Imperial College London

MEng Individual Project - Final Report

Imperial College London

Department of Computing

Fast LLM Inference in Disaggregated Environments

Supervisor:
Prof. Lluís Vilanova

Author:
Roshan Aekbote

Second Marker:
Prof. Paul Kelly

June 13, 2025

Abstract

LLM inference places substantial demands on hardware including CPUs, GPUs, and memory, yet datacenter hardware utilisation remains low due to the rigid allocation of resources in monolithic machine designs. Disaggregated computing, which separates different hardware resources into pools, offers a promising solution to improve utilisation. However, the most popular system for GPU disaggregation, rCUDA, no longer exists, and required extensive source modification. We introduce a system for transparent remote CUDA execution in disaggregated environments, targeting llama.cpp as a representative LLM inference engine. Built on FractOS, a capability-based distributed operating system, our approach intercepts CUDA API calls and forwards them to a remote GPU node, requiring minimal modifications to llama.cpp. We identify that naive remote CUDA suffers from excessive RPC overhead, particularly during the decode phase where a typical LLM may launch thousands of CUDA kernels. To address this, we implement a batched RPC optimisation that exploits CUDA graph capture to reduce the number of RPCs per token from ~ 1000 to fewer than 10 in the single-GPU case. Our evaluation shows that the prefill phase has an overhead as low as 7% for Llama 2 7B and 1% for Llama 2 70B. For the decode phase, we measure overheads as low as 12% for Llama 2 7B and 2% for Llama 2 70B. This demonstrates the favourability of our system for larger models. We also test our system on a multi-client benchmark, where for the moderately-sized Llama 2 13B model we obtain TTFT and TPS overheads of just 1% and 5% respectively. These results demonstrate that efficient remote GPU execution for LLM inference is not only feasible but can achieve performance competitive with local execution, validating the potential of disaggregated computing for AI workloads.

Acknowledgements

I would like to thank the following people for their help with this project:

- Prof. Vilanova, my supervisor, for his excellent technical guidance. Our meetings were incredibly insightful, both for the project itself and for teaching me that I still have much to learn in the field of computer systems.
- Otto White, a PhD student who also provided valuable advice and help, particularly when it comes to the more practical quirks of developing with FractOS. A consistent and welcome presence in almost all of the meetings I had over the course of this FYP.
- Wendy Yang, another PhD student who was responsible for developing the CUDA service before I made my own contributions to it. Her responsiveness and patience when I had questions about how the service worked is greatly appreciated.

Contents

Chapter 1

Introduction

LLMs (Large Language Models) have emerged as one of the most significant developments in artificial intelligence, demonstrating remarkable capabilities across a wide range of tasks from text generation to complex reasoning. The transformer architecture [1] has become the foundation for modern LLMs, with models like the Llama family [2] achieving state-of-the-art performance. However, LLMs are computationally demanding, requiring significant GPU resources for both training and inference.

Meanwhile, research has shown that hardware utilization in datacenters is low [3]. Traditional datacenter designs organize resources into monolithic machines with fixed configurations, leading to inefficient resource allocation when applications have varying hardware requirements. Disaggregation has emerged as a promising solution to this issue. Rather than organizing hardware into fixed machine configurations, disaggregated systems separate different types of resources – CPUs, memory, GPUs, and storage – into dedicated pools that can be allocated according to application needs [4].

However, realising the benefits of disaggregation for GPU-accelerated applications presents significant technical challenges. Most GPU applications, including LLM inference engines, are designed to run on systems where the GPU and CPU are co-located and communicate via high-bandwidth, low-latency interconnects like PCIe. Extending these applications to work in a disaggregated environment requires mechanisms to execute GPU operations remotely with minimal network overheads.

Previous work in this space includes rCUDA [5], a framework that intercepted CUDA runtime API calls and forwarded them to remote GPU nodes. While rCUDA demonstrated the feasibility of remote CUDA execution, the project is no longer available, and its closed-source nature limits the ability to build upon its work. Furthermore, rCUDA required significant modifications to target applications, making it laborious for application programmers.

This project aims to enable efficient remote GPU access for LLM inference in disaggregated environments. We focus specifically on llama.cpp [6], a popular open-source LLM inference engine known for its minimal dependencies and broad hardware support. We build our system on top of FractOS [7], a distributed, capability-based OS designed specifically for disaggregated environments. FractOS provides the necessary infrastructure for secure, efficient communication between disaggregated resources. In particular, its support for RDMA enables high-performance data transfers between remote nodes, which is essential for data-heavy GPU workloads.

We demonstrate that remote CUDA execution for LLM inference is not only feasible but can achieve performance competitive with local execution when properly optimised. Our work provides a foundation for future research in disaggregated LLM inference systems.

Chapter 2

Background

We provide an overview of LLM inference and the internals of llama.cpp, discussing the benefits of distributed LLM inference and existing work in this space. Then, we explain the main parts of the CUDA API that are necessary for this project. We move on to the necessary prerequisites to understand FractOS, which is a distributed OS that is built to take advantage of disaggregation. Finally, we consider existing work in remote CUDA execution, and its link to disaggregated systems.

2.1 Large Language Models

In the most general terms, an LLM is a computational model that has been trained on a very large corpus of text. Text generation has been approached with many different neural network architectures, including RNNs (Recurrent Neural Networks), LSTMs (Long Short-Term Memory networks), and CNNs (Convolutional Neural Networks) [8]. However, the transformer architecture has become nearly synonymous with LLMs for several reasons:

- Transformers have a weak inductive bias compared to other architectures, making them better at generalising as long as enough training data is available [9][10].
- Multi-head attention can be parallelised to take advantage of GPUs [11], unlike RNNs and their variants which require sequential computation.
- The self-attention mechanism can capture long-range dependencies in contrast to RNNs, which suffer from forgetting due to vanishing gradients [12], and CNNs, which have a hard limit according to the size of the convolution kernel. Even LSTMs, which were designed to handle long-range dependencies better than RNNs, still suffer from forgetting [13].

Before being fed to a transformer, natural language must first be tokenised. A token is a sequence of characters, with an associated vector embedding which represents its 'meaning'. The original transformer architecture contained an encoder, which converts a token sequence to a vector, and a decoder, which takes this vector and another token sequence to generate a transformed token sequence. This architecture was originally intended for use in machine translation [1], since the encoder can capture the entire meaning of a passage in the source language, while the decoder can auto-regressively generate tokens in the target language. For text generation based on an initial prompt, however, it is common to discard the encoder. For example, the Llama 2 family of models are decoder-only [2].

2.1.1 Self-attention

The key mechanism used by the transformer architecture is self-attention [1]. Key, Value, and Query matrices are calculated for the input token sequence according to

$$K = XW_K, V = XW_V, Q = XW_Q.$$

$X \in \mathbb{R}^{n \times d}$ is a stack of vectors, where each vector corresponds to an input token. n is the sequence length, and d is the model dimension. W_K, W_V, and W_Q are learned weight matrices. Attention is then calculated as

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

where d_k is the key/query dimension, used as a scaling factor to prevent the operand from becoming too large. Transformers use multi-head attention blocks, which compute attention in parallel with different weight matrices (allowing each head to focus on different features). Blocks of multi-head attention and feed-forward layers are then stacked together to produce the decoder.

2.1.2 KV Caching

The Key and Value entries for a given embedding only depend on the previous embeddings – this is called masked self-attention [1], and it is a consequence of the fact that tokens are generated auto-regressively. Importantly, this means that a token's Key and Value entries will not change once computed. We can store the Key and Value matrices in a KV cache, so it is only necessary for each new token to generate Key and Value entries for itself, greatly reducing the computation required. However, the KV cache grows linearly, and for long contexts this can eclipse the size of the model itself. Techniques such as PagedAttention have been developed to reduce VRAM usage by strategically offloading pages of the KV cache to RAM [14]. Of course, both the KV cache and model itself can be split across multiple GPUs, but care must be taken to minimise communication overheads.

Since the first token must generate the KV cache from scratch, LLM inference can be split into two phases with very different characteristics: the prefill phase (generating the first token) and the decode phase (all subsequent tokens). The differences between these phases are summarised in fig. 2.1.

During the prefill phase, the input token sequence must first be split into batches. Larger batches are generally faster to process due to highly efficient GPU routines for matrix multiplication, such as NVIDIA's CUBLAS library. However, they also use more GPU memory [15], which can be a problem when the context is also large, since the KV cache scales with context length. Since the sizes of the key and value matrices scale with the number of input tokens, the prefill phase is typically compute-bound, and has high GPU utilisation [16].

The decode phase generates each token using the KV cache from the previous step, updating the cache entry for the new token. Because key and value matrices only need to be calculated for a single token, the computation for this phase is much lighter. Unlike the prefill phase which involves large matrix-matrix multiplications, the decode phase instead mainly consists of matrix-vector multiplications [16]. While these are less computationally intensive, they are also less able to take advantage of highly parallelised GPU routines.

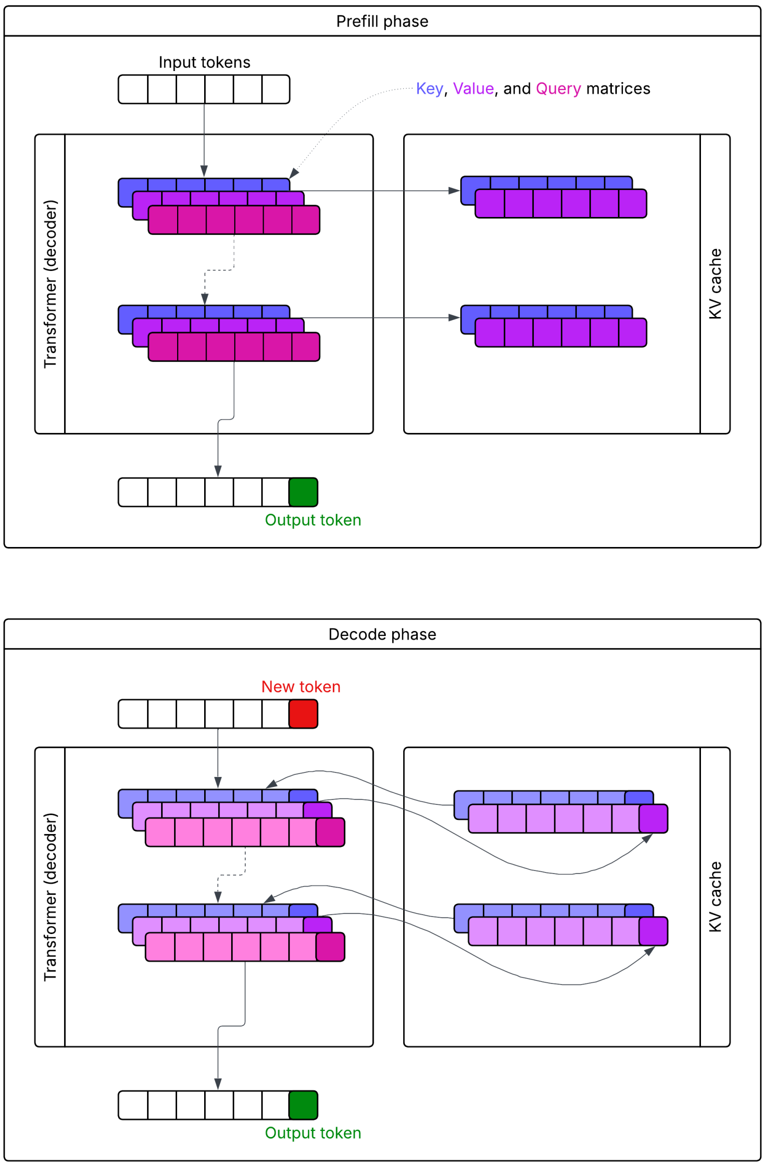


Figure 2.1: Difference between the prefill and decode phases. Note that the various matrices (input, key, value, query) are shown as sequences, where each element is a vector, to better relate them to the input tokens. In reality, these are all just flattened matrices. In the decode phase, we represent pre-computed Key and Value entries in lighter colours.

In practice, the decode phase is memory-bound, as the self-attention mechanism must frequently pull from the KV cache [16][17]. This effect is exacerbated if the KV cache is too large to fit in VRAM and must be offloaded to RAM.

2.1.3 Llama.cpp

Llama.cpp is one of the most popular open-source LLM inference engines, known for supporting a wide range of hardware and having minimal dependencies [6], though we focus only on the CUDA backend. The structure of an inference request is as follows:

- **Model loading** – The model weights are copied from storage to VRAM, with strategies such as memory-mapping to minimise transient RAM usage.
- **Tokenization and embedding** – The user provides a prompt, which is tokenized on the CPU. Tokenisation involves sequential processing and conditional logic, so it is better suited to the CPU than the GPU. The tokens are converted into embeddings, positional embeddings are added, and the tensors are copied to the GPU.
- **Token generation** – Llama.cpp generates each output token sequentially, where the first token is responsible for filling the KV cache.

For this project, we are mainly concerned with the token generation process:

- **Update embedding** – The tensor representing the current token sequence on the GPU is updated based on the previously sampled token (since sampling occurs on the CPU). This host-to-device memory copy is performed asynchronously.
- **Construct computation graph** – Each node of the computation graph produced by llama.cpp represents some primitive operation, such as a memory copy or launching a GPU kernel (e.g. softmax, matrix-matrix multiplication). This graph does not change significantly token-to-token, except for some memory addresses that need to be updated.
- **Execute computation graph** – Llama.cpp iterates over the computation graph, executing the operations asynchronously. The vast majority of these operations are kernel launches, with a minority consisting of CUBLAS GEMM operations (GEneral Matrix Multiplication) and device-to-device memory copies. In the case of multiple GPUs, there is one computation graph per device, and stream/event synchronisation is used to coordinate work.
- **Fetch output token** – A device-to-host memory copy is performed asynchronously to fetch the output logits (the raw per-token probabilities generated by the model).
- **Synchronize** – Llama.cpp waits for all GPU work to finish; note that all operations up to this stage are asynchronous. Once complete, the output logits will be present in RAM.
- **Sampling** – A sampling strategy is applied to the logits. Modern sampling strategies can involve complex conditional logic, so this is done on the CPU.

2.1.4 Inference server

Llama.cpp comes with an server application which can serve inference requests over HTTP [18]. Since this server allows for the most interesting and realistic evaluation of usage patterns, in particular, a scenario where multiple clients simultaneously have conversations with the model, this is the application we chose to work with.

The high-level architecture can be seen in fig. 2.2. The concept of 'slots' is a virtualisation of GPU memory. Each slot maintains an independent KV cache, which means the number of slots is a trade-off; more slots means that more inference requests can be processed in parallel, increasing throughput, but the maximum context length for each request is reduced.

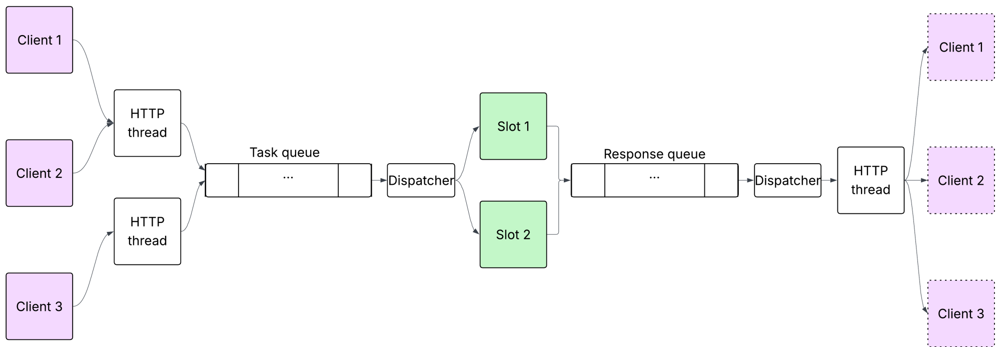


Figure 2.2: The overall architecture of the llama.cpp server. The clients shown on the right, in dotted boxes, should be interpreted as the same clients as on the left (i.e. the figure wraps around). Condition variables are the main synchronisation mechanism used to ensure new elements in the queue are processed as quickly as possible.

A batching strategy must be used to process inference requests from different clients in parallel. One example of such a strategy is static batching [19], which is where a fixed number of requests are batched together, and the batch is considered to be complete only when all requests have received an end-of-sequence token. The size of the batch does not change over its lifetime. This is quite inefficient because in real-world usage, conversations will usually have highly variable lengths. GPU utilisation suffers when long requests are batched together with short ones, since it is not possible to schedule more requests until the longest one has completed.

Instead, the llama.cpp server uses continuous batching [20]. This is also known as iteration-level batching, because the batch is updated for each generated token, rather than at the end of the longest request. In continuous batching, requests will immediately be replaced with other requests when they finish, with the goal to keep the total number of requests in the batch approximately constant. This keeps GPU utilisation high and reduces the average waiting time for requests in the queue.

A final important point is that the llama.cpp server only uses a single thread to call CUDA APIs. This allows us to simplify our design for CUDA interception later on, at the cost of not being as general.

2.1.5 Distributed LLM inference

One of the most notable trends in LLM development has been the rapidly increasing number of model parameters [21]. LLM scaling laws are empirical assertions that show that dataset size, amount of compute, and model size together predict LLM performance with a high degree of accuracy, without regard to other harder-to-quantify qualities such as the fundamental architecture or algorithms.

While smaller models can be run easily on a single GPU, larger models are often too large to fit due to limited VRAM, and must be split among multiple GPUs. Failing that, it may be necessary to split the model across multiple nodes, each with multiple GPUs. Even if it is possible to fit the model onto a single node, there may be undesirable limitations on the size of the KV cache (and therefore the maximum context length), as well as the maximum batch size, making the multi-node setup preferable.

The dominant paradigm for distributed LLM inference is to run instances of inference engines as servers, which communicate with each other to transfer tensors according to their parallelism scheme. For example, llama.cpp provides an RPC backend [22]. An overview is shown in fig. 2.3. This works by running any backend (such as the CUDA backend) as an RPC server, which an RPC client such as the llama.cpp server can connect to. The RPCs exposed by the server include operations like getting/setting tensors and executing serialised computation graphs, consistent with the interface that other backends provide. This is a very high-level approach – roughly five RPCs are called to generate a single token, compared to a thousand or so underlying CUDA operations (these numbers assume a 7B model such as Llama 2 7B).

