Serving Large Mixture of Experts (MoE) models using Edge GPU

Shivaz Sharma

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

The expansion of large language models (LLMs) based on the Transformer architecture has yielded substantial advancements in various natural language processing tasks. Nevertheless, this escalation in scale has led to a corresponding rise in computational intricacy. Sparse models, drawing inspiration from the Mixture of Experts (MoE) approach, offer a promising avenue for sustaining the growth of language models while mitigating their computational demands. Enhancing their speed during inference and enabling their deployment on GPU-constrained devices would significantly lower the obstacles to adopting sparse LLMs. Conversely, emerging computing platforms such as Edge GPUs, exemplified by NVIDIA Orin, play a pivotal role in executing critical machine-learning operations for robotics, self-driving vehicles, and unmanned aerial vehicles. However, these Edge GPUs lack the capacity to accommodate continuously expanding ML models, including large GPT and MoE-based architectures. This research delves into the unified memory architecture of Edge GPUs, crafting a system known as Archer-Edge that aligns with this architecture. The investigation demonstrates that Archer-Edge demonstrates a notable **2.2x speed-up** in inference latency when compared to the in-house library Archer. Moreover, Archer achieves a significant **14x speed-up** in inference latency when compared to the Accelerate library.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Shivaz Sharma*)

Acknowledgements

I would like to express my deepest gratitude and heartfelt appreciation to my supervisor, Dr. Luo Mai, for his unwavering support and guidance throughout the journey of completing this thesis. He motivated me with the goal that your research could be impactful to make a GPT-enabled Robot work without the internet. I would also like to thank his Ph.D. student Leyang Xue, for his invaluable support and for helping me at each step whenever I was stuck. Lastly, I would like to thank my friends and family for supporting me throughout the year.

Table of Contents

Chapter 1

Introduction

Large language models (LLMs) are essential for various reasons, including their advanced natural language processing (NLP) capabilities, improved generative capabilities, increased efficiency, and versatility in a wide range of applications [31], [21]. They have been successful in tasks such as language translation, content generation, and chatbot development [21]. LLMs can process vast amounts of text data and learn from it, enabling them to capture the nuances of human language and generate text that is indistinguishable from what a human might write. They are used in various fields, including healthcare, software development, and many other use cases [3], [10].

LLMs have significantly increased memory requirements and computation costs due to their massive size and the complexity of their architectures. The memory requirements stem from the need to store a large number of parameters, which can range from hundreds of millions to trillions, depending on the model size [2], [26]. An encouraging approach to sustain the expansion of LLMs while curbing their computational intricacies is through sparsity. Shazeer et al. [28] introduced the Sparsely-Gated Mixture-of-Experts layer (MoE) to augment model capacity and enhance training efficiency. Examples of sizable MoE models encompass Meta's NLLB [30], Google's Switch-Transformer [14], and OpenAI's GPT-4 [24]. The volume of the MoE model is notably impacted by the count of experts it encompasses.

Storing large models in GPUs is certainly a major challenge. Currently, researchers are using techniques like pruning, quantization, knowledge distillation, and Neural architecture search(NAS) for compressing deep learning models to fit in the GPU memory [19], [25]. Challenges exist with model compression methods, as compressed models tend to overfit on easy samples while struggling to generalize the complex ones [9]. Additionally, heavily compressed models can exhibit poorer performance compared to lightly compressed models [23].
The widespread adoption of smart devices and Internet of Things (IoT) sensors, has caused a substantial increase in data generation at the edge of the Internet. The

effective real-time analysis of this substantial data volume, especially by utilizing precise deep learning models, often necessitates processing the data in proximity to the data sources (at the edge of the Internet) to decrease network and processing delays [20], [17], [18]. Edge devices generally possess limited computational and memory capabilities in comparison to cloud servers. This disparity in resources poses challenges when attempting to deploy extensive models on edge devices without appropriate optimization [29]. Techniques such as model compression, aimed at accommodating models on Edge GPUs, can impact both inference efficiency and the quality of results (QOR).

Efforts have been made to tackle the challenges of large model inference. Aminabadi et al. [1] introduced the concept of "DeepSpeed Inference," which presents a heterogeneous inference methodology that extends beyond the constraints of GPU memory. It makes use of CPU and NVMe memory in addition to GPU memory resources. This combined utilization facilitates high inference throughput, particularly for larger models that exceed the capacity of aggregate GPU memory. There are other solutions as well trying to achieve similar objectives like Accelerate by Hugging Face [13].

Presently, **no solution is available to facilitate inference with large models that leverage the unified memory architecture of Edge GPUs**.In this study, our objective is to effectively deploy an MoE model (nllb-moe-54b) with a checkpoint size of 206 GB onto an Edge GPU equipped with a GPU memory capacity of 32 GB.

1.1 Research contribution

- Successfully deployed a large MoE model (nllb-moe-54b) having a checkpoint size of 206 GB on an Edge GPU (Nivdia's Orin) platform with an available GPU memory of 32 GB.
- Achieved a speed-up of 2.2x in inference latency with Archer-Edge in comparison to the in-house library Archer.
- Archer achieved 14x speed-up in inference latency in comparison to Accelerate.

In this section, we have elucidated both the research objective and the contributions made within this study. The subsequent sections provide a comprehensive breakdown of our research journey. Section 2 expounds upon the foundational knowledge necessary for a thorough investigation. Section 3 details our exploration of the unified memory architecture of Edge GPUs. In Section 4, we elaborate on the steps taken in the implementation process to augment Archer for the development of Archer-Edge. This section also encompasses discussions regarding the benchmarking of both Archer and Archer-Edge. Subsequently, Section 5 rigorously

evaluates the benchmark results, offering a performance comparison among Accelerate, Archer, and Archer-Edge. Lastly, Section 6 encapsulates the conclusive elements of this thesis paper.

Chapter 2

Background

This segment of the thesis paper furnishes vital contextual information that aids readers in comprehending the rationale, significance, and pertinence of the study. Section 2.1 delves into the intricacies of the Transformer and MoE architecture, which holds pivotal importance in comprehending the architecture of an MoE model. This section also expounds upon the practical applications of MoE models. In Section 2.2, the process of inference within the Transformer model is explicated. This elucidation covers the token generation procedure, emphasizing both the single encoder iteration and multiple decoder iterations. This understanding becomes valuable when calculating input and output throughput metrics for benchmarking purposes. Section 2.3 elucidates the architectural framework of Nvidia's Orin GPU, highlighting the distinctions between server GPUs and Edge GPUs, with a specific focus on the contrast between unified and non-unified memory architectures.

2.1 Mixture of Experts(MoE) models

The Mixture of Experts (MoE) is a machine learning methodology that involves partitioning a problem domain into distinct regions using multiple specialized networks [28], Wikipedia. This approach, which leverages conditional computation, has found application in language modeling to enhance both model capacity and efficiency [28]. In MoE-based language models, the input sequence is divided into segments, with each segment assigned to a specific expert network. To consolidate the outputs of these expert networks, a gating network is employed, which determines the appropriate expert for each input segment.

2.1.1 Architecture

Prior to delving into the architecture for MoE (Mixture of Experts) models, it is crucial to gain some understanding of the basics behind a Transformer architecture. This knowledge will be helpful, as many MoE models, such as Switch Transformer

[14] and NLLB [30], incorporate the MoE technique within the framework of an encoder-decoder based Transformer architecture. Many cutting-edge language models, such as BERT, GPT, and XLNet, now depend heavily on transformers [32].
The Transformer model is a deep learning architecture that has gained prominence in NLP applications such as machine translation, document categorization, and sentiment analysis. The Transformer model is built on an encoder-decoder architecture, with the encoder processing the input sequence and the decoder producing the output sequence. Each layer of the encoder and decoder incorporates sublayers such as self-attention, feedforward neural networks, and normalization layers [31]. The Transformer architecture illustration is shown in figure 2.1.
Key Components of Transformer:

- Encoder-Decoder Architecture: The Transformer model is built on an encoder-decoder architecture, where the encoder processes the input sequence, and the decoder generates the output sequence in a step-by-step manner.
- Multi-Head Self-Attention: Multi-head self-attention is a key component of the Transformer model. It enables the model to focus on different positions in the input sequence with varying weights, allowing it to capture diverse relationships and dependencies within the sequence. By employing multiple attention heads, the model can better understand and represent the input data.
- Positional Encoding: Transformers lack the inherent positional information found in recurrent models. To address this, positional encodings are added to the input embeddings to indicate the position of each word in the sequence. This helps the model understand the order of words in a sentence and improves its performance in sequence-to-sequence tasks.
- Feed-Forward Neural Networks: After the self-attention mechanism, the Transformer model uses feed-forward neural networks to perform non-linear transformations on the attended representations. These networks help in further processing the information and generating more complex representations of the input data.
- Layer Normalization and Residual Connections: Layer normalization and residual connections are techniques employed in the Transformer model to stabilize the training process and mitigate the vanishing gradient problem. These techniques help in maintaining the flow of information through the network and ensure that the model can be trained effectively.

Key components of Mixture of Experts (MoE) model:

- Experts and Gating Mechanism: The Mixture of Experts (MoE) model consists of multiple ”experts,” each responsible for handling specific input data regions. A ”gating mechanism” is employed to determine which expert should be activated for a given input, allowing the model to adapt to different parts of the problem space [14].

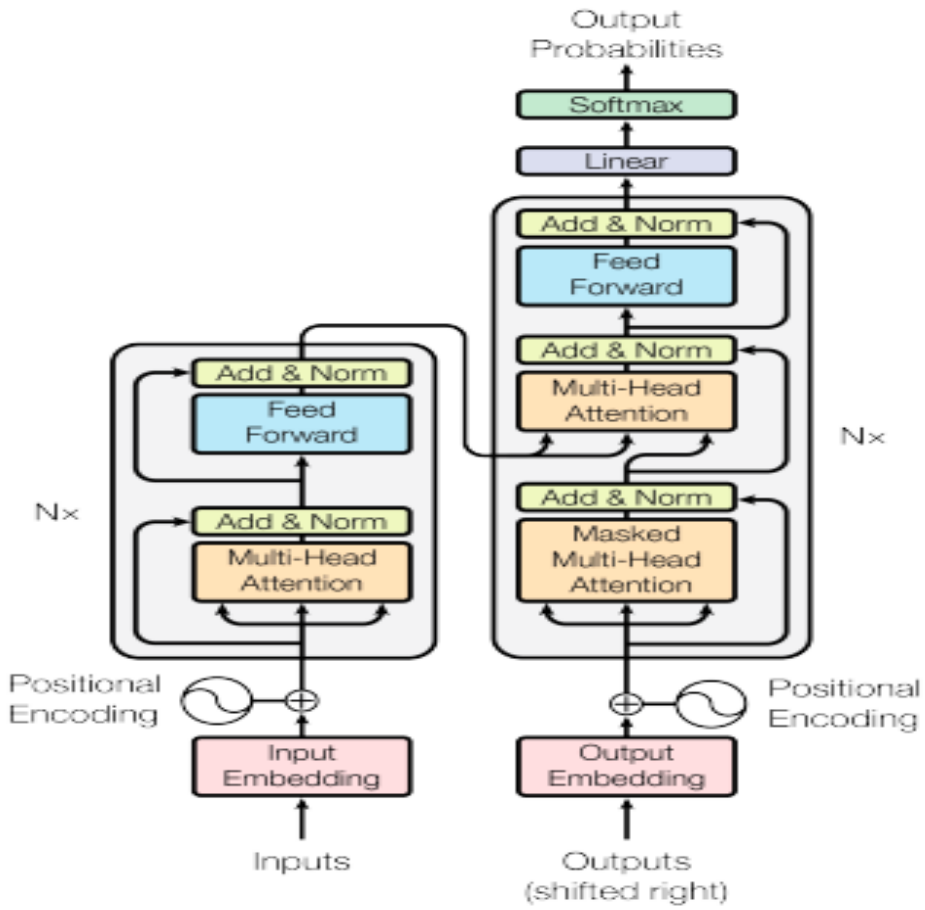


Figure 2.1: Encoder-Decoder based Transformer Model Architecture. (source: [31])

- Mixture Weights: The gating mechanism generates mixture weights that determine the contribution of each expert to the final prediction. These weights are typically learned during the training process, allowing the model to optimize the combination of expert outputs [14].

