PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU

Yixin Song, Zeyu Mi, Haotong Xie and Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

Abstract

This paper introduces PowerInfer, a high-speed Large Language Model (LLM) inference engine on a personal computer (PC) equipped with a single consumer-grade GPU. The key principle underlying the design of PowerInfer is exploiting the high locality inherent in LLM inference, characterized by a power-law distribution in neuron activation. This distribution indicates that a small subset of neurons, termed *hot neurons*, are consistently activated across inputs, while the majority, *cold neurons*, vary based on specific inputs. PowerInfer exploits such an insight to design a GPU-CPU hybrid inference engine: hot-activated neurons are preloaded onto the GPU for fast access, while cold-activated neurons are computed on the CPU, thus significantly reducing GPU memory demands and CPU-GPU data transfers. PowerInfer further integrates adaptive predictors and neuron-aware sparse operators, optimizing the efficiency of neuron activation and computational sparsity. The evaluation shows that PowerInfer significantly outperforms *llama.cpp* by up to 11.69× while retaining model accuracy across various LLMs (including OPT-175B) on a single NVIDIA RTX 4090 GPU. For the OPT-30B model, PowerInfer achieves performance comparable to that of a high-end server-grade A100 GPU, reaching 82% of its token generation rate on a single consumer-grade RTX 4090 GPU.

Keywords: LLM, LLM Serving, Sparsity, Consumer-grade GPU

1 Introduction

Generative large language models (LLMs) have garnered significant attention for their remarkable capabilities in sophisticated natural language processing tasks [6, 51, 58], especially in areas such as creative writing and advanced code generation. These models, widely deployed in data centers equipped with high-end and expensive server-grade GPUs, have significantly influenced our daily lives and work practices. Meanwhile, there is an emerging trend of running LLMs on more accessible local platforms [10, 56], particularly personal computers (PCs) with consumer-grade GPUs. This evolution is driven by the need for enhanced data privacy [32], model customization [29], and reduced inference costs [51]. In contrast to data-center deployments, which prioritize high throughput [23, 43, 57], local deployments focus on low latency in processing small batches.

Nonetheless, deploying LLMs on consumer-grade GPUs presents significant challenges due to their substantial memory requirements. LLMs, functioning as autoregressive Transformers, sequentially generate text token-by-token, each needing to access the entire model containing hundreds of billions of parameters. Therefore, the inference process is fundamentally constrained by the GPU's memory capacity. This limitation is particularly acute in local deployments where the processing of individual requests (often one at a time) [7] leaves minimal opportunity for parallel processing.

Compression techniques like quantization [15, 55] and pruning [30] can reduce the model size. However, even deeply compressed models remain too large for consumer-grade GPUs. For instance, an OPT-66B model with 4-bit precision demands approximately 40GB of memory just to load its parameters [27], exceeding the capacity of even high-end GPUs like the NVIDIA RTX 4090.

Model offloading is another approach that partitions the model between GPU and CPU at the Transformer layer level [4, 17, 43]. State-of-the-art systems like llama.cpp [17] distribute layers between CPU and GPU memories, leveraging both for inference, thus reducing the GPU resources required. However, this method is hindered by the slow PCIe interconnect and the CPUs' limited computational capabilities, resulting in high inference latency.

In this paper, we argue that the key reason for memory issues in LLM inference is the **locality mismatch** between hardware architecture and the characteristics of LLM inference. Current hardware architectures are designed with a memory hierarchy optimized for data locality. Ideally, a small, frequently accessed working set should be stored in the GPU, which offers higher memory bandwidth but limited capacity. In contrast, larger, less frequently accessed data are

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11…$15.00
https://doi.org/10.1145/3694715.3695964

better suited for CPUs, which provide more extensive memory capacity but lower bandwidth. Nevertheless, each LLM inference iteration requires accessing the entire set of model parameters whose total size is too large for a single GPU, thus showing no locality at all and thus impeding efficient locality exploitation.

Recent works have identified activation sparsity in LLM inference [28, 34, 61]. During each inference iteration, only a limited number of neurons[1] are activated, significantly influencing token outputs. These sparse activations, which can be accurately predicted at runtime, allow for accelerated inference by computing only the activated neurons. However, the set of activated neurons varies across inputs and can only be determined at runtime, necessitating the entire model to be loaded into GPU memory. This requirement limits the approach's applicability in local deployment scenarios with constrained GPU VRAM.

Fortunately, we have observed that neuron activation in an LLM follows a **skewed power-law distribution** across numerous inference processes: a small subset of neurons *consistently* contribute to the majority of activations (over 80%) across various inputs (hot-activated), while the majority are involved in the remaining activations, which are determined based on the inputs at runtime (cold-activated). This observation suggests an inherent locality in LLMs with high activation sparsity, which could be leveraged to address the aforementioned locality mismatch.

