



UNIVERSITY OF
CAMBRIDGE

Department of Computer
Science and Technology

Scheduling of Distributed LLM Serving on Heterogeneous GPUs

Nathan Rignall

Churchill College

June 2025

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 63

Main chapters (excluding front-matter, references and appendix): 51 pages (pp 9–59)

Main chapters word count: 14866

Methodology used to generate that word count:


With pdf images disabled: [

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - \
    -dFirstPage=9 -dLastPage=58 report-submission.pdf | \
    egrep '[A-Za-z]{3}' | wc -w
14866
```

]

Declaration

I, Nathan Rignall of Churchill College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: 

Date: 11/06/2025

Abstract

The significant operational cost of Large Language Model (LLM) inference presents a considerable challenge to their continued widespread adoption. Compounded by rising GPU prices and variable hardware availability, deploying large-scale homogeneous clusters is often financially prohibitive. Heterogeneous GPU clusters offer a solution to mitigate these significant costs. However, existing scheduling systems do not adequately address the unique computational patterns of new Mixture-of-Expert (MoE) models within these complex environments.

This research project addresses this gap by developing and evaluating a novel scheduling algorithm, specifically designed to optimise MoE model inference on heterogeneous GPU clusters. The algorithm is separated into two distinct processing stages: an outer loop and an inner loop. The outer loop uses Bayesian Optimisation to efficiently search the complex configuration space, partitioning an inventory of different GPUs into small optimal 'islands'. For the inner loop, this work implements a new linear programming formulation that precisely maps workload ranges, categorised by input sequence length and separated into prefill and decode phases, to the generated islands.

The scheduling algorithm was evaluated using a simulation framework on real-world workload traces. Results demonstrate that this approach, when applied to a heterogeneous GPU cluster, can achieve a 1.4x throughput improvement over a homogeneous cluster of the equivalent cost. This work concludes that a workload-aware scheduling approach can unlock substantial performance and cost-efficiency gains for serving large-scale MoE models in complex, heterogeneous environments.

Acknowledgements

I would like to express my gratitude to my co-supervisor, Y. Jiang, for his invaluable support throughout this project. His guidance was instrumental in my rapid understanding of the research problem and developing a solution to the research question. I would also like to thank my supervisor, Dr. E. Yoneki, for her constructive feedback on my drafts and her overall support in this project's completion.

Contents

1	Introduction	9
1.1	Motivation	9
1.2	Contributions	10
1.3	Outline	11
2	Background	12
2.1	Transformer Models	12
2.1.1	Computation Steps	13
2.1.2	Prefill and Decode	13
2.1.3	Key Value caching	14
2.1.4	Mixture of Experts	14
2.1.5	DeepSeek Architecture	14
2.2	Parallelism Strategies	14
2.3	Scheduler Technologies	15
2.3.1	Search-Based	16
2.3.2	Constraint Programming	16
2.4	Inference Challenges	17
3	Related Work	18
3.1	Phase Splitting	18
3.1.1	Splitwise	19
3.1.2	DistServe	19
3.2	Heterogenous Serving	20
3.2.1	HexGen	20
3.2.2	Thunderserve	21
3.3	Summary and Future Work	21
4	Design and Implementation	22
4.1	Problem Description	22
4.2	Simulator	23
4.2.1	Shallowsim Implementation	24
4.2.2	Modifactions	24

4.2.3	Usage	24
4.3	Workload Modeling	25
4.3.1	Input Sequence Length	25
4.3.2	Decode Length	26
4.4	Objectives	27
4.4.1	Throughput Modelling	27
4.4.2	Throughput Allocation	28
4.5	Constraint Optimisation	28
4.5.1	Constraint Formulation	28
4.5.2	Challenges	29
4.6	Scheduling Assumptions	30
4.6.1	Prefill Parallelism	30
4.6.2	Decode Parallelism	31
4.6.3	Solver Resolution	33
4.7	Problem Formulation	35
4.8	Inner Loop	36
4.8.1	Constraint Formulation	37
4.8.2	Implementation	40
4.8.3	Optimisation	40
4.9	Outer Solver	41
4.9.1	Direct Island Generation	41
4.9.2	Divider Island Generation	43
5	Evaluation	47
5.1	System Performance	47
5.2	Inner Loop	48
5.2.1	Solver Results	49
5.2.2	Evaluator Results	51
5.3	Outer Loop (Divider Approach)	52
5.3.1	Batch Size Analysis	52
5.3.2	Skew Range Analysis	53
5.3.3	Minimum Island Size Analysis	54
5.4	Outer Loop (Direct Approach)	54
5.4.1	Batch Size Analysis	54
5.4.2	K Slots Analysis	56
5.5	Workload Traces	56
5.6	Discussion	57
6	Conclusion	58
6.1	Future Work	59

List of Figures

4.1	Sample service architecture of LLM inference engine	23
4.2	Sample probability distribution of input sequence lengths	26
4.3	Selection of GPUs benchmarked over the prefill stage using the <i>Shallowsim</i> simulator	31
4.4	Selection of GPUs benchmarked over the decode stage using the <i>Shallowsim</i> simulator	33
4.5	Problem formulation demonstrating the data flow between the <i>inner</i> and <i>outer</i> loop operations.	36
4.6	Diagram showing effective purpose of <i>inner loop</i>	37
4.7	Diagram showing effective purpose of <i>outer loop</i>	41
4.8	Graph displaying the effect of changing the S_{skew} value	44
5.1	Prefill and decode benchmark results for test case	49
5.2	Assignment values percentage (per range) for each island test case	50
5.3	Throughput per range for each island test case	50
5.4	Throughput per sequence length for each island test case	51
5.5	Throughput per iteration number for different batch sizes	52
5.6	Execution time of scheduler for different batch sizes	53
5.7	Throughput per iteration number for different skew ranges (Divider)	53
5.8	Throughput per iteration number for different minimum island sizes (Divider)	54
5.9	Throughput per iteration number for different batch sizes (Direct SAASBO)	55
5.10	Throughput per iteration number for different batch sizes (Direct BO)	55
5.11	Execution time of direct scheduler (BO vs SAASBO) for different batch sizes	55
5.12	Throughput per iteration number for different K slots (Direct SAASBO)	56
5.13	Azure LLM Context and Generated Tokens	56

List of Tables

5.1	GPU inventory configurations with resultant performance metrics on the Azure LLM dataset for code and conversation.	48
5.2	GPU island test cases for <i>inner loop</i> analysis, with two homogeneous scenarios and two heterogeneous scenarios.	48

Chapter 1

Introduction

In this chapter, we express the motivation behind the development of a novel scheduling algorithm for Large Language Model (LLM) inference. We begin by outlining the significant cost and performance challenges associated with serving these models, then express the limitations of existing scheduling approaches. Finally, we provide an overview of our key contributions and outline the subsequent chapters of this report.

1.1 Motivation

Large Language Models (LLMs) such as GPT-4 [1], Llama3 [2], Gemini [3], and Mixtral [4] have demonstrated outstanding performance across a wide range of real-world applications. These include coding assistants, chatbots, tools for content creation, translation services, data analysis, and even with scientific research, highlighting their versatility and impact across industries. However, the significant computational resources required to serve these models make them extremely costly to operate and use [5].

Enhancing the cost-efficiency of LLM inference is therefore vital for broader adoption and for democratising access to these cutting-edge technologies. While service providers initially relied on mostly homogeneous hardware [6], rising GPU costs and variable availability [7] make such environments impractical to maintain. The rationale for embracing heterogeneous clusters is that different GPU types exhibit distinct compute and memory characteristics, which can be aligned with the different resource demands of diverse inference request types [8]. Therefore, making better use of available heterogeneous hardware to save costs is desired by both users and service providers.

Previous research has demonstrated the benefits of *phase splitting*, where the compute-bound prefill and memory-bound decode stages of inference are separated to achieve significant gains in efficiency and performance [9]. Initial systems like Splitwise [10] and DistServe [9] pioneered this approach in homogeneous environments. Building upon this, more recent works, such as HexGen [11], HexGen-2 [12] and ThunderServe [13], have

extended these principles to heterogeneous hardware, using complex algorithms to manage diverse resources. The core challenge with optimising heterogeneous LLM deployment lies in the scheduling of requests itself. Efficient model serving requires co-optimising the system deployment (resource allocation and parallelism strategies) with the request routing strategy, which is an NP-hard problem [11].

However, these advanced scheduling systems do not explicitly consider the unique architectural and computational patterns of Mixture-of-Experts (MoE) models. Recent advancements, particularly from models like DeepSeek-V3/R1 [14], introduce additional complexities such as dynamic expert routing, all-to-all communication, and sophisticated KV caching strategies [15]. Furthermore, existing schedulers often avoid highly granular workload assignment strategies due to the exploding problem space, which makes finding an optimal solution computationally prohibitive. Hence, the specific problem of scheduling MoE inference across heterogeneous hardware currently remains under-explored.

This project aims to address this gap by developing a novel scheduling algorithm specifically designed for heterogeneous GPU clusters, for use with MoE models. We simultaneously investigate how to improve the granularity of workload assignment, whilst also considering the architectural characteristics of MoE in scheduling. We propose the following research question:

To what extent can heterogeneous hardware environments improve the efficiency of serving Mixture-of-Experts (MoE) models, and what scheduling strategies are required to realise these potential gains?

1.2 Contributions

This report presents a new workload-aware Mixture-of-Experts (MoE) Large Language Model (LLM) scheduling algorithm, for use with heterogeneous GPU clusters through several key advancements:

- First, we present a new two-level scheduling algorithm containing an *inner* and *outer* loop. The *outer loop* divides an inventory of GPUs into *islands*, the *inner loop* assigns workload and configures these islands. This segmentation enables the scheduler to efficiently search the large problem space, whilst leveraging problem specific solver technologies at each level.
- We build a linear optimisation technique for rapidly assigning specific workload ranges (prefill and decode) to GPU islands inside the *inner loop*. Using precise workload characterisation, we target mapping of specific input sequence lengths to specific GPU islands. This enables the most optimal use of heterogeneous hardware, as the input sequence length can significantly affect the performance requirements of the prefill and decode phase of inference.

- Next, we demonstrate the use of Bayesian Optimisation for searching how best to divide the GPUs into islands. Using a novel encoding method, we compress the problem space into just $|T| \times 2$, where $|T|$ is the total number of different GPU types. This enables a simple Gaussian Process to search the complex division process efficiently, without incurring issues with high dimensionality.
- We build an advanced simulation system to benchmark our GPU configurations using the state-of-the-art Shallowsim [16] simulator. This simulation system enables us to model expected throughput of an entire system, given an expected request workload trace.
- Finally, we perform extensive analysis of both the scheduling algorithm and simulation system to express their performance characteristics. We use this data to demonstrate that for any given workload type (conversation and code), we see substantial efficiency gains. Comparing a heterogeneous cluster to a homogenous one, we see a performance improvement of up to $1.4\times$.

1.3 Outline

Chapter two first provides the essential background on transformer models, specifically focusing on their computational steps during inference. Next, chapter three introduces the related work, reviewing the existing literature on phase splitting and homogenous serving whilst identifying the gaps for future exploration. The design is presented in chapter four, this includes details of the novel scheduling algorithm and explains reasons behind the design decisions. Chapter five performs comprehensive evaluation of the scheduler using simulator technology, demonstrating the choice of parameters and benefits of heterogeneous versus homogenous clusters. Finally, chapter six concludes by summarising the findings and discussing potential avenues for future work.

Chapter 2

Background

Within this Chapter we introduce the concepts behind transformer models, their associated computational steps and the different phases of inference. We also introduce several technologies which can be used for building schedulers in computer systems.

2.1 Transformer Models

Transformer models are a type of deep learning architecture specifically designed to handle sequential data such as natural language. Unlike recurrent neural networks (RNNs) which process sequences one step at a time, transformers can analyse entire sequences of data in parallel. This is achieved through self-attention, where each token in a sequence can weigh and incorporate information from all other tokens. Such attention is calculated by adding positional encodings to input embeddings which transforms each token into query, key and value (KV) vectors [17].

The first transformer models were designed for encoder-decoder tasks such as translation, where an input sequence is encoded and decoded to an output sequence [18]. The encoder processes the input sequence and produces contextualised hidden representations of the text. Subsequently, the decoder generates output tokens one by one, using masked self-attention to prevent access to future tokens during generation [17]. Encoder-only models are typically used for classification and sentence embeddings [19], whereas decoder-only are often used for autoregressive generation (chatbots, coding assistants) [20].

Transformers form the basis of modern natural language processing due to their scalability, parallelisation benefits, and excellent performance on long input sequences. An LLM, or Large Language Model, is a type of transformer model that has been trained on a massive amount of text data. Their “large” size refers to the vast number of parameters they contain, which is a key factor in their ability to capture complex patterns in natural language.

2.1.1 Computation Steps

There are several computation steps involved in a transformer model’s forward pass during inference. The process begins with input embeddings and positional encodings, where each token in the input sequence is converted into a dense vector using a learned embedding matrix. Positional encodings are then added to incorporate information about the position of each token in the sequence. Next, the model computes the Query (Q), Key (K), and Value (V) vectors for each token, by projecting the input embeddings through learned linear transformations. Using the Q, K, and V vectors, the model performs scaled dot-product attention to compute attention scores and weigh the significance of other tokens in the sequence relative to each token.

The output of the attention mechanism is passed through a residual connection and a layer normalisation step. This is followed by a position-wise feed-forward network, which consists of two linear transformations with a ReLU activation (or GELU in some models) in between. In deep transformer models, these components (attention, residual connection, normalisation, and feed-forward layers) are stacked multiple times, where the output of one layer serves as the input to the next. Finally, the output of the last Transformer layer is passed through a linear projection layer to produce the final output, depending on the task.

2.1.2 Prefill and Decode

We can separate the processing phases of LLM model inference steps into two distinct categories, with clear differences in hardware processing requirements.

The prefill stage processes the initial input tokens to compute the transformer model’s intermediate states. This includes calculating the positional embeddings and populating the (KV) cache, which are subsequently used to generate the first new token [21]. This specific phase is easily parallelised because at every stage of computation (per token) the full input sequence is known. Hence, this is viewed as a matrix-matrix operation which can easily saturate a GPUs available compute [22]. *In reference to a transformer’s computation steps, all stages are performed for every token in the sequence but can be heavily parallelised.*

The decode phase generates each output token one at a time, this is because each token computation is dependent on the previous token [21]. Hence, each token must be calculated sequentially. The speed at which the decode phase can execute is heavily dependent on the speed at which weights, keys, values and activations can be loaded into the correct areas of GPU memory. This phase usually saturates the memory bandwidth of a GPU before saturating the compute performance [10]. *In reference to a transformer’s computation steps, all stages are performed, but for only the new token generated, this is due to key-value caching.*

2.1.3 Key Value caching

Key value caching is an optimisation technique used during the inference steps for transformer-based decoder-only models such as GPT [20]. This technique avoids recomputing the tensors for each input token for every decode token generated. Instead weights and key values are cached in GPU memory, to be used in the next iteration. At each decoding step the Q value of the new token is computed, reusing cached K and V values from earlier tokens in the sequence [23]. This technique comes at the cost of using more GPU memory, reducing the maximum model size that theoretically could be loaded.

2.1.4 Mixture of Experts

Mixture of Experts (MoE) is a neural network architecture designed to scale model capacity without a proportional increase in computational cost [24]. Unlike dense transformers where all parameters are used for every input, MoE models contain a pool of expert sub-networks, of which only a small subset are activated during inference for any given input token.

In a typical MoE transformer layer, a router network determines which experts to activate for each token. This is often achieved using top-k routing, where the router selects the top k experts based on a learned score for each input token. The selected experts are then applied to the input, and their outputs are combined before passing to the next layer [25].

2.1.5 DeepSeek Architecture

DeepSeek V3 and R1 are both LLMs that share a decoder-only Mixture-of-Experts (MoE) architecture with sparse expert activation, enabling over 670B total parameters while activating only approximately 37B per token. Both models support long-context inference (up to 128K tokens) and are trained using FP8 mixed-precision for efficiency [14].

A key feature of their model architecture is the use of KV cache compression during inference. Instead of storing full-precision key and value tensors for each expert, DeepSeek compresses these representations, significantly reducing memory bandwidth and enabling faster decode performance despite the model’s massive scale [15]. V3 is tuned for general conversation, while R1 builds on it with enhanced reasoning capabilities.

2.2 Parallelism Strategies

Data parallelism refers to replicating an entire model across multiple processing units, with each replica receiving a different slice of the input data. Hence, this technique is generally used to increase throughput of an overall system, especially in batched inference scenarios.

Model parallelism involves partitioning the model itself across several processing units, where each device is responsible for a portion of the model’s layers or operations. This is most helpful when a model cannot fit on a single device, due to memory demand greater than what is available.

