SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

LLM-OS: Orchestrating Edge Inference with Priority Scheduling and Adaptive KV Cache Management

Berkay Eren Ürün

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

LLM-OS: Orchestrating Edge Inference with Priority Scheduling and Adaptive KV Cache Management

LLM-OS: Orchestrierung von Edge-Inferenz mit Prioritätsplanung und adaptiver KV-Cache-Verwaltung

Author:	Berkay Eren Ürün
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Teofil Bodea
Submission Date:	15.05.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.05.2025

Berkay Eren Ürün

Acknowledgments

I want to thank Prof. Dr. Pramod Bhatotia for his close interest in his students. His door was always open to help and support in any way possible. To my advisor, Teofil Bodea, I express my deepest gratitude and thanks. He always knew how to show me the way. With his insightful feedback, he not only guided me to create better work but also helped me to improve myself. His energy, positivity and great sense of humor kept my motivation high throughout this thesis.

Abstract

Cloud-based inference for large language models (LLMs) benefits from powerful hardware such as high-performance GPUs and large memory capacities, enabling efficient processing of complex models. However, cloud deployment introduces critical limitations including data privacy concerns, increased latency due to network communication, and potential availability issues tied to connectivity and server load. Shifting LLM inference to edge devices addresses these drawbacks by enabling local processing, reducing latency, and eliminating the need to transmit sensitive data to third-party servers.

Despite these benefits, edge deployment presents new challenges: limited compute and memory resources, energy consumption constraints, and difficulties in ensuring model confidentiality when the model is locally stored on an edge device. The LLM-OS project aims to overcome these issues by designing a secure, efficient, and adaptable local inference framework for LLMs on edge devices.

In this thesis, we design, implement, and evaluate core components of LLM-OS. Our design consists of two isolated virtual machines running on top of a hypervisor. The first VM runs multiple instances of our highly configurable client that simulates a realistic user environment requiring LLM services. The second VM hosts our LLM-OS kernel that runs a custom inference server, providing LLM service with a priority-based scheduler and dynamic KV cache policies. Both VMs are connected via a communication channel for request exchange. We utilize llama.cpp as the inference engine and support GPU passthrough for acceleration.

Evaluation on both CPU and GPU setups shows that our prototype introduces less than 3% overhead on CPU and approximately 12% on GPU compared to baseline inference, while enabling concurrent multi-client inference on edge devices. These results demonstrate the feasibility of LLM-OS's design and lay the groundwork for further development.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **5**
2.1 Large Language Models 5
2.2 Optimizations for Large Language Models 6
2.2.1 KV Cache 7
2.2.2 Batching 8
2.3 LoRA 9
2.4 Unikernel 9

3 LLM-OS Overview **11**
3.1 Design Goals 12
3.2 Threat Model 13
3.3 Workflow 13

4 Design **16**
4.1 Untrusted Host OS 17
4.2 Host/LLM-OS Kernel Communication 17
4.3 LLM-OS Kernel 18
4.3.1 Scheduler 18
4.3.2 Context Manager 19
4.3.3 Inference Engine 20
4.3.4 Memory Manager 21
4.4 GPU 21

5 Implementation **23**
5.1 QEMU/KVM Hypervisor 24
5.2 Untrusted Host 25

5.3 LLM-OS Kernel 26
5.3.1 Scheduler 28
5.3.2 Context Manager 28
5.3.3 Memory Manager 29
5.3.4 Inference Engine 29
5.4 Host/LLM-OS Kernel Communication 32

6 Evaluation **35**
6.1 Experimental Setup 35
6.2 Overhead Analysis 36
6.3 Scaling with Clients 37
6.4 Impact on Host OS 39
6.5 Context Switch Latency 40

7 Related Work **42**

8 Conclusion **44**

9 Future Work **45**

List of Figures **46**

Bibliography **47**

1 Introduction

Large language models (LLM) distill large amounts of textual information. With this condensed knowledge, it is possible to create virtual assistants that can answer, sum up, translate, or analyze any given text.

These LLM assistants have been used in a variety of different fields. For example, OpenAI's ChatGPT, an example of a general chatbot, has around 400 million weekly users [41]. On the other hand, Google's Med-PaLM 2 specializes in medical questions [45]. In robotics, PaLM-E [7] collects real-world continuous sensor data to use as an LLM input.

Traditionally, many LLMs run on remote cloud servers provided by third parties with the necessary infrastructure. The reason is LLMs' high computational requirements. A performant LLM inference system must make use of accelerators like TPU or GPU to increase its throughput. Additionally, LLM models consume a significant amount of disk space and require a high bandwidth memory. Unfortunately, such a costly setup is mostly only available to third-party cloud providers. However, an external remote server for inference is not always an ideal or even a possible option because of three fundamental limitations.

The first issue with having a cloud server is data privacy. In order to use the remote LLM service, users must send their inputs to a remote cloud server that a third party owns. However, letting an external entity process the user data requires high trust and might be subject to some limitations mandated by regulations. If we assume that sending data to a third-party server is fine, we face the second issue: latency. Communicating with a remote server requires transferring messages over relatively long distances. Even in perfect conditions, latency is inevitable because of the limited speed of light and, therefore, information. This communication latency can be critical in some use cases like real-time systems. Even when data privacy is not a significant concern and the latency is acceptable for the specific use case, the availability of the cloud server could still be an issue. Dependency on a network connection for the LLM service implies the need for a relatively high bandwidth and a stable connection to the server. However, in many different scenarios, such a setup can fail. For example, user-side internet connectivity can be temporarily unideal, or it might not exist in the first place. On the other hand, a cloud server can be inaccessible or only be able to provide a limited service during a high load period, or in the case of a cyber-attack.

A solution to the aforementioned problems is moving the LLM model from the cloud onto a user's edge device [60]. With local LLM deployment, there is no need for a server connection; therefore, there are no extra latency or availability issues. Also, since the data is not sent to the outside world, there are no privacy issues stemming from the remote server.

Unfortunately, the advantages of on-device LLMs come with challenges requiring novel solutions. Model training can cost hundreds of millions of dollars [30]. In addition to the cost of compute comes the cost of the data used for training, and the cost of human resources used in the development of the model. Altogether, the high effort to create an LLM model made companies view their models as intellectual properties that must be protected and kept confidential. However, moving the models from the company's cloud to a user's edge devive is challenging with these security requirements. For example, the model becomes more vulnerable when the model weights, hyperparameters, and architecture are stored locally on the user device instead of the company's secure servers. Therefore, model confidentiality must be assessed accordingly at the edge level. Another issue with the LLM on edge is the computing capabilities. Handling LLMs requires a highly performant system. However, edge devices are far less powerful than a traditional dedicated LLM server. They lack server-grade GPUs, have less memory, and might need to adhere to energy consumption policies while handling other possibly real-time non-LLM tasks. Because of these constraints, a server-like approach to maximizing the LLM serving performance is often not ideal for the edge world.

Many research papers directly or indirectly address the latter issue of improving LLM performance and energy efficiency on edge [10]. For example, EdgeMoE [55] uses a tiered storage for the model and loads the weight dynamically to better utilize the limited available memory. PagedAttention proposed by Kwon et al [23], uses paging to better use the memory. Quantization, mapping model weights to less precious values, and pruning, removing specific weights, are researched vastly to reduce the memory footprint and the computational overhead [28] [46].

On the other hand, the security of LLM models on edge deployment has not been studied as much as their performance and efficiency. The main reason is that the works done with the assumption of remote cloud deployment do not cover the issues of the edge world. Most of these studies focus on a black box LLM that can only be accessed using a well-defined API [3] [48]. In an edge environment, however, API access does not cover all the attack surface since an attacker might have access to hardware and the privileged software running on it, such as the operating system. Some works address this by using Trusted Execution Environments (TEE) [14] [34] [17]. However, because of TEEs nature, they suffer from low performance and require modification on the user application. We discuss the state of the related work in depth in Chapter 7.

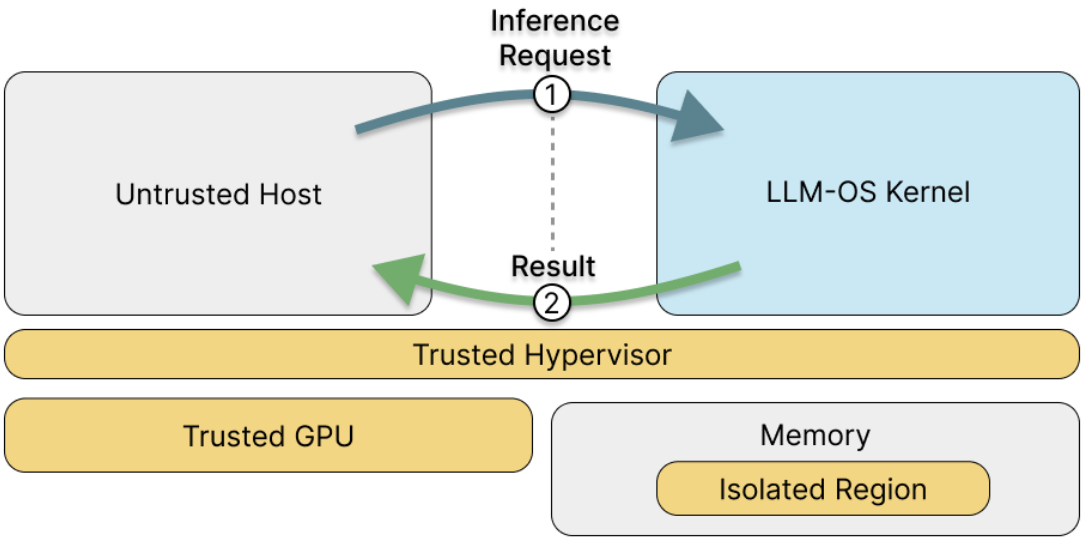

Figure 1.1: Simplified overview of LLM-OS. The untrusted host, requires an LLM service. LLM-OS kernel provides the required service

As a solution, we propose LLM-OS, an inference framework with OS-level access to low-level resources. LLM-OS aims to combine all the aforementioned open topics and achieve performant, energy-efficient, and secure on-device LLM inference that can work with other concurrent processes required by the edge environment. As illustrated in Figure 1.1, LLM-OS runs next to the untrusted host OS and acts as a secure local inference server. The trusted hypervisor isolates LLM-OS from the host. This allows LLM-OS to handle inference computations while keeping the model secure. Applications running on the untrusted host first attest to LLM-OS and then send their requests. LLM-OS will run inference in a secure, isolated environment and send the result back. Since LLM-OS is designed directly for inference, it will optimally manage the available resources, such as memory and compute power. The details of the project and goals will be discussed in Chapter 3

In this work, we focus on the LLM-OS kernel and its communication with the untrusted host. This serves as the basis and the initial prototype for the LLM-OS project.

We have built the first LLM-OS instance on top of x86_64 NixOS. This LLM-OS prototype runs our custom inference server using Llama.cpp as the inference engine. We will discuss the necessary background knowledge for our choice of inference engine and our contributions in Chapter 2. We have used QEMU/KVM [37] as the hypervisor to launch the LLM-OS kernel and create a shared memory between the

untrusted host and the LLM-OS kernel. With this setup, multiple host-side clients can make requests and receive the output in real-time. The exact implementation will be discussed in Chapter 5, right after we explain our design in Chapter 4.

Our evaluations of our implementation show around less than 12% of performance overhead on inference while using a GPU and less than 3% with a CPU. This inference service's overhead on host-side user processes is around 10%. More in-depth evaluation and the test setup are discussed in Chapter 6.

In summary, with this work, we contribute to the larger LLM-OS project by:

- Evaluating inference engines suitable for the project's context.
- Creating the base development environment.
- Developing a user-side client to simulate real-world use cases.
- Developing an efficient inter-virtual machine communication mechanism.
- Developing a custom inference server.
- Evaluating the first prototype with benchmarks.
- Inspecting the feasibility of possible future directions.

Throughout the paper, we will explain these contributions in detail. In Chapter 8, we will briefly recap our work, and in Chapter 9, we discuss the next steps of the LLM-OS project and conclude the thesis.

2 Background

LLM-OS makes use of a few fundamental large language model (LLM) concepts. We first start with how LLMs work at a high level, then explain advanced relevant details necessary for understanding our work.

2.1 Large Language Models

Large Language Models are models that are distilled from a large amount of textual information, with the goal of processing and generating human-like coherent and meaningful text.

LLMs consist of billions and sometimes trillions of parameters called weights. These parameters store the information obtained during the training process. Depending on the LLM design, weights can be implemented as integers or floating-point numbers with various precisions. The selection of the data type leads to a trade-off between the performance and the quality/size of the model [18].

LLMs have two fundamental stages in their lifecycle: training and inference. The former is the stage that updates the parameters based on the available training data. This results in a trained LLM that will be used by the users. Inference, which is more relevant for our work, happens when a user interacts with the LLM. On the highest abstraction level, when a user sends a text, LLMs split and map it into a stream of tokens for further processing. An example can be seen in Figure 2.1.

