

MENG INDIVIDUAL PROJECT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

**A New Scalable Runtime for LLM
Inference**

Author:
Hamish McCreanor

Supervisor:
Peter Pietzuch

January 22, 2025

Submitted in partial fulfillment of the requirements for the MEng Computing of
Imperial College London

Contents

1	Introduction	2
2	Background	3
2.1	Preliminaries	3
2.1.1	LLM Architecture	3
2.1.2	LLM Inference	5
2.2	Related Work	9
2.2.1	vLLM	9
2.2.2	TensorRT-LLM	10
2.2.3	llama.cpp	10
3	Project Plan	11
4	Evaluation	12
4.1	Functional Requirements	12
4.2	Performance Metrics	12
5	Ethical Issues	13
6	Bibliography	15

Chapter 1

Introduction

As large language models (LLMs) are found useful for ever-wider classes of applications, a trend has arisen focusing on the low-cost, local deployment of these systems. While the training of LLMs like LLaMA, BERT and OpenAI's GPTs typically requires months of training and is prohibitive for all but the most well-funded of organisations, performing inference on these models locally is comparatively more feasible. This enables developers to create services with tighter LLM integrations - instead of calling a black-box API provided by an LLM provider, they can instead run a local version of the LLM, tuning the inference runtime to more appropriately match the context in which it is called.

As a result, there is currently a vast body of research aiming to improve existing inference systems. The aim of this is to improve LLM inference performance along various axes. These include running on lower-powered hardware; running with improved throughput and running at greater energy efficiencies. These optimisations focus on specific elements of the inference pipeline, particularly improving KV cache usage and kernel fusion. To build systems containing these optimisations, developers frequently turn to high level languages like Python in order to quickly develop the infrastructure surrounding their new technique. Developing this way limits the ability of the system to exploit memory-access patterns and application parallelism (especially in a language like Python, with its global interpreter lock) and incurs unnecessary overhead.

This project aims to build on the existing llama.cpp inference server (see 2.2.3) to deliver a system that improves the dispatch of compute kernels by better parallelising the inference pipeline.

Chapter 2

Background

We provide an overview of transformer architecture in 2.1.1, as well as key components of the inference pipeline in 2.1.2, before detailing existing work around LLM inference runtimes in 2.2.

2.1 Preliminaries

2.1.1 LLM Architecture

Transformer Architecture

While the term “Large Language Model” can be used to describe any model trained on large volumes of textual data, it is frequently used to refer to models that use a variant of the transformer architecture described in [1]. This architecture has superseded recurrent neural networks for language-based tasks owing to its ability to capture long range dependencies between tokens. It does this via a unique attention mechanism. This, and the other components present in a transformer, are described below.

- **Embedding Layers:** Attention blocks are fed a matrix where each row represents the semantic meaning of each token. This is done by using an embedding learnt during the training process and applying this to the input tokens.
- **Positional Encoding:** Unlike recurrent neural networks, transformer models contain no a priori knowledge of the order of the input sequence. To remedy this, positional encodings are added to the input embeddings before they are fed to the attention block. There are several ways of doing this, however the approach adopted in the [1] is to use apply sine and cosine functions to the position and dimension of the input vector as follows:

$$\begin{aligned} \text{PE}_{(pos, 2i)} &= \sin(pos/10000^{2i/d_{model}}) \\ \text{PE}_{(pos, 2i+1)} &= \cos(pos/10000^{2i/d_{model}}) \end{aligned}$$

- **Multi-Head Attention:** Transformers use an attention block to describe the relationship between different tokens in a sequence. This takes as input a key,

query and value matrix K , Q , V respectively. The key and query matrices are multiplied together to produce a representation of how each token in the sequence relates to each other token. After normalising and applying a softmax function, the matrix is multiplied by V , which represents the semantic meaning of each input token.

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

In practise, multiple attention heads are used and their outputs concatenated. Instead of attention blocks just operating on the K , Q , and V directly, they are multiplied by learned matrices W^K , W^Q , and W^V to project the input vectors into different spaces. The fact that there are multiple of these matrices W_i^Q , W_i^K and W_i^V allows the model to attend to information learnt from different projections concurrently.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

- **Position-wise Feed-Forward Networks:** The output of the multi-head attention block is fed to a fully connected neural-network. This takes as input the representation at each position (making it “position-wise”) and applies two linear transformations separated by a ReLU activation:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Multiple blocks consisting of the multi-head attention and feed-forward layers are stacked on top of one another to produce a deeper transformer model.

