PowerInfer-2: Fast Large Language Model Inference on a Smartphone

Zhenliang Xue*, Yixin Song*, Zeyu Mi✉, Le Chen, Yubin Xia, and Haibo Chen

Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University

Abstract

This paper introduces *PowerInfer-2*, a framework designed for high-speed inference of Large Language Models (LLMs) on smartphones, particularly effective for models whose sizes exceed the device's memory capacity. The key insight of PowerInfer-2 is to utilize the heterogeneous computation, memory, and I/O resources in smartphones by decomposing traditional matrix computations into fine-grained neuron cluster computations. Specifically, PowerInfer-2 features a polymorphic neuron engine that adapts computational strategies for various stages of LLM inference. Additionally, it introduces segmented neuron caching and fine-grained neuron-cluster-level pipelining, which effectively minimize and conceal the overhead caused by I/O operations. The implementation and evaluation of PowerInfer-2 demonstrate its capability to support a wide array of LLM models on two smartphones, achieving up to a 29.2× speed increase compared with state-of-the-art frameworks. Notably, PowerInfer-2 is the first system to serve the TurboSparse-Mixtral-47B model with a generation rate of 11.68 tokens per second on a smartphone. For models that fit entirely within the memory, PowerInfer-2 can achieve approximately a 40% reduction in memory usage while maintaining inference speeds comparable to llama.cpp and MLC-LLM. For more details, including a demonstration video, please visit the project site at www.powerinfer.ai/v2.

1 Introduction

Large Language Models (LLMs), with their exceptional ability to comprehend and produce human-like text, have fundamentally enhanced our daily lives and transformed our work environments. The most advanced LLMs today, such as GPT-4 [26] and Claude-3 [6], are hosted in data centers equipped with state-of-the-art GPUs (e.g., NVIDIA H100 [24]). These GPUs provide extensive high-bandwidth memory and deliver computational capabilities reaching thousands of teraflops. Concurrently, there is an emerging trend towards deploying LLMs on ubiquitous smartphones [33, 38], transforming them into intelligent personal assistants. This shift aims to fully leverage rich personal data while maintaining privacy by avoiding transmission of private data to cloud services.

However, smartphones, despite their widespread use, struggle to meet the complex demands of LLM inference due to their constrained processing power and limited memory size.

To address these issues, researchers have explored two promising approaches for serving LLM inference under resource-constrained conditions. Given the limited memory capacity of smartphones, one strategy deploys scaled-down LLMs. For example, Google's Gemini Nano 3.25B [32], which uses less than 2GB of memory, represents a compromise by reducing intelligent capabilities to fit within memory constraints. This is due to larger models having enhanced intelligence, a phenomenon known as the "scaling law" [17].

Alternatively, some techniques aim to lower the computational and storage demands of LLM weights during inference. PowerInfer [30] achieves an 11-fold increase in inference speed on personal computers (PC) by allocating hot-activated neurons to the GPU and cold neurons to the CPU. Another method, LLM in a Flash [4], mitigates memory limits by using flash-based NVMe storage for large model weights. However, these solutions falter on smartphones, which have less powerful, heterogeneous hardware and storage devices with lower bandwidth and no support for concurrent accesses due to a single command queue. This makes I/O activities a frequent bottleneck in LLM inference on mobile devices.

This paper introduces *PowerInfer-2*, the first framework that performs high-speed inference of LLMs on smartphones, accommodating models with up to 47 billion parameters that surpass the device's memory capacity. PowerInfer-2 is the follow-up work to the PowerInfer project, designed specifically for smartphones. Like its predecessor, PowerInfer-2 harnesses the dynamic sparse activation inherent in LLM inference: each inference iteration requires only a subset of neurons, rather than the entirety of the model weights. This method substantially lowers computational demands during inference as PowerInfer-2 needs to process only a select group of neurons per iteration. The inherent sparsity also enhances locality, enabling PowerInfer-2 to build an efficient in-memory cache that maintains the most frequently used neurons in memory, thus mitigating the I/O overhead associated with reading weights.

Different from PowerInfer, a key challenge of LLM inference for PowerInfer-2 lies in the ability to leverage the highly heterogeneous XPUs present in contemporary smartphones,

---

*Co-first authors.

✉Corresponding author: Zeyu Mi (yzmizeyu@sjtu.edu.cn).

such as asymmetric big.LITTLE CPU cores, GPU, and NPU. Inference procedures without fully utilizing hardware features lead to suboptimal generation speed. Another challenge is the inevitable I/O overhead caused by cache misses. Although PowerInfer-2 utilizes sparse activation to reduce the amount of weights required during inference, it still incurs a substantial amount of I/O read operations to retrieve weights from storage, which can adversely affect inference performance.

To address these challenges, the core insight of PowerInfer-2 involves breaking down the coarse-grained matrix computations typical in LLM inference into fine-grained neuron cluster computations. A neuron cluster consists of multiple neurons, whose number is determined by the characteristics of the XPUs, memory, and I/O to fully harness the capabilities of the specific hardware components. Specifically, to leverage the heterogeneous XPU within smartphones, PowerInfer-2 designs a polymorphic neuron engine that provides distinct computation patterns for the prefill and decoding stages of the LLM inference process. During the prefill stage, which processes all tokens in the user input sequence concurrently, PowerInfer-2 merges all neurons into a big neuron cluster to maximize the advantages of the NPU in handling large matrix computations. Conversely, in the decoding stage, which has a batch size of one and exhibits significant sparsity, PowerInfer-2 uses small neuron clusters to exploit the flexibility of CPU cores for this comparatively lighter computational task.

The neuron cluster granularity further allows PowerInfer-2 to mitigate the impact of I/O overhead on the inference process. PowerInfer-2 introduces a segmented cache that operates in the neuron granularity. This cache is designed with specific caching strategies for different LLM weight types, effectively enhancing the cache hit rate. Furthermore, to reduce computational delays caused by I/O operations, PowerInfer-2 proposes a fine-grained neuron-cluster-level pipelining technique that overlaps I/O operations with neuron cluster computations. This approach significantly minimizes the waiting bubbles associated with I/O latency.

To support a broad range of LLMs and smartphones with different configurations, PowerInfer-2 executes an offline planner before the first inference of a new model on a smartphone. This planner receives user requirements and analyzes the model and hardware, and generates an execution plan. The plan describes the configurations of various components that guide the online inference process, including the usage ratios of different XPUs at various stages, the sizes of different cache regions.

