A New Scalable Runtime for LLM Inference

Author:
Hamish McCreanor

Supervisor:
Peter Pietzuch

June 9, 2025

Abstract

Test-time compute scaling has emerged as an alternative to expensive training for improving large language model (LLM) performance. Methods in this paradigm generate chains of reasoning in response to a user query, often sampling multiple outputs to explore a space of possible answers. This paper focuses on beam search, a test-time compute scaling approach that uses one model (the LLM) to generate reasoning steps and another (the process-reward model, or PRM) to score and guide a tree search toward a final response.

Such collaborative systems are relatively underexplored, and significant opportunities remain to optimise their performance. We examine the memory characteristics of a beam search system implemented using the popular vLLM inference library, analysing the memory usage and batching behaviour of both models, and how system performance varies with different memory allocations between them. Additionally, we evaluate vLLM's KV cache management strategy and find it well-suited to tree-search-style workloads.

Building on these insights, we propose two optimisations: (1) a decode-aware speculative tree expansion strategy that exploits differing batching preferences between the LLM and PRM, and (2) a grid search–based approach to optimise performance in a pipelined beam search system. We implement and evaluate the first of these, demonstrating a measurable improvement in throughput over a baseline.

Contents

Chapter 1

Introduction

As large language models (LLMs) are adopted for an ever-growing range of applications, scaling time spent in training has emerged as the dominant paradigm for improving model performance. However, with training costs growing prohibitively expensive, there is increasing interest in other techniques for bettering model output. Test-time compute scaling offers one such way to improve performance, with existing models achieving better results by "thinking" for longer. This is done by encouraging LLMs to output longer chains of reasoning, shifting scaling costs from training time to test time and enabling existing models to improve without incurring any extra training expense.

The most popular test-time compute scaling approaches leverage a second model to guide a tree-search towards the best answer. A generator LLM proposes steps to an answer to a user query and a verifier LLM scores these steps, with this process repeating until a set criteria is met. This is a departure from an assumption made by existing LLM inference systems, which is that only one model will run at a time. As such, they are likely to have made architectural decisions that limit the throughput and latency of a test-time scaling approach involving the interplay between multiple LLMs.

This represents an opportunity to build a new system that more effectively supports the test-time scaling use case. Key to these efforts is optimizing the memory of such a system, particularly the management of the key-value (KV) cache, which stores intermediate results reused during autoregressive token generation. Efficient use of the KV cache can significantly improve memory utilisation and computational speed.

This project analyses a beam search implementation built upon vLLM, a popular and widely used LLM inference library, to try and identify deficiencies that limit the effective use of GPU memory. Several experiments are conducted to better understand how the implementation's performance changes as a function of various parameters and optimisations to improve GPU memory utilisation are proposed and validated.

Chapter 2

Background

We provide an overview of transformer architecture in 2.1, as well as key components of the inference pipeline in 2.2, before detailing the idea of test-time compute scaling and describing relevant techniques in 2.3.

2.1 LLM Architecture

2.1.1 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:

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

- **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.

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

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.

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

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

- **Position-wise Fead-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 conjuction to the output sequence and the model is trained to predict the next token in the sequence [5].

2.2 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.

2.2.1   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 2.2.2) 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 (also referred to as the decode 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. A temperature parameter can be introduced to modify this probability distribution and control the level of randomness present in the sampling stage. 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 3).

- **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.

2.2.2   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].

2.2.3 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 3) tend to be optimised for a GPU-like architecture.

2.2.4 Distributed Computation

One approach to scaling LLMs to 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].

2.2.5   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 bandwith.

- **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 unabled to be batched, in a technique known as "selective batching" [17].

2.3 Test Time Compute Scaling

Historically, approaches to improving LLM performance have centered on training. With larger datasets, larger models and more training epochs, performance on the cross-entropy loss function used for training these models can be made to improve considerably [23]. However, scaling in this way eventually becomes prohibitively expensive due to the high cost of owning and operating the hardware required for training.

This motivates the idea of test-time compute scaling. Test-time compute scaling trades off inference latency for higher quality responses to user prompts. By allowing existing LLMs to take longer to respond to user prompts, the quality of responses can be improved considerably [24], obviating the need to retrain a bigger model on a more expansive dataset. Test-time compute scaling describes a broad approach to trading off inference latency for higher quality model outputs - specific methodologies for how to implement this are described below.

2.3.1 Chain of Thought Prompting

Chain of thought prompting uses a series of reasoning steps to improve LLM answer quality. By prompting a model with examples of answers containing structured reasoning steps and encouraging it to follow a similar structure in its own responses, model performance has been empirically shown to improve considerably [25].

2.3.2 Self Consistency Sampling

Self-consistency sampling is a simple technique to improve model outputs that replaces the decoding strategy present in chain of thought prompting. Instead of greedily sampling each token, self-consistency voting generates a diverse set of reasoning paths by sampling multiple tokens in parallel. These tokens produce a tree structure, with different branches corresponding to different token choices. When all reasoning paths conclude with an answer to the user's question, the answers from all paths are aggregated out by majority vote to produce a final response [26].

2.3.3 Best-of-N Sampling

Best-of-N sampling extends the idea of self-consistency to include some sort of external verifier. This verifier can be any machine learning model but is typically some sort of transformer model. In the basic best-of-n setup, this verifier scores all the n solutions sampled from the LLM and the one with the highest score is output as the final response. With weighted best-of-n, all the final solutions are aggregated such that reasoning paths that generate the same solution are weighted higher when combined with the verifier score [27].

2.3.4 Beam Search

Beam search provides a way of systematically exploring the tree structure created when generating multiple chains of reasoning in parallel. Multiple chains of reasoning are generated

iteratively, with scores being output for each step. These scores are then used to guide the search towards higher quality answers - reasoning steps with higher scores are expanded through further sampling and reasoning steps with lower scores are pruned [28] [29].

We maintain a fixed number N of active paths and expand each of these paths with M samples. All of these $N \times M$ new reasoning steps are scored and the top N steps are kept for the next iteration.

Implementations of beam search frequently use a Process Reward Model (PRM) to score the intermediate reasoning steps. PRMs differ from the verifier models used in best-of-n search in that they are trained to output a score for every reasoning step, rather than just the final result.

Chapter 3

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.

3.1 Inference Runtimes

3.1.1 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 called PagedAttention, achieving near-zero KV cache waste [15]. 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] [30] [31].

PagedAttention

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 [32], with blocks corresponding to pages, tokens to bytes and requests to processes [15].

By passing only the necessary KV cache blocks to the GPU kernel for each request, vLLM avoids overallocation and enables tighter packing of data from multiple requests into a single kernel invocation. This effectively increases the batch size and leads to significantly improved throughput.

