A Model-Distributed Inference Approach for Large Language Models at the Edge

BY

DAVIDE MACARIO
B.S., Politecnico di Torino, Turin, Italy, 2022

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2024

Chicago, Illinois

Defense Committee:

Hulya Seferoglu, UIC, Chair and Advisor
Erdem Koyuncu, UIC
Michela Meo, Politecnico di Torino

ACKNOWLEDGMENTS

I want to thank my grandparents, who showed me that hard work always pays off.

I want to thank my family, who gave me the exceptional opportunity to participate in this program and always pushed me to do my best.

I wouldn't have made it without my friends, the ones from Italy, who always showed their support and those I made along the way, my roommates and lab mates.

I also want to thank my advisors for their valuable insights into the research world and UIC for allowing me to make my dream of living in the USA come true.

DM

TABLE OF CONTENTS

CHAPTER			PAGE
1	**INTRODUCTION**		1
2	**PREVIOUS WORK**		3
	2.1	Model-Distributed Inference	3
	2.2	Large Language Models overview	6
	2.2.1	Tokenizers	7
	2.2.2	LLM structure	8
	2.2.3	GPT-2 architecture	13
	2.2.4	Llama Architecture	15
	2.3	LLMs at the Edge	16
3	**MDI-LLM**		21
	3.1	Testbed	22
	3.2	Rationale	23
	3.3	Distributing NanoGPT	24
	3.3.1	Model partition	25
	3.3.2	Recurrent pipelining	28
	3.3.3	Practical implementation	30
	3.4	Distributing GPT-2	35
	3.4.1	Architectural changes	36
	3.5	Distributing Llama	39
	3.5.1	Final architecture of MDI-LLM	42
	3.5.2	Nodes operation	42
4	**PERFORMANCE ANALYSIS**		47
	4.1	Results – NanoGPT	47
	4.2	Results – GPT-2	53
	4.3	Results – Llama	59
5	**CONCLUSIONS**		65
	5.1	Future developments	67
	CITED LITERATURE		70
	VITA		74

LIST OF TABLES

TABLE		PAGE
I	FLAVOR-SPECIFIC MODEL PARAMETERS, GPT-2	35
II	LAYERS ASSIGNMENT – NANOGPT OVER 2 NODES	48
III	LAYERS ASSIGNMENT – NANOGPT OVER 3 NODES	48
IV	TOTAL (RAM + VRAM) MEMORY USAGE – NANOGPT . . .	53
V	LAYERS ASSIGNMENT – GPT-2 MODELS OVER 2 NODES .	54
VI	LAYERS ASSIGNMENT – GPT-2 MODELS OVER 3 NODES .	54
VII	TOTAL (RAM + VRAM) MEMORY USAGE – GPT-2	59
VIII	LAYERS ASSIGNMENT – LLAMA MODELS OVER 2 NODES	60
IX	LAYERS ASSIGNMENT – LLAMA MODELS OVER 3 NODES	61
X	TOTAL (RAM + VRAM) MEMORY USAGE – LLAMA	64

LIST OF FIGURES

FIGURE		PAGE
1	Attention block structure (GPT architecture and following)	10
2	GPT-2 model structure	14
3	NanoGPT partition scheme	26
4	Recurrent pipeline parallelism – 3-node network	30
5	Implementation of input message queues	33
6	GPT-2 partition scheme	37
7	Implementation of input and output message queues	38
8	MDI-LLM architecture	43
9	Layer partition scheme – Llama models	44
10	Time vs. N. of tokens – NanoGPT 7 blocks	51
11	Time vs. N. of tokens – NanoGPT 9 blocks	51
12	Time vs. N. of tokens – NanoGPT 12 blocks	52
13	Time vs. N. of tokens – detail at the origin	52
14	Time vs. N. of tokens – GPT-2	55
15	Time vs. N. of tokens – GPT-2 Medium	55
16	Time vs. N. of tokens – GPT-2 Large	56
17	Time vs. N. of tokens – GPT-2 XL	56
18	Time vs. N. of tokens – NanoLlama 304M	61
19	Time vs. N. of tokens – TinyLlama 1.1B	61

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
DDI	Data-Distributed Inference
DNN	Deep Neural Network
GenAI, GAI	Generative AI
GPT	Generative Pretrained Transformer
IoT	Internet of Things
LLM	Large Language Model
MDI	Model-Distributed Inference
MHA	Multi-Head Attention (layer)
NLP	Natural Language Processing
NN	Neural Network
RoPE	Rotary Positional Embedding

SUMMARY

With this work, we propose "MDI-LLM," a framework based on the model-distributed inference (MDI) approach used to deploy large language models (LLMs) over a network of low-capability devices at the internet edge. This method consists of partitioning the transformer-based model into chunks and assigning each to a different device, with intermediate model states transmitted between the nodes. We pair MDI with "recurrent pipeline parallelism", which allows the network nodes to process samples, specifically pieces of text for LLMs, in parallel. This prevents idle periods and achieves an increased throughput in generated tokens per unit of time.

We discuss the system's design and implementation, starting with small models to validate our assumptions on the benefits and moving on to larger, more recent ones. We also highlight the specific challenges that drove the choices that gave shape to the final version of the framework. Thanks to our framework, it is possible to run models that would not fit in the memory of a single device, increase the generation rate through parallel processing, and scale the system with ease by adding nodes to the network.

CHAPTER 1

INTRODUCTION

The last decade has seen the ascension of generative AI (GenAI) to one of the most widespread applications of artificial intelligence used today. The rise of this technology allowed access to tools, such as ChatGPT, that were considered science fiction in the past. Being the most trending AI application, academic and industrial research is currently focusing on it, as proven by the growth of publications and hype around the field. The main direction that is being pursued by the industry, however, is that of *scaling-up*, by increasing more and more the size of the models and the capabilities of the hardware. In the specific case of large language models for text generation, the most used form of generative AI, the size of the deep neural networks used, started out in the order of hundreds of millions of parameters. In contrast, today's state-of-the-art models use hundreds of billions of parameters. This translates into the need for a huge amount of computation power, making it impossible, or very expensive, to deploy LLMs outside the cloud. In particular, for applications at the edge, generative AI is still an unexplored ground, as, typically, edge devices lack the capabilities to run these models on their own.

Many approaches have been devised in the literature for performing deep neural network inference at the edge. These techniques try to overcome the limitations of edge devices on both computational capabilities and connectivity through distributed computing approaches, which involve several devices cooperating to process the data. Large Language Models pose an

additional challenge, given the large size of the neural network, which makes it impossible for individual edge devices to fit the whole model. As a result, data-parallelism approaches that split the input data to be processed among the network nodes are not viable in this scenario.

We propose a model-distributed inference approach for large language models (MDI-LLM), following a framework that has been proven effective for traditional DNNs, but that has not been applied yet to generative AI.

The main contributions of this work are:

- The analysis of model-distributed inference as a viable approach for the edge.
- A discussion of large language models and their main characteristics, pinpointing the issues in their deployment at the edge.
- The implementation of MDI-LLM, with the discussion of all design choices taken during development.
- The introduction of more recent LLM architectures (Llama) by incorporating new mechanisms designed to improve inference performance.

CHAPTER 2

PREVIOUS WORK

This section contains an overview of the concepts that served as a starting point for the development of MDI-LLM.

The recent ten years have seen a massive increase in the number of publications related to AI, in particular Natural Language Processing (NLP). Still, the main contributions focus on improving the performance of such models by increasing the available resources, and their experiments are typically performed on high-end computer hardware (typically in the cloud).

In recent years, the Internet of Things (IoT) and edge computing have also asserted themselves as new and interesting scenarios that require processing massive amounts of data. Furthermore, the integration of machine learning and artificial intelligence approaches have been well-suited for processing data generated at the edge. These methodologies offer innovative analytical techniques to transform raw data into actionable insights, facilitating the extraction of meaningful interpretations from complex results.

2.1 Model-Distributed Inference

The general trend in deep neural networks (DNNs) during the last ten years has been to attempt to increase the model sizes to improve performance. While this approach can be valid, up to some extent, this resulted in a process that established the cloud as the only possible location where to deploy such models, as cloud computing resources are highly available and can

be scaled easily. For scenarios such as the edge, where it may be required to use such models to perform AI-related tasks, the established approach has been to delegate the heavy computation tasks involving deep neural networks to cloud servers, introducing high communication costs and latency alongside concerns related to data privacy and security.

As a result, studies have been performed to develop frameworks for applying DNN models at the edge. Researchers devised three main directions to achieve this goal: "data parallelism" [1; 2], "tensor parallelism" [3; 4], and "model parallelism" (with and without "pipeline parallelism") [5; 6; 7; 8].

Out of these three, tensor parallelism does not meet the requirements for being applied in edge scenarios, as it necessitates a massive communication overhead, which makes it only viable in cloud servers exchanging data through high-bandwidth connections [3].

Data parallelism is well-suited for applications of AI that involve processing of large amounts of data, such as computer vision. The idea is to split the data among the nodes of the network, which all contain the full DL model. This way, nodes will process a subset of the inputs and will be able to work in parallel. Supposing to have a fixed data set, this reduces the overall processing time by introducing parallelism at the expense of the communication cost required to transmit the input data to the nodes. Another important aspect is that this approach scales well, as the number of nodes is inversely proportional to the amount of data processed at each host.

In the specific case of inference, the application of this approach takes the name of "data-distributed inference" (DDI), and has been observed to bring relevant performance improve-

ments for models where the input size is smaller compared to the intermediate activation size. This approach, however, requires each node in the network to fit the whole model in memory, which is a constraint for deep neural networks running on devices with limited resources.

Due to the latter limitation, an alternative approach called "model parallelism" has been developed. In this case, the model is divided into chunks consisting of subsets of contiguous layers of the neural network, which then get assigned to different nodes in the network. As a result, each node will be responsible for processing its local layers and transmitting its outputs to the node containing the following layers, with the nodes forming a chain that gets traversed sequentially by each processed sample. When this approach is applied to inference, we talk about "model-distributed inference" (MDI) [6; 8].

It is possible to introduce parallelism in this system in the form of "pipeline parallelism", i.e., allowing the network to process more than one input each time, ensuring each node is always working on a different sample [7; 8]. This can be achieved by imposing that the first node in the chain, after processing one sample and transmitting the output activations to its successor in the chain, immediately starts processing the following input sample.

Compared to the previous approach, this one requires data transmission during processing to transmit the intermediate model activations.

This method has been observed to yield a better performance in terms of memory and network usage when compared to DDI in the case of models that have a large input size and a smaller activation vector size. It is essential to highlight, however, how the performance strongly depends on the network reliability and the ability to partition the model in a way

that ensures equal processing times at all nodes to prevent bottlenecks. Indeed, this research line is now focusing on introducing reliability in MDI systems by devising mechanisms that can account for unstable network connections and node failures [6].

For our specific use case, MDI is the only viable approach among the two, because of the size of large language models preventing them from fitting in the memory of typical edge devices.

Additionally, since LLMs are autoregressive, i.e., they require feeding the output back to the model input to generate long sentences, we devise a new approach towards pipeline parallelism, which we call "recurrent pipeline parallelism". Following this method, it is possible to perform large language model inference efficiently over a network of edge devices with limited resource availability.

2.2 Large Language Models overview

This section will provide an overview of Large Language Models (LLMs), focusing on the characteristics that had an impact on the design of MDI-LLM.

Large language models are a type of deep neural network (DNN) which has been designed for text generation. They are decoder-only transformer-based neural networks, following the architecture first described in [9].

The "unit of information" in LLMs is the "token", a short sequence of characters in which text can be broken up. Section 2.2.1 describes tokenizers and the tokenization process.

This type of neural network receives as input a string (typically referred to as "prompt"), and produces at the output a new token that is appended to the current sentence to produce meaningful text. Once this new token is added to the sequence, the latter will then be fed back

into the input of the LLM for a new forward pass through the network, to produce another token.

This process can be stopped either by the user, i.e., specifying a maximum number of tokens, or when encountering special tokens, e.g., the "end of sentence" (EOS).

The use of the attention mechanism makes these models able to learn and reproduce the relationships between different elements in a sentence, resulting in a remarkable ability to generate novel text which has the same characteristics as the training samples.

2.2.1 Tokenizers

The first step to use a transformer-based model is always *tokenization*. *Tokenization* consists of translating a piece of text (or any other type of data that one may want to feed to a transformer model) into a sequence of integer values, which can then be used in a mathematical model such as an LLM. This can be achieved through a *tokenizer*, which takes as input the text that needs to be tokenized and splits it into "tokens", corresponding to groups of characters associated with a specific integer value.

All the possible different tokens make up the *vocabulary*, and the number of tokens is the *vocabulary size*.

Many different types of tokenizers exist in language processing.

The simplest ones are the *character-level* tokenizers, which associate every single character to a different token. Due to their simplicity, the performance of generative models for text that use this type of tokenizer is typically poor since the relationships among individual characters in a sentence are generally complex and difficult to generalize when performing neural network

training (e.g., think about how knowing two consecutive characters in a generic piece of text does not give any context to the overall sentence).