Tensor parallelism is a specialised form of model parallelism, generally used with large deep learning models. Individual tensors are split across multiple devices (rather than whole layers), requiring frequent communication between devices to synchronise the results of the split tensor computations.

Pipeline Parallelism is another form of model parallelism, where models are divided layer-wise (sequentially) and inputs are processed in a pipeline. This is in contrast to tensor parallelism, which are split within layers and model parallelism which is more coarse grained.

Expert Parallelism is a technique used to efficiently serve MoE models, it involves placing different experts on different processing units. When the routing network selects specific experts for a given input, only those corresponding processing units become active to process the data in parallel. This allows for a significant increase in model size without a proportional increase in computation for every input.

Batching groups multiple inference requests together so that they can be processed simultaneously, maximising hardware utilisation and overall system throughput.

Static batching accumulates requests until a batch is full or a timeout is reached. All requests are processed together and must wait for the slowest one to finish, leading to inefficient GPU usage and higher latency. *Continuous batching* is a dynamic method that immediately adds new requests to the batch as soon as others complete. This keeps the hardware constantly active, significantly increasing throughput and reducing idle time.

Batching increases throughput at the cost of individual request latency. Continuous batching is superior as it offers much higher throughput while also providing lower and more predictable latency.

2.3 Scheduler Technologies

Schedulers are a fundamental concept in systems and networking, used in operating systems to cloud computing, to manage and allocate resources among competing tasks. In the context of LLM inference, their role is to solve the complex optimisation problem of mapping a high volume of diverse and unpredictable requests onto GPUs. A scheduler is the core component of an LLM serving system, implementing the logic for resource assignment that is essential for maximising hardware utilisation and meeting performance goals.

2.3.1 Search-Based

Search-based Optimisation refers to methods that systematically or semi-randomly explore a given search space to find an optimal solution. Such techniques are often used to tune a scheduler’s parameters (particularly in machine learning), but can also be used to directly construct a schedule.

Bayesian Optimisation (BO) is an efficient model-based approach for finding the maximum of expensive to evaluate functions. It builds a probabilistic surrogate model (an approximate function, that is often a Gaussian Process) of the objective function. An acquisition function then uses the surrogate model’s predictions to intelligently select the next set of parameters to evaluate. This process balances exploration (sampling in areas of high uncertainty) with exploitation (sampling where the model predicts high performance), allowing it to find good solutions with fewer evaluations compared to other methods search methods.

Standard Bayesian Optimisation often struggles when the number of parameters is very large due to the curse of dimensionality. To address this, specialised versions of BO, like **SAASBO (Sparse Axis-Aligned Subspace Bayesian Optimisation)** have been developed SAASBO overcomes this by assuming sparsity, allowing it to automatically identify and focus its search within the low-dimensional subspace of the most influential parameters [26].

Random Search operates by sampling configurations from the parameter search space uniformly at random and evaluating their performance In the context of scheduling, this involves randomly generating a number of candidate parameter solutions, evaluating the scheduler’s performance with each, and selecting the best one found.

Grid Search is a brute-force technique that performs an exhaustive search over a manually specified subset of the parameter space. It defines a discrete grid of values for each parameter and evaluates every possible combination. Its primary disadvantage is the curse of dimensionality, where the number of evaluations required grows exponentially with the number of parameters.

2.3.2 Constraint Programming

Both Constraint Programming and Mathematical Optimisation approaches formulate scheduling activities as an optimisation problem, with formal mathematical constraints and objectives.

Linear Programming (LP) techniques can solve scheduler problems where the constraints and objectives can be modelled as only linear. All decision variables are continuous and can be used in scheduling algorithms when fractional assignments of work are valid.

Mixed Integer Linear Programming (MILP) solvers can consider integer variables such as binary assignment values, extending LP. In the context of scheduling, this enables decisions to be modelled using yes/no decisions and combinatorial choices.

In the case where constraints cannot be as linear, **Non-Linear Programming (NLP)** can model problems where either the objective and/or constraints are non-linear functions. Given problems are non-linear, it is harder to guarantee a globally optimal solution is generated.

2.4 Inference Challenges

Inference for LLMs poses significant technical and operational challenges, especially at scale. Unlike traditional ML model, where inference is relatively lightweight, LLMs often require billions of parameters to be loaded and operated on for every user request. This results in high compute demands and considerable memory pressure, necessitating powerful and costly GPU infrastructure.

LLM Inference workloads also vary significantly depending on the application. For example, code generation workloads (e.g., copilots, IDE assistants) tend to have longer sequences and larger context windows, leading to slower decode-heavy operations. In contrast, conversational workloads (e.g., chatbots) often involve shorter prompts and more frequent interactions. Understanding and modelling these patterns is critical for system-level optimisation and for deploying LLMs in a cost-efficient, scalable manner.

Hence, to manage these complex demands, service operators rely on sophisticated schedulers and orchestration systems. These systems are, responsible for intercepting incoming inference requests, dynamically batching them to maximise throughput, and intelligently assigning them to the most suitable hardware. However, platforms typically operate on homogenous hardware to simplify resource management.

Chapter 3

Related Work

In this chapter we introduce the current state-of-the-art for large language model (LLM) inference techniques. We describe how *phase splitting* is currently being used to improve GPU cluster efficiency and reduce inference costs. Additionally, we describe how new advanced schedulers can be used to employ *phase splitting* in a heterogeneous environment to further optimise systems. Finally, we outline gaps in current work relating to heterogeneous *phase splitting* in Mixture-of-Experts (MoE) environments.

3.1 Phase Splitting

Previous works from Zhong *et al.* [9] and Patel *et al.* [10] introduced the concepts of *phase splitting* LLM requests into prefill and decode. Phase splitting is a technique that separates the two distinct computational stages of LLM inference, as described in subsection 2.1.2, onto separate hardware resources. This technique targets improving the overall cost efficiency of LLM inference, by improving the utilisation of hardware.

Running both LLM compute phases on the same physical hardware often leads to inconsistent end-to-end latencies [10]. This is because there is typically strong prefill-decode interference as a result of batching, where each phase waits for the other to finish (to compete for GPU resources) [9]. To solve these latency inconsistencies, service providers have resulted to overprovisioning resources to meet Service Level Objectives (SLOs) [9]. However, this results in sub-optimal use of hardware resources where available compute is often underutilised. This problem further worsens when the compute requirements for prefill and decode do not match each other, which is often the case in real-world scenarios.

Also, executing both prefill and decode on the same hardware limits the possible deployment scenarios of models, as prefill and decode have to use the same parallelism configurations [9]. Different parallelism techniques may be more optimal for prefill as compared to decode and visa versa. Hence, *phase splitting* provides important efficiency gains for LLM inference while demand from users is ever increasing.

3.1.1 Splitwise

Splitwise [10] is one of the first systems that formally introduced and analysed the benefits of *phase splitting*. The authors performed extensive analysis of LLM inference, characterising the prefill/prompt phase as compute-intensive and the decode/token phase as memory-intensive.

Splitwise divides the two inference phases onto separate machines using a hierarchical scheduling system. A Cluster-Level Scheduler (CLS) is responsible for machine pool management and for routing incoming requests to a prefill and decode machines. Pools are crafted using the analysed workload distribution (proportion of prefill vs decode) from public Azure LLM traces [10]. A Machine-Level Scheduler (MLS) then executes on each individual machine to manage its local request queue, making dynamic batching decisions.

During the initial prefill phase, the model processes the input prompt and populates the KV-cache. This cache is required for the decode phase and because Splitwise runs the prefill and decode phases on different machines, the KV-cache generated on the prefill machine must be transferred over the network to the decode machine. To minimise latency, Splitwise overlaps the KV-cache transfer with the prefill computation.

The authors perform evaluation of Splitwise on NVIDIA A100 and H100 virtual machines on Microsoft Azure. The evaluation demonstrates phase splitting brings substantial improvements in throughput, cost, and power efficiency without compromising on latency SLOs [10]. However, this work relies on high-end datacenter interconnects for its primary KV-cache transfer to function effectively, which may not always be available.

3.1.2 DistServe

Similar to Splitwise, DistServe [9] assigns the prefill and decoding phases to separate, dedicated machines, eliminating interference and allowing for independent optimisation. The authors optimise the system for “goodput”, the maximum request rate that can be served whilst still adhering to defined SLOs.

Zhong *et al.* [9] introduce an end-to-end distributed servicing system which includes a placement algorithm, orchestration system and parallel execution engine. The scheduling algorithm maximises per-GPU “goodput” by first optimising the configuration for a single model replica and then scaling this configuration to meet the required traffic rate. During online operation, DistServe uses a simple First-Come-First-Served (FCFS) scheduling policy, where incoming requests are sent to the prefill instance with the shortest queue. Subsequent decode operations are sent to the least-loaded decoding instance.

The paper evaluates the DistServe system in a homogenous environment, with four nodes containing eight NVIDIA A100s connected with NVLINK. Using public datasets [27] [28] [29], the authors compare their system against vLLM [30] and DeepSpeed [31] inference

systems. The authors claim DistServe can serve up to 7.4x more requests or handle 12.6x tighter SLOs compared to state-of-the-art systems Zhong *et al.* [9].

Like Splitwise, DistServe assumes the availability of high-quality, low-latency datacenter interconnects like NVLink for (KV) cache transfer. Hence, this system is less applicable to environments with constrained networking hardware. However, the authors do consider how parallelism strategies can be optimised for the prefill and decode phase independently, unlike Splitwise.

3.2 Heterogenous Serving

Building upon the concepts introduced with *phase splitting*, systems started to employ advanced scheduling methods to separate the prefill and decode phases onto specialised hardware. As the prefill and decode phase have different compute requirements it makes sense to target the workload deployment onto the most optimal hardware for the task. This is particularly relevant in heterogeneous environments such as the cloud, where a diverse mix of GPUs with varying costs, compute power, and memory bandwidths are available.

The scheduling algorithms introduced with Splitwise and DistServe both operate over a very small search space (due to the homogenous hardware). Hence, Jiang *et al.* [13] and Jiang *et al.* [11] work to develop more advanced scheduling algorithms, which optimise the time-to-solution and overall solution quality. The core challenge with heterogenous serving is not with the phase splitting itself, but with the scheduling algorithm used to organise the cluster.

3.2.1 HexGen

HexGen [11] is a distributed inference engine designed to serve LLMs over heterogeneous environments, aiming to mitigate the substantial costs associated with traditional centralised deployments. The authors use a two-part scheduling algorithm that solves the NP-hard scheduling problem.

The global optimisation processes uses a genetic algorithm to find the optimal partition of the global set of GPUs into multiple independent inference pipeline groups. This evolutionary algorithm gradually improves the solution by iteratively applying mutation operations to the GPU partitions, which allows the search to explore different configurations and converge on a high-quality layout. The inner pipeline optimisation process uses dynamic programming to determine the optimal layout of pipeline stages.

A unique contribution of HexGen is its support for asymmetric parallelism, meaning each pipeline stage can be assigned a different tensor parallelism degree and number of layers. This flexibility enables mapping of LLMs onto diverse GPUs with varying compute

capabilities. The authors claim that for the same budget, HexGen can achieve up to 2.3x lower latency deadlines or tolerate up to 4x more requests than a homogeneous baseline.

3.2.2 Thunderserve

Jiang *et al.* [13] build Thunderserve, another LLM serving system designed specifically for heterogeneous hardware. The authors use a novel two-level hierarchical scheduling algorithm to optimise the deployment plan across varied hardware and network conditions.

The upper level uses a search algorithm to determine the optimal group construction (how to partition GPUs) and phase designation (assigning groups to prefill or decode). Whereas the lower level, given the group assignments, deduces the optimal parallelism configuration and orchestrates request routing by framing it as a solvable transportation problem. The authors also introduce lightweight re-scheduling mechanisms, designed to adapt to fluctuating online conditions like workload shifts or node failures without service restarts. The system adapts to changing workloads by only adjusting phase designation and orchestration, to minimise user disruption from rescheduling.

Thunderserve presents a 2.1x increase in throughput and a 2.5x reduction in latency deadlines for the same price budget [13]. However, the KV cache compression is a lossy process that could negatively affect performance on tasks requiring perfect fidelity (additional analysis required) Also the search algorithm is heuristic-based so is not guaranteed to find the absolute global optimum solution.

3.3 Summary and Future Work

The initial works of DistServe [9] and Splitwise [10] established the initial concepts of *phase splitting* to improve LLM inference efficiency in primarily homogeneous settings. Building upon this, HexGen [11] and ThunderServe [13] developed more advanced schedulers to apply these principles to complex, heterogeneous cloud environments. HexGen introduced asymmetric parallelism for greater flexibility, while ThunderServe focused on holistic deployment optimisation and dynamic adaptation for the cloud.

A shared limitation across these systems is that they primarily focus on standard transformer architectures. They do not explicitly consider or optimise for the unique characteristics of Mixture-of-Experts (MoE) models, which have different architectural and computational patterns. We exploit this gap to develop a heterogeneous scheduler for MoE, targeted towards the state-of-the-art DeepSeek V3 model architecture [15].

Chapter 4

Design and Implementation

This chapter outlines the steps taken to design and implement a new scheduler for LLM workloads, with support for Mixture-of-Experts models on heterogenous hardware. We begin by outlining the problem description of an LLM scheduler, introducing the terminology used throughout this section.

Next, we explain the choice of simulator used for modelling the system’s expected throughput for a given workload. We then use this simulator to define the objectives for a bounded-constraint optimisation problem, which represents a subset of the overall scheduler. Then, we describe several scheduling assumptions that can be used as optimisation strategies in the complete scheduler.

We finish by bringing together the workload modelling, objectives, constraint optimisation and scheduling assumptions into a single problem formulation. Here, we separate the scheduler into a two-stage problem, with an *inner* and *outer* loop. The *inner loop* assigns specific workloads to GPU islands using deterministic bin-packing. In contrast the *outer loop* performs a heuristic search to find how best to slit up the GPU inventory into islands. Using this formulation, we describe the specific design and then implementation of the two loops, in preparation for evaluation and testing.

4.1 Problem Description

The purpose of an LLM scheduler within an inference system, is to organise and configure a collection of GPUs for optimal performance. We can utilise previous *phase splitting* technology, to assign the prefill and decode phases of inference to optimal hardware, given the significant efficiency gains.

To create a schedule we must first subdivide a list of GPUs into *islands*, an island is a collection of homogenous GPUs of a given type and size. Presented with a list of islands, we must then assign either prefill or decode work (and configure appropriate parallelism

levels with respect to MoE) to these islands, maximising a defined objective. In this case we are optimising for cost efficiency, so throughput with respect to cost is a good metric.

The logical architecture of such inference system is displayed in Figure 4.1, where incoming requests are mapped to prefill islands. Then, after transferring the (KV) cache, completed prefill requests are sent to the optimum decode island. We are developing a new scheduler component for this system.

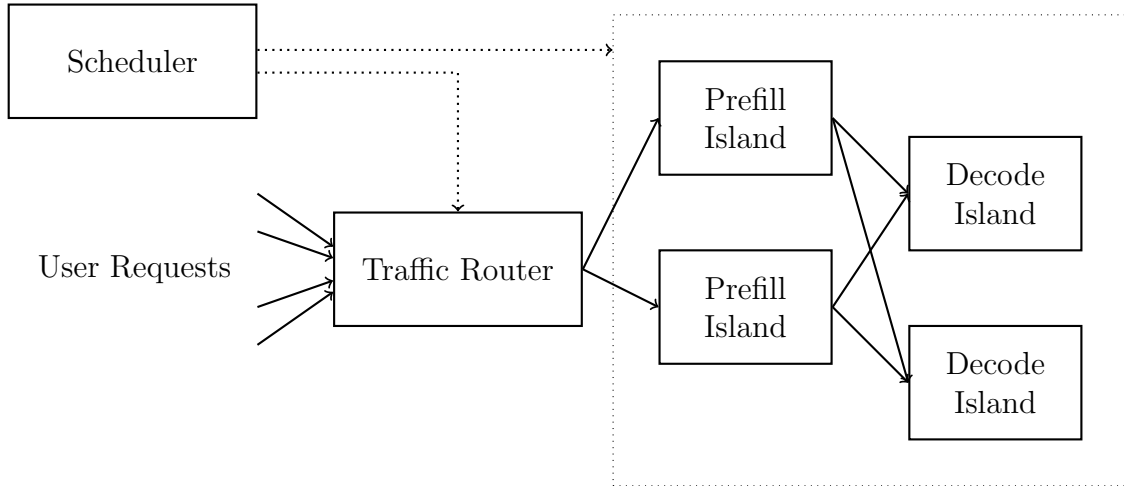


Figure 4.1: Sample service architecture of LLM inference engine, the scheduler component directs how traffic is distributed across the GPUs and configures the islands

4.2 Simulator

In order to be able to develop an LLM scheduler, we need access to many high-performance GPUs to test our schedule configurations. Unfortunately, GPU-hours are prohibitively expensive and using such hours for “dummy” workloads, to test a scheduler, would be considered a waste of resources. Instead, we utilise the latest simulator technology to estimate the performance of LLM inference given a set of input parameters. Several simulators already exist that attempt to model the performance of an LLM request by using estimation techniques.