We have implemented PowerInfer-2 by extending PowerInfer [30] with an addition of 12K lines of code (LoCs), and deployed it on two smartphones (OnePlus 12 and Ace 2), both of which are equipped with heterogeneous Qualcomm XPUs, and have 24GB and 16GB DRAM memory, respectively. PowerInfer-2 supports a diverse array of LLMs across different model sizes, including Llama-2 [29, 34] (7B, 13B),

TurboSparse-Mistral [31] (7B), and TurboSparse-Mixtral [31] (47B). Our evaluation demonstrates that PowerInfer-2 realizes an average speedup of 3.94× (up to 4.38×) and 25.4× (up to 29.2×) compared to the current state-of-the-art frameworks: LLM in a Flash [4] and llama.cpp [13]. Notably, PowerInfer-2 is the first system to support the TurboSparse-Mixtral-47B model on mobile platforms, achieving a generation speed of 11.68 tokens/s, which is 21.2× faster than that of llama.cpp. Another significant advantage of PowerInfer-2 is its ability to reduce memory usage during model inference. For instance, with smaller models such as the 7B size, PowerInfer-2's techniques can save nearly 40% of memory usage while achieving the same inference speed as llama.cpp and MLC-LLM [33].
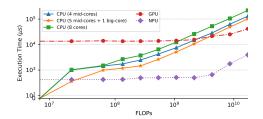
2 Background and Motivation
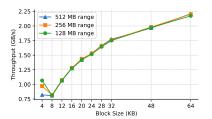
2.1 LLM Inference and Metrics

LLM inference consists of two stages: the prefill and the decoding stage. During the prefill stage, the user's prompt is processed by the LLM in a single iteration, generating the first token. The decoding stage, on the other hand, involves the LLM generating tokens sequentially, one at a time, in an autoregressive manner. The token produced during the prefill stage serves as the input for generating the second token. This second token then acts as the input for the LLM, facilitating the generation of the third word. This sequence continues until the output sequence is complete or an end-of-sequence (EOS) token is reached.

The two stages exhibit distinct computational patterns, necessitating the optimization of two key metrics: the time to first token (TTFT) during the prefill stage and the time between tokens (TBT) during the decoding stage. The prefill stage handles all prompt tokens within a single iteration, imposing a considerable computational burden; in contrast, the decoding stage processes only one token per iteration, resulting in comparatively lower computational demands. Consequently, an LLM inference system must leverage computing strategies designed for these stages specifically to optimize performance metrics efficiently.

2.2 Predictable Sparse Activations

Mainstream LLMs, such as GPT-4 and Llama-2, employ a decoder-only transformer architecture. This architecture consists of multiple transformer layers, with each layer containing an attention block and a Feed-Forward Network (FFN) block. The attention block establishes relationships between tokens in the sequence, while the FFN block interprets and processes these relationships as structured by the attention block. Recent LLMs usually adopt Group Query Attention [27], which reduces the number of weights in the attention block, making the feed-forward network (FFN) block occupy nearly 80% of the total weights. The activation function in the FFN block, such as ReLU-family functions [3, 28, 40], leads to a significant occurrence of sparse activations [19, 39]: most neurons (represented as rows or columns in the FFN weight matrix) are

(a) Execution time of different FLOPs (multiplication of a 4096×4096 matrix with a vector under different batch size) on CPU, GPU and NPU.

(b) 4KB random read throughputs at different block sizes and data ranges.

Table 1: 4KB random read throughputs on 128MB data range with different core setups.

Core Setup	Throughput (MB/s)
1 big-core (3.3GHz)	1,076.10
1 mid-core (3GHz)	1,007.95
1 little-core (2.2GHz)	761.87
1 big-core + 1 mid-core	802.42
1 big-core + 3 mid-cores	613.74

Figure 1: Analysis of XPU computational performance and I/O throughputs.

inactivated because their computations have minimal impact on the final output.

Fortunately, the activation of neurons in the FFN can be predicted before computing each FFN block, that have been explored by prior works [21,29,30,40]. For instance, PowerInfer [30] and DejaVu [21] utilizes small MLP networks before each FFN block to predict their dynamic neuron activations. With these accurate predictors, they can significantly reduce the number of neuron computations within the FFN, thereby accelerating the inference process.

2.3 Smartphone Storage Analysis

A smartphone usually lacks sufficient DRAM memory to hold an entire LLM. Consequently, a portion of the model's weights may be stored in external storage, such as the universal flash storage (UFS) 4.0 in Snapdragon 8gen3. In this section, we analyze the performance characteristics of smartphone UFS, which guide the I/O design of PowerInfer-2.

2.3.1 Read Throughput and Block Size

First, we evaluated the random and sequential read throughputs of UFS 4.0[1]. A notable feature is that the read bandwidth of UFS varies with the read block size. Generally, whether for sequential or random reads, the larger the block, the greater the read bandwidth. For example, when the block size is set to 512KB, both sequential and random read bandwidths reach their maximum at 4 GB/s and 3.5 GB/s, respectively. When the block size is reduced to 4KB, the bandwidth is at its minimum, with random read bandwidth at 450 MB/s.

2.3.2 Random Read and Data Range

UFS random reads exhibit an interesting phenomenon where the performance of random reads is influenced by the scope of the random read range. Specifically, a smaller random read range results in higher bandwidth. In UFS 4.0, as shown in Fig.1b, if the 4KB random read range is set to 128MB, 256MB, and 512MB, the bandwidth for the 128MB range is the highest, reaching 1 GB/s, while the 512MB range has the

lowest bandwidth, falling below 850 MB/s. Notably, this phenomenon is not as apparent with other block sizes. Therefore, the bandwidth of 4KB random reads within a 128MB range exceeds that of 8KB and 12KB block sizes.

2.3.3 Read Throughput and CPU Core

A third observation is that the read bandwidth is influenced by the CPU issuing the read command. A higher frequency of the CPU core correlates with increased read bandwidth. As shown in Table 1, when using a big-core with a frequency of 3.3GHz for random reads, the bandwidth for 4KB reads reaches 1 GB/s. Conversely, when a little-core with a frequency of 2.2GHz is used for the same random reads, the bandwidth is only about 760 MB/s. This correlation arises because the CPU core initiating the read needs to run the UFS driver thus a higher frequency enables faster processing of UFS-related I/O operations, including interrupts and queue management.

2.3.4 Read Throughput and Core Number

The last observation is that unlike NVMe, the UFS storage in mobile devices has only one command queue, inherently lacking internal concurrency capabilities. Therefore, initiating I/O commands using multiple cores does not result in higher I/O bandwidth compared to using a single core. As shown in Table 1, using multiple cores for 4KB random reads even deteriorates the I/O performance by up to 40% as a result of contention in the UFS command queue.

Summary: When some model weights need to be stored on a mobile device's storage medium, an efficient LLM system must fully consider the performance characteristics of the storage medium to maximize I/O bandwidth and minimize the performance overhead associated with I/O operations.

3 PowerInfer-2 Overview

Traditional LLM inference typically depends on matrix computations as the basic unit of inference, a method that introduces significant computational and I/O overhead in the heterogeneous hardware environments of smartphones. Such coarse-grained computations do not effectively leverage the flexible computational capabilities of XPUs. Worse, if a seg-

---

[1]Since LLM inference involves only weight reading, we did not consider the performance of write operations
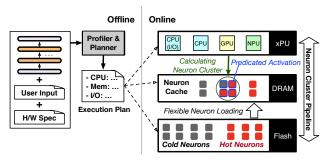
Figure 2: The architecture overview of PowerInfer-2.

ment of the matrix weights is stored on the storage device, there must be a delay for these weights to be loaded into memory before matrix computations can begin, leading to considerable I/O wait times.

This paper introduces PowerInfer-2, a high-speed LLM inference framework specifically designed for smartphones. Its design achieves three goals: 1) Low inference latency: minimizing the inference delay during both the prefill stage (TTFT) and the decoding phase (TBT); 2) Low memory footprint: reducing memory usage during inference, enabling low-latency inference of LLMs even when the model size exceeds the device's memory limit; 3) Flexibility: ensuring the design can be seamlessly adapted to smartphones with varying computational, memory, and storage capacities.