Automatic Prefix Caching

The KV cache blocks used by PagedAttention provide a natural mechanism through which prefix caching can be implemented. Prefix caching is a technique common to many inference runtimes to avoid recomputing KV cache values across shared input prompts. The KV cache blocks of processed requests are cached and reused when a new request comes in with a shared prefix.

To handle this automatic prefix caching, vLLM introduces a KV cache manager. It maintains a doubly linked list of free KV cache blocks, as well as a list of currently cached KV blocks in use. When a new request arrives, vLLM computes hash values for the token sequences in the request - these hashes are derived not only from the current tokens but also from the sequence of prefix tokens that precedes them. If a matching hash is found, indicating that the KV cache blocks for those tokens have already been computed, they can be directly reused by the new request. In the event of a prefix cache miss, vLLM allocates new KV cache blocks by popping from the list of free blocks and populating them with the computed KV values for the unmatched prefix. If no free KV cache blocks are available, vLLM evicts blocks from the cache. Eviction is done based on a least-recently-used (LRU) policy, with a preference for evicting the blocks associated with the longest prefixes first, as these are less likely to be shared by future requests.

Architecture

vLLM exposes a Python library and an OpenAI API [33] compatible web server for inference. The frontend is written using the FastAPI library [34]. 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 heirarchical 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 [35] is used for communication across distributed GPU workers.

3.1.2 SGLang

SGLang is an inference runtime designed to efficiently execute LLM-centric programs that involve multiple, often dependent, generation calls interleaved with control flow and structured inputs/outputs [36]. Unlike traditional inference engines focused solely on throughput or latency, SGLang provides a frontend programming interface and a backend runtime to optimise execution for multi-call, structured LLM workflows.

At the frontend, SGLang provides a domain-specific language embedded in Python. It exposes that allow users to express parallelism, control flow, and constrained generation. This enables modular, composable, and readable construction of LLM workflows.

The backend runtime, known as SGLang Runtime (SRT), introduces three main contributions:

- **RadixAttention:** A mechanism for automatic KV cache reuse across multiple generation calls, using a radix tree to manage shared prompt prefixes. This dramatically reduces redundant computation and memory overhead by reusing previously computed intermediate states, unlike other existing runtimes that discard cache between calls.

- **Compressed Finite State Machine (FSM):** For constrained decoding tasks, SGLang compiles regex-defined output formats into compressed FSMs. This allows multi-token decoding in a single step, bypassing the typical one-token-at-a-time bottleneck seen in standard FSM decoding engines.

- **API Speculative Execution:** For black-box API-only models like GPT-4, SGLang speculatively continues generation across calls, caching potential continuations to reduce API costs and latency. This strategy effectively amortises input token cost across multiple calls.

3.1.3 TensorRT-LLM

NVIDIA's TensorRT-LLM extends the TensorRT deep learning compiler to provide an inference runtime for LLMs [37]. 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 [37]. 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 [37].

3.1.4 llama.cpp

llama.cpp is a popular open-source runtime for LLM inference on CPUs, particularly designed to run models from the LLaMA family [31]. It focuses on memory-efficient execution through quantization and the use of the ggml tensor library, which is tailored for CPU-based matrix operations [38].

The runtime is designed to operate entirely on CPUs, using ggml to build a computation graph and perform optimized tensor computations with minimal memory overhead. It employs mmap to load models directly into memory, reducing the initial memory load and

improving scalability on devices with limited RAM. As stated earlier, llama.cpp supports the LLaMA family of models while also providing conversion tools to enable compatibility with other model formats, such as PyTorch checkpoints [39].

A key distinction between llama.cpp and other, more fully-featured runtimes like vLLM lies in their decoding strategies. llama.cpp employs a token-by-token decoding approach, generating one token at a time sequentially for each request. This minimizes memory usage and simplifies execution, making it suitable for resource-constrained environments but limiting overall throughput. In contrast, other runtimes utilise batch-level decoding, processing multiple requests and generating tokens in parallel. This approach leverages GPU parallelism to achieve significantly higher throughput, particularly for large-scale and concurrent workloads. llama.cpp does not natively support distributed inference or multi-node serving but can be integrated with external orchestration tools [40] for these purposes.

3.2 Optimisations

Here we detail optimisations that have been implemented on top of existing inference runtimes in order to better improve LLM inference performance. The optimisations here are relevant to our problem setting of a multi-model runtime,

3.2.1 Sarathi-Serve

Sarathi-Serve addresses the throughput-latency tradeoff in LLM inference by co-optimising batching strategies for the prefill and decode phases of requests [41]. Its two key contributions are:

- **Chunked prefill kernel:** Instead of computing an entire prefill request in a single iteration, Sarathi-Serve splits it into compute-sized chunks. These chunks are co-scheduled with ongoing decodes in a way that maximises GPU compute saturation while avoiding generation stalls. This is effective because prefill is compute-bound and saturates GPU cores, while decode is memory-bound and benefits more from large batch sizes. Chunking also constrains the iteration time, helping to reduce inter-token latency.

- **Stall-free scheduler:** This scheduler constructs hybrid batches by coalescing decode tokens with a chunked prefill request. It uses a token budget to ensure iteration latency remains bounded, and includes ongoing decodes before admitting new prefill chunks. This prevents long prefills from stalling in-progress decode requests, an issue seen in prefill-prioritising schedulers like vLLM, and leads to higher throughput and lower tail latency.

3.2.2 POD-Attention

POD-Attention improves GPU utilisation in hybrid batches by fusing the attention kernels used for prefill and decode operations [42]. In earlier hybrid batching systems, linear layers were fused across prefill and decode tokens, but attention layers were still computed separately, using distinct kernels optimised for each phase.

This separation is suboptimal for long-sequence workloads, where attention dominates compute time. Prefill attention is compute-bound, while decode attention is memory-bound. Computing them independently in a hybrid batch leads to resource underutilisation.

POD-Attention introduces a fused GPU kernel that computes both prefill and decode attention within the same kernel execution. This kernel assigns prefill and decode CTAs to the same streaming multiprocessor using SM-aware scheduling, enabling simultaneous usage of compute and memory bandwidth. As a result, POD-Attention achieves faster attention computation and improves end-to-end throughput and latency in LLM serving systems that use hybrid batching.

3.2.3 Prism

Prism considers the challenges of multi-LLM inference in environments where a single vendor serves many different models [42]. Its design is motivated by two empirical observations. Firstly, model popularity follows a long-tail distribution with only a select few models receiving the majority of requests and secondly, many models remain idle for long periods of time. Static memory partitioning across models fails to exploit these workload characteristics, resulting in poor GPU utilisation.
To address this, Prism proposes two main mechanisms:

- **KV cache virtual memory abstraction:** Prism allows models to dynamically share GPU memory by decoupling virtual and physical memory using NVIDIA's virtual memory APIs. Memory is allocated on-demand, enabling efficient re-use across models without pre-allocating static slices. This abstraction lies below the KV cache management mechanisms found in systems like vLLM, enabling cross-model memory coordination.

