Towards Parallel Transformer-Based Large Language Models for Fast Inference

Félix Wirth, Gael Delalleau, Loris Marchal

▶ **To cite this version:**

Félix Wirth, Gael Delalleau, Loris Marchal. Towards Parallel Transformer-Based Large Language Models for Fast Inference. RR-9573, Inria. 2025, pp.42. hal-04920049

HAL Id: hal-04920049

https://inria.hal.science/hal-04920049v1

Submitted on 31 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Towards Parallel Transformer-Based Large Language Models for Fast Inference

Félix Wirth, Gael Delalleau, Loris Marchal

RESEARCH REPORT
N° 9573
January 2025
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9573--FR+ENG

Towards Parallel Transformer-Based Large Language Models for Fast Inference

Félix Wirth*, Gael Delalleau†, Loris Marchal‡

Project-Team ROMA

Research Report n° 9573 — January 2025 — 43 pages

Abstract: Transformer-based models have reached quality levels that make them attractive for a wide range of applications, including those demanding faster generation speeds such as multimedia artifact generation. However, traditional parallelism techniques have struggled to meet these speed requirements due to the inherently sequential structure of Transformers, which limits parallel execution.

We propose a simple modeling to the parallel execution of Transformer, which identifies synchronization times across inference devices as the culprit. Leveraging real world figures from GPU manufacturers, we highlight the increasing cost of this synchronization time with ever more powerful hardware. In response, we introduce Tensor Parallel Blocks, a naturally parallelized architecture. To maintain quality, we incorporate multiple enhancements, such as delayed inter-block communication, hybrid architecture, and different parallelism gateways, which mitigate potential degradation in model output.

We design and validate a dedicated training protocol tailored to this architecture and conduct a preliminary hyperparameter study on 150M models, proving hardware-tied architecture can achieve relevant results. Using insights from these trials, we establish a set of rules for scaling the model to larger size, theorizing a 4.2x speedup in decoding time compared to an equivalently sized LLaMA based model on the target inference platform.

This document describes the work done during the 6 month internship of Félix Wirth in the ROMA team, advised by Loris Marchal and in collaboration with Gael Delalleau from the Kog company.

Key-words: LLMs, fast inference, parallelism

* CentraleSupélec & Inria
† Kog
‡ CNRS & Inria

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Une architecture de grands modèles de texte parallèle basée sur les Transformers pour l'inférence parallèle rapide

Résumé : Les grand modèles de textes utilisant des blocs de Transformers ont atteint des niveaux de qualité qui les rendent attractifs pour une large gamme d'applications, y compris celles exigeant des vitesses de génération plus rapides telles que la génération d'artefacts multimédias. Cependant, les techniques de parallélisme traditionnelles ont du mal à répondre à ces exigences de vitesse en raison de la structure intrinsèquement séquentielle des Transformers, qui limite l'exécution parallèle.

Nous proposons une modélisation simple de l'exécution parallèle de Transformer, qui identifie les temps de synchronisation entre unités de calcul comme critiques. En nous appuyant sur des chiffres réels des fabricants de GPU, nous soulignons le coût croissant de ce temps de synchronisation avec un matériel toujours plus puissant. En réponse, nous introduisons Tensor Parallel Blocks, une architecture naturellement parallélisée. Pour maintenir la qualité, nous incorporons plusieurs améliorations, telles que la communication inter-blocs retardée, l'architecture hybride et différentes passerelles de parallélisme, qui atténuent la dégradation potentielle de la sortie du modèle.

Nous concevons et validons un protocole de formation dédié adapté à cette architecture et menons une étude préliminaire des hyper-paramètres sur 150M modèles, prouvant que l'architecture liée au matériel peut obtenir des résultats pertinents. En utilisant les informations issues de ces essais, nous établissons un ensemble de règles pour passer ce modèle à plus grande échelle, en prédisant accélération de 4,2 sur le temps de décodage par rapport à un modèle basé sur LLaMA de taille équivalente sur la plateforme d'inférence cible.

Ce document décrit le travail effectué pendant le stage de 6 mois de Félix Wirth dans l'équipe ROMA, encadré par Loris Marchal et en collaboration avec Gael Delalleau de la société Kog.

Mots-clés : parallélisme, grands modèles de langage, inférence optimisée

Contents

1 Introduction **5**

2 Related Works **6**
2.1 Transformer Architecture 6
2.2 Pre-training Task 8
2.3 Dataset 8
2.4 Tokenization 9
2.5 Scaling Laws 9
2.6 Fine Tuning 9
2.7 Auto-Regressive Inference 9
2.8 Speculative Decoding 10
2.9 KV-Cache 10
2.10 Attention variants 10
2.11 Model Compression 11
2.12 Computing Device 12
2.13 Kernel Optimization 12
2.14 Parallelism 12
2.15 Parallelism-aware architectures 13

3 Problem Statement **13**
3.1 GPU modeling 13
3.1.1 Single Device 13
3.1.2 Multiple Devices 15
3.2 Transformer Inference Modeling 15
3.3 Communication bottleneck 17
3.4 Modeling Validation 20

4 Solution **20**
4.1 Tensor Parallel Blocks, Architecture 20
4.2 Delayed Communication 22
4.3 Block Mixing 23
4.4 Recipes 24
4.5 Speed-up Gains 24

5 Setting up Training Protocol **25**
5.1 Initial Training Protocol 25
5.2 First experiment wave 25
5.3 Strengthening Training Protocol 26
5.3.1 Comparison fairness 26
5.3.2 Compute Policy 27
5.3.3 Dataset Size 28
5.3.4 Learning Rate, Scheduler 29

6 Preliminary Architectural Study **31**
6.1 Methodology 31
6.2 Lane Count 32
6.3 Lane Aspect Ratio 32
6.4 Number of Tensor Parallel Blocks 34

6.5 Number of sequential classic Decoder Layers 34
6.6 Delayed Information 35
6.7 Larger Scale, Limitations 35

7 Conclusion **36**

1 Introduction

This internship was conducted under a collaboration between the Resource Optimization: Models, Algorithms, and Scheduling (ROMA) team from National Institute for Research in Digital Science and Technology (INRIA) and Kog.ai, a startup focused in providing a service creating multimedia artifacts : document, graphic, audio file, video file, presentation, or other form of digital media. Creating them demands powerful, complex modeling techniques capable of handling diverse data formats, from audio files and images to text documents and video. Traditional methods fell short in this area, lacking the flexibility and robustness needed to manage multiple types of data. Transformers emerged from this landscape as an adaptive and powerful solution, proving uniquely effective due to their architecture's inherent ability to handle various modalities simultaneously. Initially developed for natural language processing (NLP), Transformers have since extended across domains, now setting the standard for generating rich multimedia content through their versatile, multi-modal capabilities at a reasonably fast speed.

AI models such as Transformers typically operate in two main phases: training and inference. During training, the model learns to perform specific tasks by processing large datasets and adjusting its parameters accordingly—a phase that can be highly resource-intensive and time-consuming. Once trained, the model enters the inference phase, where it applies this learned knowledge to analyze and generate new content in real time. Optimizing this inference phase is crucial for meeting the demand for rapid, high-quality multimedia generation, especially as models grow larger and more complex to handle multi-modal inputs. Kog plans on delivering much higher generation speed at inference than concurrent products.

Several optimization techniques have been developed to achieve that goal. A primary approach involves increasing the computational power by running models on GPUs instead of CPUs, as GPUs offer massive computational power well-suited to the size and type of computations in Transformers. Building on this, multiple GPUs can be used simultaneously, applying mathematical techniques known as parallelism to split the workload across processors. These approaches achieve the same end result but through computationally distributed algorithms that speed up processing. In addition, kernel optimizations, which improve the underlying computation processes within GPUs, further enhance efficiency by reducing runtime for specific operations. Architectural variants of the Transformer model also exist to improve inference speed, though these usually require retraining the model, which usually is a time and resource intensive process. Kog leverages GPU parallelism and develops its own computation kernels to increase performance on their hardware and would like to develop their own optimized model architecture.

Architectural research on Transformers for inference has traditionally prioritized two main goals: enabling large-scale model deployment for high-volume inference demands and adapting models to run efficiently on limited hardware, such as mobile devices. However, the target use case for Kog requires a different approach: achieving high generation speed for a small number of simultaneous clients on one multi-GPU machine. This specific scenario presents unique challenges, as the traditional Transformer architecture was not designed with this combination of high-speed, low-latency requirements and constrained volume as target use case. Consequently, limitations in existing Transformer architectures appear in this setting, and no current solutions have fully addressed these inference speed bottlenecks.

Addressing this bottleneck requires modifying the architecture itself, leading to a costly retraining process. However, recent advances have helped lower this cost: hardware has become more affordable and efficient, and new training software has greatly streamlined model development. These improvements make it feasible to explore architectural changes optimized for multi-GPU, low-latency environments. This shift opens new opportunities for research into Transformer variants tailored for fast, high-quality multimedia artifact generation. While these

variants increase inference speed, they often come at the cost of predictive quality. Bridging this quality gap will be essential to ensure that the optimized variants continue to meet Kog's standards for multimedia artifact creation.

The INRIA/ROMA - Kog collaboration targets theorizing the distributed inference speed bottleneck and developing solutions to overcome and/or eliminate it. To keep the scope feasible and reduce the factors involved, this internship report will focus on natural language modeling only and tackle the following question : How to modify the Transformer architecture to achieve maximal decode inference speed at low volume on a distributed system without losing quality with a small LLM ?

The first half of this internship took place in ROMA's office, located in ENS Lyon. To promote AI exchanges with other researchers and students, the second half of the internship was conducted as a visiting researcher at International Laboratory on Learning Systems (ILLS) in Montréal, Canada. All along the internship, biweekly exchanges took place with Kog. This organization made it possible to bring together the different areas of expertise needed to best solve the problem. Continuous face-to-face supervision with the main supervisor throughout the internship was observed.

This document will first provide (Section 2) a broad exploration of Transformers and existing literature on training and inference optimization. This review will set the stage for our problem statement described (Section 3), which focuses specifically on the unique challenges of achieving low-volume, high-speed distributed inference. We will then introduce our proposed solution (Section 4) before designing (Section 5) and conducting experimental studies (Section 6). Finally, the document will conclude (Section 7) with a brief summary and open up potential future directions to build upon this initial work for Kog in a discussion part.

2 Related Works

Answering Kog's issue requires studying and theorizing LLM distributed inference, before suggesting an architectural variant, testing its hyper parameters, training it and comparing it to a baseline.

To this end, the following literature review will provide broad survey covering large language model (LLM) architectures (primarily Transformer-based) and the various optimizations applied to architecture, training and inference, as progress have been made on each of the 3 aspects, alone and combined, and should be considered for the conducted work : While the solution will focus on the model's architecture, it will later be integrated to other existing techniques, further increasing the speed gains.

2.1 Transformer Architecture

The foundational architecture of Transformers in natural language processing (NLP) was introduced in [1], which proposed a self-attention mechanism that allowed models to capture long-range dependencies more effectively than previous recurrent architectures, initially targeting tasks like text translation. This architecture soon evolved beyond translation tasks, leading to the development of models such as [2], which adopted a 'decoder-only' structure specifically optimized for text generation. This architecture contains an initial embedding layer, followed by a series of n_{layers} stacked decoder layers and concluded by a projection layer. Each decoder layer transforms the input representations through a Multi Head Attention and Feed Forward Network.

The embedding layer maps each input token from a vocabulary with size *vocab_size* to an embedding vector of size h, encoding its semantic features for use in subsequent processing:

$$\text{Embedding}(x) = W_E[x]$$

Multi Head Attention allows the model to simultaneously attend to different parts of the input sequence via multiple attention heads, each computing scaled dot-product attention. The outputs from these heads are combined to capture diverse relationships and dependencies within the sequence:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

$$\text{and Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Where X are the hidden states, W_i^Q, W_i^K, W_i^V are the weights matrices associated to the computation of Q, K, V vectors for each attention head i of size d.

In the task of text generation, masking is used to prevent the model from attending to future tokens in the sequence. It ensures that, during training, each token can only access information from itself and previous tokens, excluding future ones. This is achieved by setting the attention scores for future positions to a very large negative value, blocking their influence in the softmax calculation, to best simulate the future inference case.

The FFN, typically Gated Linear Unit (GLU) variants such as SwiGLU introduced in [3], creates non-linearity to learn complex patterns. It operates using 3 weights matrices W_1, W_2, W_3 and an activation function σ:

$$\text{FFN}(x) = \text{GLU}(xW_1 + b_1)W_2 + b_2$$

$$\text{where GLU}(X) = \sigma(XW_1) \odot (XW_2)$$

The final projection layer converts the output of the last decoder into a probability distribution over a vocabulary.

$$\text{Projection}(h) = h \times W_P$$

The next token is finally predicted by sampling the probability distribution. Multiple sample algorithm exist, such as argmax:

$$\hat{y} = \arg\max(\text{softmax}(\text{Projection}(h)))$$

Layer normalization (RMS Normalization [4])is applied before each of the model's components for stable training, while positional encodings (RoPe [5]) are added to the embeddings once at the beginning, to provide sequential order information. This architecture can be found in the LLaMA-(1),2,3 (Large Language Model Meta AI, [6, 7, 8]) which is widely accepted as state of the art (SOTA) open model architecture. Figure 1 summarizes the Llama-2 and Llama-3 architecture, which will be used as the baseline in this document.