Model Variants

The encoder-decoder model introduced in [1] is not the only variant of the transformer architecture that exists. Modifications to the transformer architecture have been made in order to better support different tasks:

- **Encoder-decoder:** With encoder-decoder models, an input sequence is first embedded, then added to a positional encoding before being passed as input to a stack of attention and feed-forward network layers (the “encoder”) to produce a representation of the input sequence. As the output sequence is generated, it is fed as input to a similarly structured “decoder” block. This takes an embedded output sequence, with positional encoding, as input. The output of the encoder block is fed to attention blocks in the decoder block in order to model the relationship between the input and output sequence. These models [2] tend to be used for tasks like sequence to sequence transformation, for example language translation.

- **Encoder-only:** Encoder-only models use only the encoder block of an encoder-decoder model to produce a vector representation of an input sequence. These models do not produce text directly, but are instead meant to pass their output to a downstream component for further inference, potentially for applications to sentiment analysis or named-entity recognition. Examples of these include the BERT [3] family of models.
- **Decoder-only:** Decoder-only models like those in the GPT [4] series make up the bulk of models used for auto-regressive text generation and completion. Instead of sending an input prompt to an encoder block, the prompt is instead passed in in conjunction to the output sequence and the model is trained to predict the next token in the sequence [5].

2.1.2 LLM Inference

We begin with an overview of the steps taken to perform inference for a single request, before detailing other important aspects of LLM inference like managing the KV cache, distributing computation across multiple compute nodes and request batching.

Inference Pipeline

Performing inference based on a user prompt involves a prefill phase to initialise the state of the transformer model followed by an autoregressive phase to generate an output sequence. These steps are sandwiched by tokenisation and detokenisation stages respectively in order to convert between a string and vector representation of the input and output sequence.

- **Tokenisation:** In order to convert from a string representation of an input sequence, to a vector representation that a model can understand, tokenisation is applied. This splits the input string into a sequence of tokens present in a vocabulary learnt from the training corpus [6] [7], producing a vector of indices for each token in the input string. These indices are used to index into the learnt embeddings matrix and extract an embedding vector for each input token.
- **Prefill phase:** The prefill phase refers to the stage in the inference pipeline at which the hidden states (keys and values) of the network from the input sequence are produced. These values are later used to generate output tokens. Additionally, KV cache entries (see 2.1.2) for these tokens are produced and are later used to speed up the autoregressive phase of the inference pipeline. This stage represents an operation over a large matrix and, as such, can saturate the GPU, making this stage largely arithmetic bound.
- **Autoregressive phase:** In the autoregressive phase, output tokens are generated incrementally and appended back to the output prompt. The final layer of

the transformer produces a probability distribution over all output tokens (typically using a softmax activation), which is sampled from to produce an output token. Since we are only generating one token at a time, this phase is typically bound by the rate at which vectors can be passed to GPU memory. Further, the sequential dependencies between outputs limits the parallelisability of the autoregressive phase. The sequential nature of the autoregressive phase, and its corresponding performance characteristics, has significantly shaped the design of existing LLM runtimes (see 2.2).

- **Detokenisation:** At the end of the autoregressive phase, a vector containing indices into the model vocabulary is produced - this represents the output sequence of the model. In order to convert this vector to a string representation, the vector is sent to the CPU and each element is used to index into the model vocabulary, producing a string for each token. This list of token strings is then joined to form the overall output string.

KV Cache

The KV cache represents an important area with which inference efficiency can be improved. During the autoregressive phase of inference, self-attention refers to the key and value vectors KW_i^K and VW_i^V as well as the query vector of the latest output token. These key and value vectors have already been calculated at prior iterations and so caching them can reduce the computational overhead at each autoregressive step, as we can use these vectors without recalculation. The computed key and value vectors for the newest token are then added to the KV cache, to be reused during the next autoregressive iteration. This cache is initially populated during the prefill phase but then grows with the output sequence length. The effective management of the KV cache can be used to help overcome the limited parallelisability of the autoregressive phase of inference [8] [9].

