layer}} \\
CS_w &= \frac{\sum_{l=i}^{k} C_l}{t_{i,k} - SO_w}
\end{aligned}
$$

The session overhead is an estimate of the session's static cost; for example, copying input and output tensors to and from the GPU occurs only at the start and end of the session and is independent of the number of assigned layers. The unit of session overhead is µs. The computational speed gives an estimate of the worker's speed in executing layers independent of session overhead. The unit is $\frac{ops}{\mu s}$, where *ops* is a dimensionless unit of computational operations. To compute the time it takes for the worker w to execute layers $[x, y)$, we can compute $\hat{t}_{x,y} = SO_w + \frac{\sum_{z=x}^{y} C_z}{CS_w}$. Since the layer cost C_z also has unit *ops*, the terms cancel, leaving us with the time.

The session overhead is determined once during worker initialization. The computational speed is continuously updated. To remove outliers, we store values in a ring list of 15 elements, taking the median when using the metric.

- **Network Latency**: Network latency is determined using heatbeats via WebSocket pings. We periodically ping each worker and store the result in a 7 element ring list. We found that latency increases significantly during token generation; therefore, we only included measurements taken while the worker is idle. Note that since we are using pings, the latency includes the time to the worker and back.
- **Bandwidth**: The bandwidth is determined by downloading random data from the server using HTTP for 5 seconds. The worker sends the bandwidth to the server via the `Connect` message.
- **Downloaded Layers**: Workers cache downloaded layers in case they rejoin later. As part of the `Connect` message, the worker signals which layers it has already downloaded in a previous session. The server updates the metrics when workers send the `LayerDownload` message.

4.3.3 Server State

The server has two state variables relevant for the partitioning algorithm:

- **Blocking Download**: When there is no fully loaded model, this variable is set to true. It indicates that the server has to wait for all assigned workers to finish downloading layers until they are ready to create an inference session.
- **Time of last Redistribution**: The server keeps track of when it repartitions the model. As we will discuss in Chapter 5, this variable is used by the server to move towards optimal assignments over time.

4.4 Server State Changes

The inference server has two components involved in changing state: The repartitioning loop is responsible for finding layer-to-worker assignments. It contains the dynamic partitioning algorithm. The state graph variables hold assignments.

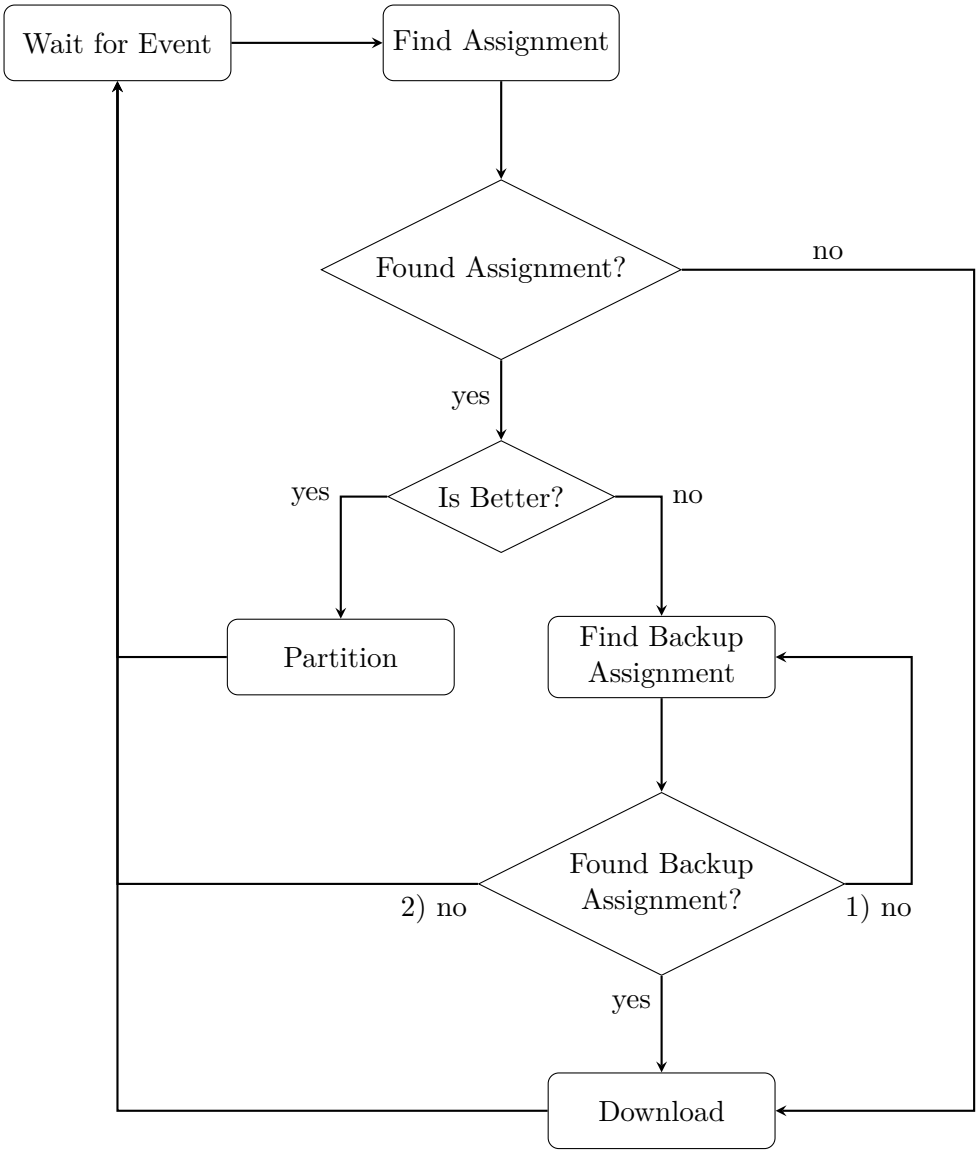


Figure 4.2: The dynamic repartitioning loop. Events such as a worker connecting, disconnecting or periodic timeouts trigger the find assignment function.

4.4.1 Dynamic Repartitioning

As shown in Figure 4.2, repartitioning the ONNX model is a multi-step process. This process is triggered by 3 events:

- **Worker Connect**: When a worker connects, the server checks if it can find a better assignment using this worker.

- **Worker Disconnect**: When a worker that was previously used in an assignment disconnects, the server attempts to find a new assignment without this worker.
- **Periodic Checks**: The server periodically checks if a better assignment can be found. A better assignment can be found if the performance of a worker degrades or if a previous assignment was chosen due to its initialization cost with a faster assignment existing.

First, the server runs the partitioning algorithm to find an assignment that loads the full model. If no assignment exists, the server has to wait for more workers to connect. While waiting, it instructs workers to download weights for a partial assignment to reduce initialization time once enough workers are connected. If the assignment meets the acceptance criteria (see Section 5.2.4), the server prepares the new assignment. Otherwise, the server tries to find a *shadow assignment*. Shadow assignments are used to reduce the initialization time when workers disconnect. We first try to find an assignment that does not reuse any worker. If no such assignment exists, we try again, but punish reusing a worker in our cost function. If a shadow assignment is found, we download its layers.

4.4.2 State Transitions

The server progresses through four states as it loads and serves a model:

- **Down**: There is no valid assignment that covers the full model. The server cannot generate tokens.
- **Preparing**: An assignment has been found and workers are downloading the weights of their partition.
- **Committing**: All weights are downloaded. The workers are creating the inference session.
- **Up**: The model is fully loaded and the server can generate tokens.

The server maintains two assignment variables: `active_graph` and `inactive_graph`.

Forward transitions follow the path: *Down* → *Preparing* → *Committing* → *Up*. The server can hot-swap to a better assignment by preparing an assignment using the `inactive_graph` while keeping inference running on the `active_graph`.

Backward transitions occur on failure. If a worker disconnects, the affected variable is invalidated and the server falls back to whichever state reflects what remains; *Preparing* if `inactive_graph` contains an assignment, or *Down* if there is no viable assignment.

Self-loops on *Preparing* and *Committing* occur when a better worker connects and the server repartitions while already repartitioning.

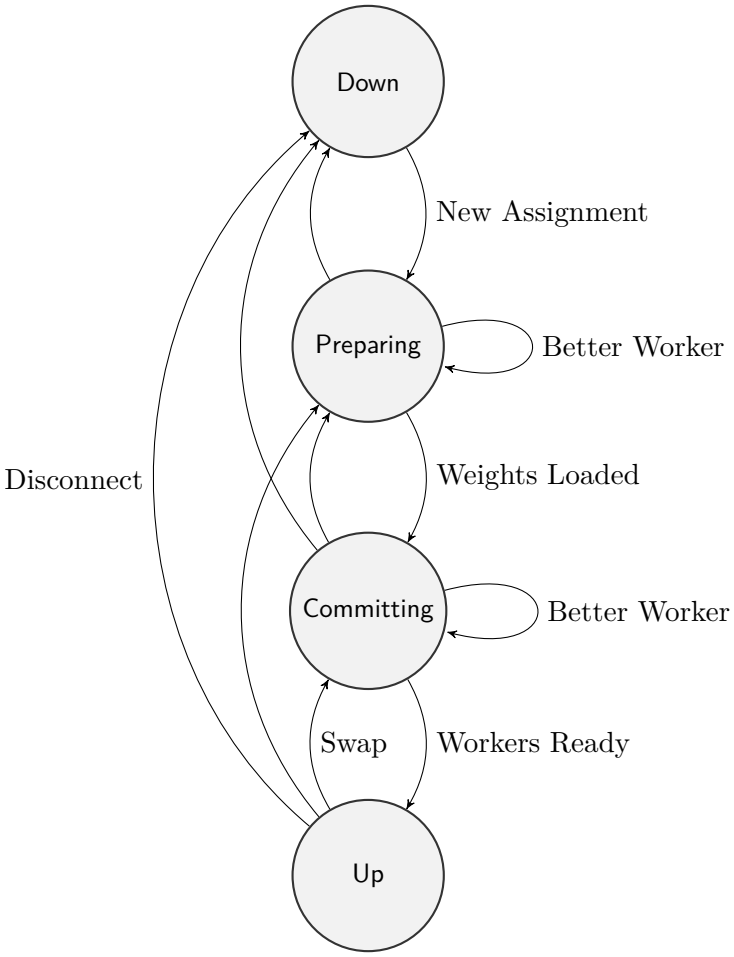


Figure 4.3: Server state transitions.

CHAPTER 5

Partitioning

In this Chapter, we discuss the model partitioning algorithm. We start with a formal definition of the problem and cost functions. We explore different variations of the problem, analyze their computational complexity, and solve the problem using dynamic programming. Lastly, we show how we use simulated annealing to search through the space of worker orderings.

5.1 Problem Formulation

As discussed in Section 2.4, we can reduce our partitioning problem to a variation of the *Ordered Partition Problem* by coarsening the computational graph to the layer level. In this section, we provide a mathematical formulation of this problem adapted to LLM partitioning and web-based distributed inference.

There are two primary distinctions between our problem and the standard *Ordered Partition Problem* as discussed in [Ski20]:

- **Sequential Execution**: In the standard ordered partition problem, the objective is to balance independent computation across workers (minimizing the maximum runtime across all partitions). However, LLM inference via pipeline parallelism requires sequential execution. Therefore, our objective is to minimize the *sum* of the runtimes (including communication) across the active pipeline stages.
- **Worker Heterogeneity & Constraints**: The standard problem assumes homogeneous workers. Our environment features heterogeneous workers with varying computational speeds, network bandwidths, and latencies. Additionally, we introduce a hard constraint: the memory required to load a partition cannot exceed the worker's available memory.

5.1.1 Notation & Inputs

Let $\mathcal{L} = \{l_0, l_1, \ldots, l_{n-1}\}$ be the ordered sequence of n layers comprising the LLM. Each layer l_i has the following metrics and units:

- C_i: Computational Cost (ops).
- W_i: Weights (bytes).
- R_i: Required memory (bytes).
- I_i: Input tensor size (bytes).
- O_i: Output tensor size (bytes).

Let $\mathcal{W}$ be the set of m connected workers. Each worker $w \in \mathcal{W}$ has the metrics:

- M_w: Available memory (bytes).
- SO_w: Session overhead (µs).
- CS_w: Computational speed ($\frac{\text{ops}}{\mu\text{s}}$).
- B_w: Network bandwidth ($\frac{\text{bytes}}{\mu\text{s}}$).
- P_w: Network link latency (µs).
- D_w: Cached model layers (set of integers).

5.1.2 Assignment Representation

An *assignment* A is defined as an ordered sequence of triples:

$$A = \langle (w_0, b_0, b_1), (w_1, b_1, b_2), \ldots, (w_{p-1}, b_{p-1}, b_p) \rangle$$

where $b_0 < b_1 < b_2 < \cdots < b_p$ are partition boundaries and worker w_k is responsible for layers $[b_k, b_{k+1})$.