More sophisticated implementations are based on “Byte-Pair Encoding” (BPE), an algorithm created by Philip Gage, dating back to 1994 [10]. Following this method, it is possible to build tokenizers based on the statistical characteristics of the text on which the tokenizer is trained. Indeed, after specifying a desired vocabulary size, the tokenizer will analyze a training text sample and will create tokens by grouping the most common sequences of characters. By tuning the number of tokens, it is possible to find a good compromise between an extensive vocabulary, which makes tokenization long and increases the number of parameters in the model, and a good enough performance of the LLM.
State-of-the-art models use BPE tokenizers with a vocabulary of tens to hundreds of thousands of tokens [11; 12; 13].

2.2.2 LLM structure

The input of an LLM consists of a sequence of integer numbers obtained through the tokenization of the input prompt, as the network is still a mathematical model working with numbers.

The first stage of any LLM is the token embedding. Each token is substituted by a fixed length (the “embedding dimension”) vector at this step through a lookup table learned during training.

In addition, the input undergoes positional encoding. This step aims to include the information of each token’s position in the sentence in its embedding by performing specific operations

on each embedded token. Typical operations consist of adding vectors or multiplying matrices with specific values (that encode the position information) to the embedded tokens. Some examples of positional encoding implementations used in state-of-the-art transformer models use the sine and cosine functions to produce vectors that are added to the input embeddings, as in the original implementation [9]. Using such functions ensures boundedness (in the range $[-1,\ 1]$) and changing the "sampling rate" of the sinusoids, uniqueness of the values for each of the positions in the considered context.

It is also possible to make positional encoding *learnable*, for example by evaluating the vector containing the positional information through a trainable lookup table in which each entry corresponds to one specific position in the context ("trainable positional encoding"). This approach is more common in recent LLM architectures, such as GPT [14], as it is more flexible and generally yields better performance.

Another approach proposed in the literature, which took over the world of LLMs (primarily open-source ones) in the last couple of years [15; 12], is to use rotary positional embeddings (RoPE) [16]. With this type of embedding, it is possible to encode both absolute and relative position information in the input vectors through the multiplication with fixed (i.e., non-trainable) rotation matrices.

An important aspect of RoPE is that its operation is embedded in the attention evaluation to maximize efficiency. This implies that this type of positional encoding is done at every attention layer and not just at the network input (as specified by the original transformer architecture).

The network's core is the "attention block", reported in figure Figure 1. Note that the figure shows the structure of the attention block for the GPT architecture (and all the following transformer-based LLMs), which differs from the original description of the transformer [9] in that the layer normalization is performed before each of the two block steps.

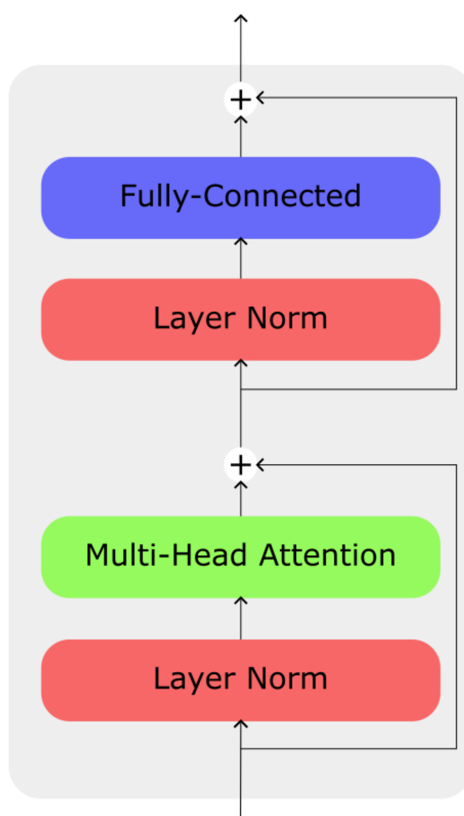

Figure 1: Attention block structure (GPT architecture and following)

Decoder-only transformer-based neural networks typically contain multiple of these blocks in a cascade following the initial embedding stage.

Both the input and the output of each block are vectors of length equal to the embedding dimension, allowing to scale up on the number of blocks to obtain bigger (and, up to some extent, better performing) LLMs.

Disregarding the layer normalizations, which are introduced to improve the training effectiveness and the overall performance [17], each attention block is composed of a *multi-head attention (MHA)* layer (possibly including RoPE), followed by a small fully connected network (typically, two layers), both surrounded by residual connections with the corresponding inputs.

The attention layers consist of multiple "attention heads" that work in parallel on different non-overlapping sub-vectors of the input. The operations performed in each head, as described in [9], are:

- Evaluation of the *Query*, *Key* and *Value* matrices on the input $\mathbf{X}_i = [\mathbf{x}_0, \ldots, \mathbf{x}_{\text{head_dim}-1}]$ of each attention head i:

$$\mathbf{Q}_i = \mathbf{W}_{i,Q}\mathbf{X}_i \tag{2.1}$$

$$\mathbf{K}_i = \mathbf{W}_{i,K}\mathbf{X}_i \tag{2.2}$$

$$\mathbf{V}_i = \mathbf{W}_{i,V}\mathbf{X}_i \tag{2.3}$$

  where $\mathbf{W}_{i,Q}$, $\mathbf{W}_{i,K}$ and $\mathbf{W}_{i,V}$ are model parameters for head i.

- Evaluation of the scaled causal self-attention at each head i:

$$\mathbf{h}_i = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\text{Mask}\left(\mathbf{Q}\mathbf{K}^T\right)}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.4}$$

  where "Mask" performs triangular masking enforcing causality, i.e., the attention score for each token is evaluated only with previous tokens in the sequence and the scaling factor d_k is the dimension of the key vectors. The Softmax function, instead, is applied on each row of the resulting matrix and will produce a probability mass function indicating the "similarity" of each of the Keys (rows of $\mathbf{K}$ with every Query vector rows of $\mathbf{Q}$.

- Stacking the result of each attention head in a vector with the same length as the input (embedding dimension):

$$\mathbf{H} = [\mathbf{h}_0, \ldots, \mathbf{h}_{\text{head_dim}-1}] \tag{2.5}$$

- Forward pass through a single linear layer.

The feed-forward network that serves as the second step in the transformer block is typically composed of two layers. The first layer has a dimension of $(4 \cdot \text{embedding_dimension})$, as in [9], while the output layer has the same dimension as the input, i.e., the embedding dimension, that keeps the output dimension equal to the input one.

After the cascade of transformer blocks (and after another layer normalization), the output activation vector is passed through a single fully connected NN layer, with an output dimension equal to the vocabulary size. Then, the softmax function is applied to the elements of this

vector, i.e., for an output vector $\mathbf{z} = [z_0, \ \ldots, \ z_{\mathcal{V}-1}]$, being $\mathcal{V}$ the vocabulary size, to obtain a probability mass function (PMF) over the token space:

$$\text{Softmax}(\mathbf{z}) = \left[\frac{e^{\frac{z_0}{\tau}}}{\sum_{i=0}^{\mathcal{V}-1} e^{\frac{z_i}{\tau}}}, \ \ldots, \ \frac{e^{\frac{z_{\mathcal{V}-1}}{\tau}}}{\sum_{i=0}^{\mathcal{V}-1} e^{\frac{z_i}{\tau}}} \right]^T \tag{2.6}$$

In Equation 2.6, the parameter τ (a strictly positive value) is the "temperature" and it is used to control how smooth the output is. For a low value of τ, i.e., $\tau \in (0, 1)$, the final vector will be more "concentrated" on the largest value of $\mathbf{z}$, i.e., that value will have a much higher probability compared to all others, while, as τ increases, the distribution will tend to be uniform.

The next token will be obtained by sampling the produced PMF.

2.2.3 GPT-2 architecture

GPT-2 indicates a family of four models that range from 124 million to 1.5 billion parameters, presented by OpenAI in 2019 and described in [11]. These models have been chosen for the implementation of Model-Distributed Inference due to their small size and identical architecture to newer models such as GPT-3. Indeed, the models of the GPT-3 family are simply a scaled-up version of the GPT-2 ones, with up to 175 billion parameters. As a result, by building a scalable MDI architecture, it should be possible to port the larger GPT-3 models to this framework with minimal effort (just by adding nodes to the network).

The general structure of this LLM is the same one reported in 2.2.2, i.e.:

- Input embedding.

- Positional embedding.
- Cascade of transformer blocks.
- Output linear layer (with softmax function).

Figure 2 reports the full model structure.

Some specific aspects of the GPT-2 architecture are the following.

The tokenizer uses a vocabulary of 50257 tokens, and the maximum context length is 1024. Positional embedding is performed through a trainable lookup table that maps integer values corresponding to the position of each token to the positional embedding vectors, that are then added to the token embeddings. The activation function used in the feed-forward layers is the GeLU function [18].

The different "flavors", i.e., the models of the GPT-2 family, differ in the number of transformer blocks, embedding dimensions, and the number of attention heads per Multi-Head Attention layer. The number of attention heads is chosen to keep the length of the sub-vector processed by each attention head, i.e., the embedding dimension divided by the number of attention heads per MHA layer, constant (equal to 64, in the case of GPT-2).

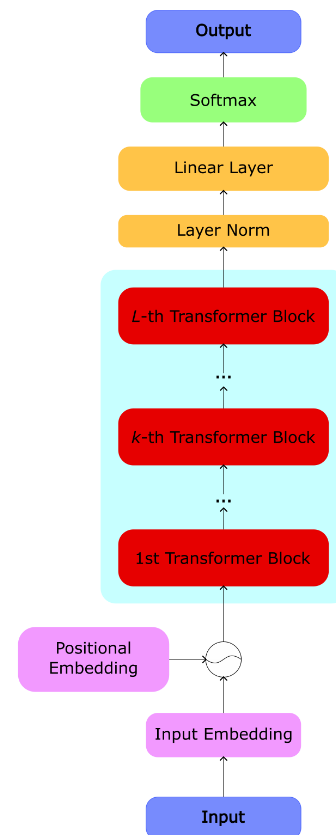

Figure 2: GPT-2 model structure

2.2.4 Llama Architecture

Llama is an open-source large language model architecture first made public by Meta in February 2023 [15]. Since its release, this architecture has gained popularity, becoming the state-of-the-art model family for both researchers and individuals who want to deploy performant LLMs on their own hardware. In July 2023, Meta released the Llama 2 family [12], which did not only include foundation models, i.e., models that only produce text by adding words to a given prompt, but also fine-tuned versions that could be used for chat. As of the time of writing, the last release of this model family is Llama 3 [19], which has been observed to be the most performant open-source LLM today on many benchmarking tests.

The Llama family includes models spanning 7 billion to 70 billion parameters. This design flexibility enables the models to be executed efficiently on diverse computing architectures, facilitating their deployment on various hardware platforms.

One of the main characteristics that makes these models stand out is their exceptional performance when compared to bigger models. Indeed, thanks to a combination of structural features, training corpus, and tokenizer used, a small model such as Llama 3 8B is able to achieve the same, if not better, performance as GPT 3.5 turbo (the model behind Chat GPT). The 70B Llama 3 model has also been observed to perform better than GPT 4 in some benchmarks.

The architecture of the Llama family has remained mostly unchanged across its different iterations, with changes mainly focusing on the training corpus data size and on the tokenizer (from Llama 2 to Llama 3). Compared to the GPT architecture, Llama 3 uses:

- SwiGLU [20] activation function instead of GeLU [18];

- Rotary positional embeddings (RoPE) [16] applied at each attention layer instead of absolute positional embeddings (applied at the model input);
- Root-mean-squared layer normalization [21] instead of layer normalization [17];
- Grouped-query attention (GQA) [22] instead of standard multi-head attention (only in Llama 3 and Llama 2 70B and larger);

Additionally, the context length of the Llama models ranges from 2048 tokens (Llama 1) to 4096 in Llama 2 to 8192 in Llama 3.

Discussing tokenizers, the Llama 1 and 2 model families have a 32k-token vocabulary, while Llama 3 introduced an improved tokenizer using 128k tokens. Consequently, the new model family performs better in the most common text-understanding benchmarks at the cost of a slight increase in the number of parameters (for the smallest model, from 7B to 8B) due to the need to accommodate a more extensive vocabulary.

2.3 LLMs at the Edge

The phenomenon of edge computing has arisen during the last decade. With the popularization of IoT and the increase in computational capabilities of the computers we commonly use in our everyday life, alongside the introduction of computer systems in common appliances and in cars, it has now become clear that empowering such technologies with deep learning capabilities can lead to significant benefits and quality of life improvements for the users.

Due to the need to process high amounts of data produced by edge devices alongside the high complexity and resources required to run deep learning models, specifically neural networks, the

state-of-the-art approach to tackle this scenario is to offload the computation to cloud servers, where resources are highly available.