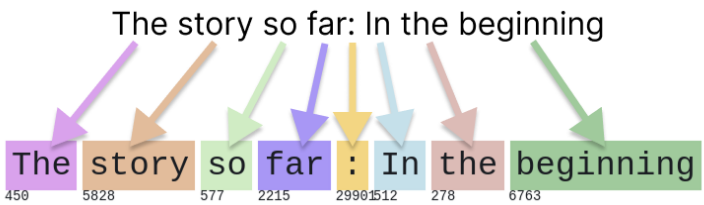

Figure 2.1: Tokenization of Llama decoder. Based on [51]

The mapped tokens are all fed to the LLM at the same time. This is called the prefill phase. Since this is the first inference execution with the given tokens, some

intermediate data structures are generated in this stage. Fortunately, prefill is highly parallel as all user input is known and can be processed concurrently. The results are a new token and data structures holding information about all the processed tokens.

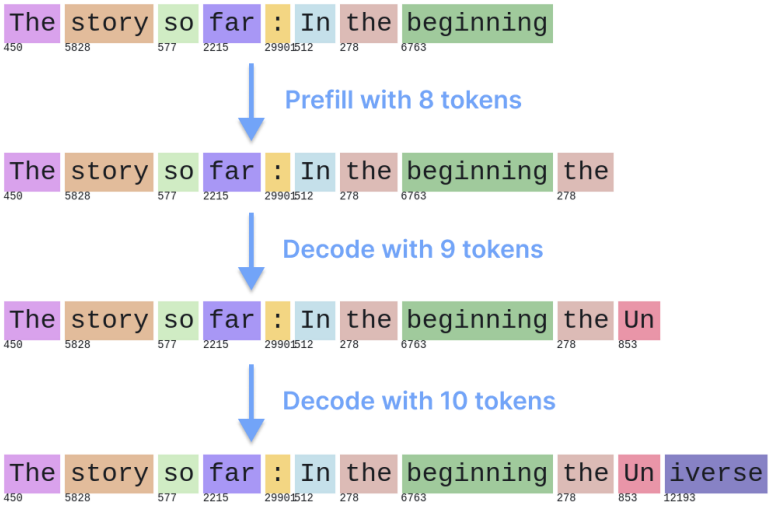

Figure 2.2: Autoregressive generation of tokens for a given sequence

After processing the initial input tokens and their related internal data structures, the next step is decoding. In this step, the LLM takes the initial tokens received from the user, appends the newly generated token from the prefill stage to the end of the tokens, and makes another run for token generation. If the intermediate data structures are available from the prefill, they will be reused; if not, decode will regenerate them. This decode run results in a single new token and either fresh or updated data structures with information on each token. The new token is appended to the existing ones. Together, they will be used as the input to the next decode step. The constant execution of decode will continue until an "end" token is generated. These iterative calls to the decode stage, illustrated in Figure 2.2, are called autoregressive generation.

2.2 Optimizations for Large Language Models

Optimization of LLMs is a highly studied field with multiple facets: model-, data-, and framework-centric views [50]. This work focuses on the relevant optimization from the model and framework-centric view, as the data-centric view is primarily applicable in the training phase.

2.2.1 KV Cache

As discussed in the LLM background, during inference, some intermediate data structures, holding information about all the processed tokens, are generated dynamically. These structures are matrices called Q (Query), K (Key), and V (Value). All these three matrices are required for token generation as they connect the token to be generated to the previous given or generated tokens' meaning context. Each row, or column, of Q, K, and V matrices stores parameters for a single token. While generating a new token, previous tokens' parameters are fetched from these matrices.

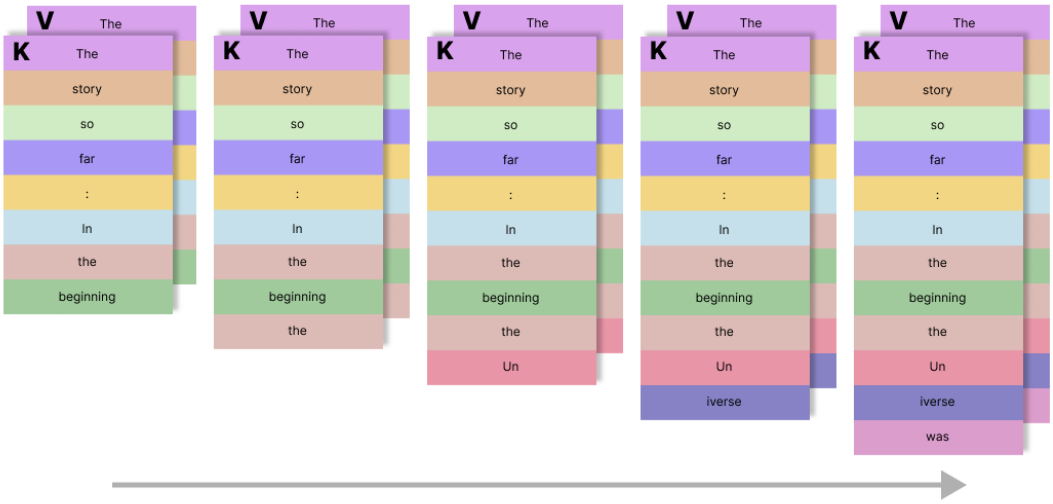

Figure 2.3: Example representation for K and V matrix of the example sequence from Figure 2.1. K and V of the KV cache store parameters for each processed token. With every generated token, both matricies grow.

The naive approach is not storing any of these matrices, and recomputing them fresh on every iteration. However, as mentioned, generating a new token requires parameters of the last token from Q and parameters of *all* previous tokens from K and V. The values of old tokens in Q, K, and V do not change after their generation. This means regenerating them is unnecessary, repeated work.

A simple way to avoid this repeated work is by caching the generated matrices. Since only the last token's parameters are read from the Q matrix, regenerating the relevant parameters is not costly. However, as illustrated in Figure 2.3, K and V grow linearly with the number of generated tokens and can quickly become expensive to regenerate. Therefore, caching mostly focuses on K and V matrices. In a simple implementation, each request, or sequence, has a separate KV cache.

Unfortunately, caching K and V matrices does not come for free. While it reduces the computation costs, it increases the system's memory consumption. Since the KV cache grows linearly with increasing tokens, it can consume a significant amount of memory. Many KV cache management ideas are proposed to solve this. Some works focus on token level optimizations, such as selecting only important tokens to store [11] [59] or reducing the precision of cached KV matrices [54] [44]. Other approaches use model level optimizations that focus on designing the model structure in a way that allows better KV cache management, for example by sharing the KV cache across the layers of the model [42] [1]. At the system level, research mainly focuses on direct memory management, scheduling and hardware-aware designs [27].

2.2.2 Batching

During inference, each token depends on all previous tokens; only with this dependency is it possible to create coherent human-like text. However, this also means that generating tokens of a single sequence is not a parallelizable operation, which leads to lower GPU utilization.

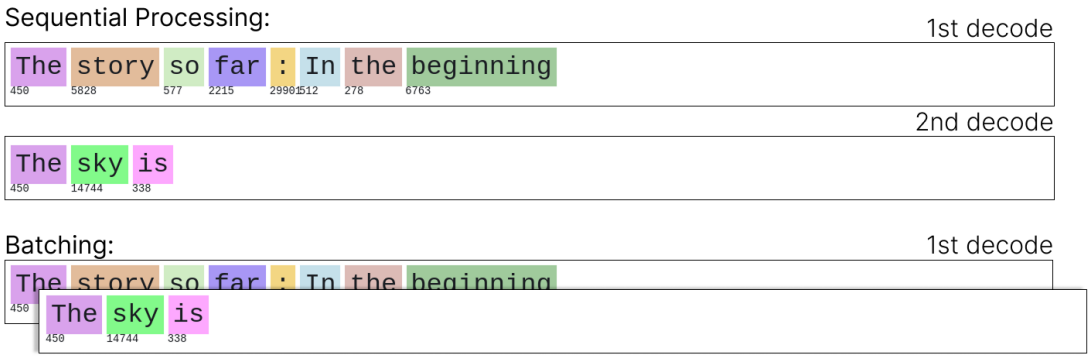

Figure 2.4: Decode with and without batching. Without batching, each sequence requires a separate decode run to generate its next token. With batching, both sequences can be combined together and a single decode run can generate two new tokens.

A solution to increase GPU utilization is batching. The main idea is to process multiple sequences together in a single decode step, as opposed to each sequence having its separate decode for a single token generation. Effectively, this means an extra tensor dimension during inference, defined by the batch size. The result is multiple new tokens, one for each sequence. An example with two separate sequences is illustrated in Figure 2.4.

While batching improves the GPU utilization, it has some constraints, such as higher memory consumption and latency [26] [39]. .

2.3 LoRA

LLMs can have more than a trillion parameters [8]. The money and time cost of training such a large model is enormous [53]. Therefore, retraining is not an option for specializing the model for a particular task.

Low-Rank Adaptation (LoRA) [16] is a method that trains only a part of a model for fine tuning or task specialization. This saves the cost of retraining or modifying all the model weights. To do that, LoRA introduces new smaller matrices that can be trained. Before deployment, the LoRA matrices can be merged with the base model. Such a deployment does not bring a runtime overhead; however, it requires extra computation before the inference can start.

LoRA can also be used as an adapter [58], meaning instead of merging the trainable weight matrices, they can be applied dynamically. With this approach, it is possible to hotswap LoRA adapters during inference runtime and change the specialization of the model. The cost of this flexibility is the introduced extra latency during inference.

2.4 Unikernel

Unikernels are compile-time specialized kernels that combine a user-level application with the required kernel parts in the form of operating system libraries [29]. The resulting image has a single address space for both OS services and the user application. A high-level comparison to the traditional OS structure can be seen in Figure 2.5. Such an image can be deployed directly on a hypervisor and requires no other operating system.

Since both the user application and the kernel find themselves in the same address space, context switches rooting from system calls do not exist, improving the performance. Additionally, during compilation, only the required components are linked. This leads to relatively low memory footprint and fast start-up times [20] [21]. Consequently, smaller images lead to reduced attack surface and more secure applications [47].

The most significant challenges Unikernel faces are the general maturity of the approach, the available tools to build/debug them, and their limited compatibility to existing applications and hardware [9].

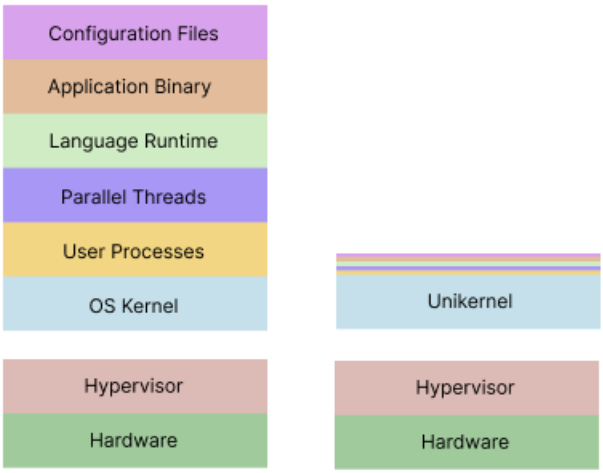

Figure 2.5: Existing OS structure compared with Unikernel approach

3 LLM-OS Overview

Deploying LLM models on edge devices instead of highly controlled, secure cloud servers raises confidentiality concerns. When the model is saved locally, any entity with access to the device might far more easily extract model parameters.

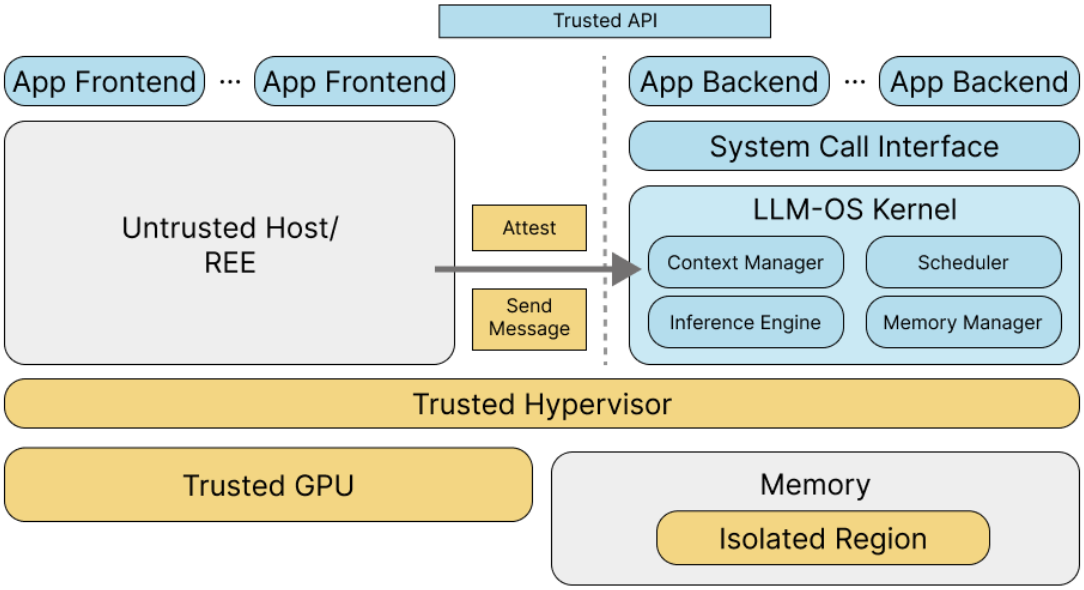

Figure 3.1: Overview of LLM-OS in detail

Our proposal to solve this is LLM-OS, visualised in Figure 3.1. With this approach, we run LLM inference in our LLM-OS kernel, a secure and isolated operating system specialized for efficient LLM processing on edge devices. The LLM-OS kernel is deployed together with a rich execution environment (REE) that runs the primary operating system. The trusted hypervisor isolates the two operating system from each other, while providing a secure communication channel in-between. Any process on the REE OS side can attest to LLM-OS kernel and send an inference request to use the LLM service. During processing these requests, LLM-OS can make use of the GPU and dynamically use LoRA adapters to provide its users with more fine-tuned answers.