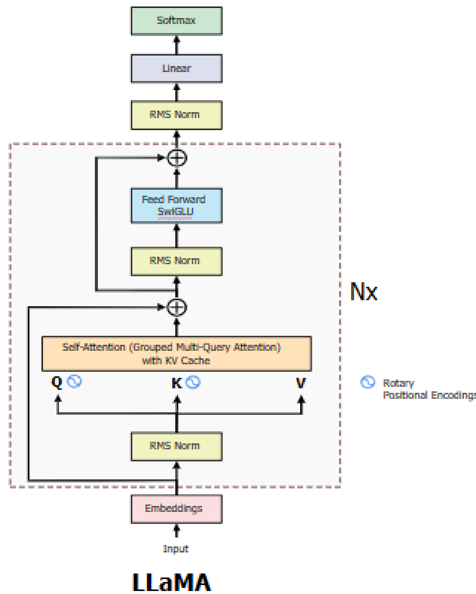


Figure 1: Llama 2 Decoder-Only architecture (Source : [9])

2.2 Pre-training Task

Current large language LLMs are pre-trained using unsupervised learning techniques [2], where the model learns patterns, structures, and semantic relationships from vast amounts of unlabeled text data. This unsupervised phase is critical as it allows the model to build a foundational understanding of language without requiring extensive labeled datasets. During pre-training, the model learns to generate text, which enables it to develop robust language representations that can later be fine-tuned for specific tasks. This initial phase is also essential for hyper-parameter selection, allowing iterative adjustments to the model's parameters (such as learning rate, batch size, or model depth) based on the model's ability to generalize from the pre-training data. By comparing performance metrics (such as the loss) with other models during this stage, we can compare our architecture to the baseline and identify the most promising one.

2.3 Dataset

In this unsupervised pre-training, the choice of dataset is fundamental to shaping the model's performance and versatility. Datasets have evolved significantly over time, both in quality and quantity, providing LLMs with richer and more diverse language patterns, as shown in [10]. Modern datasets may be general, drawing from a wide range of sources and topics to build models that understand broad linguistic contexts, or specialized, targeting specific domains such as legal, scientific, or medical language to develop models with deeper expertise in those areas. Datasets may also be multilingual, allowing the model to learn cross-lingual patterns and improving its ability to perform in multiple languages. Creating a pre-training dataset represents a massive engineering and mathematical task, requiring the collection, cleaning, and structuring of vast amounts of text data from diverse sources. We will reuse existing datasets to avoid such task and reduce external factors.

2.4 Tokenization

Once a suitable dataset is selected, it must be processed through tokenization, which breaks the text into smaller units known as tokens. This step involves defining tokenization rules that may include words, subwords, or characters. Tokenization rules are learned and refined on the dataset, usually using the BPE algorithm ([11]. This preparation through dataset selection and tokenization ensures the model can efficiently process and understand its training data. This process leads to the tokenizer being coupled with the model, which we should be careful with when comparing models between each other, to avoid unfair comparison.

2.5 Scaling Laws

The impact of each of the available hyper parameters on the model's quality of generation was studied in [12, 13], and suggests that increasing the number of parameter yields the most performance increase as long as parameters are within an 'acceptable' range. The performance of the trained model could be approximated by a power law of the total parameter count. Following that result, models are usually referred to as a combination of their architecture and total number of parameters (ranging from a few millions to trillions parameters), such as : Llama3-403B. A study targeted at smaller models (hundred of millions of parameters), [14], shows that the usual assumptions on performance do not hold exactly true at that scale and specific parameters such as number of layer have a strong impact on the performance. Other SOTA decoder-only trained with fewer than 10 billions parameters include [15, 16, 17], we chose to use 1.5B parameters modified TinyLlama [18] model as a reference due to its wide use across studies and tasks and relatively accessible training budget.

2.6 Fine Tuning

Once the architecture is defined and trained, the model can be specialized for a given task, for which labeled data is available. The naive approach is to continue training the model with that specific data and possibly adapting loss function, learning rate... depending on the task. This approach can be slow and expensive ; A cheaper alternative is to freeze almost all of the model and only train the last layer, which does not allow great adaptation of the model to the final task. In between both approaches, Parameter Efficient Fine Tuning (PEFT) regroup efficient fine-tuning methods, including training an adapter around selected layers in the LORA [19] fashion. As PEFT allow to specialize the model at a cheap price, many techniques were developed in that direction, as described in [20]. In the scope of this internship, we will only perform pre-training and hyper-parameters selection.

2.7 Auto-Regressive Inference

With the model fine-tuned and optimized for its specific tasks, it's now ready to move into the inference phase. Inference for a decoder-only generative model involves repeatedly sampling the next token from the model's predictions based on all previous tokens. Once a token is sampled, it is added to the sequence. The model is then called again, using the updated sequence as input, until the initial requirement is satisfied, usually reaching a target number of sampled tokens, or sampling the 'end of sequence' token, in a process called Auto-Regressive Generation.

2.8 Speculative Decoding

Speculative decoding techniques have emerged as innovative strategies to enhance inference speed without significantly compromising output quality. One such approach , introduced in [21], involves the use of extra decoding heads that enable the model to predict multiple tokens simultaneously, thereby accelerating the generation process, although requiring a model re-training or fine-tuning of the new parameters at least. Alternatively, leveraging a smaller more efficient model to propose several token candidates can streamline computation as proposed in [22, 23]; a larger, more powerful model then verifies these predictions to ensure accuracy and coherence. This two-tiered system balances speed and reliability by offloading preliminary predictions to the lightweight model while maintaining high standards through the verification stage. This approach avoids the cost of retraining the larger model, as only the smaller model is necessary and an off-the-shelf option can be used. In an early-stopping philosophy, [24] allows model to end the decode step before the last layer, reducing the inference time. Furthermore, beam search techniques are often integrated with speculative decoding to manage and refine the multiple potential token sequences generated, effectively exploring a broader solution space and selecting the most promising candidates. Together, these methods represent significant advancements in decoding efficiency, offering scalable solutions for deploying large language models in real-time applications.

2.9 KV-Cache

As the attention mask keeps the computations for previously processed tokens unchanged even as new tokens are added, some computations become redundant during the auto regressive generation process, significantly slowing generation. To address this, key-value caching (KV-Cache) is used to store the key and value vectors for previously processed tokens. Instead of re-computing them, they are stored and reused, leading to faster inference at the cost of additional memory to store the cache. When using the KV-Cache, the first token prediction has to fill the KV-Cache and compute the key and values for all the initial tokens. The distinction between the two steps lead to the first step being referred to as the "Prefill" step, while the following steps are named "Decode" steps (Each step predicting one token). In the case of Kog, optimizations proposed in the document aim at reducing the decode time. During the Decode phase, only the hidden states of the last-added token are processed to generate the next tokens.

2.10 Attention variants

Multi Query Attention (MQA [25]), generalized by Grouped Query Attention (GQA [26]), is an efficient variation of the standard self-attention mechanism that aims to reduce computational complexity and memory usage, which can get very expensive for the bigger models. In GQA, instead of computing keys and values for all attention heads, the queries are grouped into smaller subsets or clusters. Each group of queries attends only to a corresponding subset of keys and values, reducing the amount of K, V vectors to be computed and stored at the cost of limiting the interaction between different parts of the input. The modified attention algorithm, represented in Figure 2, can be described as followed for a n heads split between G query groups :

$$\text{GQA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

$$\text{where head}_i = \text{Attention}(XW_i^Q, XW_{i//G}^K, XW_{i//G}^V)$$

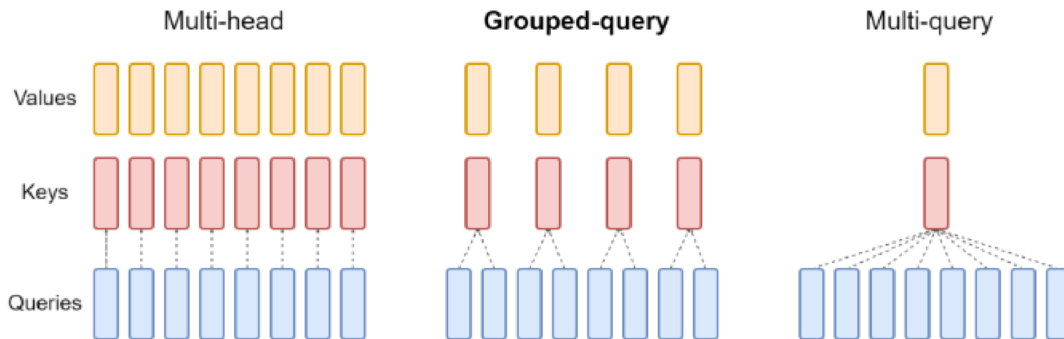


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each group of query heads, interpolating between multi-head and multi-query attention (Source:[26].

$$\text{and Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

In MQA, G=1, while In the traditional attention, G $= n$. Other values are regrouped in GQA. By reducing the number of parameters and different keys, values vectors used, the memory footprint of the KV cache is reduced by a factor G, allowing the hardware to handle bigger sequence of tokens and processing them faster.

Following the Llama architecture, we will only use GQA, but multiple other variants exist to reduce either the computation time or memory cost of attention and are worth mentioning. Multiple works approaches were tested, including applying quantification to KV-Cache [27], reducing attention computation time using linear complexity attention [28], logarithmic complexity attention[29], reducing attention size with sliding window attention in [30], or sparsification [31]. Completely different architectures have also be proposed as a replacement of attention, such as [32], [33] or [34], resulting in impressive inference speed gain although with lower generation quality. Hybrid architectures combining both approaches were tested in [35]. We will stick to the Llama architecture to best understand how our modifications impact the model's performance and minimize external factors.

2.11 Model Compression

These techniques are usually combined with model size reduction techniques. They can be applied before or after training. Usual pre-training technique uses reduced precision for weights matrices from the usual full precision float data type to lower types such as FP16, replaced by BF16 for numerical stability, and even pushed to lower sizes, with one extreme case being a 1 bit compression in BitNet [36], greatly reducing training time. Pruning with knowledge distillation as in [37], and recently applied in [38] proved how the performance of large models can be obtained by smaller replicas trained to imitate the initial model. Post-training techniques usually apply quantization [39], effectively compressing the model at reduced cost of generation quality, while allowing a higher precision training to avoid instabilities and convergence issues. Both approach can be combined together, as in [40]. Post training Quantization (possibly combined with quantization) will be an appealing solution to further increase the inference gains in a future

work.

Another recent paradigm is the use of Mixture of Experts (MoE) models, introduced in [41] and recently applied in [42, 43, 44, 45], which combines a structured sparsification and selection of the weights to allow the model to chose its weights for each token. While this technique shows great results, it is restrained by the large amounts of memory transfers during inference, especially at Decode phase. Workarounds were proposed in [46, 47], and scheduling optimizations in [47], although MoEs remain restraining in our case. Refinements over the proposed workarounds combined with a hybrid Prefill-Decode load balancing scheduler may allow the use of such architecture in future iterations of this work.

2.12 Computing Device

Once the model is fully designed and set, the only remaining task is to perform the inference. In this regard, each model can be broken down into basic mathematical operations (mostly matrix multiplications for Transformers) that have to be performed by the computing device. Modern NLP models primarily rely on GPUs, with NVIDIA [48] leading the field due to its optimized performance for deep learning. However, several alternatives exist, including GPUs from AMD [49] and entirely different devices, such as Amazon's Graviton [50] processors, Google's TPUs (Tensor Processing Units, [51]), and Groq's LPUs (Language Processing Units, [52]). Each offers unique advantages, expanding the hardware options available for NLP and AI tasks. GPUs remain the most accessible options, making them best suited to Kog's needs.

2.13 Kernel Optimization

Executing the model's computations on a device relies on computation kernels, which describe the low-level instructions for the device(s) to execute to make the computation. Due to the complexity of each device and physical differences between each of them, kernels usually need to be specifically tailored to each device and each operation. By reducing execution time and enhancing data handling, these optimizations are crucial for achieving the high performance needed in large language model computations.. In this light, training and/or inference kernels have been developed, such as llama.c ([53]), Flash-Attention [54], vLLM ([55]). Recent initiatives such as Triton [56] even allow relatively strong kernel optimization without tying code to the hardware's programming language, facilitating such optimizations. Kog produces their own computation kernels tailored for Transformer inference on their target hardware, and we will assume "perfect" kernels in the following work. It is worth noting that due to shared standard in data formats, one can use different hardware to perform training and inference.

2.14 Parallelism

Using an appropriate computing device offers impressive speed-up in computations. To further improve the gains or overcome insufficient capabilities of the device, multiple devices can be used together to perform computations in parallel. Devices are physically distributed in multiple nodes, each with multiple devices and heterogeneously connected together (inter and intra-node). Parallel computations algorithms split the workload (training or inference) across multiple devices, abstracting the complex configuration away, ideally reducing the computation time by the number of devices used in parallel. Main algorithms include Model/Data Parallel, Tensor Parallel and Pipeline Parallel and further optimized Tensor Parallel in [57]. Figure 3 provides a representation for each of them. They can be combined also be combined. Choosing the right algorithm relies on multiple factor : The workload (training or inference), its size (which model

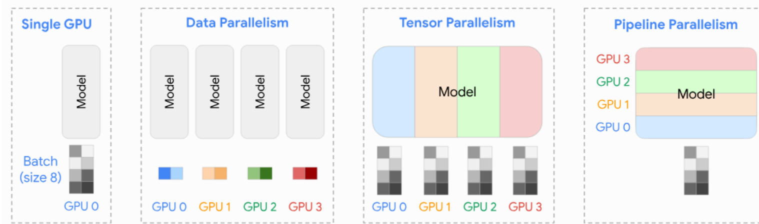


Figure 3: Comparison between Single GPU and Model/Data Parallel, Tensor Parallel and Pipeline Parallel algorithms (Source : University of Virginia [58]).

and inference volume), the computing system (Single or multi node, single or multi GPU). With respect to Kog's decode time with low volume at inference issue, Tensor Parallelism on a multi GPU is best suited and will be presented and discussed in Section 3. It is worth noting that computation kernels also need to be specialized for the parallel algorithm(s) used.