In the Transformer model, the Mixture of Experts (MoE) can be used to replace the feed-forward layer in specific Transformer blocks, allowing the model to increase its capacity without a proportional increase in computation. In a standard Transformer model, the feed-forward layer is a dense layer that applies the same weight matrix to each token position in the input sequence. In contrast, when using MoE in a Transformer model, the feed-forward layer is replaced by a group of independent feed-forward networks, each acting as an ”expert”. The gating mechanism generates a mixture of weights that determine the contribution of each expert to the final prediction. These weights are typically learned during the training process, allowing the model to optimize the combination of expert outputs. The MoE model can be used in an encoder-decoder based Transformer model by replacing the feed-forward layer with a group of independent feed-forward networks, each acting as an ”expert”. Please see the illustration 2.2 for better understanding.

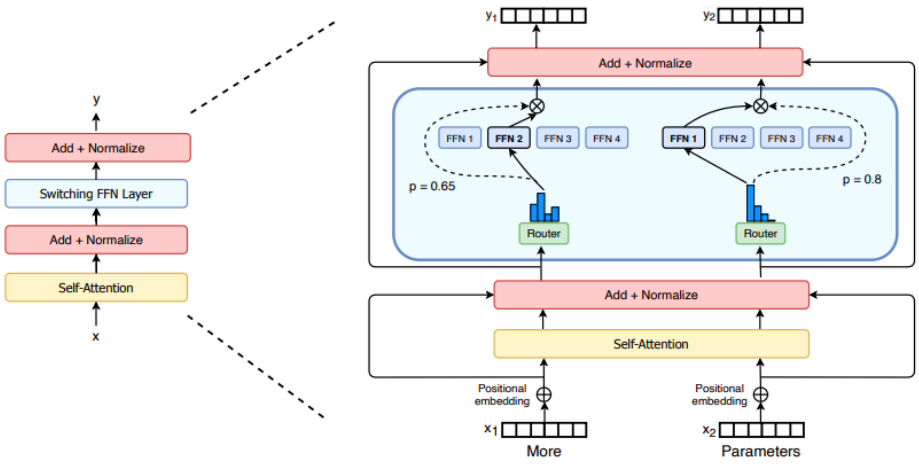


Figure 2.2: Switch-Transformer MoE model Experts Network. (source: [14])

2.1.2 Applications

Following are the applications of MoE models:

- Speech Recognition: MoE models have been applied to speech recognition tasks, particularly in noisy environments, by combining linear dynamic models with a mixture of experts architecture [33].
- Deep Learning: MoE layers have been incorporated into deep learning models to scale up model capacity while maintaining computational efficiency [5].
- Real Estate Appraisal Models: MoE models have been used to construct real estate appraisal models, providing a more accurate and reliable estimation of property values [16].
- Rank Data Analysis: MoE models have been used to analyze rank data in election studies, providing insights into the preferences of voters and the composition of the electorate [15].
- Machine Translation: MoE can be applied to machine translation tasks by employing separate experts for different language pairs or specific translation challenges, such as rare words or idiomatic expressions.
- Natural Language Generation: In tasks like text summarization or dialog generation, MoE can be employed to have distinct experts responsible for generating different types of content, resulting in more diverse and accurate outputs.

2.2 Understanding Encoder Decoder Iterations during Inference

For benchmarking Archer & Archer-Edge, it's essential to grasp the fundamentals of the inference process within a transformer model. The comprehensive description of the inference process outlined below draws its inspiration from a video tutorial provided by HuggingFace.

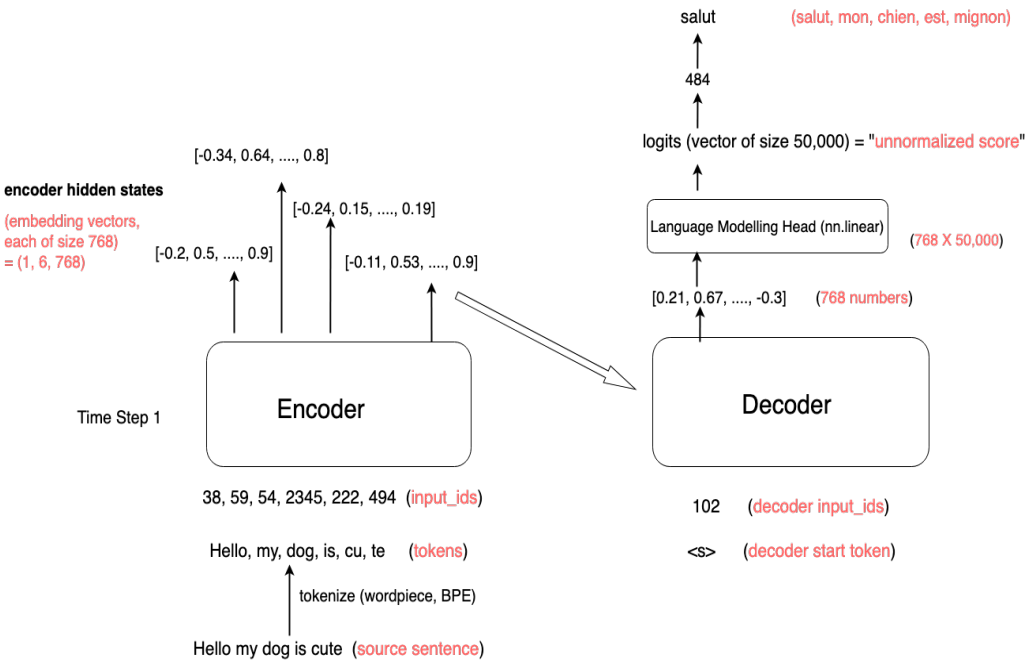


Figure 2.3: Inference at timestep 1

The illustrations presented in 2.3, 2.4, 2.5, offer high-level visual representations utilized to elucidate the process of token generation within the language model. In our context, we employ a transformer model to accomplish a machine translation task. Specifically, we are translating an English sentence, "hello my dog is cute," into French, yielding "salut mon chien est mignon." During the first time step (timestep 1) illustrated in figure 2.3, the initial task involves tokenizing the input sentence, i.e., segmenting the sentence into words or subwords. To facilitate this, we utilize a tokenizer such as wordpiece, BPE (Byte-Pair Encoding), or an alternative variant. Subsequently, these generated tokens are transformed into numerical values known as input_ids. These input_ids essentially correspond to a dictionary mapping that encompasses all the tokens supported by the transformer model. This mapping, also referred to as the model's vocabulary, is integral to the process.

These input_ids are fed into the encoder, responsible for converting each id into an embedding vector, often referred to as last_hidden_states. These vectors are aptly

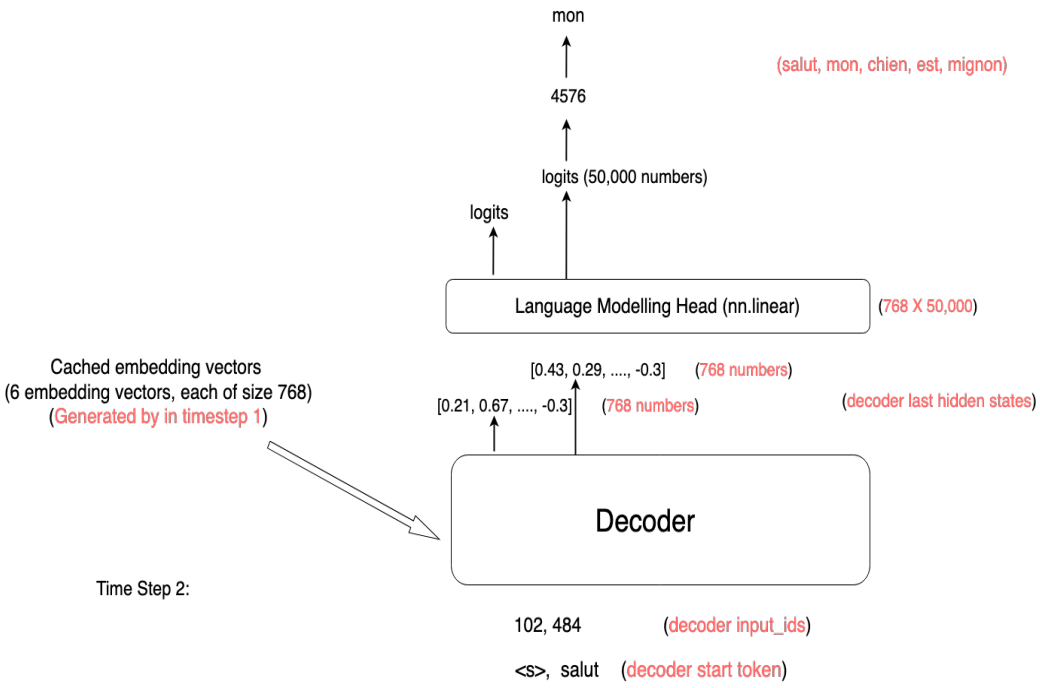


Figure 2.4: Inference at timestep 2

termed, as they emerge from the encoder's operations. Generally, an embedding vector's dimensions are 768 in a base model, signifying that, in our context, each token gets represented by a 768-dimensional vector. For instance, in our illustrative scenario, each token is represented by such a vector. In our example, after the encoder's processing concludes, we are left with encoder last_hidden_states, depicted as (1, 6, 768) encompassing all six input tokens.

Conversely, at timestep 1, the decoder involves converting a special token denoted as start_sequence into an id using the model's vocabulary. Subsequently, a solitary token, as part of the decoder input_id, traverses the decoder. The six hidden_states generated during the encoder phase are integrated into the decoder process. Upon the decoder's conclusion, a solitary embedding vector of size 768 emerges for the decoder input. This embedding vector undergoes processing via a language modeling head(matrix multiplication operation). The dimensions of this language modeling head's matrix are 768 x 50,000. Consequently, this operation yields an unnormalized logits vector of size 50,000, representing scores for tokens within the model's vocabulary, which are not yet adjusted. To finalize the process, the transformer model employs a greedySearch algorithm. This algorithm essentially entails selecting the token id linked to the highest unnormalized score. Based on the chosen token id, the corresponding word in the model's vocabulary is identified.

Upon reaching time step 2 as illustrated in figure 2.4, an encoder pass is unnecessary, as cached hidden_states values are utilized. However, during the decoder phase,

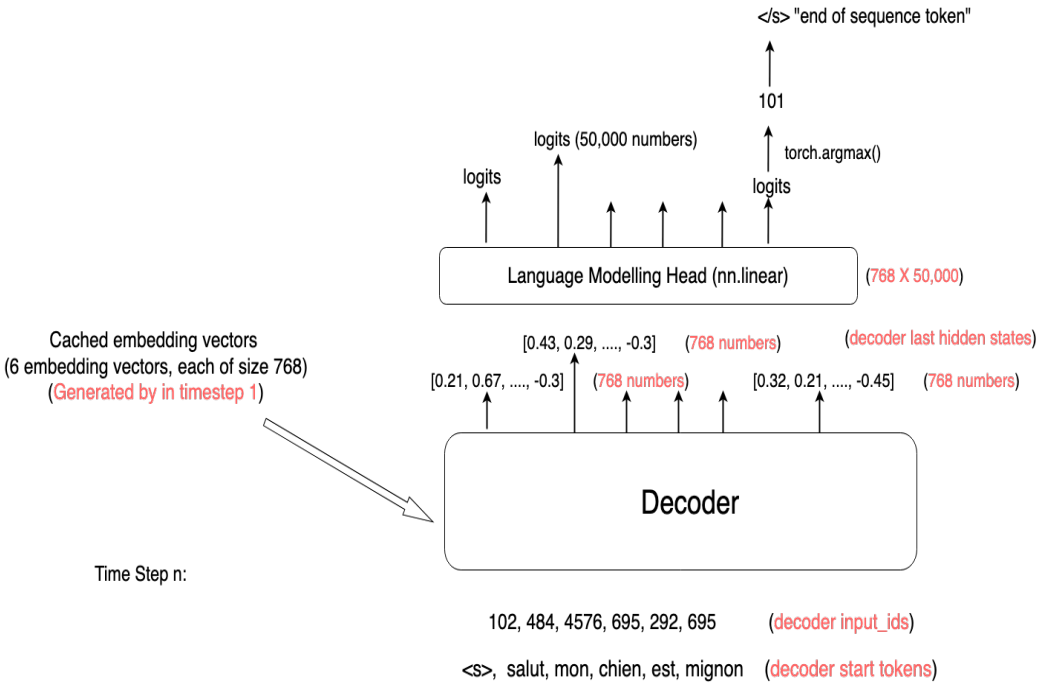


Figure 2.5: Inference at timestep n

an addition is made: the most recently generated token becomes part of the new input_ids for the decoder input. This augmentation serves the purpose of generating a fresh token. In our example, while at timestep 1 only the start of the input sequence was fed, at timestep 2, the token id for 'salut' (generated in the initial decoder pass) is supplied. Subsequently, following the decoder's conclusion, the process outlined earlier is reiterated, allowing the creation of a new token, 'mon,' in this case.