The LLM-OS project expects to share the GPU between the untrusted host and the

LLM-OS kernel. This is required as the host might require the GPU while the LLM-OS kernel is running. However, sharing the GPU poses a risk on the model's security as the model must be transferred to the GPU memory for inference. Therefore, LLM-OS plans to clean the GPU so that the untrusted host should be able to use the GPU, *without* seeing any residue of model parameters. Additionally, the same parameters, on transfer from/to the GPU or in use by the GPU must be protected according to LLM-OS projects goals.

For the security of the model while it is not in use by the GPU, the trusted hypervisor also plays a vital role. It ensures the isolation of the untrusted host and the LLM-OS kernel to protect the model and any given LoRA weights. It also serves as a part of the trust chain that allows users to attest to the LLM-OS kernel.

All the aforementioned security requirements and the inference service on the LLM-OS kernel require resources. However, we assume the underlying edge device to have limited memory, compute power, and an energy consumption threshold. This means that our service cannot consume all the available resources. Instead, we should respect the host OS's resource requirements. Therefore, in our design, the healthy execution of the host processes takes precedence over fast inference.

We sum up all the requirements and the concerns of the LLM-OS project above in our design goals.

3.1 Design Goals

The design of the LLM-OS project has three main goals to achieve feasible secure inference on edge devices:

- **Trustworthiness** The central aim of LLM-OS is securing the model weights, including weights belonging to the user-given LoRAs. This involves their confidentiality while in memory or disk, in transit to the accelerator, or when in active use by the accelerator.

- **Resource efficiency** Since the deployment environment is edge devices with limited resources, efficiency is another critical topic that our design focuses on. Multiple concurrent apps must be able to use LLM-OS, while the host runs other user-level processes without interest in LLM services. The inference service provided by LLM-OS, therefore, must respect the performance of the user processes. Supporting multiple applications, not bottlenecking user-level processes, and conforming to energy consumption limits must precede high-throughput inference.

- **Adaptability** LLM-OS works together with the user-level applications. Each application might have different expectations or use cases from an LLM model. For example, a health application might need an LLM model specialized for health-related questions. Another application might focus on summarizing patient history saved as an SQL database entry on the same device. To address this, we want the user applications of LLM-OS to be able to specify a LoRA adapter for their target task.

3.2 Threat Model

We assume three parties: the application developer, the model provider, and the attacker. The application developer creates an app frontend and uses a trusted API to communicate with LLM-OS. Developers can also provide LoRA adapters and want to secure them. The model provider is the entity interested in protecting its model. Therefore, we assume that both application developers and model providers want to protect their model/adapter weights, parameters, and architecture from any other applications running on the host and also from the host itself.

The attacker, on the other hand, is the party with intentions to extract the model and/or LoRA data. For this purpose, we assume the attacker can directly compromise any host-side application or the host OS and interact with LLM-OS using the trusted API. Our assumption also includes the possibility that the attacker can control DMA-capable peripherals, including the GPU, which itself is trusted and can be identified via an attestation protocol. Lastly, an attacker might have access to the device and can perform cold boot attacks [57]. Denial-of-service and side-channel attacks are out of the project's scope, and we assume the hardware to be implemented according to specifications.

3.3 Workflow

As mentioned, the LLM-OS project comprises two sides: an untrusted host requiring LLM inference service, and the LLM-OS kernel fulfilling this demand while keeping the model secure. The trusted hypervisor manages both sides and isolates them from each other while maintaining a communication channel between them.

The untrusted host side is expected to run any regular user-level process, ranging from a simple database to a software system interacting with its environment via actuators. Independent of the purpose of the processes, we assume that some of them require a local LLM service. To use the LLM service, the processes must first create an app frontend. This user-side interface allows user processes to pre-process their

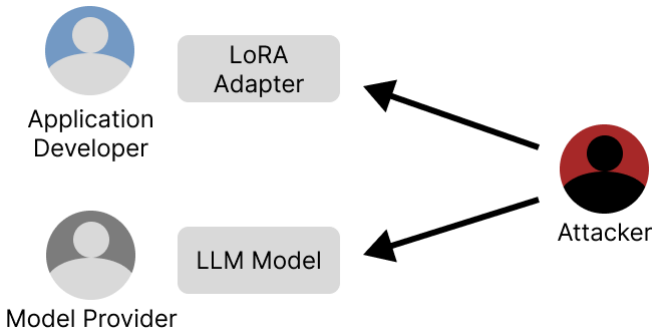

Figure 3.2: Threat model of LLM-OS. Application developers and model provider needs their weights to be protected. The attacker wants to extract the weights

inference requests and then communicate them to the LLM-OS kernel via the trusted API.

As seen in Figure 3.3, the first call to the LLM-OS kernel should attest to the kernel for security. The following call must create a backend. The resulting app backend serves as an LLM-OS kernel side pre-/post-inference processing location.

After the creation of the app backend, the user process can send a request over the communication channel. This request first lands on the app backend. Here, the backend can pre-process the request before forwarding it to the LLM-OS kernel. When the new request finally hits the LLM-OS kernel, the context manager creates a context for inference. This creation includes applying any LoRA adapter specified by the user. The created context is scheduled based on its priority among all available requests. If the request is chosen for execution, the inference engine runs inference with it, optionally using the trusted GPU. When a request is fully processed, the LLM-OS kernel forwards it to the respective app backend for optional post-processing defined by the user application. From there, requests reach to the app frontend, where an user-side optional post-processing can take place. As the last operation, the user can read the LLMs answer from the front end.

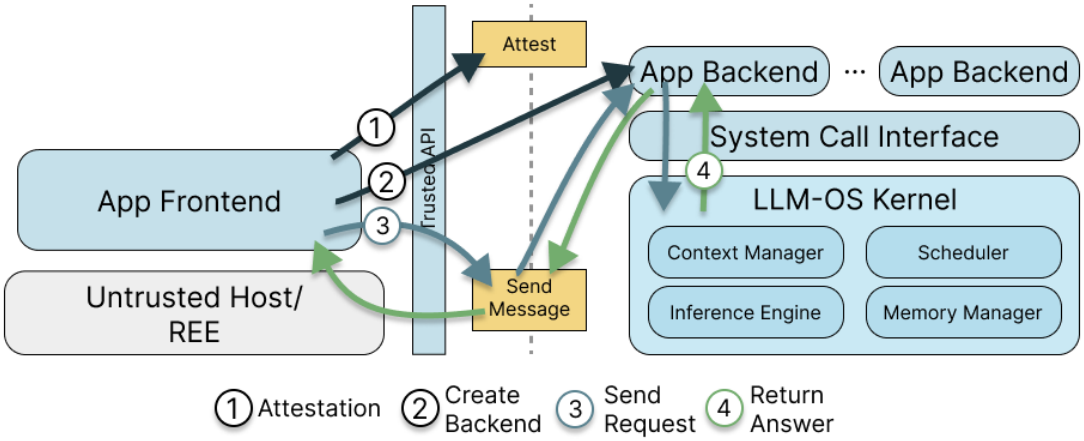

Figure 3.3: The user-side application must first attest to the LLM-OS kernel by using the respective trusted API call. Then, a backend must be created on the LLM-OS kernel side. Each sent request will travel through the communication channel, the backend and finally land in the LLM-OS kernel. The generated answer will follow the opposite route of the requests.

4 Design

The LLM-OS project is designed with the three goals in mind. Inference on edge devices must be efficient, trustworthy, and adaptable enough to address application needs. To achieve the aforementioned goals, LLM-OS proposes the components illustrated on Figure 4.1. This thesis focuses on the LLM-OS Kernel, the host-side example client, and the communication mechanism between them. We also evaluate GPU usage possibilities and use a GPU in our implementation. However, the GPU's trustworthiness is outside the scope of this thesis.

In the following sections, we will go through the design of the components that are within the scope of our thesis.

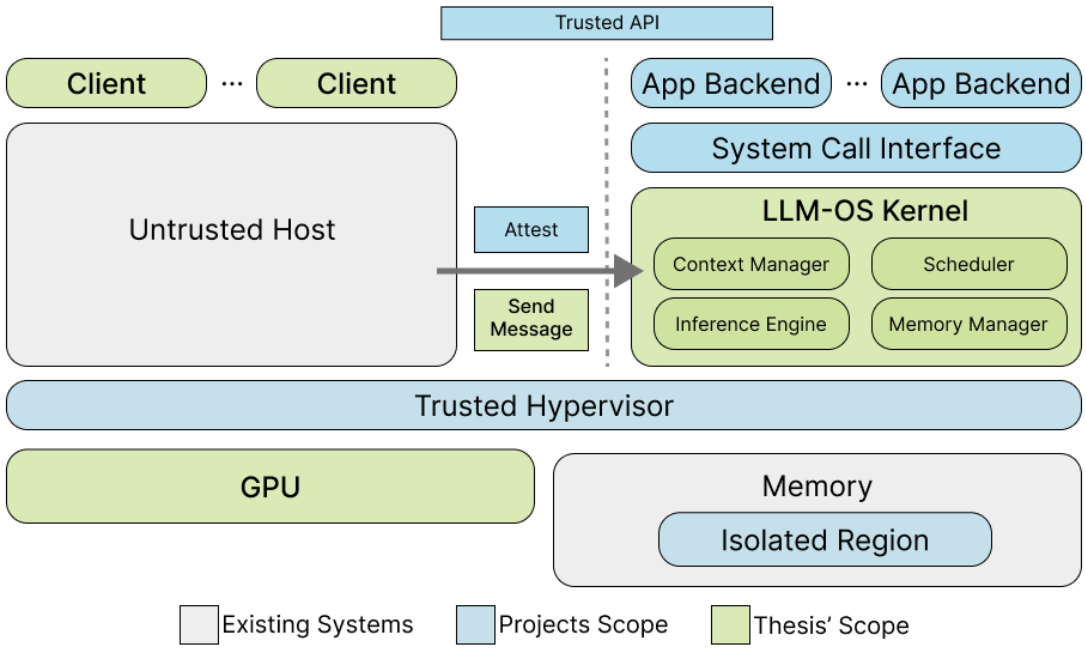

Figure 4.1: Design and implementation focus of this thesis.

4.1 Untrusted Host OS

The untrusted host OS runs possibly multiple clients that require an inference service. To provide the LLM-OS kernel with a realistic environment during its development, we have designed example clients that run on the host OS. These clients make periodic inference requests and wait for an answer. To simulate different situations, clients also have some start-time parameters. For example, it is possible to set the frequency or total number of requests, their textual content, priority, and optional LoRA adapter. It is also possible to dynamically request a throughput limit on requests, which the LLM-OS kernel will enforce.

When a client wants to send a request, it gathers the information taken from its start-up parameters. A client request contains the following fields:

- The user-given text that needs to be answered or continued
- How many characters to generate
- The unique ID of the client
- The priority of the request
- Optional LoRA adapter ID
- Optional fixed or dynamic throughput limit

With all the fields known, the client sends the request to the LLM-OS Kernel via the inter-VM communication mechanism and waits for an answer. Any answer received will be printed out directly by the client.

4.2 Host/LLM-OS Kernel Communication

The communication mechanism between the untrusted host and LLM-OS kernel is the only connection between these two isolated VMs. In our design, the untrusted host-side client uses this communication channel to write its inference requests. On the other side, the LLM-OS kernel, reads all requests written to the channel and processes them. The end result will be written back to the same channel for the client to read. When the communication is complete, the request will be deleted from the channel so that the same space can be resused for a different request.

A special case for the communication between the LLM-OS kernel and a client is the **shutdown** request. When a client sends it, the inference servers gracefully stops its execution. This feature is mostly designed for the testing environments and in future work will most likely not be a part of the trusted API.

Our main expectations from the communication channel is its performance and scalability, as it may be used by multiple host-side processes that require fast answers from the LLM model. Therefore, the channel must be able to work concurrently with multiple clients without becoming the bottleneck of the system. Additionally, any request, or an answer generated from a request, must reach the client as soon as possible. An assumption we did in our design is is having a hypervisor to manage the communication between the two sides.