Using the cloud, however, introduces new challenges and concerns [23]. First, transmitting data to and from the cloud results in an increased latency, which can be problematic for time-sensitive applications, such as autonomous driving.

This issue is further aggravated if the physical distance of the cloud is considerable, or if the network is congested. Another complicating factor is the type of data that needs to be exchanged with the cloud, as, for example, images or video streams drastically increase the transmission overhead.

System scalability is also a critical point in this scenario. Indeed, as the number of edge devices or the amount of data to be processed increases, the amount of cloud resources and network bandwidth should scale. This aspect is especially relevant for networking, as cloud resources are typically purchased on demand.

Data privacy and security are among the most concerning problems for some types of applications. By transmitting private data over the internet, the users technically lose control over it, and sensitive information can potentially be intercepted or misused.

In some other cases, it may even happen that it is impossible to ensure a reliable connection from the edge to the cloud, such as in the case of drones and UAVs, or sensor networks installed in isolated locations.

Due to these challenges, new approaches to deep learning have been introduced in the literature, and edge computing has proven to be the most suitable direction to follow. Still, it

is not as simple as moving the computation closer to the data sources, as the intrinsic nature of the edge provides some challenges as well.

The main issue in this scenario is to be able to accommodate the high resource requirements of deep learning on less powerful edge devices [23]. To tackle this limitation, research is focusing on developing new frameworks for the deployment of machine learning and AI models over networks of devices with low resource availability, with some examples already discussed in 2.1.

In the specific case of LLMs, though, the main line of work that is currently focusing on enabling their deployment closer to the edge is model quantization [24; 25].
Following this approach, it is possible to reduce the representation precision, in terms of the number of bits per parameter of the model weights, dramatically reducing the overall memory footprint required to run the language model and, as a bonus, making inference faster. This approach, however, is not scalable, as the representation precision of the model parameters cannot be reduced indefinitely, and the number of parameters used by the model remains the same.
Additionally, it has been observed how reducing the representation precision excessively can cause performance degradation [26].
Another important aspect is that not all hardware supports operations at lower precision, implying that, for some devices, model quantization will be inefficient as it will require representation conversion when processing the model – the weights will be stored in memory using a low number of bits, but the operations will require conversion to full-precision formats.

Other approaches that are currently being explored are knowledge distillation [27], in which smaller LLMs are trained to emulate larger, more performant ones, and model pruning [28], which defines rules to reduce the model size by deleting portions of it.

Still, most research that is being conducted in the field of LLMs aims at improving the models' performance by increasing their size and complexity, with the assumption that such models will only be deployed in the cloud.

Not much work has been done so far to deploy LLMs at the internet edge by spreading the computational load among devices in a network. The few attempts at distributing the computation in transformer-based models were either related to the introduction of parallelism in cloud servers to optimize training and inference times while still relying on high-end hardware and available resources allowing for model parallelism [3], or just concentrated on deploying LLMs on powerful consumer-grade hardware, still more performant than average edge devices, and relying on stable network connections [5].

This work aims to provide a proof of concept for the deployment of transformer-based models at the internet edge, demonstrating how it may be possible to avoid the limitations of the cloud in scenarios where latency or data privacy may be of interest, or the absence of a reliable network infrastructure connecting the edge of the cloud would prevent accessing additional computer resources.

We believe this approach to be exceptionally viable as it is scalable. Indeed, by adding more devices to the network, it is possible to increase the size of the shared memory pool, enabling the deployment of larger models.

We will specifically focus on LLMs (GPT-2, Llama); however, being the transformer architecture used in most generative AI applications, we believe this framework can be applied to different use cases.

The ability to utilize such powerful architecture far from the cloud opens up to new possible applications of generative AI and deep learning that have not been implemented before due to the lack of frameworks. Some examples are confidential healthcare applications, AI-based private personal assistants, edge processing in IoT, private AI-based smart home management systems and improved processing in drone fleets, to name a few.

CHAPTER 3

MDI-LLM

This chapter contains a description of the developed framework for the implementation of Model-Distributed Inference on Large Language Models: "*MDI-LLM*". We will follow the evolution of the application, concentrating on the reasons that drove the design choices taken during the development.

The initial implementation was based on NanoGPT [29], a scaled-down version of GPT-2. Having tested this first version of MDI-LLM, we moved on to porting GPT-2 [11] to this framework. Subsequently, we focused on adding support for the Llama model family, concentrating on the TinyLlama model, which has the same architectural characteristics of the Llama 2 model family, but with a reduced scale, using just 1.1 billion parameters (comparable with the largest GPT-2 variant). We referred to its implementation provided by the LitGPT project [30], a framework for deploying multiple LLM families built with an eye to the simplicity of NanoGPT, allowing us to use other models as well, e.g., Mistral 7B [13].

Switching to bigger models brought some challenges due to the different number of parameters and vector sizes, requiring minor architectural modifications. Additionally, since the Llama model family includes additional mechanisms, such as KV caching and RoPE embeddings, we also developed solutions to apply them to a distributed scenario and make them compatible with recurrent pipeline parallelism.

3.1 Testbed

The testbed used for the development of MDI-LLM consisted of a network composed of three Nvidia Jetson TX2 devkit boards connected via gigabit ethernet.

These embedded systems have 8 GB of shared RAM (i.e., accessed by both CPU and GPU), a 256-core GPU based on the Pascal architecture working at a maximum of 1.3 GHz, and two CPUs, a dual-core Nvidia Denver and a quad-core Arm Cortex A57, both clocked at 2.0 GHz peak [31]. These specs guarantee the boards a good performance in AI tasks (Nvidia estimates 1.33 TFLOPS), which is in line with common smartphones of today.

The operating system is a custom Linux distribution ("Linux4Tegra") based on Ubuntu 18.04 LTS. The application was developed in Python 3.8, using PyTorch 1.12, the latest version supporting CUDA 10.2 running on the boards.

Notice that these devices do not support operations in half-precision 16-bit floating point, which is a relevant constraint considering most state-of-the-art open-source models are deployed in float-16 of bfloat-16 (a format specifically designed for machine learning models, using 16 bits); however, this makes for a good representation of a "worst-case scenario" when dealing with edge devices. Loading the models in half-precision (float-16) is still possible, but the operations performed when using them require conversion to 32-bit representation, which introduces a slight overhead.

Unless explicitly stated, we will always use full-precision (32-bit floating point) representation throughout our analysis.

3.2 Rationale

The idea behind the MDI-LLM framework is that of integrating model-distributed inference with pipeline parallelism as described in [6] with state-of-the-art LLMs, providing a user-ready implementation that was able to achieve good performance by incorporating the benefits of the MDI framework.

To our knowledge, model-distributed inference at the edge has not been applied to recurrent or autoregressive models yet, hence our focus for this work is to devise "recurrent pipeline parallelism" in order to achieve parallel computation and maximize the system efficiency when working on many samples, i.e., pieces of text at the same time.

The underlying idea for distributing the model is to partition the transformer layers in a balanced manner among the network nodes, then set up fast communication channels to allow them to exchange intermediate model activations, forming a "ring" overlay network so that the model outputs can be fed back to the input, as required by LLMs.

Model partition should be performed in a way that ensures equal processing time at all nodes so as to prevent bottlenecks in the generation. We, however, do not specifically focus on providing algorithms to partition the model, as the literature contains several examples [32; 6]. Our focus, instead, concerns finding the best partition to ensure message transmission does not hinder the overall generation process. Indeed, in scaling up the deployed models, we have observed how ensuring a balanced message size is vital to achieving competitive performances when comparing our system to running LLMs on a single device, as message transmission can quickly become the bottleneck in a system such as ours.

3.3 Distributing NanoGPT

NanoGPT is a basic implementation of OpenAI's generative pre-trained transformer architecture provided as an open-source project by A. Karpathy [29]. It is a basic decoder-only transformer model based on the paper [11] released alongside GPT-2.

This model is a good starting point for the development of MDI-LLM, since it provides a straightforward implementation of the LLM without the overhead that is usually found in more complex and production-ready libraries. Thanks to this characteristic, it is also possible to access to some configuration parameters that usually hide behind implementation abstractions, such as the number of transformer blocks or the dimension of the embeddings. This grants total control over the models, allowing the creation LLMs of any size. In particular, it is possible to define models so small that they can fit in memory-constrained devices such as the Nvidia Jetson TX2 boards used as a testbed for this implementation to provide a baseline for the performance analysis of our implementation.

The three specific models used at this development stage have the following characteristics:

- Embedding dimension: 384;
- Context length: 128 tokens;
- Attention heads per MHA layer: 6;
- Head dimension: $384/6 = 64$;
- Number of attention blocks: 7, 9 or 12;
- Number of parameters:

- 7 blocks: $12,438,977$;
- 9 blocks: $15,985,601$;
- 12 blocks: $21,305,537$;

As a result, the model sizes in memory will be a few tens of megabytes, allowing not only to fit them in a single device, useful for benchmarking reasons, but also to train them reasonably quickly.

All these models were trained on the “Tiny Shakespeare” data set [33], published by A. Karpathy himself in 2015. This data set consists of the concatenation of all of Shakespeare’s works, resulting in approximately 1.1 MB of pure text (40,000 lines).

The tokenizer used is a character-based one, which was trained on the data set itself, i.e., the possible tokens (characters) are only the ones that appear in the text in the first place, resulting in a vocabulary size of 65 tokens.

3.3.1 Model partition

The logic behind the model partition was to start from the general model structure and naively divide it into separate portions to be assigned to a different device each. Each node will then receive the input for its local model chunk from the previous node in the communication chain, and it will process it to transmit its output to the next node.

As a result, inference requires a sequential flow of messages from the first node in the network to the last. On top of this, due to the autoregressive nature of LLMs, the output information will have to be fed back into the input to generate another token.

Figure 3 reports the partition scheme, referring to the same structure discussed in 2.2.3.

Notice that despite the testbed used in practice being composed of three devices, the model partition was implemented with scalability in mind, allowing users to potentially add as many "intermediate" nodes as required to fit any model size.

There are three node types in the architecture: the "starter", the "intermediate" and the "finisher". The first and the last are required (meaning there should always be at least two nodes in the network), while the intermediate nodes are optional, and adding more allows the application to scale. The starter node initiates the inference transmission, as it contains the first layers of the network (embeddings and a few transformer blocks). It also accepts the "logits", i.e., the output of the final linear layer, from the finisher node, on which it applies the *softmax* function to obtain the output probability mass function that is then sampled to select the next token in the sequence.

The intermediate nodes contain a set of transformer blocks (typically more than the starter node), allow-

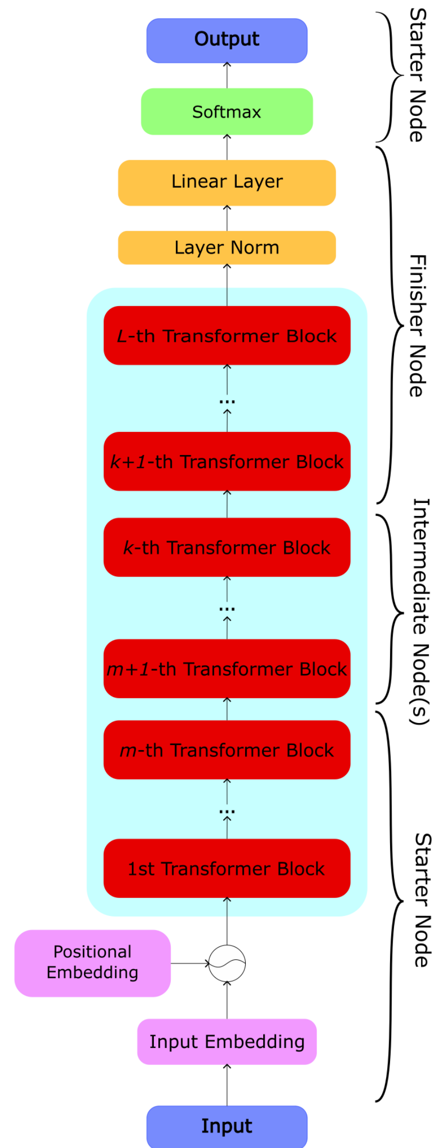

Figure 3: NanoGPT partition scheme

ing more nodes of such type to be added when the model size increases. Indeed, as discussed in 2.2, scaling up a model of a given architecture family means either increasing the number of attention heads or adding more transformer blocks. In our application, if the model size grows, it is enough to add nodes and redefine the model partition to allow the LLM to run distributedly.

The finisher node's local model chunk is composed of a few transformer blocks, alongside with the output *layer normalization* and linear layer producing the logits.

Notice how this partition scheme results in the size of the messages being different between the nodes. It should be kept in mind that the intermediate activations of the network are tensors of size S, where S is:

$$S = (\text{output dimension}) \cdot (\textit{current} \text{ context length}) \tag{3.1}$$

Here, (output dimension) is the vector dimension at the output of the current layer, and (*current* context length) is the number of tokens that are currently in the context, i.e., $\min((\text{context length}), (\text{n. tokens in current phrase}))$.