5.1.3 Constraints

Any valid assignment A must satisfy the following constraints:

1. **Memory Capacity**: The total memory required by the assigned layers must not exceed the worker's available memory.

$$\forall (w_k, b_k, b_{k+1}) \in A : \sum_{b_k \leq x < b_{k+1}} R_x \leq M_{w_k}$$

2. **Distinctness**: Each worker appears at most once in A:

$$\forall w_k, w_l \in A, k \neq l : w_k \neq w_l$$

3. **Coverage**: The entire model is covered by A:

$$b_0 = 0, b_p = n$$

A partial assignment relaxes the coverage constraint $b_p = n$ and covers all layers up to some layer q:

$$b_0 = 0, b_p = q \quad 0 < q < n$$

As discussed in Section 4.4.1, partial assignments are only used if there are not enough workers to load the model.

5.1.4 Objective Function

Our primary objective is to find an assignment A^* that minimizes the end-to-end latency of a single forward pass (decoding phase).

The execution cost $C_{\text{exec}}(w_k, b_k, b_{k+1})$ is the time it takes the worker w_k to execute layers $[b_k, b_{k+1})$ including the network time.

An optimal assignment A^* minimizes the sum of these costs across a pipeline of workers $w_0 \dots w_{p-1}$:

$$A^* = \arg\min_A \sum_{(w_k, b_k, b_{k+1}) \in A} C_{\text{exec}}(w_k, b_k, b_{k+1})$$

In the following section, we expand on this cost model, include an initialization penalty (C_{init}), and define the acceptance criteria.

5.2 Cost Modeling

5.2.1 Execution Cost

The execution cost $C_{\text{exec}}(w_k, i, j)$ estimates the time in microseconds for the worker w_k to compute layers $[i, j)$. It comprises a compute term and a network transfer term:

$$C_{\text{exec}}(w_k, i, j) = \begin{cases} \infty & \text{if } \sum_{i \leq x < j} R_x > M_{w_k} \\ \underbrace{SO_{w_k} + \frac{\sum_{i \leq x < j} C_x}{CS_{w_k}}}_{\text{Compute Time}} + \underbrace{\delta + P_{w_k} + \frac{I_i + O_{j-1}}{B_{w_k}}}_{\text{Transfer Time}} & \text{otherwise} \end{cases}$$

where $\delta = 500\mu s$ is a constant that accounts for overhead per worker, such as serialization at network boundaries or server processing. Networking is comprised of latency P_{w_k} and

data transferred. Since P_{w_k} is measured using a WebSocket ping, we only need to include it once to account for both the request and the response. For data transfer, we take the input size of the first layer and the output size of the last layer using the bandwidth to compute the time taken. If the memory required to hold the layers $[i, j)$ exceeds M_{w_k}, the cost is ∞, making the assignment infeasible.

This formulation results in slight inaccuracies; for example, δ does not depend on the compute speed of the worker, and the bandwidth is only measured as the download speed from the worker's perspective rather than upload and download. In Section 7.6, we empirically evaluate the cost model.

5.2.2 Initialization Cost

The initialization cost estimates the overhead of deploying an assignment to a worker: downloading any missing weights and creating a new inference session. The cost in microseconds is:

$$\hat{C}_{\text{init}}(w_k, i, j) = \sigma + \frac{\sum_{i \le x < j} W_x \; - \; W^{w_k}_{(i,j)}}{B_{w_k}}$$
$$W^{w_k}_{(i,j)} = \sum_{x \in [i,j) \cap D_{w_k}} W_x$$

where $\sigma = 1\text{s}$ is a static overhead for session initialization, and $W^{w_k}_{(i,j)}$ is the sum of weights already cached on w_k from a prior session.

Directly adding $\hat{C}_{\text{init}}$ to C_{exec} would cause the initialization cost to dominate since it might require downloading multiple gigabytes of data, whereas a forward pass takes milliseconds. We apply a power transform to reduce the importance of $\hat{C}_{\text{init}}$. After this transform, the cost is no longer in microseconds and should be interpreted as a weighted penalty rather than a time estimate.

The transform depends on whether the server is in state *Up*. If the server is *Up*, the initialization cost is less important, since the new assignment can be prepared in the background while serving requests. In case no model is loaded, the goal is to reach the state *Up* quickly, so initialization cost is important.

$$C_{\text{init}}(w_k, i, j) = \begin{cases} \hat{C}^{\gamma}_{\text{init}} \cdot \rho(t) & \text{if in state } \textit{Up} \\ \hat{C}^{\gamma_{\text{block}}}_{\text{init}} & \text{otherwise} \end{cases}$$

with $\gamma_{\text{block}} = 0.9$ and $\gamma = 0.75$. The milder exponent in the blocking case reflects that download cost is more consequential when no model is loaded, but should not be entirely dominant.

$\rho(t)$ is a time-decay importance factor: As time passes, the initialization penalty decreases. We eventually accept better assignments regardless of initialization cost, gradually moving towards an optimal assignment. We model the decay using a modified sigmoid function:

$$\rho(t) = 1 - \frac{1}{1 + \exp\left(-\frac{t - t_{\text{mid}}}{t_{\text{sharp}}}\right)}$$

where t is the time elapsed since the last redistribution, $t_{\text{mid}} = 60\text{s}$ is the midpoint at which the penalty has halved, and $t_{\text{sharp}} = t_{\text{mid}}/5 = 12\text{s}$ controls the steepness.

5.2.3 Total Cost

To accumulate costs, we define a cost triple $C = \langle c_e, c_s, c_m \rangle \in \mathbb{R}^3_{\geq 0}$, where c_e is the accumulated execution cost, c_s is the accumulated sum of initialization costs, and c_m is the accumulated maximum initialization cost. For a single worker assignment (w_k, i, j), the corresponding cost triple is:

$$C(w_k, i, j) = \langle C_{\text{exec}}(w_k, i, j),\ C_{\text{init}}(w_k, i, j),\ C_{\text{init}}(w_k, i, j) \rangle$$

Addition of two cost triples is defined as:

$$C_1 + C_2 := \langle c_{e,1} + c_{e,2},\ c_{s,1} + c_{s,2},\ \max(c_{m,1},\ c_{m,2}) \rangle$$

so that execution and total initialization costs accumulate linearly, while the maximum initialization cost across workers is tracked exactly. Given a complete assignment A, the accumulated triple $C(A) = \sum_{(w_k,i,j) \in A} C(w_k, i, j)$ is collapsed to a scalar via:

$$C_{\text{total}}(A) = c_e + \alpha \cdot c_m + (1 - \alpha) \cdot c_s$$

The two initialization terms reflect a trade-off: workers download weights in parallel, so the bottleneck is c_m, the slowest worker; however, on shared networks, aggregate bandwidth c_s also matters. The constant $\alpha \in [0, 1]$ balances these two regimes, with $\alpha = 1$ assuming full parallelism (e.g., weights served from a CDN) and $\alpha = 0$ treating downloads as serial (e.g., the server's uplink is the bottleneck). We use $\alpha = 0.8$ as the default, reflecting that download parallelism is the common case.

5.2.4 Assignment Acceptance Criteria

We accept a candidate assignment A_{cand} over the active assignment A_{cur} if it satisfies:

$$C_{\text{total}}(A_{\text{cand}}) < \tau \cdot C_{\text{exec}}(A_{\text{cur}})$$

where $\tau = 0.95$. For A_{cur} we ignore the initialization cost, as it is already deployed. Since we reduce the importance of initialization with time, eventually only the execution cost of both assignments matters. The threshold $\tau \in [0, 1]$ prevents thrashing where slightly better assignments result in repartitioning.

5.2.5 Fault Tolerance Modeling

When there is no better primary assignment, the algorithm searches for a *shadow assignment* to prepare for worker failures. We modify our initialization cost to discourage reusing workers from the current assignment A_{cur}:

$$C_{\text{init}}^{\text{FT}}(w_k, i, j) = \begin{cases} \infty & \text{if strict mode and } w_k \in A_{\text{cur}} \\ \infty & \text{if } w_k \in A_{\text{cur}} \text{ and } [i,j) \cap A_{\text{cur}}(w_k) \neq \emptyset \\ \lambda \cdot C_{\text{init}}(w_k, i, j) & \text{if } w_k \in A_{\text{cur}} \text{ (non-overlapping)} \\ C_{\text{init}}(w_k, i, j) & \text{if } w_k \notin A_{\text{cur}} \end{cases}$$

with $\lambda = 10$. Unused workers are preferred because they can download weights without affecting the inference performance of the active assignment.

The algorithm uses two strategies:

1. **Strict spare finding**: Workers in A_{cur} are excluded; we try to find a fully independent assignment. In case a worker disconnects, the server can switch to this assignment with minimal downtime.
2. **Redundancy finding**: Workers in A_{cur} are included, but only for non-overlapping layer ranges. Although this assignment might require weights to be downloaded, we reduce the initialization cost.

If neither strategy yields a valid assignment, the server waits for the next event as described in Section 4.4.1.

5.3 Complexity Analysis

5.3.1 Impact of Worker Ordering

Assuming a fixed order of workers, the partitioning problem can be solved in $\mathcal{O}(n^2 m)$ time using dynamic programming (DP). However, in a heterogeneous environment, relying on fixed order does not guarantee an optimal or feasible solution.

Consider a minimal counterexample: assume two workers, w_1 and w_2, with 8 GB and 16 GB of memory, respectively. The model consists of two layers that require 16 GB and 1 GB of memory. The worker ordering $\langle w_1, w_2 \rangle$ does not produce a valid assignment, as the 8 GB worker cannot load the first layer. In contrast, the ordering $\langle w_2, w_1 \rangle$ satisfies the memory constraints. Consequently, the permutation of workers inherently dictates both the feasibility and the total cost of the resulting assignment.

5.3.2 Problem Variations and Optimality

Given our problem formulation and cost function, we can categorize the problem into 4 variations:

Variant 1: Full Heterogeneity (The General Case)

According to our design goal, workers are heterogeneous. Although standard transformer layers are architecturally similar, our inclusion of the local cache metric D_w makes the initialization cost dependent on the worker ordering. Furthermore, the embedding and output layers also exhibit different characteristics from the transformer layers. To guarantee a global optimum, an algorithm must evaluate all possible worker permutations. This results in a time complexity of $\mathcal{O}(m! \cdot n^2 m)$; an NP-hard problem. For dynamic real-time repartitioning, this approach is intractable.

Variant 2: Heterogeneity via Heuristic Search (Our Approach)

To make the general case tractable, we relax the optimality constraint. We can use the $\mathcal{O}(n^2 m)$ DP algorithm to evaluate the cost of any given worker ordering, exploring the $m!$ permutation space using a metaheuristic (Simulated Annealing). This allows us to approximate the global optimum.

Variant 3: Structural Homogeneity (Ignoring Initialization State)

If we assume that all transformer layers are homogeneous, we can find the global optimum in $\mathcal{O}(m^2 \cdot n^2 m)$. Because the embedding and output layers differ, we need to permute the first and last workers in the pipeline. This reduces the number of permutations to $m(m-1)$. This solution is feasible in practice as long as we are willing to remove the initialization cost.

Variant 4: Strict Homogeneity

If we assume that all layers are homogeneous, we can solve the problem optimally in $\mathcal{O}(n^2 m)$ using the DP algorithm.

If we assume that all workers are homogeneous, we can use a greedy algorithm to solve the problem in $\mathcal{O}(m)$.

If we assume that both layers and workers are homogeneous, the problem is optimally solvable in $\mathcal{O}(1)$. We can assign $\lfloor \frac{M}{R} \rfloor$ layers per worker to the first $\lceil \frac{n}{\lfloor \frac{M}{R} \rfloor} \rceil$ workers. This guarantees that the model is fully loaded and that no layer or worker scan is required.

5.3.3 Hardness Discussion

Variant 1 is computationally intractable, and Variant 4 violates our design goal of supporting heterogeneous workers.

In scenarios where downloading the entire model beforehand on all workers is a viable strategy, we can approximate or remove the initialization cost and consider all transformer layers the same. In a web-based environment, requiring each worker to download gigabytes of model weights is impractical. Although such an approach would technically preserve

the zero-setup design goal, forcing workers to download the full model before contributing to the compute cluster introduces unacceptable latency.

We believe that the initialization cost is essential for our system and implement variant 2 using DP and SA.