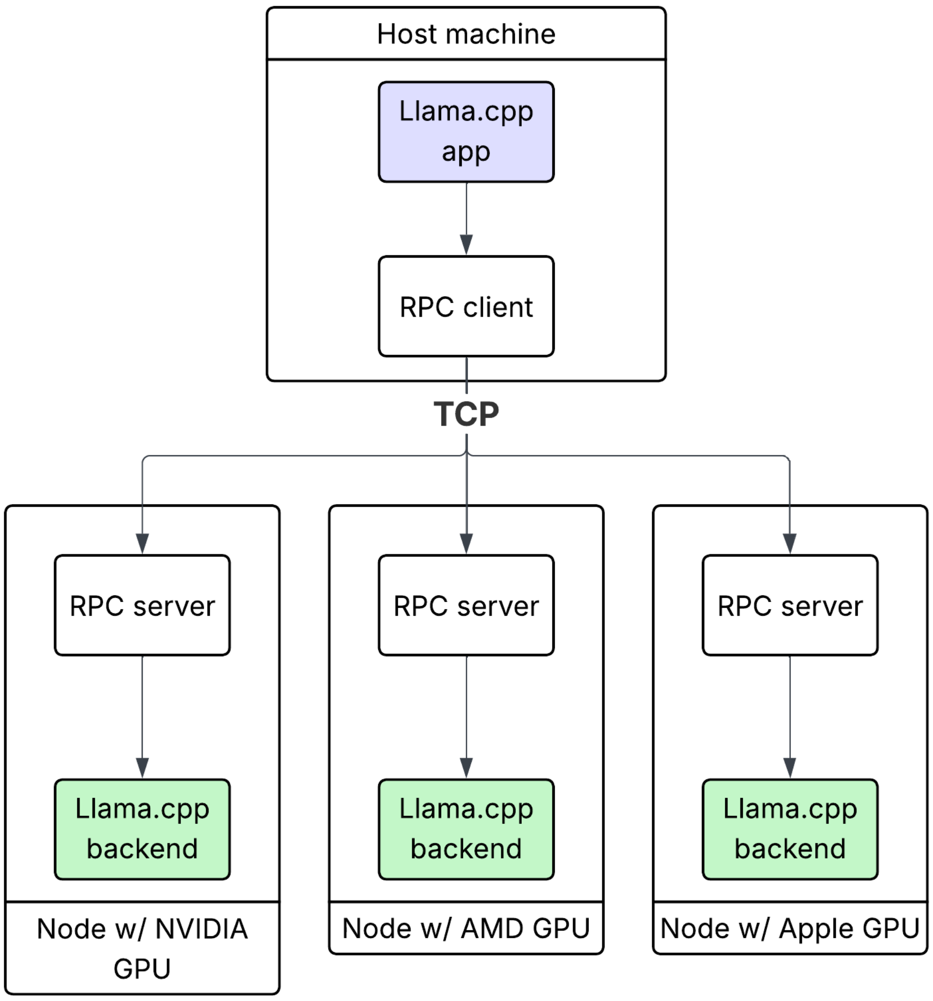


Figure 2.3: The architecture of the llama.cpp RPC backend. A single llama.cpp application can use many different backends, which may be on the same or different machines.

However, llama.cpp's support for distributed inference is relatively immature; for example, the RPC backend is unable to use multiple GPUs on a single node, making it necessary to run one instance of the server per GPU. In practice it is more common to use other inference engines like vLLM for distributed inference. This engine integrates Ray, allowing GPU computation to be scheduled across vLLM instances running in containers on GPU nodes [23]. Each of these 'Ray Actors' loads a slice of the model, either layer-wise or tensor-wise depending on the type of parallelism used. When an inference request is received by the driver process, which is responsible for coordinating communication, it broadcasts the necessary input tensors across the Ray Actors. The Ray Collective Communication Library is used to perform efficient data transfer between nodes [24]. It acts as a wrapper around libraries such as NCCL (NVIDIA Collective Communications Library), allowing for highly-optimised data transfer over NVLink or Infiniband when available [25].

2.2 CUDA

While we focus on the parts of the CUDA API that are needed by llama.cpp, it should be noted that these are all considered foundational components that are used by a wide range of CUDA programs.

2.2.1 Contexts and Memory

A CUDA context is bound to a single CUDA device, and is responsible for keeping track of state like memory, streams, and events. Contexts can be thought of analogously to CPU processes, since each context's objects are isolated from other contexts. However, Unified Virtual Addressing allows all contexts to share a single virtual address space with the host program. This means that it is not necessary to explicitly specify whether a pointer represents host or device memory, or which device that memory resides on. In the case of llama.cpp, there are only a few cases where this behaviour is relied upon - in most cases, the device is explicitly specified, likely for the sake of readability.

2.2.2 Streams and Events

A CUDA stream is an ordered queue of tasks associated with a particular context. Tasks in different streams can execute concurrently, with the interleaving controlled by the GPU scheduler. Streams allow dependencies to be expressed between operations, allowing for better GPU utilisation by, for example, overlapping compute and memory operations.

CUDA events are the primary way to express inter-stream dependencies. An event can record the state of a stream, and then force another stream to wait until the event has completed (which means all the tasks it recorded have completed). Crucially, these two streams do not have to be on the same device, allowing for inter-GPU synchronisation. Llama.cpp supports two main types of multi-GPU parallelism: pipeline parallelism and tensor parallelism.

In pipeline parallelism, the layers of a model are split across GPUs, such that each GPU passes an input through its own layers before passing the result on to the next GPU. This does not improve single-request latency, but has two key benefits: it allows running models that are too large to fit on a single GPU; and multiple requests can be pipelined, increasing throughput. Events are used to force streams on each GPU to wait for the result of the previous one.

On the other hand, tensor parallelism splits the rows of each weight matrix in a model across multiple GPUs. A 'main' GPU is chosen, which aggregates the results of the auxiliary GPUs for each layer. This does improve both latency and throughput, but involves more inter-GPU communication, the speed of which depends on the interconnect used – for example, NVLink is faster than PCIe [26]. Events are used here to force the main GPU to wait for the results of the auxiliary GPUs. Because of the higher volume of inter-GPU data transfers, it is common to use tensor parallelism for GPUs on a single node, and pipeline parallelism between GPUs on different nodes [23].

2.2.3 Kernel launch

A kernel is a CUDA function that can invoked from either the CPU (host code) or the GPU (device code), and is marked with `__global__`. Kernel launch is asynchronous, which means that it only involves setting up the arguments and placing the kernel on a GPU queue. Tasks in these queues are then dispatched by the scheduler. However, if a queue becomes full due to a large number of long-running kernels (or other asynchronous CUDA operations), then any subsequent kernel launches will block on the host until a space becomes available in the queue [27][28]. This means that the overhead of a kernel launch can grow from the time taken to enqueue a task, to the time taken for a kernel to finish execution. Unfortunately, this behaviour is undocumented by NVIDIA, which makes it hard to precisely estimate its effect.

2.2.4 CUDA graphs

Because each CUDA call has an overhead associated with accessing the GPU, CUDA graphs were developed as a way to compress a graph of CUDA operations into a single API call. CUDA graph capture targets a specific stream, so dependencies between streams must be explicitly specified using event APIs.

Once a graph has been captured, it must be 'instantiated' into an executable CUDA graph. The CUDA driver can make optimisations such as coalescing memory accesses at this stage. An executable graph can then be launched on any stream. In practice, it is unlikely that a complex application would execute the exact same sequence of CUDA operations forever – in llama.cpp, the memory addresses of copy operations change for each token. To solve this, CUDA provides APIs for extracting and modifying individual nodes within a CUDA graph, then re-instantiating the executable graph for a lower cost than instantiating it from scratch.

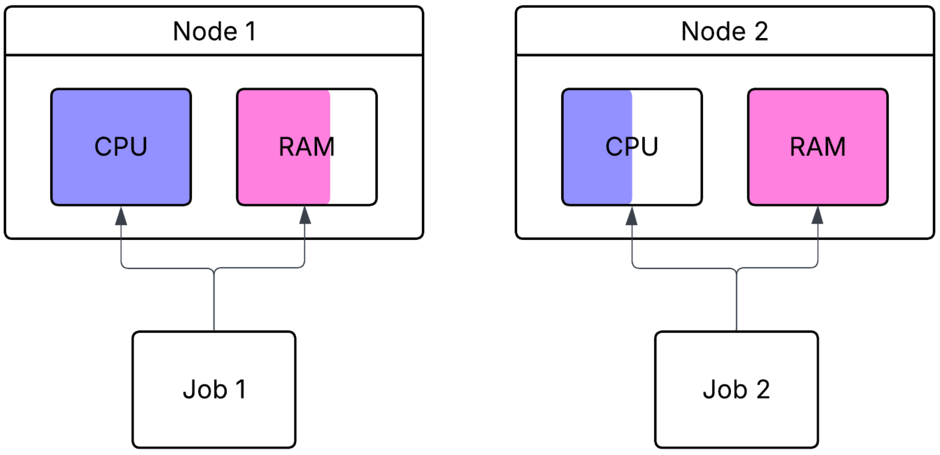
In summary, CUDA graphs provide two main benefits for applications: reduced CUDA API overhead, particularly for long sequences of kernel launches; and optimisations from the CUDA driver, that are only possible when given the entire graph.

2.3 Disaggregation

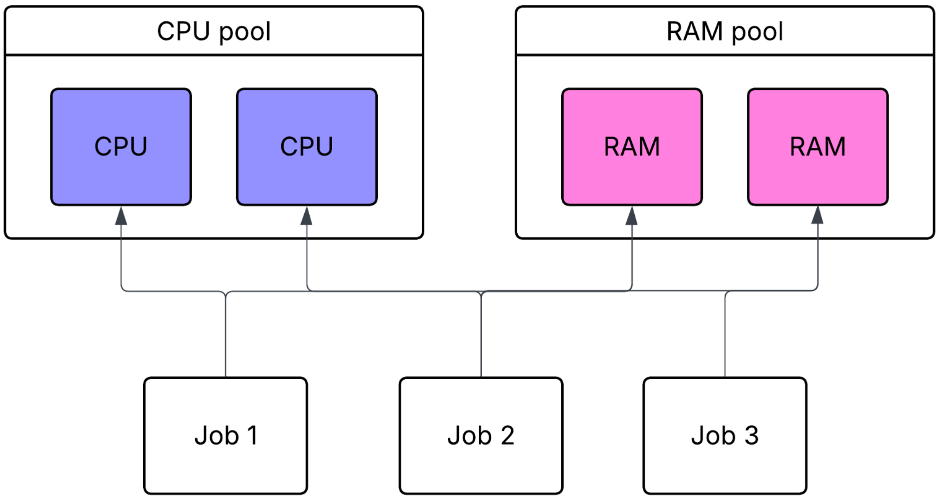
Data centres have predominantly been designed as clusters of machines with similar resource configurations. Jobs are assigned 'slices' of machines as necessary to perform their tasks [4]. This works well when the resource demands of jobs are similar, and do not change much over time. However, the increasing popularity of heterogenous and specialised hardware such as GPUs and FPGAs, for tasks such as AI inference [29], has revealed inefficiencies in this setup:

- The varying resource demands of different jobs makes it difficult to achieve high utilisation [3]. For example, one job may require all of the GPUs but only a fraction of the CPUs on a single machine. This can be viewed as a type of bin-packing problem [30].
- Changes in resource demands are unnecessarily costly – switching to a new type of hardware requires retrofitting a large number of machines, leading to high costs and loss of availability. Upgrading the GPUs in a cluster, for example, also makes the CPUs and storage of those machines unavailable for the duration.

- This design is fragile, since a single faulty hardware component can bring down an entire node, affecting unrelated hardware.



(a) Without disaggregation. Nodes 1 and 2 cannot be used by any other jobs, since they have no free CPU and RAM respectively, despite the fact that they each have a considerable amount of the other resource.



(b) With disaggregation. The free CPU and RAM are now available for another job to use.

Figure 2.4: The utilisation argument for disaggregation.

These issues motivate a design where resources are disaggregated, such that CPUs, memory, GPUs, and other types of hardware are organised into resource pools. Hardware within these pools can then be provisioned according to the needs of the currently running jobs. This argument is visualised in fig. 2.4.

2.3.1 Existing work

To realise the full benefits of disaggregation, significant effort has been dedicated to re-thinking the design of the operating system. Designs which used a simplified form of disaggregation include the Multikernel, which proposed that in order for operating systems to adapt to increasing heterogeneity in CPU cores, they must treat the cores as a kind of distributed system [31]. The Multikernel model assumes no shared memory between the kernels, instead relying purely on message-passing, much as a distributed system would.

In the context of disaggregated data centres, a distributed operating system becomes appealing as it is undesirable to have to store all hardware drivers on a single monolithic OS. Splitting the kernel into separate parts allows each part to have a well-scoped responsibility – this was the idea behind 'monitors' in LegoOS [32]. Monitors together form what LegoOS calls the 'splitkernel' model, where each monitor manages a particular hardware resource. Taking advantage of disaggregation, LegoOS offers advantages such as higher utilisation and fault-tolerance, while maintaining performance comparable with monolithic Linux servers.

Other systems leverage disaggregation specifically for memory. For example, Li et al. found that memory utilisation of cloud applications is generally quite low [33]. By applying disaggregation to memory to create a shared memory pool, it was possible to increase memory utilisation, reducing costs for data centre users and operators.

At a higher level of the stack, software frameworks such as Apache Beam are able to execute data processing pipelines over distributed systems [34]. However, executing complex application logic directly on hardware pools goes against disaggregation, since the disaggregated hardware should purely exist to provide hardware-specific services for other processes. Rather than attempting to realise the benefits of disaggregation through higher-level software layers, research in disaggregated systems has primarily sought a bottom-up approach where the underlying platform is designed with disaggregation in mind.

2.4 Capabilities

A capability is a token of authority which allows the holder to access a resource according to a set of permissions [35]. For example, a memory capability could grant the holder read-only access to a block of memory held by a process. In a capability-based operating system, any access to a resource must be accompanied by the correct capability, resulting in a fine-grained access control model. Capability-based security can be contrasted with ambient authority, which is where a process inherits the permissions of the context it is used in.

One of the vulnerabilities of ambient authority is the 'confused deputy' attack [36]. This is where a process 'tricks' another process into accessing some resource on its behalf. Because ambient authority relies only on the context within which a resource is accessed, a process with permission to a resource can be allowed to access that resource on behalf of another process without that permission. Fundamentally, the operating system is unable to differentiate between a direct resource access and one done on behalf of another process.

By contrast, a capability-based OS would require a process to not only have the name of a particular resource to be able to access it, but also a capability with the necessary permissions. In order for a process to allow another process to access a resource on its behalf, it would need to explicitly delegate the capability, eliminating the risk posed by the confused deputy attack.

There are several key operations that are fundamental to any capability-based OS:

- Creation: a capability allowing access to some object, under some set of permissions, is created.

- Invocation: a capability can be invoked to access the object it refers to.
- Delegation: a capability can be delegated to another process. In the previous example, this step would be necessary for a process to allow another to access a resource on its behalf. This delegated capability is considered a child capability.
- Revocation: a capability (and all of its child capabilities) are removed.

2.4.1 Existing work

The idea of using capabilities for access control has been around in some form since the 1960s [37], with the first successful capability-based operating system widely regarded to be HYDRA [38]. Compared to other designs in the same time period such as the Cambridge CAP Computer [39], HYDRA was the first to propose that capabilities should refer to a more abstract notion of 'object' rather than directly to a segment of memory. This is useful for representing permissions not only to access memory locations, but to make certain requests to other processes. A capability referring to a 'request object' grants the holder the right to call a particular RPC. This is useful in many-core single-node systems, but essential in distributed systems.

As computer hardware moved towards relying on large numbers of CPU cores in order to maintain performance improvements, some researchers investigated whether capability-based operating systems can scale to machines with large numbers of non-coherent, heterogeneous cores. SemperOS was a capability-based operating system which showed, for the first time, that such a system is possible [40]. While capabilities were shown to scale for many-core systems, it remained to be seen whether they could also scale to many-node distributed systems.

2.5 FractOS

FractOS is a distributed, capability-based operating system designed for use in disaggregated environments, offering high performance alongside strong security guarantees [7]. The creators of FractOS envision future data centres as consisting of pools of disaggregated hardware resources, containing co-located CPUs or SmartNICs. Notably, these CPUs are 'wimpy' CPUs, in that they have low single-core performance but are more energy-efficient than 'brawny' CPUs [41]. This makes them suitable for co-location since they ultimately delegate the heavy computation to the specialised hardware, and are only responsible for converting RPCs to hardware instructions.

Almost every operation in FractOS, from the perspective of applications running on it, is asynchronous. Most FractOS APIs return futures, which are typically either stored in some container for later retrieval, associated with a callback, or directly waited on. Asynchronicity is particularly important for FractOS, since its disaggregated nature means that common operations may have to wait significantly longer for network transfers. Returning futures from FractOS API calls gives programmers the option to design their applications in a way to avoid these overheads.

One of the key contributions of FractOS is the ability for different hardware types to directly invoke each other, without going through a central point of control. This allows applications to be designed to avoid redundant network messages, reducing latency and bandwidth usage. However, this feature of FractOS is not as relevant for our project, since we only consider communication between a CPU node, which performs the heavy CPU computation for llama.cpp, and a GPU node, which we offload the model layers and KV cache to. Nonetheless, more complex features such as streaming model weights directly from an SSD into GPUs (bypassing the main CPU) are possible in FractOS. The Future Work chapter gives examples of such features that can make use of the unique features of FractOS.

2.5.1 Capabilities

In FractOS, there are two types of capabilities that are most commonly used:

- **Request capabilities** – These represent the right to invoke an RPC with a given set of arguments. It is common to extend request capabilities by adding arguments, but arguments cannot be taken away.
- **Memory capabilities** – These represent the right to access a region of memory with specific permissions (typically read-only or read-write). They can be diminished to form new memory capabilities with a subset of the original memory region, or a subset of the original permissions.

When an application wants to use another service for the first time, it must get the initial request capability from somewhere – this is the purpose of the GNS (Global Name Service). Services advertise their RPCs on the GNS, providing an entry point for applications. A common pattern is for a client to call an RPC to create an object on a server, and the server responds with request capabilities which allow the client to manipulate and destroy that object (destroying the object revokes all capabilities). Objects will often be able to create other objects, creating a tree representation.

2.5.2 Controllers and Channels

FractOS controllers form part of the TCB (Trusted Computing Base), and take on responsibilities such as storing capabilities – user programs do not hold capabilities directly, but instead hold indices into a capability table, much like Unix file descriptors. Controllers expose a set of syscalls to FractOS processes registered with them, allowing operations such as creating capabilities, diminishing capabilities, and copying memory via RDMA.

For a process to connect to FractOS, it must be associated with a channel, which in turn is associated with a controller. Channels run event loops to handle incoming requests and data, and check if any futures have been fulfilled. A common pattern in FractOS applications is to register a function as a callback, such that the channel will automatically call that function when it detects the associated future has been fulfilled.

2.5.3 Applications and Services

Every process associated with a FractOS controller is fundamentally treated the same, but when creating systems, it is common to draw a distinction between applications and services. While applications have the usual meaning, a 'service' has a special meaning in the context of disaggregation.

Services refer to programs that are attached to pools of disaggregated hardware, and have the responsibility of converting FractOS RPCs into driver commands. For example, FractOS already has services for simple GPU and storage operations. Since it is not possible to directly run these kinds of programs on GPUs or other accelerators, each hardware pool requires a co-located CPU or SmartNIC. While this may seem like a violation of disaggregation, we argue that as long as the services running co-located with these hardware pools solely act as 'network-layer drivers' for RPCs, we can still obtain the benefits of disaggregation.