2.15 Parallelism-aware architectures

As parallel algorithms add an extra layer of complexity, they introduce new limitations due to the physical underlying hardware configuration, and algorithm used. One such limitation is the impact of communication during the work : If a device needs the result of another device to perform the next computation, they have to communicate to share data. Communication can be expensive and slows down the whole process especially in Kog's case as will be shown in Section 3. Few works tackled this issue directly by changing the model's architecture, allowing for better parallelization of the work across multiple devices. The approach presented in [59] proposes to compute both the MLP and Attention parts of each decoder layer in parallel instead of sequentially, reducing the number of communications between devices during the computations. Another solution is presented in [60] and is close to our proposal, described in Section 4. It was published on August 24th 2024, at which point we were not aware of the concurrent work being conducted and had theorized our problem and designed the solution. Our own solution is more general that their proposition and aims to maximize the speed gains at a minimized model quality loss.

3 Problem Statement

This section will formalize our GPU computation modeling and apply it to transformer's inference modeling, demonstrating the architecture's impact on the efficiency of a single-node, multi GPU system, such as a NVIDiA DGX-H100 described in Figure 4.

3.1 GPU modeling

3.1.1 Single Device

In this report, we model a GPU as a computing device with compute capabilities and an internal memory to store data as showed in Figure 5. We will consider the memory to be homogeneous, ignoring the cache hierarchy. When given an operation to do, the GPU will first move the operands from memory to its compute units before performing the computation. In this setting, the relevant parameters to simulate the computation time are : The internal memory bandwidth

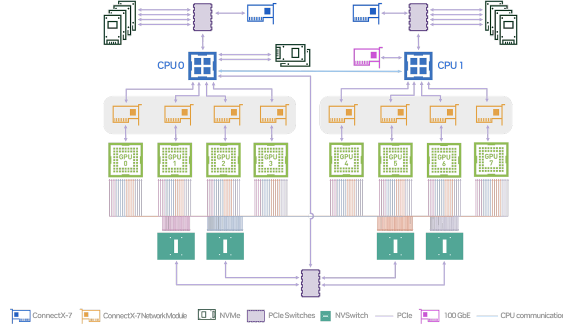


Figure 4: A single-node, multi-GPU system : NVIDIA's DGX-H100 system topology (Source: NVIDIA DGX-H100 [61]).

Figure 5: A Simple GPU modeling with compute units on top and memory (HBM) in the bottom (Source : NVIDIA blog post [62]).

β and the compute speed α. A mathematical operation between operands A and B is translated in our modeling to M bytes of data to move to the compute units (usually the sum of A and B's sizes) and C internal operations to perform with these data. The quantity C is defined by the mathematical operation (sum, multiplication...) and the computation kernel. In this setting, to compute the result a mathematical function f requiring C operations on A and B of combined size M with a bandwidth β and compute power α, we use the following formulas :

$$t_{move}(M) = M/\beta \tag{1}$$

$$t_{compute}(C) = C/\alpha \tag{2}$$

$$t_{device}(f, A, B) = t_{total}(M, c) = t_{move}(M) + t_{compute}(C) \tag{3}$$

One main strength of the GPU lies in its asynchronous execution of operations. Due to its highly parallel organization, a GPU can start preparing for the next operation before the current one is over. Leveraging that, operands can be moved while concurrently performing

computations. This concurrence leads to the execution of a sequence of operation being usually as fast as the slowest of either moving data or performing the computation, for each operation. In the case where the slowest process is moving data, the operation is memory bound. Otherwise, it is compute bound :

$$t_{device}(f, A, B) = \max(M/\beta, C/\alpha) \tag{4}$$

Given a GPU's bandwidth and compute power and a kernel's arithmetic intensity (operations per byte of data), the compute or memory bound scenario can be predicted according to the size of the operands.

3.1.2 Multiple Devices

To reduce the bottleneck's impact, the computations are distributed across multiple devices as described in Section 2.14. Matrix-Matrix and Matrix-vector multiplication can be chunked and performed in parallel on multiple device. Each device thus have less data to move and operations to do. Once the computations on each device is over, they can share their results together by communicating on a external link. This communication forces each device to synchronize internally and externally (ensure they and the other device(s) finished the computations) introducing a constant time t_{sync} before sharing M_{share} data using an external link with a bandwidth β_{inter}:

$$t_{communication}(M_{share}) = t_{sync} + M_{share}/\beta_{inter} \tag{5}$$

Leveraging again the asynchronous execution, if the result of a distributed computation is not immediately required by following computations, it can be performed in 'background' of other operations without losing device time, leading to Equation 4 being unchanged. Otherwise, there is a communication bottleneck on top of a smaller operations with smaller $M_{distributed}$ bytes of data to move and $C_{distributed}$ operations to perform across G GPUs:

$$\begin{aligned} t_{distributed}(f, A, B) &= t_{communication}(M_{share}) + t_{device}(f, A, B) \\ &= t_{sync} + M_{share}/\beta_{inter} + \max(M_{distributed}/\beta, C_{distributed}/\alpha) \end{aligned} \tag{6}$$

At which point the amount of data to be moved is reduced by a factor $\frac{M}{M_{distributed}}$ and computations to be performed by a factor $\frac{C}{C_{distributed}}$, at the cost of the communication time. The expected speed gain can be as high as the number G of GPUs to use (usually referred to as scaling factor), depending on the parallelism algorithm and computations to be performed.

To precisely simulate the transformer's inference, we will thus need to know if the inference is memory or bottlenecked, and wether there is a communication bottleneck or not. It is worth noting that the t_{sync} variable isn't usually released by manufacturers and not publicly available. Benchmarks in [63] report latencies within the range of 7-10 microseconds for the intra node V100 and P100 GPUs across the naive communication bus (PCIe) and manufacturer-provided (NVLink).

3.2 Transformer Inference Modeling

Now that our device modeling is set, we will study how performing the Transformer's computation behave on one and multiple GPUs, by highlighting bottlenecks in both cases and simulating their performance impact on the device's performance.

During the decode phase, only one token is added at a time, and only the hidden states from that token are required to predict the following one. Due to the KV-Cache, the (compute heavy)

attentions between tokens do not need to be recomputed for other tokens, greatly reducing the amount of computations to be performed. Multiple studies such as [64] show that inference is mostly compute-bound during the prefill phase, and completely memory bound during decode at low batch sizes. The computations are mostly matrix-vectors multiplication which a GPU performs them much faster than it can read the operands. Table 1 illustrates the sizes and repartition of each part of the model and in total. Each weight is stored in a chosen data format, called precision, usually on 2 bytes in the BF16 format as explained in Section 2.11.

Operand	**Shape**
W_q	(h, $n \times d$)
W_k	(h, $n_{\text{kv}} \times d$)
W_v	(h, $n_{\text{kv}} \times d$)
W_1	(h, d_{ffn})
W_2	(h, d_{ffn})
W_3	(d_{ffn}, h)
W_E	(vocab_size, h)
W_P	(vocab_size, h)
Layer block	**Total elements**
Attention	$2h^2 + 2hn_{kv}d + bsdn_{kv}$
FFN	$3hd_{\text{ffn}}$
Embedding Layer	$2 \times vocab_size \times h$
Total Elements	$n_{layers} \times (3hd_{\text{ffn}} + 2h^2 + 2hn_{kv}d + bsdn_{kv}) + 2 \times vocab_size \times h$

Table 1: Operands and Shapes / Element counts in the Transformer Architecture.

While such performance is usually more than enough for text applications, it lacks sufficient speed to achieve Kog's artifact generation goal. Memory bottleneck is also alleviated in most text generation use cases by increasing the batch size as much as the device's memory can hold, but Kog aims for maximal performance while keeping the batch size low. We will thus discuss the benefits of applying parallelism to reduce the memory bottleneck and highlight their limits.

Among the algorithms presented in Section 2.14, the one best suited to Kog's needs is Tensor Parallelism, as Model and Pipeline parallelism are meant to overcome memory constraints of the device, either due to model's size, KV Cache's size, or both. In Kog's case, the model and cache will fit on the device, and they would like to reduce the decode time. In the following sections, G denotes the number of devices used in parallel ; In a typical single node multi GPU system, G=2,4,8.

Tensor Parallelism involves partitioning individual computational blocks across the different devices and using communications to aggregate/distribute results. A descriptive figure of that algorithm, introduced in [65], can be found in Figure 6. In each layer, contiguous groups of Attention heads are placed across different devices and W_o matrix that is used to combine the output of the different heads is partitioned across rows (row parallelism). The output of the MHA block is retrieved by reducing the local output of all devices; this introduces one communication. Similarly, in the FFN block, the W_1 weight matrix is partitioned across rows and the W_2 weight matrix is partitioned across columns (column parallelism). The output of the FFN is computed by reducing the local output of all devices, thereby introducing another communication. By alternating row and column parallelization in this fashion, the reduction of data to be read from memory is almost optimal as barely every operands have a size divided by the number G of devices used, while only introducing 2 communications.

Table 2 presents an updated version of the transformer's parameters for each device using

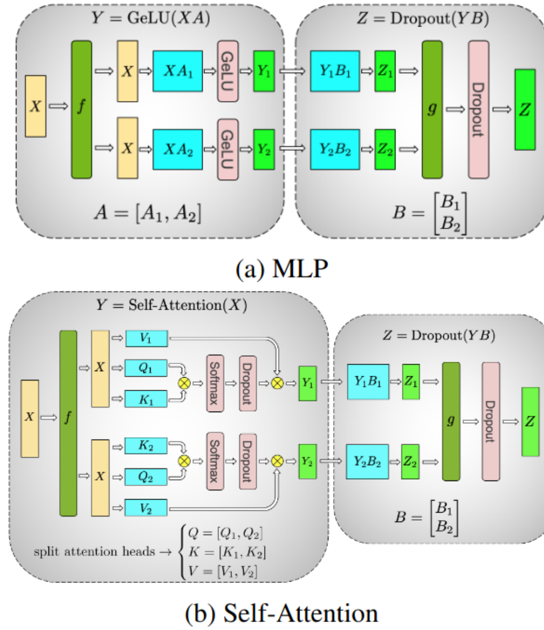


Figure 6: Tensor Parallelism algorithm as presented in [65]. Communications are represented by the g function.

Tensor Parallelism. The ceiling function highlights the need for every attention head to use at least KV head, which leads to KV heads being duplicated if they can't be correctly distributed across devices. The architecture should thus use a number of KV groups divisible by the number of devices used : The same amount of data will be moved but the model quality should increase as the model will have less redundancy.

3.3 Communication bottleneck

Applying Tensor Parallelism on G device to the sequential Transformer, communications are introduced. Tensor Parallelism inference is communication bottlenecked, on top of memory-bottlenecked operations. Each GPU will share and receive its own version of the hidden states of the last token with others representing a number of bytes of $h \times precision$. With a hidden size of 768 as in TinyLlama with a batch size of 1, this represents in INT8 precision less than 1Kb of data to share per GPU. Each GPU can share around 100Gb/s with other devices (depending on the actual model), and can thus share the data in a hundredth of microsecond. In comparison, the synchronization time t_{sync} ranges within few microseconds, orders of magnitude higher than the transfer. We will ignore the transfer time and consider performing a communication is as long as synchronizing the devices :

$$t_{communication}(M_{share}) = t_{sync} \tag{7}$$

We can then compare the time of performing a memory bottlenecked computation in distributed vs single device, assuming the distribution reduces the amount of data to move by G:

Operand	Shape
W_q	$(h, \lceil \frac{n}{G} \rceil \times d)$
W_k	$(h, \lceil \frac{n_{\text{kv}}}{G} \rceil \times d)$
W_v	$(h, \lceil \frac{n_{\text{kv}}}{G} \rceil \times d)$
KV Cache	$2 \times (b, \lceil \frac{n_{\text{kv}}}{G} \rceil, s, d)$
W_o	(h, h)
W_1	$(h, \frac{d_{\text{ffn}}}{G})$
W_2	$(h, \frac{d_{\text{ffn}}}{G})$
W_3	$(\frac{d_{\text{ffn}}}{G}, h)$

Table 2: Operands and Shapes / Element counts for Attention and FFN - Tensor parallelism.

$$\begin{aligned} t_{operation,parallel} < t_{operation,single} &\iff \frac{M}{G \times \beta} + t_{synch} < \frac{M}{\beta} \\ &\iff t_{synch} < \frac{M}{\beta} \times \frac{G-1}{G} \\ or &\iff M > \beta \times t_{synch} \times \frac{G}{G-1} \end{aligned} \tag{8}$$

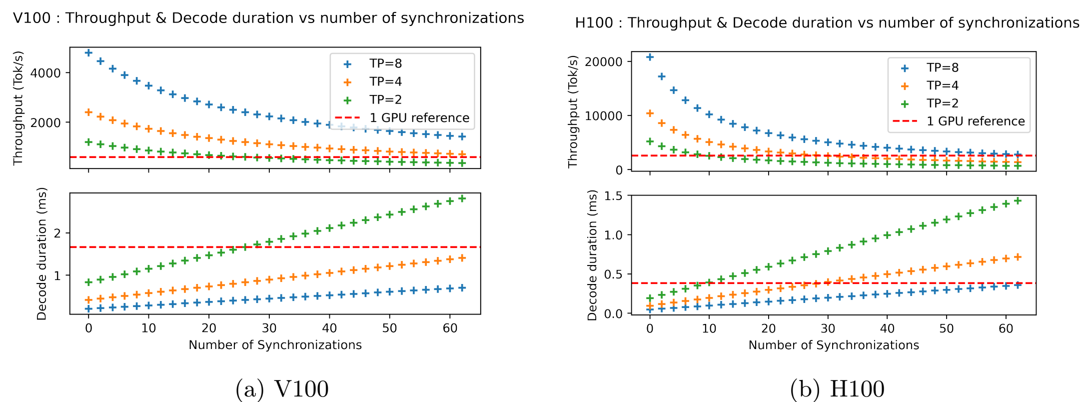
The inequalities in Equation 8 can be used by Kog to check if a model will see gains when used with Tensor Parallelism, gains with more GPUs, set a kernel optimization target to achieve speedups on a given inference system... When unsatisfied, the Tensor Parallelism algorithm can be modified to replicate part of the computations, thus eliminating a synchronization. In this scenario, more parameters could be allocated to either the Attention or FFN and see increased speedups the cost of an unbalanced / non optimal (with respect to Section 2.5) structure.