- **Two-level scheduler:** Prism uses a global scheduler to place models across GPUs based on their memory demands, and a local scheduler that prioritises incoming requests by their latency service level objectives (SLOs). This two-level design enables flexible sharing of GPU memory and compute across models with varying usage patterns and requirements, leading to improved SLO attainment and cost efficiency.

Chapter 4

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. LLMs have already been used to generate harmful or misleading content, such as phishing messages [43] and large-scale misinformation on social media [44]. 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 [45]. AI fairness is still an open area of research [46] 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 [47] where this training data has then been generated verbatim at inference time, exposing this sensitive data to an end user.

This project aims to investigate potential bottlenecks and limitations of existing inference runtimes as they pertain to scaling test-time compute. Eliminating some of these bottlenecks makes test-time compute scaling more feasible and enables users to improve the answer quality of off-the-shelf models without requiring an expensive training process. 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 present greater risk. These two elements must be carefully managed in order to adhere to reasonable ethical standards.

As a proof-of-concept rather than a production tool, this project's impact is likely to be limited and gradual, allowing ethical concerns to be addressed as they emerge.

Chapter 5

Problem Setting and System Description

Here we describe in greater detail the system we use as a baseline for an implementation of test-time compute scaling. The original system we used came from an open-source project called 'Search and Learn' [48]. This project implements several strategies for scaling test-time compute including beam search and best-of-n sampling. The 'Search and Learn' project was primarily created to provide an open-source reference implementation of popular test-time compute scaling techniques that had been detailed in other studies [24]. As such, it is unconcerned with performance and lacks several easily-made modifications that would improve performance considerably. These modifications are detailed below. We also list key parameter choices and settings before explaining and justifying our main evaluation metric.

5.1 Search and Learn

The 'Search and Learn' project implements several popular test-time compute scaling techniques. These include best-of-n sampling, beam search and a novel Diverse Verifier Tree Search (DVTS) algorithm [48]. The aim of the project is to empirically demonstrate that different scaling techniques can be used depending on the problem difficulty and compute budget. This is orthogonal to the goal of this project, which is to investigate bottlenecks in test-time scaling implementations, and so any optimisation we make or propose must not alter the output content of the system. Instead, we use 'Search and Learn' as a reference point from which to begin our analysis and benchmarking. We also use it as a starting point for our optimisation implementations. For the purposes of our analysis, we only concerned ourselves with the beam search implementation, as this was shown to be the best search strategy on average [48].

5.2 Problem Setting

5.2.1 Models

The beam search implementation uses two transformer-based models. The first, refered to as the 'LLM', generates new reasoning steps in response to a user prompt. For our system, we use Meta's `Llama-3.2-1B-Instruct` model [49]. The second, refered to as the 'PRM', produces a score for each reasoning step in the current LLM output. This is provided via the `Llama3.1-8B-PRM-Deepseek-Data` model [50] [51] [52]. The LLM generates new reasoning

steps as the beam search tree is explored. It requires a prefill stage to compute the attention states for all previous reasoning steps, but then uses the autoregressive decode stage to produce new chains of reasoning. As decode is limited by memory bandwidth rather than GPU FLOPs, we say this model is decode-bound. The PRM scores a reasoning step by taking as input all of the LLM generated reasoning up to and including that step. It emits a single output token and then the logits in the final layer for specific tokens are extracted and used as a score. This is repeated for all reasoning steps. The fact that the PRM only outputs a single token makes it heavily bound by the prefill stage of the inference pipeline.

The unique distinction between the LLM being decode-bound and the PRM being prefill-bound opens up opportunities for optimisation.

5.2.2 Dataset

The original 'Search and Learn' project uses a subset of 500 problems selected from the MATH dataset, a selection of problems taken from high-school maths competitions [53]. These problems are assigned difficulty levels from one to five. For our benchmarking, we use a set of 128 problems sampled uniformly across these difficulty levels.

5.2.3 Beam Search Parameters

To ensure our accuracy on the MATH dataset remains comparable with the original 'Search and Learn' project for the purpose of evaluation, we keep our beam search parameters the same. Namely we use a beam width of four and set a maximum tree depth of 40.

5.3 Modifications

To ensure a reasonable baseline from which to evaluate performance, several changes were made to the 'Search and Learn' reference code. These represent simple optimisations to correct obvious performance deficiencies in the original codebase and were implemented by several members of Imperial College London's Large-Scale Data and Systems Group.

5.3.1 Consistent Runtime

The original 'Search and Learn' code uses two different runtimes to run the LLM and PRM, vLLM and Hugging Face's `transformers` library [15] [54] respectively. The decision was made to use the same runtime for the two models in order to provide a consistent runtime as a starting point for analysis. vLLM is a library specifically optimized for inference performance, whereas the inference functionality in `transformers` is more of an add-on, primarily intended to demonstrate the range of models it supports. As such, vLLM was chosen over `transformers`. One significant impact of this is that it means the PRM benefits from vLLM's built in prefix caching, which is not a feature of the `transformers` library.

5.3.2 Score Cache

For each node in the beam search tree, the PRM needs to run to score the associated reasoning step. This represents repeated computation, as the PRM score for a given reasoning step will remain the same regardless of how deeply the search tree is explored. A score cache was implemented that holds the associated scores for previous PRM invocations.

5.3.3   Continuous Problem Batching

One major improvement in throughput comes from the implementation of continuous problem batching. The 'Search and Learn' implementation groups the MATH dataset questions into batches and feeds these into the beam search implementation, only running beam search on one batch at a time. Owing to the varying difficulty of problems in this dataset, problems terminate their beam search at different times, meaning that the size of the input batches to both the LLM and PRM decrease as the search progresses, leading to underutilisation of the GPU. To remedy this, a 'minimum problem batch size' is introduced. If the size of the input batch to the LLM or PRM is less than this minimum, problems from other batches are fed in to ensure that the GPU is fully utilised - this dramatically improves system throughput.

5.3.4   End of Sequence Sampling

The 'Search and Learn' implementation imposes a maximum tree depth for the beam search. When this depth is reached, instead of terminating the response, the implementation instead continues to sample from the beam until an end of sequence token is produced - we remove this feature in our implementation. Empirically, sampling in this manner does little to improve the accuracy of the system on the MATH dataset. If a problem has reached the maximum tree depth without terminating, the current chain of reasoning has likely diverged significantly from the correct answer and extra sampling is unlikely to improve the quality of the response.

End of sequence sampling dramatically changes the execution profile of the system, with a long execution period for both the LLM and PRM following the short, regular switching between models in earlier steps. This more closely resembles the execution profile of a naive, non-test time scaling implementation. As such, when we hit our maximum tree depth, we instead just output the response generated by the beam up to that point.