2.6 rCUDA

It is clear that non-disaggregated systems often suffer from suboptimal utilisation. In the case of GPUs, this is not only a waste of the hardware investment, but also highly energy-inefficient, since GPUs use a significant amount of power even when idle. This was the motivation for rCUDA, a framework for executing CUDA operations on remote GPUs [5].

rCUDA works by intercepting CUDA runtime API calls with a wrapper library, then forwarding these calls over the network to a daemon running on a machine with an NVIDIA GPU. This daemon converts incoming requests to CUDA driver API calls – the driver API was used as it provides finer control, and enables things that are not possible with the runtime API (such as managing kernel modules, which the authors describe as being necessary to get kernel launch interception to work). This system allows a large number of nodes without GPUs share a relatively small number of nodes with GPUs, reducing the total number of GPUs, which improves energy efficiency. The authors describe a few key challenges:

- Kernel launch in CUDA relies on opaque data structures and undocumented functions to go from host code stubs to the actual device code. Therefore, they manually separate host and device code in existing CUDA applications. The device code is compiled and loaded into a module on the CUDA daemon. The host code must be modified to explicitly set up arguments and invoke an RPC, completely stripping away the use of the CUDA API. Needless to say, this process is extremely laborious. The rCUDA authors describe having to modify between 0.0-11.7% of codebases to support rCUDA, which can easily become unmanageable in large codebases.
- Host-to-device and device-to-host memory copies were inefficient due to the need to first copy to RAM on the GPU node, then copy to VRAM. This also violates disaggregation (although the authors did not refer to disaggregation in their paper, we view their work through the lens of this field).

Unfortunately, the rCUDA project no longer exists. The website has been taken down, and the code was never open-source. Details of how they authors chose to implement particularly tricky operations, such as kernel launch, are not available, and any systems that we implement cannot be evaluated against rCUDA. The original paper gives us some limited insight into the challenges they faced, which we can use to inform our own exploration.

Chapter 3

Motivation

Research in disaggregated systems has shown that hardware utilisation in datacenters remains low; for example, Li et al. find that up to 25% of DRAM is unused in Azure datacenters, due to DRAM becoming 'stranded' when CPU cores on machines are fully-allocated [42]. The disaggregation of hardware resources, such as accelerators and memory, is posed as a possible solution. Meanwhile, we find that AI inference can draw on a wide range of hardware, including GPUs, CPUs, TPUs, FPGAs, and ASICs, among others [43][44]. Combined with the increasing importance of AI inference more broadly, we see this application as a particularly interesting target for disaggregation.

rCUDA is the closest piece of existing work to what we are trying to achieve, as it aimed for increased utilization on a pool of NVIDIA GPUs by separating application logic and CUDA computation. However, the project no longer exists, making it difficult to evaluate on modern LLM workloads. Thus, we wish to produce a system which allows CUDA applications to efficiently access remote NVIDIA GPUs in a disaggregated environment.

To narrow the scope of our project to a viable level, and to give us a specific application to use to track the performance of our system, we focus on implementing what is necessary for llama.cpp. This codebase is relatively easy to modify compared to other systems such as vLLM and ExLlamaV2, due to having minimal dependencies.

Fundamentally, disaggregation aims to disaggregate all different types of hardware resources from each other, not just CPUs and GPUs. One could imagine that we could extend our system to use not just remote NVIDIA GPUs, but FPGAs, AMD GPUs, and other kinds of accelerators. Thus, the wide-ranging support that llama.cpp has for different hardware backends (such as the HIP backend for AMD GPUs), with a common interface making it possible to seamlessly switch between them, makes it especially compatible with disaggregation in our view.

With this in mind, the key objectives of our project are as follows:

- Create a system to intercept CUDA API calls produced by llama.cpp, and forward them to a remote service which will actually execute the CUDA operations.
- Minimize any necessary modifications to the llama.cpp source code, with the idea that this system could eventually be extended to support a wide range of CUDA applications.
- Analyse the performance of our system when naively converting local CUDA calls to RPCs, and investigate what are the most meaningful optimisations we can make to minimise this overhead.

- In line with disaggregation, avoid adding additional responsibility to the CUDA service beyond that which is absolutely necessary to convert RPCs to CUDA driver calls.

We considered what distributed platform to build our system on top of – FractOS seemed like a natural choice. It is specifically designed to take advantage of disaggregation, and in the future, additional features such as directly loading model weights from SSD to VRAM could be added (though we do not cover this in our project). FractOS already had support for basic remote GPU operations, making it easier to get started, though note that it did not initially have support for API interception (so applications would have to explicitly use FractOS APIs). It also has a relatively low RPC overhead of about 30us, according to internal micro-benchmarks, which aligns with our goal to minimise overhead.

Chapter 4

Design

Our overall system in shown in fig. 4.1. There are four main components:

- The client, which sends inference requests to the llama.cpp server over HTTP. For evaluation purposes, we refer to this as the benchmark.
- The llama.cpp server, which is part of the llama.cpp repository. The changes to the source code of the llama.cpp server and the CUDA backend are minimal, and are explained in the Implementation chapter.
- The CUDA interceptor. This shared library is injected into the llama.cpp server via `LD_PRELOAD`, allowing it to capture CUDA API calls and replace them with RPCs to the CUDA service.
- The CUDA service. This has a library component, which exposes RPCs used by the interceptor, and a server component, which runs on a remote machine with a GPU and is responsible for executing CUDA operations via the CUDA runtime and driver API.

Essentially, our system transparently replaces local CUDA operations with remote ones by making RPCs to a remote service, similarly to rCUDA. The main difference is that we target a specific application (llama.cpp) rather than implementing the entire CUDA API (except for graphical APIs), as rCUDA did. While not as generally applicable, this allows us to invest more time into optimising our system to minimise the gap between local and remote CUDA execution.

Nonetheless, it should be mentioned that although the CUDA APIs to implement were chosen based on what is necessary to run llama.cpp, we believe we have provided a foundation upon which additional CUDA API support can be readily added to support a wide range of CUDA applications. The main CUDA APIs supported by our system are as follows:

- **Memory allocation** – `cudaMalloc`, `cudaFree`, `cudaMallocHost`, and `cudaFreeHost`.
- **Memory manipulation** – `cudaMemcpyAsync` and `cudaMemsetAsync`.
- **Kernel launch** – `cudaLaunchKernel`.
- **Stream management** – `cudaStreamCreate`, `cudaStreamDestroy`, `cudaStreamSynchronize` and `cudaStreamWaitEvent`.
- **Event management** – `cudaEventCreate`, `cudaEventDestroy`, `cudaEventRecord`, and `cudaEventSynchronize`.

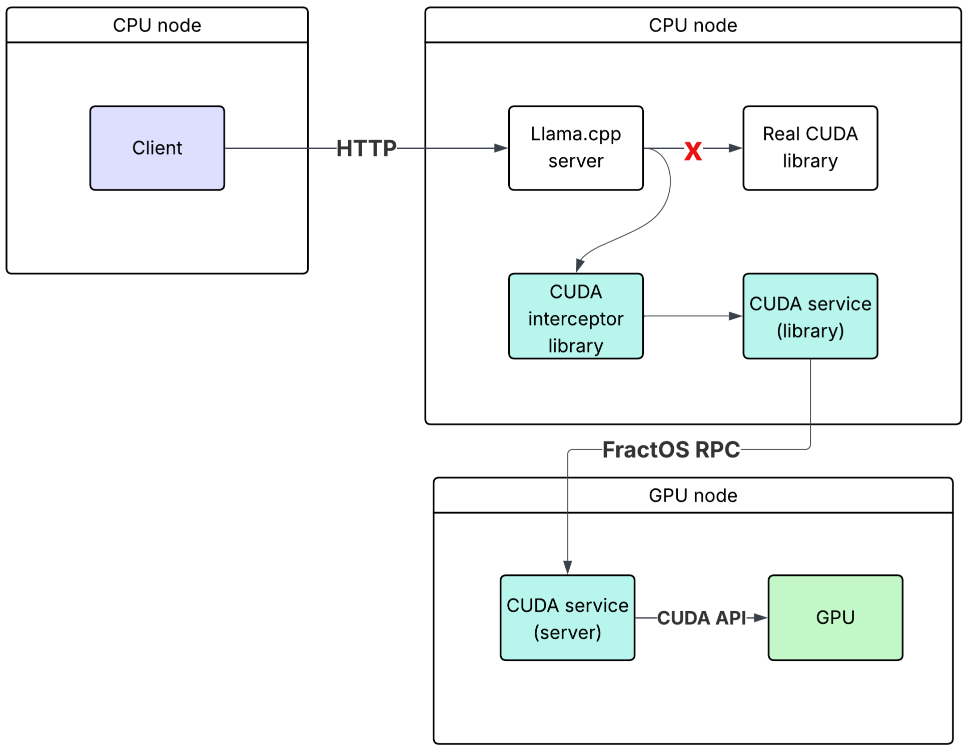


Figure 4.1: The overall system architecture. Our main contributions are shown in turquoise, noting that the CUDA service already existed, but we made significant additions to it.

- **Device management** – `cudaGetDevice`, `cudaSetDevice`, `cudaGetDeviceCount`, and `cudaGetDeviceProperties`.
- **CUDA graphs** – `cudaStreamBeginCapture`, `cudaStreamEndCapture`, `cudaGraphInstantiate`, `cudaGraphLaunch`.
- **CUBLAS functions** – `cublasCreate`, `cublasDestroy`, `cublasGemmEx`, etc.

4.1 CUDA interceptor

The CUDA interceptor captures CUDA runtime and CUBLAS API calls from the llama.cpp server. However, both of these use the CUDA driver API under the hood. Therefore, it would be more general to intercept the underlying CUDA driver API calls. However, there are technical reasons why this is difficult (we elaborate on these in the Implementation chapter), so we opted to leave this extension for future work.

Key aspects of our design of the CUDA interceptor, including our philosophy and assumptions we make on the application using it, are outlined below.

Invisible to the application – We aim for no or minimal modifications needed to the target application for it to work with our CUDA interceptor. One of the key ways we maintain this 'invisibility' is via the use of fake resource handles. Many CUDA APIs rely on handles, which are pointers to opaque data structures, to represent different CUDA objects (such as streams and events). Of course, these pointers become meaningless when referring to a remote machine, so instead we provide 'fake' handles that are generated like unique IDs. These IDs are mapped to objects which can communicate with the CUDA service via synchronous FractOS RPCs.

Note that although the RPCs themselves are synchronous (i.e. we wait for a response, typically either success or error), most of the CUDA operations themselves are asynchronous.

High performance – Of course, applications are not restricted to using CUDA remotely via the CUDA interceptor only; it is possible to directly call RPCs exposed in the library component of the CUDA service. Our project can be seen as an investigation into how efficiently we can perform remote CUDA operations, while not requiring any additional effort from the application programmer. To that end, we spend a significant portion of our efforts to search for both application-specific and general optimisations to our system.

Preference for local handling – In general, we try to do as much work using the local state as possible, since minimising the number of RPCs was found to be the best way to improve performance. An example of this is where possible, we attempt to raise errors caused by invalid arguments to intercepted CUDA APIs without having to perform an RPC. A recurring pattern is to check whether a given resource handle is valid by searching for it in the appropriate map, stored in the local interceptor state.

Caching – We make heavy use of local caches to avoid unnecessary RPC calls. One example is that when we find a function in a server-side CUDA module, we store it locally to prevent having to look it up again. Another example is using the same CUDA graph object by just resetting its state between inference requests, rather than destroying it and creating a new one via a call to the CUDA service.

4.2 CUDA service

The CUDA service existed before this project, but we modified it to support all the CUDA APIs required by llama.cpp. It was already able to support memory allocation, kernel launch, and device/context management; our main contributions are memory copy, stream/event management, CUBLAS APIs, and a limited form of CUDA graphs. The service consists of two parts: the library, which exposes RPCs that are called by the CUDA interceptor; and the server, which handles those RPCs and calls the CUDA API.

4.2.1 Library

Most of the logic in the library is straightforward, and simply packs arguments together to send them to the server using FractOS. A notable exception is CUDA graphs, which involves some more complex state tracking, and this is discussed in the Implementation chapter.

4.2.2 Server

The server component of the CUDA service runs an event loop which waits for incoming requests, then handles them according to their request capability. While CUDA runtime API functions can, for the most part, be fairly easily translated to driver API functions, this is not the case for the CUBLAS API. Therefore, we violate disaggregation slightly be directly calling CUBLAS functions on the server. Note that this is the only instance where non-driver API functions are used on the server.

The server keeps track of a tree of objects. This tree is particularly important for CUDA operations which require accessing multiple CUDA objects, such as `cudaStreamWaitEvent`, since it is possible to traverse up the tree to a common ancestor, then traverse downwards to find the desired object.

Chapter 5

Implementation

We break up the implementation by the part of the CUDA API that was targetted, which closely follows the actual development process.

5.1 Memory allocation and access

In FractOS, memory is primarily accessed and transferred using capabilities, and we follow this pattern for our CUDA interceptor. A call to `cudaMalloc` is converted to an RPC to the CUDA service, which returns a memory object. This object contains a memory capability which can be diminished as necessary to obtain smaller regions of memory. The underlying base pointer is directly exposed to llama.cpp, such that any references to memory within the memory region can be resolved, and the correct capability can be diminished. This provides the benefits of capability-based security while making it transparent to the CUDA application – it is managed entirely by the interceptor.

For a region of device memory to be visible to RDMA (which is necessary for host-to-device and device-to-host memory copies), it must be pinned using GPUDirect RDMA [45]. However, there is a limit to the maximum amount of memory that can be pinned at once. To get around this, we can either dynamically pin and unpin memory as needed, or increase the maximum amount of pinned memory. Dynamically pinning/unpinning memory was seen as undesirable as this operation is fairly expensive, so we opted to use the NVIDIA Display Mode Selector tool to increase the maximum amount of pinned memory [46]. This has the side-effect of disabling graphical output, which is not needed for HPC applications such as ours anyway. For future work in remote GPU access using FractOS, we consider this configuration change to be essential for efficiently running real applications.

Host memory in CUDA can be allocated using `cudaMallocHost`, which allocates page-locked memory and makes it accessible to the GPU. In normal CUDA applications, the CUDA driver can accelerate calls to functions like `cudaMemcpyAsync` when using memory allocated in this way, but this is not possible in our case since host and device memory reside on different machines. Therefore, our interceptor for `cudaMallocHost` simply allocates memory normally using `malloc` and registers it for use with RDMA (using FractOS APIs).

Llama.cpp (and many other CUDA applications) allocate large blocks of host and device memory in pools, and then reference smaller regions within these blocks. Therefore, we end up tracking a relatively small number of memory capabilities on the CUDA interceptor. For this reason, we simply search through the capabilities linearly whenever we need a memory region, rather than using a more elaborate data structure such as a binary tree. In practice, the superior spatial locality of the C++ vector is likely to produce better performance than other data strictures except for extremely large numbers of capabilities, which does not occur in normal operation.

A final point on memory allocation – llama.cpp uses a memory pool for both host and device memory, and performs almost all of its allocations during the first inference request, rarely needing to allocate more memory in subsequent requests. This is convenient for us, since registering pages of memory for RDMA is a fairly expensive operation. The transient increase in latency due to RDMA registration is not a major concern for us, since the llama.cpp server can be configured to simply execute a warm-up run once the model has been loaded.

5.2 Memory copy

Llama.cpp makes extensive use of the asynchronous memory copy API, `cudaMemcpyAsync`, in line with its design to make as many operations asynchronous as possible. H2H (host-to-host) and D2D (device-to-device) memory copies are straightforward to implement; H2H is just done locally, and D2D simply requires performing an RPC to the CUDA service, with the correct memory capabilities attached. However, H2D (host-to-device) and D2H (device-to-host) are more complicated if CUDA semantics are to be obeyed.

Since host and device memory reside on different machines, it is necessary to perform copies using RDMA (using the FractOS APIs). However, `cudaMemcpyAsync` still needs to be treated as a CUDA task to respect stream semantics:

- When `cudaMemcpyAsync` is submitted to a CUDA stream, it should not begin executing until all previous tasks on that stream have completed.
- All subsequent tasks submitted to that stream should not start executing until `cudaMemcpyAsync` has completed.

Forcing the copy to only start once the previous tasks have completed is not difficult to do using CUDA events – we create an event, record the contents of the stream at the time of the memcpy, and block until ready. However, forcing subsequent tasks to wait is non-trivial, and we considered several approaches:

1. Use a 'barrier kernel' to act as a proxy for the actual RDMA-based memcpy operation. The barrier kernel is so named as it blocks subsequent tasks from starting as long as it is running. The lifetime of the barrier kernel (from waiting, to executing, to finished) should closely follow that of the RDMA copy to avoid overhead.
2. Whenever we want to submit a task to a stream which is currently executing an RDMA memcpy, instead add the task to a queue managed by the CUDA service. When the copy completes, add all the tasks from that queue to the actual CUDA stream.
3. Use `cudaStreamWaitEvent` or a similar API to block the stream until the RDMA copy is finished. However, there is no straightforward way to arbitrarily trigger a CUDA event in the public API; event completion must be linked to the completion of some actual CUDA tasks. We considered this option to be unfeasible unless we were to reverse-engineer CUDA events to find a way to forcefully alter their state.

Reviewing these options, the first was considered to be the simplest, while the second was thought to likely to be the most efficient since it does not require launching an additional kernel. However, we assumed that the overhead of a single kernel performing minimal computation was likely to be quite low, so we opted for this solution, with the idea that it could be improved if it turned out to be a performance bottleneck. The barrier kernel simply loops infinitely until a flag in device memory is set, which is done when the RDMA copy is complete. The key steps of this solution are shown in fig. 5.1.

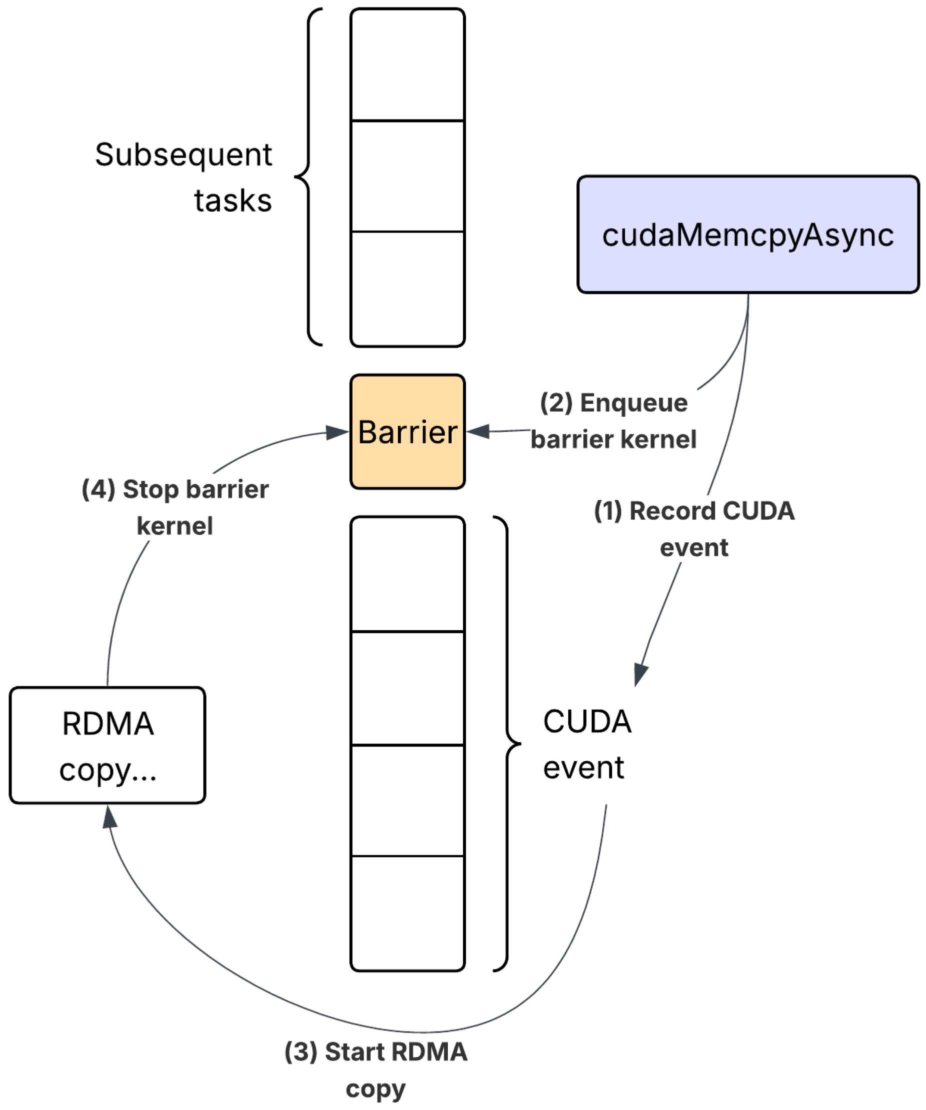


Figure 5.1: The key steps taken by our memcpy solution to ensure that CUDA stream semantics are obeyed. `cudaMemcpyAsync`, shown in blue, is the entry point.