4.3 LLM-OS Kernel

In our design, we deploy the LLM-OS kernel as a Linux instance on top of the hypervisor. However, as a part of this thesis, we inspected the possibility of deploying LLM-OS kernel as a Unikernel running on a hypervisor. The main goal of such an approach is better performance and a reduced attack surface that improves the overall security. Because of our specific requirements and general limitations of Unikernels, we used a regular Linux kernel for this work. We discuss this choice more detail in Chapter 5.

As seen in Figure 4.1, our server consists of multiple components, all compiled into a single application. In the following subsections, we will explain the design of each element in detail.

4.3.1 Scheduler

Our scheduler design focuses on resource efficiency. It uses a priority-based scheduling mechanism to keep the applications that the user deemed important more interactive. This is achieved by using an array of lists for the requests. As illustrated in Figure 4.2, each array represents a priority level. Inside a list of a specific priority, requests share the same importance and are scheduled via round-robin. However, the scheduler itself always uses the highest non-empty list. Only when the current highest priority list is empty, it will inspect the lower priority lists. This means any urgent incoming request can interrupt the actively processed request and take precedence.

From the resource efficiency perspective, our scheduler design can set a throughput limit on inference according to the user's specification. This allows host processes to slow the inference when the host requires more resources. Additionally, by default, the scheduler is run after every 10 token generations. When the scheduler schedules a request different from the actively processed one, a context switch is necessary. The context manager handles the details of the context switch. However, as the context switch requires saving the current workload and loading the new one, it is a costly operation. In some cases, constant context switches after 10 tokens might be unnecessary and energy-consuming. To mitigate the possible unnecessary context switches, the

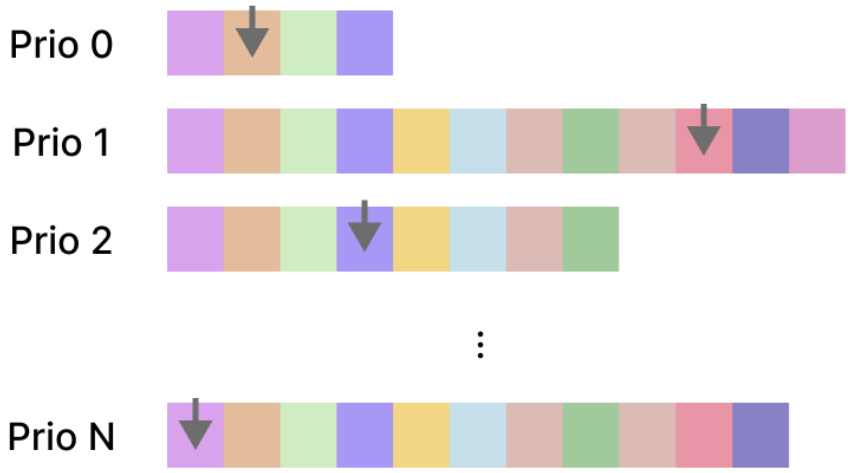

Figure 4.2: Simplified illustration of our scheduler implementation. Scheduler has multiple request lists, each representing a different priority. Each box represents a request, and the arrow represents the round-robin order of the priority level. Any new request will be placed on the respective list by its priority. The scheduler will always use the highest-priority non-empty list. Inside a specific priority level, the next request is selected by the list's own round robin order.

scheduler can set the frequency of new schedules by letting the scheduler run only after a specified number of tokens are generated, effectively reducing the number of context switches.

4.3.2 Context Manager

When a request is scheduled, it needs a context. This context includes the user's tokens, their KV cache, the user's choice of LoRA adapter, and some metadata to keep track of requests execution. If the request has never been scheduled, the context manager lets the inference engine create a new context. Otherwise, the context is either loaded or reconstructed. An essential responsibility of the context manager is deciding how to handle context in case of request switches.

In our design, we consider two handling possibilities. While evaluating the possibilities, we focus on the KV matrices as the rest of the context is memory-wise small and therefore cheap to reconstruct or save/load. The first handling option does not use any KV cache; consequently, it does not keep any KV matrices in memory. In such a case, when a new request comes in, we delete the old request's KV cache and run *prefill* with

all tokens of the new request. As the prefill stage is fast, this can be cheap when the number of tokens is low. The result is a new token and KV matrices. All the following decode steps use the generated KV matrices until the scheduler decides to evict the request for another one. The most significant disadvantage of not using a KV cache is that when the requests grow longer, re-computation of KV matrices becomes more and more expensive. The second option is using the KV cache. In this case, before switching requests, we save the current KV cache, and then load the incoming request's cache. Unfortunately, the loading and saving are slow as they mean transporting a relatively large amount of data, which adds latency. However, when the re-computation becomes expensive in the presence of long requests with many tokens, saving and loading the KV cache becomes faster.

In our design, we decided to keep both approaches as each has advantages depending on the specific situation. By analysing the request, our context manager decides which KV handling policy to use. When a request grows to a point that the KV re-computation starts to slow down, the context manager can change the request's policy dynamically. Regardless of the context policy, context manager removes any LoRAs before a context switch, and applies the LoRA of the new request before the inference.

4.3.3 Inference Engine

Inference engines or LLM serving systems are libraries that combine multiple LLM optimizations and provide inference on a wide variety of hardware. Developing an inference engine is a considerable engineering effort as it must support many different CPUs, GPUs, and TPUs. Additionally, with the rapid advances in the field, inference engines must follow, evaluate, and adopt new optimizations to not left behind. All this increases the development effort and complexity of the implementation. Llama.cpp [12], an inference engine, has nearly 300.000 lines of code. vLLM [23], on the other hand, has more than 400.000 lines of code and almost 2000 open issues on GitHub. Because of the high complexity of inference engines, we decided to evaluate existing solutions and fork the most plausible one.

Our expectations from an inference engine are its support for various LLM inference optimization techniques, Linux x86_64 and ARM support, the possibility to run inference on GPUs and use LoRA adapters. As a part of this thesis, we inspected LiteRT [13], vLLM, and llama.cpp. Ultimately, we used llama.cpp for our implementation since it fulfills our requirements and has a relatively simple and light codebase compared to the other options. Details of this decision are discussed in Chapter 5.

4.3.4 Memory Manager

The memory manager is responsible for the memory management of all incoming inference requests. It allocates memory for each request's text, argumetns and optionally its context, created by the context manager. Additionally, if the context manager sets the KV cache policy on, the memory manager is also responsible of allocating space for the KV cache. This space is used to save the KV cache when the scheduler decides to change the active request. When a request is scheduled again, the saved KV cache will be loaded. After the request it completly processed, all allocated memory is freed by the memory manager. In the future, the LLM-OS project plans to use KV cache optimizations like quantization or sharing to reduce the total memory usage.

4.4 GPU

GPUs, with their highly parallel architecture, suit LLM computations very well. They provide overall higher throughput and lower latency than CPUs in inference tasks [24].

As the LLM-OS project expects to share the GPU between the untrusted host and the LLM-OS kernel, model security from the GPU perspective becomes important. As part of future work, LLM-OS will implement a trustworthy GPU inference mechanism.

In our work's scope, we required a virtualized GPU that was usable by our LLM-OS kernel, with the possibility of sharing it with the untrusted host in the future. The trustworthiness of the GPU and its effect on model protection is out of the scope of our work. However, we kept these future directions in mind while evaluating virtualization options.

There are multiple options to virtualise GPUs. API remoting, also called split driver model, virtualizes the GPU at higher levels in the GPU execution stack, skipping the drivers [15]. The fundamental idea is to catch any GPU calls inside the guest OS and forward them to the host with a dedicated GPU. One drawback of the approach is that the added indirection affects the performance. However, it comes with a much more significant concern, security [31]. In our design, we cannot dedicate the GPU to the host as it is untrusted. If we dedicate it to the LLM-OS kernel, then all host-side user processes must send their GPU calls to the LLM-OS kernel. However, such an architecture would force us to place complex GPU drivers and management software into the LLM-OS kernel, increasing the attack surface. Another option is emulation. Conceptually, emulation is the safest virtualisation as it does not depend on physical hardware. However, the same trait causes an unacceptable performance hit [31]. Direct Device Assignment, or PCI Passthrough, dedicates the GPU to a single virtual machine. The main advantage is the performance, and the disadvantage is that by default, no other VM can use it concurrently. Nevertheless, in our testing with QEMU/KVM,

we have noticed that it is possible to hot(un)plug the GPU between VMs, allowing sequential usage by both sides. From a security perspective, these switches can result in residual data leakage. A possible solution is cleaning the GPU memory before any switches. The last virtualisation option is Direct Device Assignment with SR-IOV. In this approach, the GPU introduces itself as many smaller devices. Hypervisor then can assign the small GPU partitions to different VMs, leading to full GPU sharing with hardware-supported isolation between them. The biggest issue is the lack of such support by GPUs, especially on edge GPUs, rendering this option not applicable for us.

In this work, we decided to use Direct Device Assignment for the first prototype, as it provides high performance, wide adoption, and the possibility to hot(un)plug GPUs between the host and LLM-OS kernel.

5 Implementation

We realized our design on an x86_64 server with NVIDIA A40 GPU. Additionally, we tested the implementation on an ARM-based NVIDIA Jetson AGX Orin with an integrated NVIDIA GPU.

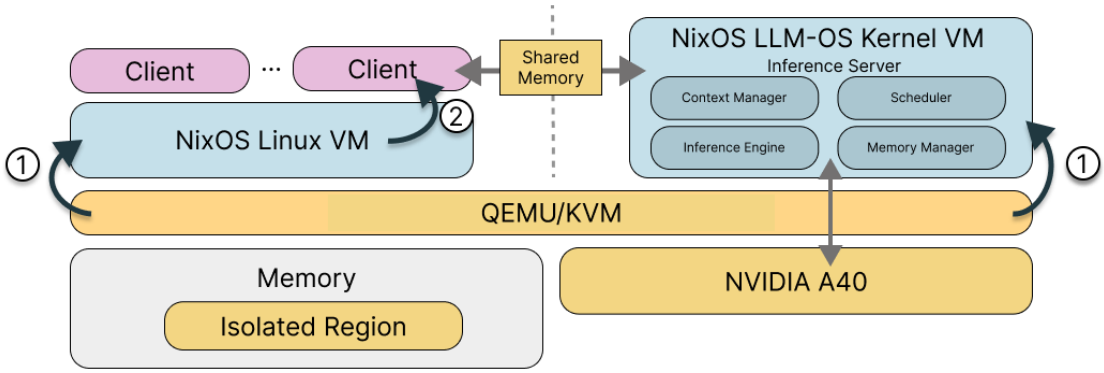

Figure 5.1: The realization of our design. We have a NixOS Linux instance and our LLM-OS kernel running side by side. Both VMs are launched via QEMU, and the shared memory is allocated and configured to be accessible from both VMs. The GPU is directly assigned to the LLM-OS kernel with QEMU/KVM's PCI passthrough device. The host side can additionally launch clients that require inference service.

As illustrated in Figure 5.1, we use QEMU/KVM to launch two VMs, both sharing the same kernel version and configuration. These two VMs are entirely isolated except for the communication channel between them. Their only difference is that the LLM-OS kernel has access to the GPU. QEMU handles the GPU passthrough and the shared memory between the two VMs. Inside the host VM, we run multiple clients, sending inference requests to the LLM-OS kernel via the shared memory. On the LLM-OS kernel side, our custom inference server consists of smaller components, which we will discuss in detail in the following sections. This inference server reads the shared memory, processes the request, optionally using the GPU, and writes the result back to the shared memory. Lastly, the client reads the result and either continues sending

more requests or stops its execution.

To explain the implementation in detail, we will go through the relevant components for the rest of the chapter.

5.1 QEMU/KVM Hypervisor

To allow GPU passthrough and create shared memory, QEMU requires some preparation and must be started with specific parameters.

For GPU passthrough to work in a specific VM, the CPU and the motherboard must support virtualization and IOMMU [2]. When all requirements are met and IOMMU is active, the GPU must be detached from any drivers using it. The detachment can be done by writing the GPU ID to the respective unbind file. To allow a VM to actually use the GPU, the ID of the GPU must be assigned to the VFIO-PCI driver. This is again achieved by writing GPU information to the VFIO-PCIs respective file. When this preparation is done, any VM started with the parameter *-device vfio-pci, host=GPU_ID,id=testGPU* can be the sole user of the GPU.

While working on GPU passthrough, we noticed the possibility of switching a GPU between two running VMs. This is possible by removing the GPU from the first VM by running *drive_del device* via the QEMU monitor. Execution of this command frees the GPU from its VM. The freed GPU can be attached to a new VM using the opposite command *drive_add*. It is possible to automate this using QEMU Machine Protocol (QMP) [36], which we left as a future work.