5.4   Evaluation Metrics

We choose to focus our evaluation on the throughput of the test-time scaling system, rather than on latency. This is motivated by the nature of the prompts that users typically submit to such systems - these are typically difficult problems involving long chains of reasoning. As a result, users typically anticipate slower response times. In this setting, overall throughput, measured in tokens or prompts processed per unit time, better captures the performance bottlenecks and scalability of the system than per-query latency.

To validate that our optimisations preserve functional correctness, we consider the accuracy of the system on the MATH dataset. We run the original 'Search and Learn' system on the MATH dataset to establish an accuracy baseline and rerun our optimised system on the same dataset to test that this accuracy has been maintained. If it has, we can conclude with some degree of confidence that our optimisations have not altered the core behaviour of the system as it pertains to output content. Notably, we do not compare outputs directly between systems, as the inherent non-determinism in vLLM, due to factors such as sampling temperature, makes meaningful per-token comparisons impossible.

Chapter 6

Investigations

This project seeks to understand what throughput improvements to a beam search system are possible by optimising memory footprint. This encompasses both inter-model memory considerations, like the split of memory between models, as well as intra-model considerations, like how a model's KV cache is managed. To that end, we sought to answer the following questions through a series of experiments and investigations:

- **What are the components of the LLM and PRM's memory requirements?**

- **How does changing the GPU memory split between LLM and PRM impact performance?**

- **What factors influence the KV prefix cache hit rate?**

These investigations, and their relation to our proposed optimisation, are detailed below.

6.1   Model Memory Requirements

We initially seek to better understand the memory footprint of a given model. By understanding this in greater detail, we can more effectively design strategies to use memory optimally.

To do this, we hook into the code that vLLM uses to determine the size of a model's KV cache. This process involves running a dry inference pass with the maximum number of tokens the model is configured to handle in a single request. After this run, vLLM records the peak memory allocated by PyTorch. It also records any memory allocations that occur outside of PyTorch - these may arise as result of libraries like NCCL using GPU memory for communication primitives. This total memory usage is then broken down into several components and used to determine the size of the KV cache.

Firstly, there is memory used for model weights - this is static and a function of the model used. After this, we infer the memory used for intermediate activations during the dry run - this is not explicitly reported but can be assumed based on the remaining PyTorch-allocated memory left after accounting for model weights. Non-PyTorch memory allocations, such as those for NCCL communicators and other runtime overheads, are accounted for separately by subtracting the peak GPU memory usage as measured by CUDA from the peak GPU memory used by PyTorch. The memory remaining after these components are accounted for is allocated for vLLM's KV cache blocks.

$$\text{Mem}_{\text{KVCache}} = \text{Mem}_{\text{Total}} \times \text{Utilisation} - (\text{Mem}_{\text{Activations}} + \text{Mem}_{\text{Weights}} + \text{Mem}_{\text{NonTorch}})$$

As a result, when we vary vLLM's `gpu_model_utilization` parameter, which controls what fraction of total available memory is used for the KV cache, we observe that only the memory allocated for the KV cache changes, with the allocations for weights, activations, and non-PyTorch components remaining fixed. This is best observed in figure From this, we can confirm that the `gpu_model_utilization` parameter exclusively affects how much space is reserved for the KV cache.

6.1.1 KV Cache Blocks

As part of its PagedAttention mechanism, vLLM breaks down the memory reserved for the KV cache into smaller segments called blocks.
A block stores the KV cache entries for a fixed number of tokens. The size of these blocks is dictated by the architecture of the model used - specifically the number of KV heads per attention layer and the total number of these layers. This determines how much memory is required to store the KV cache entry for one token.

$$\text{NumKVBlocks} = \frac{\text{Mem}_{\text{KV}}}{\text{NumTokensPerBlock} \times \text{Mem}_{\text{Token}} \times \text{NumLayers}}$$

We can see the impact of varying `gpu_model_utilization` on the number of KV blocks available for both the LLM and PRM in

6.2 Memory Split

After analyzing each model's memory requirements, we varied the `gpu_model_utilization` parameter to identify the optimal memory split between the LLM and PRM for performance. This was done primarily to establish a performance baseline, but also to explore how system dynamics shift as different amounts of KV cache are allocated to each model. Notably, system throughput does not necessarily scale linearly with changes in this split, since different models can become bottlenecks in different parts of the system.
We set a fixed batch size of and vary `gpu_model_utilization` between 0.08 and 0.5, with both models living on the same GPU. These thresholds were established based on the minimum and maximum amount of memory required by the LLM and PRM.
We can see in 6.1 that overall system throughput peaks when the `gpu_model_utilization` parameter is set to 0.3, which we adopt as the baseline configuration for performance comparisons.
At the lower bound of GPU memory allocation, PRM throughput degrades significantly, leading to a corresponding drop in overall system performance. To investigate this behavior, we analyze throughput in terms of KV cache block allocation dynamics.

6.2.1 Block Level Analysis

As described earlier in 3.1.1, vLLM employs a KV cache manager construct to allocate and manage space in the KV cache at the granularity of fixed-size blocks. While the PRM is servicing requests, the KV cache manager must ensure that sufficient blocks are available to hold the corresponding KV values. When GPU memory is constrained, and the number of free blocks falls below demand, the KV cache manager is forced to evict existing blocks to make space for new ones. This eviction incurs a recomputation cost - the evicted KV values must be regenerated by re-running the corresponding prefix through the model. As a result,
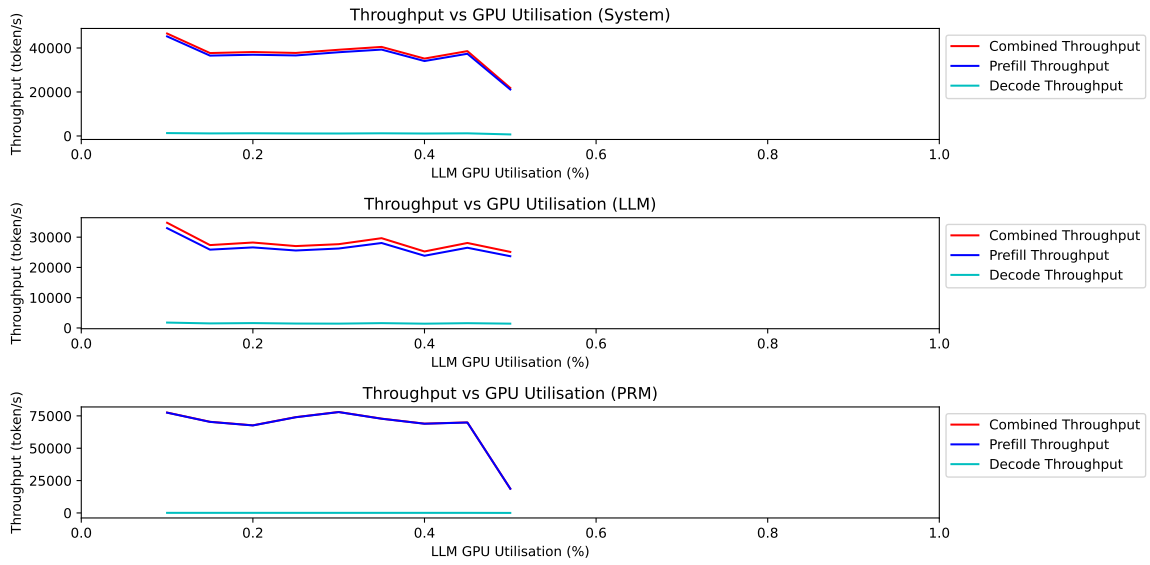
Figure 6.1: Impact on overall system throughput of varying the split in GPU memory between LLM and PRM.