As the sequence advances to time step n shown in 2.5, culminating in the last token's generation within the decoder pass, it is again introduced into the decoder. This instance births the end_of_sequence token. Upon its creation, the transformer model receives the signal that the inference process has reached completion.

2.3 Edge GPU Architecture

Understanding Edge GPU architecture is a critical aspect of this thesis paper since this paper focuses on enhancing the efficiency and deployment of large MoE models on low-resource machines, Edge GPUs serve as a crucial component of this endeavor as it allows us to grasp the constraints and capabilities of this hardware platform.

2.3.1 Motivation

Edge computing has grown in popularity over the past several years, due to its capacity to carry out data processing and analysis closer to the source of the data, reduce latency, and increase efficiency. The usage of GPUs for inference of big learning models is one of the main elements of Edge computing [4].
Using cloud computing is a popular method for meeting the computational demands of deep learning. Data must be transferred from the network Edge location of the data source such as smartphones and Internet of Things (IoT) sensors to a centralized point in the cloud in order to utilize cloud resources.
There are various obstacles associated with this possible solution of shifting the data from the source to the cloud.

- Latency: For many applications, real-time inference is essential. For instance, an autonomous car's video frames must be processed fast in order to identify and avoid obstacles, or a voice-activated assistant programme must comprehend the user's request and respond to it in a timely manner. The strict end-to-end low-latency requirements required for real-time, interactive applications cannot be met by sending data to the cloud for inference or training, as this may result in additional network queuing and propagation delays. For instance, actual experiments have shown that the entire process of offloading a camera frame to an Amazon Web Services server and carrying out a computer vision task takes more than 200 ms [27].

- Scalability: As the number of connected devices rises, network connectivity to the cloud may become congested, posing a scalability problem when data is sent from the sources to the cloud. In terms of network resource usage, uploading all data to the cloud is also inefficient, especially if the deep learning process does not require all of the data from all sources. Video streams and other bandwidth-intensive data sources should be taken very seriously[4].

- Privacy: When sending data to the cloud, people who either own the data or whose behaviours are recorded in the data may have privacy concerns. Users can be concerned about how the cloud or application will use their sensitive data if they upload it to the cloud (such as their faces or speech).

The issues with latency, scalability, and privacy mentioned earlier in this section can be resolved by Edge computing. A tiny mesh of computer resources is used in Edge computing to provide computational power near the end devices [4].

2.3.2 Nvidia's Orin GPU

Nvidia released a new series of Jetson AGX Orin GPUs. The Jetson AGX Orin series includes the Jetson AGX Orin 64GB and the Jetson AGX Orin 32GB modules. All the experiments in this thesis are performed on Jetson AGX Orin 32 GB platform. Orin architecture illustration is shown in 2.6.

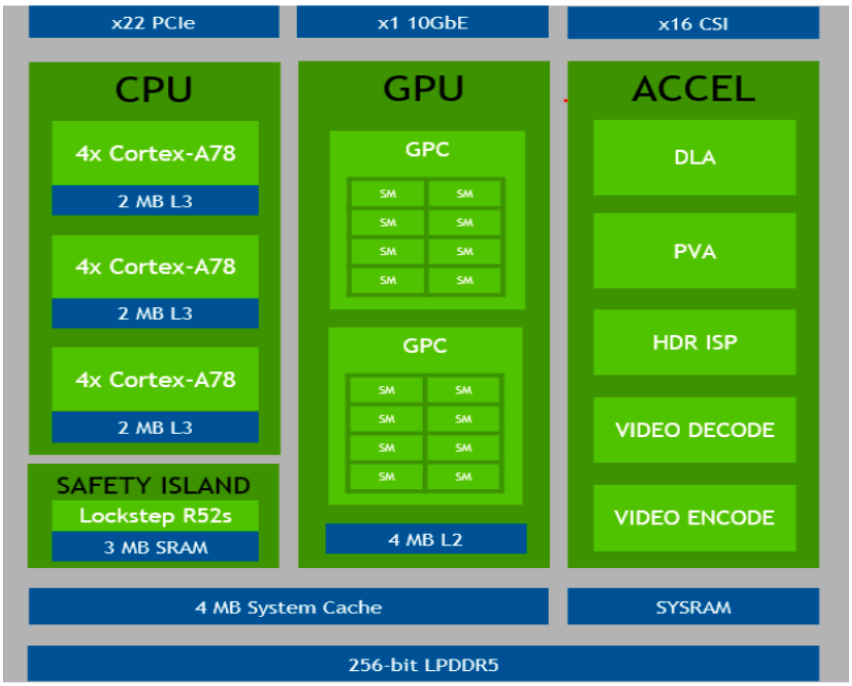


Figure 2.6: Nvidia Orin GPU Architecture. (source: [22])

To empower autonomous systems, the NVIDIA® Jetson AGX OrinTM series provides server-class capabilities, boasting an impressive AI performance of up to 275 trillion operations per second (TOPS). The NVIDIA Orin System-on-Chip (SoC) features a comprehensive suite of components, including an NVIDIA Ampere architecture GPU, an Arm Cortex-A78AE CPU, advanced deep learning and vision accelerators, a video encoder, and a video decoder. These components are integrated into Jetson AGX Orin modules, which offer a memory bandwidth of 204 GB/second, high-speed IO interfaces, and either 32 or 64 GB of DRAM. These specifications enable the modules to concurrently support multiple AI application pipelines [22]. Additional information is available in Table 2.1.

2.3.3 Unified versus Non-Unified Memory Architecture

Understanding the differences between unified and non-unified memory architecture is critical for the context of this thesis. A major difference between a normal server GPU and an Edge GPU is the memory architecture shown in image 2.7.
Any processor in a system can access unified memory, which is a single memory address space as shown in 2.8. Using this hardware/software technology, Applications can allocate data that can be read or written from code running on either CPUs or GPUs.
In non-unified memory architecture, the memories of both CPU and GPU are physically distinct and separated by a PCIe express bus. If there is any data that is shared by both the CPU and GPU must be allocated in their distinct memories and the

Feature	Jetson AGX Orin 32GB
AI Performance	200 TOPS (INT8)
GPU	NVIDIA Ampere architecture with 1792 NVIDIA® CUDA® cores and 56 Tensor Cores
Max GPU Freq	930 MHz
CPU	8-core Arm® Cortex®-A78AE v8.2 64-bit CPU 2MB L2 + 4MB L3
CPU Max Freq	2.2 GHz
Memory	32GB 256-bit LPDDR5 204.8 GB/s
Storage	64GB eMMC 5.1
UPHY*	Up to 2 x8, 1 x4, 2 x1 (PCIe Gen4, Root Port and Endpoint)
	3x USB 3.2

Table 2.1: Jetson Architecture Specifications. (source: [22])

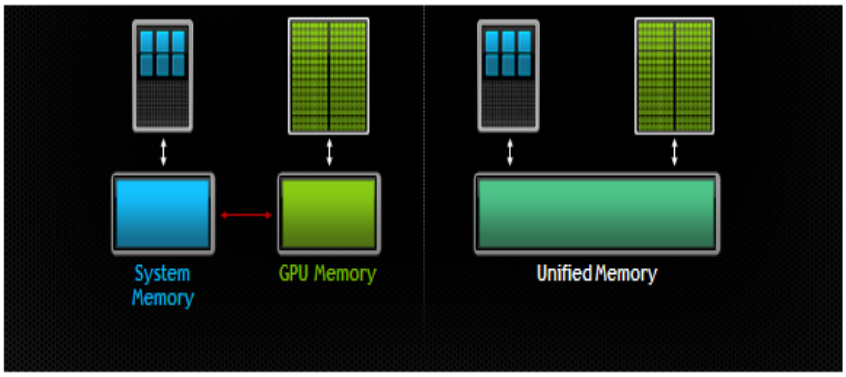


Figure 2.7: Unified Vs Non-Unified Memory View. (source: [7])

developer has to explicitly copy data between the memories.

The major advantages that unified memory provides over non-unified memory architecture are as follows

- Simpler Programming and Memory Model Programmers no longer need to spend time on the specifics of allocating and copying device memory; instead, they may go right into creating parallel CUDA kernels [7].
- Performance Through Data Locality Unified Memory can provide the performance of local data on the GPU while giving the usability of globally shared data by transferring data between the CPU and GPU on demand. The CUDA driver and runtime keep the complexity of this capability hidden, making it

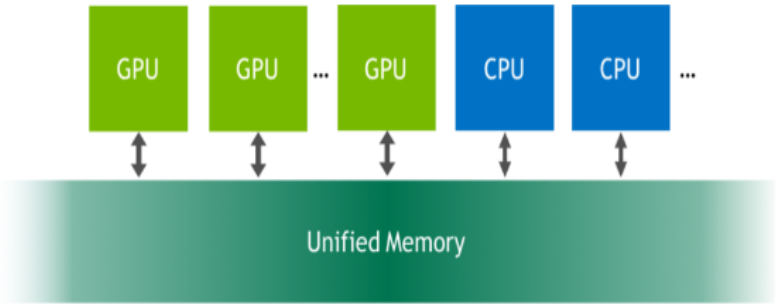


Figure 2.8: Unified Memory Architecture. (source: [7])

easier to create application code. The goal of migration is to utilize each processor's full bandwidth; a Kepler GPU's compute throughput depends on the 250 GB/seconds of GDDR5 memory [7].

Chapter 3

Preliminary Study

The preliminary investigation segment of this thesis functions as a fundamental exploration that establishes the foundational framework for subsequent research endeavors. Section 3.1 delves into the discussion of baseline libraries such as Accelerate and Archer. These libraries are harnessed for tasks such as profiling, bottleneck analysis, and performance benchmarking. In 3.2, we engaged in experimental activities on the Orin GPU platform, aiming to delve into the nuances of the unified memory architecture and gain more comprehension of the associated CUDA APIs and underlying memory transfers. In 3.3, we meticulously assessed the disk performance on the Orin platform, as offloading parameters to the disk constitute a pivotal element of this study. This data proves instrumental in the comparative analysis of channel bandwidth.
Experiments were performed on an Nvidia Orin GPU server, utilizing the Jetson developer kit to install Pytorch (version 2.0.0+nv23.05) and other necessary dependencies. Miniconda with Python 3.8 was employed to create and manage the environment, while the utilization of Archer (version 0.0.1) facilitated the process. We have employed Nvidia's Nsight tool [6] to conduct all the profiling activities.

3.1 Baselines

For this paper, we are considering Accelerate by Hugging Face [13] and in-house library Archer for big model inference since they support SSD offloading of parameters for running large models on memory-constrained GPU. We also tried to use DeepSpeed Inference [1] as a baseline but it was not compatible with the Orin GPU platform.

3.1.1 Introduction to Accelerate Library by Hugging Face

Hugging Face open-sourced Accelerate library to enable us to run large models without a supercomputer. Accelerate leverages the functionality provided in Py-

Torch. Gaining a comprehensive understanding of the operations of the Accelerate library holds significance as it provides valuable insights into the mechanics of the Accelerate interface. This understanding extends to the fundamental algorithms that facilitate the transfer of parameters, a key aspect for offloading, and proves instrumental in the subsequent bottleneck analysis discussed in section 4.1.1. Steps involved in Accelerate for deploying a large model on memory-constrained GPU are shown in 3.1.