The exact architecture details (FFN dimension, hidden size) can be abstracted, only keeping the number of parameters and model depth. Considering a model of data size M (equals to the number of parameters times the precision used) and depth n_{layers}, its decode time using tensor parallelism will be :

$$t_{decode,parallel}(M, n_{layers}) = 2 \times n_{layers} \times t_{sync} + \frac{M}{\beta \times G} \tag{9}$$

Equation 9 demonstrates that for a given number of parameters the decode time grows linearly with the number of layers at a fixed number of total parameters (ignoring marginal impact of the KV Cache at batch size 1). Figure 7 plots the decode throughput using different counts of GPUs according to the number of synchronization for a same overall model size+KV cache of 2.22Gb. Since synchronization count grows with model depths, bigger models are condemned to poor Tensor Parallelism gains. Comparing the speeds from the V100 GPU (Figure 7a) with the H100 GPU (Figure 7b) highlights how the trends in next generation hardware will further increase this gap : Newer hardware tend to have bigger, faster memory, but device-to-device latency does not follow through.

Since the synchronization time remains the same even as more GPUs are added, adding more GPU leads to each GPU being less used as proportionally more time is spent synchronizing : The more GPUs used in parallel, the less efficient each one is. Similarly The smaller the model, the less efficient GPUs become. Figure 8 illustrates both phenomena, using the Model Bandwidth utilization (Percentage of peak theoretical memory bandwidth) as a measure of efficiency as the computation is memory bottlenecked.



(a) V100 (b) H100

Figure 7: Theoretical Decode time (bottom, lower is better) and Throughput (top, higher is better) vs Number of synchronizations with 1,2,4,8 GPUs.

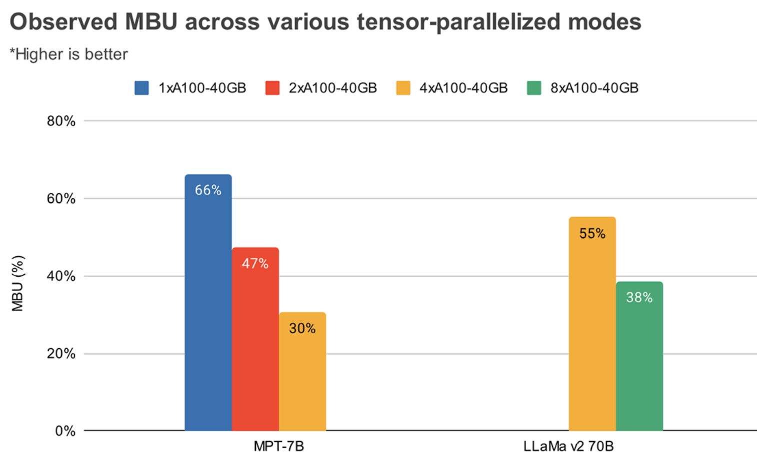


Figure 8: Individual GPU efficiency (higher is better) with Tensor Parallel degree 1,2,4,8 for a 7B model (left) and 70B model (right) using a single A100 GPU node.

As a result, models should use low depth to achieve the best tensor parallelism speed-ups. Conversely, scaling laws described in Section 2.5 suggest that reducing model depths would impact their accuracy

3.4 Modeling Validation

Single-GPU Real world comparison NVIDIA recently disclosed [66] a throughput of 1200 tokens/s with a batch size of 4 and a Llama2-7B model quantized to FP8 and a low initial context. The Llama2-7B architecture uses a hidden size of 4096, 32 KV heads (Multi Head Attention) and 32 layers. Considering a generation from almost no context to 2000 tokens and a batch size of 4, KV cache will on average weight 1.05Gb. Parameters will account for 7Gb memory space. Using our single device modeling and with β_{H100}=3.9Tb/s, we obtain an average decode time of 2.07ms. This leads to 485 decode steps per second, each generating 4 tokens : Total throughput of 1940 tokens/s. We attribute the gap between our modeling and the 1200 tokens announced to the device not achieving the peak theoretical memory bandwidth and small inaccuracies. Optimizing the memory readings performed by the computation kernels is tricky, highlighting challenges faced by Kog's kernel development team.

Multi-GPU comparison We focus on the Llama2-70B benchmarking presented in Figure 8 (right) on a system using 4 or 8 A100 GPUs (β=2Tb/s assuming $t_{sync} = 5\mu s$). They generate 512 tokens (starting from 0) with a batch size of 1. The model's parameters account for 70Gb data. With a hidden size of 4096, 80 layers and 8 KV heads, average KV cache amounts to 1.34Gb. In total, 71.34Gb of memory must be read at each decode step. With 4 GPUs, each GPU will read 17.8 Gb, and perform 160 synchronizations (duo to the 80 layers). Performing the synchronization will take $800\mu s$, while reading the 17.8Gb from memory will take 8.9ms. Synchronization time represents 9% of the total decode time. Scaling the modeling to 8 GPUs, the memory readings are reduced to 4.95ms while synchronization time remains unchanged. Synchronization time now represents 16% of the decode time, doubling from the 4 A100 scenario. In Databrick's findings, the memory goes from 55% on 4 GPUs to 38% on 8 GPUs. Taking into account kernel inefficiencies in achieving peak theoretical memory bandiwdth, the predicted loss of performance is close to the experimental value. In the case of the MPT-7B model (left), the NeoX architecture discussed in Section 2.15 is used. This architecture only introduces one synchronization per layer, which leads to less loss of performance when scaling on more GPUs.

This highlights architectural flaws within the Transformer architecture for fast distributed inference. How can we adapt the Transformer Architecture to mitigate or avoid this issue ?

4 Solution

Based on results in Section 2.5, we hypothesize we can distribute a given number of parameters while enforcing a parallel structure and achieve comparable generation capabilities with a Transformer of similar size. This section will present the proposed parallel architecture along with multiple tweaks we add to improve the generation quality of models based on it.

4.1 Tensor Parallel Blocks, Architecture

We propose to split the architecture based on Tensor Parallel Blocks presented in Figure 9. Each Tensor Parallel Block uses multiple parallel "lanes" of decoder layers. The associated forward algorithm is described in Algorithm 1 : input hidden states are distributed to the lanes using the *prepare* and gathered with *merge*.

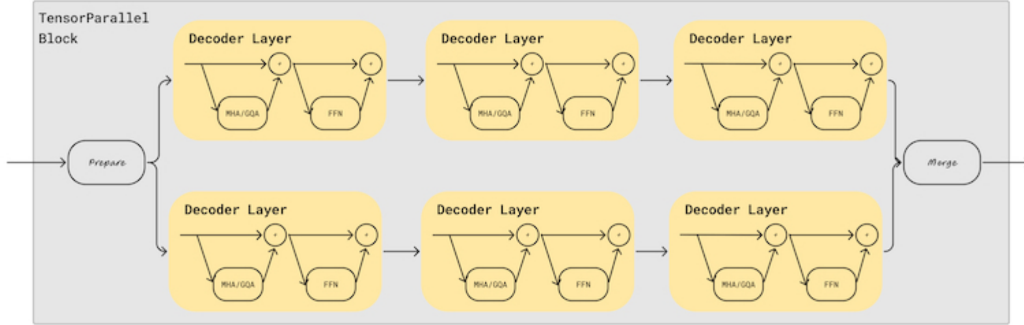


Figure 9: Proposed Tensor Parallel Block with 2 parallel lanes of depth 3.

Algorithm 1 Forward Pass for Tensor Parallel Block

```
1:  Input: lanes : List of lists of layers [n_lane, n_layer]
2:         hidden_states : Hidden states [b, s, d]
3:         lane_count : Number of parallel lanes
4:         n_layers_lane : Depth of each lane
5:  Output: outputs : Merged hidden states
6:
7:  function FORWARD(lanes, hidden_states, lane_count, n_layers)
8:      hidden_states ← PREPARE(hidden_states)
9:      for n_layer ← 0 to n_layers-1 do
10:         for lane ← 0 to lane_count-1 do
11:             hidden_states[lane] ← lanes[lane][n_layer](hidden_states[lane])
12:         end for
13:     end for
14:     outputs ← MERGE(hidden_states)
15:     return outputs
16: end function
```

The Tensor Parallel Blocks Block introduces new hyper parameters : The number of parallel lanes *lane_count*, the depth of each lanes *lane_depth*, the (*prepare*, *merge*) functions. The usual hyper parameters of each lane's Decoder Layer are also to be chosen. By setting the number of parallel lanes to a multiple of the number of devices used, lanes can be distributed at inference between devices and layers can be processed without any synchronization happening, as described in Figure 10. The number of lanes must be chosen at the definition of the model, implying that the model must be pre-trained before being used, and the number G of parallel devices must be known at training time. The associated models must use $G \times k$ lanes in parallel with $k \in \mathbf{N}+$.

Similarly to every parallel process, data must be scattered to the workers and results gathered afterward. This translates to the *prepare* and *merge* functions in our Tensor Parallel Block. We initially propose 2 gateways combinations:

- (broadcast, [sample, concatenate]), Figure 11a : Input hidden states are copied on every lane at the beginning of the lanes. Upon reaching the end of the lanes, a subset of each

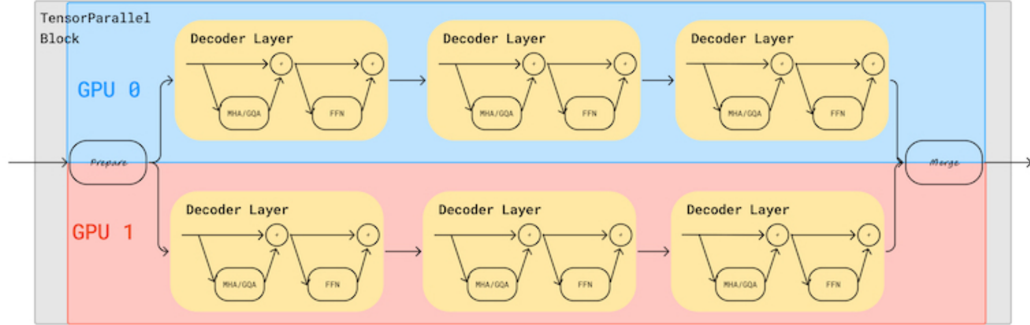
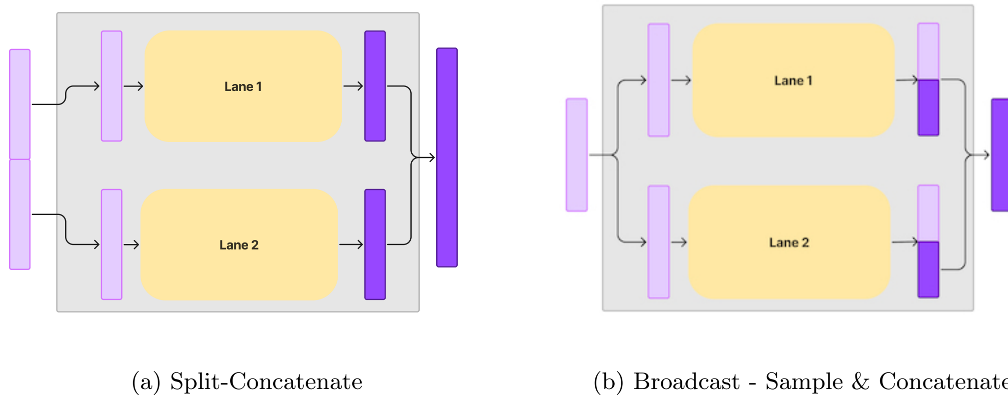


Figure 10: Tensor Parallelism using 2 GPUs on Tensor Parallel Block with 2 parallel lanes. Each GPU can process layers without synchronizations.

lane's hidden states is arbitrary sampled and concatenated to retrieve the initial shape.

- (split, concatenate), Figure 11b : Input hidden states are arbitrary split between the lanes at the beginning, and concatenated at the end.

The first combination keeps the hidden states smaller outside of the parallel block, reducing the number of parameters allocated to the embedding and projection layers.