The major limitation of this design is that it is not asynchronous. The thread on the CUDA service blocks until all previous tasks are complete (according to `cudaEventSynchronize`), and synchronously waits on the RDMA copy to finish. Making this asynchronous would require adding a new event loop to detect when a CUDA event has finished, which was not feasible in the available time. While this seems like it might lead to significant performance overheads, this is not what we found in practice – the Evaluation chapter shows this effect and explains why it happens.

5.3 Kernel launch

On the CUDA service, we use the CUDA module APIs to load modules from `.fatbin`, `.cubin`, or `.ptx` files. Kernels can then be loaded from modules by name (which is mangled and usually quite hard to read, due to llama.cpp's extensive use of templated kernels). When intercepting kernel launches from llama.cpp, the primary difficulty is in recovering the name of the kernel. `cudaLaunchKernel` takes as an argument a pointer to an opaque structure in host memory, which is used to call the function on device memory. However, there exists no public CUDA API to obtain the name of the kernel from this pointer. Therefore, we considered three options:

1. Instead of intercepting `cudaLaunchKernel`, intercept `cuLaunchKernel` (from the driver API). This takes a `CUfunction` as an argument, whose name can be obtained with `cuKernelGetName`.
2. Manually write interceptor functions for each unique kernel used by llama.cpp, then forward the name and arguments to the kernel launch RPC.
3. Find some way to convert these opaque pointers into names, likely by partially reverse-engineering CUDA kernel launch.

We deemed the first option to be unfeasible, as the CUDA driver API is not dynamically linked with the CUDA runtime shared library, and therefore cannot be intercepted with `LD_PRELOAD`. Techniques to get around this exist, such as dynamic binary instrumentation, but we thought such techniques would be too time-consuming given the time available. The third option was initially also considered unfeasible due to the closed-source nature of CUDA. Therefore, we opted for the second option, manually writing interceptors for each kernel.

Of course, this method was extremely laborious. Llama.cpp makes heavy use of templated kernels (to offload computation from run-time to compile-time), which made reconstructing the mangled names a complex task. The choice of batch size, input prompt length, and model size also affects llama.cpp's choice of kernels for some nodes in the computation graph, requiring the system to be updated frequently. This system is also extremely brittle, since different compiler configurations can produce different mangled names for kernels. Hence, we returned to the third option, reverse-engineering kernel launch in CUDA.

We constructed a simple application which only launches a single kernel to gain insight into CUDA. The relevant finding is that at the start of a CUDA program,
`__cudaRegisterFunction` is called. This undocumented function takes as arguments the pointer to the opaque struct which is used by `cudaLaunchKernel`, as well as a string containing the name of the kernel. Therefore, by intercepting this function when llama.cpp starts and storing the relevant arguments in a map, it is possible to find the name of the kernel when intercepting `cudaLaunchKernel`.

To actually launch these kernels on the CUDA service, the following steps are necessary:

- Compile llama.cpp with the nvcc flag `-keep`, which keeps all intermediate files generated by the CUDA compiler.
- Among these intermediates are `.fatbin` files containing all kernels defined in `.cu` files. For llama.cpp, there are about 30 different `.fatbin` files. Put them somewhere accessible to the llama.cpp server binary.
- The CUDA interceptor loads these files into memory, and then into modules on the CUDA service (using RDMA to copy the data across).
- Since the llama.cpp server binary and the `.fatbin` files were produced during the same compilation run, the mangled names will exactly match. The kernel names recovered when intercepting `cudaLaunchKernel` can be sent to the CUDA service, which loads and launches the correct kernel.

While this process is not as simple as just running the CUDA program with no modifications, it is considerably more convenient than manually defining interceptors for each kernel.

Unfortunately, it is not possible to know which module a given kernel resides in a priori; therefore, our system searches every module. This is quite wasteful, so a key optimisation here is to maintain a mapping from kernel names to RPC objects. This cache avoids the need to search through modules for a given kernel more than once. In practice, most of the kernels used by a particular LLM in llama.cpp will be the same between inference requests, so simply warming up the model with a single request (which the llama.cpp server does by default) is sufficient to obtain nearly all the functions needed.

5.4 Streams and events

Since streams and events are so intertwined, we consider them together. We use unique IDs to make llama.cpp think it's using handles to actual CUDA objects, whereas in fact the CUDA interceptor simply uses them to index into a map to get the RPC objects.

In CUDA, there are default streams which are used by API functions like `cudaMemcpy` when no stream is specified. APIs which take explicit streams can also take default stream handles as arguments. There are two default streams in CUDA:

- The legacy default stream, which has value 0. Each device has its own legacy default stream. A task on this stream will not start until all other streams on the device have finished executing, and all other streams will wait while the legacy default stream is executing. In other words, all streams synchronise with the legacy default stream on a given device. This stream is not used by llama.cpp.
- The per-thread default stream, which has value 2. This stream is per-thread and per-device, and unlike the legacy default stream, this stream does not synchronise with other streams. This stream is used extensively in llama.cpp.

In line with our objective to focus on what is required by llama.cpp, we only implemented the per-thread default stream. As mentioned previously, the llama.cpp server only every issues CUDA calls from a single thread. Therefore, we opted to simplify the implementation by assuming that there is only one per-thread default stream per device.

A minor complication is that default streams (both the legacy and per-thread ones) cannot be uniquely identified by their handle (always 0 or 2 respectively). Thus, if a stream with handle 2 is used, we assume it refers to the per-thread default stream of the current set device, in line with the proper CUDA semantics. Adding true per-thread default streams would require a small modification to associate these streams not just with device ordinals, but also thread IDs, which can be easily obtained in C++.

Many of the stream and event APIs, such as `cudaEventRecord` and `cudaStreamWaitEvent`, require obtaining the handles of other events or streams respectively. We implement this functionality as follows:

- The ID of an object such as a stream or event is shared between the library and server components of the CUDA service (i.e. it is the same locally and remotely).
- The RPCs exposed by the CUDA service library take stream and event objects as arguments, from which the IDs can be obtained.
- These IDs are sent to the CUDA service server, which traverses the tree of objects to find the correct one using the ID.

5.5 CUBLAS API

Llama.cpp relies on GEMM (GEneral Matrix Multiplication) kernels, available through the CUBLAS API. Because the CUBLAS kernels are not open-source, it is not possible to simply compile them and then load them into modules to handle them like ordinary CUDA kernel calls. Nor are compiled `.cubin` or `.fatbin` files publicly available. At the time, we came to the conclusion that the only viable solution is to call CUBLAS API functions directly on the CUDA service. This does somewhat violate disaggregation, since we originally envisioned the server component of the CUDA service as having the bare minimum functionality required to convert RPC calls to CUDA driver calls, like an RPC-to-hardware translation layer. Using the CUBLAS APIs comes close to running application logic on disaggregated hardware pools, which goes against the purpose of disaggregation. However, given the time constraints, and with the view than this can be explored in future work, we took this to be an acceptable limitation of our system.

Many of the GEMM functions take `alpha` and `beta` as arguments. These are pointers to floats, which are used as scaling parameters in the matrix calculation. Importantly, they can be pointers either to host or device memory - thanks to Unified Virtual Addressing, the CUDA runtime/driver can figure out which one it is and access it accordingly. This introduces complexity for our system, since host and device memory are not on the same machine. Llama.cpp always places the parameters on host memory, so to simplify our design we assumed that `alpha` and `beta` will always be pointers to host memory. Adding support for `alpha` and `beta` in device memory is not particularly complex, but we felt it was unnecessary for this project.

We also observed that before every GEMM function call, llama.cpp will call `cublasSetStream`. This sets which stream all subsequent GEMM kernels will be submitted to, since the GEMM APIs themselves do not take streams as arguments. Originally, `cublasSetStream` was implemented as an RPC, but with this knowledge, we optimised the design to store the stream in the interceptor state. Every RPC for a GEMM kernel now contains the stream handle, and calls `cublasSetStream` on the CUDA service. This effectively halves the number of RPCs used for the CUBLAS API.

Chapter 6

Performance Modelling and Optimisation

We can now predict the performance of running llama.cpp remotely on FractOS compared to running it locally. Since the prefill and decode phases have such different characteristics, we consider them separately. To allow us to perform these calculations, we make a few assumptions based on rough measurements and observations of a run of llama.cpp, taking 1024 input tokens and generating 256 tokens:

- During the prefill phase, llama.cpp performs 3000 CUDA operations, with an average of 50us per operation. Most of these operations are kernel launches.
- During the decode phase, llama.cpp performs 1000 CUDA operations per output token, with an average of 8us per operation. Again, most of these operations are kernel launches, but note how much lower the time per operation is compared to the prefill phase. This is explained in the Evaluation chapter.
- The average RPC overhead per call to the CUDA service is 40us. This is based off of actual measurements as well as results from previous experiments on the RPC performance of FractOS, which estimated the overhead to be about 30us.

Assuming that one CUDA operation maps to one RPC, which is approximately true on average, we calculate the prefill and decode times and show them in table 6.1. The prefill time increases by 80%, but even this significant overhead is overshadowed by the 500% increase in decode time. This is because the overhead per RPC is simply much more than the time taken by a CUDA operation during the decode phase, especially since the decode phase is compute-light.

Mode	Prefill time (ms)	Prefill overhead	Decode time (ms)	Decode overhead
Local	150	-	2048	-
FractOS	270	80%	12,288	500%

Table 6.1: Estimated prefill and decode times for local/remote llama.cpp. 1024 input tokens, 256 output tokens. Model: Llama-2, 7B variant.

Taking potentially 6 times as long to process an inference request is unacceptable. The issue is that the total RPC overhead is too high; therefore, candidate solutions either try to reduce the overhead per RPC, or reduce the number of RPCs. We consider three main approaches: fully-asynchronous RPCs, CUDA graphs, and batched CUDA operations.

Fully-asynchronous RPCs – So far, we have exclusively been using synchronous RPCs, meaning that the CUDA interceptor blocks until it receives a response from the CUDA service acknowledging a successful operation. However, we could instead use asynchronous RPCs, where we continue immediately after the interceptor sends the request to the FractOS channel. At some point in the future, if we require the results of the computation (in llama.cpp's case, this could be a device-to-host memcpy), we can wait on an grouped future representing the completion of all asynchronous RPCs. This is similar in concept to pipelining in protocols such as HTTP [47]. The main difficulty is ensuring in-order delivery, since this is not guaranteed by FractOS. Some additional mechanism like sequence numbers would need to be implemented in the CUDA interceptor and service.

Furthermore, this solution does not ease the high amount of network congestion caused by sending one RPC per CUDA operation. Though not the focus of our analysis, we believed that reducing network congestion was also a desirable outcome.

Note that although RPCs can be synchronous or asynchronous, the actual CUDA API calls are almost always asynchronous – for example, `cuLaunchKernel` returns once the kernel has been enqueued on the device, not once the kernel itself has started or finished executing.

CUDA graphs – Llama.cpp supports CUDA graphs, where once the graph has been captured, it can be launched with a single CUDA call (`cudaGraphLaunch`). This is very appealing since this allows us to generate a token with just a few RPCs instead of 1000, but there are several complications:

- Llama.cpp does not just launch the exact same graph repeatedly. For every token, the memory addresses of some nodes change, which would require implementing additional APIs to patch graphs on the CUDA service.
- Llama.cpp rebuilds the graph from scratch every ~ 20 tokens due to changes in the dimensions of some tensors, so in practice the number of RPCs during the decode phase would reduce by a factor of ~ 20, rather than 1000. Such large spikes in RPC count would also likely be perceptible to the user, leading to an awkward user experience.
- Implementing CUDA graphs in this way would only affect the decode phase, since the prefill phase must build the graph from scratch. Reusing the same graph between inference requests is, in general, not possible, since llama.cpp dynamically chooses some operations based on prompt length.

Batched CUDA operations – CUDA graphs have several advantages as detailed in the Background chapter, but the one we are most interested in is the ability to replace many CUDA API calls with one. The other features, such as CUDA driver optimisations, are not as relevant. We considered if there is a way to reduce the number of explicit RPCs without having to implement the full CUDA graph API.

Instead of immediately converting a captured CUDA call to an RPC, we instead save it in some state that is stored by the library component of the CUDA service. Then, at some point in the future, we package all saved CUDA calls into a single RPC, and execute it synchronously. Unlike CUDA graphs, this solution does not require rebuilding the CUDA graph every ~ 20 tokens, and it improves the performance of both the prefill and decode phases. The main remaining design consideration is when to start/stop capturing CUDA calls.

We performed a similar analysis to before to estimate the performance of each option, with the results shown in table 6.2. Note the following assumptions:

- **Asynchronous RPCs** – Only 1 RPC per token needs to be synchronous, all others can be asynchronous. An asynchronous RPC has an average overhead of 10us (compared to 40us for a synchronous RPC).
- **CUDA graphs** – For 1/20 tokens, the graph is rebuilt from scratch, and for the other 19/20 tokens, $\sim$ 5 RPCs are required (for patching the graph with new addresses and launching it).
- **Batched RPCs** – Generating a token requires one batched RPC, and $\sim$ 5 other RPCs to synchronise the GPU and copy the results to host memory.
- CPU time, for example to iterate over the llama.cpp computation graph, is negligible.

Mode	Prefill		Decode	
	Time (ms)	Overhead	Time (ms)	Overhead
Local	150	-	2048	-
Asynchronous RPCs	180	20%	4608	125%
CUDA graphs	270	80%	2749	34%
Batched RPCs	150	0%	2109	3%

Table 6.2: Estimated prefill and decode times for each of the candidate RPC optimisations, compared to the local baseline. 1024 input tokens, 256 output tokens. Model: Llama-2, 7B variant.

Evidently, batching RPCs is the most promising optimisation, achieving performance that is only slightly worse than the local baseline.

6.1 Implementing batched RPCs

Deciding when to start and stop capturing CUDA operations was the main design decision. On the one hand, it is desirable to maximise the number of operations that are captured at once, to minimise the number of RPCs. On the other hand, lazily executing CUDA operations can break the semantics of programs if implemented incorrectly. For example, if a synchronous device-to-host memory copy is not executed immediately, the host program may access host memory with the expectation that it is filled with the results of some CUDA computation. Unified Memory significantly complicates this, since it performs implicit host-device memory copies to make data available whenever is it needed. Although we did not enable Unified Memory in llama.cpp for this project, we kept it in mind when making decision decisions for the benefit for future work.

Additionally, CUDA API calls can return both synchronous and asynchronous errors. Errors that should only be returned synchronously pose a problem for batched RPCs, since these errors would be delayed. Fortunately, this is not a problem for llama.cpp for two reasons:

1. Llama.cpp has a very straightforward error handling strategy: if a CUDA operation returns anything other than a success, the program crashes. Therefore, even if an error is delayed, the program will crash eventually.
2. During token generation, llama.cpp doesn't rely on synchronous errors – all operations are asynchronous apart from calls to `cudaStreamSynchronize` or `cudaEventSynchronize`. Therefore, if we construct batches of RPCs along these boundaries, the semantics remain the same.

For this project, a strategy of simply batching all CUDA operations until a call to `cudaStreamSynchronize` or `cudaEventSynchronize` is made would maximise the number of operations that we could fit in each batch, thus minimising the overall overhead. However, this doesn't work for general CUDA programs due to features like Unified Memory. We sought a more general solution.

A key insight is that we can use the CUDA graph API to batch operations with proper semantics. When capturing on a stream, the CUDA operations are not actually executed, but instead just stored in the graph. Therefore, by batching all RPCs between calls to `cudaStreamBeginCapture` and `cudaStreamEndCapture`, we can maintain the same semantics as the original CUDA program. We implemented APIs in the library component of the CUDA service to support this feature. A local Graph object can store a sequence of CUDA operations, which can be serialised and sent to the CUDA server to be 'replayed'. We added support for kernel launch, device-to-device memcpy copy, and several CUBLAS GEMM APIs – these are the only CUDA APIs that are captured by llama.cpp.

We decided to hook into `cudaGraphInstantiate` to send the batched RPC to be replayed on the CUDA server. In principle, it makes no difference whether we use this API or one of `cudaStreamEndCapture`/`cudaGraphLaunch` for this purpose. It is worth noting that we are only hacking into the CUDA graph API to implement batching with correct semantics – at no point do we use the actual CUDA graph functionality itself (hence, we disable all other CUDA graph APIs such as those used to patch executable graphs).

Fortunately, llama.cpp uses CUDA graphs with relatively coarse granularity. During the prefill phase, one CUDA graph is used per batch per device, and during the decode phase, there is one graph per device. The graphs capture the vast majority of CUDA API calls, excluding APIs like stream/event synchronisation.

A minor point on serialisation is that we do not include the serialised sequence directly in the RPC that is called when we hook into `cudaGraphInstantiate`. Instead, we include a memory capability, and use this to copy over the serialised data via RDMA. This is because the sequences can become quite large (typically 100KB for Llama 2 7B), making their direct inclusion in RPCs unwieldy. Our serialisation strategy is to simply collect all the arguments for each CUDA operation into packed structs, then copy these structs directly into a contiguous memory region. We leave optimisation of this system, such as a more space-efficient serialisation algorithm, to future work.

We also added support for an option where instead of directly replaying the CUDA operations on the server, we capture them into a CUDA graph, instantiate it, launch it, and then destroy it. These 'ephemeral graphs' are the most naive way to use the CUDA graph API on the server, and represent a first step towards a more comprehensive system, which would allow patching executable graphs rather than rebuilding them from scratch.

Chapter 7

Evaluation

We measure the performance of the llama.cpp server in two scenarios: a single inference request, where we are interested in the properties of the prefill and decode phases; and a realistic workload with multiple clients, where we are interested in metrics such as the average TTFT (Time to First Token) and TPS (Tokens Per Second). For the single-request case, we are also interested in an analysis of which operations are taking the most time. To this end, we integrate profiling into our CUDA interceptor.

Both these scenarios rely on a benchmark program which sends requests to the llama.cpp server. For the single-request scenario, we created our own benchmark, whereas for the multi-client scenario, we use a benchmark provided in the llama.cpp repository.

7.1 Experiment design

Key aspects of our experimental setup are as follows.