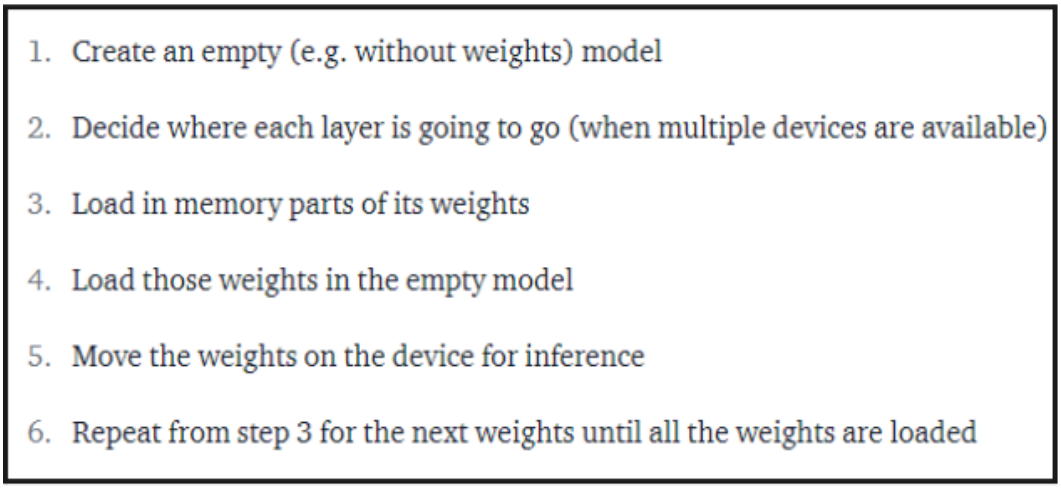


Figure 3.1: Steps to run Inference with Accelerate. (source: [13])

For creating an empty model Accelerate leverages the PyTorch support of meta device, as a result, they can generate tensors that don't have any data associated with them because a tensor on a meta device only requires a shape. They can thus generate tensors of any size while on the meta device without having to worry about CPU (or GPU) memory.
One critical component of Accelerate is computing the device map which decides where each layer/parameter/weight is going to reside before loading the pre-trained weights. This allows them to release CPU RAM each time a weight has been properly placed. Since memory occupied by weight/parameter can be computed using the shape of each tensor and its dtype(data type), this can be done with the empty model on the meta device.
Accelerate provides options like infer_auto_device_map which can automatically determine the device_map that will try to maximize the use of all available GPUs, then CPU RAM, and finally flag the weights that don't fit for disk offload. Users can provide their own device map as well. The figure shown in 3.2 is an example of a device map created by Accelerate.

The load_checkpoint_and_dispatch() method is the second tool that Accelerate introduces, and it enables you to load a checkpoint inside of an empty model. This supports both sharded checkpoints and full checkpoints, which consist of a single file containing the entire state dict. When loading a checkpoint that is sharded, the maximum RAM use will be equal to the size of the largest shard because it will

```
'model.decoder.embed_tokens': 0,
 'model.decoder.embed_positions': 0,
 'model.decoder.final_layer_norm': 0,
 'model.decoder.layers.0': 0,
 'model.decoder.layers.1': 0,
 ...
 'model.decoder.layers.9': 0,
 'model.decoder.layers.10': 'cpu',
 'model.decoder.layers.11': 'cpu',
 ...
 'model.decoder.layers.17': 'cpu',
 'model.decoder.layers.18': 'disk',
 ...
 'model.decoder.layers.39': 'disk',
 'lm_head': 'disk'}
```

Figure 3.2: Example of Device Map. (source: [13])

automatically distribute those weights across the devices you have available (GPUs, CPU RAM).
Model Inference using Accelerate Library is explained in the later section 4.1.

3.1.2 Introduction to Archer

Gaining a comprehensive understanding of Archer holds significant importance in the context of this paper, as it forms a crucial prerequisite for deploying it on the Edge GPU. This knowledge also proves valuable in the development of Archer-Edge and is instrumental in effectively profiling the Archer library.
Archer is an open-source high-performance inference engine developed by the research team led by Dr. Luo Mai at the University of Edinburgh. It is designed to optimize the efficiency and throughput of machine learning models. The project aims to create an easy-to-use and extendable engine with a focus on performance optimization, GPU resource management, and seamless integration with various applications.
Archer aims to provide a powerful and efficient solution for deploying machine learning models in real-world scenarios. **Currently, Archer does not support the unified memory architecture of the Edge GPU, this will be one of the major aims of this dissertation to enhance Archer**.

3.1.3 Prefetching offloaded experts algorithm in Archer

The following information regarding the prefetching algorithm is based on the work conducted on Archer by Leyang Xue, a PhD student under the supervision of Dr.

Luo.

The most distinctive feature of Archer, which sets it apart from existing solutions like Accelerate and Deepspeed-MoE, is its 'sparsity aware prefetching' of experts during inference. This algorithm leverages the observation that only a sparse subset of experts is activated during inference, typically about 20% of all the experts in large MoE models, such as Switch Transformer [14] and NLLB [30].

The core idea behind this approach is to automatically store the experts in external memory by default. When an inference request is received, the required experts are proactively fetched to the GPU memory in advance, ensuring efficient and speedy inference with high throughput and minimal delay. This efficient management of expert data optimizes computational resources during inference and leads to cost savings.

To determine which experts should be activated in advance and which ones should be offloaded from the GPU for replacement, Archer introduces two main features: "Expert Activation Predictor (EAP)" and "EAP-Guided MoE Inference Engine."

The EAP is designed based on the insight that once an expert is activated, subsequent layers tend to follow a skewed probability distribution, where certain experts are more likely to be activated than others. To exploit this insight, the MoE model is represented using an expert activation tracing graph.

The EAP-Guided MoE Inference Engine in Archer utilizes the probabilities generated by EAP as cues to determine the relative priority of experts to keep in the GPU memory, DRAM, and SSD. It treats DRAM and SSD as additional layers in a hierarchical cache of experts and also considers the order and occupancy of PCIe link queues for efficient data movement.

Overall, Archer's sparsity-aware prefetching and the EAP-Guided MoE Inference Engine significantly enhance the performance and efficiency of MoE models, particularly in handling large and sparse data, leading to improved inference speed and cost-effective resource utilization.

3.2 Exploration of unified memory architecture

Conducting preliminary experiments assumes significance in order to gain deeper insights into memory architecture and the underlying cuda APIs. This foundational understanding becomes pivotal for the subsequent stages of development. Aligned with theoretical expectations, we anticipate that the *cudaMallocManaged()* API would demonstrate enhanced memory efficiency when compared to the *cudaMalloc()* API.

```
#include <iostream>
#include <cuda_runtime.h>
#define N 999

// CUDA kernel to initialize an array

__global__ void initializeArray(int* d_array)
{
    int tid = blockIdx.x * blockDim.x
              + threadIdx.x;
    if (tid < N)
        d_array[tid] = tid;
}

int main()
{
    int h_array[N]; // Host array
    int *d_array; // Device array

    /* Allocate cudaMalloc memory
    *  accessible only on GPU */
    cudaMalloc((void**)&d_array,
               N * sizeof(int));

    // Launch the kernel to initialize
    *  the device array */
    initializeArray<<<1, N>>>(d_array);

    /* Wait for GPU to finish
    * before accessing on host */
    cudaDeviceSynchronize();

    // We need a deep-copy from Device
    *  GPU to Host CPU */
    cudaMemcpy(h_array, d_array,
              N * sizeof(int),
              cudaMemcpyDeviceToHost);

    // Print the result of h_array
    std::cout << "Array elements: ";
    for (int i = 0; i < N; i++)
        std::cout << h_array[i] << " ";
    std::cout << std::endl;

    // Free memory on the device
    cudaFree(d_array);
    return 0;

}
```

Listing 3.1: CudaMalloc Code

```
include <iostream>
#include <cuda_runtime.h>
#define N 999

// CUDA kernel to initialize an array

__global__ void initializeArray(int* d_array)
{
    int tid = blockIdx.x * blockDim.x
              + threadIdx.x;
    if (tid < N)
        d_array[tid] = tid;
}

int main()
{
    int *h_array; // Host array
    int *d_array; // Device array

    /* Allocate unified memory using
    * cudaMallocManaged so that
    * it is accessible on CPU and GPU both */
    cudaMallocManaged(&d_array,
                      N * sizeof(int));

    /* Launch the kernel to
    * initialize the device array */
    initializeArray<<<1, N>>>(d_array);

    /* Wait for GPU to finish
    * before accessing on host */
    cudaDeviceSynchronize();

    /* Since cudaMallocManage allocated
     * pointer that is accessible
     * on GPU as well as CPU
     * we only need a shallow copy.*/
    h_array = d_array;

    // Print the result of h_array
    std::cout << "Array elements: ";
    for (int i = 0; i < N; i++)
        std::cout << h_array[i] << " ";
    std::cout << std::endl;

    // Free memory on the device
    cudaFree(d_array);
    return 0;
}
```

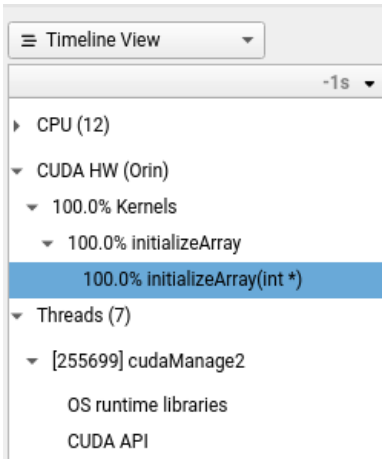
Listing 3.2: CudaMallocManaged Code

The code shown in 3.2 evaluates the functionality of *cudaMallocManaged()*, wherein an array is allocated in the device memory, i.e., GPU. As managed memory is utilized, this pointer remains accessible on both the host and the device. Notably, no deep copy is needed to access the array allocated in the device memory. The nsys profile shown in 3.3a confirms this assertion, as it reveals that there is no additional copy from the device to the host.

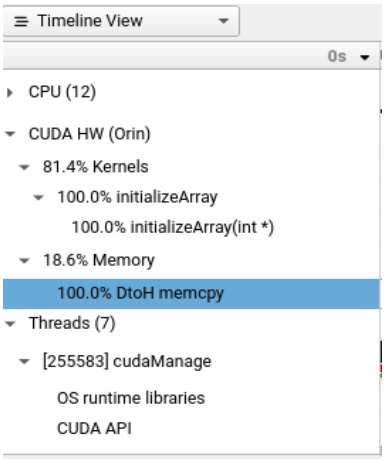
The code shown in 3.1 examines the functionality of the *cudaMalloc()* function, which allocates memory on the GPU device. In this case, direct access to the memory allocated on the device from the host is not possible. As a result, the *cudaMemcpy()* API is required to perform a deep copy from the device memory to the host memory. The nsys profile shown in 3.3b provides evidence supporting this claim, as it shows a copy operation from the device to the host. If we attempt to avoid the use of *cudaMemcpy()* for transferring data from the Device to the Host, after allocating memory with *cudaMalloc()*, any attempt to access the host memory will result in a segmentation error.

Figure 3.3: Unified Memory Cuda Profiling

(a) cudaMallocManaged Profile



(b) CudaMalloc Profile



Operation	Bandwidth
Sequential Write	1.9 GB/s
Random Write	0.15 GB/s
Sequential Read	3.88 GB/s
Random Read	0.53 GB/s

Table 3.1: FIO testing Results on Orin Platform

3.3 Testing PCIe channel bandwidth on Orin

Considering that this paper focuses on deploying large MoE (Mixture of Experts) models on memory-limited GPUs, a crucial aspect of this process involves offloading the Model weights/parameters to the Disk. During inference, the necessary model parameters/weights should be available in the GPU to perform the forward pass of the neural network. Hence, the efficient transfer of data between the GPU and Disk significantly impacts the model's inference efficiency.
To achieve this, We are conducting experiments on the Nvidia Orin GPU, which is connected to an SDD (Solid State Drive) using a PCIe Gen4 channel. The aim is to determine both the maximum and minimum bandwidth utilization of this PCIe channel on the Orin GPU platform. The collected data will then be compared with the bandwidth achieved by baseline methods. The primary goal of this comparison is to assess whether the PCIe channel's bandwidth is fully utilized or not, which will ultimately impact the overall efficiency of the MoE model inference on the memory-constrained GPU.
We are using the Flexible IO tool for benchmarking the persistent disk performance on the Orin platform. Results are shown in table 3.1

Chapter 4

Profiling, Implementation, and Benchmark

In this section of the thesis, we explore the fundamental components of profiling, implementation, and benchmarking, which constitute the central elements of our research inquiry. In 4.1, we engaged in an extensive profiling exercise of the Accelerate library on NLLB-MoE-54b (checkpoint-size: 206 GB) using the accelerate interface. Subsequently, in 4.1.1, a comprehensive analysis of the accelerate profile revealed critical bottlenecks, underscoring the dominant role of transfer time for parameters/weights between the Disk and CPU. Additionally, the transfer time between the CPU and GPU emerged as another significant factor. Section 4.2 elucidates the necessity and intricacies of deploying Archer on the Orin GPU platform. In section 4.3, we outline the efforts dedicated to integrating unified memory architecture support into Archer, resulting in the development of Archer-Edge. The subsequent section, 4.3.1, offers an exhaustive analysis of bottlenecks within Archer and Archer-Edge, introducing a unified caching allocator that enhances inference performance. Finally, section 4.4 details the methodology for benchmarking and the pertinent metrics employed in our evaluation.