3.1 Neuron Cluster and Architecture

In this paper, we propose a computational abstraction called neuron cluster, which is specifically designed for LLM inference in heterogeneous computing scenarios. PowerInfer-2 performs computation and I/O operations in the granularity of a neuron cluster which can be dynamically composed of multiple activated neurons during computation, with the number of neurons determined by the computational power of the computing unit. For example, during the decoding phase, when computation is performed by the CPU core, the size of neuron clusters assigned to each CPU core is smaller than those handled during NPU computation in the prefill phase. By using this abstraction, PowerInfer-2 can fully utilize XPUs with different computing capabilities. effectively hide the I/O overhead.

Fig.2 illustrates the overall architecture of PowerInfer-2, which is structured into online (the right part) and offline (the left part) procedures. The online part serves the inference at the neuron cluster granularity and includes four collaborative components: the polymorphic neuron engine (§4.1), the in-memory neuron cache (§4.2), flexible neuron loading (§4.3), and neuron-cluster-level I/O pipeline (§4.4).

The polymorphic neuron engine uses completely different computation patterns for the prefill and decoding phases. For the prefill phase, the neuron cluster contains all neurons from the weight matrix and relies primarily on the NPU due to its

efficiency in handling large matrix-matrix multiplications. For the decoding phase, it invokes a predictor to identify which neurons will be activated before initiating computations. The engine then merges these activated neurons into a small neuron cluster and utilizes a CPU core to dynamically calculate the neuron cluster, thereby drastically reducing computational demands and memory usage during runtime.

Before beginning computations for inference, the computing engine retrieves neuron weights from the neuron cache, which is optimized to exploit the locality of neuron-level access observed in LLM inference. In the event of a cache miss, PowerInfer-2 initiates an I/O command to fetch un-cached neuron weights from storage. To mitigate I/O latency, PowerInfer-2 introduces a novel pipeline mechanism that concurrently processes neuron cluster and I/O operations. Additionally, PowerInfer-2 minimizes I/O overhead by adaptively bundling and loading neurons, which is determined by the model's quantization.

To automatically adapt to different models or smartphones, the offline procedure is conducted once for each model initially served on a new smartphone before the online inference begins. This process involves receiving three types of inputs: model weights, user inputs, and hardware specifications. It outputs an execution plan that describes the configurations for each component involved in the online inference and guides the online procedure.

Specifically, an offline planner outputs configurations for computing, memory, and I/O. For computing, the planner determines the proportionate use of CPU and NPU during different phases or layers based on their computational strengths. In terms of memory configuration, to achieve a balance between memory usage and inference performance, the planner enables users to set a desired inference speed prior to running PowerInfer-2. Based on this speed setting, PowerInfer-2 calculates the optimal cache size needed. For I/O configuration, the planner triggers a profiler to measure the sparsity of the model and the distribution of hot and cold neurons.
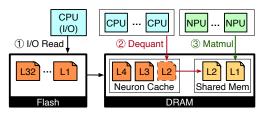
4 Neuron-Aware Runtime Inference
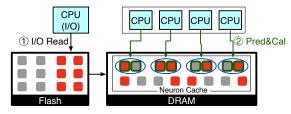
4.1 Polymorphic Neuron Engine

PowerInfer-2 introduces a polymorphic neuron engine that dynamically combines neurons into neuron clusters to take advantage of distinct computational characteristics of LLM inference stages and heterogeneous XPUs.

4.1.1 NPU-Centric Prefill

In the prefill phase, all prompt tokens are processed concurrently. Even though each of these tokens shows high sparsity and activates distinct neurons, there is a considerable decrease in overall sparsity due to the aggregation of these activations. Consequently, PowerInfer-2 does not calculate activated neurons by using predictors in the prefill stage, choosing instead to directly merge all neuron into a big neuron cluster. Given that NPUs excel at handling large matrix-matrix multiplica-

Figure 3: Two computing workflows for prefill and decoding phases. (a) The prefill phase uses an NPU-centric workflow that leverages NPU for computation and CPU for preparation; (b) The decoding phase takes a CPU-centric that only uses CPU cores to exploit the sparse activations.

tions compared to CPU cores, PowerInfer-2 leverages NPUs for the prefill phase.

Although CPU cores do not take part in matrix calculations, PowerInfer-2 utilizes them to perform essential preparatory tasks for the NPU in the prefill phase. First, due to limited memory, PowerInfer-2 relies on a CPU core to load weights stored in Flash into memory during the prefill phase. Second, as current NPUs do not support direct computation with quantized weights, PowerInfer-2 uses CPU cores to dequantize the data before computation by the NPU[2].

Fig.3-a demonstrates how CPUs and NPU collaborate to perform prefill-phase inference in the transformer layer granularity. The NPU computation requires to use a limited amount of memory that it shares with the CPU. Hence, before NPU computation starts, the CPU should preload the needed matrix weights into this shared memory. Within a specific LLM layer, before the NPU conducts any matrix multiplication, Multiple CPU mid-cores read quantized matrix weights from the neuron cache and dequantize these matrix weights into fp16 ahead of time, eventually storing the results in the shared memory between the CPU and NPU. Meanwhile, PowerInfer-2 uses another big-core to asynchronously preload all matrix weights for the next layer into the neuron cache. The mid-core's dequantization, NPU's computation, and the big-core's I/O operations proceed concurrently to reduce the I/O overhead. It is noteworthy that, as the prefill stage involves dense matrix rather than sparse calculations, weight loading via I/O

---

[2]Although Qualcomm's documentation states that the NPU supports direct int4 computations, the SDK does not yet support interfaces for int4 matrix calculations.

can leverage sequential reads to load a large block of data into memory, thereby maximizing the use of UFS's I/O bandwidth.

4.1.2 CPU-Centric Decoding

Unlike the prefill phase, the decoding phase concentrates on a single token during each iteration, demonstrating significant sparsity as only a small fraction of neurons (approximately 10%) in the weight matrix are activated and participate in the computation. Thus, when transitioning from the prefill phase to the decoding phase, the polymorphic neuron engine divides the weight matrix computations into small neuron clusters whose elements are identified as active by a predictor. We observe that when the batch size is one the latency of matrix-vector calculations on CPU cores is lower than that on NPUs. Furthermore, given the reduced number of activated neurons due to sparsity, CPU cores are optimally suited for these lighter and sparse computations among XPUs. Therefore, PowerInfer-2 exclusively utilizes CPU cores for neuron cluster computations during the decoding phase.

Specifically, PowerInfer-2 utilizes CPU cores to compute both the attention and FFN blocks during the decoding phase. Although the attention block does not exhibit sparsity, CPU cores still provide lower computational latency when the input is just a single vector. For the FFN block, PowerInfer-2 initially passes the FFN block's input vector to a predictor, which predicts which neurons in the FFN's weight matrices need to be activated and merges them into a neuron cluster. Each CPU core then takes a cluster and computes these neurons within the cluster and the input vector.

Fig.3-b illustrates the decoding-phase inference conducted by different CPU cores. CPU cores first read the weights of the attention block from the neuron cache and compute them with the input vector. Then then run the predictor to determine the activation status of neurons in subsequent weight matrices. In the third step, CPU cores divide the activated neurons into several clusters, with each core responsible for computing the activated neurons within its cluster with the input vector, and ultimately aggregating the results at a barrier. If these neurons are within the neuron cache, CPU cores will compute them with the input vector. In cases of a cache miss, where neurons are not in the neuron cache, an I/O thread running on a CPU core to asynchronously load the neurons into the cache, ultimately notifying the computation thread to complete the calculation.