Hardware

The recent success of LLMs is in large part enabled by the use of Graphics Processing Units (GPUs) [10]. GPUs are optimised for performing a large number of specialised operations in parallel, in contrast with Central Processing Units (CPUs), which are better suited to performing a more general set of operations sequentially. Modern deep neural networks can be distilled down into a sequence of vector and matrix operations. These tasks are highly parallelisable and specialised implementations like of linear algebra operations like cuBLAS [11] exist to accelerate them on GPU hardware. Other software layers like CUDA [12] also exist and can be used to implement efficient versions of common deep learning functions. As a result of this, GPUs are well suited for the kind of workloads required by LLMs.

At a high level, GPUs consist of a series of Streaming Multiprocessors (SMs), each containing multiple Stream Processors (SPs, also referred to as ‘cores’) and some shared memory[13]. Each core is designed to support a small set of relatively simple operations, like floating-point and integer arithmetic, as well as logical operations.

High-bandwidth memory (HBM) is used to transfer data between SMs, while the shared memory on each SM can be used to share common data between threads. Tensors like weights and the KV cache are typically stored in HBM and then moved into the SMs at the time of computation [9].

In addition to GPUs, other hardware platforms like ASICs, FPGAs and TPUs exist and can be used to further accelerate the operations of LLMs [14]. These platforms tend to be more heterogenous in architecture and, as such, most existing LLM runtimes (see 2.2) tend to be optimised for a GPU-like architecture.

Distributed Computation

One approach to scaling LLMs is to distribute computation amongst multiple GPUs. Model data can be shared across multiple nodes in a network to increase inference throughput, while model layers and even individual tensors can be sharded to enable the deployment of larger models. These techniques need not be used in isolation and are frequently used in combination in existing training and inference pipelines [15] [16] [17].

- **Data parallelism:** Data parallelism involves replicating the weights for an entire model across multiple nodes. Each node is now capable of handling inference requests independently, increasing the number of requests that can be serviced in parallel. This approach reduces request latency and is simple to implement and scale, but has high memory requirements, as each node must be capable of storing the weights for the entire model as well as any intermediate tensors produced during inference. This is potentially prohibitive for larger models featuring high numbers of weights [18] [19] [20] .
- **Pipeline parallelism:** Pipeline parallelism is a way to distribute computation that enables training and inference for models with a large number of weights. This approach borrows from the classic computer architecture technique of pipelining [21]. Nodes are assigned the weight tensors for individual layers of the network and are responsible for executing that layer alone. Data is then passed between nodes in sequence [22]. This method incurs a small computational overhead in moving data between nodes in the pipeline, however this is relatively minor compared to the scheduling delay incurred in other methods like tensor parallelism. The main drawback of this method is the potential for bubbles to form in the pipeline. This occurs due to nodes taking different times to compute their layer. Nodes that complete early might have to wait idle while nodes upstream of them in the pipeline execute, reducing node utilisation.
- **Tensor parallelism:** Tensor parallelism extends pipeline parallelism to allow the deployment of models whose layers or intermediate tensors may be too large to fit on just one node. Layer specific tensors, like weights or intermediate activations can be sharded across multiple nodes. Nodes then work together to complete a subset of the overall tensor computation before combining their results later. While this approach enables even larger models to be executed, it further increases the computational overhead of sharing data across nodes -

we now need to synchronise and move data on an intra-layer basis, instead of on an inter-layer basis as in pipeline parallelism [16].

Request Batching

Another method used to address the limited parallelism created by the autoregressive phase of inference is to batch requests. With request batching, we can stack tensors from multiple user requests into the same batch and issue this request to the GPU. This enables user requests to be served in parallel and increases GPU utilisation at the same time as maximising use of available memory bandwidth.