Vidur [32] is an LLM simulator developed by Microsoft, designed to address the high cost and complexity of experimentally finding the best deployment configuration of LLMs. This is achieved by profiling a model-gpu combination for expected throughput on given operations (different prefill and decode requests). Then, using a small machine learning model, the system predicts the likely performance with high degrees of accuracy for different parallelism configurations.

Shallowsim [16] is a specialist simulator designed to estimate the compute times for different LLM request types. Specifically built for DeepSeek-V3/R1 models, it has native support for simulating MoE architectures. Unlike Vidur, Shallowsim requires no profiling phase, rather a list of device performance metrics is provided e.g. `fp8` performance.

These metrics are then used to compute the expected performance for the prefill and decode phase separately

We select Shallowsim for its transparency of calculations (simplicity), as it uses no machine learning to predict performance. Instead, simple mathematical operations are used, which allows us to inspect performance and modify the calculations as necessary. The Shallowsim simulator also already has native support for Mixture of Expert models, which is our key novel design goal for the scheduler.

4.2.1 Shallowsim Implementation

4.2.2 Modifications

We make several small modifications to the Shallowsim simulator to support our use-case of efficiently profiling system configurations. As-built Shallowsim is designed to profile several different parallelism configurations and batch sizes, then report back the overall fastest configuration. We need support for profiling a specific parallelism configuration and batch size, as we may not always be able to use the most optimal for a given exact sequence length, but rather a range of sequence lengths.

To do this, we extract all the mathematical operations present in the existing code and form procedures with specific input parameters, e.g., `prefill_a2a(gpu, seq_len, tp_num)`. Here, we precisely specify the gpu model, input sequence length, and tensor parallelism degree.

Given the modifications, shallowsim now supports the following functions:

- **Prefill Performance -> Milliseconds**
`prefill_time(gpu, seq_len, kv_cache_rate, tp_num, dp_num)`
- **Decode Performance -> Milliseconds**
`decode_time(gpu, tp_num, bs_num, seq_len, decode_len, moe_num, device_num)`
- **Maximum Batch Size -> Int**
`check_max_bs(gpu, tp_num, bs_num, seq_len, decode_len, moe_num, device_num)`

4.2.3 Usage

Prefill Phase

Using the Shallowsim simulator we can measure the time t (in milliseconds) to compute a single prefill request for a given set of parameters, and convert this to throughput R (in requests per second) via $R_{\text{prefill}} = \frac{1000}{t}$.

Decode Phase

Shallowsim has support for calculating the time in milliseconds for a single token generation step. As we would typically expect several tokens to be generated, we cannot directly compute requests per second using Shallowsim, instead we must perform additional computations. The decode time of a single token is affected by the input/current sequence length, therefore we must measure the single token compute time for every token to be generated in a request.

By measuring t_i the compute time (in milliseconds) for the i th token, the total decode time for a request generating N tokens is $t_{\text{decode}} = \sum_{i=1}^N t_i$. We can then convert this total time into throughput R_{decode} (in requests per second), using the same method as prefill. $R_{\text{decode}} = \frac{1000}{t_{\text{decode}}} = \frac{1000}{\sum_{i=1}^N t_i}$, where R_{decode} represents how many full decode requests per second the configuration can sustain.

4.3 Workload Modeling

We expect LLM inference services to encounter several different classes of workloads, depending on the user's use-case. These workloads have different characteristics of input sequence length (request message plus any system prompts), and also different decode lengths (numbers of characteristics generated). For example, one would expect code prompts and responses to have different input sequence and decode lengths than to that of conversation prompts. Hence, we must design a scheduler that is able to optimise over a specific workload class.

4.3.1 Input Sequence Length

To achieve fine-grained control of request scheduling, dependant on input sequence, we model the input request length as a probability distribution. Hence, for a chosen workload type, there is a specific probability of request arriving with a given sequence length.

Let $w \in W$ denote a workload type from the set of all workload types W . For each workload type w , we define a probability mass function $p(s \mid w)$, which represents the probability of a request having sequence length $s \in S$ under workload w . This distribution satisfies: $\sum_{s \in S} p(s \mid w) = 1$ for all $w \in W$.

Instead of only considering the distribution of sequence lengths during assignment, to model system throughput, we use them to find the most optimal pairing of sequence length to GPU island. Therefore, specific GPU islands can be configured to most optimally perform prefill operations for a specific sequence length. When a request arrives, we can direct it to the GPU islands targeted to serve that request length, defined in the schedule.

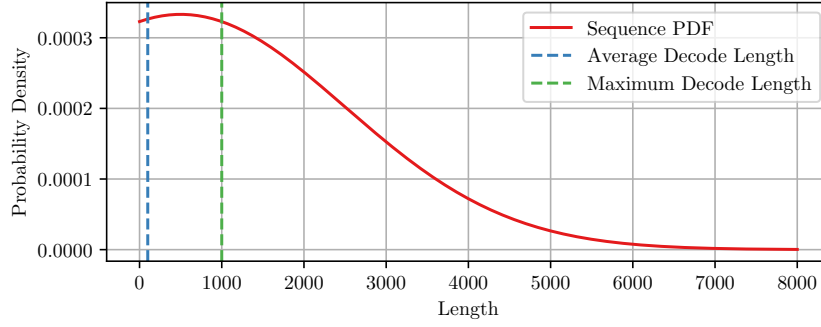


Figure 4.2: Sample probability distribution of input sequence length. Modelled using a pdf where $\mu = 500$, $\sigma = 2000$ and truncated/normalised to the range $0 \leq s \leq 8000$. Average decode length (100) and max decode length (1000) are extracted for use in the scheduler.

4.3.2 Decode Length

While decode length can also be modelled using a probability density function, this information (specific distribution) is less useful from a scheduling perspective. In contrast to input sequence length, the decode length is not known the moment a request arrives. Instead, the generation process proceeds token-by-token until an explicit or implicit stop condition is met (e.g. end-of-sequence token).

We can assume that a decode request must complete on the GPU island it started processing on, as switching island would incur additional communication overhead and significantly impact the request response time.

For some workloads there is a small correlation between input sequence length and decode length, but this relationship is often not significant. Therefore, at the point of request arrival we, do not have enough information about the decode length to make informed per-decode length scheduling decisions. Hence, creating specific GPU configurations for short, medium, long, etc. decode lengths would likely be inefficient. As we cannot accurately predict decode length, there is a high probability that a decode request would have to move island to finish the request.

Instead, we model our scheduler using an average decode length for a given workload type and design every GPU decode island to support the maximum decode length expected. Unlike the sequence length, the decode length is simply being used for system modelling activities to characterise performance, not direct traffic. However, we can still make scheduling decisions for the decode phase using the input sequence length as a feature, given the maximum decode length is affected by the input sequence length. This is due to all the prefill and decode tokens KV values being stored in GPU memory simultaneously.

4.4 Objectives

4.4.1 Throughput Modelling

Demonstrated in subsection 4.2.3, the Shallowsim simulator has support for estimating the throughput of a single request of a static input sequence and decode length. It is highly improbable that an entire workload consists of a single request type, instead we expect a distribution of sequence lengths, as shown in section 4.3. Hence, we need a method to accurately calculate the throughput of a system, with a variety of input sequence lengths.

Using the probability distribution of the workload, we can compute the weighted average of throughput, using measured performance values for each sequence length. Let $R_{\text{prefill}}(s)$ denote the measured prefill throughput (in requests per second) for a request of sequence length s . The expected prefill throughput for workload w , denoted $\mathbb{E}[R_{\text{prefill}} \mid w]$ (expectation), is therefore given by:

$$\mathbb{E}[R_{\text{prefill}} \mid w] = \sum_{s \in S} R_{\text{prefill}}(s) \cdot p(s \mid w)$$

In addition to prefill, we can also compute the expected throughput for the decode phase. While prefill throughput depends solely on the input sequence length s , decode throughput typically depends on both the input length s and the output (decode) length d . For simplicity, we assume a fixed average decode length d as a parameter. Let $R_{\text{decode}}(s, d)$ denote the measured decode throughput (in requests per second) for an input sequence length s and average decode length d . Given a workload-specific sequence length distribution $p(s \mid w)$, the expected decode throughput for workload w is:

$$\mathbb{E}[R_{\text{decode}} \mid w, d] = \sum_{s \in S} R_{\text{decode}}(s, d) \cdot p(s \mid w)$$

To account for the complete request processing pipeline, we define the overall expected throughput as the minimum of the expected prefill and decode throughputs. Since requests cannot be served faster than the slowest stage, the effective throughput is bounded by the bottleneck. The overall expected throughput is therefore given by:

$$\mathbb{E}[R_{\text{overall}} \mid w, d] = \min(\mathbb{E}[R_{\text{prefill}} \mid w], \mathbb{E}[R_{\text{decode}} \mid w, d]).$$

This value represents the effective throughput achievable by the system under the given workload and output length assumptions.

4.4.2 Throughput Allocation

Previous calculations assumed that a singular GPU island is contributing to the overall throughput of the inference system. In reality, we would like to be able to construct several GPU islands and assign each island to a request type it is best suited for. Hence, we allocate a percentage of GPU resources for a given island to a given sequence length it specialises in. Let I be the set of all GPU islands in the system, which we partition into two separate subsets I_{prefill} and I_{decode} . Let $x_i(s)$ denote the fraction of overall GPU capacity on island i devoted to handling requests of length s . We normalise these allocations so that $\sum_{s \in S} x_i(s) = 1$ for every $i \in I$. By including $x_i(s)$ into our previous per-length throughput models, the expected prefill throughput under workload w becomes

$$\mathbb{E}[R_{\text{prefill}} \mid w] = \sum_{i \in I_{\text{prefill}}} \sum_{s \in S} x_i(s) R_{\text{prefill}}^{(i)}(s) p(s \mid w),$$

while the expected decode throughput for average decode length d is

$$\mathbb{E}[R_{\text{decode}} \mid w, d] = \sum_{i \in I_{\text{decode}}} \sum_{s \in S} x_i(s) R_{\text{decode}}^{(i)}(s, d) p(s \mid w).$$

4.5 Constraint Optimisation

We can combine the previous throughput calculations to form a simple constraint optimisation problem with an objective to maximise the total expected throughput R and minimise the deviation Δ from the target workload distribution $p(s \mid w)$.

4.5.1 Constraint Formulation

The following statement describes a sample non-linear optimisation problem that can be solved for either prefill or decode throughput.

Input Data

- I (set of GPU islands),
- S (set of sequence lengths),
- $b_i(s)$ (benchmark throughput of island i on length s),
- $p(s \mid w)$ (target probability of length s under workload w), $\sum_{s \in S} p(s \mid w) = 1$,
- λ (trade-off parameter, $\lambda > 0$, balancing throughput vs. deviation).

Decision Variables

$$x_i(s) \geq 0 \quad (\text{fraction of island } i\text{'s GPU capacity devoted to length } s).$$

Constraints

$$\sum_{s \in S} x_i(s) = 1 \quad \forall i \in I.$$

Intermediate Quantities

- **Performance per cell:**

$$r_i(s) = b_i(s) x_i(s).$$

- **Total performance:**

$$R = \sum_{i \in I} \sum_{s \in S} r_i(s) = \sum_{i \in I} \sum_{s \in S} b_i(s) x_i(s).$$

- **Performance by sequence length:**

$$R_s = \sum_{i \in I} r_i(s) = \sum_{i \in I} b_i(s) x_i(s), \quad \forall s \in S.$$

- **Achieved share:**

$$\sigma_s = \frac{R_s}{R}.$$

- **Deviation from target:**

$$\delta_s = |\sigma_s - p(s \mid w)|, \quad \Delta = \sum_{s \in S} \delta_s.$$

Objective

Maximise throughput weighted down by deviation,

$$R - \lambda \Delta.$$

4.5.2 Challenges

The constraint optimisation problem as described is non-linear, this is due to the chained multiplications and divisions for the *performance per cell* and *achieved share* calculations. As a result, we require a non-linear programming (NLP) solver to obtain a solution for the proposed problem.

While the problem could be solved using such NLP techniques, this process would likely be computationally intensive and slow. This is primarily due to the non-convex and non-linear nature of the formulation, which makes it difficult to guarantee convergence to the global optimum solution. Ideally, we would like to linearise this problem to improve computational efficiency and enhance the reliability of finding the best possible solution.

Furthermore, we statically define I to a set of curated islands where the optimiser simply fits the workload to these islands. In order to generate the most optimal overall solution, we need to choose the best set of island sizes for each GPU type. However, adding this decision process to the constraint optimisation problem increases its complexity, as it effectively turns a bounded bin packing problem into an unbounded one. The optimiser must now determine not only the workload allocation but also the optimal sizes and number of islands. Hence, we need to formulate an architectural approach that incorporates this logic, which we detail in section 4.7.

4.6 Scheduling Assumptions

Before finalising our problem, we can make several assertions and assumptions in an attempt to reduce the search space of the scheduler generation process. These effectively apply heuristics to our problem, that are crafted to simplify the computation requirements while retaining solution quality.

4.6.1 Prefill Parallelism

We can assume for a given model and GPU type, there exists an optimal parallelism configuration that is consistent across input sequence lengths. Demonstrated in Figure 4.3 we can see that for all GPUs, $tp = 64$ and $dp = 1$ is the fastest parallelism configuration, for all sequence lengths above 128 tokens. Given we expect the majority of input requests to be over 128 tokens, we can reason that there is an *optimal* GPU prefill configuration for each GPU type.

By pre-computing this optimal configuration, we significantly reduce the number of parameters considered by the scheduler. This is because an island cannot be formed with several different parallelism configurations simultaneously. And hence, the scheduler does not have to contain a variable for (and consider) each sequence-length-parallelism combination.

To identify a best GPU prefill configuration across sequence lengths we use the following calculations, let G be the total number of GPUs and consider the set of all parallelism configurations $C = \{c = (tp, dp) \mid tp \cdot dp = G\}$. For each configuration $c \in C$ and sequence length $s \in S$, we measure the prefill throughput $R_{\text{prefill}}(s, c)$ and define the optimal throughput at length s by

$$R_{\text{prefill}}^*(s) = \max_{c \in C} R_{\text{prefill}}(s, c).$$

We then compute the relative deviation of configuration c at length s as

$$\delta(c, s) = \frac{R_{\text{prefill}}^*(s) - R_{\text{prefill}}(s, c)}{R_{\text{prefill}}^*(s)}.$$

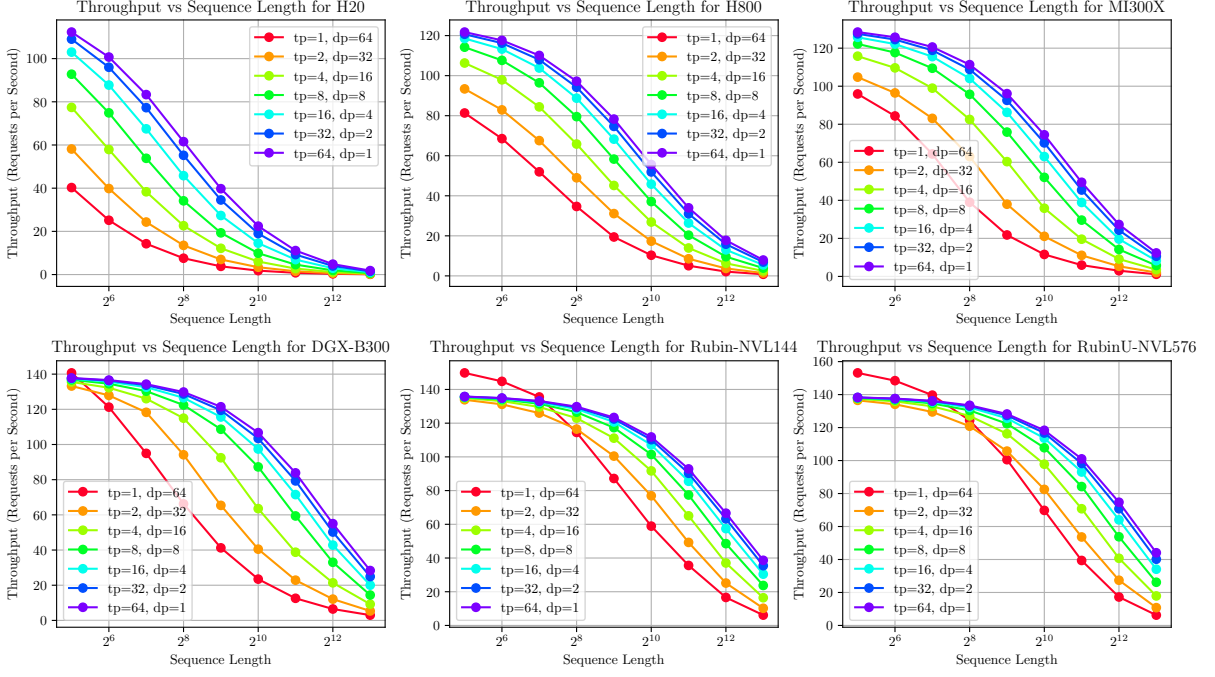


Figure 4.3: Selection of GPUs benchmarked over the prefill stage, with a variety of parallelism configurations using the *Shallowsim* simulator. All configurations use a static number of GPUs (64).