Building on this locality insight, we introduce **PowerInfer**, an efficient LLM inference system optimized for local deployments using a single consumer-grade GPU. The key idea of PowerInfer is to exploit the locality in LLM inference by assigning the minor hot neurons to the GPU, while cold neurons, which constitute the majority, are managed by the CPU. Specifically, PowerInfer exploits the locality in LLM inference through a two-step process: (1) PowerInfer preselects hot and cold neurons based on their statistical activation frequency, preloading them onto the GPU and CPU, respectively, during an offline phase. (2) At runtime, it employs online predictors to identify which neurons (both hot and cold) are likely to be activated for each specific input. This approach allows the GPU and CPU to independently process their respective sets of activated neurons, thereby minimizing the need for costly PCIe data transfers.

However, there are significant challenges that complicate the design of PowerInfer. First, the online predictors, which are essential for identifying active neurons in LLM layers and are typically situated on the GPU, occupy a considerable amount of GPU memory. This memory could otherwise be used for the LLM. To address this, PowerInfer introduces an adaptive method for constructing smaller predictors for layers with higher activation sparsity and skewness. This iterative process reduces the size of the predictors while maintaining their accuracy, thus freeing up GPU memory for LLM inferences.

Second, leveraging LLM sparsity requires the use of sparse operators. Conventional libraries like cuSPARSE [37] are not optimal due to their general-purpose design, which includes tracking each non-zero element and converting dense matrices into sparse formats [54, 63]. In contrast, PowerInfer designs neuron-aware sparse operators that directly interact with individual neurons, thereby bypassing operations on entire matrices. This approach enables efficient matrix-vector multiplication at the neuron level and removes the need for specific sparse format conversions.

Lastly, the optimal placement of activated neurons between the GPU and CPU in PowerInfer is a complex task. It involves evaluating each neuron's activation rate, intra-layer communication, and available hardware resources like GPU memory sizes. To effectively manage this, PowerInfer utilizes an offline phase to generate a neuron placement policy. This policy uses a metric that measures each neuron's impact on LLM inference outcomes and is framed as an integer linear programming problem. The policy formulation considers factors such as neuron activation frequencies and the bandwidth hierarchy of CPU and GPU architectures.

The online inference engine of PowerInfer was implemented by extending llama.cpp with an additional 4,200 lines of C++ and CUDA code. Its offline component, comprising a profiler and a solver, builds upon the transformers framework [53] with approximately 400 lines of Python code. PowerInfer is compatible with various popular LLM families, including OPT (7B-175B), LLaMA2 (7B-70B), and Falcon-40B, and supports consumer-grade GPUs like the NVIDIA RTX 4090 and NVIDIA RTX 2080Ti.

Performance evaluation reveals that PowerInfer, when deployed on a PC equipped with a single NVIDIA RTX 4090 GPU, delivers an average generation speed of 13.20 tokens/s for quantized models and 8.32 tokens/s for non-quantized models, maintaining model accuracy. These results significantly surpass llama.cpp's performance, exhibiting up to 8.00× and 11.69× improvements for quantized and non-quantized models, respectively. Significantly, the inference speed achieved on an NVIDIA RTX 4090 GPU (priced at approximately $2,000) is only 18% slower compared to the performance on a top-tier A100 GPU (costing around $20,000) that can fully accommodate the model. PowerInfer's code has been open sourced completely.

2 Background and Motivation

2.1 LLM Inference & Architecture

LLM inference, an autoregressive model, generates each token based on previous ones. The process starts with a prompt and unfolds in two phases: first, the prompt phase outputs

[1]This paper defines a neuron as a specific row/column in a weight matrix.

an initial token, then the generation phase sequentially produces tokens until a maximum limit or an end-of-sequence (<EOS>) token is reached. Each token generation, an inference iteration, requires running the full LLM model.

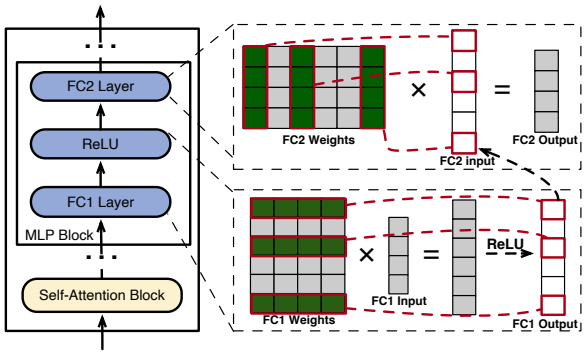


Figure 1. The architecture of a Transformer layer and how neurons are sparsely activated in FC1 and FC2 layers due to the ReLU function. The neurons that are activated are represented as green rows or columns encircled by red lines. The output vector from FC1 is then supplied to FC2 as its input vector.

The LLM architecture includes multiple Transformer layers, each comprising a self-attention and an MLP (Multi-Layer Perceptron) block (see Figure 1, left). The self-attention block generates embedding vectors by capturing the relationships among input tokens. In this process, different heads focus on extracting distinct feature information. The computation results from these different heads are aggregated and then utilized as the input for the MLP block. The MLP block applies non-linear transformations via fully connected layers and activation functions to refine the input sequence representation. The output either advances to subsequent layers or forms the LLM's final output.