4.1 Profiling the “Accelerate” library on MoE models

We are conducting profiling on the ”Accelerate” library applied to MoE (Mixture of Experts) model to thoroughly assess its performance characteristics and identify potential bottlenecks. By profiling, we aim to understand how the library functions when handling these models, gain insights into its efficiency and pinpoint any areas where optimization might be necessary. Our expectation is that this profiling will provide valuable information about the execution behavior of the library on MoE models, helping us identify any inefficiencies and areas for improvement.
We are using the Accelerate interface provided by Hugging Face for running inference with models ”facebook/nllb-moe-54b” and ”google/switch-base-256”. Both

models' checkpoints (pre-trained weights) are available for download from the Hugging Face website [11], [12]. The pre-trained "facebook/nllb-moe-54b" has a checkpoint size of 206 GB and "google/switch-base-256" has a checkpoint size of 55 GB. Accelerate library interface for running inference with nllb-moe-54b is shown in 4.1.

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from transformers import AutoConfig
from accelerate import init_empty_weights
import os
import torch
import time

device = ('cuda' if torch.cuda.is_available() else 'cpu')
checkpoint = "facebook/nllb-moe-54b"
config = AutoConfig.from_pretrained(checkpoint)
with init_empty_weights():
  model = AutoModelForSeq2SeqLM.from_config(config)
model.tie_weights()

from accelerate import load_checkpoint_and_dispatch
from accelerate import infer_auto_device_map
my_device_map = infer_auto_device_map(model,no_split_module_classes=
                    ["NllbMoeEncoderLayer", "NllbMoeDecoderLayer"])

start2 = time.time()
model = AutoModelForSeq2SeqLM.from_pretrained(checkpoint,
                            device_map=my_device_map,
                            offload_folder="/mnt/shivaz/offload/",
                            offload_state_dict = True)
model.eval()

from transformers import AutoTokenizer
print(f"Stage 3: Start intiliazing tokenizer")
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

input_text = "UN Chief says there is no military solution in Syria"
input_ids = tokenizer(input_text, return_tensors="pt").input_ids
print(f"Input Token ID: {input_ids}")
print(f"Input Tokens Length: {input_ids.shape[1]}")

st = time.time()
with torch.no_grad():
    input_ids = input_ids.to(device)
    translated_tokens = model.generate(input_ids,
                          forced_bos_token_id=
                          tokenizer.lang_code_to_id["fra_Latn"],
                          max_length=100)
print(f"Generation time: {time.time() - st}")

print(f"Output Tokens ID: {translated_tokens}")
print(f"Output Tokens Length : {translated_tokens.shape[1]}")

print(f"Stage 6: Start generating output")
print(f"Required translation : {tokenizer.decode(translated_tokens[0],
                                  skip_special_tokens=True)}")
```

Listing 4.1: Accelerate Interface for nllb-moe-54b

PyTorch incorporates specific functions known as hooks, which are automatically activated after a specific event. These hooks are registered for every Tensor or nn.Module objects that are triggered during the forward or backward pass of the respective object.
Accelerate takes advantage of PyTorch's hooks feature to facilitate the execution of Model inference. To achieve this, Accelerate inserts hooks into the model. These hooks play a vital role behind the scenes, enabling the library to perform

certain actions during the forward or backward pass of the model. The precise functions and actions performed by these hooks are tailored to enhance the efficiency and performance of the Model inference process, thereby optimizing the overall execution. How Accelerate runs inference of big models that do not fit on the GPU is explained below:

- The inputs are placed on the appropriate device at each layer, so even if your model spans multiple GPUs, it still functions.
- Just before the forward pass, the weights are offloaded on the CPU and put on the GPU, and cleaned up after the forward pass.
- The weights that were previously offloaded onto the hard drive are loaded in RAM, transferred to a GPU right before the forward pass, and then cleared out immediately after.

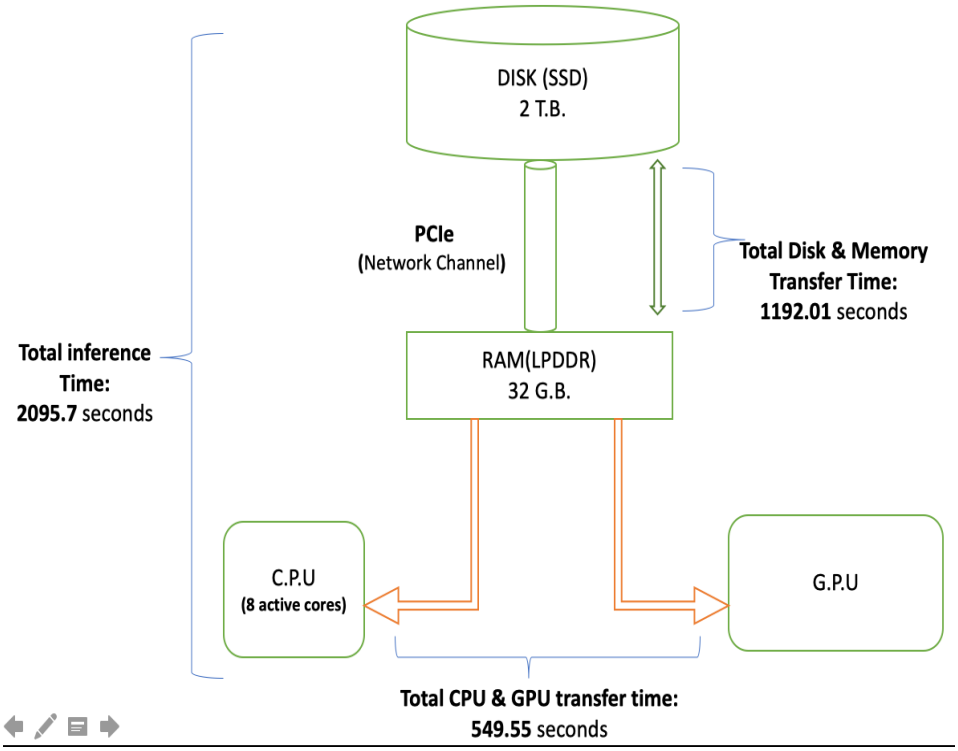


Figure 4.1: Accelerate Profile Generated for nllb-moe-54b Model

4.1.1 Identification of major bottlenecks in baseline

The Accelerate Library, designed to facilitate inference with large models, experiences considerable time overhead even for processing just 38 tokens. Attempting to employ Accelerate with more than 100 tokens becomes practically unviable due

to the extensive runtime required for completion. To profile the baseline performance, Nvidia Nsight tool couldn't be utilized, so a manual annotation approach was adopted to comprehend the sections of the Accelerate library code responsible for the most significant runtime. As previously observed, executing large models necessitates transferring weights from DISK to CPU and then from CPU to GPU. Subsequently, the released weights from the GPU need to be copied back to the CPU.

Referring to figure 4.1, the complete inference process requires approximately 2095 seconds, with a significant portion (around 55% or 1192 seconds) dedicated to data transfer from disk to memory. Additionally, about 550 seconds are spent on the transfer between the GPU and CPU. A crucial observation is that Accelerate lacks support for Unified Memory Architecture. Consequently, when moving a weight or parameter from disk to the GPU, it follows a two-step process: first bringing the weight to the CPU and then transferring it to the GPU.

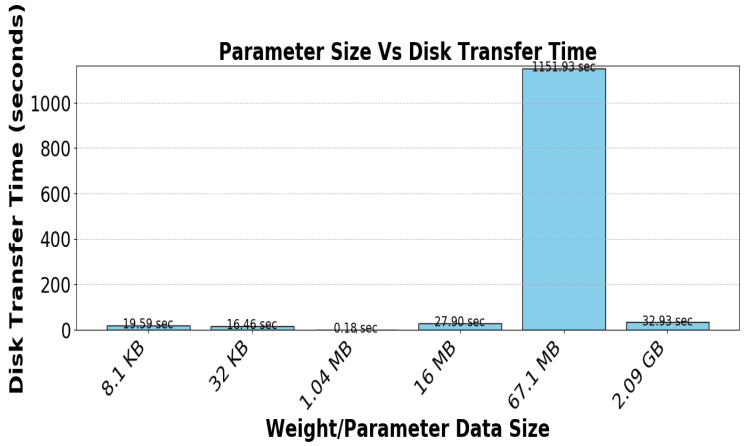


Figure 4.2: Disk Transfer Time for Parameter

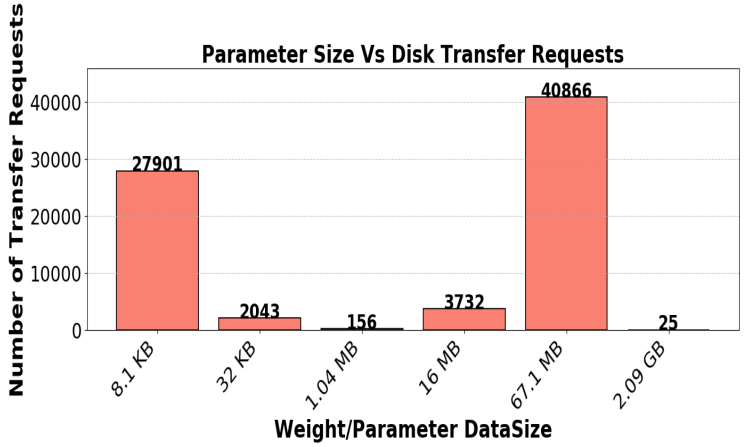


Figure 4.3: Number of Parameter Requests from Disk

The analysis based on the provided bar graph 4.2 reveals important insights regarding the transfer of tensor weights/parameters from the SSD(Disk) to the CPU. The x-axis represents the six unique sizes of tensor weights, while the y-axis indicates the frequency of transfers from the disk to the CPU. Notably, tensors with a size of 67.1 MB are the most frequently requested, followed by 8.1 KB tensors as the second highest in demand. On the other hand, 2.09 GB tensors are the least requested.

Moving to graph 4.3, it showcases a correlation between the requested tensors and the disk transfer time. The most requested tensors take the longest disk transfer time, while 8.1 KB tensors contribute insignificantly to the transfer time. The cumulative transfer size for all parameter sizes is 1893.3 GB, relative to the model checkpoint size of 206 GB.

Another bottleneck lies in the weight transfer between CPU and GPU. Addressing this issue through support for a unified memory architecture in Accelerate could potentially alleviate the runtime bottleneck, resulting in improved performance and efficiency. Analysis results for profiling Accelerate on nllb-moe-54b are summarized in table 4.1.

Table 4.1: Latency and Throughput Statistics

Metric	**Value (s=seconds)**
Maximum bandwidth achieved	3.2 GB/s
Minimum bandwidth achieved	0.8 MB/s
Average bandwidth achieved	1.1 GB/s
FIO Tool Reported Bandwidth	3.88 GB/s
Accelerate Achieved Bandwidth	1.1 GB/s
Input Throughput	0.131 tokens/s
Output Throughput	0.108 tokens/s

4.2 Deploying Archer on Edge GPU

As previously elucidated, the exploration of employing a substantial MoE (Mixture of Experts) model for inference on an embedded GPU remains uncharted territory. Deploying an extensive model on such a platform posed a considerable challenge in itself. The primary obstacle encountered during the course of this dissertation project revolved around encountering OOM (Out of Memory) errors and prolonged inference runtimes. The ongoing active development within Archer introduced

certain complications as well. However, after meticulous adjustments to the code, we succeeded in executing inference using the facebook/nllb-moe-54b (206 GB) model with Archer. To our astonishment, Archer exhibited noteworthy speed in processing an equivalent number of tokens. The data presented in table 4.2 was generated through initial analysis during Archer's initial deployment on the Edge GPU platform. Subsequent enhancements were applied to Archer to further optimize inference, and these outcomes are shared in the evaluation section 5.

Model	Checkpoint Size	Tokens	Accelerate	Archer	Speed-up
facebook/nllb-moe-54b	206 GB	38	2095.7 sec	467.6 sec	4.48x

Table 4.2: Model Performance Comparison

Having previously analyzed the Accelerate library in section 4.1.1, we identified a bottleneck centered around the time required for transferring weight(s) from Disk to CPU. **The suboptimal performance of Accelerate can be attributed to its lack of consideration for the inherent sparsity of the MoE (Mixture of Experts) model**. This shortcoming becomes apparent in its handling of data transfer operations. Unlike Archer, which employs a selective approach by transferring only activated Experts, Accelerate transfers all Experts for a given layer from the disk to the GPU. Accelerate incurs a substantial overhead, transferring an extensive volume of tensor weights totaling around 1893.3 GB from the Disk to the CPU. In contrast, Archer demonstrates a more efficient strategy, transferring a mere 187 GB of weights. The impact of this inefficiency is particularly evident when examining specific tensor weights. Notably, tensors with a weight size of 67.1 MB are transferred a staggering 40866 times from the disk to the GPU. Additionally, tensors weighing 8.1 KB are moved 27901 times.

Another factor contributing to the variance in inference times between Accelerate and Archer lies in the underlying mechanism employed for moving weights from the disk to the CPU.
The Accelerate library utilizes the numpy.memmap() utility for facilitating weight transfers from disk to CPU. This utility creates a memory-mapped connection to an array stored in a binary file on disk. Memory-mapped files serve the purpose of accessing discrete sections of large files on disk without necessitating the complete loading of the entire file into memory. This definition is sourced from the numpy documentation [8]. It's important to acknowledge that memory-mapped performance might fluctuate based on factors such as the storage medium, the access pattern, and the specifics of the hardware.

In contrast, Archer employs the DeepSpeed IO transfer architecture to facilitate the transfer of weights from disk to the CPU, or in the case of unified memory, to the GPU. The DeepSpeed IO module has been meticulously designed to enhance data loading and reading within deep learning applications. It boasts features such as multi-threaded data loading, asynchronous data loading, and optimizations tailored

for a variety of data formats.

4.3 Archer-Edge Implementation

This section holds a pivotal role within the context of this paper. Here, we will delve into the enhancements made to Archer, leading to the development of Archer-Edge. The core objective of this paper was to delve into the realm of unified memory architecture and to successfully deploy a substantial MoE model. To achieve this objective and ensure Archer's compatibility with the unified memory architecture, it was imperative to grasp the intricacies of the existing implementation and to thoroughly comprehend the operational flow of Archer. The majority of efforts dedicated to this enhancement were in C++.

Up until this point, Archer had been characterized by two distinct memory pools: the DeviceMemoryPool and the HostMemoryPool. These interfaces exercised control over memory allocation on the host (CPU) and the device (GPU) respectively. On the host side, memory allocation employed the *cudaHostAlloc()* API, which facilitated memory allocation on the CPU while also locking pages. On the device side, memory allocation was achieved using the *cudaAlloc()* API. Notably, the pointers allocated via these APIs were confined to their respective memory pools, creating a situation where if data existed on the device and needed to be accessed on the host, it became the user's responsibility to perform explicit data copying. This additional step introduced a performance overhead. However, Archer-Edge introduced an addition in the form of the UnifiedMemoryPool. A pointer allocated within this pool boasts accessibility from both the host and the device, and this is achieved using the *cudaMallocManaged()* API.

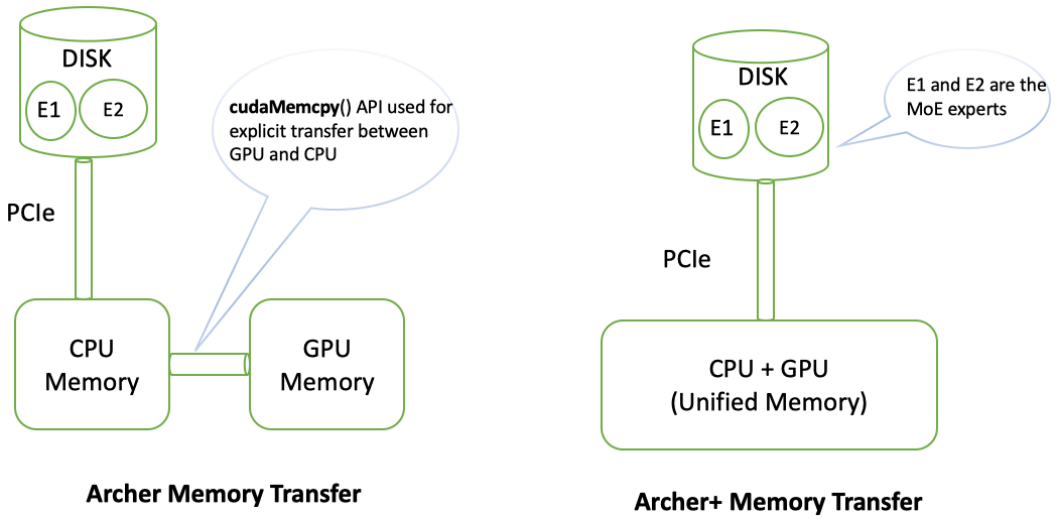


Figure 4.4: Memory Transfers in Archer & Archer-Edge

Central to the Archer operational flow are parameter offloading and prefetching, which play a critical role. Conventionally, parameters were transferred from disk to CPU and subsequently from CPU to GPU. However, Archer-Edge revolutionized this process by directly transferring data from disk to GPU, bypassing CPU intervention entirely, as depicted in the visual representation 4.4. This enhancement was supported by the introduction of novel APIs that are robust. Additionally, a high-level flag "UNIFIED_MEMORY" was incorporated, providing the ability to activate the unified memory operational flow by setting it to true within the environment. An essential aspect of this enhancement was the thorough understanding of the core CUDA APIs that played a pivotal role in its realization.
All the code implemented for Archer-Edge is present on a 'unified_mem_dev' branch in a private Github repository. Currently, the work is not open-sourced, but for access please contact Dr. Luo's Team. The high-level algorithm of Archer-Edge is shown 4.2