sequences relying on prefix caching no longer benefit from fast lookup and instead trigger repeated compute overhead, directly impacting throughput.

This explains the throughput degradation noted earlier and is best illustrated in figures 6.2 and 6.3. When the PRM is allocated less KV cache memory, it holds fewer KV cache blocks and so evictions are necessary to service requests, reducing throughput.
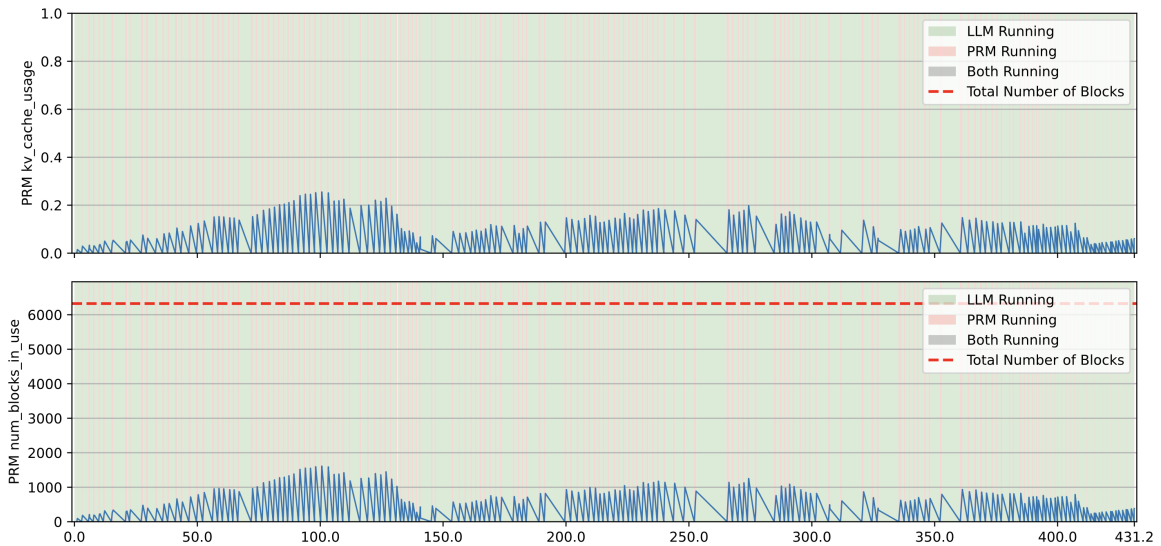

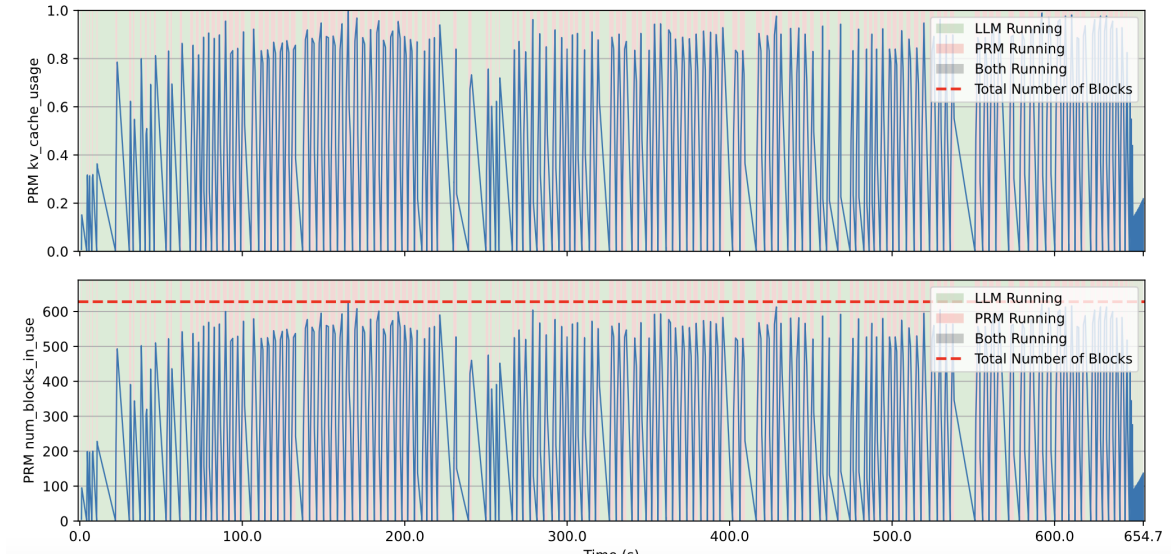**Figure 6.2:** Number of PRM KV cache blocks allocated at each scheduling step for `gpu_model_utilization`=0.3.

Figure 6.3: Number of PRM KV cache blocks allocated at each scheduling step for `gpu_model_utilization`=0.5.

6.3 Batch Size

In order to achieve optimal system performance, we sought to balance the throughput between the LLM and PRM in order to avoid one particular model becoming a bottleneck. This requires tuning the batching parameters of both models individually.

Unlike traditional implementations where each request statically allocates a contiguous chunk of KV cache memory based on the maximum size of a potential request, vLLM avoids overallocation through its use of PagedAttention, passing only the KV values necessary for the given input tokens. This enables the system to fit more active sequences into memory simultaneously, increasing effective batch sizes.

In the context of batched inference, we are concerned not only with how many tokens are passed to each attention kernel invocation, but also with the composition of these tokens in terms of how many distinct sequences they represent. Choosing larger batch sizes may improve hardware utilisation, but excessively heterogeneous batches composed can potentially also introduce fragmentation if block sizes are large relative to typical sequence lengths, thus limiting efficiency.

To explore this tradeoff, we varied two key parameters in vLLM's scheduling configuration:

- `max_num_batched_tokens`: This sets the maximum number of input tokens passed into a single attention kernel, directly controlling the overall compute workload per kernel call.