5.4 The Dynamic Programming Algorithm

5.4.1 Recurrence

For our DP algorithm, we use a matrix of size $(n+1) \times (m+1)$. Each location (i, j) in the matrix stores the minimum cost of using the first j workers of a given order to run the first i layers. This results in recurrence:

$$dp[0][j] = 0 \quad \forall j \in \{0, \dots, m\}$$
$$dp[i][j] = \min_{0 \leq x \leq i} \{dp[x][j-1] + C(w_j, x, i)\}$$

The optimal assignment of layers $[0, i)$ using j workers is decomposed into the optimal assignment of layers $[0, x)$ using $j-1$ workers plus the cost of assigning $[x, i)$ to worker w_j.

We make 3 modifications to the recurrence as defined in [Ski20] for the *Ordered Partition Problem*:

- **Skipping Workers**: We allow $x = i$, indicating that a worker can be assigned 0 layers. Otherwise, if there are more workers than layers, the algorithm would break, or workers with little memory could block the algorithm from finding assignments if they appear first in the ordering.
- **Sum Of Costs**: Instead of minimizing the maximum of partition, which would be useful if partitions could be computed in parallel, we use the sum. The result gives us the total end-to-end latency in microseconds for the execution cost and a measure of the initialization penalty for the entire pipeline.
- **Custom Cost Function**: Our cost function is composed using multiple metrics instead of just integers. To model hard constraints, we use ∞ in our cost function.

5.4.2 Algorithm

We can transform the recurrence formulation into a DP algorithm that fills the matrix $dp_{(n+1)\times(m+1)}$ from $(0, 0)$ to (n, m). After running the algorithm, (n, m) contains the minimum cost of assigning all layers using all workers. We introduce a second matrix $d_{(n+1)\times(m+1)}$ where we store x. Each location (i, j) gives us the layer range $[x, i)$ assigned to the worker w_j. This results in Algorithm 5.1.

Two implementation details are noteworthy: First, the innermost loop iterates x downward from i to 0, so the range $[x, i)$ increases with each step. Once $C_{\text{exec}}(w_j, x, i) = \infty$, indicating that the memory constraint is violated, no smaller value of x can produce a feasible range, and the loop terminates early. Second, if $\text{dp}[n][m] = \infty$ after the table is filled, there is no valid assignment for the current worker ordering, as the coverage constraint is violated. The algorithm walks backward from n to find the deepest layer i with $\text{dp}[i][m] < \infty$, returning a partial assignment. As discussed in Section 4.4.2, partial assignments are used to download weights when the model cannot be fully covered.

5.4.3 Complexity

We can see that the algorithm runs in $\mathcal{O}(n^2 m)$ time and uses $\mathcal{O}(nm)$ space. Note that for this to hold, the cost function needs to use prefix sums instead of iterating the arrays (see Section 6.3.1).

In practice, the inner loop is bounded by $\lfloor \frac{M_{w_j}}{R} \rfloor$ since we stop the iteration as soon as the memory constraint is violated. LLMs typically have fewer than 100 transformer layers. For example, Qwen3-0.6B has 28 transformer layers, Qwen3-4B has 36 layers, and Deepseek-V3.2 has 61 layers. We therefore expect the DP algorithm to run in microseconds. We only expect performance issues if there are many heterogeneous workers, resulting in a large permutation space of worker ordering.

5.5 Worker Ordering Optimization

5.5.1 The Ordering Problem

As discussed in Section 5.3.1, finding a globally optimal solution is intractable with heterogeneous layers and workers. We can find an optimal solution for a fixed worker ordering; therefore, we shift our focus to searching through the space of worker permutations.

5.5.2 Exhaustive Search

For $m < 8$ and $n^2 m < 20{,}000$, we evaluate all $m!$ permutations using the DP algorithm to find the globally optimal solution. Permutations are evaluated in parallel using a thread pool. The size constraint ensures that the algorithm completes without stalling the server for large models ($n > 50$).

5.5.3 Simulated Annealing

For problems of size $n^2 m < 50{,}000$, for example, up to roughly 20 workers for 50 layers, we use simulated annealing (SA) to search through the space of worker permutations. Each candidate permutation is evaluated using the DP algorithm, so each SA iteration costs $O(n^2 m)$.

Neighborhood Given a current permutation, a neighbor is generated by one of five perturbation operations chosen at random:

- **Range swap** (20%): Two non-overlapping subranges of equal length are exchanged, producing a large structural change.
- **Range reversal** (20%): A contiguous subrange is reversed.
- **Swap of first/last worker** (20%): The first or last worker is swapped with another random worker. The first and last layers typically require more memory than other layers, so having workers with little memory first or last might lead to infeasible solutions even if a solution with a different ordering exists.
- **Element swap** (40%): Two random workers are swapped.

Temperature schedule We estimate a temperature by taking 10 neighborhood steps and computing the average differences $\bar{\Delta}$ between consecutive orderings. The initial temperature $T_0 = 10\bar{\Delta}$ corresponds to accepting approximately 90% of bad moves initially ($e^{-1/10} \approx 0.9$). The schedule uses geometric cooling over 1000 iterations to a final temperature of $T_f = 10{,}000$µs, at which point the cost differences are considered negligible:

$$T_k = T_0 \cdot \alpha^k, \quad \alpha = \left(\frac{T_f}{T_0}\right)^{1/1000}$$

We use the Metropolis criterion for accepting moves: a bad move with cost increase δ is accepted with probability $e^{-\delta/T_k}$. The search ends early if no improvement is found after 200 iterations.

The values of T_0, T_f, the iteration count, and the neighborhood function have not been systematically evaluated. Parameter tuning using real-world data is left for future work (8.4).

5.5.4 Time-Bounded Random Search

For problems of size $n^2m \geq 50{,}000$, SA becomes too slow. Instead, we perform a parallel random search: each thread in the thread pool randomly shuffles the worker sequence, evaluates the permutation with the DP algorithm, and reports the result. The best result across all threads is returned after a fixed budget of 100ms.

At this problem scale, we expect the ordering to be less critical in practice. With many workers available, the DP algorithm can skip more workers, reducing the penalty for suboptimal orderings. Problems exceeding $n^2m \geq 10^8$ are rejected outright, as no strategy can produce a result within a useful time budget.

Algorithm 5.1: Dynamic programming for layer-to-worker assignment

Input: Number of layers n; ordered workers $w_1, \ldots, w_m$
Output: A (partial) assignment A

```
   // Initialization
 1 dp[0][j] ← 0   ∀j ∈ {0, ..., m};
 2 dp[i][j] ← ∞   ∀i ∈ {1, ..., n}, j ∈ {0, ..., m};
 3 d[i][j] ← ⊥   ∀i ∈ {1, ..., n}, j ∈ {0, ..., m};
   // Fill DP table
 4 for j ← 1 to m do
 5 |   for i ← 1 to n do
 6 |   |   for x ← i to 0 do
 7 |   |   |   worker_cost ← C_total(w_j, x, i);
 8 |   |   |   if worker_cost = ∞ then
 9 |   |   |   |   break // Constraint violated
10 |   |   |   end
11 |   |   |   cost ← dp[x][j − 1] + worker_cost;
12 |   |   |   if cost < dp[i][j] then
13 |   |   |   |   dp[i][j] ← cost;
14 |   |   |   |   d[i][j] ← x;
15 |   |   |   end
16 |   |   end
17 |   end
18 end
   // Find deepest covered layer
19 i ← n;
20 while i > 0  and  dp[i][m] = ∞ do
21 |   i ← i − 1;
22 end
   // Reconstruct assignment by backtracking
23 A ← ⟨⟩;
24 for j ← m to 1 do
25 |   x ← d[i][j];
26 |   if x < i then
27 |   |   A ← A ⌢ ⟨w_j, x, i⟩;
28 |   end
29 |   i ← x;
30 end
31 A ← Reverse(A);
32 return A
```

CHAPTER 6

Implementation Details

In this Chapter, we will cover additional implementation details focusing on the data structures, partitioning algorithm, and ONNX graph manipulation.

6.1 Internal Representations

We use several data structures to represent the LLM dataflow graph on the server:

- **GraphProto**: The LLM is stored as an ONNX file. We use the ONNX Protocol Buffer definition to read and parse the file into Rust structs. When we partition the model, we create new instances of *ModelProto* and serialize them to files so that workers can load the partitions as ONNX.
- **SimpleGraph**: As described in Section 2.3, ONNX represents machine learning dataflow graphs using nodes with named inputs and outputs. We create an internal representation containing minimal information called *SimpleGraph*. To store the nodes, we use binary trees, which allows us to retain the topological sort of the ONNX graph representation. Using this custom internal representation removes the complexity of working with protobuf definitions during partitioning; we only need to convert *SimpleGraphs* to and from ONNX models.
- **CoarsenedGraph**: Inspired by METIS, we add the structure *CoarsenedGraph*. It is a different representation of *SimpleGraph* in which multiple nodes are grouped into subgraphs, yielding a list of topologically sorted *SimpleGraphs*. We use this structure to represent LLMs layers, abstracting over individual operators.
- **CompiledModel**: Once a model is loaded on a set of workers, the server creates a *CompiledModel*. It is a more efficient internal representation of the model pipeline using dependency counting to determine which partitions of the model

can be computed next given a set of already computed inputs. Although this data structure is not required for linear transformer models, we could use it to extend the system to arbitrary models in the future.

6.2 Graph Processing Pipeline

This section describes how the server moves between graph representations to facilitate dynamic partitioning.

6.2.1 Preprocessing

At startup, the server reads the ONNX model using protobuf. We perform the following preprocessing steps:

- **Strip Graph**: We remove unnecessary data so that the ONNX graph is as lean as possible.
- **Externalize Weights** We externalize the weights so that each initializer is stored in a separate file using the SHA-256 hash as the filename. This allows workers to cache weights more easily.
- **Constant Folding**: We perform basic constant folding to avoid having to duplicate constants across multiple layers when partitioning the model.

6.2.2 ONNX to SimpleGraph

To convert the ONNX model to a *SimpleGraph*, we read the graph's input and output names and use them one-to-one for the *SimpleGraph*. For nodes, create a binary tree using the node's index in ONNX as the key. The values are *SimpleNodes* with input and output names that correspond to the edges in the ONNX graph. We use a binary tree since we need to store the original node indices for conversion back to ONNX after partitioning the graph.

6.2.3 SimpleGraph to CoarsenedGraph

Creating a *CoarsenedGraph* from a *SimpleGraph* involves two steps: First, we find the indices of layer boundaries in the topological sort. Second, we use the indices to split the *SimpleGraph*, creating a new graph for each layer.

Layer Boundary Detection

Since LLMs follow strict naming conventions when converted to ONNX with Microsoft Olive, we can use pattern matching on tensor names to find layer boundaries. To find the boundaries, we iterate through all nodes in the *SimpleGraph*. If a node contains input tensors with the prefix `/model/layer.N/` and output tensors with the prefix

`/model/layer.(N+1)/`, we use the node's index as a partition point. This approach is efficient and works on all modern models we tested (Qwen3, Phi-4, gpt-oss, and Gemma 2).

Graph Splitting

To divide the LLM into individual layers, we iterate through the partitioning points in reverse. We split off all nodes with an index larger than the current split point and create a new *SimpleGraph*. We need to iterate in reverse to accumulate the inputs of all layers.

Splitting the graph involves splitting the nodes and computing the new inputs and outputs of the subgraphs. Let $\mathcal{I}$ and $\mathcal{O}$ denote the original graph inputs and outputs, and let I_k and O_k denote the union of all node inputs and outputs of subgraph k. The boundary tensors, those produced by subgraph 1 and consumed by subgraph 2, are:

$$\beta = (O_1 \cap I_2) \setminus O_2$$

Subtracting O_2 excludes constants that appear in both O_1 and O_2. The new inputs and outputs of each subgraph are:

$$\begin{aligned} \text{in}_1 &= \mathcal{I} \cap I_1, & \text{out}_1 &= (\mathcal{O} \cap O_1) \cup \beta \\ \text{in}_2 &= (\mathcal{I} \cap I_2) \cup \beta, & \text{out}_2 &= \mathcal{O} \cap O_2 \end{aligned}$$

Subgraph 1 exposes β as additional outputs; subgraph 2 receives β as additional inputs, forming the communication boundary between the partitions.

6.2.4 Partitioning

Given an assignment A of layers to workers, the server merges each layer range into one *SimpleGraph*. To minimize network transfer, partitions expose only the output required by the downstream partitions or the global model output $\mathcal{O}$.

We iterate backwards over partitions, maintaining an accumulator of required outputs, initialized as $\mathcal{R} = \mathcal{O}$:

$$\begin{aligned} \text{in}_k &= I_k \setminus O_k \\ \text{out}_k &= \mathcal{R} \cap O_k \\ \mathcal{R} &\leftarrow \mathcal{R} \cup \text{in}_k \end{aligned}$$

As a result, we create a new *CoarsenedGraph* where subgraphs match the layer ranges of A and the graph boundaries minimize tensor transfers.