Aggregating these deviations over all lengths gives

$$\bar{\delta}(c) = \frac{1}{|S|} \sum_{s \in S} \delta(c, s), \quad (\text{uniform average over } S),$$

$$\tilde{\delta}(c) = \sum_{s \in S} \delta(c, s) p(s | w), \quad (\text{weighted by workload distribution}).$$

Finally, the best overall configuration is chosen as the one minimising the average deviation from the individual best configurations,

$$c_{\text{prefill}} = \arg \min_{c \in C} \tilde{\delta}(c).$$

4.6.2 Decode Parallelism

We can compute the same best configuration for the decode phase, with small modifications to the set of C configurations taking into account Mixture-of-Experts. Unlike prefill, dp (the data-parallelism factor) is not a native parameter of the *Shallowsim* simulator. Instead, we pre-compute how many GPUs to pass to the simulator and then scale the resultant throughput by dp . Let G be the total number of GPUs, tp tensor parallelism factor, and n the total number of routed experts in the model. We can form every pairing of valid factors using:

$$T = \{tp \in \{1, \dots, G\} \mid G \bmod tp = 0\}, \quad D = \{dp \in \{1, \dots, G\} \mid G \bmod dp = 0\},$$

$$C' = T \times D = \{ (tp, dp) \mid tp \in T, dp \in D \}.$$

For each $(tp, dp) \in C'$, set

$$G_{\text{sim}} = \frac{G}{dp}, \quad ep = \left\lceil \frac{n}{G_{\text{sim}}} \right\rceil,$$

and define the simulator configuration

$$c = (G_{\text{sim}}, tp, ep) \in C.$$

Then, for each $c \in C$ and sequence length $s \in S$, the simulator returns

$$R_{\text{prefill}}(s; c),$$

and under dp -way data parallelism the effective total prefill throughput is

$$R_{\text{prefill}}(s; dp, c) = dp \times R_{\text{prefill}}(s; c) = dp \times R_{\text{prefill}}\left(s; \frac{G}{dp}, tp, ep\right).$$

Before calculating the best parallelism configuration out of the set, we also need to decide which batch size to use. Since we would like to optimise our configuration for overall throughput, we need to maximise the batch size for each input sequence length, such that we remain within the memory constraints of the island. Let $\max_bs(s; c)$ denote the maximum batch size returned by the simulator for sequence length s under configuration c . We then round down to the nearest multiple of 8 and clamp between 8 and 1024 to keep the batch size aligned to the GPU memory and compute capabilities. This ensures efficient hardware utilisation and avoids potential performance degradation caused by unaligned memory access or suboptimal parallelism.

$$\text{bs}(s; c) = \min\left(\max(\max_bs(s; c) - (\max_bs(s; c) \bmod 8), 8), 1024\right).$$

Similar to prefill, we seek a single best decode parallelism configuration across sequence lengths. For each simulator configuration $c \in C$ and sequence length s , we first compute the batch size $\text{bs}(s; c)$ as before, then measure the decode throughput $R_{\text{decode}}(s, d; c, \text{bs}(s; c))$. We define the optimal decode throughput at length s by

$$R_{\text{decode}}^*(s, d) = \max_{c \in C} R_{\text{decode}}(s, d; c, \text{bs}(s; c)),$$

and the relative deviation of configuration c at s by

$$\delta_{\text{decode}}(c, s) = \frac{R_{\text{decode}}^*(s, d) - R_{\text{decode}}(s, d; c, \text{bs}(s; c))}{R_{\text{decode}}^*(s, d)}.$$

Aggregating these deviations over all lengths gives

$$\bar{\delta}_{\text{decode}}(c) = \frac{1}{|S|} \sum_{s \in S} \delta_{\text{decode}}(c, s), \quad (\text{uniform average over } S),$$

$$\tilde{\delta}_{\text{decode}}(c) = \sum_{s \in S} \delta_{\text{decode}}(c, s) p(s | w), \quad (\text{weighted by workload distribution}).$$

We can then select the decode configuration that minimises the average deviation across all expected sequence lengths,

$$c_{\text{decode}} = \arg \min_{c \in C} \tilde{\delta}_{\text{decode}}(c).$$

Shown in Figure 4.4, for each GPU there is generally a better parallelism configuration for most sequence lengths. Unlike prefill, this configuration is not consistent across GPU types, but rather significantly impacted by the GPU specifications. Configurations with zero throughput are due to the configuration exceeding the memory capabilities of that island.

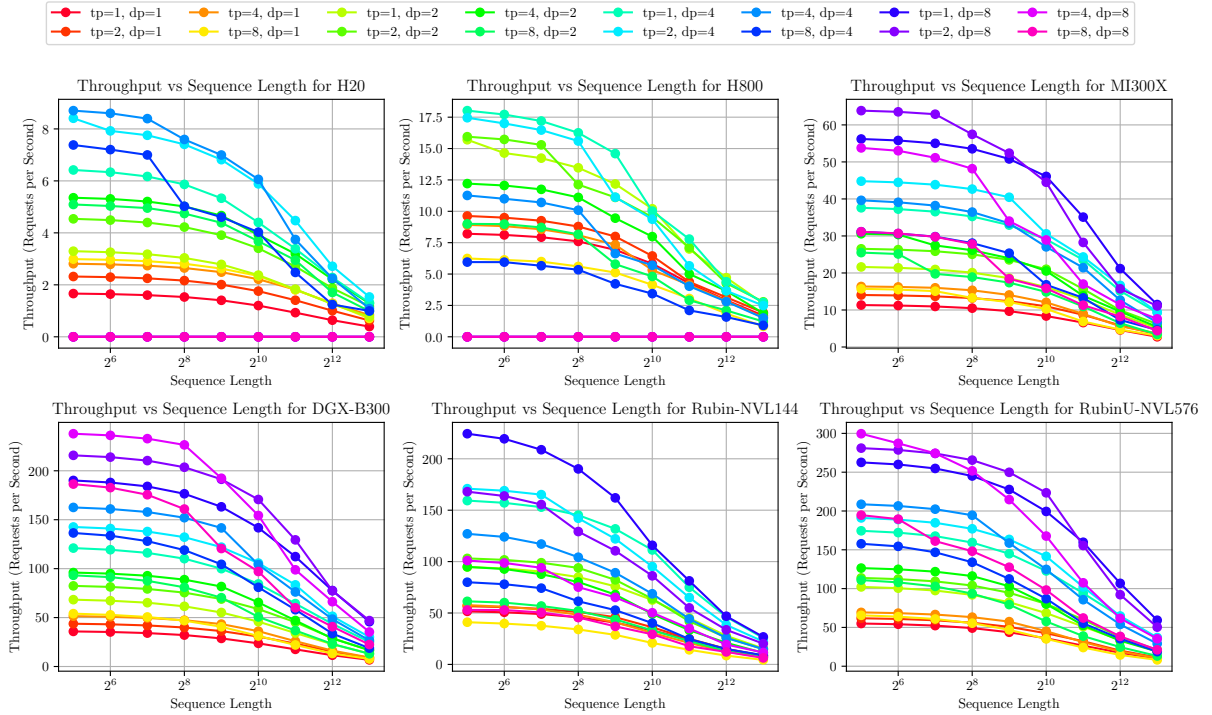


Figure 4.4: Selection of GPUs benchmarked over the decode stage, with a variety of parallelism configurations using the *Shallowsim* simulator. All configurations use a static number of GPUs (64).

4.6.3 Solver Resolution

We can use the simulator to benchmark the prefill and decode throughput of an island for a given input sequence length. While it is feasible to consider each sequence length ($s \in S$) individually (in our optimisation problem), we can use a binning approach to

reduce the search space of the scheduler. The width of the ranges effectively control the resolution of the solution generated, which can be dynamically adjusted depending on input parameters.

We partition the sorted list of lengths S into equally-sized intervals of width W , called *ranges*. Rather than measure every s_i , we sample a single representative length from each range at its midpoint. With a positive integer width, we compute the midpoint offset as $\text{mid} = \lfloor \frac{W}{2} \rfloor$. For each integer k such that $0 \leq k < \lfloor |S|/W \rfloor$, we let

$$\text{range_start}_k = k W, \quad \text{range_end}_k = k W + W, \quad \text{range_mid}_k = \text{range_start}_k + \text{mid},$$

and take range_mid_k as the representative length for the k -th range. We then also compute the total probability of that range by summing

$$p_k = \sum_{i=\text{range_start}_k}^{\text{range_end}_k-1} p(s_i | w),$$

and record for range k the tuple $(\text{range_mid}_k, p_k, \text{range_start}_k, \text{range_end}_k)$. Therefore, the set of all ranges is

$$\mathcal{R} = \{ (\text{range_mid}_k, p_k, \text{range_start}_k, \text{range_end}_k) \mid 0 \leq k < \lfloor |S|/W \rfloor \}.$$

By sampling only at these midpoints and aggregating probabilities, we approximate the full set of sequence lengths in each range with a single point (range_mid_k), thereby greatly reducing the number of simulator calls while preserving workload-weighted accuracy.

Prefill Throughput

With the formed ranges \mathcal{R} , each with their representative length s_r and total probability p_r , we can redefine the allocation variables $x_i(s)$ to act at the range level. Now island i assigns fraction $x_i(r)$ of its GPU capacity to range $r \in \mathcal{R}$, with

$$\sum_{r \in \mathcal{R}} x_i(r) = 1, \quad x_i(r) \geq 0 \quad \forall i \in I, r \in \mathcal{R}.$$

Within range r , every sequence length $s \in S_r$ is approximated by s_r , so that

$$\sum_{s \in S_r} R_{\text{prefill}}^{(i)}(s) p(s | w) \approx R_{\text{prefill}}^{(i)}(s_r) \sum_{s \in S_r} p(s | w) = R_{\text{prefill}}^{(i)}(s_r) p_r.$$

By including $x_i(r)$ into our per-range throughput models, the expected prefill throughput under workload w becomes

$$\mathbb{E}[R_{\text{prefill}} | w] = \sum_{i \in I_{\text{prefill}}} \sum_{r \in \mathcal{R}} x_i(r) R_{\text{prefill}}^{(i)}(s_r) p_r.$$

Decode Throughput

The same approach applies to the decode phase. For each range r with representative length s_r and probability p_r , we measure the decode throughput $R_{\text{decode}}^{(i)}(s_r, d)$ on island i for average decode length d . Island i allocates fraction $x_i(r)$ of its capacity to r , and within that range every sequence length is approximated by s_r . Thus,

$$\sum_{s \in S_r} R_{\text{decode}}^{(i)}(s, d) p(s | w) \approx R_{\text{decode}}^{(i)}(s_r, d) p_r.$$

Normalising $x_i(r)$ as before and summing over islands and ranges yields the expected decode throughput under workload w and decode length d :

$$\mathbb{E}[R_{\text{decode}} | w, d] = \sum_{i \in I_{\text{decode}}} \sum_{r \in \mathcal{R}} x_i(r) R_{\text{decode}}^{(i)}(s_r, d) p_r.$$

The overall expected throughput remains

$$\mathbb{E}[R_{\text{overall}} | w, d] = \min(\mathbb{E}[R_{\text{prefill}} | w], \mathbb{E}[R_{\text{decode}} | w, d]).$$

4.7 Problem Formulation

During section 4.5 we formulated a non-linear bounded bin-packing problem, by constraining the islands to a pre-defined type and size. In a true scheduling problem we expect to be allocated an inventory of different GPU types, not formed islands. Let T be the set of GPU types and for each $t \in T$, let I_t be the number of GPUs of type t allocated in the inventory. The full GPU inventory is then the set $\{I_t : t \in T\}$, with total number of GPUs in the system modelled as $N = \sum_{t \in T} I_t$.

Similar to ThunderServe [13], we can separate this scheduling problem as two separate, distinct stages. The first stage is the island layout generation, which we can treat as a search-like optimisation problem in which we explore combinations of GPU island sizes and types. The second stage is the bounded bin-packing step, a deterministic problem that assigns workload ranges to the pre-computed islands to maximise throughput. By decoupling layout design from the packing phase, we can leverage specialised solvers for each subproblem and significantly reduce overall computational complexity.

We refer to the bounded bin packing problem as the *inner loop*, and the outer optimisation process as the *outer loop*, as illustrated in Figure 4.5. The *outer loop* is responsible for generating promising island layouts, which are then passed to the *inner loop* for resource packing. The resulting throughput is then returned to the *outer loop*, allowing the optimisation process to evaluate and refine its search for good configurations. This iterative cycle can be repeated multiple times to progressively generate an optimum solution.

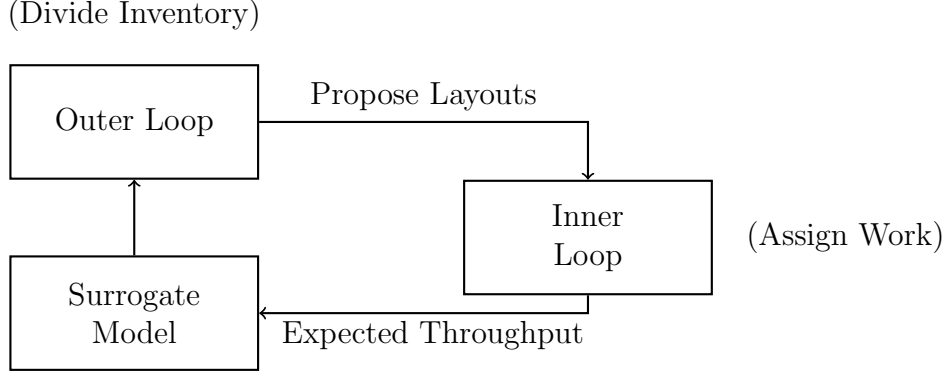


Figure 4.5: Problem formulation demonstrating the data flow between the *inner* and *outer* loop operations.

4.8 Inner Loop

The *inner loop* receives an island layout proposal from the *outer loop*, this layout is specified as a finite index set $\mathcal{I} = \{i = 1, \dots, |\mathcal{I}|\}$, where each island i is represented by a pair (t_i, s_i) . Here, $t_i \in T$ denotes the GPU type assigned to island i , and $s_i \in \mathbb{Z}_{\geq 0}$ indicates the number of GPUs of type t_i allocated to that island. The *outer loop* guarantees that these allocations respect the total-inventory constraints.

Given the predefined island layout, the *inner loop's* only task is to assign workloads of different prefill and decode lengths. No further modification of t_i or s_i takes place at this stage, as the inner loop focuses entirely on determining the optimal bin-packing configuration.

We can outline the steps of the *inner loop* as follows:

1. **Formulate Ranges.** Take the specific workload distribution w , and separate into a series of ranges \mathcal{R} using a specified resolution (subsection 4.6.3)
2. **Prefill Configuration Generation & Benchmark.** Identify the best prefill configuration (c_{prefill}) for each island i , and return the benchmark results describing the throughput for each range (subsection 4.6.1).
3. **Decode Config Generation & Benchmark.** Identify the best decode configuration (c_{decode}) for each island i , and return the benchmark results describing the throughput for each range (subsection 4.6.2).
4. **Problem Specification.** Formulate the exact optimisation problem based on the provided prefill and decode benchmarks (subsection 4.8.1).
5. **Solve.** Invoke the optimiser on the specified problem, to generate the best solution given the input parameters.
6. **Results.** Extract the final workload-assignment solution from the solver's output, including any intermediary variables for debugging.