4.2 In-Memory Neuron Cache

Efficient cache design can prevent costly storage I/O activities, thereby optimizing end-to-end inference performance. The effectiveness of a caching system depends on the presence of locality in the inference process. However, traditional LLM inference requires traversing all weights for generating each token, showing no locality and rendering any cache design ineffective.

LLM in a Flash [4] proposes leveraging sparse activations

to selectively load weights during inference. It also bundles co-activated neurons and loads them together from Flash to reduce I/O operations. However, this method overlooks the skewed distribution of neuron activations, where a few hot neurons activate more frequently and are highly connected to most other neurons. This brings challenges to designing effective cache strategies. First, these popular neurons reside in different neuron bundles and are redundantly loaded from Flash, wasting I/O bandwidth. Second, our findings show that removing these hot neurons reduces the likelihood of co-activation among the remaining neurons to below 20%, rendering the bundling mechanism ineffective in reducing I/O operations.

To address this, PowerInfer-2 introduces a segmented neuron cache design tailored for various data types within LLMs. It divides the cache into multiple regions, each with specific prefetching and eviction policies. The attention block weights, being smaller and less sparsely activated, are preloaded and retained throughout runtime.

In contrast, the FFN block, prone to frequent activations of hot neurons, uses a dynamic eviction strategy based on Least Recently Used (LRU) for these neurons. This approach ensures that hot neurons are more likely to remain in the cache, while cold neurons are frequently evicted and loaded on demand from Flash. Importantly, the eviction process does not involve writing to storage but simply discards weights from memory.

PowerInfer-2 leverages a classic dual-queue approach to implement its LRU neuron cache, which manage LLM weights at the granularity of individual neurons. The system maintains two doubly linked list queues, labeled *active* and *inactive*, where the order of neurons within the queues is determined by the time of their most recent accesses, with the most recently accessed neurons at the head of the queue.

At runtime, all neurons initially join the inactive queue. Upon re-access, they are promoted to the front of the active queue. Neurons already in the active queue are moved to the head upon subsequent access. To manage cache capacity, when the active queue fills up to 90% of the cache space, neurons from the tail of the active queue are moved to the inactive queue until the active queue's occupancy drops below 90%. If the cache reaches capacity, neurons at the tail of the inactive queue are discarded to make room for new entries.

4.3 Flexible Neuron Loading

Equipped with the neuron cache that effectively stores active neurons, the inference process still inevitably incurs I/O operations for uncached neurons. To optimize I/O read throughput and minimize I/O operations, PowerInfer-2 also bundles associated neurons. Although co-activation within a single FFN weight matrix becomes infrequent once hot neurons are removed, neurons at corresponding positions across different matrices often activate together. For instance, the co-activation probability of the i-th neurons across the

Gate, Up, and Down matrices is as high as 80%. Therefore, PowerInfer-2 opts to store neuron weights based on neuron granularity rather than matrix structure, concatenating weights of the i-th neurons from the Gate, Up, and Down matrices into a single entry.
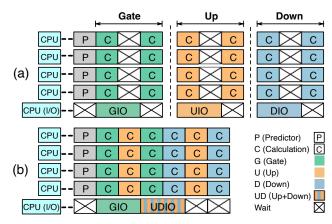
PowerInfer-2 further introduces distinct I/O loading strategies for different models, considering their quantization methods and the inherent characteristics of UFS I/O. For models without quantization, owing to the large storage space each neuron occupies, PowerInfer-2 uses random reads with a larger granularity to boost I/O bandwidth. For example, an individual neuron in Llama-7B-FP16 occupies 8KB, the combined size of neurons from the Gate, Up, and Down matrices amounts to 24KB. PowerInfer-2 efficiently transfers the entire 24KB activated bundle into memory through a single random I/O read.

For 4-bit quantized models, the bundle size is set at 8KB. Considering the Llama-7B model as an example, where each neuron is quantized to 4-bit precision and occupies 2.5KB (2KB for quantized int4 values, and 0.5KB for FP16 scales of quantization groups), the combined bundle size reaches 7.5KB. To align with the storage medium's minimum read granularity of 4KB, PowerInfer-2 supplements the bundle with an additional 0.5KB, rounding the total to 8KB. However, rather than loading these 8KB bundles in a single I/O operation, PowerInfer-2 opts for a 4KB granularity. This choice is based on our analysis in §2.3 showing that the bandwidth from two separate 4KB random reads exceeds that from a single 8KB read, thereby optimizing the I/O reading process.

Moreover, considering the co-activation likelihood of 80% within these bundles, there is still nearly 20% probability that these bundled neurons are not co-activated. Thus, combining the two 4KB random reads could potentially lead to bandwidth wastes. To mitigate this, for models using 4-bit quantization, PowerInfer-2 delays the second 4KB read until the outcomes from the Gate neuron multiplications are obtained. Specifically, PowerInfer-2 uses the predictor to determine the activation of neurons within the Gate matrix, initiating the load for the first part of the bundle based on this information. Afterwards, if the output from the Gate neuron (passing through the activation function) is non-zero, PowerInfer-2 proceeds to load the second part of the bundle, thus minimizing unnecessary I/O operations.

4.4 Neuron-Cluster-Level Pipeline

PowerInfer-2 is also designed to hide I/O overheads by overlapping computation with I/O activities. A straightforward approach is the matrix-level overlapping, which issues I/O commands to retrieve matrix neurons from storage while concurrently processing neurons already in memory. As neurons from storage are loaded, they are immediately processed. Although this matrix-level overlapping method can somewhat hide the cost of I/O operations within the computation process, it still requires the system to wait for the completion of

Figure 4: Two types of pipelines that combine matrix-vector multiplications and I/O operations on five cores (4 calculation cores and 1 I/O cores). Assuming that there are 8 neuron cluster per matrix, 4 of them are in memory and the other 4 are in Flash before the computation starts. (a) The matrix-level pipeline separates the pipeline into isolated matrix units; (b) The neuron-cluster-level pipeline in PowerInfer-2 breaks the matrix barrier and mixes their calculation and I/O operations in the neuron cluster granularity.

Figure 5: The workflow of neuron-cluster pipeline in PowerInfer-2.

all neurons within the matrix, include those fetched from storage, before moving on to the next. As illustrated in Fig.4-a, suppose a matrix contains 8 neuron clusters, with 4 residing in memory and the remaining 4 in storage. A portion of I/O operations can be hidden behind the cached neurons computations. But due to the lengthy I/O times, there would still be instances where CPU cores have to wait for I/O completion.

To eliminate waiting times for I/O operations, PowerInfer-2 introduces a neuron-cluster-level pipeline mechanism. This mechanism is based on an insight: by focusing on the neuron cluster as the granularity, it's possible to overlap the I/O operations within neuron cluster computations from multiple matrices. Concretely, PowerInfer-2 breaks down the barriers between matrix computations; as soon as one neuron cluster finishes computing, it immediately starts the computation of a neuron cluster in the next matrix that are in memory. This mechanism effectively reduces waiting bubbles, as illustrated in Fig.4-b.