An important aspect to keep into consideration throughout this whole analysis is how the messages *grow in size* during generation until they reach a maximum value, given by:

$$S = (\text{output dimension}) \cdot (\text{context length}) \tag{3.2}$$

This value corresponds to the case where the context window is full, i.e., the number of tokens in the current piece of text is greater or equal to the model's context length.

The activation vectors transmitted from the starter node to the first intermediate node, between intermediate nodes, and from the last intermediate to the finisher node are all outputs of transformer blocks and, therefore, all have size equal to the embedding length multiplied by the current context size. The message transmitted from the finisher node back to the starter node, instead, has a size equal to (*current* context length) · (vocabulary size), as it contains the output logits (since it will yield the output distribution over the token space). In the specific case of NanoGPT, which uses a character-level tokenizer having a vocabulary size of 65 tokens, the message size ends up being smaller than the activations transmitted between the other nodes. Upgrading the tokenizer to any state-of-the-art one, such as the GPT-2 tokenizer, will affect the size of this vector. As a result, we can anticipate this will be a critical aspect of the architecture when porting GPT-2 to the MDI framework – see 3.4.1.

At this stage of the development, the layer assignment for each node was done intuitively, without any quantitative time analysis, to qualitatively split the computation load fairly among the devices.

3.3.2 Recurrent pipelining

The relevant contribution of this work is the theorization and implementation of *recurrent pipeline parallelism*, a technique used to improve the generation rate of LLMs running in a model-distributed setting. Through this approach it is possible to achieve computational parallelism when performing inference, i.e., generating text.

The rationale is to minimize as much as possible the idle times of the network nodes by ensuring they are processing data at all times. This can only be achieved when we deal with more than one sample at a time, i.e., the batch size is greater than one – in the case of large language models, the term "sample" refers to an independent piece of text. Indeed, due to the need to feed the output back into the input to generate a new token, only working on one sample would imply that only one node is actively processing such sample at any time instant, with all the others waiting. By raising the number of generated samples at inference stage, it is possible to have each computer process a different one at any time, provided the processing time of each node is the same. At each processing step, where "processing step" indicates the local processing of one sample at every node, the network hosts will process a different sample through their local model layers; then, they will forward the output to the next node in the communication chain.

Let us consider the case of a three-node network generating three samples with recurrent pipeline parallelism. Figure 4 reports this scenario. The arrows represent the message flow through the network, otherwise referred to as the "inference direction".

At the beginning of inference, the starter node will process the first sample, while all other nodes are idle. Then, it will send the output associated with the first sample to the intermediate node and start processing the second sample. Assuming the time for a single processing step in all nodes is the same, as the starter node finishes processing the second sample, the intermediate node should finish processing the first sample. Then, the starter node will transmit the output

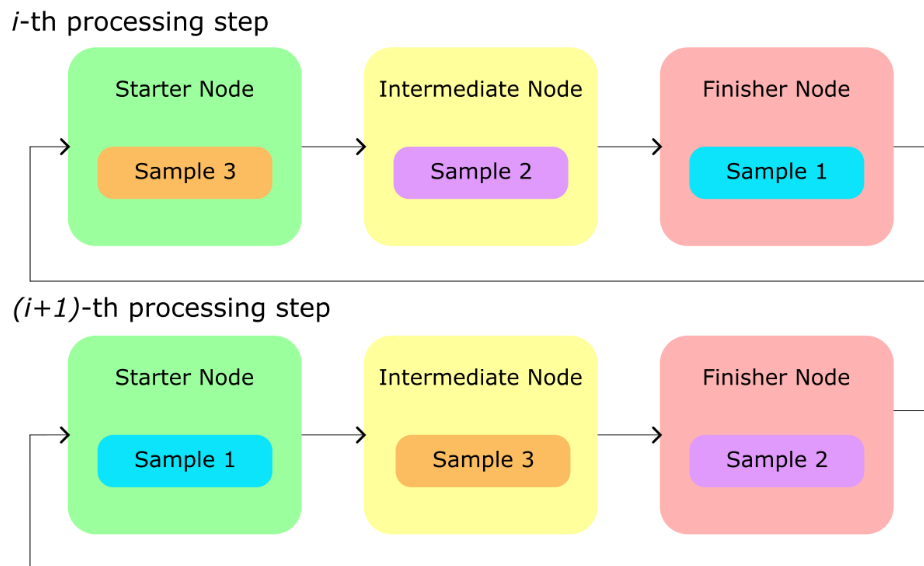

Figure 4: Recurrent pipeline parallelism – 3-node network

associated with sample number 2 to the intermediate node and start processing sample number 3 while the intermediate node will pass the output vector for sample 1 to the finisher node.

Again, let us highlight the importance of ensuring the same time duration for each processing step at every node, which requires optimizing the model partition. Thanks to some design choices that will be explained later, however, it is possible to mitigate slight disparities in processing times between the different nodes, which become inevitable when we factor in the effect of the random transmission delay introduced by the message exchange.

3.3.3 Practical implementation

This section deals with the practical details of the implementation of MDI-LLM. The need to translate the theoretical architecture into code has influenced many aspects of the application.

A fundamental aspect of the application is networking. To allow for message transmission, we must create an overlay network by setting up connections between the nodes in advance. This requires coordination, and our approach is that of using the starter node as a "central authority" to initialize all other network participants before launching the generation.

Initialization is performed through a secondary configuration channel running over HTTP. As a result, each node in the network also acts as an HTTP server, supporting different types of requests to accommodate various operations.

Before the initialization, the "non-starter" nodes (intermediate and finisher) do not know their roles, nor do they possess their model chunk. They are initialized through a POST request sent by the starter node, which informs them of their role in the network, assigns them a model chunk, transmits all the configuration parameters for the model and specifies their neighboring nodes (predecessor and successor) in the message transmission chain. In the case of NanoGPT, the model can be split at runtime by the starter node before being transmitted since its size is limited.

Upon receiving this information, each node will first check the integrity of the message, create the local model and load the parameters, initialize the connections with the neighbors, and start waiting for incoming messages to be processed.

The connections used for transmitting the intermediate activations have been designed with performance and reliability in mind. In practice, they have been implemented through Python sockets, running on top of TCP/IP. At the application layer, the implemented protocol is minimal to reduce the overhead while still counting on the reliability introduced by TCP. The

application layer message is composed of a fixed-length header, which only contains the current payload size in bytes, and the payload, containing a byte stream obtained by dumping the Python object containing the activation vector (a PyTorch tensor). Following this approach, upon receiving a message, each node can read precisely the number of bytes occupied by the received tensor, preventing issues related message truncation.

Throughout every run, the communication sockets between the nodes remain the same, preventing the time loss that would have resulted by re-creating them at each message exchange.

Implementing recurrent pipelining in practice requires handling and coordinating message transmission between the nodes. The idea is to ensure the receiver node is always expecting the message when the sender has to transmit its output. This requirement, however, is problematic, as message transmission and reception would block the processing loop, meaning that until the message is transmitted (at the sender) or received (at the receiver), the node will wait, resulting in undesired idle periods. In practice, this mechanism causes total blocking of the application, as it results in a situation where all nodes attempt the transmission of their current sample to the following one in the chain without any node being ready to receive the message.

To solve this major problem, we decided to implement an input message FIFO queue which gets filled on a secondary thread on each network node. The thread will, therefore, always be listening for incoming messages, unpack them, and place them in the queue. The main processing loop, instead of having to wait for the messages over the sockets will pop the oldest message from the queue and process it. Following this approach, the receiving side of message transmission will occur in parallel to the main loop, preventing blocking, and each sender will

always be able to transmit since the receiving side will always be ready. Figure 5 shows the updated architecture, including the queues.

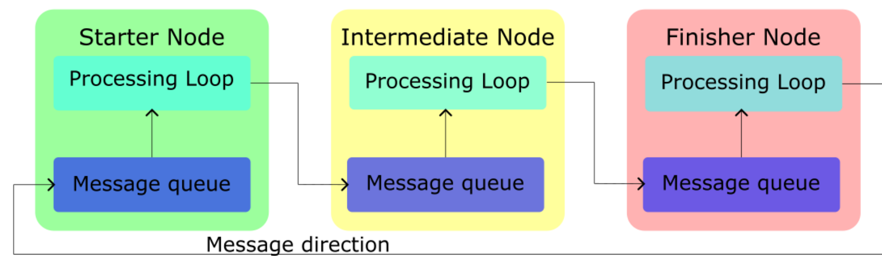

Figure 5: Implementation of input message queues

It can be noticed that this approach does not prevent short idle periods due to slight processing time differences between the nodes, as the main thread may still have to wait if the message queue is empty.

This solution proved very effective in streamlining the transmission procedure, and came with the bonus of an architecture that is more resilient to network delay jitter (which was the cause of blocking in the first place). Indeed, the queues help buffer in case some nodes are processing sample at a slightly faster rate.

The introduction of the message queues also opens up the possibility of processing a number of samples larger than the number of nodes, as it provides a way to temporarily store the

messages related to samples that are not currently being processed. This feature, however, was implemented only later – see 3.5.1.

By using the starter node as the network coordinator, it is possible to make it handle the generation loop to minimize the points of failure. Specifically, the starter node is the only one that knows the actual number of tokens that have to be generated, meaning that it is the only node whose processing loop has a finite duration. The other types of nodes will keep processing the samples as they arrive until the starter node sends a special termination message signaling the end of its generation loop. The only check the "non-starter" nodes perform is on the sample ID of each message, ensuring they process samples in the correct order (periodically, from sample 1 to sample N, where N is the number of nodes, equal to the number of samples in this case).

The use of different dependent threads in the application also requires particular care when terminating the execution. As anticipated before, the starter node transmits a termination message over the output socket once it finishes the generation loop. As the nodes receive this message, they will interrupt their main loop, additionally terminating the receiving queue thread and forwarding the same message to the following node in the chain. The starter node, in addition, also sends out a PUT request to advertise the end of the run. This mechanism was designed to allow the message sent over the sockets to indicate the end of the current generation loop, while the PUT requests to address the termination of the whole application, to logically separate these two aspects of the software.

3.4 Distributing GPT-2

Having created a first working implementation of MDI-LLM for NanoGPT, we could focus on scaling up the architecture and switching to the GPT-2 family of models. Since MDI-LLM was designed with scalability in mind, the transition to bigger models was almost seamless: it was enough to change the configuration parameters to the ones of the new models, as the underlying architecture, i.e., the type of layers and the network structure, remained the same. However, the increased model size raised some issues that required tweaking some aspects of the implementation. Additionally, we focused on additional improvements to optimize the code and make the system more reliable.

As discussed previously, the GPT-2 model family contains four different model variants ("flavors"), which just differ in scale. Table I reports the flavor-specific parameters (number of attention blocks, embedding dimension, number of attention heads and total number of model parameters).

Flavor	N. attn. blocks	Embed. dim.	N. heads	N. params.
GPT-2	12	768	12	124 M
GPT-2 medium	24	1024	16	350 M
GPT-2 large	36	1280	20	774 M
GPT-2 XL	48	1600	25	1558 M

TABLE I: FLAVOR-SPECIFIC MODEL PARAMETERS, GPT-2

The common features, instead, are a context length of 1024 tokens and a vocabulary size of 50357 tokens.

3.4.1 Architectural changes

As explained in previous sections, GPT-2 uses a tokenizer with more than fifty thousand possible tokens, unlike NanoGPT, whose character-level tokenizer uses few tens of tokens. As a consequence, in the former model, the dimension of the logits vector, i.e., the output of the last linear layer, is much greater compared to the latter ($52,000$ vs. 65).

Therefore, when deploying GPT-2 models on the MDI framework developed for NanoGPT, this results in a much larger size of the message transmitted by the finisher node to the starter, causing a heavy bottleneck due to the longer transmission time needed.

This issue required redefining the model partition scheme in order to ensure that messages of the same size were transmitted between the nodes. To achieve this, we decided to move the last layer normalization step and the final linear layer to the starter node, as depicted in Figure 6. In the same way the starter node was used to apply the softmax function on the logits received by the finisher in the initial architecture, now it evaluates the logits themselves, which means it receives from the last node in the chain a message containing the output of the last transformer block, i.e., having dimension $S = (\text{output dimension}) \cdot (\textit{current} \text{ context length})$, which is the same as the messages exchanged between the other nodes.

Notice how this new partition scheme effectively gets rid of the distinction between the intermediate and finisher nodes, as now they will both only contain a subset of the transformer blocks making up their local model.

These two node types are now replaced by "secondary" (worker) nodes, resulting in a simpler architecture and improved scalability. Indeed, now the system must have at least one node (the starter) to run against the two necessary nodes for the previous architecture iteration (starter and finisher), making the application more flexible and supporting standalone inference.

Alongside this improvement, required by the increase in model size, other features have been introduced in this version of MDI-LLM.