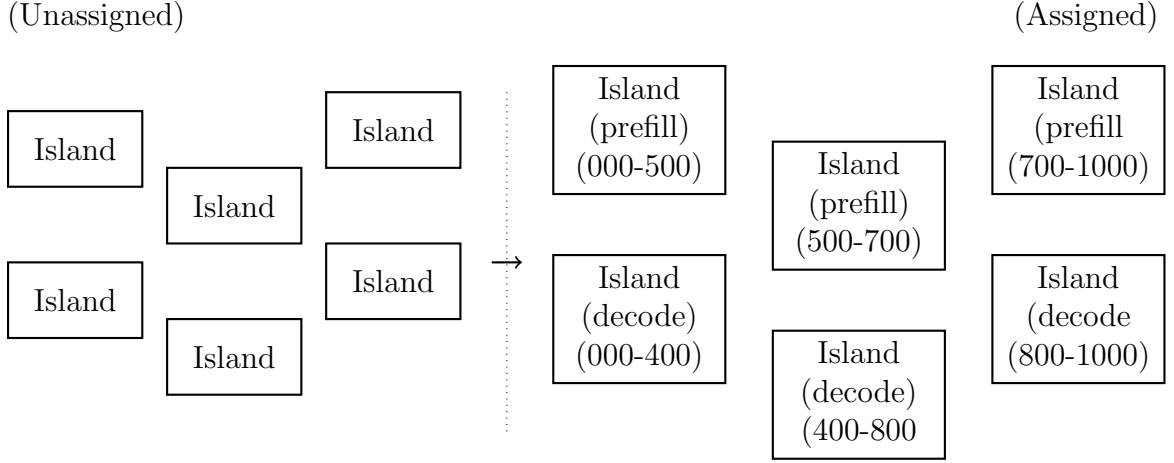


Figure 4.6: Diagram showing effective purpose of *inner loop*, prefill and decode workload is assigned to islands.

4.8.1 Constraint Formulation

We take the non-linear problem formulation from section 4.5, which contained ratios such as $\sigma_s = R_s/R$, and transform it into an entirely linear program by introducing auxiliary variables and linear constraints. We then incorporate the “range” abstraction from subsection 4.6.3 so that instead of summing over every sequence length, we only sum over a small set of ranges J . We also modify the problem to contain assignment variables (x values) for both prefill and decode. The result is a single, linearised load-balancing problem that jointly optimises prefill and decode throughput.

Linearisation Steps

In the original non-linear problem, each range’s achieved share σ_j involved dividing by the total throughput R , ($\sigma_j = R_j/R$). We avoid division by introducing one auxiliary variable per range and enforcing deviations through linear inequalities:

$$R_j - p_j R \leq \delta_j, \quad p_j R - R_j \leq \delta_j, \quad \delta_j \leq \tau p_j R,$$

for $R_j = R_j^{\text{prefill}}$ or R_j^{decode} and $R = R_{\text{prefill}}$ or R_{decode} .

Whenever an absolute deviation $|R_j - p_j R|$ appears in the original objective, we replace it by a nonnegative slack variable δ_j and two linear inequalities, ensuring $\delta_j \geq |R_j - p_j R|$. By defining R_j^{prefill} , R_j^{decode} , R_{prefill} , and R_{decode} as linear combinations of the x -variables, all throughput expressions become linear functions of the decision variables. Finally, to prevent zero-throughput solutions, we add linear lower-bounds $R_{\text{prefill}} \geq \varepsilon$, $R_{\text{decode}} \geq \varepsilon$, where $\varepsilon = 0.01$ is a small positive constant.

Combining Prefill and Decode

Because we must serve both prefill and decode for each range j , we include separate variables $x_{\text{prefill},ij}/x_{\text{decode},ij}$ and corresponding per-range throughput variables $R_j^{\text{prefill}}/R_j^{\text{decode}}$. We also add a δ_{match} variable to encourage equal balancing of throughput across both prefill and decode which is enforced by another pair of linear inequalities per range. so that $\delta_{\text{match},j} \geq |R_j^{\text{decode}} - R_j^{\text{prefill}}|$. By penalising these matching deviations in the objective, we ensure a balance between prefill and decode stages.

We also ensure each island is either performing prefill or decode by introducing a binary variable $z_i \in \{0, 1\}$ per island. Along with the constraints: $x_{\text{prefill},ij} \leq 1 - z_i$, $x_{\text{decode},ij} \leq z_i$, $\forall i \in I$, $j \in J$, so that if $z_i = 1$, the island's $x_{\text{prefill},ij}$ are forced to zero (decode-only), and if $z_i = 0$, the island's $x_{\text{decode},ij}$ are zero (prefill-only).

Input Data

I	(set of GPU islands),
J	(set of ranges, indexed by j),
b_{ij}^{prefill}	(measured prefill throughput of island i on range j),
b_{ij}^{decode}	(measured decode throughput of island i on range j),
p_j	(target probability of range j under workload w), $\sum_{j \in J} p_j = 1$,
τ	(maximum allowable per-range deviation factor),
M	(large constant for scaling deviation weights),
$\lambda_{\text{dev},j}$	$= \frac{M}{1 + p_j}$ (trade-off weight for deviations in range j).
ε	$= 0.01$ (minimum allowable throughput).

Decision Variables

$$\begin{aligned}
x_{\text{prefill},ij} &\geq 0, \quad \forall i \in I, j \in J, \quad (\text{fraction of island } i\text{'s capacity to prefill range } j); \\
x_{\text{decode},ij} &\geq 0, \quad \forall i \in I, j \in J, \quad (\text{fraction of island } i\text{'s capacity to decode range } j); \\
z_i &\in \{0, 1\}, \quad \forall i \in I, \quad (z_i = 1 \text{ if island } i \text{ is assigned to decode, else prefill}).
\end{aligned}$$

Intermediate Quantities

- **Per-range throughputs:**

$$R_j^{\text{prefill}} = \sum_{i \in I} b_{ij}^{\text{prefill}} x_{\text{prefill},ij}, \quad R_j^{\text{decode}} = \sum_{i \in I} b_{ij}^{\text{decode}} x_{\text{decode},ij}, \quad \forall j \in J.$$

- **Total throughputs:**

$$R_{\text{prefill}} = \sum_{j \in J} R_j^{\text{prefill}}, \quad R_{\text{decode}} = \sum_{j \in J} R_j^{\text{decode}}.$$

- **Per-range deviations:**

$$\delta_{\text{prefill},j} \geq 0, \quad \delta_{\text{decode},j} \geq 0, \quad \delta_{\text{match},j} \geq 0, \quad \forall j \in J.$$

- **Total deviation:**

$$D_{\text{total}} = \sum_{j \in J} (\delta_{\text{decode},j} + \delta_{\text{prefill},j} + \delta_{\text{match},j}).$$

Constraints

1. **Each island allocates exactly one unit of capacity across prefill/decode and ranges:**

$$\sum_{j \in J} (x_{\text{prefill},ij} + x_{\text{decode},ij}) = 1 \quad \forall i \in I.$$

2. **Exclusive prefill/decode assignment per island:**

$$x_{\text{prefill},ij} \leq 1 - z_i, \quad x_{\text{decode},ij} \leq z_i, \quad \forall i \in I, j \in J.$$

3. **Enforce a minimum performance value:**

$$R_{\text{prefill}} \geq \varepsilon, \quad R_{\text{decode}} \geq \varepsilon.$$

4. **Deviation constraints for decode (per range j):**

$$\begin{aligned} R_j^{\text{decode}} - p_j R_{\text{decode}} &\leq \delta_{\text{decode},j}, \\ p_j R_{\text{decode}} - R_j^{\text{decode}} &\leq \delta_{\text{decode},j}, \quad \forall j \in J. \\ \delta_{\text{decode},j} &\leq \tau p_j R_{\text{decode}}, \end{aligned}$$

5. **Deviation constraints for prefill (per range j):**

$$\begin{aligned} R_j^{\text{prefill}} - p_j R_{\text{prefill}} &\leq \delta_{\text{prefill},j}, \\ p_j R_{\text{prefill}} - R_j^{\text{prefill}} &\leq \delta_{\text{prefill},j}, \quad \forall j \in J. \\ \delta_{\text{prefill},j} &\leq \tau p_j R_{\text{prefill}}, \end{aligned}$$

6. Match throughput between prefill and decode (per range j):

$$\begin{aligned} R_j^{\text{decode}} - R_j^{\text{prefill}} &\leq \delta_{\text{match},j}, \\ R_j^{\text{prefill}} - R_j^{\text{decode}} &\leq \delta_{\text{match},j}, \end{aligned} \quad \forall j \in J.$$

7. Total deviation across all ranges:

$$\sum_{j \in J} (\delta_{\text{decode},j} + \delta_{\text{prefill},j} + \delta_{\text{match},j}) = D_{\text{total}}.$$

Objective

$$\max \left(R_{\text{decode}} + R_{\text{prefill}} - \sum_{j \in J} \lambda_{\text{dev},j} (\delta_{\text{decode},j} + \delta_{\text{prefill},j} + \delta_{\text{match},j}) \right).$$

4.8.2 Implementation

We implement this constraint optimisation problem using **pulp**, a linear and mixed integer programming modeller written in Python. Given our problem uses both integer (z assignment variables) and float (x load variables), we need to use a Mixed-Integer Linear Programming Solver (MILP). The mixed-integer nature worsens the solve performance but allows for the dynamic selection of which islands are provisioned for prefill and decode.

4.8.3 Optimisation

Given the *inner loop* will be executed several times by the *outer loop* during optimisation, we can improve performance through caching various results.

First, we can cache the **Formulate Ranges** step by recording the computed range values in a dictionary with the bin width W . For each execution of *inner loop*, these values will be constant so caching prevents unnecessary re-computation.

Second we can cache the **Prefill Configuration Generation & Benchmark** process for islands of the same type and size. For a constant workload, we can expect the prefill configuration and benchmark output to be constant for the same island size and type. The proposed island size-type combination will likely repeat several hundred times in the optimisation process, saving significant compute time.

Third, we can complete the same caching for the **Decode Configuration Generation & Benchmark** step. This step is executed during every iteration, like prefill, so we can expect equivalent performance gains.

4.9 Outer Solver

The *outer loop* receives an input inventory, containing a list of GPU types and quantities, represented with the set $\{I_t : t \in T\}$. Given the inventory, the *outer loop's* job is to propose new island layouts (\mathcal{I}), which can be subsequently evaluated by the *inner loop*.

The *inner loop* execution is somewhat expensive, hence a complete grid search would incur significant compute time. Instead we use a refined Bayesian Optimisation (BO) search method with two different methods for searching, *direct* (subsection 4.9.1) and *divided* (subsection 4.9.2).

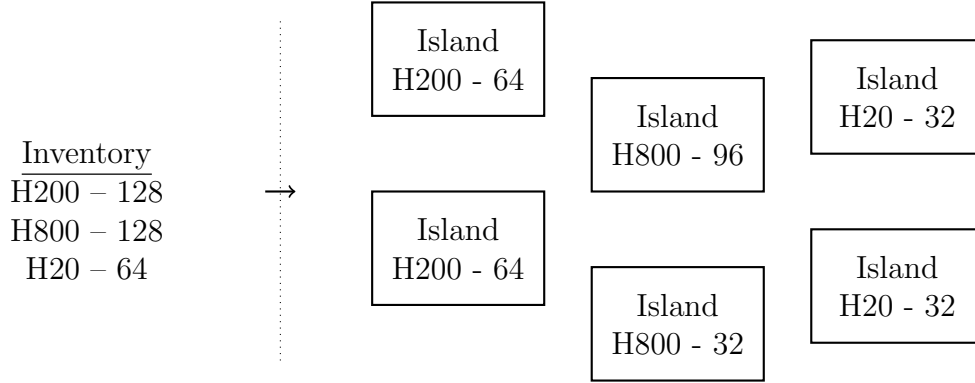


Figure 4.7: Diagram showing effective purpose of *outer loop*, GPU inventory is divided into groups of islands.

We select Bayesian Optimisation specifically due to its ability to efficiently explore high-dimensional, expensive-to-evaluate spaces with only a few hundred evaluations, rather than the exponential cost of a full grid search. By maintaining a surrogate model of the island-layout performance (gaussian process over layouts), Bayesian Optimisation balances exploration (sampling layouts in poorly understood regions) with exploitation (refining around the current best layouts).

Unlike reinforcement learning, which often requires thousands of samples to learn a policy, BO is designed for “one-shot” black-box optimisation. With the correct parameters it can deliver near-optimal solutions with far fewer simulator calls, minimal hyper-parameter tuning, and strong convergence. This applies to our-use case where the *inner loop* and constraint optimisation solver is effectively black-box.

4.9.1 Direct Island Generation

In the *direct* island generation approach, islands are explicitly defined within the Bayesian Optimisation process. Configurations correspond to a fixed number of islands, with each candidate solution representing a specific allocation strategy. The number of GPUs in each island is directly part of the output space, where specific variables represent the GPU count per island ‘slot’. This setup provides precise control over how GPUs are grouped,

but it also requires well-defined constraints to ensure that all generated configurations are valid.

Bayesian Search Parameters

- T : the set of GPU types, e.g., $T = \{\text{A100}, \text{V100}, \dots\}$
- K : the number of GPU slots per type
- $x_{t,k}$: the integer decision variable representing the number of GPUs of type $t \in T$ assigned to slot $k \in \{1, \dots, K\}$
- I_t : the total inventory of GPU type $t \in T$

Bayesian Search Constraints

For each GPU type $t \in T$, we enforce that the total number of GPUs allocated across all K slots does not exceed the inventory I_t :

$$\sum_{k=1}^K x_{t,k} \leq I_t, \quad \forall t \in T$$

Implementation

The implementation of this approach uses the **Ax** [33] library with **BoTorch** [34] to perform the Bayesian Optimisation. The search space is defined programmatically such that for each GPU type, a fixed number of K slots are created. Each slot is a **RangeParameter** varying from 0 to the total inventory capacity for that GPU type. A **SumConstraint** is then applied for each GPU type, to ensure that the total number of allocated GPUs across all its slots does not exceed the available inventory.

The optimisation process begins with a warm-start phase, where a small number of valid randomly generated layouts are evaluated. Following this initialisation, the main optimisation loop begins.

This problem is high-dimensional ($|T| \times K$), which can pose a challenge for standard Bayesian optimisation. To address this, we use SAASBO (Sparse Axis-Aligned Subspace Bayesian Optimization) [26], specifically the **SaasFullyBayesianSingleTaskGP** model from **BoTorch**. SAASBO is effective in high-dimensional spaces because it identifies and focuses on a low-dimensional active subspace of the most influential parameters, thus improving sample efficiency.

In each iteration of the loop, the SAASBO surrogate model is fitted to all the data collected so far. The model then generates a new batch of candidate layouts, which are subsequently evaluated by the *inner loop* solver. The results are used to update the

model’s understanding of the search space, and the process repeats for a fixed number of iterations.

We configure the solver to minimise the `rho_max` value, which is calculated as the inverse of the system’s overall throughput. Where throughput (as defined in subsection 4.4.1) is the minimum value of R_{prefill} and R_{decode} . The Bayesian Optimisation process is effectively maximising the system’s overall performance.

However, with this direct generation approach, finding a good solution can be challenging. This is because even with SAASBO the search space can be vast. Additionally, in some scenarios many parameters for the slots may be optimally zero (singular large island), creating a sparse problem that can be difficult for the optimiser to propose good layouts. Also, keeping the total number of GPUs used across the slots below the total inventory count can also be difficult. Without creating a complex custom acquisition function, BoTorch has to reject proposed solutions outside the constraints. This causes many proposed solutions to be pruned, which is very inefficient. To combat these challenges, we propose the *divider* island generation approach.

4.9.2 Divider Island Generation

The *divider* island generation method uses a more abstract representation that substantially reduces the dimensionality of the BO problem. Explicit allocation at the slot level requires one parameter per slot per GPU type, resulting in a parameter space of size $|T| \times K$. This approach becomes quickly computationally expensive due to the high number of search parameters and the need for constraints to ensure valid configurations.

To address this, the divider-based approach reduces the search space to just two parameters per GPU type: the total number of islands and a vector value that controls the balance of GPU allocation across those islands. This formulation results in a significantly smaller parameter space of size $|T| \times 2$, enabling more efficient optimisation and faster convergence.

An advantage of this method is that it avoids the need for explicit resource constraints, which are otherwise necessary to enforce inventory limits and prevent invalid configurations being proposed. By shifting from low-level allocation to higher-level distribution strategies, divider island generation maintains control over performance-influencing characteristics while offering a more scalable and interpretable optimisation process.

Input Variables

Let G_{total} denote the total number of GPUs available for that type, and let N_{target} represent the desired number of islands to create. We can allow the actual number of islands may be lower if there are not enough GPUs to satisfy the minimum-size requirement. The parameter S_{skew} controls how any leftover GPUs (after each island has been assigned the

minimum) are distributed, shown in Figure 4.8: when $S_{\text{skew}} \approx 0.0$, the extra GPUs are spread as evenly as possible; if $S_{\text{skew}} > 0.0$, islands with higher indices receive proportionally more of the remainder; and if $S_{\text{skew}} < 0.0$, islands with lower indices receive more. Finally, I_{min} specifies the minimum number of GPUs that any island must contain.