- `max_num_seqs`: This limits how many distinct requests can contribute tokens to that kernel, thereby determining the diversity and potential fragmentation of the batch.

Figures 6.4 and 6.5 illustrate the effect of these parameters on PRM and LLM throughput, respectively.
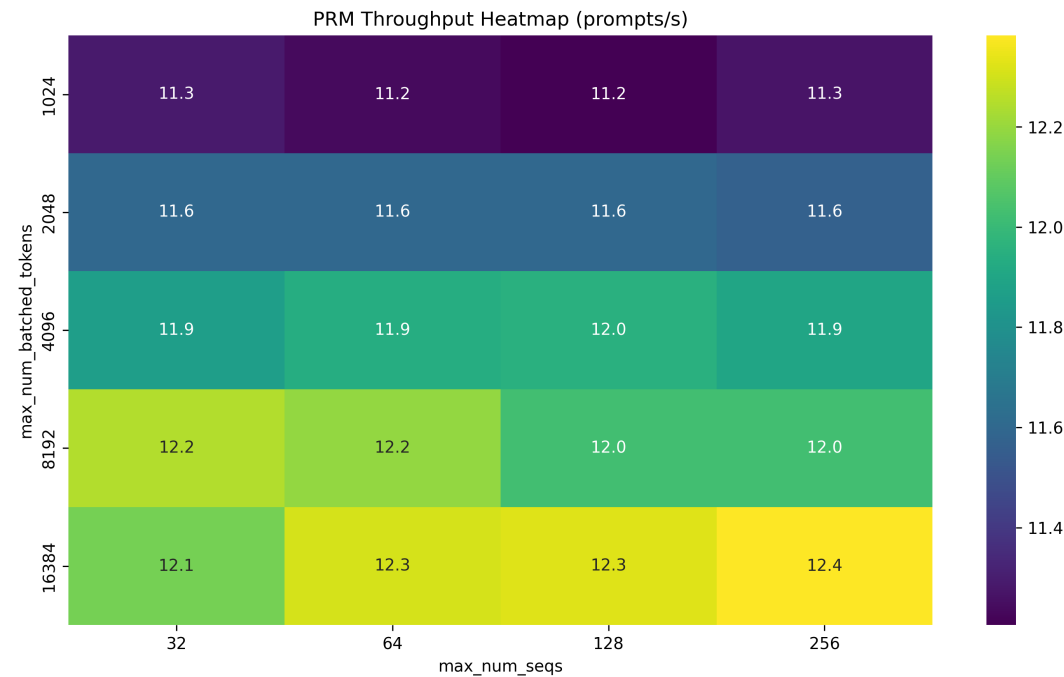
Figure 6.4: Analysing the effect of varying `max_num_seqs` and `max_num_batched_tokens` on PRM throughput.
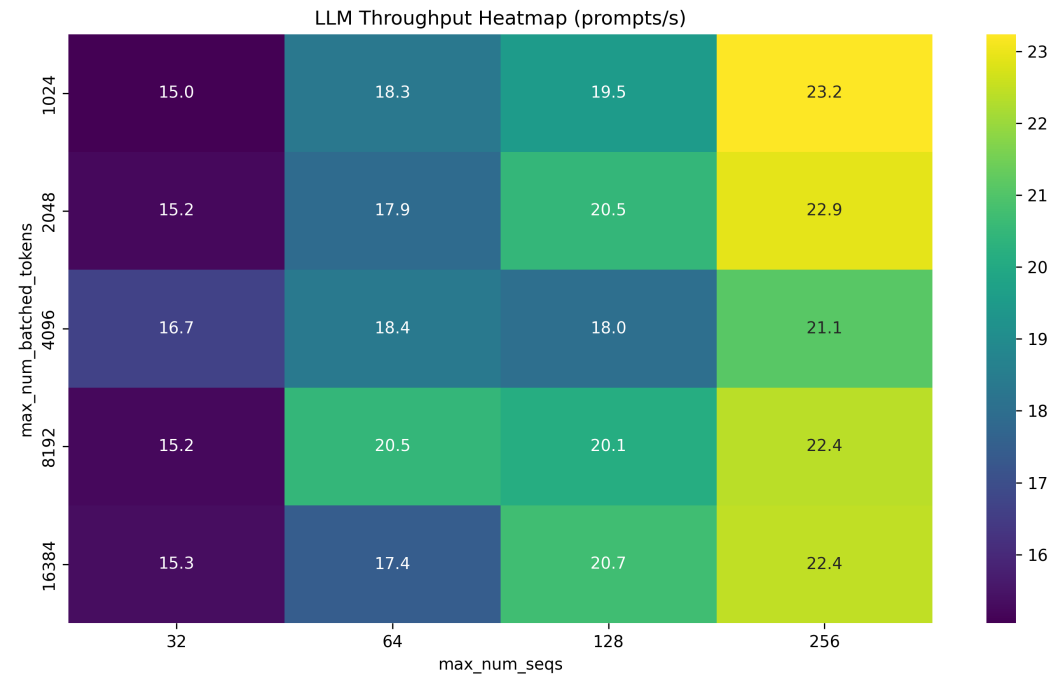

Figure 6.5: Analysing the effect of varying `max_num_seqs` and `max_num_batched_tokens` on LLM throughput.

6.4 KV Prefix Cache

Efficient use of the GPU during inference relies on two complementary strategies - maximizing compute saturation and minimizing redundant computation. Batch size directly affects GPU utilisation - larger batches enable more parallelism and ensure each kernel invocation does more useful work per unit time. At the same time, reducing redundant computation via KV cache reuse is critical to maintaining high throughput in tree-search systems like ours where prefix reuse is common. High prefix cache hit rates avoid recomputation of the tokens in these shared prefixes while freeing up compute for new inference work.