Model – We use the Llama 2 family of models, which come in three sizes: 7B, 13B, and 70B (referring to the number of parameters in billions). Llama models are among the most popular open-weights models, so we considered them to be a representative example to test our system on. At the start of the project, the most recent release was Llama 3. However, we chose to use the Llama 2 family instead as it has almost the exact same architecture, while having a much more manageable parameter range – Llama 3 comes in 8B, 70B, and 400B sizes, the last of which is extremely unwieldy to use, even with quantization.

Hardware – All machines used in all experiments are identical. Each machine has a 24-core AMD EPYC 7402P CPU and 4 NVIDIA RTX A6000 GPUs, each with 48GB of VRAM. The GPUs are connected by NVLink, minimising the overhead of multi-GPU parallelism. The Infiniband connections between the machines have a bandwidth of approximately 10GB/s.

Llama.cpp options – For this project, we disable Unified Memory, as it significantly complicates host-device memory copies. When running without FractOS (either locally or using the RPC backend), we also disable CUDA graphs. This is because when we run llama.cpp with FractOS, we only hook into the CUDA graph API to enable batched RPCs, rather than actually using underlying CUDA graphs implementation. We do not benefit from things like driver optimisations that are normally possible with CUDA graphs. Thus, leaving CUDA graphs enabled when running locally or using the RPC backend would be an unfair comparison.

7.2 Configurations

Our experiments run in one of three broad configurations:

- **Local** – All processes run on a single GPU node.
- **RPC** – The llama.cpp RPC backend runs on a GPU node, while the llama.cpp server and benchmark run on CPU node.
- **FractOS** – Similar to the RPC configuration, except we run the CUDA service instead of the RPC backend on a GPU node. Additionally, we run one FractOS controller on each node, as well a few other processes required by FractOS.

The local, RPC, and FractOS configurations are shown in fig. 7.1, fig. 7.2, and fig. 7.3 respectively. Our contributions are highlighted in turquoise. The purpose of the llama.cpp RPC backend is primarily as an additional baseline to compare our system against. We describe the main measures we took to ensure a fair evaluation.

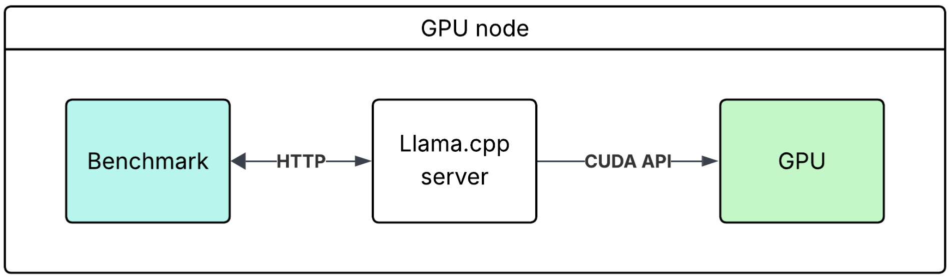


Figure 7.1: High-level view of local experiments.

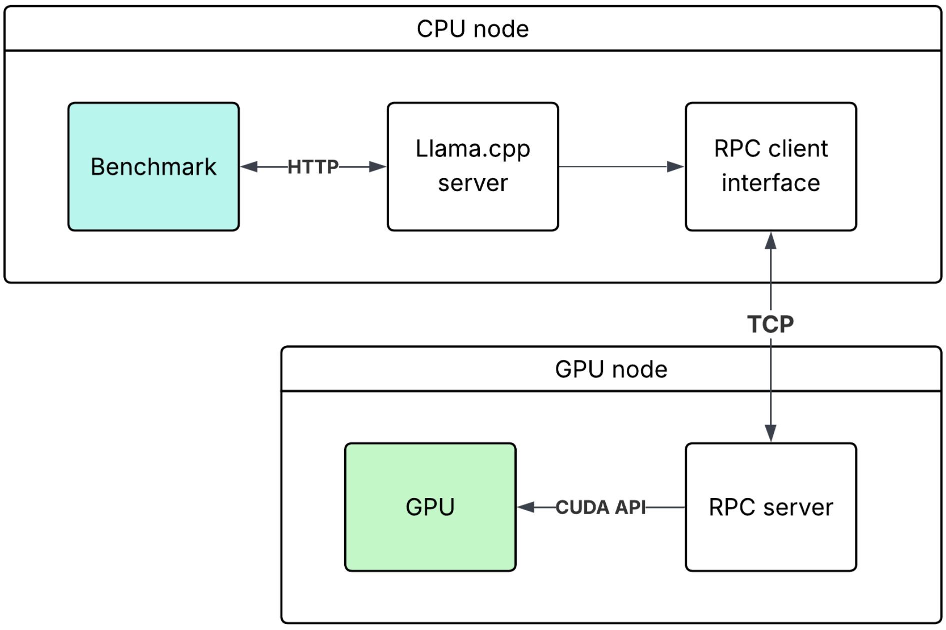


Figure 7.2: High-level view of llama.cpp RPC backend experiments.

Same hardware – Although we refer to 'GPU node' and 'CPU node' separately, these two machines are actually identical. However, we disable the GPUs on one of them using

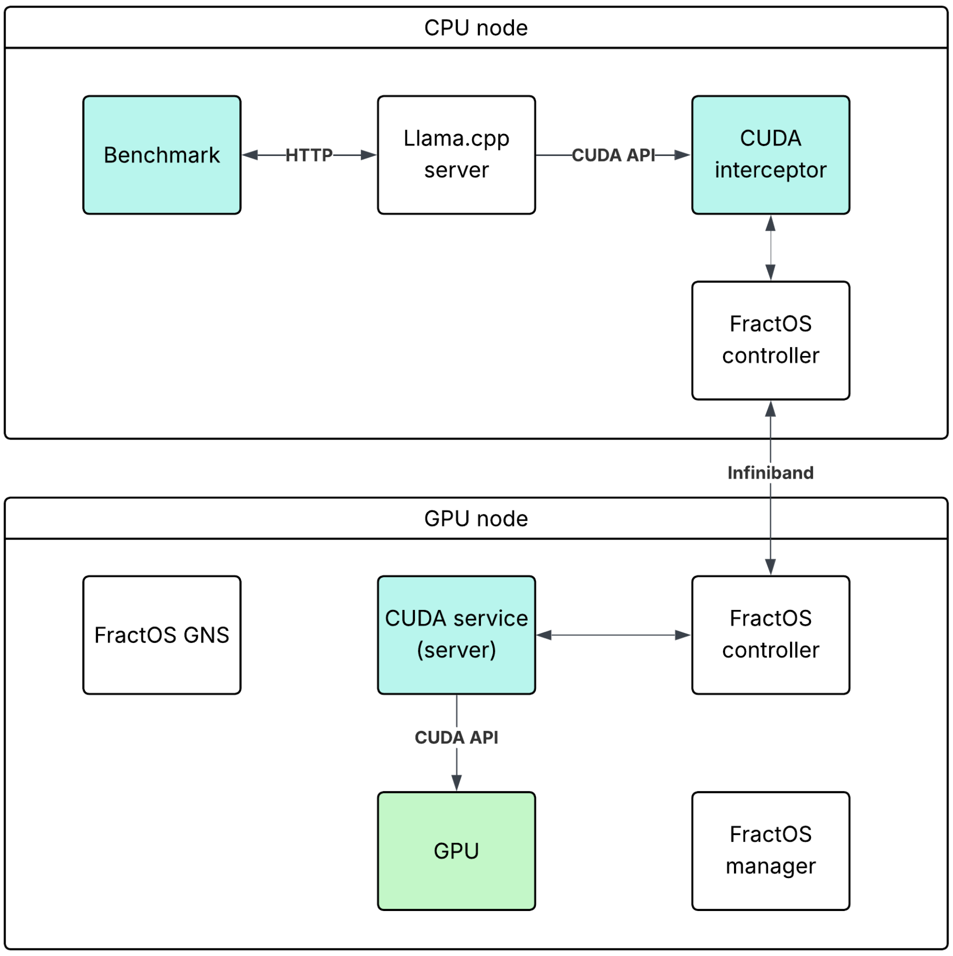


Figure 7.3: High-level view of FractOS experiments. Both FractOS controllers communicate with the FractOS manager and GNS, but for clarity these data transfers are omitted.

`CUDA_VISIBLE_DEVICES` to obtain an effectively CPU-only node. We do this to ensure that the processor and other non-GPU hardware is the same for both local and remote llama.cpp, to ensure a fair comparison.

Core isolation – We use `numactl` to ensure each process runs on a set of CPU cores that is disjoint from all others, to prevent scheduling conflicts and improve repeatability.

Exclusive access – We ran our experiments at times when the machines were not being used by anyone else, to avoid any interference from other processes as far as is reasonable.

7.3 Latency benchmark

The purpose of the latency benchmark is to obtain clear insight into the performance of the prefill and decode phases, including an understanding of which operations are bottlenecking the system. The reason we consider the prefill and decode phases separately is because they have such different characteristics, that profiling them together would lose significant information.

We use the cpr framework in C++ [48] to send one request at a time to the llama.cpp server. We attach callbacks to these requests, which allow us to record the time when the first token is received (assuming that the llama.cpp server is being run in streaming mode, sending each token as it is generated), and the last token. From these we can calculate the prefill and decode times separately. We also implemented a functionality to send multiple requests concurrently, but this was superseded by the separate load benchmark that is provided by llama.cpp.

Input sequence length – To control the prompt length in tokens, we tokenize the chosen prompt using one of the llama.cpp server endpoints, then select as many tokens as desired, discarding the rest.

Output sequence length – By default, LLMs stop generating tokens once they predict a special 'end of sequence' token. This behaviour is desirable in normal use cases, since responses should not be unbounded. However, this is undesirable for our experiments, since we want to strictly control the output token sequence length between runs. Therefore, we use the `-ignore-eos` option to force the model to continue generating even if it were to predict an `eos` (end of sequence) token as the most likely.

Slot reuse – One of the main optimisations available in the llama.cpp server is slot reuse. This is a mechanism where the server will search for a slot whose last processed prompt shared the longest common prefix with the current prompt. Because LLMs use masked self-attention, where a token's embedding is affected only by previous and not subsequent tokens, if two prompts share a common prefix then their KV caches will also share a common prefix. Therefore, the server can keep a prefix of the KV cache and discard the rest. However, this behaviour is undesirable for evaluation, since we want each request to be fully independent. Therefore, we disable this feature using `cache_prompt=false`.

7.3.1 Detailed profiling

To obtain insight into the inner workings of our system, we considered it essential to implement profiling for each CUDA operation and FractOS RPC.

For the machine actually executing the CUDA operations, this can be done using Nsight Systems, NVIDIA's profiling solution. This gives us metrics such as the average and standard deviation of each CUDA API used by the program, as well as breakdowns of all the kernels launched. Importantly, most of the CUDA APIs used by llama.cpp are asynchronous, so Nsight Systems records the time to set up an operation and submit it to a queue on the GPU, not the time to complete the task itself. On the other hand, kernel-level breakdowns do show us how much time a kernel actually spent executing on the GPU. Breakdowns are also available for memory transfers, but these are not as useful to us as we use RDMA in FractOS, which is not visible to this profiler.

However, Nsight Systems collects a huge amount of data, as one of its main uses is to show a graphical timeline of CUDA operations (which we do not use). Due to this limitation, we only run experiments with Nsight Systems enabled for a single inference request. We then extrapolate the breakdown from the profiler-enabled run to the profiler-disabled run, giving us insight into bottlenecks while providing accurate per-request times.

When running llama.cpp with FractOS, we also required profiling on the CUDA interceptor, so we implemented a basic system for recording the accumulated time and count of each operation. It is not straightforward to compare the profiler breakdowns between local and FractOS-based llama.cpp for several reasons:

- Local llama.cpp uses the CUDA runtime API, whereas the CUDA service on FractOS uses the driver API (except for CUBLAS calls).
- The implementation of memory copy on the CUDA service involves launching a barrier kernel and waiting on events, which contributes to the total times for `cuLaunchKernel` and `cuEventSynchronize`, while not calling `cuMemcpyHtoDAsync` or `cuMemcpyDtoHAsync` at all.
- In practice, running llama.cpp remotely causes some CUDA operations to 'consume' others - this is explained in detail later.

Since our latency benchmark performs a few warmup runs to achieve a steady state first, we wanted a way to control when to start and stop profiling. This can be done with the CUDA profiler APIs. We hacked two of the endpoints of the llama.cpp server that we do not use (`GET /lora-adapters` and `POST /lora-adapters`) and repurposed them to start/stop profiling respectively. When running with FractOS, the CUDA interceptor captures this call, starts its own profiling, and then forwards the call to the CUDA service so that it can start profiling with Nsight Systems.

7.3.2 Prefill phase

We start with an analysis of the prefill phase, where we request llama.cpp to generate a single token.

Prompt length

fig. 7.4 shows how prefill time varies with prompt length. The performance of FractOS suffers for short contexts, with an overhead of 25% for 200 input tokens (without batching). This is due to the network overhead imposed by the minimum FractOS RPCs needed to generate a token, which is not dependent on context length. However, FractOS performs surprisingly well for larger contexts, with an overhead of as little as 3% without batching. Interestingly, enabling batching actually degrades performance for all prompt lengths. It is clear that the FractOS overhead is somehow 'absorbing' a portion of the time that is normally spent on CUDA API calls.

To gain insight into this behaviour, we can look at the detailed profiler breakdown for FractOS without batching in fig. 7.5. We see that both the asynchronous APIs (kernel launch, memset, event record, etc.) as well as the only synchronous API, stream synchro-

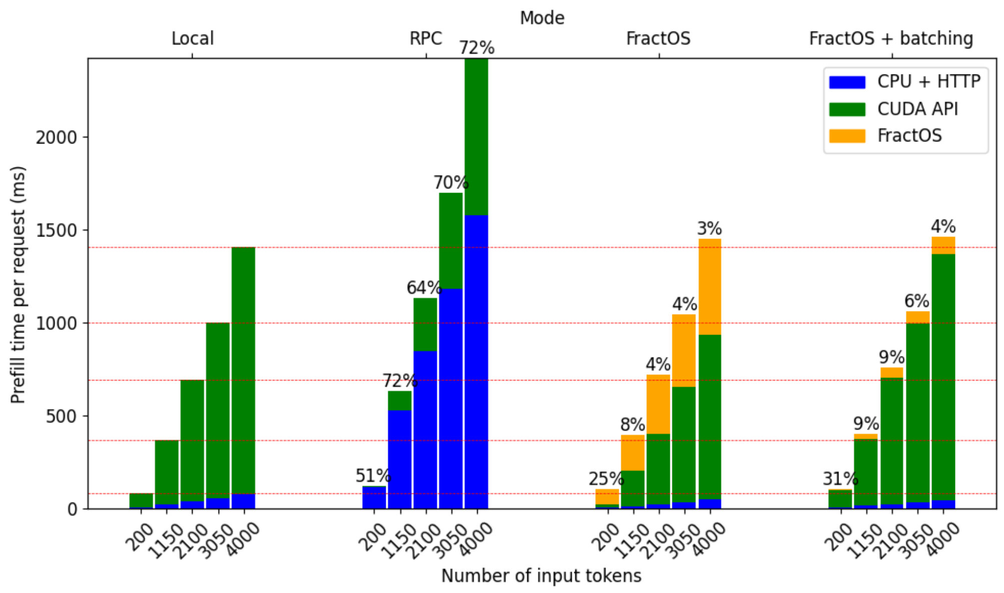


Figure 7.4: Effect of prompt length on prefill time. Llama 2 7B on 1 GPU.

nize, take significantly longer when running locally. We provide separate explanations for these phenomena.

Stream synchronization – Because FractOS introduces significant overhead when batching is disabled, the GPU is able to make significant progress on its tasks while we are waiting for RPCs to complete. In other words, we get implicit parallelism between RPC calls and GPU computation. At the end of each token generation step, when we call `cudaStreamSynchronize` to wait for all the tasks to complete, we do not have to wait as long since significant progress has already been made.

Asynchronous APIs – When calling an asynchronous API such as `cudaLaunchKernel`, the CPU will block until space is available on the GPU task queue. Because the prefill phase is compute-heavy, this happens often when running locally as the queue gets backed-up with long-running kernels, causing the average time per kernel launch to increase. However, this isn't an issue when running on FractOS, because the time spent waiting for RPCs provides a 'buffer' for the queue to make progress, reducing or eliminating the time that the CPU has to block for.

The fact that FractOS performs worse when batching is enabled is thus due to its inability to overlap FractOS RPCs and GPU computation as effectively. Nonetheless, it should be noted that the penalty that batching imposes is very minor.

It should be noted that when asynchronous APIs block in this way, it is actually often seen as a good sign, as it indicates high GPU utilisation [27][28]. Running llama.cpp on FractOS does not seem to decrease utilization, since if it did, the execution time would be noticeably longer for the compute-bound prefill phase. Instead, it simply spaces out the API calls (mostly kernel launches) at no cost to execution time. Of course, this is dependent on the specific hardware used, and in principle a fast enough GPU would not exhibit this effect, making the effective RPC overhead higher.

Number of GPUs

Next, we examine the effect of the number of GPUs used, fixing the prompt length at 1000 input tokens (the equivalent of $\sim$ 800 words, or a few paragraphs), to represent a

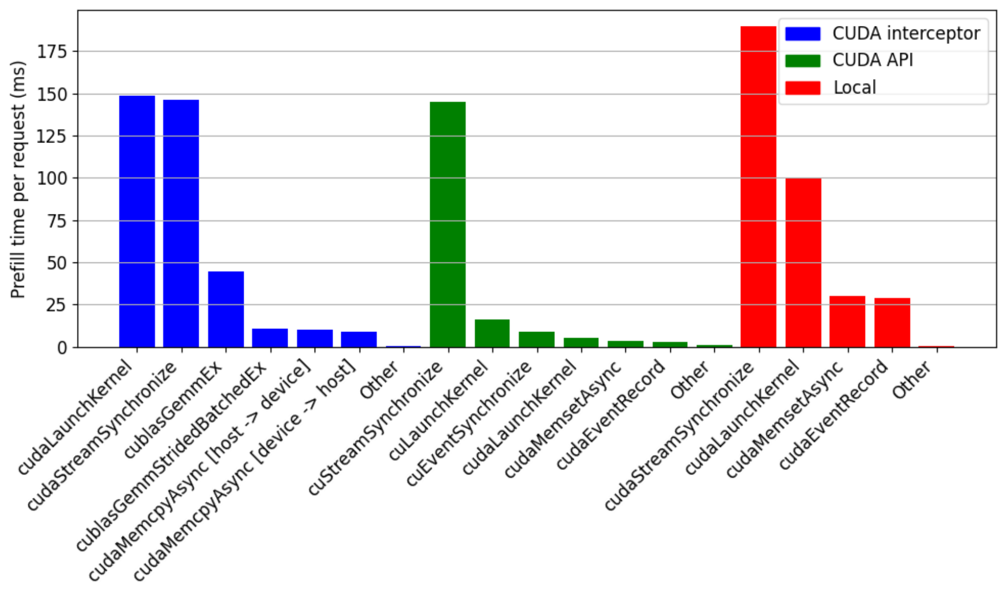


Figure 7.5: Profiler breakdown comparing FractOS (no batching) to local llama.cpp. Prefill phase only, 1150 input tokens. Llama 2 7B on 1 GPU.

moderately-sized workload. There are a few limitations with experiments involving multiple GPUs:

- A single llama.cpp RPC server can only use a single GPU. To use multiple GPUs, multiple servers need to be launched, which we did not implement in this project. Hence, we leave out measurements of the llama.cpp RPC backend from our multi-GPU experiments.