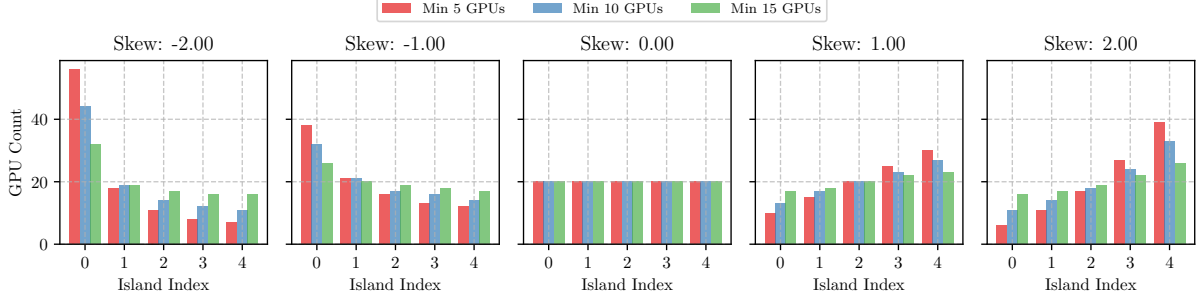


Figure 4.8: Graph displaying the effect of changing the S_{skew} value where $G_{\text{total}} = 100$ and $N_{\text{target}} = 5$. Instances with high skew values (absolute), display higher island size variance.

Preconditions

Initially, we verify that the input parameters meet a series of constraints, if any of these conditions is not met, the procedure raises an error and terminates immediately. We require $I_{\text{min}} > 0$, G_{total} and N_{target} must be nonnegative. Next, we perform trivial checks to assess whether it is possible to create at least one island given the input parameters. If $N_{\text{target}} = 0$, $G_{\text{total}} = 0$ or $G_{\text{total}} < I_{\text{min}}$, then no islands can be created and the function returns the empty set \emptyset . Similarly, if , there are insufficient GPUs to form even a single island of size I_{min} , the function again returns \emptyset .

Number of Islands

To establish how many islands can feasibly be created given the available GPUs and the minimum-size requirement, we first compute $N_{\text{max_possible}} = \lfloor G_{\text{total}} / I_{\text{min}} \rfloor$, which represents the maximum number of islands if each island is allocated exactly I_{min} GPUs. We then set $N_{\text{actual}} = \min(N_{\text{target}}, N_{\text{max_possible}})$, so that the actual number of islands cannot exceed either the user's target or the physical limit from the GPU pool. If $N_{\text{actual}} = 0$, this indicates that even a single island of size I_{min} cannot be formed (either because $N_{\text{target}} = 0$ or because $G_{\text{total}} < I_{\text{min}}$), and the function returns the empty set \emptyset . Otherwise, N_{actual} islands will be created,

Base Allocation

In the base allocation step, we allocate I_{min} GPUs to each of the N_{actual} islands, so that the total GPUs used for this initial distribution is $G_{\text{base_used}} = N_{\text{actual}} \cdot I_{\text{min}}$. The number of GPUs remaining to be distributed is then $G_{\text{remaining}} = G_{\text{total}} - G_{\text{base_used}}$. At this point,

every island has been provisioned with its minimum allocation of I_{\min} GPUs, and the remaining $G_{\text{remaining}}$ GPUs will be allocated in subsequent steps.

Island Distribution

In the case where $G_{\text{remaining}} = 0$, all GPUs have already been assigned to the N_{actual} islands at their minimum size, so we make no weighted adjustments. When $N_{\text{actual}} = 1$, there is only a single island so we add all $G_{\text{remaining}}$ GPUs to that island. Otherwise we can distribute the remaining GPUs using the S_{skew} value.

We first compute a weight w_k for each island k , where k ranges from 1 to N_{actual} , computed as $w_k = k^{S_{\text{skew}}}$. However, if S_{skew} is numerically close to zero, each island is given an equal weight of $w_k = 1.0$. These weights are then normalised so that we can derive proportions p_k , indicating each island's share of the remaining GPUs. Each proportion is determined by dividing the weight of an island by the weight sum:

$$W_{\text{sum}} = \sum_{j=1}^{N_{\text{actual}}} w_j, \quad p_k = \frac{w_k}{W_{\text{sum}}}.$$

Using these proportions, we compute the ideal (possibly fractional) allocation s_k^{ideal} for each island: $s_k^{\text{ideal}} = p_k \cdot G_{\text{remaining}}$.

Integer Adjustment

Because GPUs can only be allocated in integer units, we use the largest remainder method to convert the fractional allocations into integers. First, each island k is provisionally assigned a base number of GPUs: $s_k^{\text{base.int}} = \lfloor s_k^{\text{ideal}} \rfloor$. We need to compute the discrepancy between the total allocated GPUs and the intended $G_{\text{remaining}}$. This discrepancy,

$$D = \text{round} \left(G_{\text{remaining}} - \sum_{j=1}^{N_{\text{actual}}} s_j^{\text{base.int}} \right),$$

represents the number of GPUs that are still unassigned due to rounding down the fractional values. To distribute the D leftover GPUs, we calculate the fractional part for each island as $f_k = s_k^{\text{ideal}} - s_k^{\text{base.int}}$, and identify the D islands with the largest fractional parts. Each of these selected islands receives one additional GPU.

At the end of this process, each island has a final size given by $S_k = I_{\min} + s_k^{\text{base.int}} + \delta_k$, where $\delta_k \in \{0, 1\}$ depending on whether the island received an extra GPU during the discrepancy distribution. The total sum of all final island sizes equals G_{total} , satisfying the constraint that all available GPUs are used.

Bayesian Search Parameters

- T : the set of GPU types, e.g., $T = \{\text{A100}, \text{V100}, \dots\}$
- n_t : the target number of islands for GPU type $t \in T$
- s_t : a vector value representing the skew or balance in GPU allocation across the n_t islands for GPU type t
- I_t : the total available inventory of GPU type t

These two parameters (n_t and s_t) define the entire resource distribution strategy, allowing the Bayesian Optimisation process to search over high-level allocation patterns while implicitly satisfying inventory constraints by design.

Implementation

Like the direct approach, this method is implemented using the **Ax** [33] Python library with **BoTorch** [34]. However, the search space is constructed in a significantly simpler way. For each GPU type, only two parameters are created: n_t as an integer **RangeParameter**, and s_t as a float **RangeParameter**.

During evaluation, these high-level parameters are passed to the previously described **divider** function, which generates a final list of integer island sizes. This resultant layout is then passed to the same *inner loop* solver to calculate the objective, **rho_max**.

A key advantage of this method’s low-dimensional search space (two parameters per GPU type) is that a standard Gaussian Process (GP) model is sufficient for optimisation. Hence, this implementation uses the BoTorch **SingleTaskGP** as the surrogate model. We can expect this method to be significantly faster to execute, as compared to the *direct* approach, whilst also yielding better results.

Chapter 5

Evaluation

The purpose of this chapter is to analyse the performance of the proposed LLM scheduling algorithm. We specifically investigate, for a given GPU budget, whether a heterogenous cluster is faster (higher throughput) than a homogenous cluster. The goal is to demonstrate that heterogenous clusters are still more cost efficient than homogenous ones in a MoE environment.

We also investigate the performance of the two different scheduling levels separately, to try understand if both are performing effectively. For the *inner loop* we execute several test cases with pre-defined island sizes on a constant workload, here we evaluate the effectiveness of the linear optimisation function. We perform several more test cases on the *outer loop*, investigating how different search parameters affect the scheduling result.

Finally we execute several real-world tests on a variety of production workload traces on several different inventories.

5.1 System Performance

To answer the core research question, “*do heterogenous clusters outperform homogeneous clusters in MoE environments?*”, we need to perform several experiments. Given a constant hourly GPU budget, we craft three different inventories for testing and execute the complete schedule algorithm.

Given the Bayesian Optimisation process is non-deterministic (due to the random initialisation), the scheduling algorithm is executed five times over with the same parameters. Then, we take the average of the best performances from each trial and also extract the deviation score of the inner loop from these best trials. This allows us to demonstrate how well-matched the solutions is to the targeted workload. We execute these experiments using the Azure LLM [35] datasets for both code and conversation, which are later explored in section 5.5.

The key results are presented in Table 5.1, where the heterogenous cluster (a) is able to perform up to $1.4\times$ faster than the homogenous cluster (c) for the same price (code), $1.2\times$ (conversation). We use the conservative pricing estimate that the H20 is 1\$/hour, H800 is 2\$/hour, H200 is 4\$/hour and model our overall GPU budget is 512 \$/hour. Shallowsim is specifically designed for DeepSeek, which is currently profiled for either china-specific or unreleased GPUs, therefore it is challenging to extract pricing information for all GPUs.

Type	GPU Inventory			Code (Azure LLM)		Conversation (Azure LLM)	
	H200	H800	H20	R_{total}	D_{total}	R_{total}	D_{total}
(a) Hetero	64	64	128	176 ± 20	18 ± 18	138 ± 16	5 ± 9
(b) Hetero	32	128	128	199 ± 21	$22pm15$	126 ± 12	45.5 ± 46
(c) Homo	128	0	0	139 ± 46	25 ± 22	114 ± 5	7 ± 10

Table 5.1: GPU inventory configurations with resultant performance metrics on the Azure LLM dataset for code and conversation. **Experiment Parameters:** batch size = 16, iterations = 15, trials = 5, min island size = 2, skew range = ± 5.0 .

5.2 Inner Loop

We test the *inner loop* using four test cases (island layouts) shown in Table 5.2, this is to confirm that the *inner loop* is able to precisely map workloads to a variety of island scenarios. One scenario from both homogeneous and heterogeneous uses GPU islands of the same size, the other uses GPU islands of different sizes. These islands are crafted to show the importance of selecting appropriate island sizes and potential value of heterogeneous clusters.

Island	GPU Model	Sizes (Size \times Count)
(a) Homogeneous Same Size	DGX-B300	32×8
(b) Homogeneous Different Sizes	DGX-B300	$32 \times 2, 16 \times 4, 8 \times 8, 4 \times 16$
(c) Heterogeneous Same Size	DGX-B300	32×2
	Rubin-NVL144	32×2
	H800	32×2
	H20	32×2
(d) Heterogeneous Different Sizes	DGX-B300	$32 \times 1, 16 \times 1, 8 \times 1, 4 \times 2$
	Rubin-NVL144	$32 \times 1, 16 \times 1, 8 \times 1, 4 \times 2$
	H200	$8 \times 4, 4 \times 8$
	H20	$8 \times 4, 4 \times 8$

Table 5.2: GPU island test cases for *inner loop* analysis, with two homogeneous scenarios and two heterogeneous scenarios.

As the *inner loop* solution generation is deterministic we only execute each test case once, unless stated otherwise. This is because the linear solver will produce the exact same solution upon every execution, given the same input variables.

5.2.1 Solver Results

We first simultaneously select the optimum parallelism configuration and benchmark the provided GPU islands for both prefill and decode. The resultant throughput values are displayed in Figure 5.1, where we can see that every island class is able to contribute to prefill throughput, but not all islands are capable of decode. This is due to the decode phase requiring more memory to execute, hence not all islands are compatible.

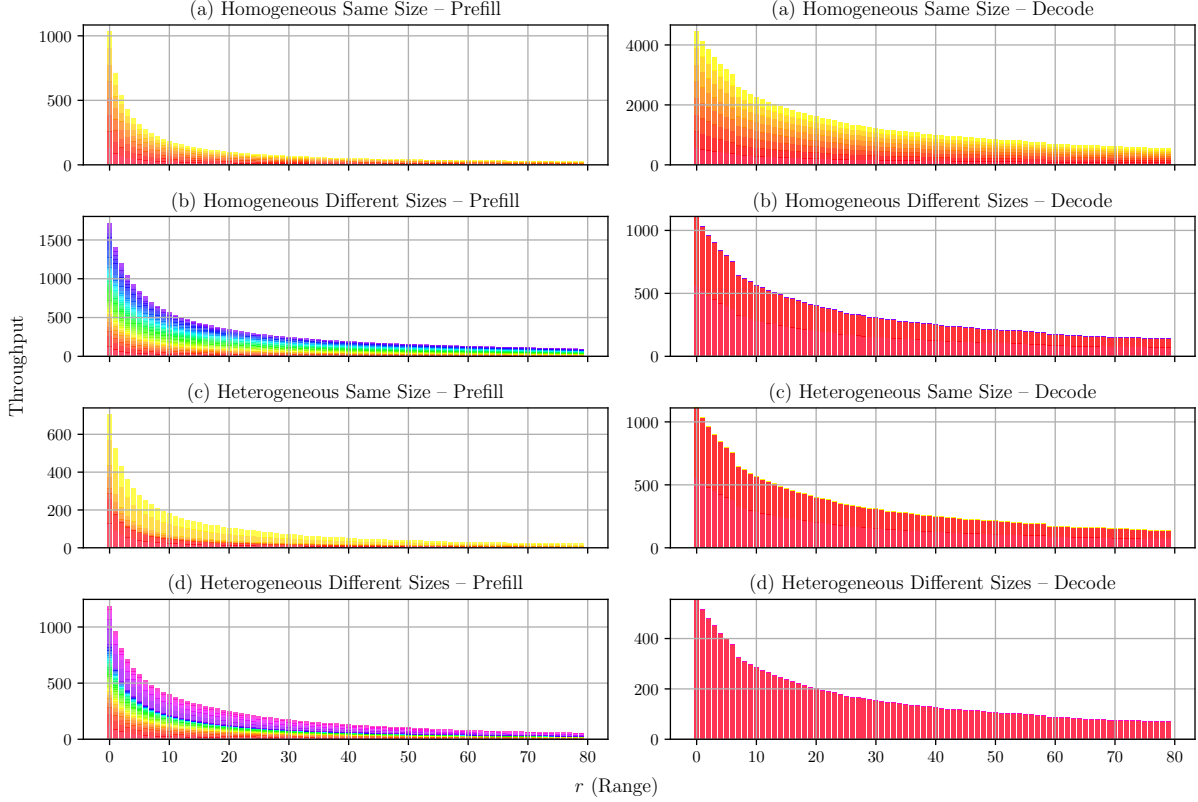


Figure 5.1: Prefill and decode benchmark results for test case, each colour represents a different island in the configuration.

Using the collected benchmark values, we populate the b_{ij}^{prefill} and b_{ij}^{decode} variables in the linear solver. Then, using artificial workload as previously shown in Figure 4.2, we execute the solve function to find the optimum $x_{\text{prefill}, ij}$ and $x_{\text{decode}, ij}$ values. The resultant assignment percentages are displayed in Figure 5.2 for both the prefill and decode phase.

In the homogenous scenario, the assignment allocation percentages closely resemble the target workload. In contrast, heterogeneous assignment allocation percentages appear random. This is to be expected, as it reflects the heterogeneous nature of island performance, where one island might be allocated few ranges due to its “slow” performance. Also, for all scenarios, one large island is generally allocated for decode and many islands for prefill, this displays that prefill is the bottleneck for throughput.

Collating the assignment percentages with throughout measurements (Figure 5.2 and Figure 5.1) we can plot the actual throughput per island, shown in Figure 5.3.