To that end, we examine how prefix cache hit rates vary with different settings of `gpu_model_utilization`.
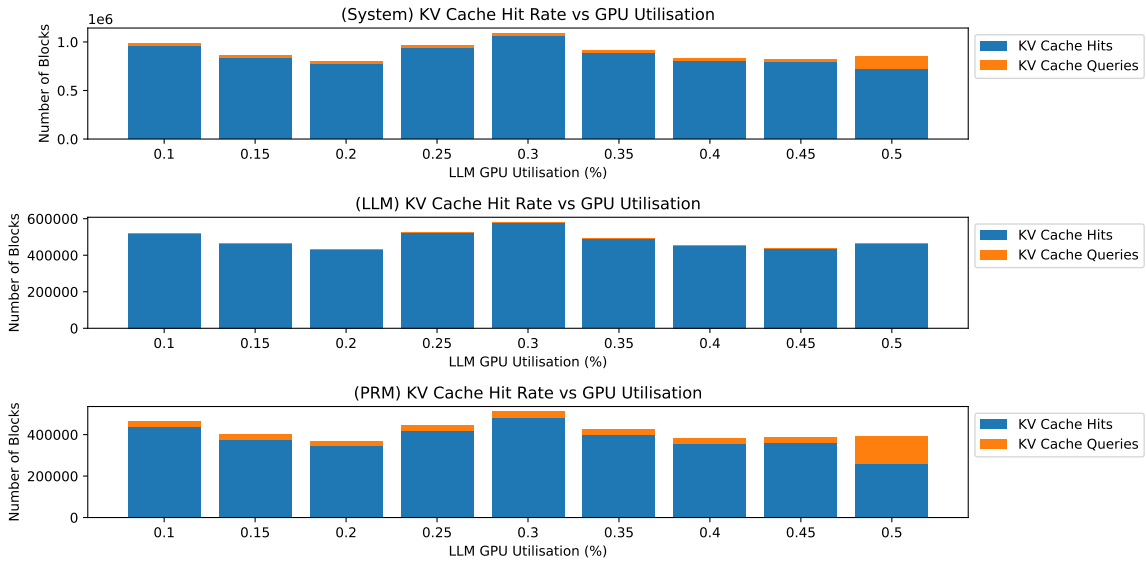
6.4.1 Cache Hit Rates

Figure 6.6: KV prefix cache hit of both the LLM and PRM across various gpu utilisation splits.

Figure 6.6 shows that the prefix cache hit rate remains high across a wide range of utilisation levels, indicating stable reuse behavior despite memory pressure. This suggests that the access patterns inherent in our joint LLM-PRM search space exploration are well suited to vLLM's existing eviction policy - prefixes are reused frequently and close in time to when they were first cached.

6.4.2 Cache Thrashing

To further evaluate whether the existing vLLM eviction policy causes performance regressions, we analysed cache thrashing — cases where a prefix block is evicted and later needed again, resulting in a recomputation that could have been avoided. To detect this, we modified vLLM's KV cache manager code to maintain a global set of seen KV block hashes. On each prefix cache miss, we check whether the block has been seen before but is no longer present in memory. If so, this indicates a missed opportunity for reuse due to eviction.

As shown in Figure 6.7, such thrashing events are extremely rare.

This confirms that vLLM's existing eviction policy is well matched to the temporal reuse patterns of our workload. This tightly scoped reuse window, with cache entries being reused
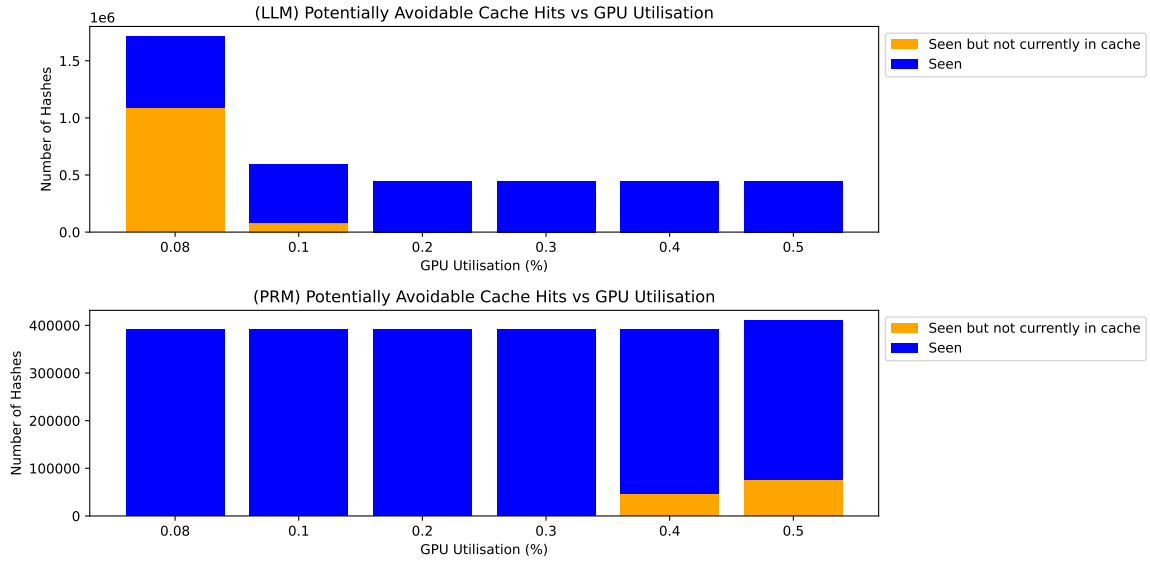
Figure 6.7: Frequency of prefix cache thrashing events across utilisation levels.

immediately in the next search step or not at all, explains why cache thrashing is rare and why the current eviction policy performs so well. Overall, our results indicate that prefix caching and memory management in our system are highly aligned with the needs of search-based multi-model inference workloads.

Chapter 7

Optimisations

Here we propose two optimisations motivated by our investigations into the performance characteristics of our beam search system implemented in vLLM. We explain how we were able to integrate one of these optimisations into our existing test-time compute scaling implementation, before detailing and justifying another potential optimisation based on trends we observed in our investigations.

7.1 Decode-Aware Speculative Tree Expansion

A key challenge in our setup is the mismatch in throughput between the LLM and PRM components of our pipeline. While the LLM can process prompts at significantly higher throughput, the PRM is comparatively slower and often acts as the bottleneck. To address this imbalance, we introduce a decode-aware speculative execution strategy that allows the LLM to speculatively decode multiple reasoning steps ahead, even before the PRM has scored all current beams.

This approach enables the LLM to work with larger batch sizes and more fully utilise its decoding capacity. Crucially, it also has the potential to reduce the number of future LLM invocations, which can be expensive both in terms of latency and compute. The method complements the decode-heavy nature of the LLM by effectively amortising LLM work over speculative future states.

7.1.1 Implementation

Our implementation required two main modifications - one to the underlying vLLM system, and one to our beam search implementation:

1. **Sampling API Extension for Controlled Speculation:** We extend the existing vLLM sampling parameters API by introducing a new parameter `n_stop_occurrences`. The standard API only provides a `stop` parameter, which halts generation upon the first occurrence of a stop string. However, our speculative expansion requires finer control, allowing the LLM to continue decoding until a stop string has been encountered a specified number of times.

   To implement this, we modify vLLM's `IncrementalDetokeniser` class to track and decrement a counter for `n_stop_occurrences`. Only when this counter reaches zero do we issue a stop signal. This allows the LLM to decode multiple reasoning steps in one go, while still providing a clear boundary for where to stop speculative decoding.