6.2.5 SimpleGraph to ONNX

Using the *GraphProto* of the original ONNX model and the partitioned *CoarsenedGraph*, we create one ONNX model per worker. We use the *SimpleGraphs* contained in the *CoarsenedGraph* to obtain the nodes and construct the inputs and outputs of the new *GraphProto*. Note that the original ONNX model must contain shape information for all new input and output tensors.

6.3 Partitioning Algorithm

6.3.1 Cost Function

As mentioned in Section 5.4, to satisfy the computational bound of $\mathcal{O}(n^2m)$ for the DP algorithm, we need to use prefix sums during cost calculation. For the layer metrics W_i (weights in bytes), R_i (required memory), and C_i (computational cost), we only need to compute the prefix sums once as they remain static. Since D_w (cached layers) can change on every invocation of the partitioning algorithm, we need to compute each time before starting the algorithm.

To reuse the same algorithm for execution and fault tolerance costs, we define our *find assignment* function as generic over the cost calculator. This allows us to use the DP algorithm with arbitrary cost functions.

6.3.2 DP Algorithm Core

We switch the matrix defined in Algorithm 5.1 to a single vector. We define this vector once when starting the algorithm and reuse it when evaluating worker permutations to avoid reallocation on every DP invocation.

To evaluate permutations in parallel, we use the Rust crate *rayon*. Using an internal thread pool, rayon provides convenient extensions to Rust iterators. Using parallel iterators, we speed up the evaluation of permutations for small problem sizes and discover more permutations in the time-bounded search.

CHAPTER 7

Evaluation

In this Chapter, we empirically evaluate the system and partitioning algorithm. We start by explaining the experiment setup, including the test bed, models, and workloads. We establish baseline measurements, exploring the system's performance characteristics in different settings. Next, we present the measurements we use to answer research questions 3-5. Lastly, we conduct case studies showing the system's behavior in specific scenarios.

7.1 Experimental Setup

7.1.1 Testbed

Table 7.1: Device Hardware Specifications

Device	CPU	RAM	GPU
Littlecorn	AMD Ryzen 7 5700G	64 GB DDR4-3200	RTX 4070 Ti (12GB)
Aidos	AMD EPYC 7452	96 GB (6x16GB)	4x RTX A4000 (16GB)
Lenovo ThinkStation PGX 30KL0004GF	ARM Cortex-X925	128 GB LPDDR5x	GB10 Grace Blackwell
Jetson AGX Thor	ARM Neoverse-V3AE	128 GB LPDDR5x	Blackwell (2560-core)
Jetson AGX Orin	ARM Cortex-A78AE	64 GB LPDDR5	Ampere (2048-core)
Jetson Orin Nano	ARM Cortex-A78AE	8 GB LPDDR5	Ampere (1024-core)

Hardware As shown in Table 7.1, we use 6 devices for our evaluation. The distributed inference server always runs on Littlecorn; other devices are used as workers. Aidos has 4 homogeneous GPUs with 16GB of VRAM each. For all other devices, we use the CPU. As shown in Table 7.2, this makes Aidos the fastest worker.

Software Environment We use lock files and fixed compiler versions to ensure reproducibility and facilitate distribution. We use Rust *nightly-2026-01-15* to compile the server. Littlecorn runs Ubuntu 22.04.4 LTS, Aidos runs Ubuntu 24.04.1 LTS, ThinkStation

Table 7.2: Worker Metrics and TPS obtained by running inference on each worker (Qwen3-0.6B, chat, 1 worker)

Worker	Memory (GB)	Bandwidth (MB/s)	Latency (ms)	TPS
Aidos	15.72	117.49	0.64	59.22
Thor	118.07	32.43	1.87	48.08
Pgx	116.15	37.53	1.58	25.23
Agx	59.08	31.86	1.97	18.94
Nano	6.63	31.26	2.52	9.96

runs Ubuntu 24.04.3 LTS, Thor runs Ubuntu 24.04.3 LTS, and all other Jetson devices run Ubuntu 22.04.5 LTS. Workers use Python 3.12, ONNX Runtime 1.23.2, and CUDA 12.

Network Table 7.2 shows metrics collected by the inference server. We observe that connections have low latency, with a maximum of 2.5 milliseconds, and bandwidth between 30 and 120 MB/s. Devices are connected via LAN and are in physical proximity.

7.1.2 Models

Table 7.3: Model Characteristics. We obtained model sizes from the Ollama model library [Oll] (quantized), Hugging Face [Hug25a] (full precision), and the filesystem after conversion with Microsoft Olive (ONNX).

Model	Size (quant)	Size (full)	Size (ONNX)	Layers	Hidden Size
Qwen/Qwen3-0.6B	523 MB	1.5 GB	3 GB	28	1024
Qwen/Qwen3-1.7B	1.4 GB	3.8 GB	6.9 GB	28	2048
Qwen/Qwen3-4B	2.5 GB	7.5 GB	16.1 GB	36	2560
Qwen/Qwen3-8B	5.2 GB	15.2 GB	32.7 GB	36	4096
Qwen/Qwen3-14B	9.3 GB	27.5 GB	59 GB	40	5120

We evaluate our system using the Qwen3 model family. Using Microsoft Olive, we convert the Hugging Face model to ONNX. Table 7.3 shows the models we use, their size, number of transformer layers, and hidden size. We use full-precision models in our evaluation, leading to increased model size and slower inference speeds compared to quantized models. We mainly use Qwen3-0.6B for our baseline evaluations since it can run on all devices and Qwen3-4B for experiments with multiple workers. Other models are used to provide insights on model scaling. We observe that Qwen3-4B in ONNX is already too large to run on most consumer devices; we use it to show how the inference server behaves when single-device deployment is infeasible.

We observe that the hidden size increases with model size. Intermediate tensors are of

shape `[batch_size, sequence_length, hidden_size]` and need to be transferred at partition boundaries; the network transfer time increases with model size.

7.1.3 Workloads

We evaluate the system using 3 prompting strategies:

- **Chat**: We use a short input prompt (25 tokens) and generate a short response (128 tokens). Unless specified otherwise, we use this prompting strategy.
- **Creative**: We use a short prompt to generate a long response (1024 tokens).
- **Summarization**: We use a long prompt (1024 tokens) to generate a short response (128 tokens).

For all experiments, we run inference requests 5 times and report the mean and standard deviation.

Since we use fewer than 8 workers, the algorithm always chooses the global minimum according to the cost model. We ignore the initialization cost in our experiments to ensure a fair comparison across runs.

7.1.4 Metrics

Typically, LLMs inference speed is evaluated using:

- **Token Per Second (TPS)**: TPS is a measure of throughput. We primarily use this metric to evaluate inference performance.
- **Time To First Token (TTFT)**: As described in Section 2.1.2, the prefill and decoding phases have different performance characteristics. TTFT measures the time it takes to generate the first token. Since we focus on decoding and end-to-end latency, we do not consider TTFT.
- **Time Per Output Token (TPOT)**: TPOT measures the time it takes to generate each token in the decoding phase. It is equivalent to the end-to-end latency we defined in our algorithm. When comparing algorithm estimates and measurements, we use TPOT as the metric.

We collect information about partitioning decisions and assignments from server logs. We use this information to analyze the algorithm's behavior.

7.1.5 Limitations

Our evaluation is subject to 3 main limitations:

Quantization We use floating-point 32 models in our evaluation. There are two main reasons for this decision: During development, we were unable to create quantized models that produced coherent outputs using Microsoft Olive. Although this problem was fixed in later versions of Olive, it affected the development and evaluation strategy. Secondly, we were unable to run quantized models with Puppeteer. Using the standard desktop Chrome with WebGPU, we were able to run 16-bit floating-point models. However, using Puppeteer, we were unable to load the *shader16* module, preventing us from running quantized models in our test bed.

Jetson Devices We were unable to run ONNX models on Jetson devices using the GPU. ONNX Runtime with CUDA is not well supported on Jetson devices; other execution engines, such as TensorRT, also did not work. Due to time constraints, we decided to use the devices with ONNX Runtime and the CPU. We believe that this has a limited impact on our evaluation. Using Aidos, we can observe the system's behavior on the GPU. We can show that the algorithm exhibits the same characteristics on the CPU and GPU.

Model Diversity In our evaluation, we only use the Qwen3 model family. We were able to run other models, such as Phi-4, Gemma 2, and gpt-oss-20b during development; however, we do not evaluate them.

7.2 Baseline

7.2.1 Experimental Design

Objective

Observe the system's behavior and performance characteristics across different dimensions. The baseline is used to position this work in the broader field of AI and to explain the results of other experiments.

Methodology

We evaluate the system in the following dimensions:

- **Engines**: To determine the system's overhead, we implement two native centralized inference solutions using ONNX Runtime with CUDA. The implementations are based on Python and Rust. We run the chat prompt with Qwen3-0.6B using Aidos. For our system, we use 1 worker on Aidos with the server running on Littlecorn.
- **Devices**: We compare the performance of the devices in the test bed using Qwen3-0.6B with the chat prompt. The server runs on Littlecorn.
- **Prompt Types**: We run all 3 prompts using Qwen3-0.6B on Aidos.

- **Web vs. Native**: We compare native and web-based execution by running all 3 prompts on Aidos using Puppeteer and Qwen3-0.6B. Both the inference server and the website run on Littlecorn.
- **Model Size**: To compare model sizes, we use 3 workers, each worker running on a different GPU on Aidos. We force the system to use all 3 workers by limiting each worker's memory. We use the models Qwen3-0.6B, Qwen3-1.7B, and Qwen3-4B with the chat prompt.
- **Worker Scaling**: We determine the system's scalability with increasing worker count by running Qwen3-0.6B with the chat prompt on Aidos. We use 1, 2, 3, and 4 workers, forcing worker usage by limiting memory.
- **Network**: We run the worker scaling experiment again, introducing an artificial latency of 20 milliseconds, a download rate of 10 MB/s, and an upload rate of 2 MB/s. We compare the results to determine the impact of slow interconnects on inference performance.

7.2.2 Results

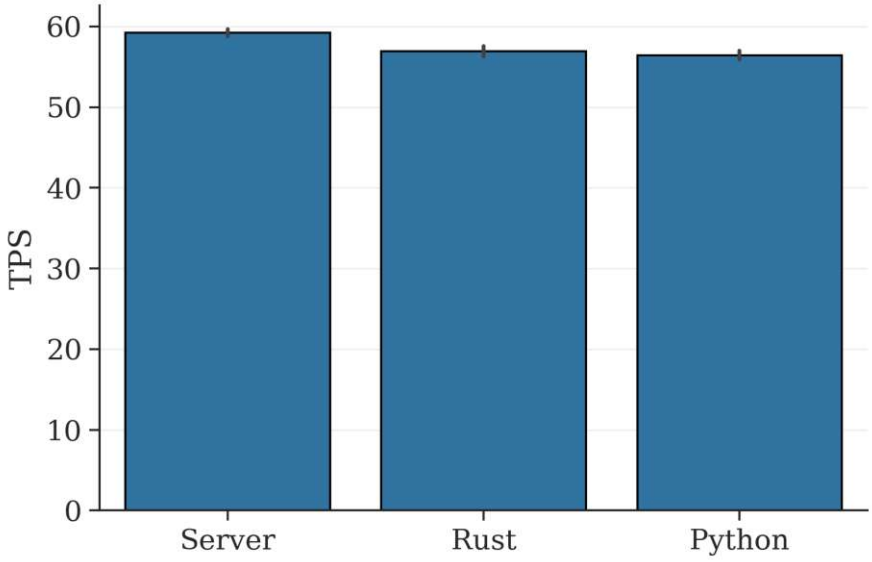


Figure 7.1: Comparison of the server to centralized inference using ONNX Runtime.

As shown in Figure 7.1, the system performs better than naive centralized inference. This result is initially surprising, given that the system adds communication overhead between Littlecorn and Aidos. However, we implement optimizations in the system that are not present in the centralized versions. For example, we move token decoding into the ONNX graph, running greedy sampling directly on the GPU. Processing the server's logs, we observe that less than 0.1% of the inference time is spent on the server and less

than 5% on the network. This shows that the server, data serialization, and networking add negligible overhead for one-worker configurations with fast interconnects.