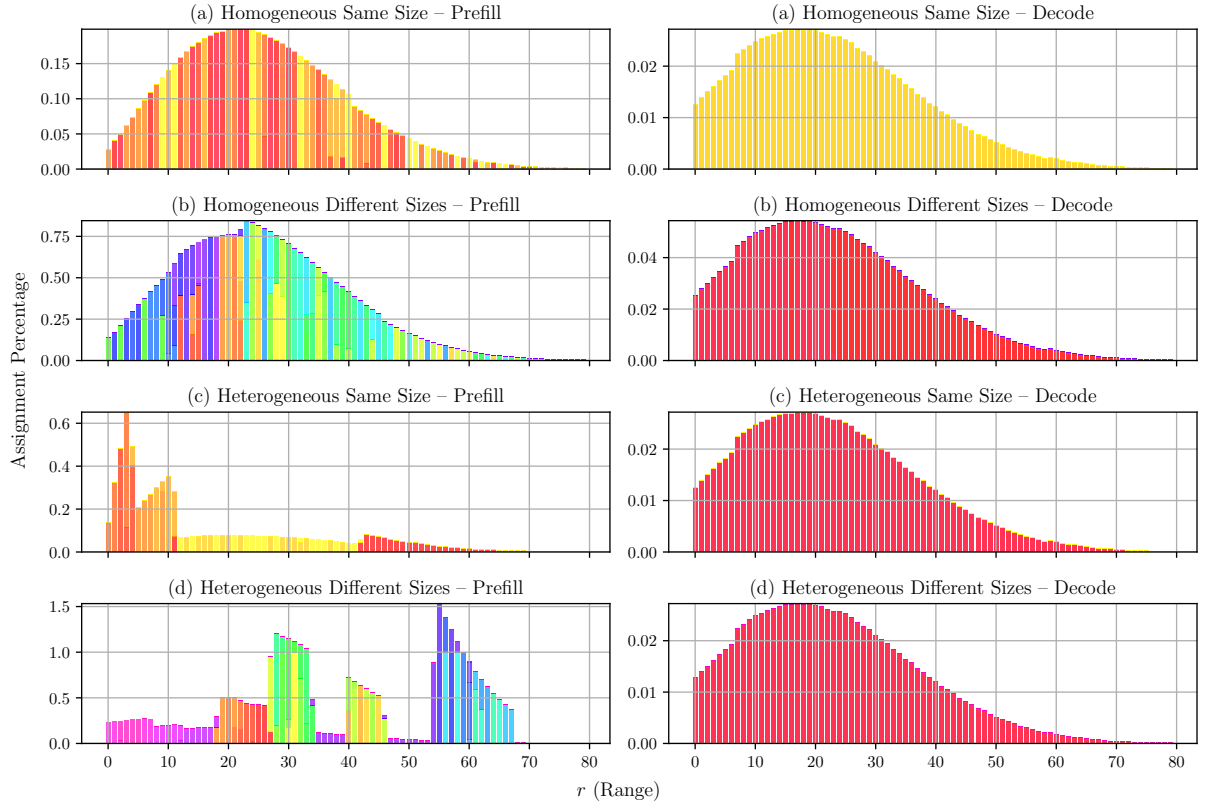


Figure 5.2: Assignment values percentage (per range) for each island test case, using artificial workload as described in Figure 4.2

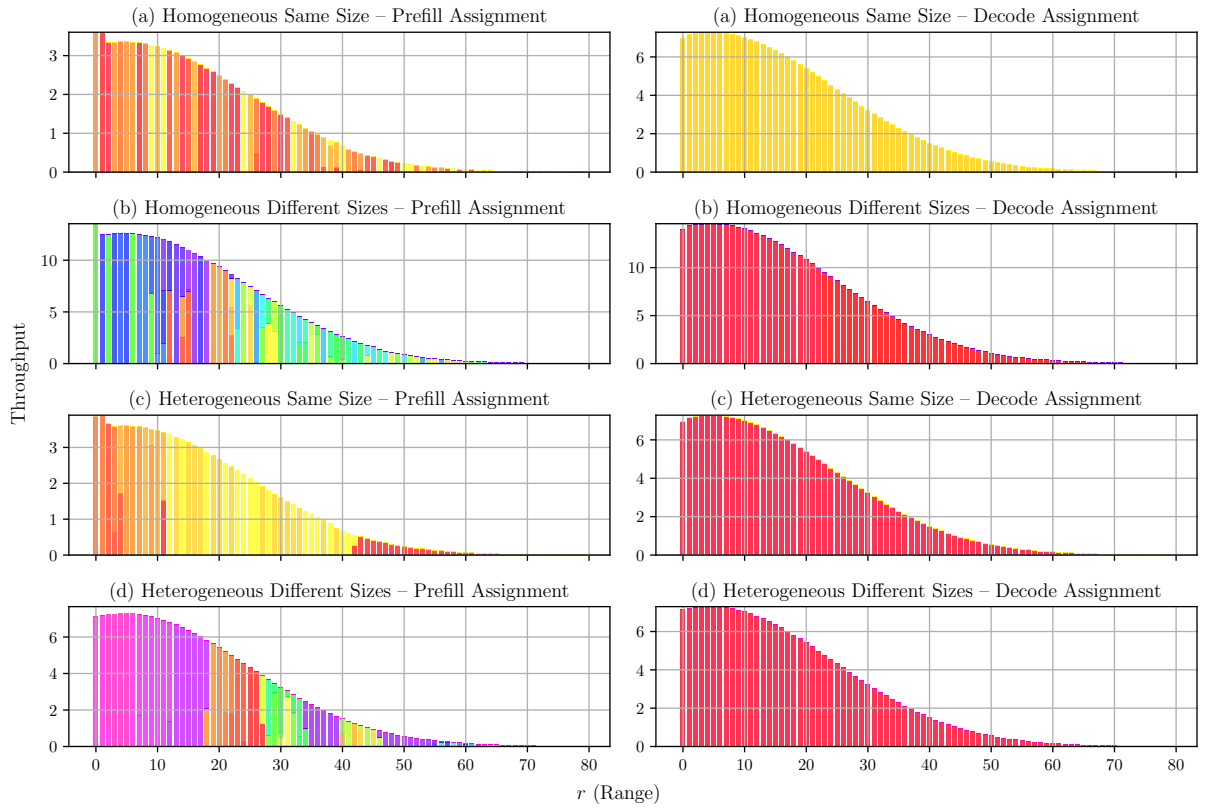


Figure 5.3: Throughput per range for each every island test case, using artificial workload as described in Figure 4.2

5.2.2 Evaluator Results

Previous plots demonstrate the allocation results using direct outputs from the linear solver at the specified range resolution. We need to evaluate the solutions using the complete expected input sequence length distribution $p(s | w)$ to highlight any differences in performance. These results are displayed in Figure 5.4, where *evaluated* refers to results from the evaluator and *model* refers to results directly from the solver. We also include the workload *trace* as a reference to what the target throughput distribution is.

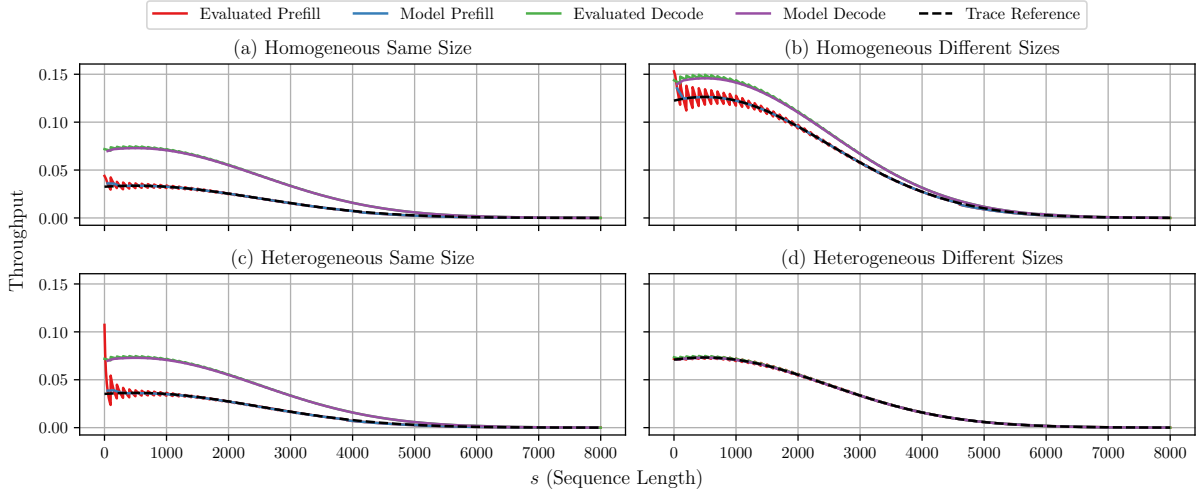


Figure 5.4: Throughput per sequence length for each island test case, using artificial workload as described in Figure 4.2. Such probability trace is overlaid and scaled to demonstrate solution divergence.

Displayed in Figure 5.4, the linear optimisation precisely maps work from the input trace to the GPUs. However, for all but experiment (d), the prefill and decode phase throughputs are miss-aligned (diverging). This is because the islands provided do not have a ‘naturally’ aligning solution, due to their shape and size it is impossible to perfectly allocate prefill and decode, given they cannot be mixed. We can expect the outer optimiser to solve this issue, as it will be able to perfectly allocate island sizes.

The oscillation displayed between expected and model throughput is expected, this is because the model executes at a lower resolution. Hence any throughput allocations will likely be over allocated for sequence lengths shorter than the midpoint, but under allocated for longer lengths.

Evaluator Method

The true throughput is calculated by first benchmarking each exact sequence length (whilst also selecting the best parallelism configuration), for each island using the same method as described in section 4.8. Then, the throughputs are weighted with the p value of each sequence length s along with the island percentage allocation x , of the range the sequence length resides within.

5.3 Outer Loop (Divider Approach)

We test the *outer loop* in the *divider* island mode using an inventory containing 128 NVIDIA DGX-300 and 256 NVIDIA H200. We select this single inventory due to the diverse nature of the DGX-300 and H200, given the DGX-300 has approximately three times the compute. This inventory allows us to execute experiments on a strongly heterogeneous cluster. The goal of these experiments is to confirm parameter choices for the Bayesian Optimisation loop.

Experiment Setup As the *outer loop* is non deterministic, we execute the experiment several times and take averages of the results (trials). The number of trials is specifically shown under every figure, where we balance accuracy versus compute time.

Plot Description Each *outer loop* plot contains a line of mean best throughput, this is the mean throughput across the best solutions in each trial, calculated at a specific iteration number. Plots also contain min-max ranges and standard deviations, this is calculated similarly to mean throughput; values are computed at each specific iteration index across all trials at that point. Finally, a line of maximum throughput is also plotted, this is the maximum throughput across all trails, iterations and experiments. Figure 5.5 contains a reference of this style of plot.

5.3.1 Batch Size Analysis

“How does changing the batch size, of solutions tested at each iteration, affect the results of the Bayesian Optimisation outer loop?”

We test the throughput performance of the scheduler using different batch size parameter values for the Bayesian Optimisation model. This increases the number of solutions proposed at each iteration of the acquisition function. Increasing the batch size has several benefits, specifically higher quality solutions with less variance. However this comes at the drawback of extra compute time for every iteration cycle.

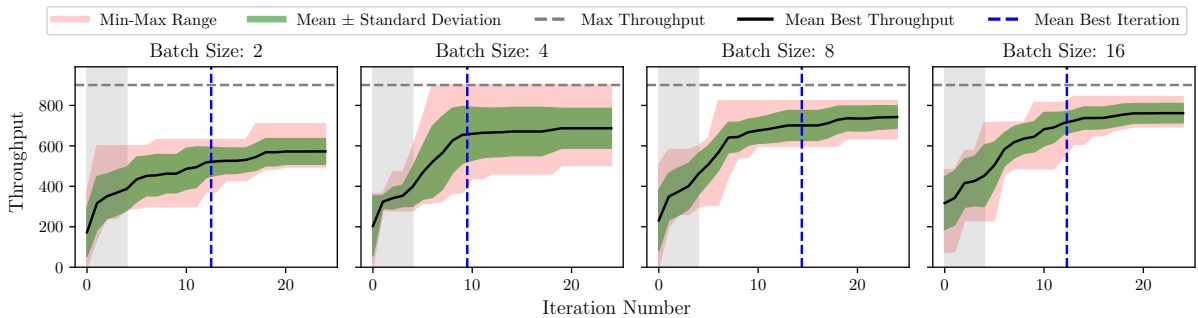


Figure 5.5: Throughput per iteration number for different batch sizes. **Experiment Parameters:** iterations = 20, trials = 10, min island size = 2, skew range = ± 5.0 .

Displayed in Figure 5.5, as the batch size increases, the variance of solutions decreases. This is expected behaviour as if we increase the number of solutions proposed, it is more likely one of these solutions is good. Shown in Figure 5.6, this does however result in increased execution time, where execution time is approximately proportional to batch size. Execution time is defined as the complete time from start to finish of all iterations, measured per trial. We select batch size 16 for the final scheduler, but acknowledge this comes at the cost of requiring more compute time to complete.

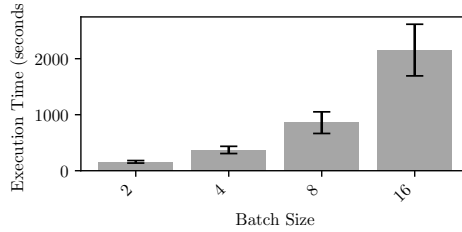


Figure 5.6: Execution time of scheduler for different batch sizes. Using experiment parameters as of Figure 5.5

5.3.2 Skew Range Analysis

“How does changing the range of the skew search parameter affect the results of the outer loop whilst using the divider approach?”

Next we test how altering the skew search parameter affects the scheduling throughput value. Displayed in Figure 5.7, higher skew ranges result in marginally higher throughputs, whilst retaining similar variance to smaller ranges.

This result is logical as we are able to traverse more of the island division search-space when the range is higher, unlocking possibly better solutions. Despite the increased search space, this completes in approximately the same execution time. This is expected, as we are not altering the parameters of the Bayesian Optimisation execution itself, but rather the parameters it uses to search. Hence, we use the full ± 5.0 skew range in the main scheduler due to its improved solution generation.

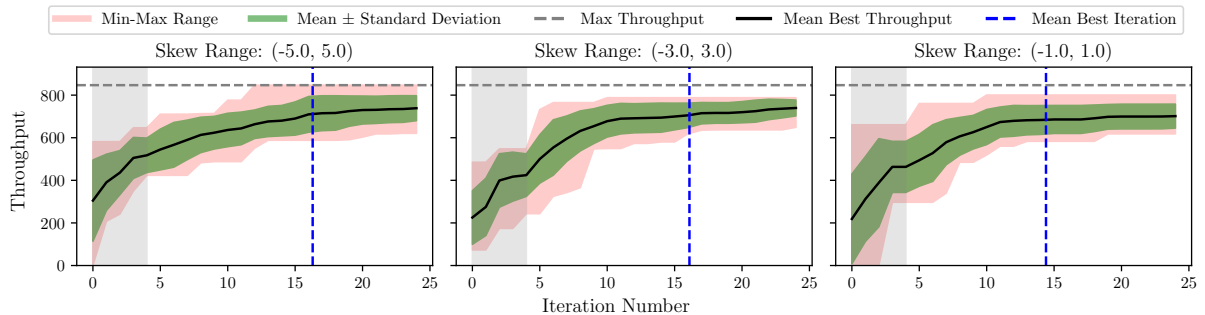


Figure 5.7: Throughput per iteration number for different skew ranges (Divider). **Experiment Parameters:** batch size = 8, iterations = 20, trials = 10, minimum island size = 2

5.3.3 Minimum Island Size Analysis

“How does changing the minimum number of GPUs in an island affect the results of the in outer loop whilst using the divider approach?”

We also test how altering the minimum island size affects the result of the optimisation. Like the skew value analysis, this is altering a search parameter, not an execution parameter of the Bayesian Optimisation.

Figure 5.8 displays that for every increase in minimum island size, effective throughput halves. This is due to the prefill phase preferring more smaller islands, than fewer bigger islands, and by increasing the minimum size, fewer small islands are generated. We select the minimum number of islands to be two for the final scheduler. While we would likely see performance gains with a minimum size of one, this parameter acknowledges that GPU allocation may be restricted to pairs of GPUs in cloud environments.

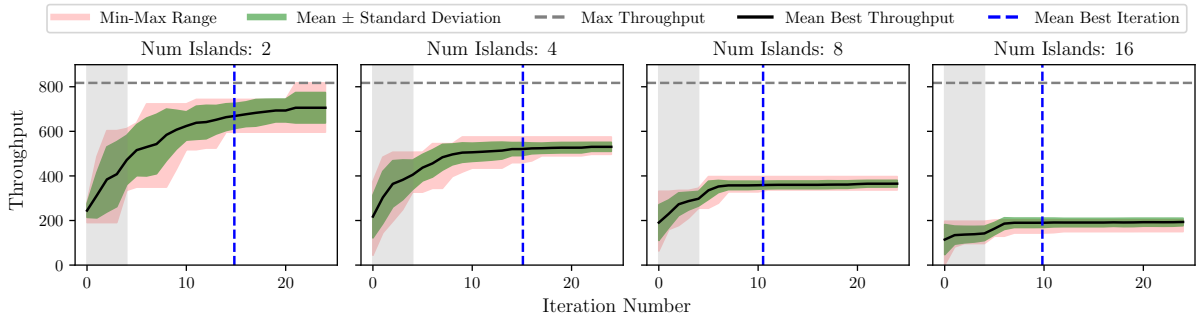


Figure 5.8: Throughput per iteration number for different minimum island sizes (Divider). **Experiment Parameters:** batch size = 8, iterations = 20, trials = 10, skew range = ± 5.0 .

5.4 Outer Loop (Direct Approach)

We also test the *outer loop* in *direct* mode, to demonstrate the performance relative to the *divider* island generation approach. These experiments use the exact same experiment setup, input inventory and workload trace as of section 5.3. We use these experiments to analyse the difference in performance between the standard BO and SAASBO optimisers.

5.4.1 Batch Size Analysis

“How does changing the batch size, of solutions tested at each iteration, affect the results of the outer loop whilst using the direct approach?”

We first test using the SAASBO Bayesian Optimisation approach. Figure 5.9 displays that for every increase in batch size, throughput also improves, at the cost of higher variance. Comparing against the *direct* island generation performance, this approach performs significantly worse, with maximum throughput values of just 300 requests/s.