First, since message transmission was still performed during the main processing loop, this implied that it could have affected the performance in case the network slowed down temporarily. Introducing a mechanism analogous to that of the input queues at the transmitting side allows for mitigating this issue. Now, each time a node finishes processing a sample, it will simply place it inside an output FIFO message queue, where another program thread will

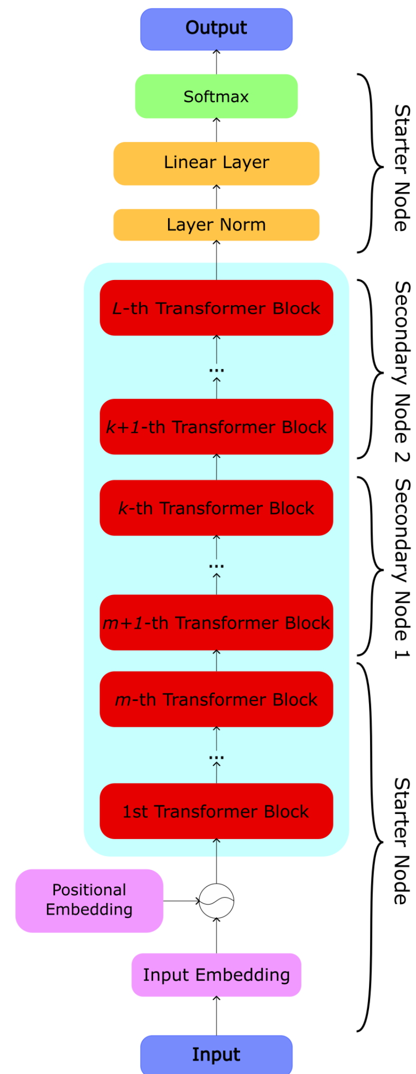

Figure 6: GPT-2 partition scheme

extract it in order and transmit it to the next node. The operation of this mechanism is shown in Figure 7.

Secondly, as anticipated previously, another possible improvement for the application is to allow for the generation of a number of samples higher than the number of network nodes. Introducing both input and output queues ensures samples that are not being processed can be temporarily stored while still achieving parallel processing. Additionally, since the nodes are "overwhelmed" by the samples, this ensures that, even if there are slight discrepancies in the local processing times between the nodes, there will always be samples to process in the main loop, as the generation will not have to wait for message transmission or reception thanks to the queues. Notice that increasing the number of samples reduces the benefits of parallel computation, as the samples have to wait before being processed. Still, since it is possible to prevent idle nodes, the overall system utilization is maximized.

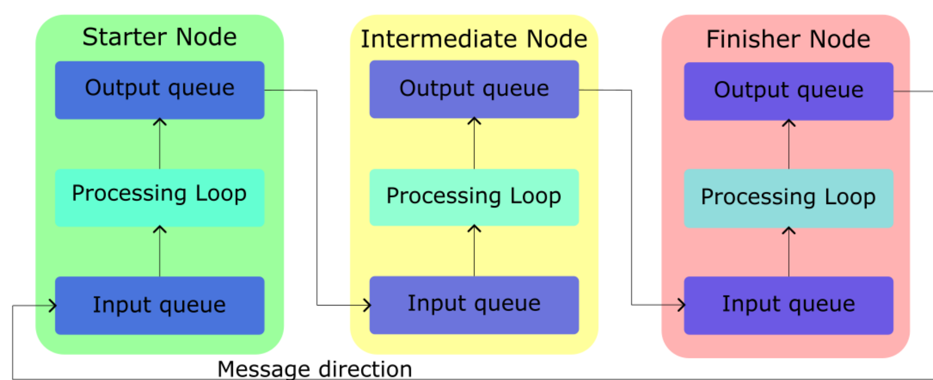

Figure 7: Implementation of input and output message queues

Allowing for more flexibility on the number of generated samples, it is also possible to set it to a value lower than the number of nodes, allowing us to observe the situation in which the network works inefficiently (i.e., at all times, some nodes are idle). Still, by distributing the model on multiple nodes, it is possible to let the LLM run on various devices that could not load all the model parameters in memory.

Another detail that has not been implemented so far is the support for user prompts. Until now, generation would start from a single "`\n`" token and would generate a predefined number of tokens. By adding prompt support, it is possible to provide an initial string, which will be tokenized and will be used as the beginning of the generated text to provide context.

3.5 Distributing Llama

The port of the Llama 2 model family was based on the LitGPT project [30], which implements today's most common open-source models. Despite our focus being Llama, specifically TinyLlama 1.1B and Llama 2 7B, the project also supports other interesting families, such as Llama 3, Mistral, and Falcon. This analysis will not focus on these families because their architecture is similar to Llama 2's, implying that the same design choices can easily be applied to support them as well.

Llama 2 includes a series of architectural changes when compared to GPT-2, allowing it to achieve better performance both in terms of LLM benchmarks and generation speed.

First, positional embedding is achieved through RoPE, which, as described previously, is a mechanism that is embedded in the self-attention layer. With RoPE, it is possible to use relative positional encoding, which has proven to be the best-performing encoding approach.

Secondly, in larger models of the Llama 2 family, grouped-query attention is used. This technique allows the sharing of the query matrices in the attention layer, reducing the amount of computation required at each forward pass in the model and speeding up inference.

Thirdly, due to the larger embedding dimensions used and to prevent repeating the same operations when generating every token, Llama uses KV caches. This means that, instead of re-evaluating the key and value matrices at each self-attention layer for all the token embeddings in the context, the model will keep them in memory, allowing the propagation of just the new token through the network at each subsequent forward pass. It is important to note that propagating all of the prompt tokens is still necessary to build the cache at the first forward pass.

The first two improvements could be implemented effortlessly in our framework, as they are compatible with the distributed setting and are not affected by how the system achieves pipeline parallelism.

KV caching, instead, required an adaptation since, in MDI-LLM, every node always processes different samples one after the other, as shown in Figure 4. Therefore, we devise a mechanism we call "rotating KV caches", according to which each node has to swap the currently active KV cache for the one associated with the sample it has to process. Choosing a suitable data structure to minimize this mechanism's overhead and avoiding reading and writing large amounts of data can allow the system to achieve very high generation rates, resulting in a functional implementation.

Our development was conducted using three LLMs with different memory footprints to make developing and testing easier. To provide a baseline, we trained a custom model we named “NanoLlama“using 304 million parameters. Due to its restricted size, running it on a single device to evaluate the performance improvements derived from using MDI-LLM was possible.

Then, we performed tests on TinyLlama 1.1B [34], a scaled-down version of Llama 2 7B, trained as a proof of concept to showcase the impact training data can have on the performance of LLMs, even if they are ”small” compared to state-of-the-art ones. In this case, the model footprint is comparable to GPT-2 XL (1.5B).

It was also possible to deploy Llama 2 7B. Since this model is much bigger than the others that we had analyzed so far, we were not able to fit it in the memory of our testbed devices (also considering the lack of support for half-precision representation on the Nvidia Jetson TX2 boards), and it was only tested on different hardware. For this reason, we choose not to provide results related to this model, as they would not be relevant to the scope of our research.

What follows is the description of the characteristics of the tested models.

- NanoLlama 304M
  - Maximum context length: 2048 tokens.
  - Embedding dimension (activation length): 1024.
  - Number of transformer blocks: 12.
  - Number of attention heads per Multi-Head Attention (MHA) layer: 16.

  - Vocabulary size: 32000 tokens.

- TinyLlama 1.1B
  - Maximum context length: 2048 tokens
  - Embedding dimension (activation length): 2048
  - Number of transformer blocks: 22
  - Number of attention heads per Multi-Head Attention (MHA) layer: 32
  - Vocabulary size: 32000 tokens

3.5.1 Final architecture of MDI-LLM

The final architecture of the system is reported in Figure 8. As noted previously, MDI-LLM requires at least one node (the starter) to run, resulting in a flexible framework that can also be used to deploy the models in standalone mode (single device).

Figure 9, instead, reports the layer partition scheme for the Llama architecture. The only difference compared to GPT-2 is that positional encoding is now embedded in the attention blocks, specifically in the multi-head attention layers.

3.5.2 Nodes operation

In this section, we will explain the actions performed by each node during the application's runs. The following assumes that all the nodes have already been configured, have loaded their own model chunk into memory, have instantiated the message queues, and have set up the communication channels with their own neighbors (i.e., predecessor and successor in the

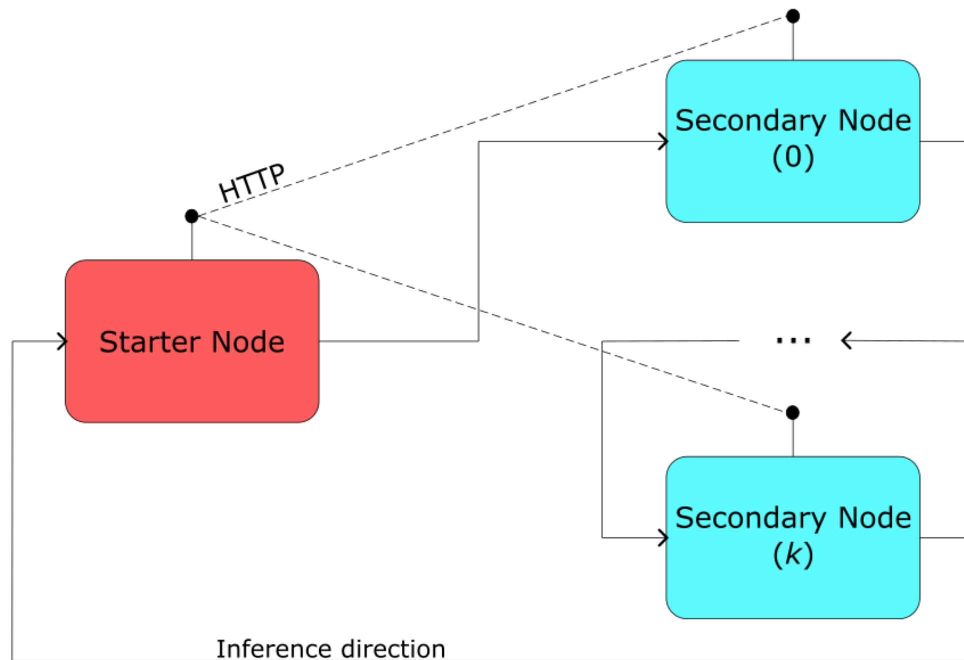

Figure 8: MDI-LLM architecture

communication chain), launching the threads used to handle message transmission (i.e., filling the input queue and emptying the output queue).

Algorithm 1 describes the main processing loop of the starter node. It should be remembered that this node knows how many tokens it has to generate for each sample; its generation loop will have a pre-determined duration, with a number of iterations equal to the number of processed samples multiplied by the number of tokens to be generated.

Also notice how the executed operations change based on the current iteration number. During the first iteration for each sample ($k \in \{0, ..., n_samples - 1\}$), the node has to initialize

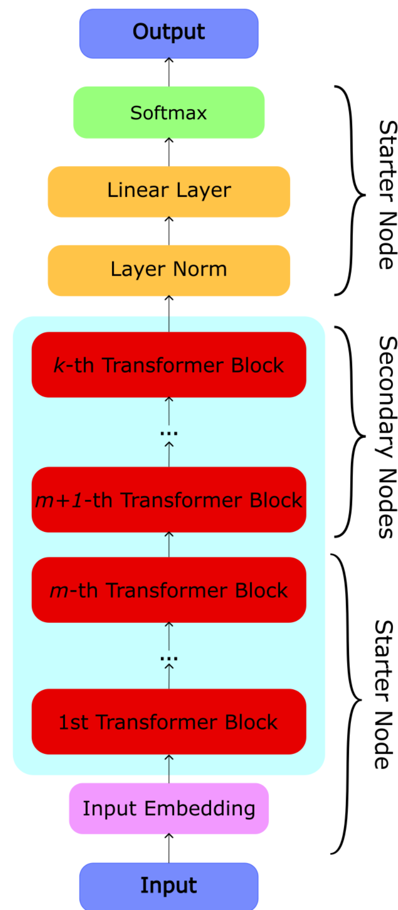

Figure 9: Layer partition scheme – Llama models

the KV cache, i.e., it allocates the memory to contain key and value matrices that will be used to speed up inference; then, it will process the mode input (embedding and local transformer blocks).

For the final iteration, instead, the node will only process the output layers (linear layer with softmax and probability mass function sampling) to obtain the last token.

```
Data: n_samples, n_tokens
Tokenize prompts (one for each sample);
Place input tensors in input message queue;
n_iterations ← n_samples × n_tokens;
for k = 0 to n_iterations do
    Extract sample s_n (n = k mod n_samples) from input queue;
    if k < n_samples then
        // First iteration for each sample: initialize cache
        init_KV_cache(sample = s_n);
    else
        // Process model output (final layers) - no need for KV cache
        Process output layers;
        Sample output probability distribution and get token;
        Append new token to the current sample s_n;
    end
    // Process model input (initial layers)
    if k < n_samples × (n_tokens − 1) then
        // If here, not in the final iteration for the samples
        Activate n-th KV cache;
        Forward sample s_n through input layers;
        Place result in output message queue;
    end
end
Send stopping message to nodes (over sockets);
Decode samples s_n ∀n ∈ {0, ..., n_samples − 1} with tokenizer;
Result: List of generated pieces of text
```

Algorithm 1: Starter node main processing loop

For all other iterations, the node will first process the output layers to obtain the new token, then pass this new output through the model input, giving start to tle following forward pass, before placing the activation result in the output queue to be transmitted to the next node.