(a) Split-Concatenate (b) Broadcast - Sample & Concatenate

Figure 11: Proposed Prepare-Merge combinations. Hidden states are represented in light and dark purple

4.2 Delayed Communication

To mitigate the impact of this reduction, we propose to reintroduce communications between devices, although in a delayed fashion. Starting at lane depth *communincation_start* and every *communication_interleave* layers further, each lane will share hidden states with the others, multiple layers deeper as set by the *communication_length* hyper parameter. We perform

communication by broadcasting the hidden states at initialization and summing them at reception. In the work presented in [60], *communication_start*=0, *communication_length*=1 and *communication_interleave*=0 are used, as represented in Figure 12. Leveraging the delayed use of the communication, GPUs can initiate and receive the communication while processing the lane layers. Since Kog knows its inference hardware, we can set the length of the delay to ensure it won't bottleneck the computations. For example, performing a 2Gb data inference (parameters + KV-cache) on 8 parallel lanes of 22 layers each leads to each decoder layer requiring 12.5Mb of memory read. A system using 8 H100 GPU ($t_s ync$= 5µs, β=3.9Mb/µs) would perform such communications for free if *communication_length* $\geq$ 1.56.

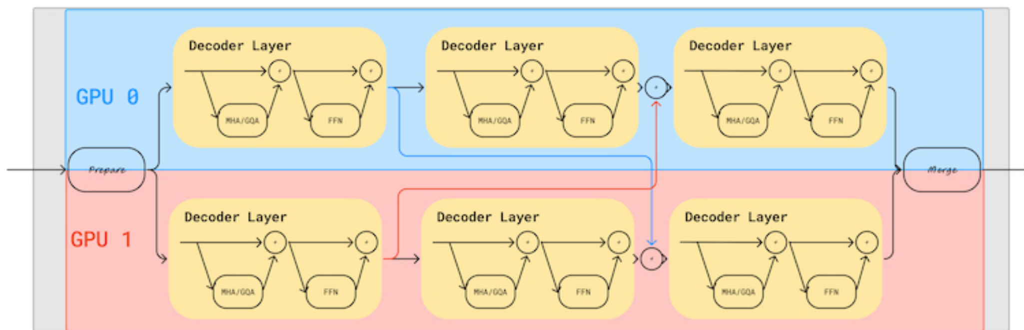


Figure 12: Delayed Synchronization example. Start=0, Length=1

4.3 Block Mixing

Finally, we can interleave Tensor Parallel Blocks with classic Decoder Layers to mitigate the impact of the parallel split on the model generation quality. This adds extra hyper parameters, namely *TP_block_repetition*, the number of Tensor Parallel Blocks repeated, with the addition of *n_classic_layers* classic dense layers in between as shown in Figure 4.3. The dense layers will use a different hidden size, depending on the gateway functions used in the Tensor Parallel Blocks. This reintroduces a few communications but allows creating and studying 'degraded' versions of various degree and compare them with our baseline.

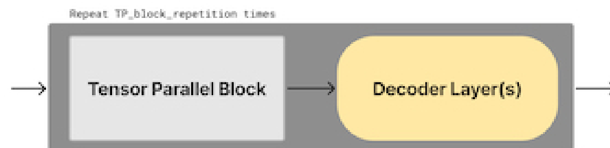


Figure 13: Hybrid Architecture.

4.4 Recipes

Sandwich and Burger We define 2 architectures template, or recipes, based on the 2 combinations presented in Section 4.1. We orthogonally use delayed communication described in Section 4.2 and block mixing from Section 4.3. In the following parts, models using the first recipe will be referred to as "Sandwich" models, whereas models using the 2nd recipe will be named "Burger".

FFN Projection in Burger recipe Burger models also contain a tweak in every lane's FFN : The W_3 down projection matrix of the FFN is truncated from $[d_{ffn}, h]$ to $[d_{ffn}, \frac{h}{lane_count}]$. This truncation, although unusual, is motivated by reproducing the Tensor Parallel mechanism : In a usual Transformer, each GPU computes part of the final hidden states, which the truncation reproduces, and the remaining hidden states are usually provided by the other GPUs. In our case, we delay such provision with the delayed communications : In the meantime, the remaining features of the hidden states are copied from before the FFN. While we chose to only apply this reasoning to the FFN in this report, we could have applied it to the attention layer too.

Summary Table 3 summarizes the key differences between both recipes.

Recipe Name	**Prepare**	**Merge**	**Lane to model hidden size factor**	**FFN W_3 projection to hidden size factor**
Sandwich	Split	Concatenate	$\frac{1}{lane_count}$	1
Burger	Copy	Sample + concatenate	1	$\frac{1}{lane_count}$

Table 3: Burger and Sandwich architectural differences.

4.5 Speed-up Gains

Both recipe remain bottlenecked by the total amount of data M to be moved and the number of synchronization required to perform the decode pass. Both recipes have one synchronization per tensor parallel blocks, and two per classic layers used. Assuming communications were sized to avoid communication bottleneck, we can update Equation 9:

$$t_{decode,parallel}(M, TP_block_repetition, n_classic_layers) = \\ TP_block_repetition \times (2 \times n_classic_layers + 1) \times t_{sync} + \frac{M}{\beta \times G} \quad (10)$$

A recipe minimizing the decode time would use only one tensor parallel block without block mixing, leading to the best case scenario where only one synchronization is required at the end of the lanes:

$$t_{decode,parallel}(M) = t_{sync} + \frac{M}{\beta \times G} \quad (11)$$

5 Setting up Training Protocol

While the inference speed gains are mathematically predictable, the impact of each recipe on the generation quality is hard to theoretically quantify. Experimenting the impact of each and every hyper parameter is essential to building the fastest model while keeping the Transformers' generation quality. To this end, it is essential to properly isolate and evaluate the impact of architectural changes. This requires a robust training protocol that minimizes external variability and ensures consistency across experiments along with a baseline reference. The objective of this work was to develop and refine such a protocol.

To this end, we iteratively build our training protocol by overcoming encountered issues. We introduce our initial training protocol in Section 5.1, followed by iterations performed to obtain comparative results between experiments in Section 5.3.

5.1 Initial Training Protocol

Following standard practices, we design our protocol to compare architectures at a relatively close number of parameters. We target a scale of 1 to 2 billions parameter final model and experiment with 150 million parameters models. We include the embedding parameters in the total count as they are relevant to our decode speed modeling.

We initially use the CodeParrot [67] dataset and code search net [68] tokenizer, with vocab size of 50,000. The dataset is filtered and tokenized using a context size of 128 tokens following [69], leading to 2.1B training tokens and a test set of 12M tokens. We perform initial trainings of 1 epoch (2.1B tokens) using a default learning rate of 0.0005 with a cosine scheduler doing 1 cycle over the whole train set. FP16 precision is used. We compare the losses over the train and test sets, without shuffling the dataset between experiments.

Models and training scripts are implemented using torch and transformers libraries. Experiments in this section were conducted using 1 A100-40Gb GPU.

Hyperparameter	**Value**
hidden_size	768
d_ffn	3072
n_layer	12
n_head	12

(a) GPT-2 Model

Hyperparameter	**Value**
hidden_size	768
d_ffn	1536
num_hidden_layers	12
num_attention_heads	12
num_key_value_heads	2

(b) Llama Model

Figure 14: Comparison of Hyperparameters for GPT-2 and Llama Models

5.2 First experiment wave

In our first proof of concept experiments, we use 2 baselines : a 125M GPT-2 model, a 135M Llama-2 model. The models hyper parameters are provided in Table 14. Baselines are trained with a Cross Entropy Loss and AdamW optimizer. We test a wide range of Burger based experiments, whose hyper parameters are describe in Table 4, with sizes ranging from 135M parameters to 147M parameters. The goal of this first wave is to identify training issues before conducting an extensive parameter search. Models have different batch size due to an issue in the experimental configurations. All models use the same tokenizer and vocabulary size, without embedding tying. When using multiple nodes or GPUs, we use Pytorch Distributed

Data Parallel (torchrun) to perform the training. We initially used Pytorch Data Parallel, but witnessed extremely poor performance scaling when using more GPUs.

	Burger-0	**Burger-1**	**Burger-2**	**Burger-3**	**Burger-4**
lane_count	8	8	8	8	8
lane_depth	4	8	6	4	15
n_classic_layers	1	1	1	1	1
TP parallel blocks	4	5	2	2	1
Attention Heads	8	8	8	8	8
KV heads (lanes)	1	1	1	1	1
KV heads (classic layers)	4	4	4	4	4
hidden size	384	256	320	512	384
classic layers d_ffn factor	1.5	1.5	1.5	1.5	1.5
lane layers d_ffn factor	2	1.5	1.33	1.5	1.5
vocab size	50000	50000	50000	50000	50000
epoch	1	1	1	1	1
Parameters count	147M	142.3M	135.1M	146.4M	135.9M

Table 4: Model configuration for initial Burger experiments

To this end, we report the training and test loss after one epoch training, and runtime on different hardware in Table 5. Results suggest our 'Burger-0' model beats the GPT-2 reference and matches the Llama2 model, and achieves the best scores overall. We report no overfitting between train and test sets, which is consistent with the use of only 1 epoch.

We identify several takeaways. Experiments do not exhibit the same loss, but they seem consistent at equal effective batch size. We chose to take into account embedding (in/out) parameters and use a vocabulary size of 50k tokens without embedding tying, embedding parameters quickly take a lot of space. In our Llama2 model (hidden size 768), 76.8M parameters are dedicated to embedding! In comparison, the Burger-0 architecture only has 38.4M embedding parameters. The 38.4M parameters were thus reallocated to more layers, probably explaining the close performance. Finally, we notice some burgers took much longer to train than others. We attribute that to the increased number of decoder layers, much higher than the hidden size reduction. Finally, we are unsure wether the task (code generation) is complex enough to make poor models fail, nor if 128 context size is enough. We would also like to ensure the models converged, but lack extra tokens to do so.

5.3 Strengthening Training Protocol

5.3.1 Comparison fairness

As a result of the previous limitations, we modify our training protocol to ensure two results can be fairly compared with each other :

- We change the training dataset to FineWeb-Edu Dedup, introduced by [70]. This increases the training data quality and available number of tokens.
- We take advantage of the training dataset change to use a lighter tokenizer : The Llama2-32k tokenizer, trained on general language, with a vocabulary size of 32000 tokens. Using

Arch	Parameter Count	Effective Batch Size (tokens)	Training System	Training Time	Train Loss	Test Loss
GPT-2	124.7M	32k	1 A100	14h20	1.14	1.07
Llama2	135.8M	32k	1 A100	8h	1.00	1.02
Burger-0	147M	32k	1 A100	35h40	0.99	1.01
Burger-1	142.3M	9.7k	4 x 8 MI250	4h	1.84	1.85
Burger-2	135.1M	9.7k	4 x 8 MI250	2h	1.92	1.94
Burger-3	146.4M	9.7k	4 x 8 MI250	1h40	1.60	1.62
Burger-4	135.9M	12.5k	2 x 8 MI250	11h	1.52	1.54

Table 5: Initial Experiments loss and duration. 1 epoch, 2.1B tokens

a smaller tokenizer leads to less parameters being attributed to the embeddings matrices.

- We select a higher context size of 2048 and tokenize the FineWeb-Edu dataset with the Llama2-32k tokenizer. We obtain 81B tokens. Although this is less than we expected (over 200B tokens), we randomly selected 50 samples (100k tokens), reconstructed them and found no issues. We split it to keep 400k tokens as a test set and save the dataset. No shuffling was applied between experiments.
- We apply embedding tying to every model, dividing by 2 the number of embedding parameters, reducing the impact of hidden size changes on parameters allocation between models.
- To bring more fairness to the comparison between experiments, we keep the effective batch size equal for all models. As this requires adapting the batch size to the number of devices used in parallel, we leverage gradient accumulation and compute the right value in the training script. We select 1M tokens as the effective batch size for all experiments, based on values used in [14].
- We also increase the learning rate to 0.003 (slightly above the 0.002 value used in MobileLLM) to converge and experiment faster.
- We drop the GPT-2 reference.
- We switch to BF16 as the much bigger batch size and higher learning rate lead to numerical instability (NaN gradients / 0 loss). Switching to BF16 removed the instabilities.

5.3.2 Compute Policy

We would like to reduce the training duration. To this end, we leverage model parallelism to split the workload across multiple GPUs and nodes. During this internship, we had access to 3 different platforms :

- INRIA ROMA's GPU machine with 2 x A100 GPUs.
- Adastra [71], a French supercomputer hosted at CINES. We implemented slurm submission scripts to launch experiments at scale.
- A private cluster with 3 GPU nodes with 8 H100 GPUs each. Training submission were done solely using torchrun.

Due to the heterogeneity between platforms, we chose not to dive in optimized inference framework to remain flexible and keep the same torchrun-based distributed training script across all platform. We implemented the architectures and experimental configurations using INRIA's machine or personal computer, before launching the experiments at a high scale on the supercomputers (mainly Adastra). This workflow allowed us to benefit largely from the clusters and iterate quickly. We mainly used the private solution as a backup. With an effective batch size of 2M tokens, a batch will at most contain 1024 sequences of 2048 tokens. We distribute the 1024 sentences across multiple GPUs. Using 4 nodes of 8 GPUs each, the required batch size per device is 16. Available GPUs (MI250, MI300A) quickly get underutilized due to too few samples being processed at a time for a 150M model. Depending on the available resources on the cluster, we used 4 to 6 nodes per experiment and let the cluster execute them. Launching multiple experiments in parallel keeps the scaling going on more GPUs.

While optimizing for maximal hours consumption was not the main objective, we report a maximum of 101 experiments using a total of 1800 GPU-hours in a 24h timeframe. This yields an experiment time of 33 minutes on average (4 nodes), showing the great capabilities of the experimental policies and allowing us to explore more of the hyper parameter space.