Compared to GPU passhtrough, creating a shared memory is more straightforward. The QEMU device *ivshmem-plain* provides us with the required functionality. With *ivshmem-plain*, we can create a shared memory, accessible by the untrusted host, and the LLM-OS kernel. For our prototype, we used a 32MB memory region. However, it is possible to set it to any size. It is worth noting that the *ivshmem-plain* device only supports machines with a PCI bus. Since our x86_64 machines have a PCI bus, *ivshmem-plain* works for our use case. QEMU also provides *ivshmem-flat* device to support machines that lack a PCI bus.

The created shared memory region is only accessible by the hypervisor and VMs with that specific memory area in their configuration. However, the host-side processes can currently see other processes' requests, which should be addressed in future work.

All in all, the QEMU parameters we use to configure the shared memory and the GPU passthrough are the following:

- *-enable-kvm* Allows us to use KVM hypervisor.

- *-cpu host* Used for selecting a specific CPU architecture to emulate. As we do not want to emulate a different architecture, we use the *host* option.
- *-device vfio-pci,host=ca:00.0* Lets QEMU use a virtual device allowing GPU passthrough. The host ID belongs to the real GPU.
- *-device ivshmem-plain,memdev=shm,bus=pci.0,addr=0x12,master=on* Adds a virtual shared memory device to the VM. Inside the VM, the shared memory is represented as a file.
- *-object memory-backend-file,size=32M,share=on,mem-path=/dev/shm/shm,id=shm* Specifices the backend of the shared memory device. In this case, the shared memory is 32 MB large. The hypervisor can access the shared memory at the defined *mem_path*.
- *-smp 32,sockets=1,cores=32,threads=1,maxcpus=32* Defines the number of cores that the VM can use.
- *-m 64G* The memory assigned to the VM.

5.2 Untrusted Host

In our implementation, the untrusted host is an x86_64 NixOS instance with the same kernel version and OS configuration as the LLM-OS kernel. However, the Untrusted host cannot access the GPU, as the GPU is dedicated to the LLM-OS kernel.

This x86_64 NixOS instance runs our example clients, sending requests to the LLM-OS kernel, simulating a realistic environment. Clients parse the user-given arguments on startup and connect to the shared memory. These arguments can be about the request itself or the client's behavior. An inference request's structure, which is seen in Figure 5.2, consists of the former request parameters of the client. The latter, behavioral parameters, can set the frequency or total number of requests or allow users to dynamically set the throughput limit during runtime via a listener thread.

After handling the arguments and connecting to the shared memory, the client reaches the main loop, which sends a request by writing it to the shared memory and then waits for an answer. From the inference server side, each generated token is directly sent to the client so that the client can see each of them as they are generated. The answer is complete when either the *n_chars_to_gen* is reached or an End Of Generation (EOG) token is generated. Depending on the user-given arguments, the client can continue sending more requests by staying in the main loop, or exit it to end its execution.

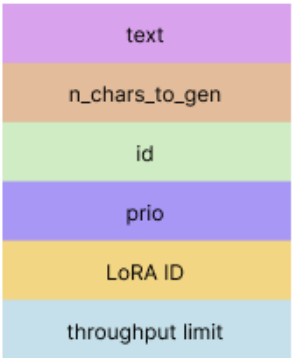

Figure 5.2: Request package of a client

5.3 LLM-OS Kernel

LLM-OS's primary responsibility in our prototype implementation is running our custom inference server. App backend and system call interface are out of the scope of this thesis.

In our implementation, the LLM-OS kernel is deployed as a regular Linux instance. However, as a part of this thesis, we also inspected to compile the LLM-OS kernel into a Unikernel. As the LLM-OS project targets ARM devices, one of our constraints was ARM architecture support. Additionally, as we want to use any existing GPUs on the device, GPU support was another requirement. For this purpose we inspected Nanos [35], Unikraft [19], IncludeOS [4], Gramine [49] and Unikernel Linux (UKL) [38]. In our research, we discovered the lack of GPU support across the Unikernel scene, Nanos being the only exception. Because of the missing GPU support, limited ARM support, and the hardness of compiling applications with many dependencies to a Unikernel, we have dropped the Unikernel from this thesis scope. Instead, the LLM-OS kernel is a regular Linux kernel that runs our custom inference server.

On startup, this inference server initializes the shared memory between the host and the LLM-OS kernel, and loads the base model. Then it launches two threads, one for continuously scanning new requests and one for checking any signals to stop the server. Conversely, the main thread runs in a loop, scheduling detected requests, processing them, and answering them.

By using the configuration parameters, this execution can be customized before runtime to fit specific needs. The relevant parameters are the following:

- **model** Specifies the base model to use for all incoming requests.

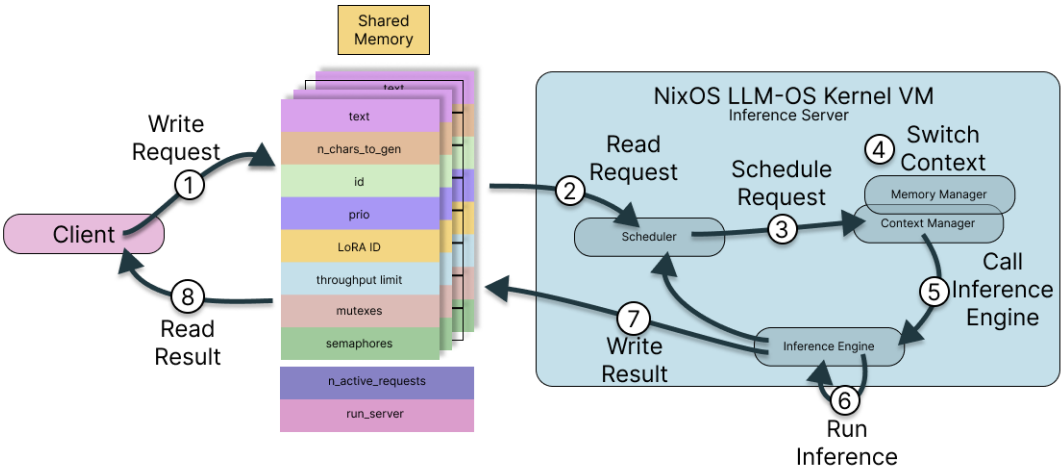

Figure 5.3: The workflow of our implementation. At the first step, the client writes its request to the shared memory. Our inference server reads this request, schedules it. If any memory allocation or context handling is required then the memory manager or the context manager is called. When the context is ready, inference engine is called and inference is executed in a loop. As the last steps, the result is written to the shared memory, where the client can read it.

- **lora** Path to a LoRA model. It is possible to specify multiple LoRAs. Each will receive a unique ID, which the clients can use to specify their preferred LoRA. Future work involves letting the user side give the LoRA model.
- **kv-save** Lets the server use KV cache for all requests. On a context switch, the old KV cache is saved to memory, and the new one is loaded. When *kv-save* is not set, all KV matrices are recomputed on demand.
- **auto-policy** Automatically changes the policy when a request reaches a certain token length. Our implementation uses the 2000 token threshold for policy switch, based on our test results in Figure 6.6.
- **throughput-limit** Limits the throughput for all requests. This can be overridden in request granularity by the client requests.
- **n_threads** Number of CPU threads to use during inference.
- **token-to-gen** Specifies the number of tokens to generate before between the

scheduler calls. This is especially useful to avoid frequent and costly context switches.

Different components inside our inference server use the parameters mentioned above. To explain our exact implementation, we will go through each element. While doing that, we will follow the server's natural control flow seen in Figure 5.3, starting with receiving a request and ending with an answer.

5.3.1 Scheduler

When the inference server detects a new request on the shared memory, the first step is to run the scheduler to decide which request to process next. As mentioned earlier, the scheduler uses a priority-based scheduling, which is demonstrated in Figure 5.4. Therefore, if the new request's priority exceeds all other existing requests, it will be scheduled. If it shares the priority with others, the scheduler will append the new request to the respective priority list, where it will wait for the round-robin iteration to hit it. On the other hand, if there are some requests with higher priority than the new one, the incoming new requests will wait until the higher-priority requests are complete.

When a request is selected, the scheduler calls the main inference loop to generate a specified number of tokens. This number is defined by the parameter **token-to-gen**. The inference server will not call the scheduler until the number of generated tokens reaches this parameter. However, when a request with a higher priority is detected, the current generation loop will be interrupted, and the scheduler will be run immediately.

5.3.2 Context Manager

After the scheduler selects a request, the preparation to enter the main inference loop will be started. The first action before this loop is checking whether the request is new or requires a context switch. If the request is new, a fresh content is generated using our chosen inference engine, llama.cpp. If the request is not new, then the current context must be unloaded, and the new one should be handled. To do that, the context manager checks the active context management policy. If the policy uses the KV cache, the current context is saved, and the new one is loaded from memory. On the other hand, if the policy does not use a KV cache, then the current request's KV matrices are freed. The inference engine will regenerate the KV of the incoming request on the first inference run.

When the context restoration is done, the context manager applies any specified LoRA adapter via the inference engine before running the actual inference. When the

```
def schedule():
    priority = MAX_PRIORITY
    while priority => 0:
        same_priority_requests = requests[priority]
        if same_priority_requests == NULL:
            priority--
            continue

        index_under_same_priority = round_robin_index[priority]
        # wrapping around is left out for readability
        round_robin_index[priority]++
        return same_priority_requests[index_under_same_priority]
    return NULL
```

Figure 5.4: Pseudocode of our scheduler implementation. It starts with higher priority requests and then moves to lower priority ones.

scheduler schedules a different request later, the applied LoRA will be removed by the context manager.

After this configuration, the main inference loop starts generating tokens using the inference engine.

5.3.3 Memory Manager

The memory manager handles the memory required by our inference server. In this prototype, other than allocating memory for the contexts themselves, the primary responsibility of the memory manager shows up during context switches. Depending on the policy, context switches require saving the KV cache and loading it back. To save the KV cache, it is necessary to allocate memory for it. Therefore, before a switch, the memory manager allocates memory in the heap and saves the KV cache there. When a request is completely processed, the memory manager frees the KV cache together with the whole context of the request. In our current implementation, we use a naive KV cache without compression or sharing. For future work, this component constitutes an essential point for further optimizations.

5.3.4 Inference Engine

With the decision of forking an inference engine for our implementation, we inspected LiteRT [13], vLLM, and llama.cpp to find the most suitable option.

LiteRT, the first considered option, is a runtime created for on-device AI. Most of LiteRT's goals align with our goals. According to Google, LiteRT also aims at on-device AI with data privacy, low latency, and power consumption. [13]. However, a major drawback is the lack of accelerator support. Even though LiteRT supports GPUs on Android and iOS, there is no GPU support for Linux. vLLM, in contrast, does have GPU support on Linux and can work on both ARM and x86_64. It allows LoRAs and has many LLM optimizations, such as PagedAttention [23] and speculative decoding [25] already implemented. Even though the relevant optimizations are interesting to us, vLLM focuses on high-throughput serving, and many features are not necessary for our project. The extra features lead to extra complexity and size of the vLLM. This is especially important when the switch to Unikernels is a part of future work. A solution is stripping down vLLM by removing its unneeded components. However, because of the project's complexity, we decided this was not a feasible option. The last considered inference engine is llama.cpp. Like LiteRT, llama.cpp also has the inference on the edge idea. It supports CPU and GPU inference on Linux. Compared to vLLM, llama.cpp has less complexity while still having some inference optimization we can utilize, like batching and quantization. It works on both our required architectures and supports LoRAs. Ultimately, we decided to use llama.cpp since it manages to stay simple enough to ease our possible future modifications while fulfilling our requirements.

In our implementation, the inference engine runs in a loop, demonstrated in Figure 5.5. When a request is scheduled and its context is ready, the main execution loop starts. At the start of the loop, the first inference engine call is *llama_decode*. As a parameter, this call takes the context of the request and a batch object. In our case, the batch object consists of a single sequence belonging to the request actively being processed. However, even though our implementation does not use it, it is also possible to batch multiple requests together with this batch object, improving performance by increased parallelism. Before *llama_decode* returns, it updates the context. To get the actual generated token from the updated context, a *llama_sampler_sample* call is required. This call samples a token from the last *llama_decode* calls result using the provided sampling procedure. Our implementations use the default sampler of llama.cpp. The result is the actual generated token, represented as an integer. We make one last call to our inference engine to translate it into a text format the user can understand. *llama_token_to_piece* converts the token into text, using the model's vocabulary. Our main loop sends the resulting text to the respective client right after the translation. With this last operation, depending on the **token-to-gen** or the existence of any interrupts, the loop either runs again, or returns and leaves the execution to the scheduler.

```
def do_inference(context):
    tokens_generated = 0
    # token-to-gen: number of tokens between scheduler calls
    # n_chars_to_gen: number of characters the user requested
    while tokens_generated < token-to-gen:
        if scheduler_interrupt:
            break

        llama_decode(context, context.batch)
        token = llama_sampler_sample(context, context.sampler)
        token_string = llama_token_to_piece(token)

        send_user(context.userId, token_string)

        if token == EOG or toCharLength(tokens_generated) > n_chars_to_gen:
            break

        tokens_generated++
    return
```

Figure 5.5: Pseudocode of our main inference loop.