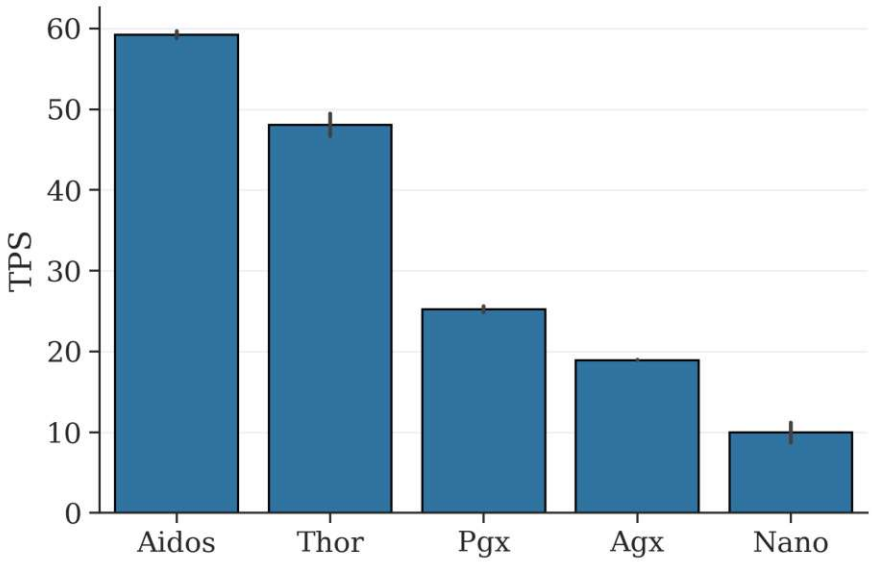


Figure 7.2: Inference Speed on different devices for Qwen3-0.6B.

Figure 7.2 shows the inference speed for different devices. Aidos outperforms all other devices. Nano, the slowest worker, has about $\frac{1}{6}$ of Aidos' performance. Using these devices, we can evaluate the system across a wide variety of settings with devices of differing hardware capabilities.

As shown in Figure 7.3, TPS drops significantly with increasing sequence length. Prompts with sequence lengths of more than 1000 tokens retain only 25% of the inference speed compared to short prompts with about 100 tokens. This suggests that ONNX Runtime does not scale well with increasing sequence lengths. Interestingly, web-based inference scales better with increasing sequence length. It is unclear why this is the case; possible explanations include that the overhead of WebGPU kernel initialization becomes less important as the sequence length increases or that ONNX Runtime Web uses GPU kernels that are better optimized for LLM inference.

Figure 7.4 shows how TPS scales with model size. We observe a slight sub-linear scaling. This is likely because the ratio of session overhead to computation decreases as the size of the model increases.

As shown in Figure 7.5a, increasing the worker count adds little overhead in a fast network. We observe a decrease in performance from 1 worker to 2 workers. The difference in time can be attributed mainly to worker computation, rather than the network or the server. It is unclear why this drop is not as pronounced for more workers. Possible explanations are that ONNX Runtime can perform optimizations for the full model, which are not possible in the split model, or that switching from 1 to 2 workers introduces hardware

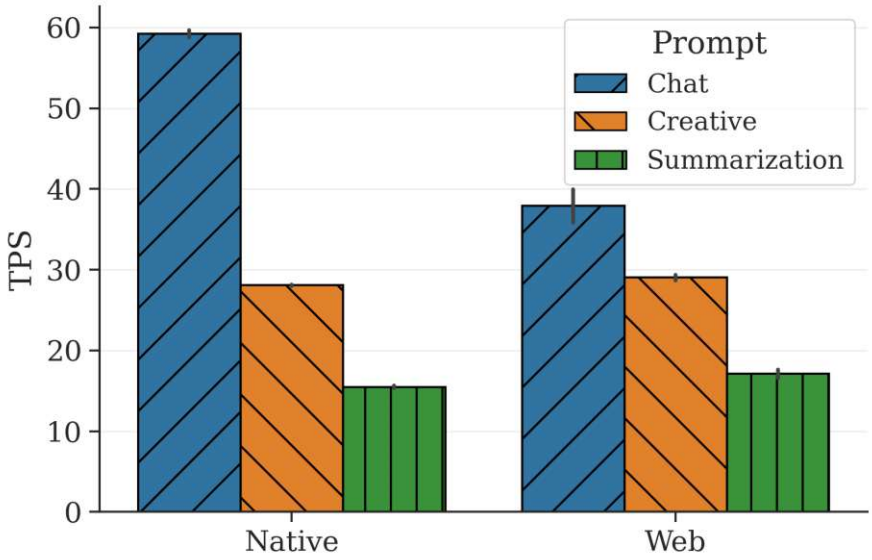


Figure 7.3: Comparison of sequence lengths in a native and web-based setting.

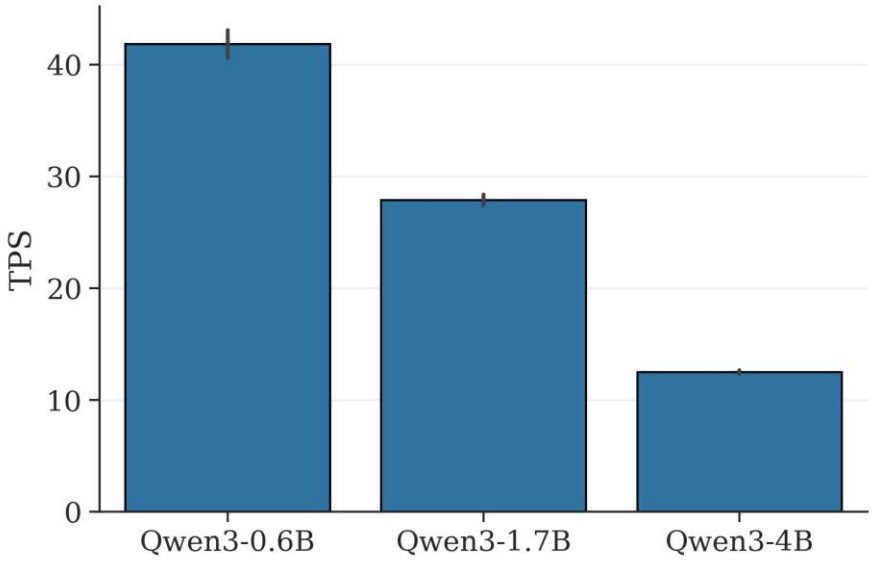
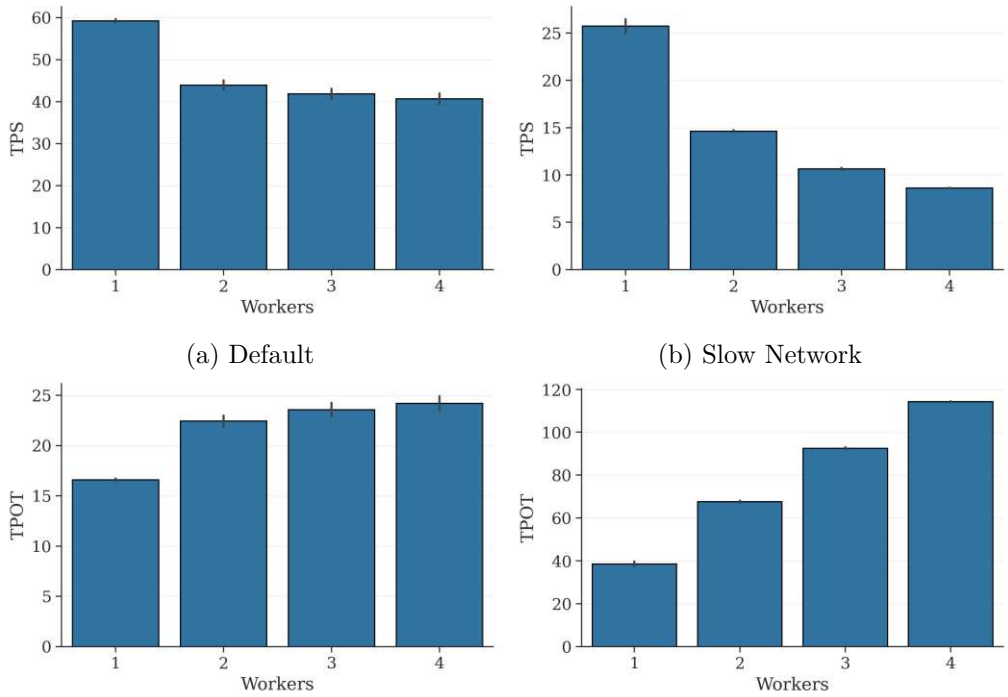


Figure 7.4: Comparison of TPS for increasing model size.

contention (CPU, CPU cache, PCIe) since workers use different GPUs on the same machine.

Introducing network constraints leads to a drop in performance. For 1 worker, the TPOT increases from 16 to 39 milliseconds. For 2 workers, the TPOT increases by 45 milliseconds compared to the fast-network case. This indicates that in slow networks,



(a) Default (b) Slow Network

(c) Default (d) Slow Network

Figure 7.5: Comparison of inference performance with increasing worker count. For the default configuration, refer to the network speed of Aidos in 7.2. For the slowed configuration, we use 20 milliseconds of latency, 10 MB/s download, and 2 MB/s upload.

latency is the biggest factor. However, bandwidth should not be ignored, especially if it drops below 1 MB/s; the intermediate tensors become relevant at this point. Since the intermediate tensors scale with model size, high bandwidth is more important for larger models. For Qwen3-0.6B, intermediate tensors total 8KB; for Qwen3-8B, they total 32KB.

7.3 RQ3: Cost Model Validation

7.3.1 Experimental Design

Objective

Empirically validate the accuracy of the cost model by comparing estimates with real-world measurements. The success criterion is a mean absolute percentage error (MAPE) of less than 20%.

Methodology

We evaluate the system in 3 settings:

- **Homogeneous Devices**: We use only devices of a single type to run models. This means we can run up to 4 workers using Aidos' GPUs; for other devices, we use 1 worker. In this setting, we focus on smaller models.
- **Heterogeneous Devices**: We use different combinations of devices; examples include *Aidos+Thor*, *Aidos+Pgx+Nano*, or *Agx+Nano*.
- **Heterogeneous Devices + Artificial Network**: We introduce artificial latency and bandwidth using *tc* to simulate slower networks. For example, we introduce a latency of 10 milliseconds on Littlecorn or constrain bandwidth and latency on Aidos.

For each setting, we vary the model size, the number of workers, and the device type. We record the algorithm's initial estimate, in which it uses the metrics collected during initialization. We compare this estimate with the mean TPOT of all 5 inference requests. Additionally, we update estimates using metrics collected while the worker is connected. For each of the 5 inference requests, we pair the TPOT with the corresponding running estimate.

7.3.2 Results

Table 7.4: Cost Model Error by Model for Initial Estimates

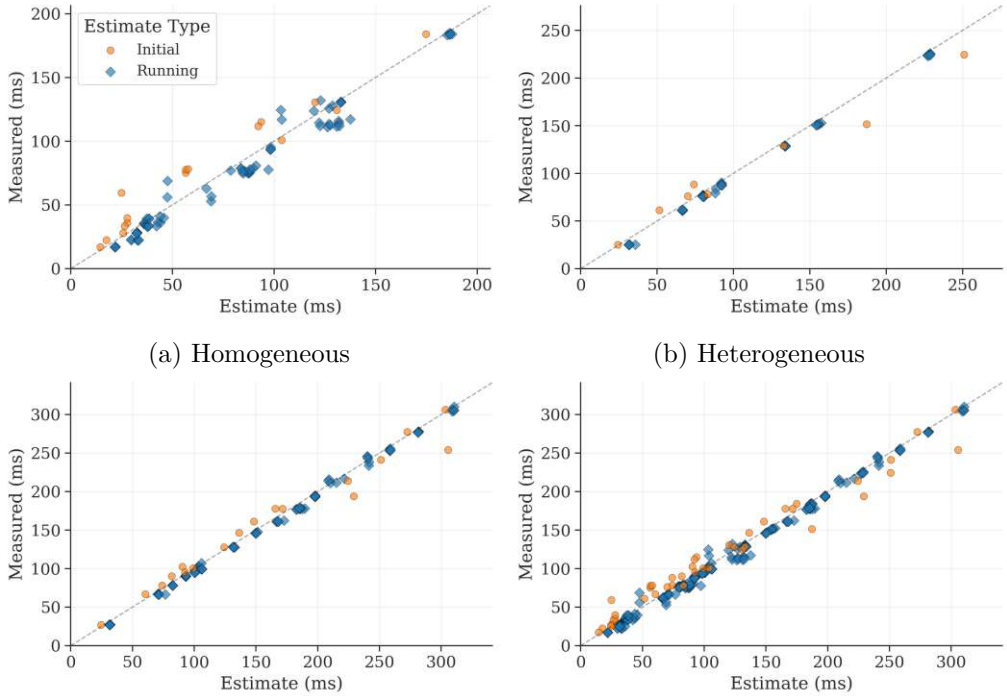
Model	MAE (ms)	MAPE (%)	Count
Qwen3-0.6B	4.5	11.7	8
Qwen3-1.7B	10.0	15.6	12
Qwen3-4B	10.7	10.0	13
Qwen3-8B	23.2	13.1	9

Table 7.5: Cost Model Error by Model for Running Estimates

Model	MAE (ms)	MAPE (%)	Count
Qwen3-0.6B	4.9	16.6	40
Qwen3-1.7B	4.7	8.3	60
Qwen3-4B	6.8	6.3	65
Qwen3-8B	6.5	4.4	45

Tables 7.4 and 7.5 show MAE and MAPE for initial and running estimates in all configurations. Initial estimates achieve a MAPE of 10-16% across all model sizes. The MAE grows with model size, while the MAPE remains roughly constant.