PowerInfer-2 divides the execution process of a neuron cluster into 5 sequential stages, which are: determining whether the rows/columns of the Gate, Up, and Down matrices is activated through the predictor (Pred), reading the weights of the rows of the Gate matrix from storage (GIO), calculating the product of the rows of the Gate matrix and the input vector (GC), reading the rows/columns of the Up and Down matrices from storage (UDIO), and calculating the product of the rows/columns of the Up and Down matrices with the input vector respectively (UDC). PowerInfer-2 creates multiple computing threads and one I/O thread to handle the computations and I/O operations for these 5 stages, respectively. The specific number of these threads and the cores on
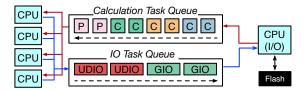
which they will execute are determined by the offline planner.

Fig.5 shows how computing and I/O threads work to implement the neuron-cluster pipeline. At the start of each FFN block, all neurons are initially in the Pred stage, are inserted into the computing queue. Computing threads process these neurons, advancing only those activated to subsequent stages. These activated neurons are then merged into neuron clusters. If the Gate weights of a neuron cluster are available in memory, the neuron cluster progresses to the GC stage and returns to the computing queue. If not, it is set to GIO and moved to the I/O queue. Meanwhile, computing threads continue to process the next neuron cluster from the queue. In parallel, I/O threads takes out neurons from the I/O queue, executing I/O tasks as needed. The execution of UDIO and UDC follows a similar pattern to GC and GIO.

5 Execution Plan Generation

Today's smartphones are equipped with a variety of hardware specifications, such as differing CPU capabilities, I/O throughput, and DRAM sizes. Users deploying LLMs on these devices also have diverse objectives. Some may prioritize a balance between generation speed and memory usage, while others aim to maximize hardware utilization for increased speed. Additionally, the models themselves vary in weight numbers, structures, and sparsity levels. To manage this complexity, PowerInfer-2 includes an offline planner specifically designed to develop execution plans that optimally meet these varied requirements.

5.1 Execution Plan

All symbols labeled as *Var* in Table 2 represent the outputs of the execution plan, describing the configuration of the runtime inference components. $c_{compute}$ denotes the set of CPU cores allocated for computation, while c_{io} specifies the CPU core designated for I/O operations. The variable n represents the total number of selected CPU cores, encompassing both computation and I/O cores. The memory limit m sets the size of the neuron cache during runtime. Finally, the planner outputs an estimated generation speed s for the execution plan.

5.2 Input Parameters

Table 2 also lists three categories of input parameters:

- **Hardware:** Parameters profiled from the hardware, such as CPU FLOPS, I/O throughput, and memory bandwidth.
- **User:** Parameters specified by the user, such as CPU

constraints, memory limit, and lower bound of decoding speed.

- **Model:** Parameters about the model collected by an offline profiler, such as the size of the model, sparsity levels and caching characteristics, etc.

To accurately measure hardware and model characteristics, the planner utilizes an offline profiler to determine their specific parameters. For hardware, the profiler conducts a series of microbenchmarks to evaluate the performance of individual components, including CPU function $F_{\text{cpu}}(i)$, I/O bandwidth $B_{\text{io}}(i, s)$, and memory bandwidth B_{memory}. Regarding model parameters, the profiler initially measure static information such as neuron size and count. It then gathers dynamic data such as neuron hotness, $P_{\text{activated}}$ and $P_{\text{miss}}(m)$ by running the model on a generalized dataset sampled from sources like Wikipedia and RefinedWeb, involving up to 10 million tokens. Additionally, the profiler measures $P_{\text{miss}}(m)$ under varying memory constraints for the specific model.

5.3 Cost Model

After collecting the input parameters, the planner uses a cost model to generate the execution plan. The goal is to maximize the generation speed s (as defined by Equation 1) while adhering to user-specified constraints (Formulas 3-5). The decoding speed s is inversely proportional to the time taken to decode one token (Equation 1), which is determined by the computation times for that token (Equation 2), as we efficiently overlap the computation and I/O operations. As we have defined the objective function and the constraints, the constructed model can be solved by mature SMT solvers. In our implementation, we utilize the Z3 solver [11] to solve the cost model.

$$\text{Maximize} \quad s = 1/T_{\text{decode}} \tag{1}$$

$$T_{\text{decode}} = T_{\text{attn}} + T_{\text{pred}} + \max(T_{\text{ffn}}, T_{\text{io}}) \tag{2}$$

$$n \le n_{\max}, \; m \le m_{\max}, \; s \ge s_{\min} \tag{3}$$

$$c_{\text{io}} \notin c_{\text{compute}} \tag{4}$$

$$c_{\text{compute}} \cup \{c_{\text{io}}\} \subseteq c_{\text{allowed}} \tag{5}$$

To compute the decoding time, we first model the times for computation. As we observed that memory opeartion is not a significant factor compared to the computation, we do not consider it in the computation time. Computation time (Equation 6) is primarily influenced by the attention blocks, predictors, and FFN blocks. The calculation involves dividing the computational workload of these components by the CPU flops (defined in Equation 7- 8). The flops of the selected CPU cores are specified in Equations 9.

$$T_{\text{cpu}} = T_{\text{attn}} + T_{\text{pred}} + T_{\text{ffn}} \tag{6}$$

$$T_{\text{attn}} = C_{\text{attn}}/F_{\text{cpu}}, \; T_{\text{pred}} = C_{\text{pred}}/F_{\text{cpu}} \tag{7}$$

$$T_{\text{ffn}} = C_{\text{ffn}} \cdot P_{\text{activated}}/F_{\text{cpu}} \tag{8}$$

$$F_{\text{cpu}} = \sum_{i \in c_{\text{compute}}} F_{\text{cpu}}(i) \tag{9}$$

Table 2: Symbols used in execution planning.

Symbol	Type	Description
n	Output	Number of CPUs
c_{compute}	Output	CPU set for computation
c_{io}	Output	The CPU for I/O
m	Output	Memory usage
s	Output	Decoding speed
$n_{\max}$	User Input	Maximum number of CPUs
c_{allowed}	User Input	Allowed CPUs
$m_{\max}$	User Input	Maximum memory usage
$s_{\min}$	User Input	Minimum decoding speed
$F_{\text{cpu}}(i)$	HW Input	FLOPS of the ith CPU
$B_{\text{io}}(i, s)$	HW Input	I/O throughput on ith CPU at block size s
N_{total}	Model Input	Number of neurons
C_{attn}	Model Input	Number of weights in attention blocks
C_{pred}	Model Input	Number of weights in predictors
C_{ffn}	Model Input	Number of weights in FFN blocks
$P_{\text{activated}}$	Model Input	Average activation rate of neurons
$P_{\text{miss}}(m)$	Model Input	Cache miss rate at memory limit m
S_{neuron}	Model Input	Size of a neuron on flash storage
S_{param}	Model Input	Size of one model weight

As FFN block computation overlaps with neuron loading, the planner must also account for I/O transmission time. This is calculated by dividing the volume of neurons transferred from flash storage (Equation 10) by the I/O bandwidth. This transferred volume depends on both the activation rate and the cache miss rate.

$$V_{\text{io}} = N_{\text{total}} \cdot P_{\text{activated}} \cdot P_{\text{miss}}(m) \cdot S_{\text{neuron}}, \tag{10}$$

$$T_{\text{io}} = V_{\text{io}}/B_{\text{io}}(c_{\text{io}}, S_{\text{neuron}}) \tag{11}$$

Finally, the planner calculates the time to load neurons from memory, which relates to the weight sizes of attention blocks, predictors, and neurons activated at runtime. The memory time is determined by dividing the total weight of activated neurons for one token by the memory bandwidth (Equation 11).