- In llama.cpp, CUDA graphs and tensor parallelism are incompatible due to the complexity of the memory buffers used. We considered it unfeasible to add support for this in time, so instead we exclusively use pipeline parallelism for multiple GPUs.

The results are shown in fig. 7.6. We see that, contrary to our expectations, increasing the number of GPUs does reduce the prefill time. While not a particularly well-documented phenomenon, we hypothesise that splitting the model across multiple GPUs might reduce memory pressure. The overhead of copying tensors between GPUs is minimal as the GPUs all have fast NVLink interconnects.

FractOS (without batching) – The prefill time reduction associated with multiple GPUs seems to plateau for more than one GPU. The CUDA API time is already very small, and it appears that it cannot be reduced any further by overlapping GPU work with RPC overhead. Thus, the total time remains approximately constant.

FractOS (with batching) – We can see a slight increase in absolute FractOS overhead as the number of GPUs increases. This happens because we only batch CUDA calls that would normally be captured in a CUDA graph (between `cudaStreamBeginCapture` and `cudaStreamEndCapture`). Using multiple GPUs involves additional CUDA API calls for coordinating work that are excluded from this capture, such as `cudaEventRecord` and `cudaStreamWaitEvent`, which introduces additional RPC overhead. However, this effect appears to be quite minimal, and the increased overhead appears to absorb some of the CUDA API time, keeping the total prefill time close to local execution.

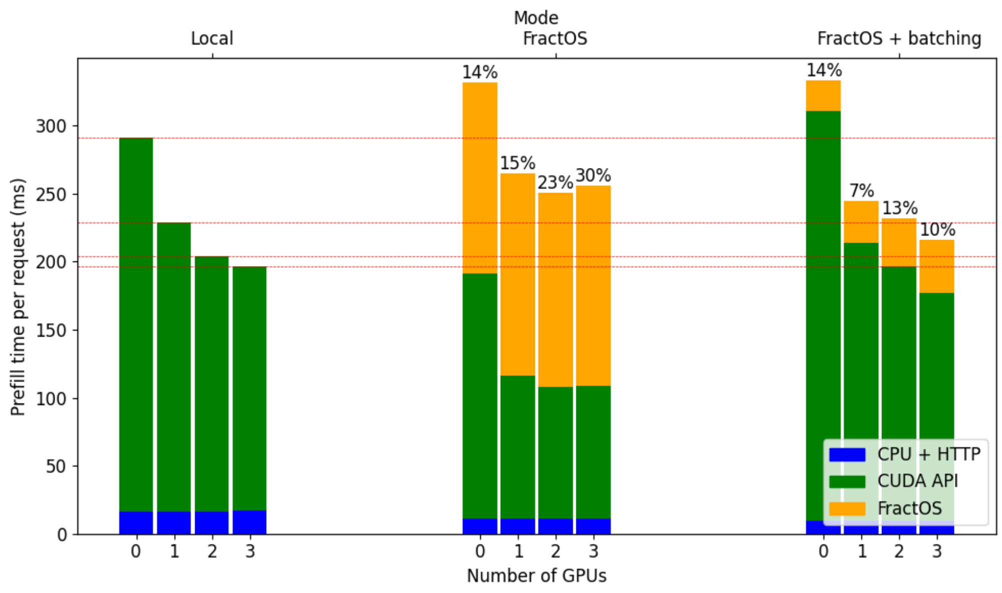


Figure 7.6: Effect of GPU count on prefill time. 1000 input tokens, Llama 2 7B.

Model size

Finally, we consider the effect of model size, comparing results for the Llama 2 7B, 13B, and 70B models. fig. 7.7 shows that the time spent on CUDA API calls increases with model size, which is as we would expect. However, we observe that the increase in FractOS overhead is sub-linear with respect to the increase in CUDA API time, even if batching is disabled.

This is because model size affects two things: the size of the weight tensors, and the number of layers. As more layers are added, the FractOS RPC overhead increases as the size of the serialised CUDA sequences increases. However, the size of the weight tensors does not change the number of CUDA calls – llama.cpp will dynamically choose kernels, or change the grid/block dimension, based on tensor size, but this does not increase the FractOS overhead.

Summary

- The compute-heavy nature of the prefill phase means that FractOS overhead is effectively overlapped with GPU work, making FractOS without batching perform quite well (3% overhead for 4000 input tokens).
- Batching slightly degrades performance as we lose some of this parallelism, though this effect is quite small.
- FractOS performance suffers for small contexts due to RPCs that are always made regardless of context length. Although high in relative terms, this overhead is small in absolute terms ($\sim$ 30ms).
- FractOS overhead increases slightly with GPU count due to RPCs used to synchronise GPUs. This overhead can absorb CUDA API time to prevent increasing prefill time, to an extent.
- Relative FractOS overhead is smaller for larger models (just 1% overhead for the 70B model, with batching enabled).

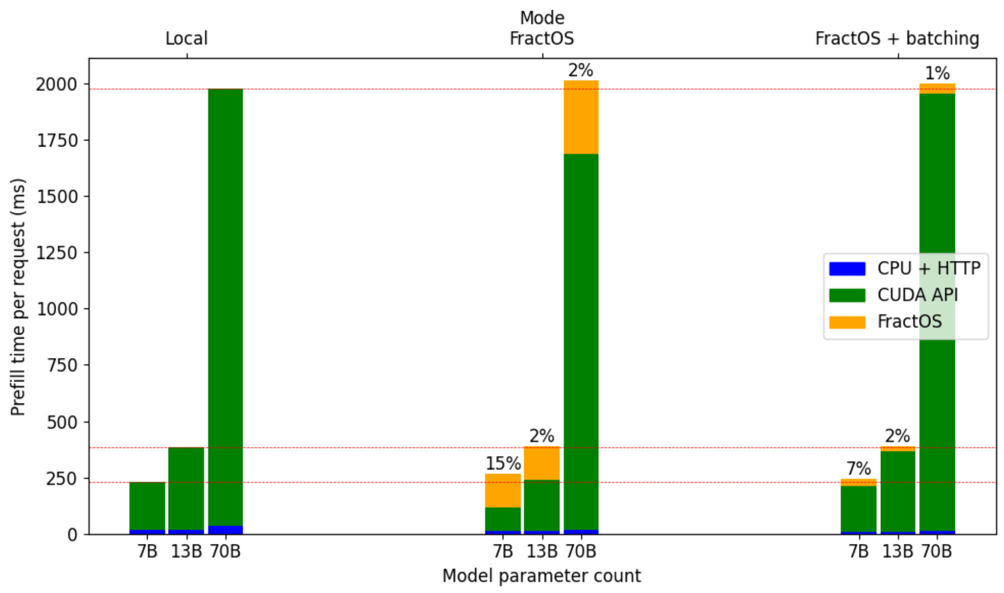


Figure 7.7: Effect of model size on prefill time. 1000 input tokens. We use 2 GPUs to ensure the 70B model can fit and leave adequate space for the KV cache, and then replicate this for the 7B and 13B models for a fair comparison.

7.3.3 Decode phase

In order to profile only the decode phase, we repeat the same experiments with $n \neq 1$ output tokens and 1 output token, then find the difference. Essentially, we discard the contribution of the prefill phase.

Output sequence length

We start by observing the effect of output sequence length on decode time. fig. 7.8 shows that the performance of FractOS without batching significantly degrades in the decode phase, whereas FractOS with batching performs much better, with an overhead of just 14% that is constant with output sequence length.

To identify where exactly the FractOS overhead is coming from, we show detailed profiler breakdowns for FractOS with and without batching, in fig. 7.9a and fig. 7.9b respectively. When batching is disabled, the RPC overhead is dominated by calls to `cudaLaunchKernel`. This is not surprising, since a single token requires $\sim$ 1000 kernel launches just for the small 7B model. Having to execute this many RPCs sequentially is very time-consuming, as shown in our calculations in the Optimisation chapter. The breakdown for FractOS with batching reveals some interesting insights.

Shifting synchronization – When running locally, stream synchronization takes up the majority of the CUDA API time. However, for FractOS with batching, we instead see this role filled by event synchronization, even though we only use a single GPU (and thus llama.cpp does not call `cudaEventSynchronize` during the decode phase). In fact, this event synchronization is caused by our implementation of memcpy, which uses an event to ensure the memcpy does not start until all previous tasks in the target stream have completed. This reveals an interesting property of our system, which is that we shift synchronization from one API to another, almost for free.

More concretely, recall the llama.cpp token generation process – it consists of many asynchronous operations, followed by an asynchronous device-to-host memcpy, followed by

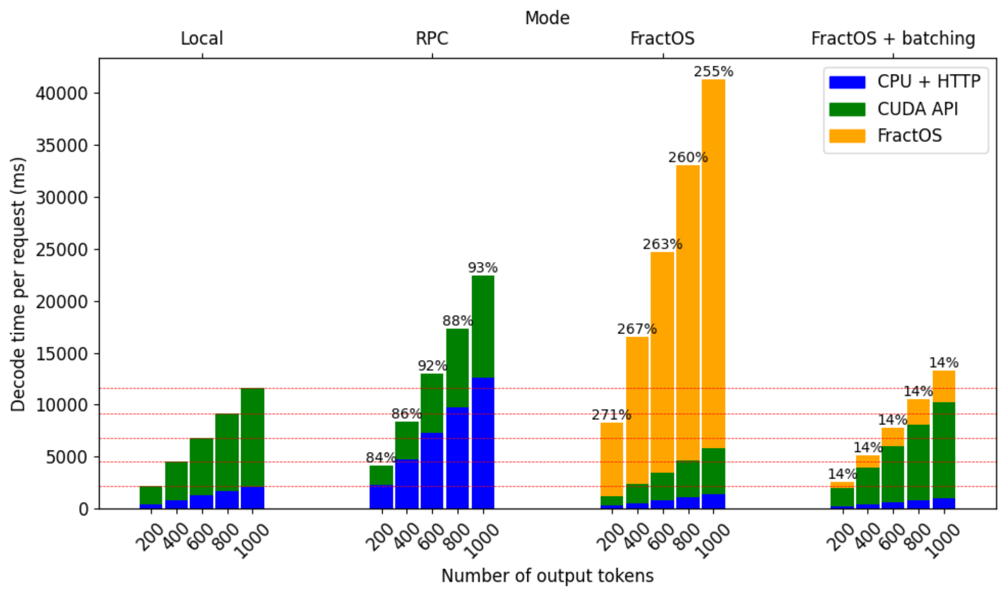


Figure 7.8: Effect of output sequence length on decode time. 1000 input tokens, Llama 2 7B, 1 GPU.

stream synchronization. By making that final memcpy synchronous, we move the burden of synchronization away from the final stream synchronization call (which is why it doesn't appear on our breakdown for FractOS with batching – it is negligible). We still obtain most of the benefits of asynchronicity, since the ~ 1000 or so kernel launches for each token are still asynchronous.

Batched RPC overhead – We can see that `cudaGraphInstantiate`, which we use to replay the stored CUDA sequence on the CUDA service, takes up noticeably more time than all of the CUDA API calls combined. As mentioned, this system is not well-optimised, and we give examples of how this could be done in the Future Work chapter.

Number of GPUs

In fig. 7.10, we can more clearly see an effect that was present in the prefill phase, which is that increasing the number of GPUs increases the RPC overhead. When batching is enabled, the total decode time is not affected because some of the CUDA API time is absorbed into RPC time. However, there is a limit to this, which is what we observe for FractOS without batching – the RPC overhead directly contributes to the decode time as we reach as lower bound on the CUDA API time. Unfortunately, we are unable to test this effect for more GPUs as we can at most use all 4 GPUs on a single machine.

The reason why the number of RPCs increases with GPU count is because the CUDA calls for orchestrating work between GPUs are not captured in the graph. However, batching only those operations that would normally be captured in a CUDA graph is a matter of choice, and we could extend our system to implicitly batch operations across larger timescales. The main consideration is to ensure that executing CUDA operations lazily does not break the program.

For llama.cpp, batching `cudaEventRecord`, `cudaStreamWaitEvent`, and `cudaMemcpyPeerAsync` would not break the semantics (as long as they are executed in-order), and these are the only additional APIs used for cross-GPU synchronisation. If we were to start implicit batching when any CUDA operation other than a host-to-device or device-to-host memcpy is called, and stop batching when a device-to-host memcpy is called, then we should be able to capture almost all of the operations used by llama.cpp while ensuring its correctness. Of course, this optimisation is specific to llama.cpp, which is why we opted for a more general solution instead, but we believe application-specific optimisations like this are still worth investigating in future work.

Model size

We conclude our analysis of the decode phase by comparing different model sizes in fig. 7.11. We see that increasing model size reduces the relative FractOS overhead, as this overhead increases at a slower rate than the CUDA API time. For the 70B model, FractOS with batching enabled has just 2% overhead compared to running llama.cpp locally, an overhead which is imperceptible in ordinary use.

It is instructive to compare the sources of the overheads for small and large models – we compare the 7B and 70B models. We show the profiler breakdowns in fig. 7.12a and fig. 7.12b. Compared to the smaller model, the overheads of the larger model are much more focused around just two operations: replaying the captured CUDA operations on the CUDA service, and performing device-to-host memcpy. This is because model size influences these two operations more than any others:

- Larger models perform more kernel launches, increasing the size of the CUDA sequence which must be serialized and sent across the network.
- Llama 2 70B has an embedding length of 8192 (this is the dimension of the vector used to embed a token), compared to 4096 for Llama 2 7B. The size of the data transfer when copying logits from device to host for each token is therefore larger.

For the 70B model, we can see that the time spent on memory copies between host and device, from the perspective of the interceptor, almost exactly matches the time spent on `cudaStreamSynchronize` for local llama.cpp. This implies that there is almost no RPC overhead for this operation – it is dominated by the (useful) synchronisation time. By contrast, there is a noticable overhead associated with `cudaGraphInstantiate`, which is the API we use to replay the CUDA operations. So as the model size increases, the dominant overhead becomes that of replaying CUDA operations, making this the main target for optimisation in future work.

Summary

- With batching enabled, FractOS has a decode time overhead of 14% (single-GPU case), constant with respect to output sequence length.
- FractOS without batching performs significantly worse – batching is a key optimisation to make remote CUDA feasible for real workloads.
- Increasing GPU count increases FractOS overhead, more so than in the prefill phase. For a larger number of GPUs, we would expect this to harm the decode time, though more work is needed to verify this.
- The relative FractOS overhead decreases with increasing model size, from 12% for the 7B model to just 2% for the 70B model (in both cases, using 2 GPUs).

7.4 Load benchmark

To simulate a realistic workload, we use the provided llama.cpp server benchmark, built on top of the k6 framework [49]. This simulates a fixed number of clients, each of which sends an inference request, waits for a response, sleeps for a short delay, and then repeats the process. The total number of requests sent by all clients is fixed – prompts are divided among the clients via work stealing.

To allow the llama.cpp server to handle concurrent requests, we set the number of slots to 4. This allows the server to manage 4 KV caches and batch 4 requests at once according to continuous batching. The dataset we use to source prompts is ShareGPT, which is a collection of real conversations shared by ChatGPT users. These conversations vary in length and complexity, although we discard conversations that are 'too short' or 'too long' according to the default benchmark parameters in [18].

An important note is that while we are able to run the 7B and 13B models on a single GPU, and therefore include measurements of the llama.cpp RPC backend, this is not the case for the 70B model. Therefore, we use 2 GPUs for the 70B model and exclude the RPC backend.

7.4.1 TTFT

The average TTFT results are shown in fig. 7.13. As expected, we observe a significant increase in TTFT for more than 4 clients – if there are more concurrent requests than slots, then some requests will be forced to wait until other requests complete, extending some TTFT values to the full decode time. This is particularly bad for FractOS without batching, since it suffers from much higher decode overheads than prefill overheads. For FractOS with batching, we observe performance only slightly worse than the local baseline, and this overhead does not appear to increase under high load.

As expected, these overheads also decrease with model size, to the point where FractOS without batching has practically the same performance as our local baseline.

7.4.2 TPS

The average TPS results are shown in fig. 7.14. As expected, FractOS with batching performs significantly better than both the llama.cpp RPC backend, and FractOS without batching. For the single-client case, FractOS with batching gives similar overheads as our decode phase latency measurements. We observe significant improvements for the larger model sizes, consistent with all our other results so far. Notably, we do not observe any significant performance degradation as the number of clients increases.

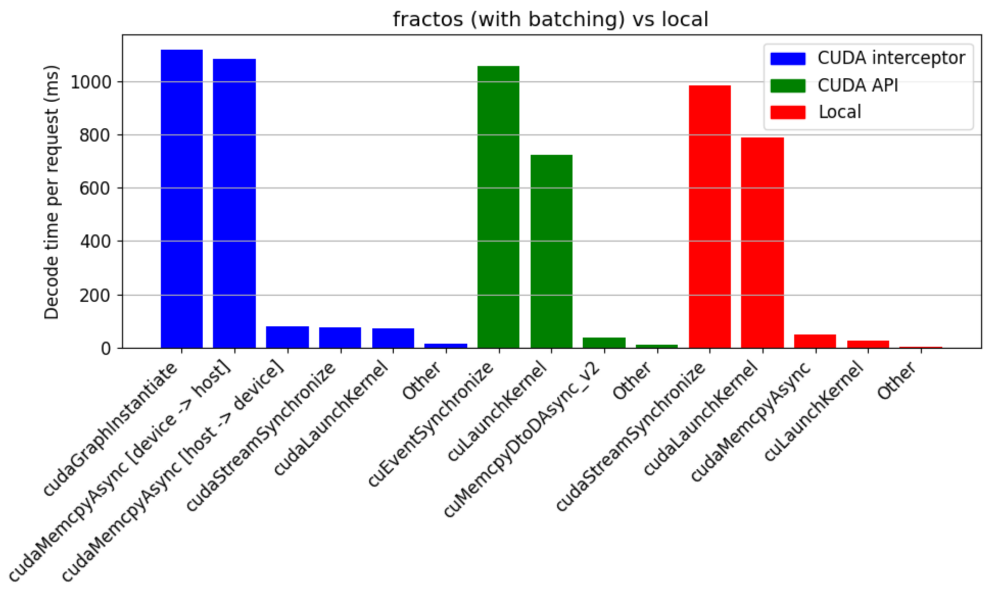
Fundamentally, the reason why the number of clients does not degrade the performance of llama.cpp with FractOS is because having multiple clients only affects batching (specifically, continuous batching). From the perspective of the CUDA interceptor, the work it performs is essentially the same, just that the parameters of the RPCs would vary to accommodate these different batch sizes.