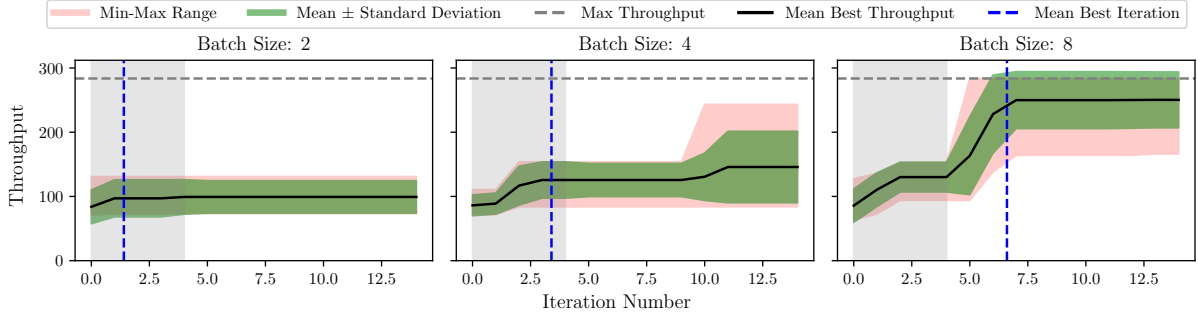


Figure 5.9: Throughput per iteration number for different batch sizes (Direct SAASBO). **Experiment Parameters:** iterations = 10, trials = 5, k slots = 16

We also complete the same analysis of the *direct* island generation approach whilst using standard Bayesian Optimisation. Surprisingly Figure 5.10 performs better (faster convergence) than the SAASBO approach of Figure 5.9. Furthermore the BO approach executes significantly faster than the SAASBO, Figure 5.11. Whilst the *direct* generation approach is categorically worse, it is interesting to observe that SAASBO is not necessarily required on such high dimensionality optimisation.

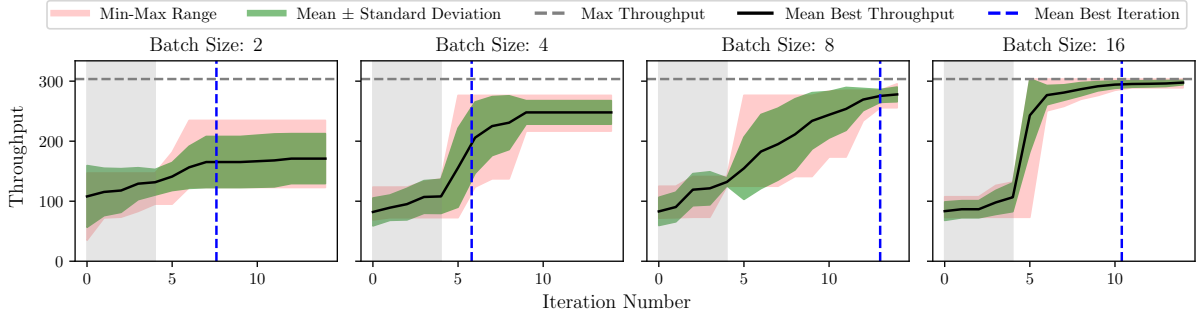


Figure 5.10: Throughput per iteration number for different batch sizes (Direct BO). **Experiment Parameters:** iterations = 10, trials = 5, k slots = 16

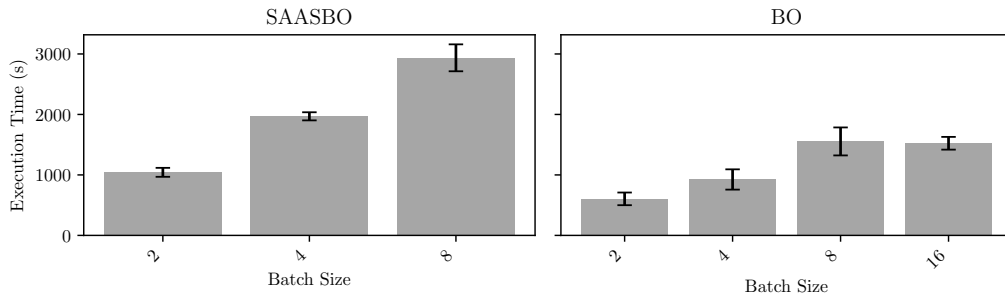


Figure 5.11: Execution time of direct scheduler (BO vs SAASBO) for different batch sizes. Using experiment parameters as of Figure 5.10 and Figure 5.9

5.4.2 K Slots Analysis

“How does changing K slots value, of solutions tested at each iteration, affect the results of the outer loop whilst using the direct approach?”

We perform one last experiment to try understand the behaviour of the *divider* island generation approach. Specifically, we adjust the k slots variable (variable responsible for the high dimensionality of this approach).

Displayed in Figure 5.12, as k slots increases so does performance, at the cost of significantly higher variance. As demonstrated in subsection 5.3.3 more small islands are better for prefill performance. Hence, increasing k slots improves performance, but with the downside of exploding dimensionality.

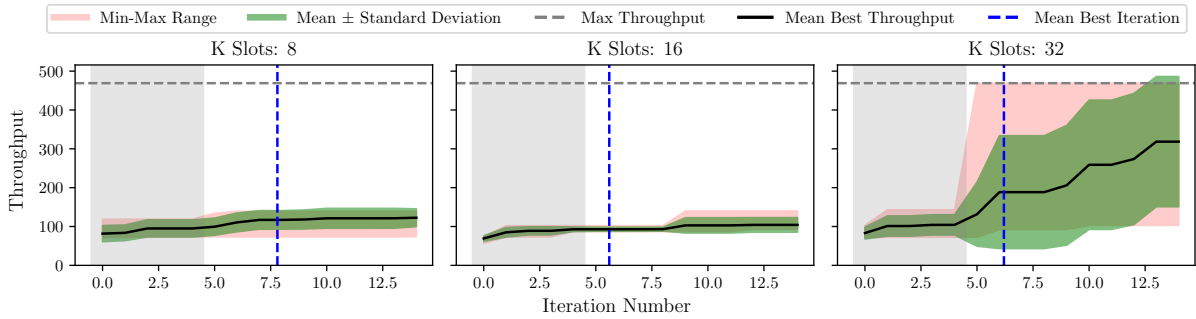


Figure 5.12: Throughput per iteration number for different K slots (Direct SAASBO). **Experiment Parameters:** batch size = 4, iterations = 10, trials = 5

5.5 Workload Traces

Azure LLM

We use the Azure LLM [35] workload traces for our experiments with the developed scheduler. The datasets provide a sample lists of requests in a time period, served by Azure’s servers. We convert these into probability models shown in Figure 5.13, these are used as the input workload to the scheduler.

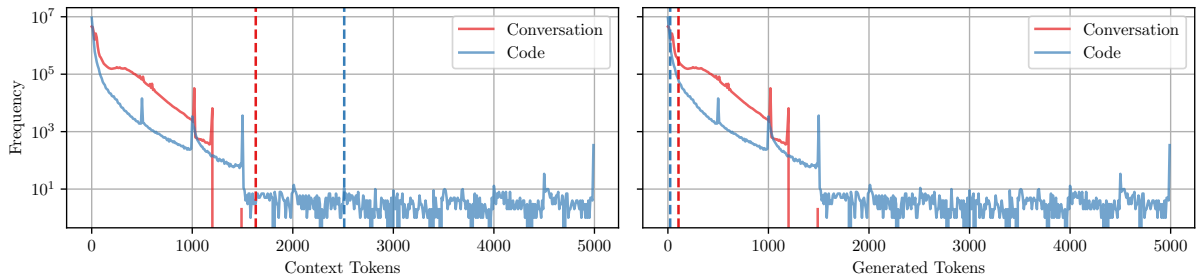


Figure 5.13: Azure LLM Context and Generated Tokens

5.6 Discussion

Within the evaluation section we successfully demonstrated the performance of the scheduling algorithm, comparing heterogeneous and homogenous clusters executing the same workload. The heterogeneous cluster performed with $1.4\times$ higher throughput than the homogenous for code and $1.2\times$ higher for conversation workloads.

The investigation inside the *inner loop* demonstrated the scheduler is able to precisely map allocation using the expected workload probability model. The detailed *outer loop* analysis investigated the best search parameters for the Bayesian Optimisation process. The high skew and low minimum island count produced better throughput results. The results also displayed a larger batch size significantly improves performance, but increases execution time.

We also demonstrated the *divider* island approach performs significantly better than the *direct* approach. Even under best conditions for the *direct* method, it performs at over half the throughput compared to the *divider* approach.

Chapter 6

Conclusion

Motivated by the significant operational costs of Large Language Model (LLM) inference and the under-explored challenge of optimising Mixture-of-Experts (MoE) models on heterogeneous hardware, this work has developed a novel, two-level scheduling algorithm. Building upon previous work in *phase splitting* [9] [10] and *heterogenous serving* [13] [11] we integrate this technology into a MoE environment. Specifically, We explored a new architecture that decouples the problem into two distinct stages: an *outer loop* for island generation and an *inner loop* for workload assignment.

We presented an *outer loop* that uses Bayesian Optimisation to efficiently search the complex configuration space, partitioning an inventory of GPUs into optimal homogeneous ‘islands’. For the *inner loop*, we devised and implemented a new linear programming formulation that precisely maps workload ranges, categorised by input sequence length, to the generated islands. The linear nature of this solver allows us to map workload at a much higher precision than previous scheduling algorithms, whilst retaining fast performance.

We evaluated the scheduling algorithm using our simulation framework, backed by Shallowsim [16], on real-world workload traces. This demonstrated that the proposed approach, on a heterogeneous GPU cluster, can achieve a 1.4× throughput improvement over a homogeneous cluster of the equivalent cost.

One limitation of our work is that the evaluation is based solely on simulated performance rather than a physical deployment. Also, due to the specialised nature of Shallowsim’s [16] GPU support, we were unable to perform detailed price-performance comparisons with different cluster types.

6.1 Future Work

This report demonstrated the development of a new scheduling algorithm and tested its performance using state-of-the-art simulator technology. A natural next step would be to implement this scheduler on a model-serving system, to test with real-world traffic and GPUs. This would require the development of a performant request routing solution, that is able to obey the precise output of the workload-aware scheduling algorithm.

There also exists several areas to add additional features to the scheduling algorithm itself:

- **Communication Models** - Currently, the communications between each of the GPUs is modelled naively. We assume that every GPU in the island has a high-bandwidth connection to other GPUs in that island. An expanded scheduler might take a GPU graph as an input to the *outer loop*, then using the edges as bandwidth, separate the GPUs into the most optimal islands.
- **Dynamic Inventory** - A key advantage of Bayesian Optimisation is that it can be expanded to contain multiple objective variables. Hence, we could modify the scheduler *outer loop* to use a dynamic inventory, where live GPU pricing is fed into the model. Then, given the target throughput distribution and magnitude, the scheduler could select the most optimal GPUs to achieve the task. This would allow cloud providers to serve models according to user demand, using the cheapest possible resources.
- **Live Rescheduling** - The current scheduling algorithm assumes a constant workload and hence a constant GPU arrangement. In reality, it is likely this workload would change over time depending on user demand, future work could expand the scheduler to consider the complexities of live model adjustment. This would introduce the concept of minimal change to the *inner loop* linear solver, where it is preferred, that islands do not toggle between prefill and decode whilst in operation.

References

- [1] OpenAI *et al.* “GPT-4 Technical Report.” Comment: 100 pages; updated authors list; fixed author names and added citation. arXiv: 2303.08774 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2303.08774>, pre-published.
- [2] A. Grattafiori *et al.* “The Llama 3 Herd of Models.” arXiv: 2407.21783 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2407.21783>, pre-published.
- [3] G. Team *et al.* “Gemini: A Family of Highly Capable Multimodal Models.” arXiv: 2312.11805 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2312.11805>, pre-published.
- [4] A. Q. Jiang *et al.* “Mixtral of Experts.” Comment: See more details at <https://mistral.ai/news/mixtral-of-experts/>. arXiv: 2401.04088 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2401.04088>, pre-published.
- [5] A. S. Luccioni *et al.*, “Power Hungry Processing: Watts Driving the Cost of AI Deployment?” In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, pp. 85–99. DOI: 10.1145/3630106.3658542. arXiv: 2311.16863 [cs]. [Online]. Available: <http://arxiv.org/abs/2311.16863>.
- [6] J. Zhao *et al.* “LLM-PQ: Serving LLM on Heterogeneous Clusters with Phase-Aware Partition and Adaptive Quantization.” arXiv: 2403.01136 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2403.01136>, pre-published.
- [7] X. Miao *et al.* “SpotServe: Serving Generative Large Language Models on Preemptible Instances.” Comment: ASPLOS 2024. arXiv: 2311.15566 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2311.15566>, pre-published.
- [8] Y. Jiang *et al.* “Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs.” arXiv: 2502.00722 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2502.00722>, pre-published.
- [9] Y. Zhong *et al.* “DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving.” Comment: OSDI 2024. arXiv: 2401.09670 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2401.09670>, pre-published.

- [10] P. Patel *et al.* “Splitwise: Efficient generative LLM inference using phase splitting.” Comment: 12 pages, 19 figures. arXiv: 2311.18677 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2311.18677>, pre-published.
- [11] Y. Jiang *et al.* “HexGen: Generative Inference of Large Language Model over Heterogeneous Environment.” Comment: Accepted by ICML 2024. arXiv: 2311.11514 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2311.11514>, pre-published.
- [12] Y. Jiang *et al.* “HexGen-2: Disaggregated Generative Inference of LLMs in Heterogeneous Environment.” Comment: ICLR 2025. arXiv: 2502.07903 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2502.07903>, pre-published.
- [13] Y. Jiang *et al.* “ThunderServe: High-performance and Cost-efficient LLM Serving in Cloud Environments.” Comment: MLSys 2025. arXiv: 2502.09334 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2502.09334>, pre-published.
- [14] DeepSeek-AI *et al.* “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.” arXiv: 2501.12948 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2501.12948>, pre-published.
- [15] DeepSeek-AI *et al.* “DeepSeek-V3 Technical Report.” arXiv: 2412.19437 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2412.19437>, pre-published.
- [16] icezack12, *Icezack12/shallowsim*. [Online]. Available: <https://github.com/icezack12/shallowsim>.
- [17] A. Vaswani *et al.* “Attention Is All You Need.” Comment: 15 pages, 5 figures. arXiv: 1706.03762 [cs]. (), [Online]. Available: <http://arxiv.org/abs/1706.03762>, pre-published.
- [18] M. Lewis *et al.* “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension.” arXiv: 1910.13461 [cs]. (), [Online]. Available: <http://arxiv.org/abs/1910.13461>, pre-published.
- [19] J. Devlin *et al.* “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: 1810.04805 [cs]. (), [Online]. Available: <http://arxiv.org/abs/1810.04805>, pre-published.
- [20] A. Radford *et al.*, “Improving Language Understanding by Generative Pre-Training,”
- [21] Y. Sheng *et al.* “FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU.” arXiv: 2303.06865 [cs]. (), [Online]. Available: <http://arxiv.org/abs/2303.06865>, pre-published.
- [22] “Mastering LLM Techniques: Inference Optimization,” NVIDIA Technical Blog. (),

- [Online]. Available: <https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/>.
- [23] R. Pope *et al.* “Efficiently Scaling Transformer Inference.” arXiv: 2211.05102 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2211.05102>, pre-published.
 - [24] N. Shazeer *et al.* “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.” arXiv: 1701.06538 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/1701.06538>, pre-published.
 - [25] D. Lepikhin *et al.* “GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding.” arXiv: 2006.16668 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2006.16668>, pre-published.
 - [26] D. Eriksson and M. Jankowiak. “High-Dimensional Bayesian Optimization with Sparse Axis-Aligned Subspaces.” Comment: To appear in UAI 2021.
arXiv: 2103.00349 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2103.00349>, pre-published.
 - [27] “ShareGPT.” (), [Online]. Available: <https://sharegpt.com>.
 - [28] M. Chen *et al.* “Evaluating Large Language Models Trained on Code.” Comment: corrected typos, added references, added authors, added acknowledgements.
arXiv: 2107.03374 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2107.03374>, pre-published.
 - [29] Y. Bai *et al.* “LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding.” Comment: ACL 2024. arXiv: 2308.14508 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2308.14508>, pre-published.
 - [30] W. Kwon *et al.* “Efficient Memory Management for Large Language Model Serving with PagedAttention.” Comment: SOSP 2023.
arXiv: 2309.06180 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2309.06180>, pre-published.
 - [31] *Deepspeedai/DeepSpeed*, deepspeedai.
[Online]. Available: <https://github.com/deepspeedai/DeepSpeed>.
 - [32] A. Agrawal *et al.* “Vidur: A Large-Scale Simulation Framework For LLM Inference.” arXiv: 2405.05465 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2405.05465>, pre-published.
 - [33] *Facebook/Ax*, Meta. [Online]. Available: <https://github.com/facebook/Ax>.
 - [34] *Pytorch/botorch*, pytorch.
[Online]. Available: <https://github.com/pytorch/botorch>.

- [35] J. Stojkovic *et al.* “DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency.” arXiv: 2408.00741 [cs]. (),
[Online]. Available: <http://arxiv.org/abs/2408.00741>, pre-published.