6 Implementation

PowerInfer-2 is developed on top of PowerInfer [30], a state-of-the-art serving framework designed for sparsely-activated LLMs, by integrating an additional 12K lines of C++ code into PowerInfer [30]. These enhancements encompass several key areas, including the polymorphic neuron engine, neuron cache, flexible neuron loading, and neuron-cluster-level I/O pipeline.

Since PowerInfer-2 depends on privileged system APIs (e.g., `mlock` that locks pages in memory) that needs the root permission, we built it on the Android [5] platform. Even though there is no need to alter the system kernel, a rooted Android system still provides us with considerable flexibility in developing and debugging our system. Furthermore, PowerInfer-2 is inherently designed with no modifications to the kernel, making it easily portable to other operating systems, including iOS [14] platform.

The current implementation of PowerInfer-2 supports a diverse array of LLMs with varying model sizes, including Llama-2 family [27] (7B, 13B), TurboSparse-Mistral [31] (7B), and TurboSparse-Mixtral [31] (47B).

Table 3: Hardware specifications of smartphones we used in the evaluation. "DRAM" is the physical memory size. "Available" is the maximum memory size that can be occupied by an application.

Device Name	DRAM / Available	Storage	SoC
OnePlus 12	24GB / 19GB	UFS 4.0	Snapdragon 8 Gen 3
OnePlus Ace 2	16GB / 11GB	UFS 3.1	Snapdragon 8+ Gen 1

7 Evaluation

In this section, we evaluate the performance of PowerInfer-2 for various models and smartphone hardwares.
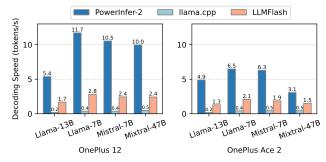
7.1 Experimental Setup

Hardware. We select one high-end and one mid-end One-Plus [25] smartphone for evaluation, with details listed in Table 3. We choose these smartphones for two reasons. First, the hardware configurations are representatives of high-end and mid-end smartphones. OnePlus 12 is a high-end smartphone model equipped with flagship hardware including a top-tier Qualcomm SoC, while OnePlus Ace 2 represents the previous generation of smartphones. Second, both phones allow rooting so we can bypass vendor-specified application constraints to unlock the full computing capabilities of smartphones.

Models. We choose four language models of varying architectures and model sizes, namely TurboSparse-Mistral-7B[3] [31] ("Mistral-7B" in figures for short), sparse Llama-7B/13B [29], and TurboSparse-Mixtral-47B[4] [31] ("Mixtral-47B" in figures for short).

Baselines. We compare PowerInfer-2 with three state-of-the-art LLM inference frameworks: llama.cpp [13], LLM in a Flash [4], and MLC-LLM [33]. Llama.cpp is currently the fastest large model inference framework that supports offloading part of model weights to flash storages (via `mmap`), and also serves as the backend for many other frameworks, such as Ollama [2]. LLM in a Flash (called LLMFlash in the evaluation) is designed for **the PC context** and not open-sourced. Therefore we have ported it to llama.cpp by implementing sparsity prediction, row-column bundling, neuron data caching and memory management in llama.cpp according to the descriptions in the original paper. MLC-LLM utilizes mobile GPUs to accelerate large model inference but it does not support weight offloading. It cannot run when the size of model parameters exceeds the available memory size. Therefore, we only compare PowerInfer-2 with MLC-LLM in scenarios where inferences can be performed entirely in memory.

For PowerInfer-2 and LLMFlash, we deployed our sparsified models, while for other baseline systems, we employed the original models for speed comparison.

Workloads. The workloads for evaluation are selected from

Figure 6: Decoding speeds of PowerInfer-2, llama.cpp and LLM-Flash. The Y axis is the generation speed. 50% model weights of FFN blocks are offloaded to flash storage for all models except TurboSparse-Mixtral-47B on OnePlus Ace 2, which requires offloading at least 75% of FFN weights.

practical LLM tasks, including multi-turn dialogue [35], code generation [9], math problem solving [10], and role play [36], in order to fully demonstrate the efficiency of PowerInfer-2. Notably, these selected workloads are the top representatives of real-world tasks from the huggingface community. We use prompt lengths of 128 and 512 tokens to test prefill performance. For decoding test, we use prompts of at most 64 tokens and generate up to 1,024 tokens. All test runs are repeated 10 times to average out fluctuations.

Key Metrics. As we focus on low latency setting, our primary evaluation metric centers on the *end-to-end* generation speed. In our evaluation, we adopted *prefill speed* (tokens/s) and *decoding speed* (tokens/s) as our metrics to provide a more straightforward representation of the system's performance.

7.2 Offloading-Based Performance

In this section, we evaluate the performance of decoding and prefill of PowerInfer-2 on different models when the amount of available memory is restricted, and compare PowerInfer-2 with llama.cpp and LLMFlash.
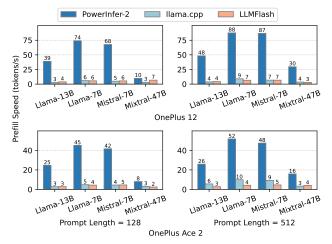
7.2.1 Decoding Performance

We first compare the decoding speed of PowerInfer-2 with the state-of-the-art LLM inference frameworks. We limit the placement of weights of FFN blocks in DRAM to 50% for all models except TurboSparse-Mixtral-47B on OnePlus Ace 2, which can only place up to 25% of FFN weights in DRAM due to its relatively low available memory size.

Fig.6 illustrates the generation speeds for various models. On the high-end OnePlus 12, PowerInfer-2 achieves 3.94× (up to 4.38×) and 25.4× (up to 29.2×) speedup on average compared to LLMFlash and llama.cpp, respectively. For the mid-end OnePlus Ace 2, PowerInfer-2 achieves 2.99× speedup on average than LLMFlash and 13.3× speedup on average than llama.cpp.

When the model size is beyond available DRAM, model weights are swapping between flash storage and memory, introducing significant I/O overhead. Although LLMFlash employs a neuron cache to store recently accessed neurons

9

Figure 7: Prefill speeds of PowerInfer-2, llama.cpp and LLMFlash in offloading scenarios at prompt lengths of 128 and 512 tokens. The X axis is the model. The Y axis is the prompt processing speed (tokens/s). The placement of weights in FFN blocks is the same as Fig. 6.

instead of directly using `mmap`, which makes it 5.35× faster than llama.cpp on average, about 10% of activated neurons still need to be loaded from flash in a blocking way, resulting in long delays before computations. In contrast, PowerInfer-2 not only utilizes an efficient neuron pipeline that overlaps I/O operations with computation, but also loading neurons with flexible bundles to improve the I/O throughput, which effectively eliminates this overhead and achieves better performance across different models.