5.3.3 Dataset Size

Hyper Parameter	**Value**
lane_count	4
lane_depth	4
n_classic_layers	1
TP parallel blocks	2
Attention Heads	12
KV heads (lanes)	4
KV heads (classic layers)	4
hidden size	768
classic layers d_ffn factor	1.5
lane layers d_ffn factor	1.5

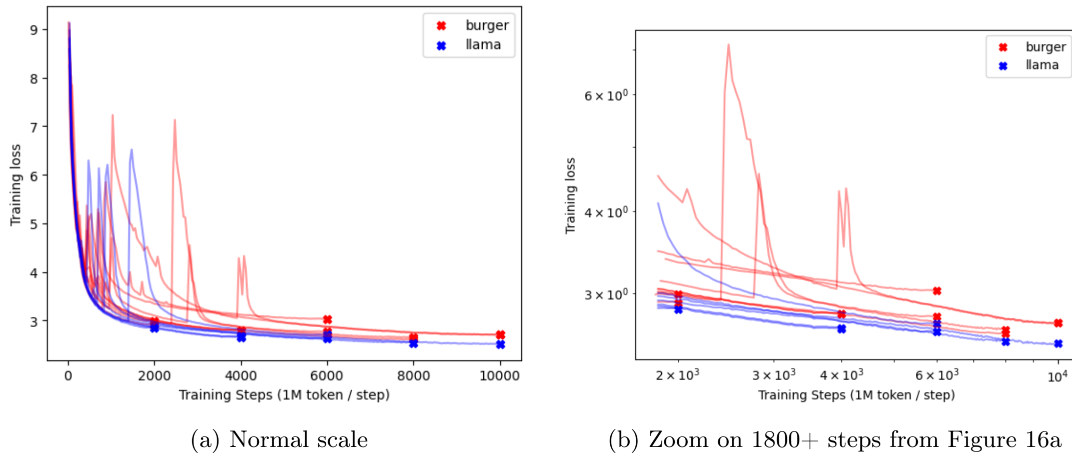
(a) Burger Model

Hyperparameter	**Value**
hidden_size	768
d_ffn	1536
num_hidden_layers	13
num_attention_heads	12
num_key_value_heads	4

(b) Llama Model

Figure 15: Llama and Burger configurations for Dataset Size experiment

We conducted a series of experiments to explore how early training could be stopped while maintaining reliable results. Specifically, we trained the same models on varying amounts of tokens and evaluated how soon the relative performance order, typically observed at the maximum training size, would emerge with fewer tokens. This approach allowed us to identify a potential threshold for early stopping that preserves the model's ranking and effectiveness, optimizing resource use without losing the desired information. In this scenario, we focus on the relative order of each model instead of the actual loss value. The model's configurations are described in Table 15, with Llama model in Table 15b, and Burger model in Table 15a. We reduced the lane count from 8 to 4 to lower the training time and experiment cost. Each model is trained independently on 2, 4, 6, 8 and 10B tokens. We ran the 4B and 6B experiment twice to assess possible variance due to the higher learning rate. The training loss, logged every 40 steps, and



(a) Normal scale

(b) Zoom on 1800+ steps from Figure 16a

Figure 16: Llama and Burger trained over 2B, 4B, 6B, 8B, 10B tokens. 4B and 6B experiments were launched twice to identify possible variance issues. Colored by model configuration.

final convergence point are plotted in Figure 16. We notice on Figure 16a that Llama always converges better than Burger. We also notice important spikes in the training during the high learning rate time frames. As this blocks us from correctly studying the early training dynamics, we would like to eliminate them. Furthermore, Figure 16b highlights at log scale important variance for burger. We believe this is also related to the learning rate being too high.

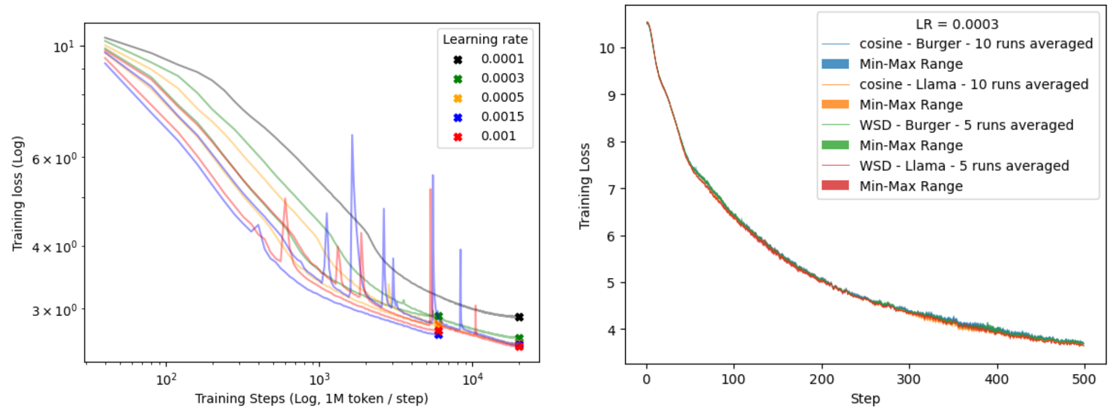
5.3.4 Learning Rate, Scheduler

The previous results hint to our learning rate being too high. We want to tune the learning rate value to find the right balance between convergence speed and training stability We also test the warmup-stable-decay (WSD or trapezoidal) scheduler used in [72, 73] to further increase the training time frame at high learning rate. Due to the stable part of the WSD scheduler, we also can restart a training from a checkpoint and pursue the training further than initially intended, which wouldn't be optimal the cosine scheduler set to one cycle. This particularity is especially interesting in the case of a final model where the training is expensive and we may need to extend the training for various reasons such as scaling law inaccuracies. Burger and Llama configurations used are unchanged (detailed in Figure 15).

We first perform a range search for various learning rate values with the cosine scheduler, by running the same architecture over 6B and 20B tokens with different learning rate value. No shuffling is performed, although we do not set the seed in model's initialization. We plot the training loss to visualize spikes in training and chose a learning rate.

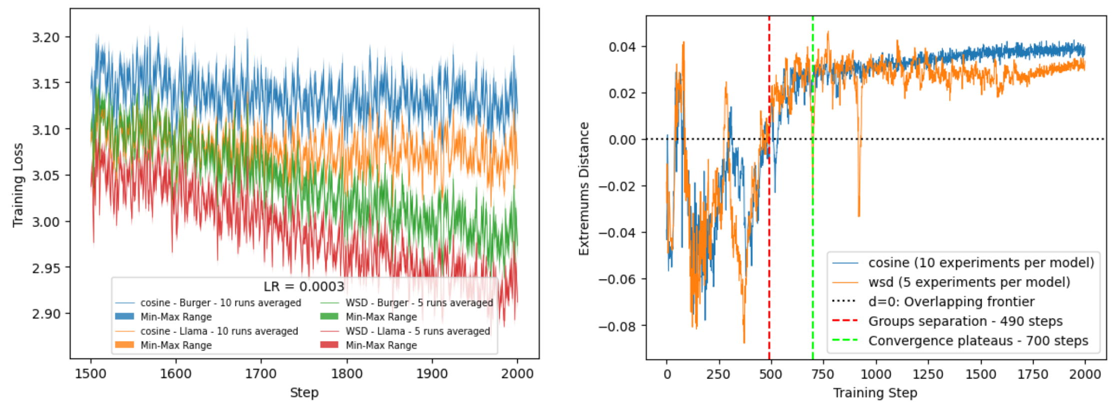
Based on the range search, we select one learning rate value and conduct a variance assessment. Both a Llama and Burger are launched 10 times using a cosine scheduler, and 5 times using a WSD scheduler. We lower the number of experiments for WSD as the whole experience was expensive with regards to our compute budget. We consider replacing WSD by cosine in the future to leverage the training extension capabilities, and would like to assess its stability. We log the training loss at every step (1M tokens). We then evaluate the mean variance for each combination of (model, scheduler) at each step.

Results are provided in Figure 17. Starting with the learning rate range search in Figure 17a, experiments confirm higher learning rates converge faster to a given training loss, at the cost of



(a) Training losses obtained with various learning rates, Burger, 2B tokens.

(b) Training losses with LR=0.003, Burger and Llama models, WSD and cosine schedulers. Average and extremums per scheduler and model.



(c) Zoom on Figure 17b, last 500 steps.

(d) Distance between models extremums, per shceduler. WSD in orange, Cosine in blue. A positive distance indicates no overlap between group of models.

Figure 17: Learning rate and variance analysis.

spikes in training loss. Based on this figure, we selected a learning rate of 0.0003 as candidate for our variance assessment and future experiments. We also considered using 0.0005, but discarded the value to keep some margin of error. Comparing this value to the ones used in [72, 14], and others, the optimal learning rate for our architecture is lower.

Moving on to the variance experiment with the selected learning rate of 0.0003, the averaged training losses and extrememums (min/max for the 10 trainings) are plotted in Figure 17b, colored by combination of architecture (burger or llama) and scheduler (cosine or WSD). We notice no unexpected behavior nor spike in any of the results, confirming 0.0003 as a stable learning rate in our setting. Interestingly, WSD scheduler (red, green) appears to provide better end of training results than cosine scheduler (blue, orange) as seen in Figure 17c.

It is unclear on the previous figures if each groups of model can be easily separated. We argue that the lack of smoothness is introduced by the high learning rate (0.0003) and effective batch size (1M) combination. To eliminate this scaling effect, we retrieve at each step the [min loss, max loss] extremums per model, and compute the extremum distance using :

$$d_{extremums}(min_{Bugers}, max_{Burgers}, min_{Llamas}, max_{Llamas}) = \\ -(\min(max_{Burgers}, max_{Llamas}) - \max(min_{Burgers}, min_{Llamas})) \quad (12)$$

We plot the results in Figure 17d. The extremum distance appears mostly positive above 490 training steps. The distance increases up to 700 steps, at which point it mostly plateaus. Interestingly, this behavior is shared between both schedulers. We thus attribute the plateauing to the relative convergence of each architecture. WSD still exhibits a slightly less stable behavior.

Based on these results, we considered using a truncated cosine scheduler (reduce the number of cycle to less than 1) to reach the convergence plateau and stop the training. This would result in a schedule between the ones tested in this WSD and cosine experiment. We also considered launching multiple experiments with WSD on a smaller training set, leveraging an average training loss instead one loss. Combining this idea with a highly parallel experimental platform, we could launch multiple experiments at the same time and obtain final results faster.

Finally, we kept the previous ideas for later work and decided to use the cosine scheduler with 1 cycle over 1B tokens (1000 training steps) for short experiments, increased to 2B if we consider the tested hyper parameters values to be too extreme to reach proper convergence in 1000 steps.

6 Preliminary Architectural Study

6.1 Methodology

Our final training protocol is defined in Table 6. We use the usual Cross Entropy Loss with a Causal Language Modeling task. We log the training loss at every training step, and use this value to conclude on each experiments. We keep the evaluation loss to check if any result exhibits abnormal behavior. In each of the following experiments, we set all the parameters but the one parameter or combination of parameters. We then adapt the hidden size to achieve a similar total number of parameters (embedding included). We used a total budget of 135M or 150M according to the experience. We always use an intermediate size double the hidden size (for both lanes and classic decoder layers). The change of value, and some experiments missing are a result of training failures for various reasons during this preliminary study. They will be relaunched in the next iteration.

Hyperparameter	Value
Dataset	FineWeb-Edu-dedup
Available Data	81B tokens
Tokenizer	Llama2-32k
Vocabulary Size	32001
Scheduler	Cosine
Learning Rate	0.003
Scheduler Warm-up	10% steps
Weight Decay	0.1
Optimizer	AdamW
Effective Batch Size	1M tokens
Training Steps	1,000 (1B tokens)
Precision	BF16

Table 6: Training Protocol

6.2 Lane Count

We first study the training loss vs the number of lanes in parallel. This series of experiments are motivated by 2 goals:

- We demonstrated the requirements of using a number of lanes multiple of the number of GPUs used in parallel. We would like to see how using more than one lane per GPU behaves. While we expect 1 lane per GPU to be the best, it might not be the case. Considering a similar question of sizing the attention heads, studies in Section 2.5 proved the existence of an optimal range of values.
- We observed in Section 5.1 that using more lanes increased the training budget. We hypothesize that using more lanes will reduce the performance of the model, and vice versa. If that hypothesis is confirmed, we'll be able to experiment with only 2 or 4 lanes instead of 8 (or more), and conduct experiments faster, while extrapolating the results to the inference speed optimal value.

We perform experiments using lane_count=1, 2, 4 8, 16, with a lane depth of 12, plus one classic decoder layer. Hidden size is adjusted to keep the parameters count within budget.

Figure 18 plots the final training loss according to the number of lanes. The similar results using 1 lane was expected as no prepare/merge happened. Sandwiches outperform burgers at every lane, which we attribute to the burgers using a higher hidden size across the lanes. Results confirm using more lanes increases the loss. Based on this confirmation, we chose to conduct the following experiments using 4 parallel lanes, to reduce experiment time. We use different hyper parameters than the Llama Baseline (such as the lane d_ff coefficient), explaining the difference between it and the 1 Lane Baseline. Without considering this difference, there isn't much different between 1 and 8 lanes.

6.3 Lane Aspect Ratio

In a similar fashion to the Transformer architecture analysis, we study the impact of aspect ratio (ratio between lane hidden size and the depth of the lane) on the training loss. This serves 2 needs :

- Understanding this will allow us to scale experiments to a bigger size for Kog's product.