```
 1: Initialize Archer-Edge Modules

 2: Load model checkpoint weight
    2.1: Read tensors/weights from checkpoint state dict
    2.2: Register tensors by allocating a pinned-memory
    2.3: Construct an archer_index in disk
         /*archer_index maps tensor_id Vs tensor/weight pointer */
    2.4: Delete memory allocated to tensor/weight
    /*Above mechanism helps in loading a 200 GB in a 32 GB memory*/

3: Construct a "Node" tracing graph
    2.1: For each layer of the model
    2.2: Create a node
    2.3: Bind tensor_ids(parameters) of the layer to that node
    /*Example: Node 0 tensor ids 512
    Node 1 tensor ids 514 515 516 517 518 519 520 521 522 523 528
    Node 2 tensor ids 524 525
    */

4: In pre-forward call
    4.1: Acquire tensor_id from disk
    4.2: Bring the Node that contains the tensor_id
    4.3: Since Node contains multiple tensor_ids, we need to
         transfer all tensor_ids from Disk to GPU directly
        /*
           Here we utilize "UNIFIED_MEMORY" architecture.
           Otherwise, the only way to transfer parameters
           is to first copy from DISK to CPU and then copy
           from CPU to GPU.
        */

    if pre-fetching is enabled:
        4.4: On a parallel thread, bring the nodes based on
             Experts activated probability for the next layer.
    /*
       Pre-fetching helps to parallelize data transfer
       of tensors/parameters from disk to GPU and
       computation of forward-pass.
    */

5: In post-forward call
    5.1 Release tensors that are not required anymore
```

Listing 4.2: Archer-Edge System Implementation

4.3.1 Profiling Archer & Archer-Edge

We implemented Archer-Edge and now it was time to test our implementation. We ran nllb-moe-54b (206 GB) with Archer and Archer-Edge, we found that Archer-Edge was 25% faster than Archer shown in table 4.3. This comparison is conducted considering the initial implementation of Archer-Edge. For the final comparison results, please refer to section 5.

Model	Checkpoint Size	Tokens	Archer	Archer-Edge	Speed-up
facebook/nllb-moe-54b	206 GB	42	467.6 sec	351.3 sec	+25%

Table 4.3: Model Performance Comparison

We saw improvement in Archer-Edge compared to Archer, now it was time to analyze the bottlenecks in the current implementation and improve on them. We used Nvidia's Nsight tool [6] for generating profiles for inference with Archer and Archer-Edge on nllb-moe-54b with only 22 tokens. The nsys profiles for the runs conducted using Archer and Archer-Edge can be found in the appendix under sections A.1 and A.2 respectively. From these profiles, we have extracted crucial data, which has been condensed into summary tables as shown in tables 4.4, 4.5, 4.6, and 4.7. Table 4.4 provides insights into the percentage of time consumed by memory operations in the case of Archer, while table 4.5 presents the most time-consuming Cuda APIs utilized by Archer. Similarly, tables 4.6 and 4.7 outline corresponding statistics for Archer-Edge. The observed differences in memory utilization within the profiles were substantial, enabling us to gain a comprehensive understanding of the existing functionality and bottlenecks.

Profile run with Archer and Archer-Edge reveals that parameter/weights transfer between host and device is the major bottleneck and needs to be addressed to improve the latency and throughput. Findings from the profiles show that a major amount of time is taken by operations *cudaMemcpy*, *cudaMemcpyAsync*, *cudaFreeHost* and *cudaHostAlloc* for both Archer and Archer-Edge.

- cudaMemcpy: copies data between host and device.
- cudaMemcpyAsync: Copies data between host and device, but it is asynchronous with respect to host so the function call may return before the copy is complete.
- cudaFreeHost: Frees page-locked memory on the host.
- cudaHostAlloc: Allocated page-locked memory on the host.

Upon careful code examination, we identified instances of redundant invocations of *cudaMemcpy* and *cudaFreeHost* during the parameter transfers between the disk, CPU, and GPU within the unified memory configuration. By optimizing the code

Memory Operation	Contribution
Memset	5.8%
HtoD memcpy	65.8%
DtoH memcpy	28.3%
DtoD memcpy	0.1%

Table 4.4: Archer Profile Memory Operations

Cuda API	Total Time Taken	Contribution
cudMemcpy	25.763 s	20.6%
cudMemcpyAsync	20.641 s	16.5%
cudaFreeHost	18.483 s	14.8%
cudaHostAlloc	16.533 s	13.2%
cudaMallocManaged	11.752 s	9.4%
cudaMemSet	10.072 s	8.1%
cudaFree	9.142 s	7.3%
cudaLaunchKernel	8.621 s	6.6%

Table 4.5: Cuda APIs Statistics for Archer

and eliminating these superfluous memory copies, a notable improvement in inference latency was achieved. Additionally, we introduced a unified memory caching allocator to further bolster performance.

Unified Memory Caching Allocator

This idea for implementing a unified memory caching allocator was inspired by the work of another Ph.D. student in Dr. Luo's team. Certain mobile operating systems, like the Pixel 3, have shown a tendency to promptly release memory back to the system, which can lead to page faults on occasion, ultimately impacting overall performance negatively. This caching allocator has been developed to tackle this issue. Additionally, it offers users the flexibility to define their own memory allocator by creating their own implementations of the allocate and free virtual interfaces. This caching allocator further helped in improving the performance. Below are a few APIs shown in 4.3 related to the implementation of the caching allocator.

Memory Operation	Contribution
Memset	5.8%
HtoD memcpy	65.8%
DtoH memcpy	28.3%
DtoD memcpy	0.1%

Table 4.6: Archer-Edge Profile Memory Operations

Cuda API	Total Time Taken	Contribution
cudMemcpy	95.666 s	45.6%
cudMemcpyAsync	28.543 s	13.6%
cudaFreeHost	25.305 s	12.1%
cudaHostAlloc	19.197 s	9.1%
cudaMemSet	12.969 s	6.2%
cudaMalloc	9.452 s	4.5%
cudaLaunchKernel	9.234 s	4.4%
cudaFree	9.122 s	4.3%

Table 4.7: Cuda APIs Statistics for Archer-Edge