- **Naive batching:** The naive approach to batching consists of grouping multiple input prompts into a batch and running inference on it - inference is scheduled at the granularity of requests [17]. In order for the input and output batches to be valid matrix shapes, they must be padded with special padding tokens to a fixed size. While simple to implement, this approach can lead to increased latency as well as underutilisation of the GPU. This is because prompts within a batch do not necessarily generate output sequences of the same length. If one output sequence has terminated while there are other sequences for which tokens are still being generated it will be unable to be returned to the user until all other sequences terminate. As such the request latency is bounded by the time taken to generate the longest sequence in the batch [17]. This also limits the system from serving new requests as they arrive: the new request will need to wait for completion of the entire batch of requests, even if the batch contains requests for which an output has already been generated. As a result, naive batching is limited in its use, only achieving maximal GPU utilisation under very specific usage patterns.
- **Continuous batching:** Continuous batching is proposed as an improvement to naive batching and schedules inference at the granularity of autoregressive iterations [17]. At the end of each autoregressive iteration, a set of new tokens is produced for every request in the batch. A scheduler then monitors the batch to determine the completion status of every request in the batch, dynamically scheduling waiting requests when it detects that a complete output sequence has been generated. This achieves greater utilisation of the GPU since it is no longer executing operations for sequences that have already terminated. It also reduces latency for user requests, as output sequences can be returned as soon as they are terminated. However, this batching technique comes at the cost of increased scheduling overhead as well as limited batching of certain matrix operations. This is because some operations, like computing attention, require their input sequences to be aligned to the same length and position in order to be batched. Because requests are dynamically dispatched, this property no longer holds and thus operations like computing attention blocks are unable to be batched, in a technique known as “selective batching” [17].

2.2 Related Work

Various runtimes for performing LLM inference already exist. These implement some of the techniques described earlier in order to provide efficient inference for a variety of use cases. In order to understand where it might be possible to offer a novel contribution with regard to performance, we first need to understand the features offered by existing solutions. Here we present a selection of popular runtimes and detail their main contributions and points of difference.

2.2.1 vLLM

vLLM is a popular LLM runtime designed for high throughput serving of inference requests [15]. It addresses key challenges in memory management that arise during LLM serving, particularly those related to KV cache memory. Existing systems often suffer from memory inefficiencies due to fragmentation and over-allocation, limiting their ability to process large batch sizes effectively. vLLM solves this problem through a novel memory management mechanism, achieving near-zero KV cache waste [15].

Key Contributions

The main contribution of vLLM is its PagedAttention mechanism. On receiving a new request, many inference runtimes pre-allocate a contiguous chunk of memory for the KV cache equal in size to the request's maximum output. This can cause internal fragmentation, if the actual output is less than the size of memory allocated, as well as external fragmentation, when smaller requests are unable to be scheduled due to overallocation. Moreover, for requests that generate multiple outputs, as is the case with parallel and beam search, the KV cache is stored in a different location for each output and so there is no possibility of memory reuse.

To address this, PagedAttention divides the request's KV cache into smaller units called blocks. These contain a fixed number of key and value vectors associated with a set of tokens. Blocks for a given request's KV cache need not be located in physically contiguous locations and are allocated dynamically as the KV cache for a request grows. This eliminates external fragmentation as each block is the same size. The blocks themselves are chosen to be small in size in order to limit internal fragmentation [15]. This is analagous to the paging technique common to modern operating systems [23], with blocks corresponding to pages, tokens to bytes and requests to processes [15]. By optimising memory allocation, vLLM allows larger batch sizes, significantly improving throughput.

The rest of the vLLM system is built to support this use of PagedAttention and integrates several other common optimisation techniques, including continuous batching, as well as pipeline and tensor parallelism. It supports a variety of popular LLMs, including those in the GPT, OPT and LLaMA series [4] [24] [25]. The PagedAttention optimisation also integrates well with popular decoding techniques like parallel and beam search, facilitating memory sharing across request outputs [15].

Architecture

vLLM exposes a Python library and an OpenAI API [26] compatible web server for inference. The frontend is written using the FastAPI library [27]. Python is used to implement the bulk of the project, with C++ and CUDA code being used to implement the custom CUDA kernels required to support the PagedAttention mechanism. Importantly, control-specific components like the batch scheduler are implemented in Python, representing a potential source of overhead.

vLLM uses a hierarchical class structure to distribute requests amongst GPU workers. A single executor is maintained through the lifespan of the runtime. This is responsible for creating and running worker processes. These are processes dedicated to running model inference on specific GPUs and are tasked with preparing tensors and orchestrating the execution of the model on that node. Importantly, this means that a separate Python process needs to be created for every single worker node, incurring a performance penalty.