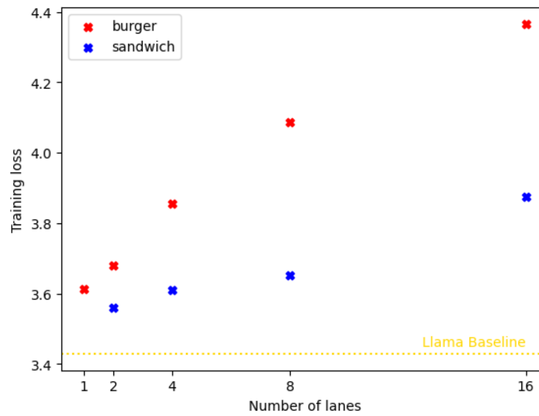


Figure 18: Training Loss vs Number of lanes in parallel. All models using 12 depth in lane, plus one classic decoder layer. 1B training tokens.

- We want to see if, as per the results in Section 2.5, an acceptable range of value exist for our architecture. We will use this range to combine with the delayed communications : The deeper the model, the more delayed communications we can use.

All models use 4 lanes in parallel, 1 Tensor Parallel Block, 1 classic decoder layer. We set a model depth and adjust the hidden size to keep the parameters count within budget. Burger models use (lane depth, lane hidden size): (8, 688), (20,448), (32, 368). Sandwiches : (3, 560), (8, 460), (16, 368), (20, 348), (32, 292).

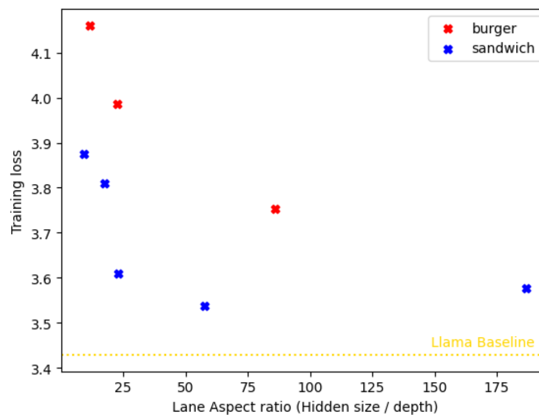


Figure 19: Training loss according to Lane Aspect Ratio. All models containing 1 Tensor Parallel Block with 4 parallel lanes, plus 1 decoder layer.

Results are displayed in Figure 19 : The range of values between sandwich and burger are not the same as we first set the lane depth, then computed the adjusted hidden size, and used the same lane depths for both architecture. The split function leads to such differences between hidden sizes. We notice sandwiches achieve the same loss with a lower lane aspect ratio. While we see trends in this figure, we believe it should be completed by further experiments to cover more values for both architectures. Another possible explanation is the impact of the merging

algorithm for burger ; Burgers generally had a slightly higher lane hidden size than sandwich, but a much lower out of lane hidden size.

6.4 Number of Tensor Parallel Blocks

We experiment the impact of keeping some hard synchronization in the model. This translates in our architecture to the number of Tensor Parallel Blocks used.

- As demonstrated in Section 3, communications greatly reduce the inference speed. We want to know if we can completely eliminate them, or still need to keep some in the architecture.
- We can always build a bigger model without synchronization, and compare it with a loss-equivalent model, and quantify the cost of splitting it in numbers of parameters.

We design the experiments by changing the number of tensor parallel blocks while keeping the sum of all Tensor Parallel Blocks' depth equal to 12. This leads to a higher total model depth for models with a higher number of Tensor Parallel Blocks, as the total depth is equal to 12 plus the number of Tensor Parallel Blocks. The number of synchronizations for these experiments is equal to 3 per tensor parallel blocks used (1 for the TP block itself plus 2 for the classic decoder layer that follows each of them).

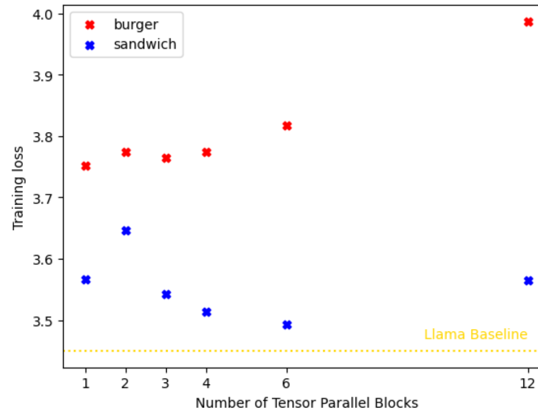


Figure 20: Training loss vs Number of Tensor Parallel Blocks. Lane depth × number of TP blocks kept constant equal to 12.

We plot the results in Figure 20. The main conclusion we draw from this experience is the number of tensor parallel blocks does not appear to influence a lot on the final training loss. As shown in Section 4, using less tensor parallel blocks maximizes the inference decode speed. Experience's results suggest we could only use one of them and still keep most of the performance. We also notice that the sandwiches are systematically better in that experience, but this comparison is not the first objective of the experience. This last result maybe hints at convergence issue, which we need to investigate more, although the cost of scaling experiments to higher training size was not affordable during the internship.

6.5 Number of sequential classic Decoder Layers

We would like to see if instead of interleaving one decoder layer every Tensor Parallel Block, we should interleave multiple of them. Both schemes are close in decode inference speed, which

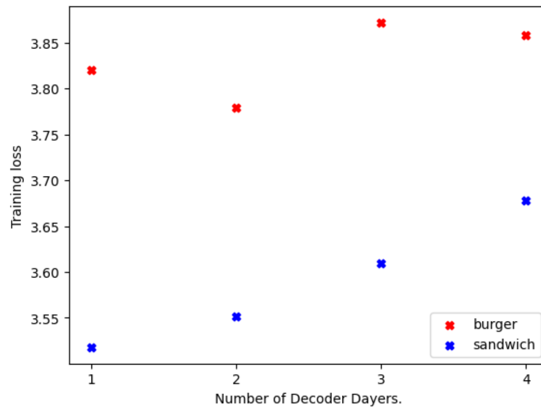


Figure 21: Training Loss vs number of decoder layers after each Tensor Parallel Blocks (4 TP Blocks, Lane depth 3, 1B tokens).

makes us wonder if we can make better use of the synchronizations by placing them in a different way. To this end, we set a model with 4 blocks of 4 parallel lanes, each lane with a depth of 3. We then place 1, 2, 3, 4 classic decoder blocks, and adjust the hidden size to keep the total parameter count equal or close.

Figure 21 shows using more decoder layers increases the training loss for a given number of tensor parallel blocks. We attribute this to the reduction of overall hidden size, over constraining the model. This phenomenon is much more pronounced within the sandwiches than the burgers. Using only one decoder layer after each tensor parallel blocks is good decode speed-wise, as it reduces the number of expensive synchronization to perform for the inference hardware.

6.6 Delayed Information

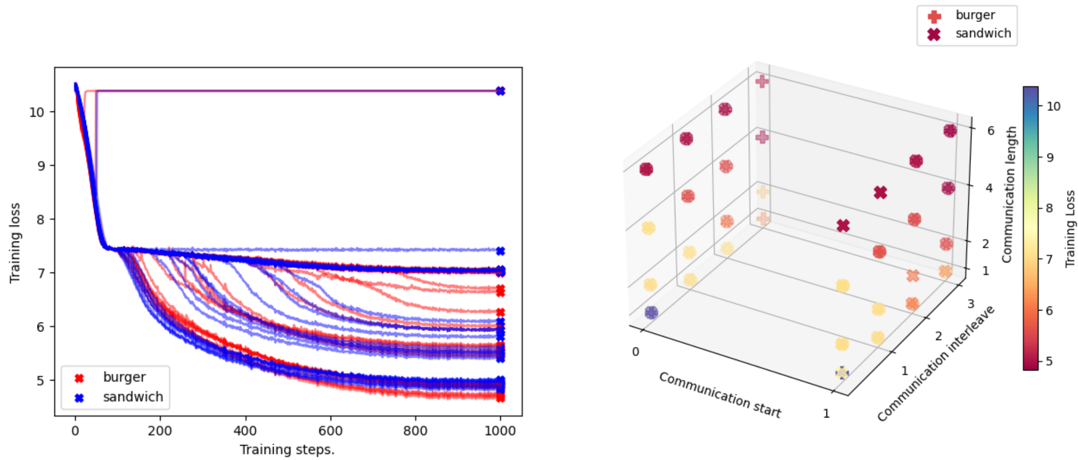
Finally, we test the impact of various combinations of the delayed communication's parameters: start, interleave, length (see Section 4.2) on the training loss on the same model. For these experiments, we test all combinations using communication start in $\{0, 1, 2\}$, interleave in $\{0, 1, 2, 3\}$ and length in $\{1, 2, 4, 6\}$. All other architectural parameters are the same. Communications are performed by broadcasting hidden states at initialization and summing them at reception.

Results are shown in Figure 22. We believe from the high values in Figure 22a that our implementation leads to gradient issues due to the summing, even if hidden states are normalized (RMS Normalization) after reception. We still represent an analysis in Figure 22b, in which longer communications are performing better. Although we take this result without confidence, it would fit well with increasing the decode speed : Hardware will enforce the communication length in order to achieve it in the background.

6.7 Larger Scale, Limitations

Based on the previous preliminary results in Section 6, we submitted to Kog the following rules to scale our architecture to 1.5B parameters, meant for a 8-GPUs inference node :

- 8 Parallel Lanes
- 1 Tensor Parallel Block



(a) Overall Training losses, 1B tokens. Training losses are much higher than the other experiments.

(b) Communication (start, interleave, length) 3d plot. Colored by training loss.

Figure 22: Delayed Communications experiments.

- 1 Final Decoder Layer
- 8 KV Heads in the shared decoder layer, 1 per lane decoder layer
- Scale lane depth and hidden size while keeping an aspect ratio of 100 for Sandwich, ≈ 400 for Burger.
- No delayed communications (yet).

We declare a few limitations with regards to this set of rules. Based on other papers, the training dataset should either be reused (multiple epochs) or expanded by using other available dataset. The tokenizer should also be retrained on that target dataset. We believe delayed communications should bring better performances to the model, although our first experiments did not show it. In a future iteration, we would like to change the initialization and reception to avoid gradient issues. We launched a 1.5B experiment following the previous rules, although it faced hardware failures and could not be finalized yet. This model would only require 3 synchronizations per decode pass. Considering a total size (KV-Cache+Parameters) of 2.2Gb (similar to the TinyLlama 1.5B), a 8 H100 inference server would compute a decode step in 70us, leading to 14181 tokens/s. In comparison, the TinyLlama and its 22 hidden layers would require 290us and achieve a throughput of 3442 tokens/s. Our model, subject to kernel optimizations, would achieve a 4.2 fold decode speed increase ! As highlighted in Section 3, this value is also likely to get bigger as memory bandwidth increases at a much higher pace than inter device latency, future-proofing our design.

7 Conclusion

This internship was done in the context of an INRIA-Kog collaboration; its objective was to improve the inference speed of Transformer based LLMs. To this end, we studied the available techniques in Section 2. We focused on the throughput of a single-node, multi-GPU distributed

inference with a low batch size. To better understand the inference dynamics, we introduced and validated in Section 3 formulas to predict the inference time of a Decoder-only Transformer using Tensor Parallelism. Based on this simulation, we demonstrate the impact of the number of layers on the poor performance scaling of a multi-GPU node, due to expensive synchronization times. We also highlight specific hyper parameters rules that are best suited to the multi-GPU node, such as the number of KV-Head. We proposed in Section 4 an alternative to the inherently sequential Transformer architecture, which relies on the parallel organization of the Transformer architecture based on Tensor Parallel Blocks. We propose to reduce the impact of this alternative architecture on prediction quality by keeping communication between layers, although in a delayed fashion. We also consider mixing Tensor Parallel Blocks with usual architecture, introducing a trade-of between inference throughput and model quality. We settle in Section 4.4 on two main templates, namely Burger and Sandwich architectures. We develop in Section 5 a robust training protocol and resource management policy. This protocol and resource management policy allowed us to perform up to 101 experiences within a day without compromising on experiments convergence, delivering to Kog an industrial grade experimentation platform. Leveraging the platform, we conduct in Section 6 a study over the hyper parameters of Burgers and Llamas. Based on this exploration, we propose hyper parameters rules for either a Sandwich or Burger based 1.5B model, a hybrid Tensor Parallel Based model, and theorize a 4.2 speedup gain compare to a size-equivalent Llama based model on 8xH100 GPUs at batch size 1, based on architecture alone.