5.4 Host/LLM-OS Kernel Communication

To allow clients to send requests and read the answers back, a communication mechanism between the untrusted host Linux and the LLM-OS kernel is required. At the implementation level, different possible techniques are possible to realize this.

The first option is communication through the network stack. This allows us to use well-established network protocols and libraries to create the communication interface. However, network stack-based communication has a significant drawback: performance [40] [22]. TCP/IP is designed for communication between different interconnected physical machines. The implications of this are higher latency and slower throughput caused by the overhead of the TCP/IP protocol stack [40].

The second option is using shared memory. Compared to network-based communication, this approach has lower latency and higher throughput as the memory is mapped and accessed directly. However, to achieve adequate performance and support multiple requests coming from numerous processes of the host, a protocol must be implemented by us. Defining a protocol requires extra development work. Additionally, any implementation details about the shared memory must be hidden from the host-side user to ease the user-side application development.

Since our target edge devices have limited resources, we decided to trade extra development effort for performance and used the shared-memory approach. This allows us to achieve performant communication between the multiple host processes and the LLM-OS kernel. And since the goal is to provide the host-side users with an API, we can hide the low-level shared-memory handling behind the interface functions of the trusted API.

As illustrated in Figure 5.6, our implementation divides our shared memory into multiple slots. Concurrent requests are forced to use separate slots. This allows us to have multiple processes lined up and processed simultaneously. The maximum number of parallel client requests depends on the number of slots. In our implementation, we have a shared memory with 512 slots, effectively meaning the possibility of a maximum of 512 applications accessing the inference server simultaneously.

As visualized in Figure 5.7, when a user-side client sends a request, it first searches for a slot not used by any other client. When a slot is found, the client locks it using the slot's respective **slot** mutex. Any other client who wants to send a request must find a different free slot. When the slot is locked, the client writes information about its request. After all required data is written, the client informs the server via **serverNotifier** semaphore and increments the total number of active requests by updating the **n_active_requests**. In our implementation, when **n_active_requests** is more than zero, the server loops through each slot and checks its **serverNotifier**. When a slot with an active notification is found, its request will be read, and the LLM-OS

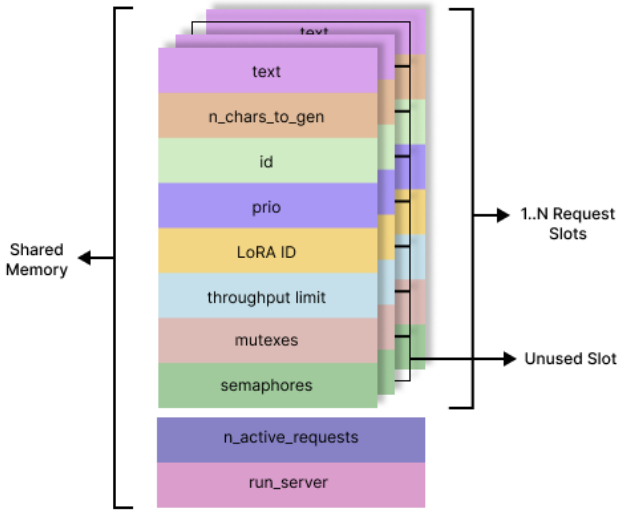

Figure 5.6: The structure of the shared memory. It consists of a fixed number of request slots. At the start, all slots are empty. Each client finds an empty slot and locks it. The slot will be freed for reuse when the inference server answers the request.

kernel will generate tokens according to the specification. The client can dynamically lock the **throughput_mutex** and change the server throughput during this process. Upon the completion of the request, the server will write the result back to the text section of the shared memory, notify the client via **clientNotifier** about the completion, and decrement the number of active requests. On the completion notification, the client will release its slot by unlocking the **slot** mutex so that it can be reused later on.

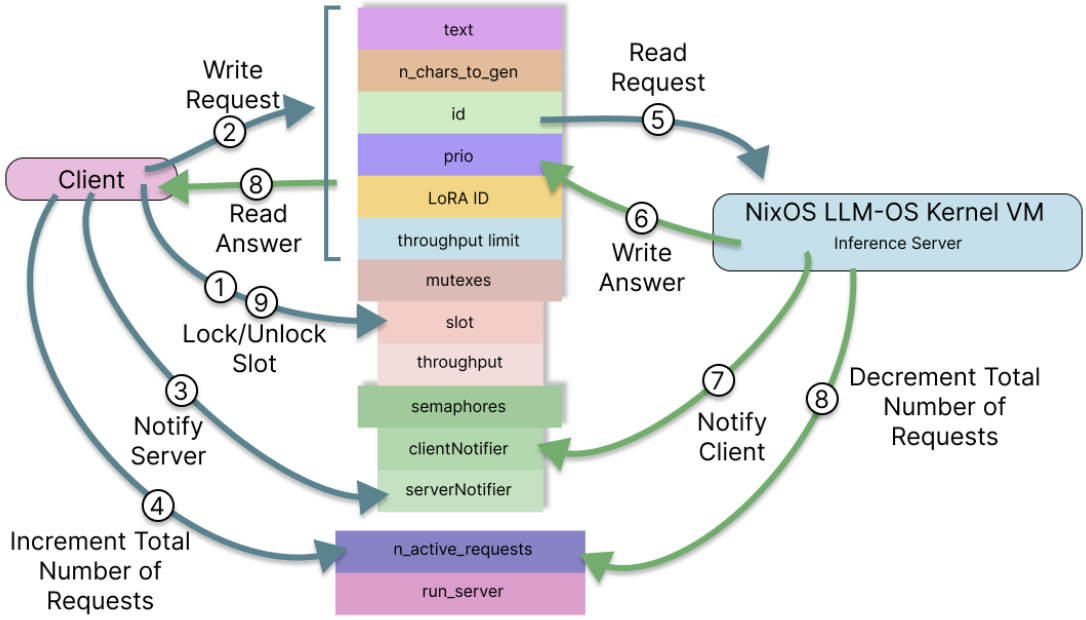

Figure 5.7: Workflow of shared memory based communication. The client must first lock an empty slot using its respective mutex. Then the details about the request must be written. When all data is written, the client must inform the server via the semaphores. On the notification from the semaphores, the LLM-OS kernel find the used slot and read the written request, process it, and write it back. Then the client will be notified so that it can release its slot.

6 Evaluation

The second design goal of LLM-OS is resource efficiency. Therefore, we have measured few metrics to evaluate our implementation. This includes the overhead our approach causes on inference efficiency and our impact on user processes running on the host side. In the following sections we discuss the conducted experiments and their results.

6.1 Experimental Setup

We have evaluated our implementation on an x86_64 Intel Xeon Gold 6326 server with an NVIDIA A40 GPU. All tests use Meta's Llama 3.1 8B model, in a 4-bit quantized form. Additionally, both CPU and GPU inference have been used for comparison. In all VM setups, host side VM has access to 16 cores while the LLM-OS kernel has access to 32. For benchmarking, we utilized llama.cpp's tool, llama-bench, to provide us baselines to compare with our implementation. Both llama-bench and our custom inference server is run with 16 threads in all tests.

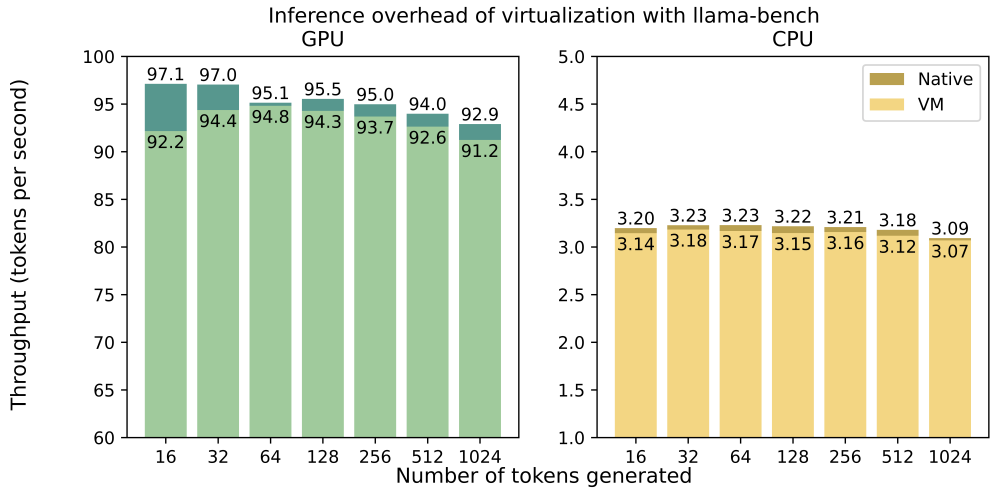

Figure 6.1: The cost of virtualisation on inference throughput. Both on CPU and GPU, the overhead is less than 2%.

6.2 Overhead Analysis

Our implementation uses virtualisation. Therefore, an essential metric for us is the overhead it adds. If, the virtualisation in the first place adds an unacceptable overhead, such as >5%, then it would mean the that our design is not feasible. The first figure, Figure 6.1, showcases the overhead of using virtualisation while running llama-bench. The virtual machine is started with 32 threads and we have let llama.cpp generate 16, 32, 64, 128, 256, 512, and 1024 tokens in separate runs on GPU and CPU using 16 threads. In all runs, the overhead is less than 2%.

Figure 6.2: Overhead of our inference server, serving a single client. Compared to llama-bench running in a VM, CPU inference overhead is around 2%, while GPU overhead is less than 10%. When the cost of virtualisation added up, total CPU and GPU overhead of our implementation is around 3% and 12% respectively.

Another possible root of overhead is our implementation. To evaluate that, we have compared our inference server to llama-bench baselines from a VM and from native execution. The LLMOS setup consists of two virtual machines, one running a client requesting inference service, and the other running our custom inference server. Our server uses the same parameters as the llama-bench. Both use 16 threads and have a context size of 2048 tokens. However, there are some differences. For example, llama-bench internally generates a single random sequence to run inference on it. Our implementation however, receives the input sequence from the client residing in a

different VM. As a part of this experiment, we have a set of predefined user inputs that the client randomly selects and sends. When a request of the client is fully processed, client sends another one immediately.

Figure 6.2, visualises the results. Compared to llama-bench in VM, our implementation causes around 2% slower throughput with CPU, while using the GPU the overhead is less than 10%. The most likely explanation of the added overhead is that our inter-vm communication, scheduling operations during the inference. Additionally, llama-bench creates the context before the experiment. However, we create the context when the request is scheduled.

When the virtualisation cost is added up to this test results, our total overhead is around 3% while using the CPU for inference. In case of using a GPU, total overhead of our work is around 12%.

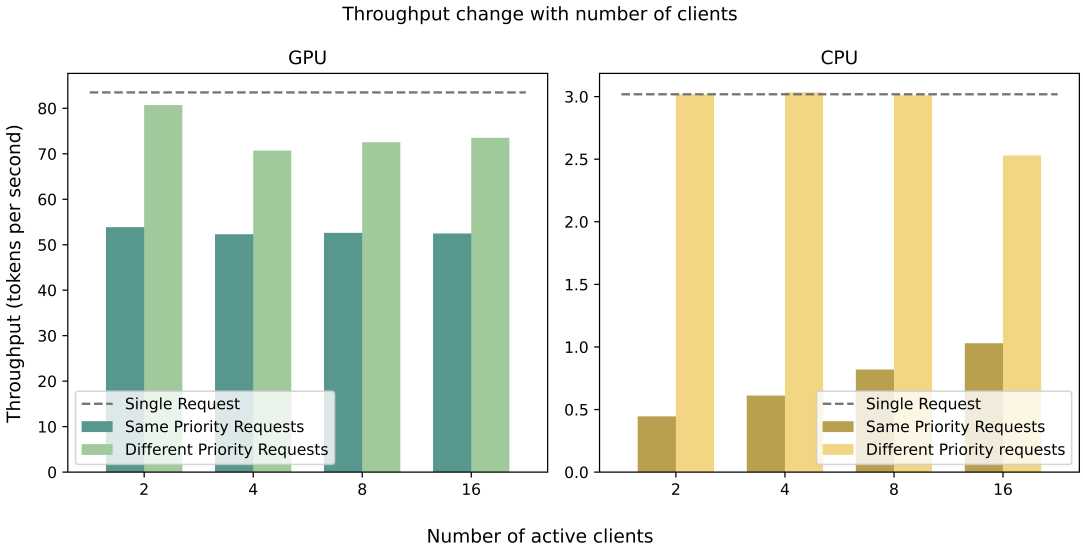

Figure 6.3: Introduction of the second client, and therefore context switches, drops the inference throughput significantly. Additional clients however, does not cause such an effect.