On individual GPU workers, a block engine is used to allocate a contiguous chunk of GPU RAM and divide it into physical KV cache blocks. A block manager is created to maintain the mapping between logical and physical KV blocks and this is also implemented in Python. NCCL [28] is used for communication across distributed GPU workers.

2.2.2 TensorRT-LLM

NVIDIA's TensorRT-LLM extends the TensorRT deep learning compiler to provide an inference runtime for LLMs [29]. The TensorRT compiler is a high-performance deep learning inference optimizer designed to transform trained neural network models into highly efficient computational graphs suitable for deployment on NVIDIA hardware. It achieves this by applying a range of optimizations as well as making use of NVIDIA-specific hardware features [29]. This integration with TensorRT, and its associated performance on NVIDIA products is the principal advantage of this runtime.

TensorRT-LLM provides a Python library similar to that of vLLM through which inference is performed. Similarly to vLLM, TensorRT-LLM also implements pipeline and tensor parallelism as well as continuous batching [29].

2.2.3 llama.cpp

Chapter 3

Project Plan

- Identifying inefficiencies:
- Project report:

Chapter 4

Evaluation

Use ShareGPT [cite] and Alpaca [cite] datasets

4.1 Functional Requirements

4.2 Performance Metrics

- Throughput (tokens/s):
- Time to First Token (s):
- Time Per Output Token (ms):

Chapter 5

Ethical Issues

The principal two ethical concerns of a project in this field relate to the potential for misuse as well as provenance issues surrounding the dataset on which the model was trained.

The potential for misuse of LLMs is vast, with many instances of LLM abuse already being documented. LLM abuse typically involves the use of the model to produce harmful or misleading content. There already exist proof-of-concepts for LLMs [30] being used to generate phishing messages, with the intent to produce emails that sound more plausible and are more likely to be engaged with by a target. In addition to this, LLMs can be used to produce vast quantities misinformation or biased content that are then published to social media platforms [31]. End users may be unable to distinguish between content created by a genuine user and content generated by an LLM and thus end up misinformed.

The large size of the datasets required to train these models create potential ethical and data protection issues. Concerns exist regarding the ability for generative models to amplify existing biases in their training data, with some of these concerns borne out in cases like Microsoft's Tay chatbot [32]. AI fairness is still an open area of research [33] and it is unlikely that existing LLM models will be completely free of bias at inference time. At the same time, the provenance of this training data is also an important ethical consideration. Private or sensitive data has the potential to be incorporated into training sets and there exist cases [34] where this training data has then been generated verbatim at inference time, exposing this sensitive data to an end user.

If successful, our project broadens access to LLMs by making better use of available hardware to perform inference. This increases the viability of local inference and opens up these models to a greater proportion of hardware configurations and thus a greater number of users. While this represents a boon for the accessibility of this technology, with users no longer limited to a handful of offerings by large companies, it also increases the number of potentially malicious actors who are able to use LLMs. Small and local deployments likely have less of the oversight that large LLM providers experience, and thus are more able to misuse this technology. These two elements must be carefully managed in order to produce a project that adheres to reasonable ethical standards.

As this project represents a proof-of-concept, rather than a full-featured inference

engine, any advances made are unlikely to immediately be adopted and thus any ethical concerns are likely to be uncovered at a pace with which they can be identified early on and mitigated quickly.

Chapter 6

Bibliography

- [1] Vaswani A. Attention is all you need. *Advances in Neural Information Processing Systems*. 2017. pages 3, 4
- [2] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*. 2020;21(140):1-67. pages 4
- [3] Kenton JDMWC, Toutanova LK. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of naacL-HLT*. vol. 1. Minneapolis, Minnesota; 2019. p. 2. pages 5
- [4] Radford A. Improving language understanding by generative pre-training. 2018. pages 5, 9
- [5] Dai AM, Le QV. Semi-supervised sequence learning. *Advances in neural information processing systems*. 2015;28. pages 5
- [6] Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I, et al. Language models are unsupervised multitask learners. *OpenAI blog*. 2019;1(8):9. pages 5
- [7] Sennrich R. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:150807909*. 2015. pages 5
- [8] Shi L, Zhang H, Yao Y, Li Z, Zhao H. Keep the Cost Down: A Review on Methods to Optimize LLM's KV-Cache Consumption. *arXiv preprint arXiv:240718003*. 2024. pages 6
- [9] Pope R, Douglas S, Chowdhery A, Devlin J, Bradbury J, Heek J, et al. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*. 2023;5:606-24. pages 6, 7
- [10] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*. 2012;25. pages 6