Acknowledgements

This work was granted access to the HPC resources of IDRIS under the allocation 2024-A0171015576 made by GENCI.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” Aug. 2023, arXiv:1706.03762 [cs]. [Online]. Available: http://arxiv.org/abs/1706.03762

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” OpenIA blog, 2019. [Online]. Available: https://insightcivic.s3.us-east-1.amazonaws.com/language-models.pdf

[3] N. Shazeer, “GLU Variants Improve Transformer,” Feb. 2020, arXiv:2002.05202 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2002.05202

[4] B. Zhang and R. Sennrich, “Root Mean Square Layer Normalization,” Oct. 2019, arXiv:1910.07467 [cs]. [Online]. Available: http://arxiv.org/abs/1910.07467

[5] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, “RoFormer: Enhanced Transformer with Rotary Position Embedding,” Nov. 2023, arXiv:2104.09864 [cs]. [Online]. Available: http://arxiv.org/abs/2104.09864

[6] “LLaMA.” [Online]. Available: https://research.facebook.com/publications/llama-open-and-efficient-foundation-language-models/

[7] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," Feb. 2023, arXiv:2302.13971 [cs]. [Online]. Available: http://arxiv.org/abs/2302.13971

[8] "LLama 3," Jun. 2024, original-date: 2024-03-15T17:57:00Z. [Online]. Available: https://github.com/meta-llama/llama3

[9] V. Yaadav, "Exploring and building the LLaMA 3 Architecture : A Deep Dive into Components, Coding, and…," Apr. 2024. [Online]. Available: https://medium.com/@vi.ai_/exploring-and-building-the-llama-3-architecture-a-deep-dive-into-components-coding-and-43d4097cfbbb

[10] Y. Liu, J. Cao, C. Liu, K. Ding, and L. Jin, "Datasets for Large Language Models: A Comprehensive Survey," Feb. 2024, arXiv:2402.18041 [cs]. [Online]. Available: http://arxiv.org/abs/2402.18041

[11] "FEB94 A New Algorithm for Data Compression." [Online]. Available: http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM

[12] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," Jan. 2020, arXiv:2001.08361 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2001.08361

[13] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. v. d. Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training Compute-Optimal Large Language Models," Mar. 2022, arXiv:2203.15556 [cs]. [Online]. Available: http://arxiv.org/abs/2203.15556

[14] Z. Liu, C. Zhao, F. Iandola, C. Lai, Y. Tian, I. Fedorov, Y. Xiong, E. Chang, Y. Shi, R. Krishnamoorthi, L. Lai, and V. Chandra, "MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases," Jun. 2024, arXiv:2402.14905 [cs]. [Online]. Available: http://arxiv.org/abs/2402.14905

[15] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7B," Oct. 2023, arXiv:2310.06825 [cs]. [Online]. Available: http://arxiv.org/abs/2310.06825

[16] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. T. Lee, "Phi 1.5," Sep. 2023, arXiv:2309.05463 [cs]. [Online]. Available: http://arxiv.org/abs/2309.05463

[17] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu, "Qwen Technical Report," Sep. 2023, arXiv:2309.16609 [cs]. [Online]. Available: http://arxiv.org/abs/2309.16609

[18] P. Zhang, G. Zeng, T. Wang, and W. Lu, "TinyLlama: An Open-Source Small Language Model," Jun. 2024, arXiv:2401.02385 [cs]. [Online]. Available: http://arxiv.org/abs/2401.02385

[19] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “LoRA: Low-Rank Adaptation of Large Language Models,” Oct. 2021, arXiv:2106.09685 [cs]. [Online]. Available: http://arxiv.org/abs/2106.09685

[20] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, “Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment,” Dec. 2023, arXiv:2312.12148 [cs]. [Online]. Available: http://arxiv.org/abs/2312.12148

[21] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, “Medusa - Mutiple Decoding Heads,” Jan. 2024, arXiv:2401.10774 [cs]. [Online]. Available: http://arxiv.org/abs/2401.10774

[22] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, “Speculative Sampling,” Feb. 2023, arXiv:2302.01318 [cs]. [Online]. Available: http://arxiv.org/abs/2302.01318

[23] ——, “Accelerating Large Language Model Decoding with Speculative Sampling,” Feb. 2023, arXiv:2302.01318 [cs]. [Online]. Available: http://arxiv.org/abs/2302.01318

[24] T. Schuster, A. Fisch, J. Gupta, M. Dehghani, D. Bahri, V. Q. Tran, Y. Tay, and D. Metzler, “Confident Adaptive Language Modeling,” Oct. 2022, arXiv:2207.07061 [cs]. [Online]. Available: http://arxiv.org/abs/2207.07061

[25] N. Shazeer, “Multi Query Attention,” Nov. 2019, arXiv:1911.02150 [cs]. [Online]. Available: http://arxiv.org/abs/1911.02150

[26] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, “Grouped Query Attention: Training Multi-Query Transformer Models from Multi-Head Checkpoints,” Dec. 2023, arXiv:2305.13245 [cs]. [Online]. Available: http://arxiv.org/abs/2305.13245

[27] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, “KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization,” Apr. 2024, arXiv:2401.18079 [cs]. [Online]. Available: http://arxiv.org/abs/2401.18079

[28] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, “Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention,” Aug. 2020, arXiv:2006.16236 [cs]. [Online]. Available: http://arxiv.org/abs/2006.16236

[29] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The Efficient Transformer,” Feb. 2020, arXiv:2001.04451 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2001.04451

[30] I. Beltagy, M. E. Peters, and A. Cohan, “Sliding Window Attention: Longformer,” Dec. 2020, arXiv:2004.05150 [cs]. [Online]. Available: http://arxiv.org/abs/2004.05150

[31] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating Long Sequences with Sparse Transformers,” Apr. 2019, arXiv:1904.10509 [cs]. [Online]. Available: http://arxiv.org/abs/1904.10509

[32] A. Gu and T. Dao, “Mamba: Linear-time sequence modeling with selective state spaces,” 2024. [Online]. Available: https://arxiv.org/abs/2312.00752

[33] S. De, S. L. Smith, A. Fernando, A. Botev, G. Cristian-Muraru, A. Gu, R. Haroun, L. Berrada, Y. Chen, S. Srinivasan, G. Desjardins, A. Doucet, D. Budden, Y. W. Teh, R. Pascanu, N. De Freitas, and C. Gulcehre, "Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models," Feb. 2024, arXiv:2402.19427 [cs]. [Online]. Available: http://arxiv.org/abs/2402.19427

[34] A. Botev, S. De, S. L. Smith, A. Fernando, G.-C. Muraru, R. Haroun, L. Berrada, R. Pascanu, P. G. Sessa, R. Dadashi, L. Hussenot, J. Ferret, S. Girgin, O. Bachem, A. Andreev, K. Kenealy, T. Mesnard, C. Hardin, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, A. Joulin, N. Fiedel, E. Senter, Y. Chen, S. Srinivasan, G. Desjardins, D. Budden, A. Doucet, S. Vikram, A. Paszke, T. Gale, S. Borgeaud, C. Chen, A. Brock, A. Paterson, J. Brennan, M. Risdal, R. Gundluru, N. Devanathan, P. Mooney, N. Chauhan, P. Culliton, L. G. Martins, E. Bandy, D. Huntsperger, G. Cameron, A. Zucker, T. Warkentin, L. Peran, M. Giang, Z. Ghahramani, C. Farabet, K. Kavukcuoglu, D. Hassabis, R. Hadsell, Y. W. Teh, and N. de Frietas, "RecurrentGemma: Moving Past Transformers for Efficient Open Language Models," Apr. 2024, arXiv:2404.07839 [cs]. [Online]. Available: http://arxiv.org/abs/2404.07839

[35] O. Lieber, B. Lenz, H. Bata, G. Cohen, J. Osin, I. Dalmedigos, E. Safahi, S. Meirom, Y. Belinkov, S. Shalev-Shwartz, O. Abend, R. Alon, T. Asida, A. Bergman, R. Glozman, M. Gokhman, A. Manevich, N. Ratner, N. Rozen, E. Shwartz, M. Zusman, and Y. Shoham, "Jamba: A Hybrid Transformer-Mamba Language Model," Mar. 2024, arXiv:2403.19887 [cs]. [Online]. Available: http://arxiv.org/abs/2403.19887

[36] W. Wang, W. Chen, Y. Luo, Y. Long, Z. Lin, L. Zhang, B. Lin, D. Cai, and X. He, "Model Compression and Efficient Inference for Large Language Models: A Survey," Feb. 2024, arXiv:2402.09748 [cs]. [Online]. Available: http://arxiv.org/abs/2402.09748

[37] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," Mar. 2020, arXiv:1910.01108 [cs]. [Online]. Available: http://arxiv.org/abs/1910.01108

[38] S. Muralidharan, S. T. Sreenivas, R. Joshi, M. Chochowski, M. Patwary, M. Shoeybi, B. Catanzaro, J. Kautz, and P. Molchanov, "Compact Language Models via Pruning and Knowledge Distillation," Jul. 2024, arXiv:2407.14679 [cs]. [Online]. Available: http://arxiv.org/abs/2407.14679

[39] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale," Nov. 2022, arXiv:2208.07339 [cs]. [Online]. Available: http://arxiv.org/abs/2208.07339

[40] Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, and J. E. Gonzalez, "Train large, then compress: Rethinking model size for efficient training and inference of transformers," *CoRR*, vol. abs/2002.11794, 2020. [Online]. Available: https://arxiv.org/abs/2002.11794

[41] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "MoE introduction," Jan. 2017, arXiv:1701.06538 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1701.06538

[42] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E.

Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui, "GLaM: Efficient Scaling of Language Models with Mixture-of-Experts," Aug. 2022, arXiv:2112.06905 [cs]. [Online]. Available: http://arxiv.org/abs/2112.06905

[43] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mixtral of Experts," Jan. 2024, arXiv:2401.04088 [cs]. [Online]. Available: http://arxiv.org/abs/2401.04088

[44] Z. Su, F. Mo, P. Tiwari, B. Wang, J.-Y. Nie, and J. G. Simonsen, "Mixture of Experts Using Tensor Products," May 2024, arXiv:2405.16671 [cs]. [Online]. Available: http://arxiv.org/abs/2405.16671

[45] Q. Anthony, Y. Tokpanov, P. Glorioso, and B. Millidge, "BlackMamba: Mixture of Experts for State-Space Models," Feb. 2024, arXiv:2402.01771 [cs]. [Online]. Available: http://arxiv.org/abs/2402.01771

[46] R. Hwang, J. Wei, S. Cao, C. Hwang, X. Tang, T. Cao, and M. Yang, "Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference," Apr. 2024, arXiv:2308.12066 [cs]. [Online]. Available: http://arxiv.org/abs/2308.12066

[47] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "DeepSpeed-MoE," Jul. 2022, arXiv:2201.05596 [cs]. [Online]. Available: http://arxiv.org/abs/2201.05596

[48] "NVIDIA Data Center Platform For Every Workload." [Online]. Available: https://www.nvidia.com/en-us/data-center/

[49] "AMD Instinct™ Accelerators." [Online]. Available: https://www.amd.com/en/products/accelerators/instinct.html

[50] "ARM Processor - AWS Graviton Processor - AWS." [Online]. Available: https://aws.amazon.com/ec2/graviton/

[51] "Tensor Processing Units (TPUs) | Google Cloud." [Online]. Available: https://cloud.google.com/tpu?hl=en

[52] "Groq is Fast AI Inference," Dec. 2023. [Online]. Available: https://groq.com/

[53] Andrej, "karpathy/llama2.c," Nov. 2024, original-date: 2023-07-23T05:15:06Z. [Online]. Available: https://github.com/karpathy/llama2.c

[54] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," Jun. 2022, arXiv:2205.14135 [cs]. [Online]. Available: http://arxiv.org/abs/2205.14135

[55] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "vLLM, PagedAttention," Sep. 2023, arXiv:2309.06180 [cs]. [Online]. Available: http://arxiv.org/abs/2309.06180

[56] "Introducing Triton: Open-source GPU programming for neural networks | OpenAI." [Online]. Available: https://openai.com/index/triton/

[57] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models,” May 2020, arXiv:1910.02054 [cs]. [Online]. Available: http://arxiv.org/abs/1910.02054

[58] “Part 4.1: Tensor Parallelism — UvA DL Notebooks v1.2 documentation.” [Online]. Available: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/scaling/JAX/tensor_parallel_simple.html

[59] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, “GPT-NeoX-20B: An Open-Source Autoregressive Language Model,” Apr. 2022, arXiv:2204.06745 [cs]. [Online]. Available: http://arxiv.org/abs/2204.06745

[60] R. B. Prabhakar, H. Zhang, and D. Wentzlaff, “Kraken: Inherently Parallel Transformers For Efficient Multi-Device Inference,” Aug. 2024, arXiv:2408.07802 [cs]. [Online]. Available: http://arxiv.org/abs/2408.07802

[61] “Introduction to NVIDIA DGX H100/H200 Systems — NVIDIA DGX H100/H200 User Guide.” [Online]. Available: https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html

[62] “Improving GPU Memory Oversubscription Performance,” Oct. 2021. [Online]. Available: https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

[63] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking,” Apr. 2018, arXiv:1804.06826 [cs]. [Online]. Available: http://arxiv.org/abs/1804.06826

[64] Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, Z. Zhou, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, Y. Yan, B. Chen, G. Sun, and K. Keutzer, “LLM Inference Unveiled: Survey and Roofline Model Insights,” May 2024, arXiv:2402.16363 [cs]. [Online]. Available: http://arxiv.org/abs/2402.16363

[65] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” Mar. 2020, arXiv:1909.08053 [cs]. [Online]. Available: http://arxiv.org/abs/1909.08053

[66] “Optimum-NVIDIA Unlocking blazingly fast LLM inference in just 1 line of code.” [Online]. Available: https://huggingface.co/blog/optimum-nvidia

[67] “transformersbook/codeparrot · Datasets at Hugging Face.” [Online]. Available: https://huggingface.co/datasets/transformersbook/codeparrot

[68] “huggingface-course/code-search-net-tokenizer at main,” Oct. 2021. [Online]. Available: https://huggingface.co/huggingface-course/code-search-net-tokenizer/tree/main

[69] “Training a causal language model from scratch - Hugging Face NLP Course.” [Online]. Available: https://huggingface.co/learn/nlp-course/chapter7/6

[70] “HuggingFaceTB/smollm-corpus · Datasets at Hugging Face,” Aug. 2024. [Online]. Available: https://huggingface.co/datasets/HuggingFaceTB/smollm-corpus

[71] “Adastra’s architecture — Adastra user documentation.” [Online]. Available: https://dci.dci-gitlab.cines.fr/webextranet/architecture/index.html

[72] “SmolLM - blazingly fast and remarkably powerful.” [Online]. Available: https://huggingface.co/blog/smollm

[73] A. Hägele, E. Bakouch, A. Kosson, L. B. Allal, L. V. Werra, and M. Jaggi, “Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations,” Oct. 2024, arXiv:2405.18392 [cs]. [Online]. Available: http://arxiv.org/abs/2405.18392

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399