After updating the execution speed metric during inference, the MAE remains below 7 milliseconds regardless of the model size; the MAPE drops to 4.4% for Qwen3-8B. At sub-20 millisecond token latencies for Qwen3-0.6B on Aidos, small inaccuracies constitute a larger fraction of the total. This makes it harder to accurately model low-latency settings with small models.



(a) Homogeneous

(b) Heterogeneous

(c) Heterogeneous + Slow Network

(d) Combined

Figure 7.6: Cost model validation across network and device configurations.

Figure 7.6 shows that the estimates under artificial network slowdown align closest to the diagonal. The homogeneous setting exhibits the greatest spread, particularly for initial estimates of smaller models. Running estimates typically correct this initial bias, confirming that accuracy is primarily limited by the quality of the execution-speed metric rather than the cost model's structure.

7.4 RQ4: Static vs. Dynamic Partitioning

7.4.1 Experimental Design

Objective

Evaluate the system's behavior and performance when partitioning models into an equal number of layers per partition compared to dynamic partitioning. We aim to show that dynamic partitioning is strictly superior to equal splitting in terms of inference speed.

Methodology

We evaluate inference speed in 5 settings:

- **Equal (3/5 workers)**: We partition the model into equal parts, each of which has the same number of layers assigned.
- **Dynamic (3/5 workers)**: We use the same workers as the corresponding equal-split setting. We force the usage of workers by limiting the memory; however, we leave some room for the dynamic partitioning algorithm to assign different layer ranges to the devices.
- **Dynamic (free)**: We use all 5 devices with full memory. This allows the dynamic partitioning algorithm to find the optimal assignment according to the cost model.

For all configurations, we use Qwen3-4B with the chat prompt. We use all devices in all configurations, letting the algorithm assign model partitions to workers based on the cost model.

For the equal-split strategy, we compute the number of layers per partition as $L_p = \lceil \frac{L}{\text{splits}} \rceil$. The first partition can have fewer layers; all others have exactly L_p layers. The embedding layer is typically larger than the other layers, so the required memory for each partition evens out. We reuse the partitioning algorithm to evaluate worker orderings.

7.4.2 Results

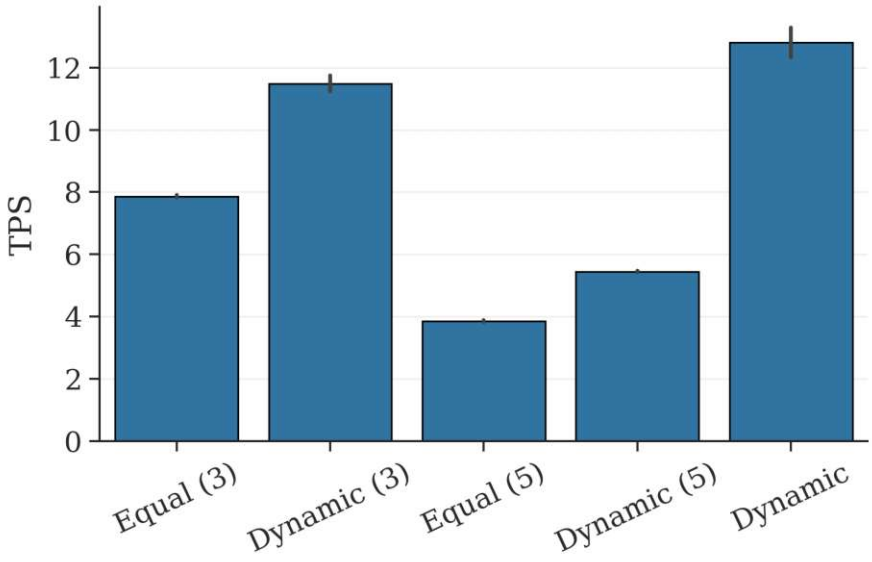


Figure 7.7: Comparison of TPS for different partitioning strategies.

As shown in Figure 7.7, dynamic partitioning is superior to equal-splitting for each corresponding setting. Dynamic partitioning uses the full capabilities of each device to achieve the highest TPS.

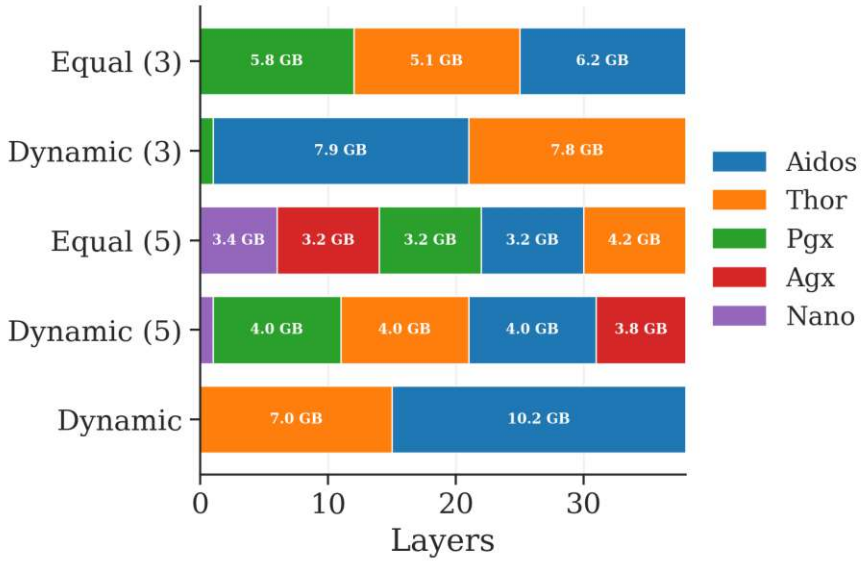


Figure 7.8: Comparison of Assignments for different partitioning strategies.

Figure 7.8 shows the layer assignments for each strategy. We observe that all strategies choose faster workers over slower ones. Within the top three devices, Pgx has the lowest throughput; the dynamic partitioning algorithm minimizes Pgx's use when 3 workers are enforced. Since the memory limit is set to 12 GB and we reserve $\frac{1}{3}$ for activation tensors and KV cache, the maximum assignable weights are 8 GB. Both Aidos and Thor are used to their maximum.

Similarly, for 5 workers, the proportion of Nano and Agx is reduced. Aidos, Thor, and Pgx are used to their limit. When the algorithm chooses workers freely, it uses only Aidos and Thor, resulting in the fastest inference speed.

For equal-splitting with 3 workers, the algorithm assigns Aidos to the last layers. This is because the output layer is the most computationally intensive. For dynamic partitioning, Thor is assigned the last layer. In this case, memory is the bottleneck. Since the output layer is larger, Aidos can do more work by executing transformer layers. Additionally, the output of the last layer is only a few bytes (the next token); since Thor has a slower network connection, using it last reduces end-to-end latency. For the *Dynamic* configuration, the assignment switches again since the embedding layer is least computationally intensive.

7.5 RQ5: Metric Ablation Study

7.5.1 Experimental Design

Objective

Determine the effect of each metric on partitioning decisions and inference throughput.

Methodology

We run two experiments with Qwen3-4B and the chat prompt:

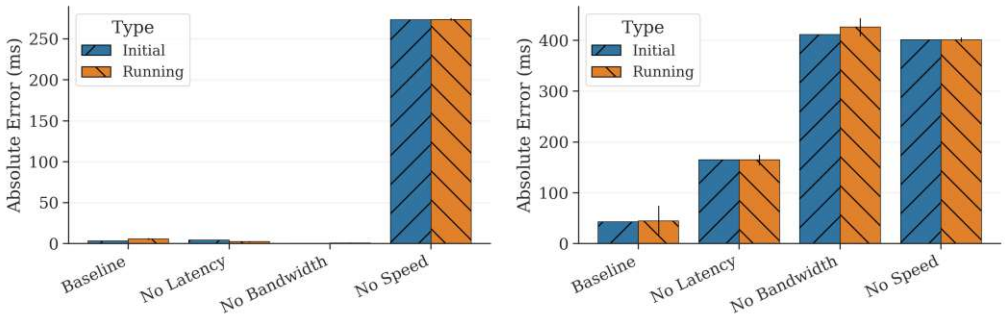
- **Cost Estimation**: To estimate the impact of each metric on the partitioning algorithm, we conduct an ablation study. We start with a baseline measurement, recording the end-to-end latency estimate and TPOT. Then, one by one, we disable the metrics for computational speed (SO_w and CS_w), latency (P_w), and bandwidth (B_w), measuring their impact on the cost estimate. We run the experiment twice: once with the default network speeds as shown in Table 7.2 and once with artificial latency and bandwidth. We use the following network configuration:
  - **Littlecorn**: 5 milliseconds of latency, 50 MB/s download, and 10 MB/s upload.
  - **Aidos**: 100 milliseconds of latency.
  - **Pgx**: 128 KB/s of upload and download.

  We force the usage of all devices by restricting memory.
- **Assignment Traps**: We rerun the experiment with the slowed network. We set the memory so that 3 devices are required. We expect the algorithm to fall into the designed traps; for example, choosing Pgx when the bandwidth measurements are disabled.

7.5.2 Results

Figure 7.9 shows the cost model's error when different metrics are disabled. Using the default network with less than 2 milliseconds of latency and more than 20 MB/s bandwidth for all workers, the network has virtually no impact on end-to-end latency. Disabling execution speed, the cost model estimates an end-to-end latency of 13 milliseconds and an error of 275 milliseconds. In the network-constrained configuration, we observe larger estimation errors across all settings. The baseline's standard error is particularly high. Interestingly, this is because TPOT varies; the algorithm's estimate remains consistent. The reason is unclear; possibly, latency is being applied twice for some inference requests due to packet loss.

Figure 7.10 shows the TPS of the system as we disable individual metrics. The baseline uses Aidos, even though it adds 100 milliseconds of latency, resulting in the same



(a) Default (b) Slow Network

Figure 7.9: Algorithm estimates during the metric ablation study.

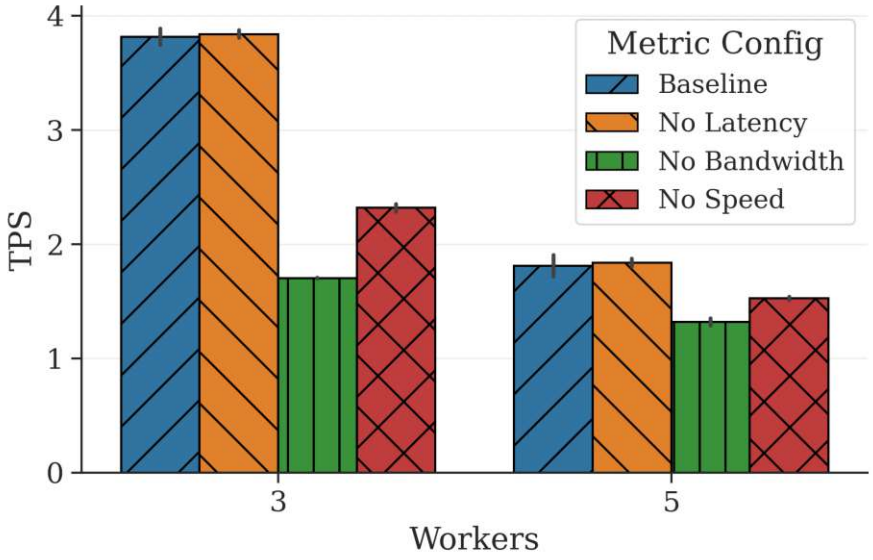


Figure 7.10: Comparison of TPS during an ablation study of the metrics.

performance as with latency disabled. Disabling bandwidth leads to the largest drop in performance, as the algorithm chooses Pgx, which has a bandwidth of 128 KB/s. At this speed, even tensors the size of a few KB affect end-to-end latency.

For the 5-worker configuration, disabling computational speed leads to slower devices being used more. Regarding bandwidth, we find that the order of workers matters. With bandwidth disabled, Pgx is assigned layers in the middle of the model, requiring the transfer of larger intermediate tensors.