2. **PRM-Aware Beam Search Adjustments:** We adapt the beam search algorithm to only invoke the PRM on the non-speculative steps—i.e., those for which we are guaranteed to need scoring. Once the LLM has speculatively decoded future steps, we then check if any of these align with the beams selected by the PRM. If they do, we reuse the speculative outputs, avoiding redundant LLM invocations. If not, we discard the speculative paths and continue decoding as usual. This selective reuse saves computation and reduces latency without compromising the correctness of beam selection.

7.2 Throughput Min-Max Tuning

In the single-GPU setting, LLM and PRM components must share limited memory and compute resources. Each model has distinct batch scaling characteristics, and allocating GPU memory to one model reduces the resources available to the other. This introduces a trade-off in choosing optimal batch sizes and memory allocations.

Given the data dependency between LLM and PRM, pipelining them offers a natural solution for improved performance. In a pipelined setup, the system throughput is bounded by the slowest stage, which motivates a min-max tuning strategy: we aim to select utilisation levels and batch sizes that maximise the overall throughput, given that the pipeline throughput is determined by its minimum-performing component.

7.2.1 Implementation

We formalise this optimisation using profiling data collected for various batch sizes and utilisation levels for both the LLM and PRM. Specifically:

- We construct a two-dimensional array $T \in \mathbb{R}^{m \times n}$, where each row corresponds to a particular GPU memory allocation (i.e., utilisation level for the LLM vs PRM), and each column corresponds to a batch size.

- For each model (LLM and PRM), we obtain throughput values T^{LLM} and T^{PRM} of shape $m \times n$.

- For each utilisation level i, we compute the element-wise minimum throughput across batch sizes:
$$\mathrm{min}_i = \min\left(T^{\mathrm{LLM}}_{i,:},\ T^{\mathrm{PRM}}_{i,:}\right)$$

- Then, we take the maximum over batch sizes for each utilisation level:
$$\mathrm{max}_i = \max_j\left(\mathrm{min}_i[j]\right)$$

- Finally, we select the utilisation level i^* that yields the highest throughput:
$$i^* = \arg\max_i\left(\mathrm{max}_i\right)$$

This gives us the optimal utilisation level i^* and corresponding batch size j^* to maximise pipeline throughput, respecting the throughput bottleneck imposed by the slower model.

Parallel Execution Note: For this pipelined configuration to be effective on a single GPU, it is essential that both models can run concurrently. This is not possible with standard Python threading due to the GIL and the GPU's default single-stream execution behaviour. We therefore rely on NVIDIA's CUDA Multi-Process Service (MPS), which enables concurrent execution of workloads from different processes. This ensures that both LLM and PRM can make progress simultaneously, fully utilising GPU resources.

Chapter 8

Evaluation

Chapter 9

Conclusion and Future Work

Chapter 10

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, 11

[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, 11, 12, 20

[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] Kaplan J, McCandlish S, Henighan T, Brown TB, Chess B, Child R, et al. Scaling laws for neural language models. arXiv preprint arXiv:200108361. 2020. pages 9

[24] Snell C, Lee J, Xu K, Kumar A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. arXiv preprint arXiv:240803314. 2024. pages 9, 18

[25] Wei J, Wang X, Schuurmans D, Bosma M, Xia F, Chi E, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems. 2022;35:24824-37. pages 9

[26] Wang X, Wei J, Schuurmans D, Le Q, Chi E, Narang S, et al. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:220311171. 2022. pages 9

[27] Stiennon N, Ouyang L, Wu J, Ziegler D, Lowe R, Voss C, et al. Learning to summarize with human feedback. Advances in neural information processing systems. 2020;33:3008-21. pages 9

[28] Feng X, Wan Z, Wen M, McAleer SM, Wen Y, Zhang W, et al. Alphazero-like tree-search can guide large language model decoding and training. arXiv preprint arXiv:230917179. 2023. pages 10

[29] Yao S, Yu D, Zhao J, Shafran I, Griffiths T, Cao Y, et al. Tree of thoughts: Deliberate problem solving with large language models. Advances in neural information processing systems. 2023;36:11809-22. pages 10

[30] 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 11

[31] 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 11, 14

[32] Kilburn T, Edwards DB, Lanigan MJ, Sumner FH. One-level storage system. IRE Transactions on Electronic Computers. 1962;(2):223-35. pages 12

[33] OpenAI API;. Accessed: 2025-01-22. `https://openai.com/blog/openai-api`. pages 12

[34] FastAPI;. Accessed: 2025-01-22. `https://github.com/fastapi/fastapi`. pages 12

[35] Nvidia NCCL;. Accessed: 2025-01-22. `https://developer.nvidia.com/nccl`. pages 13

[36] Zheng L, Yin L, Xie Z, Sun CL, Huang J, Yu CH, et al. Sglang: Efficient execution of structured language model programs. Advances in Neural Information Processing Systems. 2024;37:62557-83. pages 13

[37] 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 14

[38] ggml;. Accessed: 2025-01-22. `https://github.com/ggerganov/ggml`. pages 14

[39] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems. 2019;32. pages 14

[40] Ray;. Accessed: 2025-01-22. `https://www.ray.io`. pages 14

[41] Agrawal A, Kedia N, Panwar A, Mohan J, Kwatra N, Gulavani B, et al. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24); 2024. p. 117-34. pages 15

[42] Kamath AK, Prabhu R, Mohan J, Peter S, Ramjee R, Panwar A. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2; 2025. p. 897-912. pages 15

[43] Hazell J. Spear phishing with large language models. arXiv preprint arXiv:230506972. 2023. pages 17

[44] 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 17

[45] 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 17

[46] 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 17

[47] 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 17

[48] Beeching E, Tunstall L, Rush S. Scaling test-time compute with open models;. Available from: `https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute`. pages 18

[49] Grattafiori A, Dubey A, Jauhri A, Pandey A, Kadian A, Al-Dahle A, et al. The llama 3 herd of models. arXiv preprint arXiv:240721783. 2024. pages 19

[50] Dong H, Xiong W, Pang B, Wang H, Zhao H, Zhou Y, et al. RLHF Workflow: From Reward Modeling to Online RLHF. arXiv preprint arXiv:240507863. 2024. pages 19

[51] Wang H, Xiong W, Xie T, Zhao H, Zhang T. Interpretable Preferences via Multi-Objective Reward Modeling and Mixture-of-Experts. In: The 2024 Conference on Empirical Methods in Natural Language Processing; 2024. . pages 19

[52] Xiong W, Dong H, Ye C, Wang Z, Zhong H, Ji H, et al. Iterative Preference Learning from Human Feedback: Bridging Theory and Practice for RLHF under KL-Constraint. ICML. 2024. pages 19

[53] Hendrycks D, Burns C, Kadavath S, Arora A, Basart S, Tang E, et al. Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:210303874. 2021. pages 19

[54] Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, et al. Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:191003771. 2019. pages 20