```
inline void* UnifiedCachingAllocator::allocate_and_cache(const size_t bytes)
{
    // NOLINTNEXTLINE(cppcoreguidelines-init-variables)
    void* ptr;
    at::cuda::CUDAStreamGuard guard(CUDA_STREAM_H2D_VIEW(0));
    auto cuda_err = cudaMallocManaged(&ptr, bytes);
    if (cuda_err != cudaSuccess) {
        free_cached();
            cuda_err = cudaMallocManaged(&ptr, bytes);
        if (cuda_err != cudaSuccess) {
            throw std::runtime_error("cudaMallocManaged failed");
        }
    }

    allocation_map_[ptr] = bytes;
    return ptr;
}

void* UnifiedCachingAllocator::allocate(const size_t bytes)
{
    std::lock_guard<std::mutex> guard(mutex_);
    const auto& it = available_map_.find(bytes);
    if (it == available_map_.end() || it->second.empty()) {
        return allocate_and_cache(bytes);
    }
    return it->second.pop_back_val();
}

void UnifiedCachingAllocator::free(void* ptr)
{
    // NB: since we are not really freeing the memory
    // the cases such as quantization code freeing original weights
    // on mobile, will not quite work, as we likely will hold
    // onto that memory.
    // NB: We can also enable max memory cached for better memory
    // management such that free will actually free the memory if
    // we are nearing or above the watermark.
    std::lock_guard<std::mutex> guard(mutex_);
    // If this allocation was done before caching allocator was enabled
    // then free regularly
    const auto& it = allocation_map_.find(ptr);
    if (it == allocation_map_.end()) {
        at::cuda::CUDAStreamGuard guard(CUDA_STREAM_H2D_VIEW(0));
        cudaFree(ptr);
        return;
    }
    const size_t alloc_size = it->second;
    available_map_[alloc_size].push_back(ptr);
}
```

Listing 4.3: CudaMalloc Code

4.4 Benchmarking MoE model using Archer and Archer-Edge

We recognized the performance limitations of Accelerate, as discussed in section 4.1.1, and endeavored to address these constraints within the frameworks of Archer and Archer-Edge. Benchmarking MoE model using both Archer and Archer-Edge is a crucial step to assess the effectiveness of the proposed enhancements. This comparison allows us to quantify the performance gains achieved by Archer-Edge over the original Archer framework. By subjecting both implementations to rigorous benchmarking, we can objectively measure and analyze various metrics, including inference latency, throughput, and other relevant performance indicators. This evaluation serves to validate the improvements introduced in Archer-Edge and provide empirical evidence of its superiority in terms of inference efficiency. The anticipated outcome of this benchmarking is to demonstrate that Archer-Edge indeed delivers a notable reduction in inference latency and potentially other performance enhancements when compared to the baseline Archer model.
To gauge the effectiveness of our solutions, we developed a new utility for benchmarking. This utility quantifies essential metrics such as overall inference latency, input and output latencies, and throughput for input and output tokens. Explanation of important metrics reported in the benchmark report:

- **Inference Latency**: This represents the time required for an inference task to be completed. It's measured in seconds.
- **Input Latency**: Input latency quantifies the time the encoder takes to produce the final hidden states for the input tokens. A single encoder pass occurs in each inference.
- **Output Latency**: Output latency gauges the time taken by all decoder passes to generate output tokens. Multiple passes of the decoder happen in a single inference, generating new tokens in each pass.
- **Throughput (Input)**: Input throughput is measured as the input tokens processed per second. Its value is calculated as (Total number of input tokens/ Input Latency).
- **Throughput (Output)**: Output throughput is measured as the output tokens processed per second. Its value is calculated as (Total number of output tokens/ Output Latency).

As the primary emphasis of this paper does not revolve around the Quality of Results (QoR) for the Machine Translation (MT) task, our primary concern is directed toward evaluating the inference performance. To construct a benchmarking dataset, we extracted samples from the TREC dataset, encompassing diverse token counts spanning from 10 to 200 for both input and output tokens. The inputs were classified into three distinct categories, namely Group 1, Group 2, and Group 3, delineated by

the token count they encompass.

For running the benchmark, we are using Nvidia's Orin GPU platform with 32 GB memory. We are running nllb-moe-54b having a checkpoint size of 206 GB. **We disabled prefetching in Archer & Archer-Edge as it was causing some I/O contention issues explained in section 5.3**. Algorithm 4.4 shows steps involved in the benchmarking script.