The processing loop for secondary nodes, instead, is reported in algorithm 2. As discussed, secondary nodes are agnostic about the number of iterations. Their only task is to receive a

```
Data: n_samples
Initialize empty list of n_samples caches;
while Stopping message not received do
    Extract sample s_n from input queue;
    if n-th KV cache is None then
        // First iteration for current sample
        init_KV_cache(sample = s_n);
    end
    Activate KV cache n;
    Forward sample s_n through local model layers;
    Place result in output message queue;
end
```

Algorithm 2: Secondary node main processing loop

sample, activate or create the KV cache for it, and forward it through the local model chunk before sending it to the next node. This allows secondary nodes to focus only on processing, effectively acting as worker nodes, completely agnostic of the rest of the application. Once again, this characteristic was implemented with scalability in mind, allowing the effortless addition of extra nodes to support larger models.

CHAPTER 4

PERFORMANCE ANALYSIS

This chapter provides an in-depth performance analysis of the developed MDI-LLM framework for all the tested models.

The main metrics we will focus on are the token generation rate, i.e., the amount of tokens generated per unit of time and the memory usage of each device. We will also analyze the message size for each model and observe its impact on overall system performance.

We tested the models when running in standalone mode (one node), over two nodes and over three. For one of the considered LLMs (GPT-2 XL), however, it was not possible to fit the whole model in the memory of a single device; hence, we use it as a proof of concept of the memory usage reduction while still being able to compare the results to the other models of the same family.

4.1 Results – NanoGPT

The analyzed NanoGPT models have been described in 3.3. Table II and Table III report the partition scheme, i.e., the number of transformer blocks assigned to each node for the two- and three-node cases, respectively. It should be noted that this model was deployed on the first MDI-LLM version, which used an architecture involving three node types ("starter", "intermediate", and "finisher"). Hence, the finisher node also contains the output linear layer.

Model	N. blocks, node 1	N. blocks, node 2
7 blocks	3	4
9 blocks	4	5
12 blocks	5	7

TABLE II: LAYERS ASSIGNMENT – NANOGPT OVER 2 NODES

Model	N. blocks, node 1	N. blocks, node 2	N. blocks, node 3
7 blocks	1	3	3
9 blocks	2	4	3
12 blocks	3	5	4

TABLE III: LAYERS ASSIGNMENT – NANOGPT OVER 3 NODES

Figures Figure 10, Figure 11 and Figure 12 report the plots of the time versus the number of generated tokens for this model's variants (7, 9 and 12 blocks models, respectively). These plots were obtained by observing the generation of 2000 tokens in all three scenarios. For the 1-node case, we generated two 1000-token samples sequentially (as if the single device had to fulfill the request of two users); in the 2-node case, the system generated two samples of 1000 tokens each (to achieve parallel computation through recurrent pipeline parallelism in the 2-node case), while the 3-node line resulted from the generation of three samples of 1000 tokens each, with the plot being truncated at 2000 tokens since, in order to ensure proper pipeline parallelism, we need to generate at least as many samples as the number of network nodes.

In all three cases, it is possible to observe an increase in the generation rate when adding nodes to the system, proving how achieving parallel processing for the samples results in greater efficiency.

However, performance increases less when going from two to three nodes in the network than when going from one to two. Specifically, for the 7-block model, the generation rates are: 16.0 tokens/s for using a single node, 20.9 tokens/s for two nodes, and 22.3 tokens/s for using three, resulting in an increase of 30.6 % in going from one to two devices, but only 6.7 % in going from two to three.
For the 9-block model, the rates are 14.4 tokens/s (single node), 18.7 tokens/s (two nodes), and 20.7 tokens/s (three nodes), associated with performance increases of 29.9 % (one to two nodes) and 10.7 % (from two to three).
Last, for the 12-block model, the rates are 10.7 tokens/s (single node), 15.8 tokens/s (two nodes), and 17.9 tokens/s (three nodes). The associated performance increases are 47 % in using two devices versus one, and 13.7 % in going from two to three.

This lower improvement can be explained by considering the impact of the message transmission and the small model size in all these scenarios.

Indeed, since these models are already small (few tens of megabytes overall), by partitioning them into smaller chunks, the local model of each node will be even smaller, resulting in a low processing time, i.e., a low time to complete each local forward pass. At the same time, the size of the transmitted messages is the same in all cases, i.e., it will take the same amount of time.

Specifically, since these models do not use KV caching, the size of the intermediate activations (that make up the messages) will be, at most, equal to:

$$(\text{context length}) \cdot (\text{embedding dimension}) \cdot 4\text{ B} = 128 \cdot 384 \cdot 4\text{ B} = 196,608\text{ Bytes} \quad (4.1)$$

since we use 32-bit floating point representation.

This size is an upper bound, reached when the number of tokens in the generated sequence is greater than or equal to the context size (128). Indeed, as the number of tokens exceeds the context length, the system will truncate the input sequence to consider the last 128 tokens.

This aspect is clearly a limitation of this version of MDI-LLM, as the message size greatly impacts the overall generation speed. Theoretically, a higher generation rate could be achievable if it was possible to neglect the transmission effect. In any case, with the later introduction of KV caching – see 4.3, tackling the message size reduction will be possible.

In other words, when partitioning the model over more nodes, the processing time will become smaller than the message transmission time, making message exchange a bottleneck. Consequently, when working with small models, the improvements brought by MDI-LLM are only marginal. To maximize the system's efficiency, it is required to maximize the utilization of the network nodes, which is equivalent to maximizing the local chunk size – a scenario that will be analyzed more in detail when discussing larger models later.

In other words, adding nodes does not indefinitely improve the generation rate, as the processing time of the local model becomes overwhelmed by the transmission time.

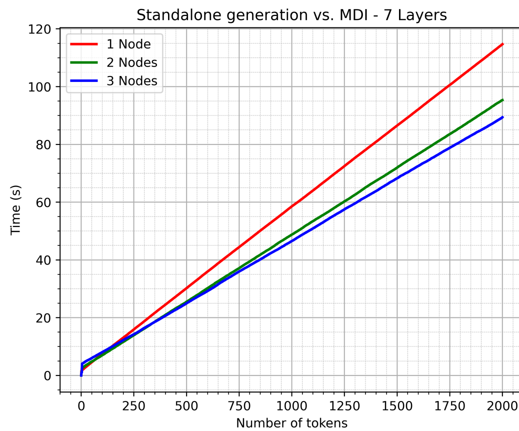

Figure 10: Time vs. N. of tokens – NanoGPT 7 blocks

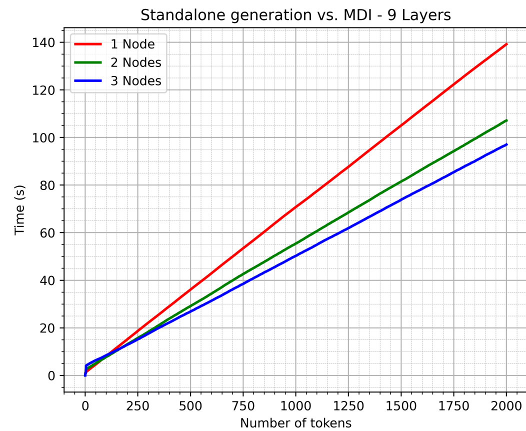

Figure 11: Time vs. N. of tokens – NanoGPT 9 blocks

We also included a detail of the behavior at the origin of the 12-blocks model in Figure 13, which shows a shape that can also be observed in the other plots. The step-like behavior at the very beginning can be attributed to the time required for the Torch library to initialize the internal model state in every node. Indeed, the "number of steps" equals the number of nodes for each line. This behavior is unavoidable in our setting, as the nodes can only initialize the model state once they receive the first sample to process since we assumed a realistic scenario in which the nodes do not know the samples they will work on in advance.

Another observation that can be made on the same image is that the lines associated with the multi-node cases are jagged, while the line for the standalone case is straight. This is clearly the effect of the variable transmission delay resulting from the variable network conditions, which causes the non-constant time required for transmitting one sample through the whole network

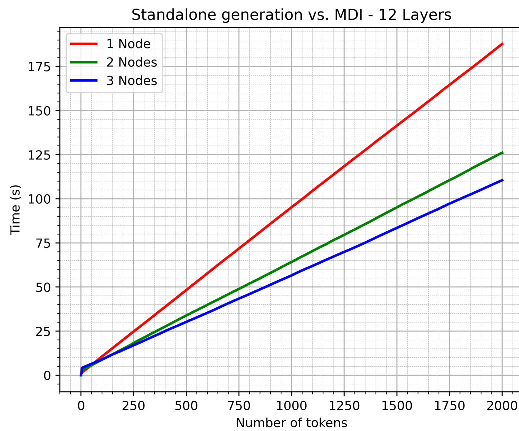

Figure 12: Time vs. N. of tokens – NanoGPT 12 blocks

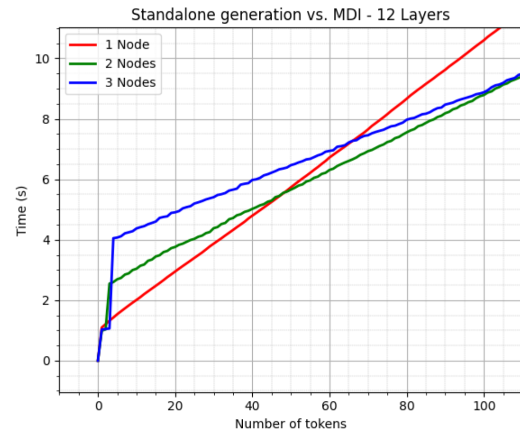

Figure 13: Time vs. N. of tokens – detail at the origin

to obtain each new token. It can also be observed that the three-node case produces a line with bigger “oscillations” due to the presence of an additional transmission step when compared to the two-node case, resulting in a higher variance (assuming the transmission delays to be independent random variables, summing them results in a random variable with a variance equal to the sum of the variances of the two). This behavior should be considered when scaling the system even further, i.e., introducing even more nodes; however, it can be mitigated by reducing the message size, as, typically, in communication systems, it is reasonable to assume the variance of the transmission delay to be proportional to the message size, since larger messages cause network congestion faster.

The other important performance metric for our system is the memory usage. Table IV reports the memory usage for each node in every configuration for the considered models.

	1 Node	2 Nodes		3 Nodes		
Node	1	1	2	1	2	3
7 blocks	991 MB	981 MB	874 MB	974 MB	865 MB	866 MB
9 blocks	1.07 GB	1.04 GB	899 MB	1.01 GB	874 MB	875 MB
12 blocks	1.17 GB	1.13 GB	934 MB	1.105 GB	899 MB	891 MB

TABLE IV: TOTAL (RAM + VRAM) MEMORY USAGE – NANOGPT

Notice how, in this case, the memory savings are marginal (a few tens of MB). This behavior is due to our working with very small models. In these cases, most of the memory used by the application on the nodes is actually devoted to loading the Python libraries and handling the connections (sockets for transmitting activations and HTTP server for configuration purposes). It is necessary to work with larger models whose memory requirements are higher than the application's "baseline" amount used to store libraries and connection information (only in the multi-node scenarios) to see relevant reductions in memory usage when adding nodes.

4.2 Results – GPT-2

This section reports the performance analysis for the GPT-2 model family. The four models that are part of this family (GPT-2, GPT-2 Medium, GPT-2 Large, and GPT-2 XL) have already been described in 2.2.3. In Table V and Table VI, we report the number of transformer blocks assigned to each node in the two- and three-node scenarios. Deciding the partition scheme in this case was easier, especially for the three-node case, thanks to the simplified partition scheme introduced when porting these models to the framework. Indeed, since the "non-starter" nodes all contain the same amount of model, i.e., only a sequence of transformer

blocks, assigning them the same number of blocks is enough to guarantee equal processing times among them since our testbed is made up of identical devices.

Model	N. blocks, node 1	N. blocks, node 2
GPT-2	5	7
GPT-2 Medium	10	14
GPT-2 Large	16	20
GPT-2 XL	22	26

TABLE V: LAYERS ASSIGNMENT – GPT-2 MODELS OVER 2 NODES

Model	N. blocks, node 1	N. blocks, node 2	N. blocks, node 3
GPT-2	2	5	5
GPT-2 Medium	4	10	10
GPT-2 Large	10	13	13
GPT-2 XL	14	17	17

TABLE VI: LAYERS ASSIGNMENT – GPT-2 MODELS OVER 3 NODES

To evaluate the generation rate, we generate two samples of 400 tokens each in the one- and two-node cases, while for the three-node case, we generate three samples and truncate the plot to 800 tokens to provide a fair comparison.