As shown in Fig.6, the acceleration ratio of PowerInfer-2 varies with models. The variation is attributed to the actual number of activated parameters. For example, although TurboSparse-Mixtral-47B has 47 billion parameters, with the mixture-of-expert architecture and high sparsity, it only activates about 3B parameters for one token, which is close to that of TurboSparse-Mistral-7B. And that is the reason why these two models have similar performance. Notably, TurboSparse-Mixtral-47B achieves a generation speed of 9.96 tokens/s but still does not exhaust all available memory. By enlarging the neuron cache, the decoding speed can be further improved, which will be evaluated in later sections (§7.2.3). Llama-13B exhibits lower sparsity, which activates nearly 2× more parameters compared to TurboSparse-Mistral-7B, ending up with 2× slower than TurboSparse-Mistral-7B.
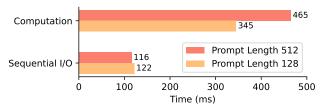
7.2.2 Prefill Performance

In this section, we evaluate the prefill performance of PowerInfer-2 at prompt lengths of 128 and 512 tokens. Fig.7 illustrates the prompt processing speeds of PowerInfer-2 and other baseline systems. For prompt length of 128 tokens, PowerInfer-2 is 6.95× faster than LLMFlash and 9.36× faster than llama.cpp on average on the high-end OnePlus 12, 7.15× and 6.61× faster on the mid-end OnePlus Ace 2. For prompt length of 512 tokens, the speedup of PowerInfer-2 reaches up

to 13.3× compared to LLMFlash when running Llama-7B on OnePlus 12 with 2.5GB memory.

The speedup is attributed to PowerInfer-2's choice to NPU-centric prefill. First, the NPU is more powerful than the CPU and GPU when performing computations of large batch sizes. The baseline systems use GPU in the prefill phase, while PowerInfer-2 takes full advantage of the NPU to accelerate the computation. Second, PowerInfer-2 fully utilizes sequential I/O at prefill stage. As the batch size gets larger, the probability of a single neuron being activated also increases. In TurboSparse-Mixtral-47B, with the batch size of 128, the probability of a neuron being activated would be 99.99%. Combining with the fact that sequential I/O is 4 GB/s, three times faster than random I/O, PowerInfer-2 uses dense computing at prefill stage, loading the whole FFN block into memory sequentially. Third, weight loading can be efficiently overlapped with the computation in PowerInfer-2 by prefetching weights of the next layer during the computation of the current layer.

Taking advantages of all these techniques, the speed of computation, rather than the speed of sequential read, has become the bottleneck of PowerInfer-2 at prefill stage. As shown in Fig.8, the computation, which includes dequantization of weights and execution of operators, takes 2.83× time than sequential I/O at prompt length of 128 tokens. With prompt length of 512 tokens, the gap grows to 4× because the computation takes more time while sequential I/O remains the same.
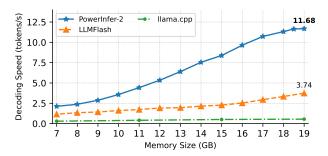

Figure 8: Computation and sequential I/O time in a layer with prompt lengths of 128 and 512 tokens at the prefill stage of TurboSparse-Mixtral-47B on OnePlus 12. The X axis is time used in milliseconds.
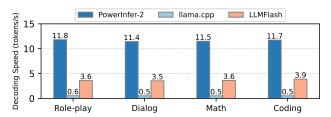
7.2.3 Performance with Various Memory Capacities

In the context of real-world usage, smartphones are tasked with running multiple applications simultaneously [1], resulting in varying available memory. To evaluate the adaptability of PowerInfer-2 under these conditions, we evaluate its performance across a range of memory capacities from 7GB to 19GB. Fig.9 presents the decoding speeds under different memory configurations for TurboSparse-Mixtral-47B.

When the available memory is limited to just 7GB, PowerInfer-2's decoding speed is 2.13 tokens/s. By design, non-FFN layer weights[5] (1GB), predictor weights (2.6GB) and FFN weights' quantization scales (2.7GB) are resident in

---

[5]Including token embeddings, self-attention layers, and the final output projection.

Figure 9: Decoding speeds on various memory configurations with TurboSparse-Mixtral-47B on OnePlus 12.

Figure 10: Decoding performance on different downstream tasks of TurboSparse-Mixtral-47B on OnePlus 12. All available memory is used during decoding.

memory, and 300MB memory is reserved for intermediate tensors, KV cache, and other runtime memory of PowerInfer-2, resulting in 6.6GB memory used in total. Given the overall limitation of 7GB, the size of the neuron cache that stores FFN weights is set to 400MB, which only caches 1.8% of FFN weights of TurboSparse-Mixtral-47B. Such a small cache renders reusing cached neuron among tokens nearly impossible. Therefore all neurons have to be fetched from flash storage for each token. Nevertheless, PowerInfer-2 still performs 1.84× faster than LLMFlash due to efficient flexible neuron loading and neuron-level pipeline. PowerInfer-2's decoding speed scales linearly with the memory size up to 18GB memory size, as the I/O is the bottleneck and the required I/O operations are reduced almost linearly with the increase in neuron cache size. Notably, when using up all available memory (19GB), PowerInfer-2 achieves a decoding speed of 11.68 tokens/s, which is 3.12× faster than LLMFlash and 21.2× faster than llama.cpp.

7.2.4 Decoding Speed Distribution

To evaluate the robustness of PowerInfer-2 in different practical LLM tasks, we investigate the distribution of decoding speeds of PowerInfer-2 at both task-level and token-level.

For task-specific decoding, we measure the average decoding speeds for four representative real-world tasks: role-play, multi-turn dialogue, math problem solving, and code generation. Fig.10 shows that PowerInfer-2 consistently achieves at least 11.4 tokens/s decoding speed across different tasks, which demonstrates the robustness of PowerInfer-2 in handling diverse tasks. The decoding speed differs slightly due to the fact that the activation sparsity of the model varies when

Table 4: Decoding latencies of PowerInfer-2 in milliseconds when offloading 50% of FFN weights.

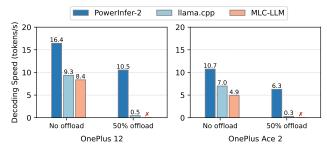|	TurboSparse-Mixtral-47B	TurboSparse-Mistral-7B
Mean	99.76	96.83
P50	97.42	93.92
P90	116.16	116.29
P99	140.56	162.91

performing different tasks.

To examine the distribution of per-token decoding speeds, we measured the average, 50th percentile (P50), 90th percentile (P90), and 99th percentile (P99) decoding speeds of 1,024 tokens on the TurboSparse-Mixtral-47B and TurboSparse-Mistral-7B. We constrained both models to place only 50% of FFN weights in DRAM. Table 4 illustrates the decoding speed distribution. For the TurboSparse-Mixtral-47B, 10% of tokens are generated 16.5% slower than the average decoding speed, and the 99th percentile decoding latency is 40.9% slower than the average latency. This performance variation is caused by varying activation similarities of neighboring tokens. Tokens with similar activations can share commonly activated neurons already resident in the neuron cache, reducing the need for I/O to fetch neurons from the flash storage. Taking TurboSparse-Mixtral-47B as an example, we observed that the average per-token neuron cache miss rate is 3.5%, while the P99 miss rate is 18.9%, which is 5.4× higher than the average. It indicates that activation similarities differs greatly among tokens. Higher miss rates result in more neurons being swapped between the neuron cache and the flash storage, and thus longer decoding time.