6.3 Scaling with Clients

In a real-world deployment, we expect our inference server to handle multiple requests from multiple clients. Therefore, how well our implementation scales is an important criterion for us . Figure 6.3 showcases the throughput in the presence of numerous

client applications making requests. The test setup consists of 1, 2, 4, 8, or 16 clients on the host side, sending requests. In the first experiment all requests share the same priority. Therefore, our inference engine has to use round-robin to select the following request to process, leading to many context switches. **token-to-gen** is set to 10 tokens, meaning that after generating 10 tokens, a context switch will take place. The test results show a significant throughput drop when introducing a second user application. With a GPU, more applications do not lead to lower throughput. The reason is the initial high cost of expensive context switches. On the CPU, more clients than two lead to relatively higher throughput. We suppose the primary explanation behind this is the shorter context sizes because of the division of the total throughput to multiple clients. This means that even though the total throughput is higher with more clients, each client gets a significantly lower share of throughput. For example, according to our testing displayed on Figure 6.3, in 2-client setups, each client is served with 0.22 tokens per second. In a 16-client setup, this drops to 0.064 tokens per second. This leads to shorter context sizes, as the per-request generation speed is lower and, therefore, the context switch overhead rates are lower. However, we believe future inspection is required to validate this finding.

On the other hand, when all the created requests have a different priority, the inference server can focus on the higher-priority request first, reducing the number of context switches and related performance hit.

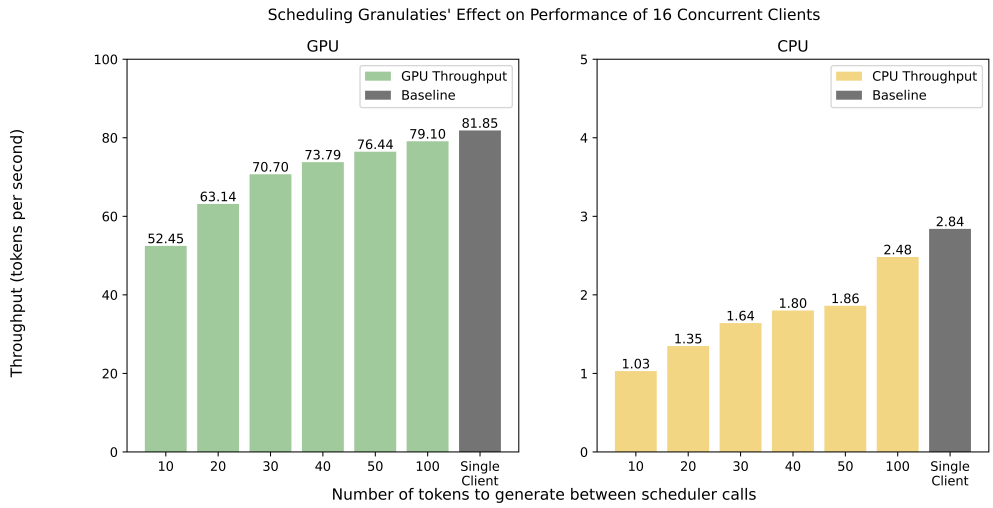

Figure 6.4: Number of generated tokens between scheduler calls effect the inference efficiency directly. More scheduler calls, lead to possibly more context switches adding an overhead.

To understand the cost of context switches, we run a similar experiment with 16 clients sending same priority requests. However, this time, we used **token-to-gen** as a variable. As seen in Figure 6.4, higher **token-to-gen** leads to a higher throughput, both on CPU and GPU. The reason is the lower number of context switches consuming processing time. A draw back of having a high **token-to-gen** is that the requests will need to wait more to get scheduled. Therefore an ideal middle ground between the high throughput and fair scheduling is important.

6.4 Impact on Host OS

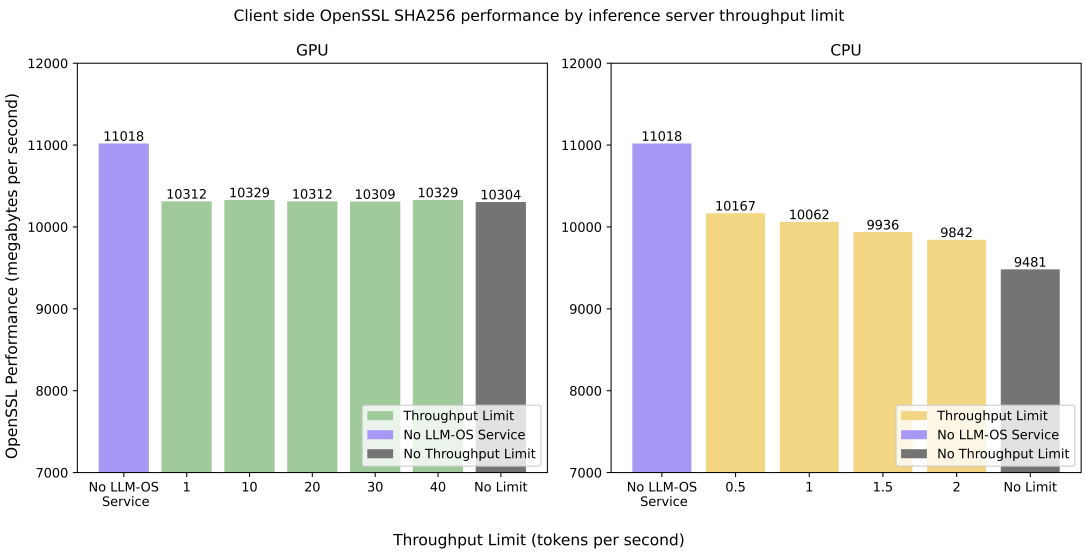

Figure 6.5: Effects of inference server throughput limiting on host side CPU intensive OpenSSL SHA256 benchmark. Lower limit leads to better host performance on CPU. When using a GPU, limiting does not seem to have an effect on host performance.

While serving multiple clients running on the host side, many other applications might be focusing on other tasks that does not make use of our LLM service. Regardless of whether they require an inference service, our server must not cause a resource bottleneck on the host processes. Our server's throughput-limiting function aims to solve this. To test it, we launched a CPU-intensive Phoronix OpenSSL SHA256 benchmark on the host side. At the same time, we let a client is run and make inference requests to the LLM-OS Kernel. Across different experiments, we observed

Phoronix benchmark results while having different inference throughput limits. As seen in Figure 6.5, limiting the throughput on GPU inference does not have a noticebly effect. However, on CPU, using a throughput limit increases the host side performance.

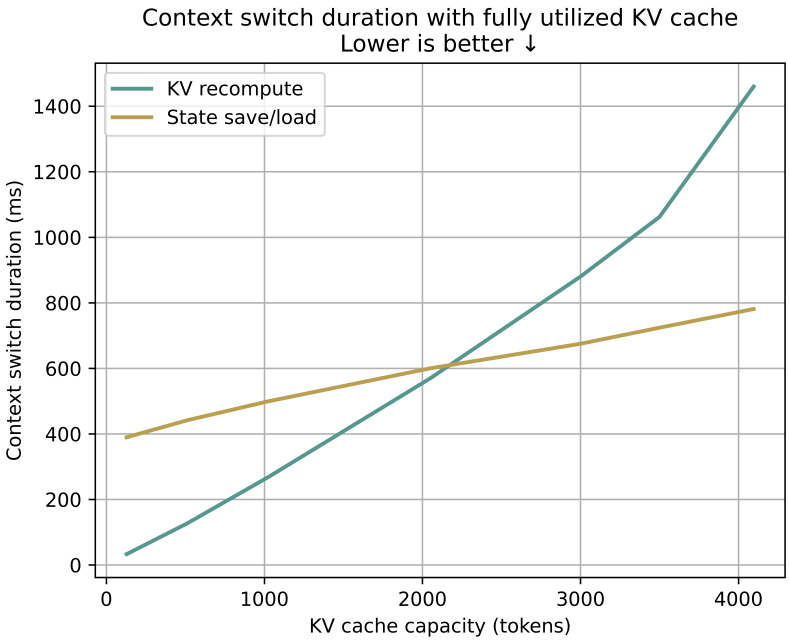

Figure 6.6: Context switch duration for two different context policies. The optimal policy to use changes at the intersection. Experiment done on NVIDIA A40. On edge devices with less computing power, recompute may take longer, and therefore, the intersection will be shifted to the left.

6.5 Context Switch Latency

In addition to the macrobenchmarks we presented, we also used a more specialized context switching benchmark to evaluate the optimal handling policy for KV matrices. The test setup consists of a single client and our inference server. Across multiple experiments, the client sends requests with different lengths, and we measure the context switch time for each request with and without the KV cache. As seen in Figure 6.6, the optimal policy changes at some point. Our test setup uses a powerful NVIDIA A40 GPU, so recomputation is fast relative to less powerful edge devices. The main implication is that we expect recomputation to be infeasible even on lower request lengths with an edge device. However, regardless of the processing power, there will

always be a length at which recomputation is cheaper than saving and loading the KV matrices. Therefore, our implementation uses both options by dynamically setting the best approach depending on the request length.

7 Related Work

Security Against Model Stealing Attacks In case of non-invasive physical attacks that have access to the hardware and software, most solutions for local model protection are based on TEEs [17]. For example, MLCapsule uses Intel SGX to create a formally verified secure environment for the model [14]. To do this they assign the model layers to different enclaves. However, because of the high cost of frequent switches between enclaves, an overhead of up to 2.31 times is observed. DarkneTZ [32], similar to MLCapsule assigns layers to enclaves, however some layers are still executed outside of the secure world, leading security concerns [34]. ASGARD [33] share similar motivation and goals with our project. They argue that the edge inference is better for privacy but it also puts the model in risk. They use a hypervisor based approach that makes us of accelerators via passthrough. In contrast to our work, they lack of focus on resource efficiency and power consumption. Also they target DNNs and not specifically LLMs.
Resource Efficiency Efficiency of LLMs is a highly populated field as any small improvement can lead to great cost savings. While many works focus on a LLM service on cloud deployment, fortunately, most techniques are transferable to edge devices. PagedAttention [23], uses OS like paging mechanism to better utilize the memory resulting 2 to 4x improvement. EdgeMoE [55] makes use of tiered storage structure and dynamic weight loading to achieve the same goal. FlexGen finds efficient patterns to store and access tensors during inference. It also quantizes the model and KV matrices to 4-bit with negligible accuracy loss [44]. Similarly, ZeroQuant [54] reduce precision of LLM models for better efficiency. FlashAttention [6], uses tiling to reduce the number of memory reads and writes, leading faster inference. [56] combines multiple KV cache optimization techniques and decouples LLM context management from the user applications memory management to provide LLM as a system service on mobile devices. H_2O [59] dynamically selects and stores the important tokens. Some other work prune the model to reduce the number of weights and therefore the overall size [46].

All the aforementioned, resource efficiency works are orthogonal to LLM-OS projects model confidentiality goals.
Lora Serving Systems In our implementation, we apply user specified LoRA adapters dynamically to the base model. Such a dynamic usage introduces extra latency on adapter switches. Punica addresses this problem on shared GPU clusters. [5] compresses

multiple LoRA jointly to reduce loading operations by keeping the resulting joint LoRA in memory. S-LoRA [43], uses a unified memory pool to manage adapter weights and KV cache to reduce GPU memory use and reduce fragmentation. Additionally, S-LoRA proposes a new tensor parallelism strategy for batching of LoRA computation to reduce the latency. dLoRA [52] increases serving efficiency and reduces latency by dynamically merging and unmerging LoRA adapters with the base model.

8 Conclusion

Under the larger scope of the LLM-OS project, our work constitutes the first prototype, in which we have designed and implemented an efficient inter-VM communication mechanism that connects two isolated virtual machines, a highly configurable user-side client for simulating realistic user environments. Most importantly, we have developed the core of the LLM-OS Kernel that can prioritize user requests, dynamically select request switching policies for better performance, and run inference with GPU or CPU to concurrently provide multiple clients with an LLM service. Additionally, as a part of our work, we considered future possible directions, such as the feasibility of Unikernels and possible GPU virtualisation options.

Altogether, our resulting implementation can provide an LLM service with less than a 12% overhead while using a GPU and less than 3% while using a CPU-based inference. When serving to multiple host side clients, we can increase our inference performance up to 140% by reducing scheduler frequency, with the trade-off of more waiting time for requests. In low resource conditions, our dynamic throughput limiting functionality can reduce the throughput and increase the host side performance by up to 7%.

The outcome of this thesis enables the future work of the larger LLM-OS project to bring model security to edge LLM deployments while respecting devices' hardware and energy consumption limitations.

9 Future Work

As a part of this thesis, we have created the first prototype of the LLM-OS project, which will serve as a basis for future work. On top of our implementation, the Trusted API must be added. This addition will enable host processes to easily interact with the LLM-OS Kernel via the well-defined API, without worrying about low-level shared memory interactions. At the same time, such a high-level abstraction will restrict processes from reading each other's requests in shared memory. App Frontend/Backend will allow users to pre-/postprocess their requests. In the current implementation, LoRA adapters are provided by the LLM-OS Kernel to users. In the next iterations, users should be able to provide their own LoRA adapters, and the LLM-OS Kernel should protect them.