7.6 Case Studies

7.6.1 Case Study 1: 3 Heterogeneous Devices

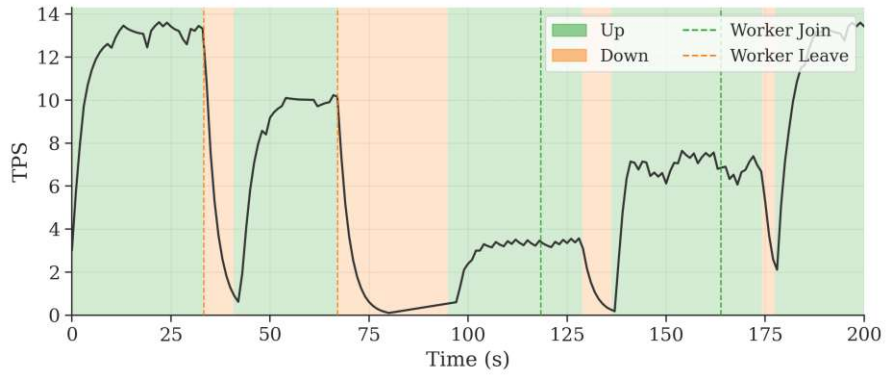


Figure 7.11: System performance and state over time as workers connect and disconnect.

With this case study, we show how the system behaves over time using 3 heterogeneous devices. We use Aidos, Thor, and Agx to run Qwen3-4B with the chat prompt. The chat prompt runs in a loop, continuing to generate until the experiment ends.

We start the experiment with all 3 workers connected. Using Aidos and Thor (the two faster workers), we achieve 13 tokens per second. At 35 seconds, Aidos disconnects. The server recovers after 7 seconds and continues token generation using Thor. Inference speed drops as the fastest worker is no longer available. At second 67, Thor disconnects. Since Agx is less powerful than Thor, creating the new ONNX Runtime session on Agx takes 27 seconds. At second 118, Aidos reconnects, taking 10 seconds during initialization. The server prepares the new assignment in the background (which requires negligible time since all weights are already downloaded) and commits it at second 128. Eight seconds later, the server resumes generation with higher throughput. At second 163, Thor reconnects, triggering repartitioning. The server restores its original assignment, achieving maximum throughput.

7.6.2 Case Study 2: Large Model

Table 7.6: Inference Metrics (Qwen3-14B)

Metric	Mean	Std
TPS (tok/s)	3.17	0.06
TTFT (ms)	815.68	73.73
TPOT (ms)	312.02	6.15

In this case study, we show that the system can find optimal assignments for large models on heterogeneous devices. We limit device memory: 16 GB for Aidos, 64 GB for Thor,

Table 7.7: Worker Layer Assignment (Qwen3-14B)

Worker	Layers	Estimate (ms)	RTT (ms)	Compute (ms)	Network (ms)
Pgx	0–1	4.34	2.79	0.19	2.60
Thor	1–35	335.06	280.84	277.88	2.96
Aidos	35–42	34.56	32.29	31.14	1.14

and 32 GB for Pgx. This roughly corresponds to a user running a desktop PC (GPU and CPU) and a secondary device.

Table 7.6 shows the inference statistics; we achieve 3.35 tokens per second for Qwen3-14B in full precision.

As shown in Table 7.7, Pgx is assigned the embedding layer, as it is the least computationally intensive yet the largest in memory footprint. Aidos runs the output layer.

Regarding cost estimation, we initially overestimate the cost of Thor by 55 milliseconds. This error is reduced to 10 milliseconds by updating the metrics while running the model.

CHAPTER 8

Discussion

This Chapter interprets the results of our evaluation, answers each research question, and discusses limitations and future work.

8.1 Research Questions

8.1.1 LLM Partitioning

> What methods and tools exist for partitioning ONNX-based LLMs for distributed inference?

As described in Section 2.3, Python libraries such as `onnx` and `olive` can be used to manipulate ONNX models and convert LLMs to the ONNX format. We used `olive` to convert Hugging Face LLMs to ONNX. Rather than using `onnx` for model partitioning, we deserialized the ONNX model into Rust structs using Protobuf. This approach accelerated development and provided greater flexibility when modifying the ONNX graph. For example, we added custom operators that execute token decoding in ONNX, thus reducing network traffic.

In Section 6.2, we describe the model partitioning process. The ONNX graph is first converted into an internal representation used during partitioning. To distribute the resulting partitions to workers, this representation is converted back to ONNX using the original model as a reference.

8.1.2 Algorithmic Complexity

> What computational complexity trade-offs arise in the proposed worker-aware dynamic partitioning algorithm, and under what conditions is it tractable?

We use dynamic programming to find layer to worker assignments. The algorithm runs in $\mathcal{O}(n^2m)$ for n layers and m workers and finds a solution that is optimal for a given worker ordering. However, finding a global optimum is intractable as all worker permutations would need to be evaluated. For less than 8 workers, we explore all permutations and find the globally optimal solution. For larger problem sizes, we use simulated annealing or time-bounded random search to explore the permutation space.

In Section 2.4, we justify our choice of algorithm. We found that the structure of LLMs, where the dataflow graph is a linear set of transformer layers, does not require general graph partitioning techniques such as the Kernighan–Lin algorithm [KL70] or METIS [KK98]. The intermediate tensors between LLM layers can be considered min-cuts for practical purposes (technically, there are better cuts, for example, by severing a single node). Choosing small cuts is important for reducing the data transfer between partitions in pipeline parallelism. Using this information, we reduce the problem to the *Ordered Partition Problem* [Ski20].

In Chapter 5, we provide a formulation of the problem along with a cost function to minimize end-to-end latency during token generation. We adapt the dynamic programming algorithm in [Ski20] to LLM partitioning.

Finding globally optimal solutions using our formulation is NP-hard. In practice, the DP algorithm can find close to optimal solutions, as transformer layers are homogeneous when optimizing for end-to-end latency. The problem becomes more complex only when initialization cost is introduced; for example, by accounting for which layers a worker has already downloaded. Although the problem is solvable in polynomial time if we consider layers or workers homogeneous, we believe that heterogeneity is essential in practice. To facilitate web-based distributed inference with ephemeral workers, reducing downtime after a disconnect is vital; we need to avoid multi-gigabyte downloads when the server has no active assignment.

8.1.3 Cost Model Validation

> How closely does the end-to-end inference latency estimated by the worker-aware dynamic partitioning algorithm match real-world measurements?

We achieve a MAPE of 12.6% when using metrics collected during worker initialization and 8.4% after updating metrics during token generation. The initial error remains at about 12% regardless of the model size. After updating the metrics, the MAPE for Qwen3-4B drops to 4.4%; the MAE is reduced to a constant 5 to 7 milliseconds regardless of the model size. The largest errors resulted from inaccurate estimates of computational speed rather than of network conditions. For small models such as Qwen3-0.6B, even small inaccuracies led to large increases in MAPE, as token latency is already below 20 milliseconds when using our most powerful device.

We achieved our success criterion of MAPE below 20%, showing that the system can accurately estimate the real-world token latency. The cost model and algorithm can

find near-optimal assignments for heterogeneous workers under a wide variety of network conditions.

8.1.4 Static vs. Dynamic Partitioning

> How does dynamic worker-aware partitioning compare to static equal-layer splitting in terms of inference throughput?

As shown in Section 7.4, dynamic partitioning consistently outperforms equal-layer splitting. Using device information such as computational speed, the dynamic partitioning algorithm is able to assign larger parts of the model to fast workers, achieving higher TPS. When using 3 workers for both strategies, dynamic partitioning achieves 46% higher TPS; for 5 workers, we achieve 41.3% higher TPS.

Given an accurate cost model, dynamic partitioning can consistently find better solutions than equal-layer splitting. To achieve high throughput with heterogeneous workers, dynamic partitioning is essential.

8.1.5 Metrics

> How does removing individual worker metrics from the partitioning algorithm affect inference throughput?

In Section 7.5, we analyze the impact of latency, bandwidth, and computational speed on the algorithm's cost estimates. We show that the importance of each metric is mostly dependent on the system's environment. In fast-network environments, computational speed is the most important metric. If workers with slow connections are involved, latency and bandwidth ensure that the worker is excluded from the assignment.

Regarding throughput, we found that bandwidth can affect the order of workers that the algorithm chooses. By changing the order of workers, the algorithm reduced network time by 25%.

8.2 Limitations

8.2.1 Quantization

As discussed in Section 7.1.5, we used full-precision models in our evaluation due to issues with Olive and Puppeteer. This complicates the comparison of our system with other approaches. Regarding the cost model and dynamic partitioning, we believe that the system would behave the same for quantized models; however, we did not empirically prove it.

8.2.2 Diversity

Our evaluation uses one type of GPU; other devices run on the CPU. Due to compatibility issues, we were unable to use ONNX Runtime with the GPU on Jetson devices. Since the cost model is accurate for the CPUs and GPU we tested, we believe that the system generalizes to other hardware.

In our evaluation, we used only the Qwen3 model family. We were able to run other models, such as Phi-4, Gemma 2, and gpt-oss-20b, but we did not evaluate them.

8.2.3 Web

We evaluated web-based inference in a single-worker configuration using Qwen3-0.6B. We experimented with Qwen3-4B; however, we ran into issues when trying to load parts of the model with ONNX Runtime Web. Since collecting debug information would have required compiling ONNX Runtime Web from source and bundling it into our application, we opted not to pursue the issue further. A possible explanation is the 4 GB memory limitation of WebAssembly. We were able to run Qwen3-0.6B (3 GB), but not Qwen3-1.7B (6.9 GB) using WebAssembly. Limiting the memory to 4 GB, we were able to run a part of Qwen3-1.7B in the browser and the rest using Python. Although WebAssembly 3.0 introduced 64-bit addressing, it is unclear whether ONNX Runtime Web can currently utilize it.

Regarding our evaluation, Chrome does not support assigning specific GPUs to a browser instance. We attempted to use Docker to assign GPUs 2-4 on Aidos. On Jetson devices, we attempted to start a browser instance with WebGPU enabled. We were unable to make progress using either approach and decided not to investigate further due to time constraints.

8.2.4 Cost Model

We showed that our cost model can accurately estimate the token latency in our test bed. We did not test across a wide variety of real-world networks with higher packet loss or WiFi connections.

In our evaluation, we disabled the initialization cost to obtain consistent runs across experiments. During development, the system was able to reduce downtime using initialization cost; however, we did not empirically evaluate this.

8.2.5 Algorithm

In our experiments, we used a maximum of 5 workers. Therefore, we always found the global optimum according to the cost model. We did not evaluate simulated annealing or time-bounded random search. Running SA during development resulted in better solutions than using random worker orderings; however, we did not tune its parameters or compare the solutions with the global optimum.

8.3 Practical Viability

Our system can dynamically partition LLMs in heterogeneous environments. For lower worker counts, assignments approach the global optimum given the accuracy of the cost model. Using native Python workers, we were able to run large models while requiring short setup times. The setup involves 2 commands: cloning the repository and starting the worker using `uv`. The setup complexity of native workers might increase if, for example, CUDA is not configured. Using the web as a deployment channel, we were able to achieve zero-setup deployment for small models.

We identify 3 obstacles preventing wider adoption:

- **Performance**: ONNX Runtime is not optimized for LLM inference and is slower than LLM-specific engines such as Ollama [oll25] or vLLM [vll25]. In our evaluation, we found that performance decreases drastically with the sequence length. This is likely due to inefficiencies in how ONNX Runtime handles the KV cache. For each forward pass, the entire cache is copied to a new location in memory. LLM inference engines achieve better performance with sequence length through optimizations such as PagedAttention [KLZ+23].
- **ONNX**: Using existing tools, we were unable to convert models larger than Qwen3-14B to ONNX due to out-of-memory issues. For wider adoption, the system requires a reliable way to convert LLMs with higher parameter counts to quantized ONNX models.
- **The Web**: WebGPU is not yet widely supported, requiring experimental flags on Chrome in Linux. Using ONNX Runtime Web, we were unable to run large models even with enough VRAM.

We believe that all 3 problems are solvable. The last two require a more mature ecosystem around ONNX and machine learning in the browser. To solve the first issue, it should be possible to convert the partitioned ONNX models to more efficient representations for LLM inference. We will discuss this idea in future work.

8.4 Future Work

In this Section, we provide directions for future work. We start with immediate system improvements that involve mostly engineering problems. Secondly, we discuss promising avenues that were excluded due to scope and resource constraints.

8.4.1 System Extensions

Multi-Turn Conversations

The system currently only supports single prompt questions, discarding the KV cache afterwards. Extending the server to support multi-turn conversations requires a limited

time investment but is required for inference systems operating in practice.

Metrics

Required Memory The memory required to run a layer is currently determined by multiplying the weights of the layer by a constant. This metric can be improved by considering information such as the number of parallel inference requests or the sequence lengths.

Robustness As shown in our evaluation, inaccuracies in the cost model are directly related to the metrics. Making metrics collection more robust, for example, for scenarios where many workers connect in a short time frame, improves the quality of assignments.

Other Metrics The cost model can be extended to include more metrics. For example, we can divide bandwidth into upload and download. Another option is to add dimensionless metrics such as worker reliability, nudging the algorithm to prefer stable long running workers.