Figure 14, Figure 15, Figure 16 and Figure 17 contain the plots reporting the generation time versus the number of tokens for all models of the GPT-2 family.

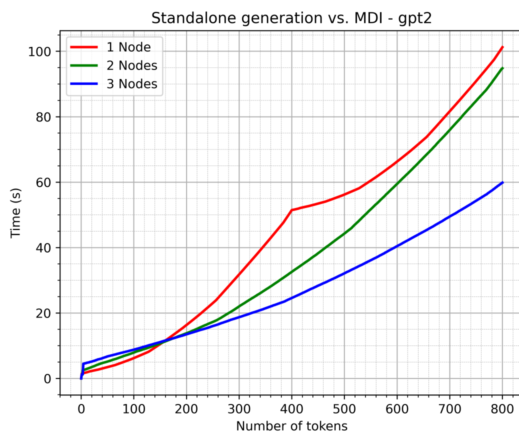

Figure 14: Time vs. N. of tokens – GPT-2

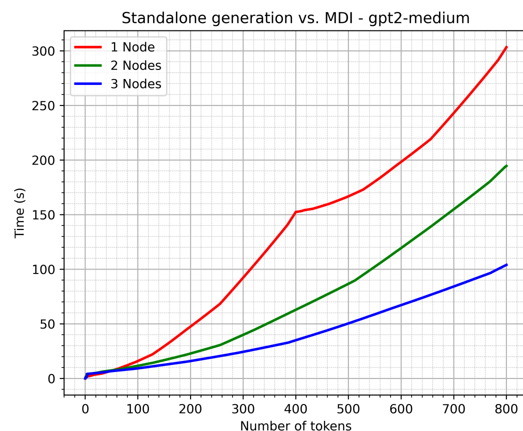

Figure 15: Time vs. N. of tokens – GPT-2 Medium

It should be noted that, due to its size, GPT-2 XL (1.5B parameters) cannot be fit in less than three devices using 32-bit floating point representation. While this proves MDI-LLM's ability to scale the model size with the number of network nodes, we cannot compare this model's performance in different settings. Still, comparing the results of different scenarios, it is possible to observe how GPT-2 XL achieves a better generation rate when deployed on three nodes compared to, for example, GPT-2 Large on a single node.

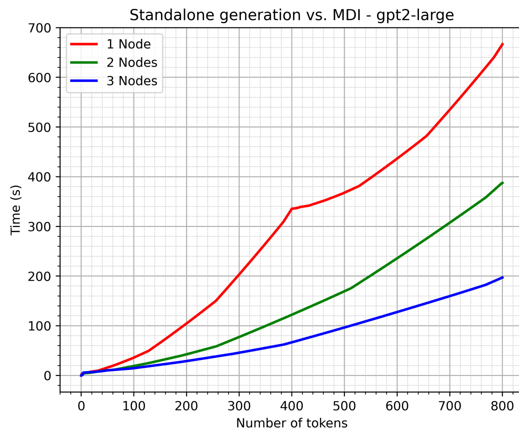

Figure 16: Time vs. N. of tokens – GPT-2 Large

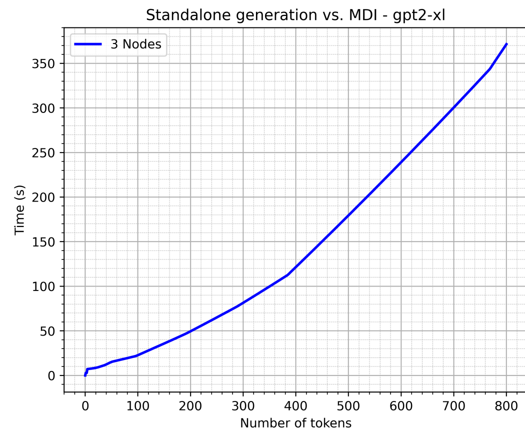

Figure 17: Time vs. N. of tokens – GPT-2 XL

We chose not to reduce the model precision to provide a valid comparison with the other models, which have all been deployed using 32-bit floating point representation.

In this case, the generation rates, evaluated on the full run (start to finish), are the following:

- **GPT-2**:
  - 1 node: 7.91 tokens/s
  - 2 nodes: 8.43 tokens/s
  - 3 nodes: 13.3 tokens/s
- **GPT-2 Medium**:
  - 1 node: 2.63 tokens/s
  - 2 nodes: 4.09 tokens/s

  - 3 nodes: 7.67 tokens/s

- **GPT-2 Large**:
  - 1 node: 1.12 tokens/s
  - 2 nodes: 2.07 tokens/s
  - 3 nodes: 4.03 tokens/s

- **GPT-2 XL**:
  - 3 nodes: 2.15 tokens/s

One of the most notable aspects of all these performance plots is the curvature of the lines. This behavior can be attributed to the larger context length (1024 tokens), which requires processing more data at each forward pass and transmitting bigger messages.

In Figure 14, in particular, it is possible to observe how the effect of message transmission, which is absent in the single-node case, results in a reduced generation rate improvement when increasing the number of nodes. Repeating the same calculations as in the previous section and referring to Table I, it can be seen how the maximum message size (when the context window is full) is huge in this scenario:

$$\text{GPT-2: (ctx. length)} \cdot \text{(emb. dim.)} \cdot 4 \text{ B} = 1024 \cdot 768 \cdot 4 \text{ B} = 3,145,728 \text{ Bytes} \quad (4.2)$$

$$\text{GPT-2 Medium: } 1024 \cdot 1024 \cdot 4 \text{ B} = 4,194,304 \text{ Bytes} \quad (4.3)$$

$$\text{GPT-2 Large: } 1024 \cdot 1280 \cdot 4 \text{ B} = 5,242,880 \text{ Bytes} \quad (4.4)$$

$$\text{GPT-2 XL: } 1024 \cdot 1600 \cdot 4 \text{ B} = 6,553,600 \text{ Bytes} \quad (4.5)$$

In our specific tests, the maximum size has not been reached, as we only generated 400 tokens per sample, which is insufficient to fill the context. However, the effect of larger embedding dimensions and context size causes a relevant slowdown as more tokens are generated, resulting in the curvature of the plots; in other words, the matrices that need to be processed by the model grow larger as more tokens are produced. This effect is undesirable in LLMs, as it translates into a generation slowdown when dealing with longer sentences. This is the reason why KV caching has been introduced in the literature and why we chose to include it in the final MDI-LLM iteration.

Still, despite this issue, distributing the large language models results in a faster generation compared to the standalone case.

It is also possible to observe the same behavior at the origin reported in Figure 13, as the initialization of the models still happens in the same way as before.

Table VII contains the memory usage measurements for the models. Again, it is possible to see how each device's memory usage reduces when adding nodes, even though the total

amount of memory increases due to the need to accommodate the libraries and handle the communication channels. Still, the memory savings on the individual nodes allow GPT-2 XL to be run by splitting it among three nodes.

	1 Node	**2 Nodes**		**3 Nodes**		
Node	1	1	2	1	2	3
GPT-2	1.65 GB	1.43 GB	1.14 GB	1.25 GB	1.04 GB	1.04 GB
GPT-2 Medium	2.9 GB	2.26 GB	1.87 GB	1.65 GB	1.57 GB	1.57 GB
GPT2 Large	5.6 GB	3.87 GB	3.17 GB	3.06 GB	2.37 GB	2.37 GB
GPT-2 XL	N/A	N/A	N/A	4.85 GB	3.87 GB	3.87 GB

TABLE VII: TOTAL (RAM + VRAM) MEMORY USAGE – GPT-2

4.3 Results – Llama

This section contains the performance analysis for the final MDI-LLM version, which differs from the previous ones in that it uses KV caching (with rotating caches), RoPE, and allows the generation of a number of samples greater than the number of nodes.

As anticipated in previous chapters, we deployed two Llama models on our testbed, NanoLlama 304M and TinyLlama 1.1B, based on the Llama 2 architecture. Refer to section 3.5 for the exact specifications.

While NanoLlama's parameters were loaded using 32-bit floating point representation (as all previously analyzed models), we decided to use 16-bit representation for TinyLlama, as the original model was trained using this specific representation, meaning we did not have to quantize

the model and potentially lose performance. As remarked previously, using this floating point representation on the testbed devices (Nvidia Jetson TX2) requires converting 32-bit values when performing GPU operations, which could cause a slight operation overhead. However, we believe this situation could be representative of common edge devices. Furthermore, we did not see a relevant performance loss in our tests due to using a low-precision representation.

Tables Table VIII and Table IX report the model partition for the two considered Llama models. As discussed in previous sections, these models have a slightly different structure compared to GPT-2, as the positional encoding is not performed at the input (adding encoding vectors) but rather at each multi-head attention layer (RoPE). In our distributed system, this means that the starter node only has to perform token embedding before passing the sample through its local transformer blocks.

Model	**N. blocks, node 1**	**N. blocks, node 2**
NanoLlama	5	7
TinyLlama	10	12

TABLE VIII: LAYERS ASSIGNMENT – LLAMA MODELS OVER 2 NODES

We performed our tests generating three samples of 800 tokens in all three settings, thanks to the possibility, introduced in this last MDI-LLM version, of handling as many samples as desired, to allow for a fair comparison.

Model	N. blocks, node 1	N. blocks, node 2	N. blocks, node 3
NanoLlama	2	5	5
TinyLlama	6	8	8

TABLE IX: LAYERS ASSIGNMENT – LLAMA MODELS OVER 3 NODES

It should be noted that, for the two-node case, this results in one sample waiting in a message queue at all times since we have more samples than nodes. This slightly impacts the overall generation speed of each sample but achieves the same token generation rate overall. In other words, the wait time (between sending the request and receiving the result) for three users requesting one sample each will be slightly higher compared to generating two samples, but the system will generate tokens at the same rate.

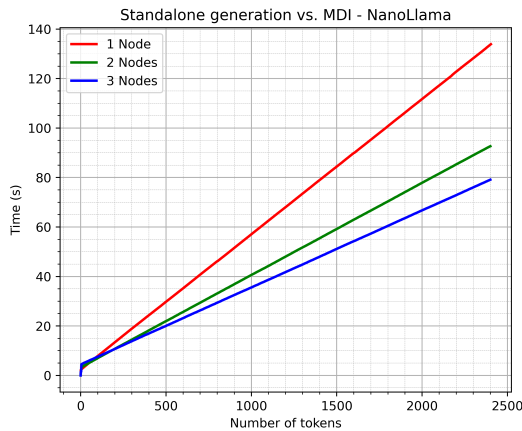

Figure 18: Time vs. N. of tokens – NanoLlama 304M

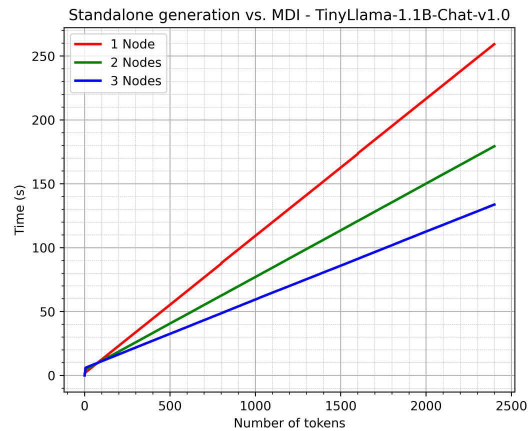

Figure 19: Time vs. N. of tokens – TinyLlama 1.1B

Figure 18 and Figure 19 contain the time vs. number of generated tokens for NanoLlama and TinyLlama, respectively. Once again, the trend is that of a token generation rate that increases with the number of nodes.

The specific token generation rates are:

- **NanoLlama 304M**:
  - 1 node: 17.87 tokens/s
  - 2 nodes: 25.97 tokens/s
  - 3 nodes: 30.46 tokens/s
- **TinyLlama 1.1B**:
  - 1 node: 9.23 tokens/s
  - 2 nodes: 13.33 tokens/s
  - 3 nodes: 18.0 tokens/s

For Figure 18, the same observations that have been made for NanoGPT in 4.1 are valid: due to the small size of the model, the improvement in generation rate going from two to three nodes (16.3 %) is lower than the one going from one to two nodes (45.3 %), as the transmission time becomes a bottleneck.

Compared to section 4.2, the figures also show the benefit of including KV caching, as both models have a maximum context length of 2048 tokens, which is higher than the one of all GPT-2 models (1024), but do not incur in the generation slowdown observed for the previous models.