```
Initializing Archer
Start logging
Loading nllb-moe-54b model using Archer
For input in group\_ids: // Iterate on inputs in group\_id
    model.generate(input) // Inference API
    Capture all metrics
End logging
```

Listing 4.4: Benchmark Script Pseudo Code

Tables 4.8 and 4.9 present key metrics such as total latency, input and output latencies, and input and output throughput across different token counts, categorized by the group_id, for both Archer and Archer-Edge respectively.

Table 4.8: Archer Model Performance on nllb-moe-54b

Group	Tokens		Latency (seconds)			Throughput (tokens/seconds)	
	Input	Output	Input	Output	Overall	Input	Output
G1	17	20	19.755	95.319	115.075	0.21	0.861
	15	19	12.837	83.727	96.565	0.227	1.168
	14	20	14.663	89.791	104.454	0.223	0.955
	17	26	15.376	116.702	132.078	0.223	1.106
	16	21	15.09	95.998	111.09	0.219	1.06
	20	28	17.163	128.083	145.246	0.219	1.165
	20	23	16.496	104.785	121.282	0.219	1.212
	17	31	15.915	143.14	159.069	0.217	1.068
	19	22	16.325	98.696	115.023	0.223	1.164
G2	69	88	35.806	409.812	445.619	0.215	1.927
	63	92	33.924	430.519	464.443	0.214	1.857
	68	84	34.551	384.716	419.268	0.218	1.968
	71	94	35.324	434.345	469.67	0.216	2.01
	83	119	36.79	552.039	588.842	0.216	2.256
	65	89	29.107	408.241	437.349	0.218	2.233
	71	97	34.574	448.98	483.555	0.216	2.054
	71	93	31.618	427.61	459.229	0.217	2.246
	59	79	30.534	368.009	398.543	0.215	1.932
	52	61	31.023	282.997	314.021	0.216	1.676
G3	158	191	47.707	799.701	847.409	0.239	3.312

Table 4.9: Archer-Edge Model Performance on nllb-moe-54b

Group	Tokens		Latency (seconds)			Throughput (tokens/sec)	
	Input	Output	Input	Output	Overall	Output	Input
G1	17	20	18.228	41.337	59.566	0.484	0.933
	15	19	15.316	34.275	49.592	0.554	0.979
	14	20	14.998	34.555	49.554	0.579	0.933
	17	26	15.064	43.28	58.345	0.601	1.129
	16	21	15.307	34.827	50.134	0.603	1.045
	20	28	15.457	50.359	65.816	0.556	1.294
	20	23	15.393	38.516	53.909	0.597	1.299
	17	31	14.955	52.292	67.248	0.593	1.137
	19	22	15.34	37.595	52.936	0.585	1.239
G2	69	88	15.842	159.509	175.352	0.552	4.356
	63	92	15.312	165.862	181.175	0.555	4.114
	68	84	15.524	154.628	170.152	0.543	4.38
	71	94	15.827	174.877	190.704	0.538	4.486
	83	119	15.576	217.792	233.368	0.546	5.329
	65	89	15.829	168.625	184.455	0.528	4.106
	71	97	15.784	182.495	198.279	0.532	4.498
	71	93	15.514	169.24	184.755	0.55	4.577
	59	79	15.399	144.898	160.297	0.545	3.831
	52	61	15.416	107.469	122.886	0.568	3.373
G3	158	191	15.944	349.005	364.949	0.547	9.91

Chapter 5

Evaluation

The evaluation segment of this thesis explores the proposed methodologies in a comprehensive manner. Section 5.1 focuses on illuminating and contrasting the profiles generated for Accelerate, Archer, and Archer-Edge. In 5.2, we elucidate the performance enhancements observed in Archer-Edge compared to Archer, alongside an examination of the benchmark outcomes.

5.1 Comparing profiles for Accelerate/Archer/Archer-Edge

A critical part of this paper was the evaluation of the performance of different libraries on nllb-moe-54b model (206 GB). As previously observed, the inference performance of Accelerate on a large MoE model is notably suboptimal. When attempting to process 12 input tokens and generate 26 output tokens, Accelerate's performance deteriorated significantly, requiring more than 2000 seconds to complete. Despite my efforts to execute the benchmark script with Accelerate, the process was excessively time-consuming, eventually compelling me to terminate it due to the constraints on available time.
Further analysis reveals that Archer achieves an average bandwidth of **1.8 GB/seconds** approximately on the Orin platform whereas Accelerate achieves around **1.2 GB/seconds** approximately.

Table 5.1 presents an overview of the inference latency for Accelerate, Archer, and Archer-Edge utilizing the most up-to-date code. As we can see in table 5.2 there is a speed-up of 14x in Archer while comparing it to Accelerate, we have already discussed the reasons for slowness in Accelerate in 4.2.

The suboptimal performance of Accelerate can be attributed to its lack of consideration for the inherent sparsity of the MoE (Mixture of Experts) model. This shortcoming becomes apparent in its handling of data transfer operations. Unlike

Model	Checkpoint Size	Tokens	Accelerate	Archer	Archer-Edge
facebook/nllb-moe-54b	206 GB	38	2095.7 sec	147.33 sec	71.76 sec

Table 5.1: Model Performance Comparison on NLLB-MoE-54b

Accelerate to Archer	Archer to Archer-Edge	Accelerate to Archer-Edge
14x	2.2x	29x

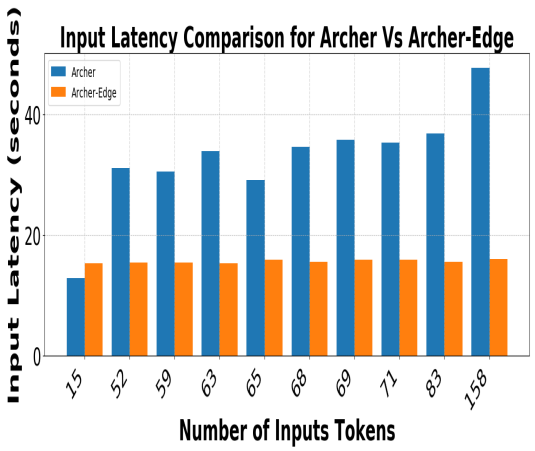
Table 5.2: Inference Speed-up Comparison between Accelerate/Archer/Archer-Edge

Archer, which employs a selective approach by transferring only activated Experts, Accelerate transfers all Experts for a given layer from the disk to the GPU.

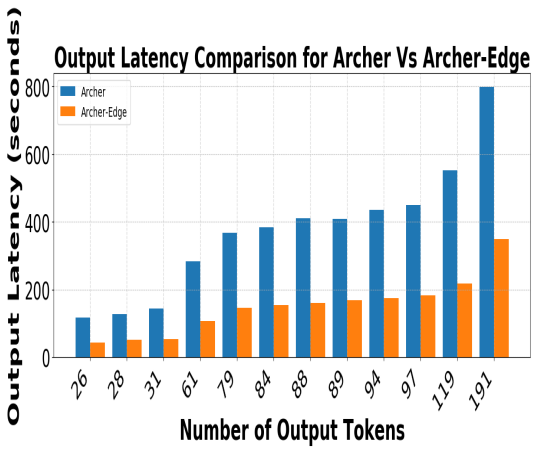
5.2 Improvements in Archer-Edge compared to baseline

Figure 5.1: Input and Output Latency Plots for Archer vs Archer-Edge

(a) Input Latency Plot Archer vs Archer-Edge



(b) Output Latency Plot Archer vs Archer-Edge



The information gleaned from Plot 5.1a and 5.1b shows the input and output latency comparison for Archer vs Archer-Edge. The x-axis of the graph represents the number of input/output tokens, and the y-axis represents the input/output latency in seconds. It indicates that Archer-Edge exhibits reduced input/output latency compared to Archer as the count of input tokens rises. This trend remains consistent for output latency as well.

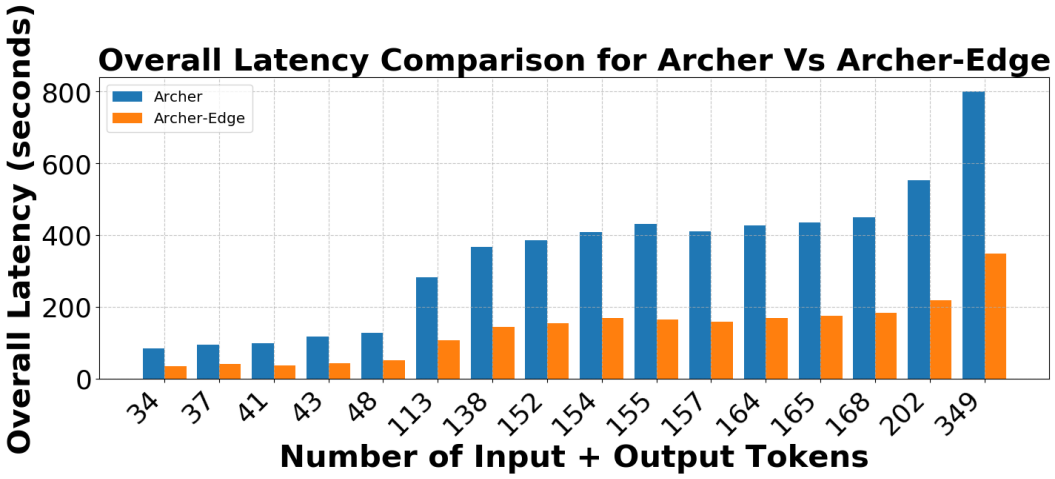
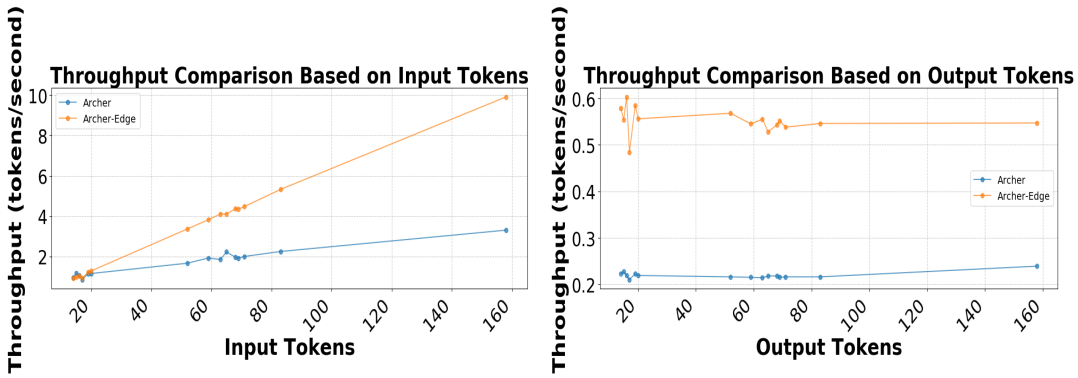


Figure 5.2: Overall Inference Latency Archer Vs Archer-Edge

Figure 5.3: Throughput Plots for Archer vs Archer-Edge

(a) Input Throughput Plot Archer vs Archer-Edge (b) Output Throughput Plot Archer vs Archer-Edge



The image 5.2 shows a graph of the overall latency comparison between Archer and Archer-Edge. The graph is a line graph, with the x-axis representing the number of input and output tokens, and the y-axis representing the overall latency in seconds. The blue line represents the latency for the Archer, and the orange line represents the latency for the Archer-Edge. As the number of input and output tokens increases, the latency for both models increases. However, the Archer-Edge has consistently lower latency than the Archer.

The images 5.3a & 5.3b show graphs of the throughput comparison based on input/output tokens for Archer and Archer-Edge. The x-axis of the graph represents the number of input/output tokens, and the y-axis represents the throughput in tokens per second. The blue line represents the throughput for Archer, and the orange line represents the throughput for Archer-Edge. As the number of input/output tokens

increases, the throughput for both models increases. However, **the Archer-Edge has a consistently higher throughput than the Archer.**
In the context of input tokens, the throughput demonstrates a linear increase with the expansion of token count for both Archer and Archer-Edge. Conversely, concerning output token generation, the throughput remains constant for both Archer and Archer-Edge despite an escalation in token quantity.

5.3 Limitations in Archer-Edge

It is crucial to emphasize that Archer-Edge exhibits superior performance compared to Archer. However, it's important to note that certain issues, which remain unresolved due to the time constraints imposed by this thesis project, are imperative to address for the sake of Quality of Results (QoR).

- Archer-Edge faces a PyTorch issue wherein the tensor for certain weights fetched from the disk is not being appropriately configured.
- In both Archer and Archer-Edge, the process of prefetching has resulted in I/O contention. While Archer's prefetching strategy aims to proactively retrieve weights from disk to the GPU for optimal GPU usage, this approach is causing an Out of Memory (OOM) error on the Edge platform.

Chapter 6

Conclusions

This thesis paper was a novel attempt to deploy a large MoE model on Edge GPU. The following points are the main contributions of this paper:

- Exploration of unified memory architecture. Understanding the major differences between a normal server GPU compared to an Edge GPU.
- Archer surpasses Accelerate significantly in terms of inference latency, achieving a substantial 14x speed-up in overall inference speed.
- Developed Archer-Edge that is compatible with the unified memory architecture of Edge GPU.
- Showed a 2.2x speed-up in inference latency in Archer-Edge when compared to Archer. Archer-Edge beats Archer in throughput metric as well.
- Issues that need to be addressed in Archer-Edge.

Addressing the persistent PyTorch problem in Archer-Edge is imperative for enhancing the Quality of Results (QoR) of the machine translation (MT) task. There exists significant potential for achieving improved outcomes by resolving the IO contention issue associated with prefetching on the Edge GPU. In conclusion, this investigation has been a captivating endeavor carried out within the confines of restricted time. The ongoing surge in the prominence of extensive language models and their broad acceptance underscores the need for enhancements in inference speed. This study aims to further enhance research effectiveness in harnessing large-scale deep learning models on mobile Edge devices.

Bibliography

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[3] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models, 2023.

[4] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, PP:1–20, 07 2019.

[5] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. Towards understanding mixture of experts in deep learning, 2022.

[6] NVIDIA CORPORATION. Nvidia nsight systems user guide.

[7] Nvidia Corporation. Unified memory in cuda 6, 2017.

[8] NumPy Official Documentation. Numpy documentation.

[9] Mengnan Du, Subhabrata Mukherjee, Yu Cheng, Milad Shokouhi, Xia Hu, and Ahmed Hassan Awadallah. What do compressed large language models forget? robustness challenges in model compression. *CoRR*, abs/2110.08419, 2021.

[10] Christof Ebert and Panos Louridas. Generative ai for software practitioners. *IEEE Software*, 40(4):30–38, 2023.

[11] Hugging Face. nllb-moe-54b checkpoint.

[12] Hugging Face. switch-base-256 checkpoint.

[13] Hugging Face. Handling big models for inference, 2017.

[14] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. 1 2021.

[15] Isobel Claire Gormley and Thomas Brendan Murphy. A mixture of experts model for rank data with applications in election studies. *Annals of Applied Statistics*, 2:1452–1477, 12 2008.

[16] Magdalena Graczyk, Tadeusz Lasota, Zbigniew Telec, and Bogdan Trawiński. Application of mixture of experts to construct real estate appraisal models, 2010.

[17] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. Characterizing the deployment of deep neural networks on commercial edge devices. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–48, 2019.

[18] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. Characterizing the deployment of deep neural networks on commercial edge devices. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–48, 2019.

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[20] Jianwei Hao, Piyush Subedi, Lakshmish Ramaswamy, and In Kee Kim. Reaching for the sky: Maximizing deep learning inference throughput on edge devices with ai multi-tenancy. *ACM Trans. Internet Technol.*, 23(1), feb 2023.

[21] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. 7 2023.

[22] Leela S Karumbunathan. Nvidia jetson agx orin series a giant leap forward for robotics and edge ai applications technical brief version date description of change, 2022.

[23] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E. Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *CoRR*, abs/2002.11794, 2020.

[24] OpenAI. Gpt-4 technical report, 2023.

[25] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *CoRR*, abs/1802.05668, 2018.

[26] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[27] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, jan 2017.

[28] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. 1 2017.

[29] Chelliah PR Prakash E Hewage C Surianarayanan C, Lawrence JJ. A survey on optimization techniques for edge artificial intelligence (ai). 2023.

[30] NLLB Team, Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, and Jeff Wang. No language left behind: Scaling human-centered machine translation, 2022.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 6 2017.

[32] Chenguang Wang, Mu Li, and Alexander J. Smola. Language models with transformers, 2019.

[33] Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts, 2021.

Appendix A

First appendix

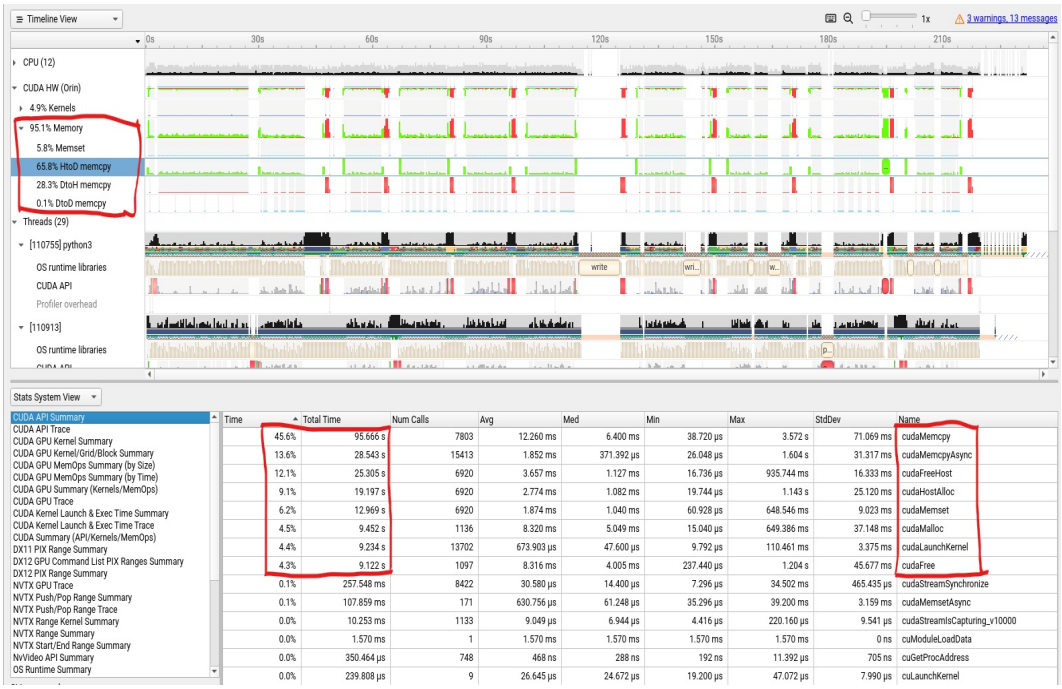


Figure A.1: Profile with Archer

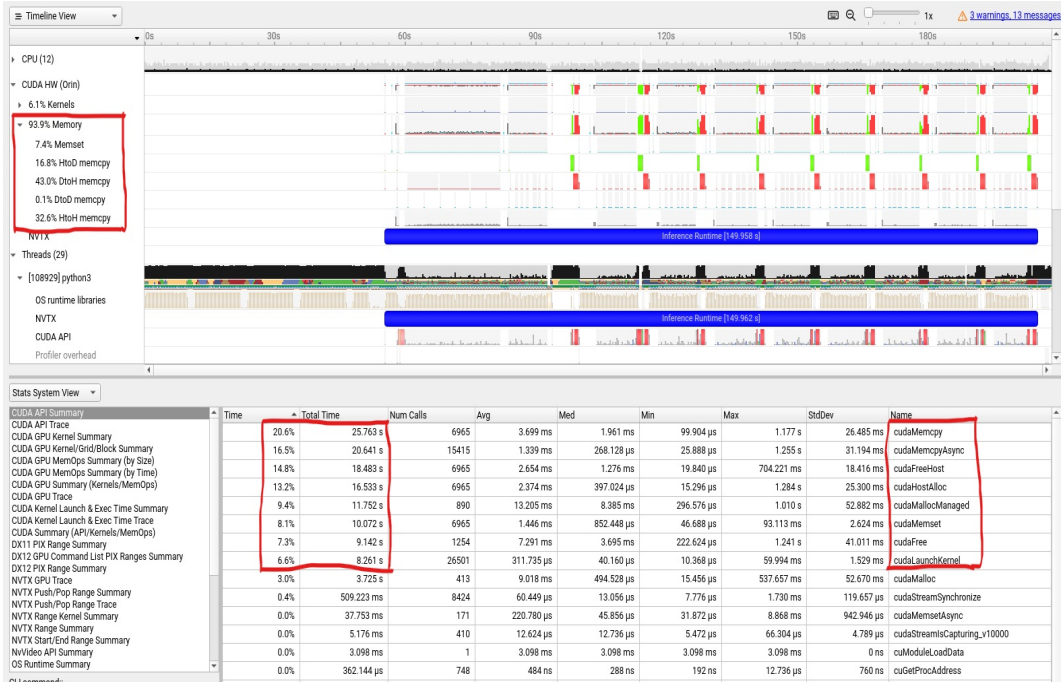


Figure A.2: Profile with Archer-Edge