The first prototype does not make use of many possible KV cache optimizations. Any future work in this regard has the possibility of improvement in performance and efficiency.

From a security point of view, the attestation of the LLM-OS Kernel is an open point that will be addressed by future work. The security of the Trusted GPU in concurrent usage by the host and the LLM-OS Kernel is similarly left to future work. Compiling LLM-OS Kernel into a Unikernel to reduce the attack surface is another security-related future direction of the LLM-OS project.

Lastly, as our implementation is done on x86_64, one of the primary future tasks is complete the migration to ARM to cover a larger edge device space.

List of Figures

1.1 Simplified overview of LLM-OS 3

2.1 Tokenization of Llama decoder. Based on [51] 5
2.2 Autoregressive generation of tokens for a given sequence 6
2.3 Example representation for K and V matrix 7
2.4 Visualisation of Batching 8
2.5 Existing OS structure compared with Unikernel approach 10

3.1 Overview of LLM-OS in detail 11
3.2 Threat model of LLM-OS 14
3.3 Workflow of LLM-OS 15

4.1 Design and implementation focus of this thesis. 16
4.2 Visualisation of the Scheduler 19

5.1 The realization of our design 23
5.2 Request package of a client 26
5.3 The workflow of our implementation 27
5.4 Pseudocode of the Scheduler 29
5.5 Pseudocode of the main inference loop 31
5.6 The structure of the shared memory 33
5.7 Workflow of shared memory based communication 34

6.1 The cost of virtualisation on inference throughput 35
6.2 Overhead of our inference server 36
6.3 Scaling with Clients 37
6.4 Scheduler Frequency's Effect on Throughput 38
6.5 Impact on Host OS 39
6.6 Context Switch Duration 40

Bibliography

[1] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245 [cs.CL].

[2] Arch Linux Wiki contributors. *PCI passthrough via OVMF*. https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF. Accessed: 2025-05-02. n.d.

[3] L. Birch, W. Hackett, S. Trawicki, N. Suri, and P. Garraghan. *Model Leeching: An Extraction Attack Targeting LLMs*. 2023. arXiv: 2309.10544 [cs.LG].

[4] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services." In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[5] R. Brüel-Gabrielsson, J. Zhu, O. Bhardwaj, L. Choshen, K. Greenewald, M. Yurochkin, and J. Solomon. *Compress then Serve: Serving Thousands of LoRA Adapters with Little Overhead*. 2025. arXiv: 2407.00066 [cs.DC].

[6] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG].

[7] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. *PaLM-E: An Embodied Multimodal Language Model*. 2023. arXiv: 2303.03378 [cs.LG].

[8] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui. *GLaM: Efficient Scaling of Language Models with Mixture-of-Experts*. 2022. arXiv: 2112.06905 [cs.CL].

[9] N. El Ioini, A. El Majjodi, D. Hastbacka, T. Cerny, and D. Taibi. "Unikernels Motivations, Benefits and Issues: A Multivocal Literature Review." In: *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*. eSAAM '23. Ludwigsburg, Germany: Association for Computing Machinery, 2023, pp. 39–48. ISBN: 9798400708350. DOI: 10.1145/3624486.3624492.

[10] O. Friha, M. Amine Ferrag, B. Kantarci, B. Cakmak, A. Ozgun, and N. Ghoualmi-Zine. "LLM-Based Edge Intelligence: A Comprehensive Survey on Architectures, Applications, Security and Trustworthiness." In: *IEEE Open Journal of the Communications Society* 5 (2024), pp. 5799–5856. DOI: 10.1109/OJCOMS.2024.3456549.

[11] S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao. *Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs*. 2024. arXiv: 2310.01801 [cs.CL].

[12] G. Gerganov. *llama.cpp*. https://github.com/ggerganov/llama.cpp. Accessed: 2025-05-01. 2023.

[13] Google. *LiteRT*. ai.google.dev/edge/litert.

[14] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. *MLCapsule: Guarded Offline Deployment of Machine Learning as a Service*. 2019. arXiv: 1808.00590 [cs.CR].

[15] C.-H. Hong, I. Spence, and D. S. Nikolopoulos. "GPU Virtualization and Scheduling Methods: A Comprehensive Survey." In: *ACM Comput. Surv.* 50.3 (June 2017). ISSN: 0360-0300. DOI: 10.1145/3068281.

[16] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[17] M. Isakov, V. Gadepally, K. M. Gettings, and M. A. Kinsy. *Survey of Attacks and Defenses on Edge-Deployed Neural Networks*. 2019. arXiv: 1911.11932 [cs.CR].

[18] R. Jin, J. Du, W. Huang, W. Liu, J. Luan, B. Wang, and D. Xiong. *A Comprehensive Evaluation of Quantization Strategies for Large Language Models*. 2024. arXiv: 2402.16775 [cs.CL].

[19] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, J. Nider, M. Rapoport, and C. Lupu. "Unleashing the power of unikernels with unikraft." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR '19. Haifa, Israel: Association for Computing Machinery, 2019, p. 195. ISBN: 9781450367493. DOI: 10.1145/3319647.3325856.

[20] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici. "Unikernels Everywhere: The Case for Elastic CDNs." In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 15–29. ISBN: 9781450349482. DOI: 10.1145/3050748.3050757.

[21] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan. "A Linux in unikernel clothing." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387526.

[22] M. Kurtadikar, A. Patil, P. Toshniwal, and J. Abraham. "An Inter-VM Communication Model Supporting Live Migration." In: *2013 International Conference on Cloud & Ubiquitous Computing & Emerging Technologies*. 2013, pp. 63–68. DOI: 10.1109/CUBE.2013.22.

[23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG].

[24] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann. *On-Device Neural Net Inference with Mobile GPUs*. 2019. arXiv: 1907.01989 [cs.LG].

[25] Y. Leviathan, M. Kalman, and Y. Matias. *Fast Inference from Transformers via Speculative Decoding*. 2023. arXiv: 2211.17192 [cs.LG].

[26] B. Li, Y. Jiang, V. Gadepally, and D. Tiwari. *LLM Inference Serving: Survey of Recent Advances and Opportunities*. 2024. arXiv: 2407.12391 [cs.DC].

[27] H. Li, Y. Li, A. Tian, T. Tang, Z. Xu, X. Chen, N. Hu, W. Dong, Q. Li, and L. Chen. *A Survey on Large Language Model Acceleration based on KV Cache Management*. 2025. arXiv: 2412.19442 [cs.AI].

[28] S. Li, X. Ning, L. Wang, T. Liu, X. Shi, S. Yan, G. Dai, H. Yang, and Y. Wang. *Evaluating Quantized Large Language Models*. 2024. arXiv: 2402.18158 [cs.CL].

[29] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: library operating systems for the cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167.

[30] N. Maslej, L. Fattorini, R. Perrault, V. Parli, A. Reuel, E. Brynjolfsson, J. Etchemendy, K. Ligett, T. Lyons, J. Manyika, J. C. Niebles, Y. Shoham, R. Wald, and J. Clark. *Artificial Intelligence Index Report 2024*. 2024. arXiv: 2405.19522 [cs.AI].

[31] C. Maurice, C. Neumann, O. Heen, and A. Francillon. "Confidentiality Issues on a GPU in a Virtualized Environment." In: *Financial Cryptography and Data Security*. Ed. by N. Christin and R. Safavi-Naini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 119–135. ISBN: 978-3-662-45472-5.

[32] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi. "DarkneTZ: towards model privacy at the edge using trusted execution environments." In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. MobiSys '20. ACM, June 2020, pp. 161–174. DOI: 10.1145/3386901.3388946.

[33] M. Moon, M. Kim, J. Jung, and D. Song. "ASGARD: Protecting On-Device Deep Neural Networks with Virtualization-Based Trusted Execution Environments." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, Feb. 2025. DOI: 10.14722/ndss.2025.240449.

[34] T. Nakai, D. Suzuki, and T. Fujino. "Towards Trained Model Confidentiality and Integrity Using Trusted Execution Environments." In: *Applied Cryptography and Network Security Workshops*. Ed. by J. Zhou, C. M. Ahmed, L. Batina, S. Chattopadhyay, O. Gadyatskaya, C. Jin, J. Lin, E. Losiouk, B. Luo, S. Majumdar, M. Maniatakos, D. Mashima, W. Meng, S. Picek, M. Shimaoka, C. Su, and C. Wang. Cham: Springer International Publishing, 2021, pp. 151–168. ISBN: 978-3-030-81645-2.

[35] NanoVMs. *Nanos*.

[36] QEMU Project. *QEMU Machine Protocol (QMP)*. https://wiki.qemu.org/Documentation/QMP. Accessed: 2025-05-02. n.d.

[37] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori. "KVM: The Linux virtual machine monitor." In: *Proceedings Linux Symposium* 15 (Jan. 2007).

[38] A. Raza, T. Unger, M. Boyd, E. B. Munson, P. Sohal, U. Drepper, R. Jones, D. B. De Oliveira, L. Woodman, R. Mancuso, J. Appavoo, and O. Krieger. "Unikernel Linux (UKL)." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 590–605. ISBN: 9781450394871. DOI: 10.1145/3552326.3587458.

[39] P. G. Recasens, F. Agullo, Y. Zhu, C. Wang, E. K. Lee, O. Tardieu, J. Torres, and J. L. Berral. *Mind the Memory Gap: Unveiling GPU Bottlenecks in Large-Batch LLM Inference*. 2025. arXiv: 2503.08311 [cs.DC].

[40] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao. "Shared-Memory Optimizations for Inter-Virtual-Machine Communication." In: *ACM Comput. Surv.* 48.4 (Feb. 2016). ISSN: 0360-0300. DOI: 10.1145/2847562.

[41] K. Rooney. *OpenAI tops 400 million users despite DeepSeek's emergence*. Feb. 2025.

[42] N. Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. arXiv: 1911.02150 [cs.NE].

[43] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica. *S-LoRA: Serving Thousands of Concurrent LoRA Adapters*. 2024. arXiv: 2311.03285 [cs.LG].

[44] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang. *FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU*. 2023. arXiv: 2303.06865 [cs.LG].

[45] K. Singhal, T. Tu, J. Gottweis, R. Sayres, E. Wulczyn, L. Hou, K. Clark, S. Pfohl, H. Cole-Lewis, D. Neal, M. Schaekermann, A. Wang, M. Amin, S. Lachgar, P. Mansfield, S. Prakash, B. Green, E. Dominowska, B. A. y Arcas, N. Tomasev, Y. Liu, R. Wong, C. Semturs, S. S. Mahdavi, J. Barral, D. Webster, G. S. Corrado, Y. Matias, S. Azizi, A. Karthikesalingam, and V. Natarajan. *Towards Expert-Level Medical Question Answering with Large Language Models*. 2023. arXiv: 2305.09617 [cs.CL].

[46] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter. *A Simple and Effective Pruning Approach for Large Language Models*. 2024. arXiv: 2306.11695 [cs.CL].

[47] M. Sung, P. Olivier, S. Lankes, and B. Ravindran. "Intra-unikernel isolation with Intel memory protection keys." In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 143–156. ISBN: 9781450375542. DOI: 10.1145/3381052.3381326.

[48] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. *Stealing Machine Learning Models via Prediction APIs*. 2016. arXiv: 1609.02943 [cs.CR].

[49] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. "Cooperation and security isolation of library OSes for multi-process applications." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: 10.1145/2592798.2592812.

[50] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang, M. Chowdhury, and M. Zhang. *Efficient Large Language Models: A Survey*. 2024. arXiv: 2312.03863 [cs.CL].

[51] S. Willison. *Llama encoder and decoder*. June 2023.

[52] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin. "dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving." In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 911–927. ISBN: 978-1-939133-40-3.

[53] Y. Xia, J. Kim, Y. Chen, H. Ye, S. Kundu, C. Hao, and N. Talati. *Understanding the Performance and Estimating the Cost of LLM Fine-Tuning*. 2024. arXiv: 2408.04693 [cs.CL].

[54] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He. *ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers*. 2022. arXiv: 2206.01861 [cs.CL].

[55] R. Yi, L. Guo, S. Wei, A. Zhou, S. Wang, and M. Xu. *EdgeMoE: Empowering Sparse Large Language Models on Mobile Devices*. 2025. arXiv: 2308.14352 [cs.LG].

[56] W. Yin, M. Xu, Y. Li, and X. Liu. *LLM as a System Service on Mobile Devices*. 2024. arXiv: 2403.11805 [cs.OS].

[57] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors." In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 313–324. DOI: 10.1109/HPCA.2017.10.

[58] R. Zhang, J. Han, C. Liu, P. Gao, A. Zhou, X. Hu, S. Yan, P. Lu, H. Li, and Y. Qiao. *LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention*. 2024. arXiv: 2303.16199 [cs.CV].

[59] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, Z. Wang, and B. Chen. *H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models*. 2023. arXiv: 2306.14048 [cs.LG].

[60] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen. *A Review on Edge Large Language Models: Design, Execution, and Applications*. 2025. arXiv: 2410.11845 [cs.DC].