Indeed, the messages exchanged during generation will only include the single embedding vector associated with the new token, with the exception of the first forward pass for each sample, which will contain the embeddings of all tokens in the prompt. We can evaluate the message size (after the first forward pass) as follows:

$$\text{NanoLlama: (embedding dimension)} \cdot 4\text{ B} = 1024 \cdot 4\text{ B} = 4096\text{ B} \tag{4.6}$$

$$\text{TinyLlama: (embedding dimension)} \cdot 4\text{ B} = 2048 \cdot 4\text{ B} = 8192\text{ B} \tag{4.7}$$

when using 32-bit floating point values. Since in our tests we deployed TinyLlama using half-precision (16-bit) floating point numbers, the actual message size in our scenario was 4096 B.

KV caching introduces a dramatic improvement, allowing bigger samples to be generated in less time than with GPT-2.

The memory usage measurements are found in Table X. Notice how deploying TinyLlama using 16-bit floating point representation results in a much lower memory consumption with respect to GPT-2 XL (Table VII) despite the comparable model size.

It should be noted that KV caching speeds up computation at the cost of higher memory usage. Indeed, the cached matrices will be stored in the device memory, and the amount of stored data will increase with the number of processed samples.

RoPE embedding matrices are also cached in our implementation to improve the throughput (as done in state-of-the-art production LLMs), as they are not trainable parameters. In

particular, the RoPE sine and cosine values are initialized at the beginning of each run for all the possible positions within the context. It should be noted that the memory used for the RoPE cache adds to the boilerplate memory used by each node (libraries, web server, and connection state information) since they are used in every multi-head attention layer; hence, each node will have to initialize its copy.

	1 Node	**2 Nodes**		**3 Nodes**		
Node	1	1	2	1	2	3
NanoLlama	2.6 GB	1.76 GB	1.76 GB	1.34 GB	1.46 GB	1.46 GB
TinyLlama	3.41 GB	2.6 GB	2.6 GB	1.98 GB	1.99 GB	1.99 GB

TABLE X: TOTAL (RAM + VRAM) MEMORY USAGE – LLAMA

CHAPTER 5

CONCLUSIONS

This work provides the implementation details and performance analysis of the application of pipeline parallelism to large language model inference. We also show how properly implementing recurrent pipeline parallelism allows a network of nodes to achieve better performance in terms of throughput when compared to running the model on a single device, as it forces each node to be working at all times, preventing idle waiting times.

We started our development with small models based on one of the first LLM architectures to prove the validity of our approach in achieving efficient parallelism. We then proceeded to scale up the system, supporting bigger models with billions of parameters and including some architectural improvements introduced in more recent LLMs. By devising an approach that allows nodes to exchange messages as fast as possible and reducing the message size using KV caching, we also succeeded in minimizing the network's effects on overall system performance. These benefits add to MDI-LLM's main objective: to allow the deployment of massive models that would not fit on a single device using a network of interconnected devices.

We believe this research is an important step towards deploying LLMs and transformer-based models at the Internet edge. This would significantly increase AI accessibility, allowing the design of innovative applications that would benefit from processing happening closer to the user.

Most of the skepticism concerning AI applications today is related to the end users' lack of control over their own data. By allowing models to run on low-capability hardware and on-premises, we can avoid having data sent to third-party computers, as all the processing is done locally and does not require expensive hardware.

The developed framework could also find its way into more peculiar edge scenarios like drone fleets or smart mobility. Allowing different hosts to process data concurrently using bigger models that cannot be fully deployed on a single node would make performing more demanding operations and improving the overall system capabilities possible.

Scalability is another very relevant benefit that should not be overlooked. Indeed, adding nodes to the network is enough to support bigger, more complex models.

Thanks to these characteristics, model-distributed inference is a framework that can potentially adapt to support future AI models, especially as we increase their computational requirements. Increasing the network size makes it possible to process more samples efficiently too. This characteristic makes MDI-LLM a valid framework for making the deployed model highly available, as effective generation is achieved by processing many samples concurrently.

Still, our implementation is a proof of concept that showcases the best theoretical scenario for deploying distributed large language models, as it does not have to deal with practical limitations related to the deployment in actual edge scenarios. A crucial aspect is the device interconnection, which, in our testbed, was ideal (gigabit ethernet) compared to actual edge scenarios where communication can be unreliable or slow. Thanks to the introduction of KV caching, we achieved the lowest message size possible in the final version of MDI-LLM. Still,

unreliable connections could represent a limitation, given the number of message exchanges that need to be performed at each run. This issue requires devising strategies to introduce resilience to node failure and recovery, of which some examples have already been analyzed for different applications of MDI [6]. Additionally, we have observed how the overall effect of message exchange increases with the number of nodes. This factor should be considered when discussing larger networks.

All these points should be properly taken into account to be able to deploy the system in practical scenarios.

5.1 Future developments

We propose here some ideas for future development of this line of work, especially to try tackling the aspects we highlighted previously and that, as of now, represent obstacles preventing the actual deployment of the MDI-LLM framework in "production" environments.

An aspect we neglected in our analysis is the introduction of strategies to increase system reliability. An example of such methods is introducing redundancy in the processing to ensure the system can produce the responses even if nodes fail. Another idea is to devise recovery methods to tackle connection issues in remote scenarios, where the quality of the communication channels may degrade due to environmental changes. Some more complex approaches could be to analyze systems in which the physical network topology changes over time, forcing the overlay topology, i.e., the specific node-to-node connections used to transmit the model activations, to change as well.

Other interesting studies can be carried out to minimize message transmission's effect on the overall system performance and stability. As observed, the large number of messages that need to be exchanged can greatly impact system performance, especially when the number of nodes increases, affecting the generation rate stability in time. A straightforward approach that could be followed is that of trying to introduce compression, as reducing the message size makes transmission faster and reduces the risk of network congestion bottlenecks. Still, compression would introduce processing overhead, depending on the complexity of the specific technique used.

A more sophisticated work could pair compression with batching, i.e., processing samples "in parallel" at the same node by increasing the "batch size" – in our implementation, the batch size was set to 1. This approach would significantly increase the number of processed samples and, compared to the scenario we described, reduce the number of messages exchanged when working with the same number of samples, as now, each message will contain the activations of more than a single sample. In other words, the amount of transmitted data would increase, but the number of messages would decrease significantly.

Lastly, in recent years, new architectures have been proposed as alternatives to transformers. Most notably, the "Mixture of Experts" [35] proposes itself as an interesting structure to be deployed in a distributed fashion. This new model type could be brought to the internet edge by assigning the "experts" to different subsets of network nodes. Concerning new generative AI types, instead, it would be relevant to attempt the deployment of multi-modal models using

model-distributed inference, e.g., by delegating different tasks to different groups of nodes that process different foundational models.

With so many advancements in AI, it is more than ever necessary to devise ways to make these new innovations more accessible to researchers and general users. With this work, we investigate one possible approach to moving away from a private, geographically distant cloud. This approach not only avoids limitations related to privacy and slow response times, but it also allows the deployment of large language models in a scalable way at the edge, resulting in the ability to run state-of-the-art models on low-power hardware. The MDI-LLM framework has potentially many applications, from just enabling the deployment of models that are too large to fit in the memory of a single computer to more innovative uses of generative AI and transformer architecture, such as distributed sensing, UAV fleets, and self-driving vehicles.

CITED LITERATURE

1. Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S.: Pytorch distributed: Experiences on accelerating data parallel training, 2020.

2. Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E.: Measuring the effects of data parallelism on neural network training, 2019.

3. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B.: Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

4. Wang, B., Xu, Q., Bian, Z., and You, Y.: Tesseract: Parallelize the tensor parallelism efficiently. In Proceedings of the 51st International Conference on Parallel Processing, ICPP ,Äô22. ACM, August 2022.

5. Borzunov, A., Ryabinin, M., Chumachenko, A., Baranchuk, D., Dettmers, T., Belkada, Y., Samygin, P., and Raffel, C.: Distributed inference and fine-tuning of large language models over the internet, 2023.

6. Li, P., Koyuncu, E., and Seferoglu, H.: Adaptive and resilient model-distributed inference in edge computing systems. IEEE Open Journal of the Communications Society, 4:1263–1273, 2023.

7. Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z.: Gpipe: Efficient training of giant neural networks using pipeline parallelism. CoRR, abs/1811.06965, 2018.

8. Hu, Y., Imes, C., Zhao, X., Kundu, S., Beerel, P. A., Crago, S. P., and Walters, J. P. N.: Pipeline parallelism for inference on heterogeneous edge computing, 2021.

9. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I.: Attention is all you need, 2017.

10. Gage, P.: A new algorithm for data compression. http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM, 1994. [Online; accessed: 2024-07-15].

CITED LITERATURE (continued)

11. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I.: Language models are unsupervised multitask learners. 2019.

12. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T.: Llama 2: Open foundation and fine-tuned chat models, 2023.

13. Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E.: Mistral 7b, 2023.

14. Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I.: Improving language understanding by generative pre-training. 2018.

15. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Roziére, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G.: Llama: Open and efficient foundation language models, 2023.

16. Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y.: Roformer: Enhanced transformer with rotary position embedding. arXiv preprint arXiv:2104.09864, 2021.

17. Ba, J. L., Kiros, J. R., and Hinton, G. E.: Layer normalization, 2016.

18. Hendrycks, D. and Gimpel, K.: Bridging nonlinearities and stochastic regularizers with gaussian error linear units. CoRR, abs/1606.08415, 2016.

19. AI, M.: Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/, 2024. [Online; accessed].

20. Shazeer, N.: Glu variants improve transformer, 2020.

21. Zhang, B. and Sennrich, R.: Root mean square layer normalization, 2019.

22. Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebròn, F., and Sanghai, S.: Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

23. Chen, J. and Ran, X.: Deep learning with edge computing: A review. Proceedings of the IEEE, 107(8):1655–1674, 2019.

24. Zhao, Y., Lin, C.-Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., and Kasikci, B.: Atom: Low-bit quantization for efficient and accurate llm serving, 2024.

25. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K.: A survey of quantization methods for efficient neural network inference, 2021.

26. Jaiswal, A., Gan, Z., Du, X., Zhang, B., Wang, Z., and Yang, Y.: Compressing llms: The truth is rarely pure and never simple, 2024.

27. Xu, X., Li, M., Tao, C., Shen, T., Cheng, R., Li, J., Xu, C., Tao, D., and Zhou, T.: A survey on knowledge distillation of large language models, 2024.

28. Sun, M., Liu, Z., Bair, A., and Kolter, J. Z.: A simple and effective pruning approach for large language models, 2024.

29. Karpathy, A.: NanoGPT. https://github.com/karpathy/nanoGPT, 2022.

30. AI, L.: Litgpt. https://github.com/Lightning-AI/litgpt, 2023.

31. Nvidia: Jetson tx2 technical specifications. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.

32. Chen, Y., Luo, T., Fang, W., and Xiong, N. N.: Edgeci: Distributed workload assignment and model partitioning for cnn inference on edge clusters. ACM Trans. Internet Technol., 24(2), may 2024.

33. Karpathy, A.: Tiny shakespeare data set. https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt, 2015.

CITED LITERATURE (continued)

34. Zhang, P., Zeng, G., Wang, T., and Lu, W.: Tinyllama: An open-source small language model, 2024.

35. Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E.: Mixtral of experts, 2024.

36. Macario, D.: A Model-Distributed Inference Approach for Large Language Models at the Edge, April 2024.

37. Habibi, P., Farhoudi, M., Kazemian, S., Khorsandi, S., and Leon-Garcia, A.: Fog computing: A comprehensive architectural survey. IEEE Access, 8:69105–69133, 2020.

VITA

NAME	Davide Macario
EDUCATION	
	Master of Science in "Electrical and Computer Engineering", University of Illinois at Chicago, *Month* 2024, USA
	Master of Science in "ICT for Smart Societies", *Month* 2024, Polytechnic of Turin, Italy
	Bachelor's Degree in "Electronics and Communications Engineering (ECE)" - September 2022, Polytechnic of Turin, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2022 - IELTS examination (8.0/9 – C1)
	A.Y. 2023/24 One Year of study abroad in Chicago, Illinois
	2019–2023 Lessons and exams attended exclusively in English
SCHOLARSHIPS	
Spring 2024	Graduate Hourly (GH) position (10 hours/week) with stipend
Fall 2023	Italian scholarship for TOP-UIC students
2022–2024	"Alta Scuola Politecnica (ASP)" program, with full tuition waiver, Polytechnic of Turin and Polytechnic of Milan
2019–2022	"Giovani Talenti" program, with full tuition waiver, Polytechnic of Turin, Italy
TECHNICAL SKILLS	
Basic level	Kubernetes, C++
Average level	Docker, Linux, C, MATLAB, Data Science
Advanced level	Python, PyTorch, Computer Networking, Neural Networks, Machine Learning

VITA (continued)

WORK EXPERIENCE AND PROJECTS

2023	"FREISA - Four-legged Robot Ensuring Intelligent Sprinkler Automation"
	Winner of the OpenCV AI Competition (2023)
2023	Creation and implementation of a heuristic solution for the "Container Loading" problem, Operations Research
2023	Development of a microservices application for a smart greenhouse in Python
2022	Design and implementation of a digital synthesizer in MATLAB