7.3 In-Memory Performance

If the model size fits entirely within the device's memory, PowerInfer-2 can save memory usage while maintaining a high decoding speed. This section evaluates the decoding performance of TurboSparse-Mistral-7B when sufficient memory is available. We compare PowerInfer-2 with llama.cpp and MLC-LLM, which exemplify CPU and GPU decoding on smartphones, respectively. The results are illustrated in Fig.11.

When no offloading is used and thus all model weights are resident in memory, PowerInfer-2 is 1.64× and 2.06× faster than llama.cpp and MLC-LLM on average, respectively. This is mainly attributed to the benefit of model's activation sparsity, which reduces about 70% percent of FFN computations. TurboSparse-Mistral-7B requires at least 4GB memory to store all of its model parameters. By offloading half of the FFN weights, PowerInfer-2 can save approximately 1.5GB of memory, achieving nearly 40% in memory savings while maintaining comparable performance to llama.cpp and MLC-LLM, which do not offload any weights. By contrast, offloading such amount of FFN weights deteriorates llama.cpp's decoding performance by 20.8×, and even makes MLC-LLM fail to run. It demonstrates that PowerInfer-2 is capable of

Figure 11: Decoding speeds of PowerInfer-2, llama.cpp, and MLC-LLM on TurboSparse-Mistral-7B with different offloading setups. "50% offload" means 50% model weights of FFN blocks are offloaded to flash storage. "No offload" means all model parameters are resident in memory. A red label of "✗" indicates an execution failure due to the lack of weight offloading support.

effectively reducing memory consumption for models that are already fit in memory and maintaining the user experience at the same time.

8 Related Work

Resource-Efficient LLM. Deploying LLMs on resource-restricted devices has become more and more popular [37]. A representative framework is MLC-LLM [33], which enables native deployment of many large language models on mobile devices with GPU acceleration. However, it is limited to in-memory computation scenarios, and fails to run when the model is too large to fit in memory. There are also other approaches, such as network pruning [15, 22], knowledge distillation [16], and quantization [8, 20] to reduce model memory footprints. These approaches are orthogonal to PowerInfer-2, and can be used together with PowerInfer-2 to further improve the efficiency of deploying LLM on mobile devices.

Speculative Decoding. Speculative decoding can also be utilized to enhance inference speed [7, 12, 18]. This technique uses a smaller model (e.g., 1B parameters) to quickly generate multiple candidate tokens and then validates them with a larger model (e.g., 13B parameters) in a batch. Only tokens accepted by the larger model will be displayed to users. By verifying multiple tokens at a time, SpecInfer [23] can reduce the number of decoding steps. In the offloading scenario, however, the large amount of I/O loading from flash storages becomes the bottleneck of speculative decoding, especially for MoE models, which requires to load all experts for one batch, losing the benefit of sparse activation of experts.

9 Conclusion

This paper introduces PowerInfer-2, a framework that supports high-speed inference of LLMs on smartphones, especially for models exceeding the device's memory capacity. The key insight of PowerInfer-2 is the use of heterogeneous smartphone resources to adapt matrix computations into more manageable neuron cluster computations. Evaluation on two smartphones demonstrates that PowerInfer-2 achieves up to

29.2× speedup over existing SOTA systems and is the first inference framework that manages to run extremely large language models like the TurboSparse-Mixtral-47B model on a smartphone efficiently.

References

[1] Multitasking the Android way. https://android-developers.googleblog.com/2010/04/multitasking-android-way.html, 2010.

[2] Ollama: Get up and running with large language models locally. https://github.com/ollama/ollama, 2024.

[3] Abien Fred Agarap. Deep learning using rectified linear units (ReLU), 2019.

[4] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a Flash: Efficient large language model inference with limited memory, 2024.

[5] Android. https://www.android.com/, 2024.

[6] Anthropic. https://www.anthropic.com/news/claude-3-family, 2024.

[7] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv: 2401.10774*, 2024.

[8] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. QuIP: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36, 2024.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[10] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of LLM inference using lookahead decoding, 2024.

[13] Georgi Gerganov. ggerganov/llama.cpp: Port of Facebook's LLaMA model in C/C++. https://github.com/ggerganov/llama.cpp, 2024.

[14] iOS. https://www.apple.com/ios/ios-17/, 2024.

[15] Ajay Jaiswal, Shiwei Liu, Tianlong Chen, Zhangyang Wang, et al. The emergence of essential sparsity in large pre-trained models: The weights that matter. *Advances in Neural Information Processing Systems*, 36, 2024.

[16] Jaehun Jung, Peter West, Liwei Jiang, Faeze Brahman, Ximing Lu, Jillian Fisher, Taylor Sorensen, and Yejin Choi. Impossible distillation: from low-quality model to high-quality dataset & model for summarization and paraphrasing. *arXiv preprint arXiv:2305.16635*, 2023.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[18] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

[19] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. The lazy neuron phenomenon: On emergence of activation sparsity in transformers, 2023.

[20] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[21] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja Vu: Contextual sparsity for efficient LLMs at inference time, 2023.

[22] Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-Pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.

[23] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating generative large language model serving with speculative inference and token tree verification, 2023.

[24] NVIDIA. https://www.nvidia.com/en-us/data-center/h100/, 2024.

[25] OnePlus. https://www.oneplus.com/global, 2024.

[26] OpenAI. https://openai.com/gpt-4, 2023.

[27] Konstantinos I Roumeliotis, Nikolaos D Tselikas, and Dimitrios K Nasiopoulos. Llama 2: Early adopters' utilization of Meta's new open-source pretrained model. 2023.

[28] Noam Shazeer. GLU variants improve transformer, 2020.

[29] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, and Maosong Sun. ProSparse: Introducing and enhancing intrinsic activation sparsity within large language models. *arXiv preprint arXiv:2402.13516*, 2024.

[30] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. *arXiv preprint arXiv:2312.12456*, 2023.

[31] Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. Turbo sparse: Achieving llm sota performance with minimal activated parameters. *arXiv preprint arXiv:2406.05955*, 2024.

[32] Google: Get started with Gemini Nano on Android (on device). https://ai.google.dev/gemini-api/docs/get-started/android_aicore, 2024.

[33] MLC team. MLC-LLM. https://github.com/mlc-ai/mlc-llm, 2024.

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[35] Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, Nathan Sarrazin, Omar Sanseviero, Alexander M. Rush, and Thomas Wolf. Zephyr: Direct distillation of LM alignment, 2023.

[36] Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Man Zhang, Zhaoxiang Zhang, Wanli Ouyang, Ke Xu, Wenhu Chen, Jie Fu, and Junran Peng. RoleLLM: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. *arXiv preprint arXiv: 2310.00746*, 2023.

[37] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. A survey of resource-efficient LLM and multimodal foundation models. *arXiv preprint arXiv:2401.08092*, 2024.

[38] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. LLM as a system service on mobile devices, 2024.

[39] Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. MoEfication: Transformer feed-forward layers are mixtures of experts. In *Findings of ACL 2022*, 2022.

[40] Zhengyan Zhang, Yixin Song, Guanghui Yu, Xu Han, Yankai Lin, Chaojun Xiao, Chenyang Song, Zhiyuan Liu, Zeyu Mi, and Maosong Sun. ReLU^2 wins: Discovering efficient activation functions for sparse LLMs, 2024.