Optimized Workers

Instead of using ONNX Runtime as the inference engine, we imagine using ONNX only as the transport format. Workers can implement custom optimized kernels for LLM inference, significantly improving inference throughput.

8.4.2 Research Areas

Peer-To-Peer Communication

We use WebSockets for communication due to our requirement of web-based LLM inference. This led to a hub-and-spoke topology. Using WebRTC, we could keep the server as the central authority while instructing workers to communicate directly among each other. This would reduce network related latency by roughly half, which is essential in slower network environments.

Parallel Assignments

The system only supports a single active model assignment. By adapting the system to allow for multiple active graphs, we can scale to a larger number of workers. Combining this idea with peer-to-peer communication, we could scale to hundreds or thousands of geo-distributed workers, enabling internet-scale collaborative inference. This requires a specialized partitioning algorithm that can handle large worker counts while considering already deployed models.

CHAPTER 9

Conclusion

In this thesis, we design and implement a dynamic worker-aware LLM partitioning algorithm. Using our cost function, the algorithm finds layer-to-worker assignments that minimize end-to-end latency during token generation. We integrate the algorithm into our distributed inference server. Using metrics collected during runtime, the server automatically repartitions the LLM, moving towards optimal assignments as time progresses.

We implement the inference server in over 5,500 lines of Rust code. The server exposes a WebSocket API for users to send inference requests and for workers to share computational resources. We implement two workers: a web-based worker running in the browser using WebGPU, and a native worker using Python and CUDA. During runtime, the server is responsible for orchestrating distributed LLM inference, collecting metrics, and partitioning the LLM.

We find that due to the architecture of LLMs, where the model's dataflow graph is a set of transformer layers connected by small intermediate tensors, we can reduce the graph partitioning problem to a variant of the *Ordered Partition Problem*. This problem is solvable in $\mathcal{O}(n^2m)$ for n layers and m workers. We design a cost function that jointly minimizes execution cost and initialization cost. As metrics for execution cost, we use execution speed, latency, and bandwidth. To reduce downtime in case of worker disconnects, we leverage cached model weights.

We find that the cost model can accurately estimate end-to-end latency during token generation, achieving a MAPE of 12.6% using metrics collected during worker initialization. By updating metrics during runtime, we reduce the MAPE to 8.4% overall, and to 4.4% for large models. This shows that our algorithm can find near-optimal assignments in real-world settings.

Dynamic worker-aware partitioning consistently outperforms static equal-layer splitting, with the metrics ablation study confirming that the algorithm correctly avoids or down-weights sub-optimal workers. In two case studies, we show that the system can

automatically recover from unexpected worker disconnects and that we are able to run large models with the algorithm choosing the optimal assignment.

We position this work not as a replacement for engines like Ollama or vLLM, but as a first step towards reducing the entrance barrier for distributed LLM inference. Using the browser for code distribution and dynamic partitioning, we enable distributed inference across heterogeneous devices, allowing users to run models that exceed the capacity of any single participant.

Overview of Generative AI Tools Used

LLMs were used during the development of the distributed inference server. We primarily used Gemini to better understand the complex ecosystem of ONNX and LLM inference.

During the writing of this thesis, LLMs were used as writing assistants. Gemini 3 Pro and Claude Sonnet 4.5 were used as assistants, helping to structure the thesis. The resulting outputs were not used 1 to 1, always requiring modifications. However, they served as great sources for inspiration, especially when rerunning prompts and merging their different outlooks on ideas.

To rewrite paragraphs, LLMs were used to continue the flow of writing without getting stuck on formulation.

List of Figures

2.1 A single layer of Qwen3-0.6B stored as an ONNX file and visualized using Netron. 8

4.1 The system's topology with the server as the central orchestrator. Arrows represent the data flow during inference. 23
4.2 The dynamic repartitioning loop. Events such as a worker connecting, disconnecting or periodic timeouts trigger the find assignment function. 29
4.3 Server state transitions. 31

7.1 Comparison of the server to centralized inference using ONNX Runtime. . 53
7.2 Inference Speed on different devices for Qwen3-0.6B. 54
7.3 Comparison of sequence lengths in a native and web-based setting. 55
7.4 Comparison of TPS for increasing model size. 55
7.5 Comparison of inference performance with increasing worker count. For the default configuration, refer to the network speed of Aidos in 7.2. For the slowed configuration, we use 20 milliseconds of latency, 10 MB/s download, and 2 MB/s upload. 56
7.6 Cost model validation across network and device configurations. 58
7.7 Comparison of TPS for different partitioning strategies. 59
7.8 Comparison of Assignments for different partitioning strategies. 60
7.9 Algorithm estimates during the metric ablation study. 62
7.10 Comparison of TPS during an ablation study of the metrics. 62
7.11 System performance and state over time as workers connect and disconnect. 63

List of Tables

2.1 The model size in bytes for different models. The model size for full precision was taken from Hugging Face [Hug25a], for the quantized model size we used `Q4_K_M` GGUF from the Ollama model library [Oll]. 10

3.1 Comparison of LLM inference systems 19

7.1 Device Hardware Specifications 49
7.2 Worker Metrics and TPS obtained by running inference on each worker (Qwen3-0.6B, chat, 1 worker) 50
7.3 Model Characteristics. We obtained model sizes from the Ollama model library [Oll] (quantized), Hugging Face [Hug25a] (full precision), and the filesystem after conversion with Microsoft Olive (ONNX). 50
7.4 Cost Model Error by Model for Initial Estimates 57
7.5 Cost Model Error by Model for Running Estimates 57
7.6 Inference Metrics (Qwen3-14B) 63
7.7 Worker Layer Assignment (Qwen3-14B) 64

List of Algorithms

5.1 Dynamic programming for layer-to-worker assignment 43

Bibliography

[ABC+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[ALd+23] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore, December 2023. Association for Computational Linguistics.

[AMY18] Fan Angela, Lewis Mike, and Dauphin Yann. Hierarchical Neural Story Generation. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[BMS+16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, pages 117–158. Springer International Publishing, Cham, 2016.

[BRC+23] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A. Raffel. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. *Advances in Neural Information Processing Systems*, 36:12312–12331, December 2023.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3nd edition, 2009.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[CMSL25] Zhiyang Chen, Yun Ma, Haiyang Shen, and Mugeng Liu. WeInfer: Unleashing the Power of WebGPU on LLM Inference in Web Browsers. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, pages 4264–4273, New York, NY, USA, April 2025. Association for Computing Machinery.

[Dec25] Decoding Strategies in Large Language Models. `https://huggingface.co/blog/mlabonne/decoding-strategies`, September 2025.

[DFE+22] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, December 2022.

[DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, December 2022.

[DLF+24] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou,

Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model, June 2024.

[DLM+25] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y. Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu,

Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models, December 2025.

[DLZ+24] Shengang Deng, Zhi Ling, Mingrui Zhu, Shuangwu Chen, Xiaofeng Jiang, and Jian Yang. Adaptive DNN Partitioning Method for Pipeline Parallel Training with Time-Varying Resources. In *2024 6th International Conference on Frontier Technologies of Information and Computer (ICFTIC)*, pages 571–577, December 2024.

[Epo25] Epoch AI. Data on AI Models, July 2025.

[exo25] Exo-explore/exo. `https://github.com/exo-explore/exo`, December 2025.

[FAHA23] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, March 2023.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[ggm25] Ggml-org/llama.cpp. `https://github.com/ggml-org/llama.cpp`, December 2025.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[HBD+20] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The Curious Case of Neural Text Degeneration. In *Eighth International Conference on Learning Representations*, April 2020.

[HCB+19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[HIZ+22] Yang Hu, Connor Imes, Xuanang Zhao, Souvik Kundu, Peter A. Beerel, Stephen P. Crago, and John Paul Walters. PipeEdge: Pipeline Parallelism for Large-Scale Model Inference on Heterogeneous Edge Devices. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 298–307, August 2022.

[HL22] Chenghao Hu and Baochun Li. Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 330–339, May 2022.

[Hug25a] Hugging Face – The AI community building the future. `https://huggingface.co/`, December 2025.

[hug25b] Huggingface/optimum. `https://github.com/huggingface/optimum`, December 2025.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

[KLZ+23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626, New York, NY, USA, October 2023. Association for Computing Machinery.

[LLM] LLM Inference guide for Web | Google AI Edge | Google AI for Developers. `https://ai.google.dev/edge/mediapipe/solutions/genai/llm_inference/web_js`.

[LTT+24] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and

Acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, May 2024.

[LZG+21] 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 *Proceedings of the 38th International Conference on Machine Learning*, pages 6543–6552. PMLR, July 2021.

[mic25] Microsoft/Olive. `https://github.com/microsoft/Olive`, December 2025.

[MP23] Anthony Moi and Nicolas Patry. HuggingFace's Tokenizers. `https://github.com/huggingface/tokenizers`, April 2023.

[MY17] Freitag Markus and Al-Onaizan Yaser. Beam Search Strategies for Neural Machine Translation. *Proceedings of the First Workshop on Neural Machine Translation*, 2017.

[Ngu25] Xuan-Son Nguyen. Ngxson/wllama. `https://github.com/ngxson/wllama`, December 2025.

[NHP+19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, October 2019. Association for Computing Machinery.

[NKQ+25] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A Comprehensive Overview of Large Language Models. *ACM Trans. Intell. Syst. Technol.*, 16(5):106:1–106:72, August 2025.

[Oll] Ollama Model Library. `https://ollama.com/library`.

[oll25] Ollama/ollama. `https://github.com/ollama/ollama`, December 2025.

[ONN18] ONNX Runtime developers. ONNX Runtime. `https://github.com/microsoft/onnxruntime`, November 2018.

[onn25] Onnx/onnx. `https://github.com/onnx/onnx`, December 2025.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang,

Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 721, pages 8026–8037. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[pro25] Protocolbuffers/protobuf. `https://github.com/protocolbuffers/protobuf`, December 2025.

[RMF+24] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. *IEEE Access*, 12:26839–26874, 2024.

[RQZ+24] Charlie F. Ruan, Yucheng Qin, Xun Zhou, Ruihang Lai, Hongyi Jin, Yixin Dong, Bohan Hou, Meng-Shiun Yu, Yiyan Zhai, Sudeep Agarwal, Hangrui Cao, Siyuan Feng, and Tianqi Chen. WebLLM: A High-Performance In-Browser LLM Inference Engine, December 2024.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[SAL+24] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with Rotary Position Embedding. *Neurocomputing*, 568:127063, February 2024.

[sgl25] Sgl-project/sglang. `https://github.com/sgl-project/sglang`, December 2025.

[Ski20] Steven S. Skiena. *The Algorithm Design Manual*. Texts in Computer Science. Springer International Publishing, Cham, 2020.

[SPP+20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020.

[SYC+24] Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. A Thorough Examination of Decoding Methods in the Era of LLMs, February 2024.

[TBB+25] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao

Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi K2: Open agentic intelligence, 2025.

[Tra] Transformers.js. `https://huggingface.co/docs/transformers.js/en/index`.

[TTHT21] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. Automatic Graph Partitioning for Very Large-scale Deep Learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1004–1013, May 2021.

[vll25] Vllm-project/vllm. `https://github.com/vllm-project/vllm`, December 2025.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[Wer90] P.J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, October 1990.

[YGW+25] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. EdgeMoE: Empowering Sparse Large Language Models on

Mobile Devices. *IEEE Transactions on Mobile Computing*, 24(8):7059–7073, August 2025.

[YJK+22] 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, 2022.

[YLY+25] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report, May 2025.

[YWN+25] Zhengyi Yuan, Xiong Wang, Yuntao Nie, Yufei Tao, Yuqing Li, Zhiyuan Shao, Xiaofei Liao, Bo Li, and Hai Jin. DynPipe: Toward Dynamic End-to-End Pipeline Parallelism for Interference-Aware DNN Training. *IEEE Transactions on Parallel and Distributed Systems*, 36(11):2366–2382, November 2025.

[ZCZ+21] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. CoEdge: Cooperative DNN Inference With Adaptive Workload Partitioning Over Heterogeneous Edge Devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, April 2021.

[ZLZ+22] 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. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[ZS19] Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[ZSB+19] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium*

on Edge Computing, SEC '19, pages 195–208, New York, NY, USA, November 2019. Association for Computing Machinery.

[ZSC+25] Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. EdgeShard: Efficient LLM Inference via Collaborative Edge Computing. *IEEE Internet of Things Journal*, 12(10):13119–13131, May 2025.

[ZSL+24] Junchen Zhao, Yurun Song, Simeng Liu, Ian G. Harris, and Sangeetha Abdu Jyothi. LinguaLinked: Distributed Large Language Model Inference on Mobile Devices. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 160–171, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[ZZL+23] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.