7.5 Usability

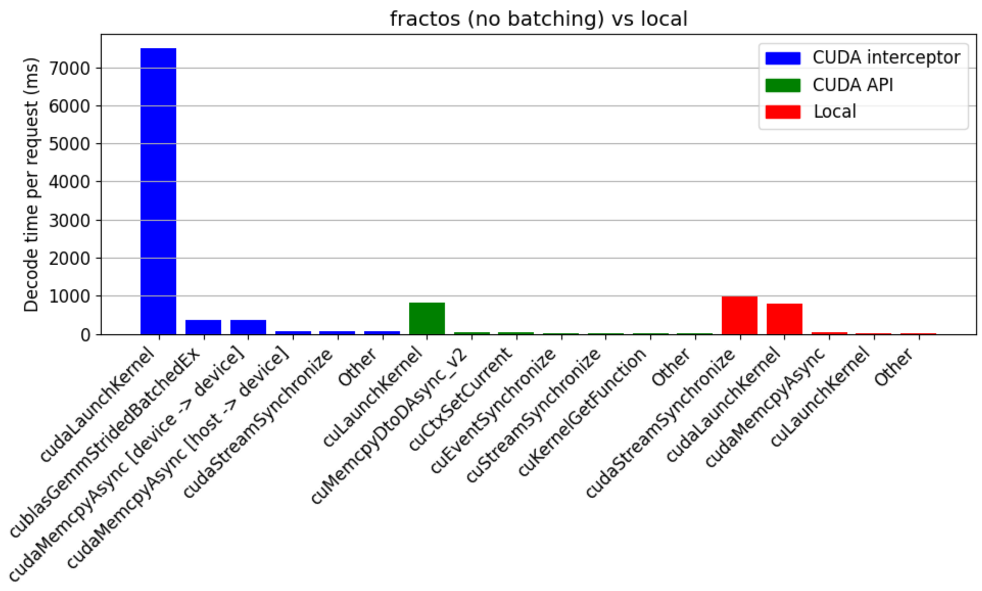
We conclude our evaluation with a discussion of the ease-of-use of our system. Assuming that FractOS has already been set up, and that batching is enabled, the following steps are needed to make llama.cpp work with FractOS:

- Disable Unified Memory and remove the CUDA graph API functions that are used to patch the graph. This means that the CUDA graph will be rebuilt from scratch for every token.
- Compile llama.cpp with the CUDA backend, modifying the nvcc command to output the intermediate files. Move the `.fatbin` files to a directory accessible to the llama.cpp server binary.
- Run the llama.cpp server with `LD_PRELOAD=<location of CUDA interceptor shared library>`.

Compared to previous solutions in this space such as rCUDA, our system requires very little additional work to transparently replace local CUDA calls with remote ones. Since our CUDA interceptor does not support the full CUDA API, it cannot currently run arbitrary CUDA programs. However, if support were to be added in the future, then the procedure above would apply to general CUDA applications as well.



(a) With batching.



(b) Without batching.

Figure 7.9: Profiler breakdowns comparing FractOS with local execution. 1000 input tokens, 200 output tokens, Llama 2 7B, 1 GPU.

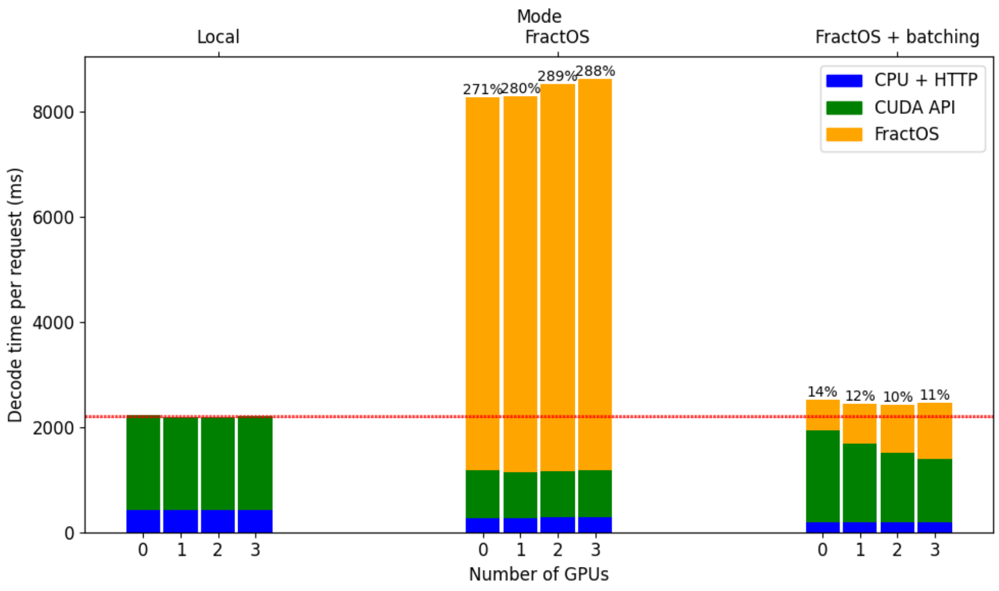


Figure 7.10: Effect of the number of GPUs on decode time. 1000 input tokens, 200 output tokens, Llama 2 7B.

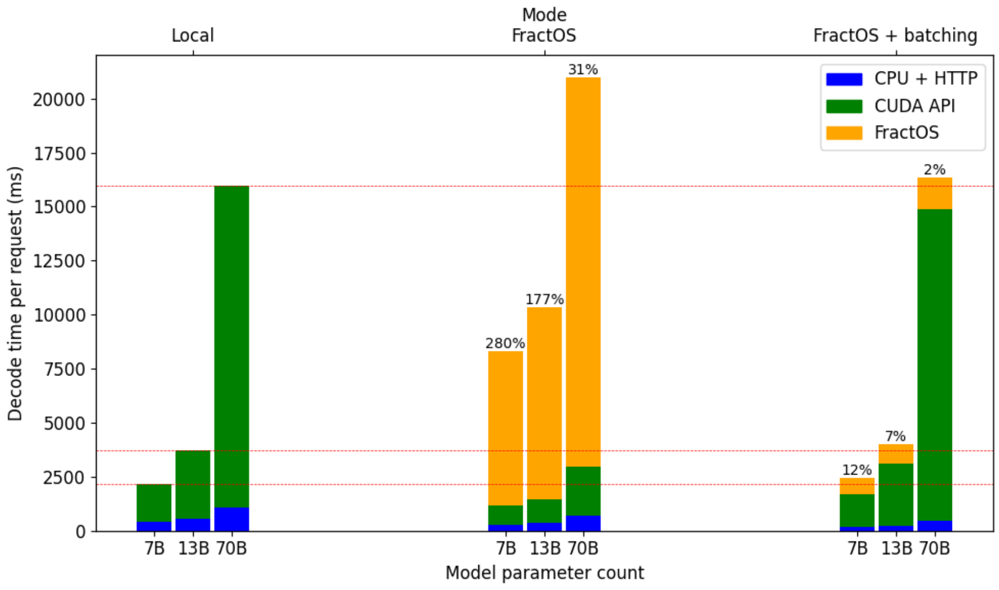
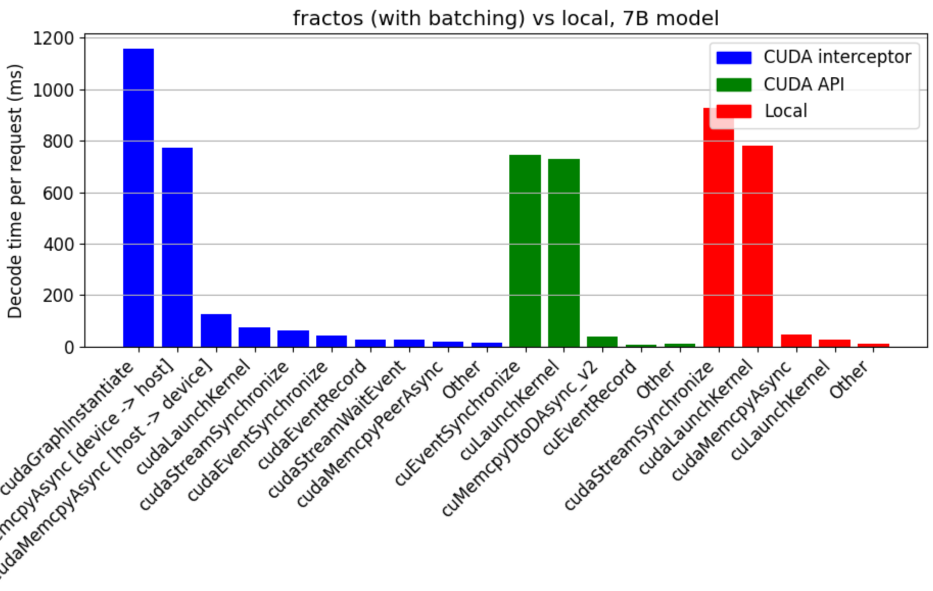
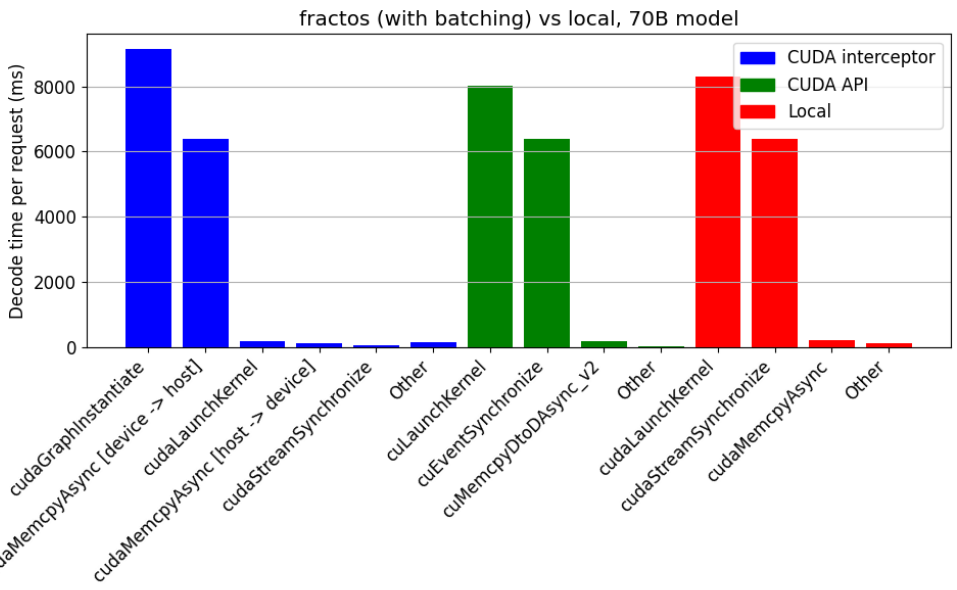


Figure 7.11: Effect of model size on decode time. 1000 input tokens, 200 output tokens. Like in the analysis for the prefill phase, we use 2 GPUs for all models.

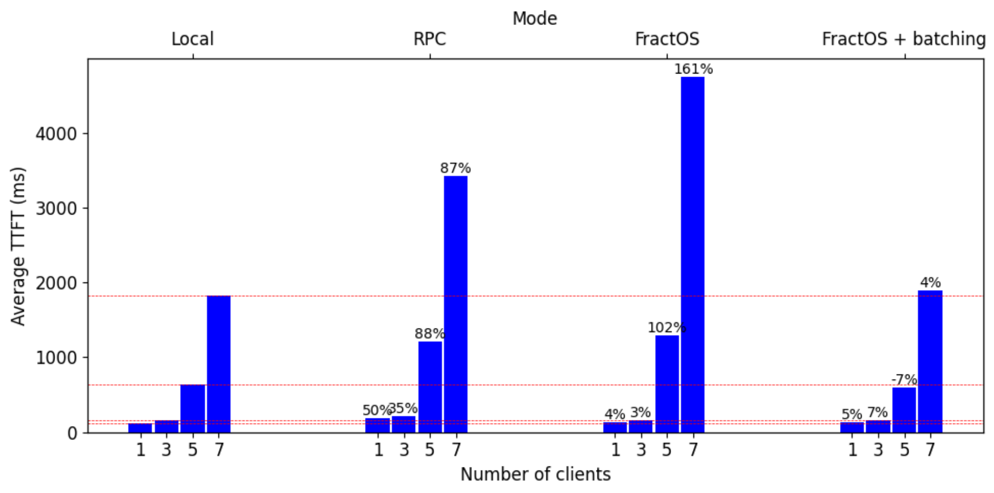


(a) 7B model.

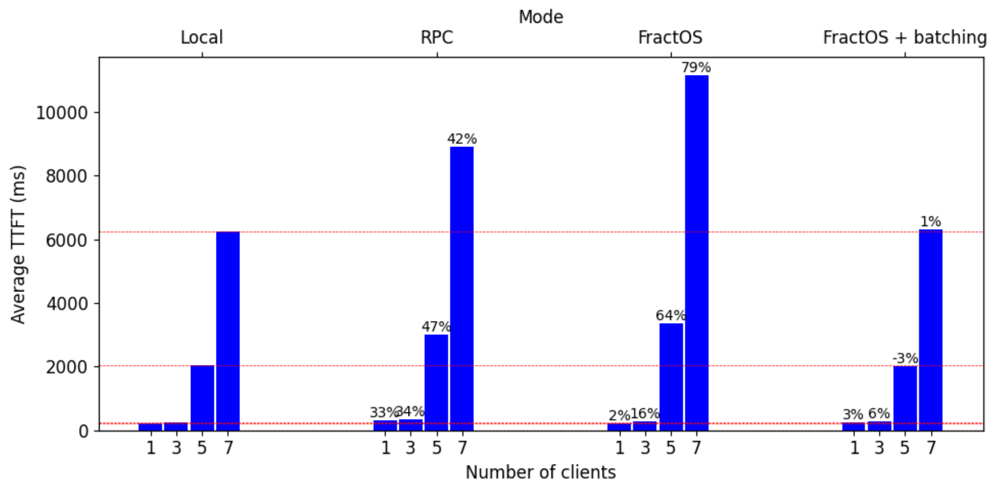


(b) 70B model.

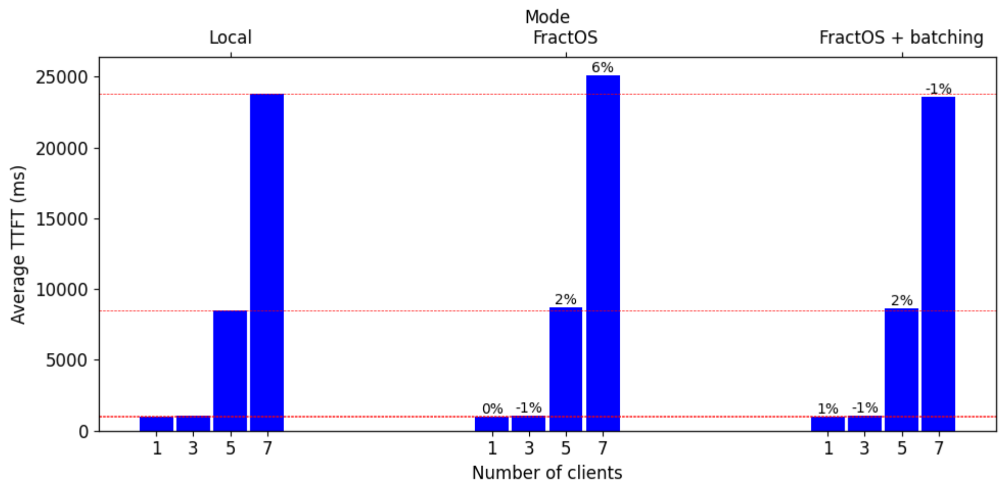
Figure 7.12: Profiler breakdown comparing FractOS and local llama.cpp for different model sizes. 1000 input tokens, 200 output tokens, 2 GPUs.



(a) 7B.

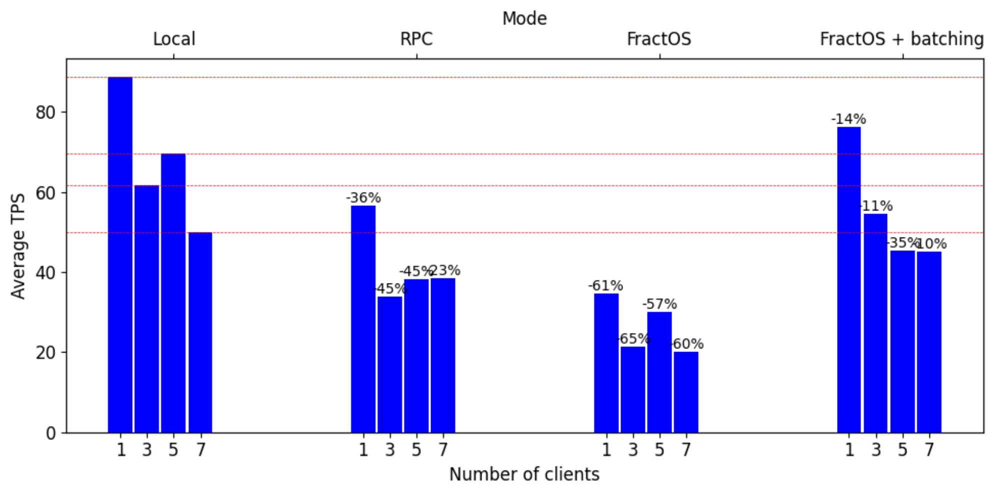


(b) 13B.

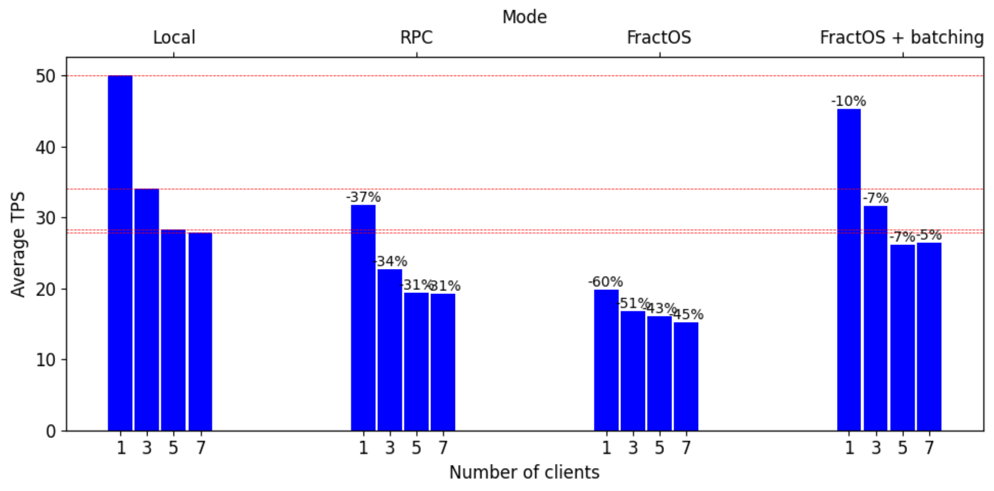


(c) 70B.

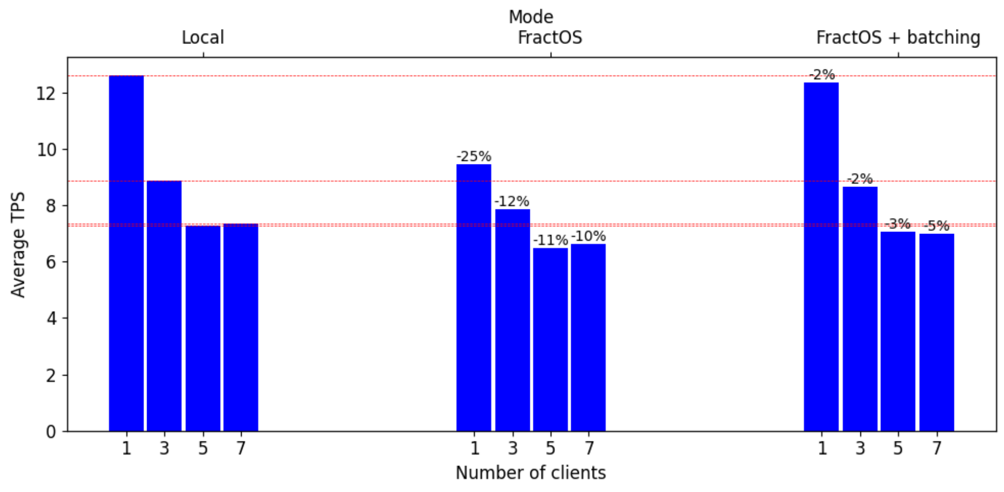
Figure 7.13: Average TTFT of different llama.cpp configurations across different numbers of clients. Comparison between model size. 100 requests total.