- [11] Nvidia cuBLAS;. Accessed: 2025-01-21. <https://developer.nvidia.com/cublas>. pages 6
- [12] Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*. 2008;6(2):40-53. pages 6
- [13] Choquette J, Gandhi W, Giroux O, Stam N, Krashinsky R. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*. 2021;41(2):29-35. pages 6
- [14] Li J, Xu J, Huang S, Chen Y, Li W, Liu J, et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:241004466*. 2024. pages 7
- [15] Kwon W, Li Z, Zhuang S, Sheng Y, Zheng L, Yu CH, et al. Efficient memory management for large language model serving with pagedattention. In: *Proceedings of the 29th Symposium on Operating Systems Principles*; 2023. p. 611-26. pages 7, 9
- [16] Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:190908053*. 2019. pages 7, 8
- [17] Yu GI, Jeong JS, Kim GW, Kim S, Chun BG. Orca: A distributed serving system for {Transformer-Based} generative models. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*; 2022. p. 521-38. pages 7, 8
- [18] Rae JW, Borgeaud S, Cai T, Millican K, Hoffmann J, Song F, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:211211446*. 2021. pages 7
- [19] Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, et al. Language models are few-shot learners. *Advances in neural information processing systems*. 2020;33:1877-901. pages 7
- [20] Chowdhery A, Narang S, Devlin J, Bosma M, Mishra G, Roberts A, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*. 2023;24(240):1-113. pages 7
- [21] Hennessy JL, Patterson DA. *Computer architecture: a quantitative approach*. Elsevier; 2011. pages 7
- [22] Huang Y, Cheng Y, Bapna A, Firat O, Chen D, Chen M, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*. 2019;32. pages 7
- [23] Kilburn T, Edwards DB, Lanigan MJ, Sumner FH. One-level storage system. *IRE Transactions on Electronic Computers*. 1962;(2):223-35. pages 9

- [24] Zhang S, Roller S, Goyal N, Artetxe M, Chen M, Chen S, et al. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:220501068. 2022. pages 9
- [25] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:230213971. 2023. pages 9
- [26] OpenAI API;. Accessed: 2025-01-22. <https://openai.com/blog/openai-api>. pages 10
- [27] FastAPI;. Accessed: 2025-01-22. <https://github.com/fastapi/fastapi>. pages 10
- [28] Nvidia NCCL;. Accessed: 2025-01-22. <https://developer.nvidia.com/nccl>. pages 10
- [29] Deploying AI Models with Speed, Efficiency, and Versatility - Inference on NVIDIA's AI Platform;. Accessed: 2025-01-22. <https://www.nvidia.com/en-us/lp/ai/inference-whitepaper/>. pages 10
- [30] Hazell J. Spear phishing with large language models. arXiv preprint arXiv:230506972. 2023. pages 13
- [31] Williams AR, Burke-Moore L, Chan RSY, Enock FE, Nanni F, Sippy T, et al. Large language models can consistently generate high-quality content for election disinformation operations. arXiv preprint arXiv:240806731. 2024. pages 13
- [32] Lee P. Learning from Tay's introduction; 2016. Accessed: 2025-01-14. <https://web.archive.org/web/20241127051442/https://blogs.microsoft.com/blog/2016/03/25/learning-tays-introduction/>. pages 13
- [33] Xivuri K, Twinomurinzi H. A systematic review of fairness in artificial intelligence algorithms. In: Responsible AI and Analytics for an Ethical and Inclusive Digitized Society: 20th IFIP WG 6.11 Conference on e-Business, e-Services and e-Society, I3E 2021, Galway, Ireland, September 1–3, 2021, Proceedings 20. Springer; 2021. p. 271-84. pages 13
- [34] Nasr M, Carlini N, Hayase J, Jagielski M, Cooper AF, Ippolito D, et al. Scalable extraction of training data from (production) language models. arXiv preprint arXiv:231117035. 2023. pages 13