(a) 7B.



(b) 13B.



(c) 70B.

Figure 7.14: Average TPS of different llama.cpp configurations across different numbers of clients. Comparison between model size. 100 requests total.

Chapter 8

Future Work

Extending a CUDA program to efficiently use remote GPUs in a disaggregated environment is a project with enormous scope, and there were many features we were unable to implement in time. For future work, we consider both general improvements, as well as improvements specifically targetting llama.cpp.

Asynchronous host-device memory copy – The current system only supports synchronous host-device memory copies. As shown in the Evaluation chapter, this does not lead to significant overhead for llama.cpp, but this could definitely be the case for arbitrary CUDA programs. Additionally, we use a barrier kernel to prevent subsequent CUDA tasks from starting, which does incur some overhead. We provide the high-level design of a fully-asynchronous solution that does not use a barrier kernel:

- A CUDA event loop runs in a separate thread, allowing callbacks to be executed when a CUDA event is detected to have finished its recorded work.
- When a host-device memcpy is called, register a callback in the CUDA event loop so that the copy only starts once all existing work in the target stream has finished. Additionally, set a flag for the target stream to show that a host-device memcpy is present (even though it hasn't started yet).
- While the flag is set, all subsequent CUDA tasks targetting that stream will instead to enqueued to a data structure managed by the CUDA service.
- Since the RDMA copy is a FractOS operation returning a future, it can be chained into another callback. In this callback, called only when the copy has finished, unset the flag from earlier and submit all tasks to the actual CUDA stream.

This solution makes the two main 'waiting' parts of the copy, waiting for previous tasks and waiting for the copy itself, fully asynchronous, while preserving CUDA stream semantics. The most complex part of this solution is designing an efficient event loop for detecting CUDA events.

Additional CUDA features – We disabled two optional CUDA features used by llama.cpp to keep the scope of this project manageable: Unified Memory and APIs to patch CUDA graphs. Implementing these would also benefit a wide class of CUDA applications.

CUDA driver interception – Our CUDA interceptor targets the runtime API because of the difficulty of using `LD_PRELOAD` with the CUDA driver, which is not dynamically linked to the CUDA runtime shared library. Future work should consider how this could be achieved, since intercepting the CUDA driver API is more general and can support arbitrary CUDA applications, regardless of which API they use.

Additionally, intercepting the underlying CUDA Driver API calls used by CUBLAS on the client would avoid the need to call CUBLAS on the CUDA service. The CUDA service exclusively uses the driver API, with the exception of CUBLAS API calls. In general, it is ill-advised to mix runtime and driver API calls, so if the underlying CUDA driver calls could be intercepted on the client, this could be avoided. This would also be more in-line with disaggregation, since we want to restrict the CUDA service to the absolute minimum functionality required to translate RPCs to hardware commands.

Optimised RPC batching – Our system for batching RPC requests in FractOS could be significantly optimised. A few examples of such optimisations:

- Re-use the memory capability that is used to hold the serialised CUDA sequence, instead of re-creating it every time. For a single inference request, the total size of the sequence changes very rarely, so a cache storing memory capabilities of different sizes would likely be effective.
- Use a more efficient serialisation method. Currently, we naively pack arguments into structs and then pack these into a contiguous memory region. Finding a way to compress the data transfer could reduce the overall time.
- In practice, only small parts of the CUDA sequence change from token to token. Therefore, it may be more efficient to only transfer the delta between the previous and current sequences, and rely on the CUDA server to update its stored sequence.

In the case of llama.cpp specifically, we discussed how it should be possible to perform implicit batching over a longer time horizon. It is worth exploring how effective this is, whether it does indeed maintain the correctness of llama.cpp, and how this idea could be extended to other CUDA programs.

Better evaluation – Our evaluation could be improved in the following ways:

- Test more open-weights models, such as PaLM, Gemma, and Mistral.
- Avoid using Nsight Systems due to the large amount of data it produces, and instead implement a custom CUDA API profiling system using `LD_PRELOAD`.
- Include the llama.cpp RPC backend in the multi-GPU evaluation by launching one RPC server per GPU. Although the RPC backend is already established to perform significantly worse than both local execution and FractOS (with batching), it would be good to check experimentally if this holds for multiple GPUs.

Multiple remote GPU nodes – Our system currently only supports connecting to a single machine to use remote GPUs, which is quite limiting. A better system would allow the client to use remote GPUs from multiple machines by connecting to multiple services through something like a router. From there, FractOS's ability to coordinate RDMA transfers between devices on different machines without going through the main CPU could be used. This would bring our system closer to existing state-of-the-art distributed LLM inference solutions such as vLLM, allowing for a better comparison between disaggregated and non-disaggregated systems.

Streaming model weights directly to VRAM – Currently, we load model weights from storage into RAM, and then into VRAM. Leveraging FractOS's capabilities, we could instead co-ordinate an RDMA transfer from a remote storage node to the remote GPU node, which is more in line with disaggregation. While this wouldn't affect request metrics like prefill or decode time, it would allow llama.cpp to be spun up more quickly.

Chapter 9

Conclusion

We successfully demonstrate that efficient remote CUDA execution for LLM inference is achievable in disaggregated environments. Through the design and implementation of a comprehensive system built on FractOS, we have shown that the performance gap between local and remote GPU execution can be minimized to practically imperceptible levels when appropriate optimisations are applied.

Our CUDA interceptor provides a transparent interface that requires minimal modifications to existing applications, addressing one of the key limitations of previous approaches like rCUDA. Batched RPC operations are a key optimisation that fundamentally changes the performance characteristics of remote GPU execution, transforming the overhead from prohibitive to manageable. This optimisation proves particularly effective for the decode phase, where the lightweight computational requirements would otherwise be dominated by RPC overhead.

Our evaluation reveals several important insights. We found that the prefill phase performs almost as well remotely as locally, even without the batching optimisation. This is because FractOS RPCs can very effectively be overlapped with GPU work during the compute-heavy prefill phase. For the decode phase, our batched RPC optimisation achieves overhead as low as 7% compared to local execution for the small 7B model, demonstrating that disaggregated GPU execution is practical for LLM inference.

Furthermore, we do not observe any degradation in multi-client performance for the 7B and 13B models. For the 70B model, there is a slight increase in TPS degradation, from 2% to 5%, but larger experiments are needed to validate this.

The scalability characteristics of our system are favourable. Performance improves with model size, as the computational work grows faster than the communication overhead. For the Llama 2 70B model, our system has overheads of 1% and 2% for the prefill and decode phases respectively, suggesting that disaggregated approaches may become even more attractive as models continue to grow in size. However, more work is needed on the performance of our system for large numbers of GPUs (we were limited to four).

Beyond the immediate technical contributions, this work validates the broader vision of disaggregated computing for AI workloads. By demonstrating that complex, performance-critical applications like LLM inference can operate efficiently in disaggregated environments, we provide evidence that datacenter operators could realize significant improvements in resource utilization without sacrificing application performance. The capability-based security model of FractOS adds an additional layer of value, providing strong isolation guarantees that are essential in multi-tenant cloud environments.

In conclusion, this work provides a significant foundation for making disaggregated computing practical for AI inference workloads. By achieving performance competitive with local execution while providing the flexibility and utilisation benefits of disaggregation, we demonstrate that the future of datacenter computing need not be constrained by the limitations of monolithic machine designs. The techniques and insights developed in this project provide a foundation for future research and development in disaggregated AI systems, potentially enabling more efficient and flexible AI infrastructure at scale.

Bibliography

[1] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is All you Need. In: Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc.; 2017. Available from: https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[2] Llama 2: Open Foundation and Fine-Tuned Chat Models | Research - AI at Meta;. Available from: https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/.

[3] Jin X, Bai Z, Zhang Z, Zhu Y, Zhong Y, Liu X. DistMind: Efficient Resource Disaggregation for Deep Learning Workloads. IEEE/ACM Transactions on Networking. 2024 Jun;32(3):2422-37. Conference Name: IEEE/ACM Transactions on Networking. Available from: https://ieeexplore.ieee.org/abstract/document/10414009.

[4] Gao PX, Narayan A, Agarwal R, Karandikar S, Ratnasamy S, Carreira J, et al. Network Requirements for Resource Disaggregation;. .

[5] Duato J, Peña AJ, Silla F, Mayo R, Quintana-Ortí ES. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing & Simulation; 2010. p. 224-31. Available from: https://ieeexplore.ieee.org/document/5547126.

[6] ggml-org/llama.cpp. ggml; 2025. Original-date: 2023-03-10T18:58:00Z. Available from: https://github.com/ggml-org/llama.cpp.

[7] Vilanova L, Maudlej L, Bergman S, Miemietz T, Hille M, Asmussen N, et al. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In: Proceedings of the Seventeenth European Conference on Computer Systems. Rennes France: ACM; 2022. p. 352-67. Available from: https://dl.acm.org/doi/10.1145/3492321.3519569.

[8] Iqbal T, Qureshi S. The survey: Text generation models in deep learning. Journal of King Saud University - Computer and Information Sciences. 2022 Jun;34(6, Part A):2515-28. Available from: https://www.sciencedirect.com/science/article/pii/S1319157820303360.

[9] Akkaya IB, Kathiresan SS, Arani E, Zonooz B. Enhancing performance of vision transformers on small datasets through local inductive bias incorporation. Pattern Recognition. 2024 Sep;153:110510. Available from: https://www.sciencedirect.com/science/article/pii/S0031320324002619.

[10] Duan S, Shi Y, Xu W. Attention Bias as an Inductive Bias: How to Teach Transformers Simple Arithmetic; 2024. Available from: https://openreview.net/forum?id=Ei4bzOt8NG.

[11] Alammar J. The Illustrated Transformer;. Available from: https://jalammar.github.io/illustrated-transformer/.

[12] Le P, Zuidema W. Quantifying the Vanishing Gradient and Long Distance Dependency Problem in Recursive Neural Networks and Recursive LSTMs. In: Blunsom P, Cho K, Cohen S, Grefenstette E, Hermann KM, Rimell L, et al., editors. Proceedings of the 1st Workshop on Representation Learning for NLP. Berlin, Germany: Association for Computational Linguistics; 2016. p. 87-93. Available from: https://aclanthology.org/W16-1610/.

[13] Chien H, Turek J, Beckage NM, Vo VA, Honey C, Willke T. Slower is Better: Revisiting the Forgetting Mechanism in LSTM for Slower Information Decay. ArXiv. 2021 May. Available from: https://www.semanticscholar.org/paper/Slower-is-Better%3A-Revisiting-the-Forgetting-in-LSTM-Chien-Turek/27baa4f89a3314b34c1c876cfad4ae0a6f3b16d9.

[14] Kwon W, Li Z, Zhuang S, Sheng Y, Zheng L, Yu CH, et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. In: Proceedings of the 29th Symposium on Operating Systems Principles. SOSP '23. New York, NY, USA: Association for Computing Machinery; 2023. p. 611-26. Available from: https://dl.acm.org/doi/10.1145/3600006.3613165.

[15] LLM Inference Performance Engineering: Best Practices; 2023. Available from: https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices.

[16] Mastering LLM Techniques: Inference Optimization; 2023. Available from: https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/.

[17] dblp: LLM Inference Unveiled: Survey and Roofline Model Insights.;. Available from: https://dblp.org/rec/journals/corr/abs-2402-16363.html.

[18] llama.cpp/tools/server at master · ggml-org/llama.cpp;. Available from: https://github.com/ggml-org/llama.cpp/tree/master/tools/server.

[19] Achieve 23x LLM Inference Throughput & Reduce p50 Latency;. Available from: https://www.anyscale.com/blog/continuous-batching-llm-inference.

[20] Yu GI, Jeong JS, Kim GW, Kim S, Chun BG. Orca: A Distributed Serving System for {Transformer-Based} Generative Models; 2022. p. 521-38. Available from: https://www.usenix.org/conference/osdi22/presentation/yu.

[21] Scaling Laws for Neural Language Models;. Available from: https://www.researchgate.net/publication/338789955_Scaling_Laws_for_Neural_Language_Models.

[22] llama.cpp/tools/rpc at master · ggml-org/llama.cpp;. Available from: https://github.com/ggml-org/llama.cpp/tree/master/tools/rpc.

[23] Distributed Inference and Serving - vLLM;. Available from: https://docs.vllm.ai/en/latest/serving/distributed_serving.html#running-vllm-on-multiple-nodes.

[24] Ray Collective Communication Lib — Ray 2.46.0;. Available from: https://docs.ray.io/en/latest/ray-more-libs/ray-collective.html.

[25] Overview of NCCL — NCCL 2.27.3 documentation;. Available from: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html.

[26] NVLink & NVSwitch for Advanced Multi-GPU Communication;. Available from: https://www.nvidia.com/en-gb/data-center/nvlink/.

[27] Cuda kernel blocking launch - CUDA / CUDA Programming and Performance; 2023. Section: Accelerated Computing. Available from: https://forums.developer.nvidia.com/t/cuda-kernel-blocking-launch/246254/3.

[28] Some kernel launch is taking much longer (100x) than others in the same Cuda Stream - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums;. Available from: https://forums.developer.nvidia.com/t/some-kernel-launch-is-taking-much-longer-100x-than-others-in-the-same-cuda-stream/282394/2.

[29] Khatri DP, Song G, Zhu T. Heterogeneous Computing Systems.

[30] Papaioannou AD, Nejabati R, Simeonidou D. The Benefits of a Disaggregated Data Centre: A Resource Allocation Approach. In: 2016 IEEE Global Communications Conference (GLOBECOM). Washington, DC, USA: IEEE; 2016. p. 1-7. Available from: http://ieeexplore.ieee.org/document/7842314/.

[31] Baumann A, Barham P, Dagand PE, Harris T, Isaacs R, Peter S, et al. The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. Big Sky Montana USA: ACM; 2009. p. 29-44. Available from: https://dl.acm.org/doi/10.1145/1629575.1629579.

[32] Shan Y, Huang Y, Chen Y, Zhang Y. {LegoOS}: A Disseminated, Distributed {OS} for Hardware Resource Disaggregation; 2018. p. 69-87. Available from: https://www.usenix.org/conference/osdi18/presentation/shan.

[33] (PDF) First-generation Memory Disaggregation for Cloud Platforms; 2024. Available from: https://www.researchgate.net/publication/358951192_First-generation_Memory_Disaggregation_for_Cloud_Platforms.

[34] Apache Beam®;. Available from: https://beam.apache.org/.

[35] Kain RY, Landwehr CE. On Access Checking in Capability-Based Systems. IEEE Transactions on Software Engineering. 1987 Feb;SE-13(2):202-7. Conference Name: IEEE Transactions on Software Engineering. Available from: https://ieeexplore.ieee.org/abstract/document/1702200.

[36] (PDF) Capability Myths Demolished;. Available from: https://www.researchgate.net/publication/2927483_Capability_Myths_Demolished.

[37] Dennis JB, Van Horn EC. Programming semantics for multiprogrammed computations. Commun ACM. 1966 Mar;9(3):143-55. Available from: https://dl.acm.org/doi/10.1145/365230.365252.

[38] Wulf W, Cohen E, Corwin W, Jones A, Levin R, Pierson C, et al. HYDRA: the kernel of a multiprocessor operating system. Commun ACM. 1974 Jun;17(6):337-45. Available from: https://dl.acm.org/doi/10.1145/355616.364017.

[39] Needham RM, Walker RDH. The Cambridge CAP computer and its protection system. In: Proceedings of the sixth ACM symposium on Operating systems principles.

SOSP '77. New York, NY, USA: Association for Computing Machinery; 1977. p. 1-10. Available from: https://dl.acm.org/doi/10.1145/800214.806541.

[40] Hille M, Asmussen N, Bhatotia P, Härtig H. SemperOS: A Distributed Capability System.

[41] Gupta V, Schwan K. Brawny vs. Wimpy: Evaluation and Analysis of Modern Workloads on Heterogeneous Processors. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum; 2013. p. 74-83. Available from: https://ieeexplore.ieee.org/document/6650873.

[42] Li H, Berger DS, Hsu L, Ernst D, Zardoshti P, Novakovic S, et al. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery; 2023. p. 574-87. Available from: https://dl.acm.org/doi/10.1145/3575693.3578835.

[43] Kachris C. A Survey on Hardware Accelerators for Large Language Models. Applied Sciences. 2025 Jan;15(2):586. ArXiv:2401.09890 [cs]. Available from: http://arxiv.org/abs/2401.09890.

[44] Silvano C, Ielmini D, Ferrandi F, Fiorin L, Curzel S, Benini L, et al. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. ACM Comput Surv. 2025 Apr. Just Accepted. Available from: https://dl.acm.org/doi/10.1145/3729215.

[45] 1. Overview — GPUDirect RDMA 12.9 documentation;. Available from: https://docs.nvidia.com/cuda/gpudirect-rdma/.

[46] NVIDIA Display Mode Selector Tool;. Available from: https://developer.nvidia.com/displaymodeselector.

[47] Fielding RT, Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Internet Engineering Task Force; 2014. RFC 7230. Num Pages: 89. Available from: https://datatracker.ietf.org/doc/rfc7230.

[48] libcpr/cpr. libcpr; 2025. Original-date: 2015-03-31T01:10:42Z. Available from: https://github.com/libcpr/cpr.

[49] Load testing for engineering teams | Grafana k6;. Available from: https://k6.io.

Chapter 10

Declarations

10.1 Use of generative AI

We used Claude 3.5/3.7/4 (Anthropic, https://claude.ai/new) and Gemini 2.5 Pro (Google, https://aistudio.google.com/prompts/new_chat) for the following tasks:

- Performing certain routine technical tasks, such as asking the model to generate complex Bash commands that would be laborious to do by hand.
- Identifying potential sources using the 'search' feature, then sifting through them to find ones suitable for this report.
- Summarising technical concepts for my own understanding.

10.2 Ethical considerations

As our project is on the topic of LLM inference, we consider the ethical implications of LLMs and generative AI more broadly. Unfortunately, generative AI poses opportunities for a wide range of societal ills, such as deep-fakes, automated scams, and large-scale disinformation. However, we do not believe our project elevates these capabilities beyond their current level, so we do not see any direct ethical issues with our project.

10.3 Sustainability

We minimised energy usage throughout this project by avoiding running unnecessary experiments, and generally taking care to avoid performing obviously wasteful computation. We regularly checked to make sure there were no idle processes running on the GPU, since GPUs have high power usage.

10.4 Availability of data and materials

All of our code is available in repositories at https://lsds.doc.ic.ac.uk/gitlab/groups/fractos. Access is available on request from authorised members of the LSDS group within the Department of Computing.