In Figure 1 (right), the MLP block's layers, FC1 and FC2, generate vectors through matrix multiplication. Each output element comes from the dot product of an input vector and a neuron (a row/column in a weight matrix). Activation functions (like ReLU [2] and SiLU [40]) act as gates to selectively retain or discard values in a vector, influencing neuron activations in FC1 and FC2. For example, ReLU in this figure filters out negative values, allowing only positively valued neurons in FC1 to influence the output. These neurons, which contribute to the output, are considered activated in this paper. Similarly, these values also affect which neurons in FC2 are activated and involved in the computation of its output.

Activation Sparsity. LLM inference exhibits notable sparsity in neuron activation, a phenomenon observed in both self-attention and MLP blocks [26, 28, 59]. In self-attention, nearly half of the attention heads contribute minimally, while in MLP blocks, sparsity is largely due to activation function characteristics. This MLP sparsity is particularly pronounced in ReLU-based architectures, with DejaVu [28] reporting that

Table 1. Average activation sparsity for various LLMs in the MLP blocks. For ReLU-based models, sparsity is the proportion of neurons with zero activation. For SwiGLU-based models, it's the proportion of neurons that can be dynamically pruned with less than 1% impact on perplexity.

LLM	Activation Function	Sparsity
OPT-30B	ReLU	97%
LLaMA2-13B	SwiGLU	43%
Yi-34B	SwiGLU	53%

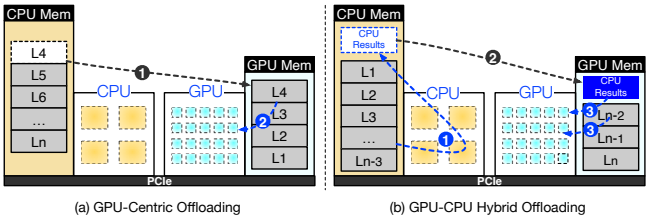


Figure 2. Typical existing offloading solutions. (a) shows a GPU-centric approach, while (b) is the CPU-GPU hybrid offloading approach.

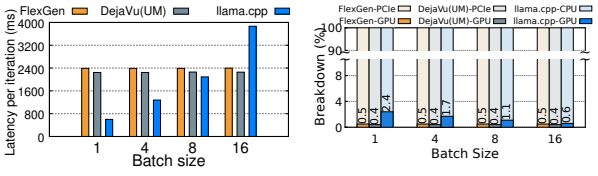
approximately 80% of neurons in the OPT-30B model remain inactive during inference. Moreover, this phenomenon is not limited to ReLU-based models; it is also observed in architectures using other activation functions such as SwiGLU. For instance, Table 1 shows that LLaMA2-13B and Yi-34B, which employ SwiGLU, exhibit sparsity levels of 43% and 53% respectively. These findings align with prior research results from studies like CATS [24] and ReLU2 [61].

Moreover, it is possible to predict neuron activations a few layers in advance within the ongoing model iteration. Based on this observation, DejaVu [28] employs MLP-based online predictors during inference and only processes the activated neurons, achieving over a 6x speedup while maintaining an impressive accuracy rate of at least 93% in predicting neuron activation. However, the activation sparsity is input-specific for each inference iteration, meaning that the activation of specific neurons is directly influenced by the current input and cannot be predetermined before the model's inference iteration begins.

2.2 Offloading-based LLM Serving

The offloading technique, which leverages the CPU's additional computational and memory resources, presents a more viable solution for accommodating LLMs that exceed the GPU's memory capacity. In this section, we delve into the analysis of offloading systems to uncover the factors contributing to their sluggish performance. Figure 2 illustrates two main offloading approaches:

GPU-centric offloading utilizes CPU memory to store portions of the model parameters that exceed the GPU's capacity. During each iteration, as depicted in Figure 2a), it



(a) (b)

Figure 3. *Performance comparison and analysis for serving OPT-30B on NVIDIA RTX 4090 GPU.* The *yellow* blocks refer to FlexGen, the *gray* blocks refer to DejaVu (UM) and the *blue* blocks refer to llama.cpp. (a) The Y-axis indicates execution time for one iteration and the X-axis represents batch sizes for input. (b) The Y-axis indicates the proportion of execution time, and the X-axis indicates batch sizes for input.

processes the parameters located in the GPU memory, transferring more from the CPU as needed. This strategy enables the inference of LLMs of varying sizes, provided that sufficient combined CPU memory and hard disk storage are available. FlexGen [43] is a typical example that adopts a zig-zag scheduling approach to prioritize throughput over latency, processing batches sequentially for each layer. Nonetheless, this method leads to substantial per-token latency in latency-sensitive scenarios (Figure 3a), mainly due to frequent data transfers between GPU and CPU. Over 99.5% of processing time is consumed by transferring LLM weights from CPU to GPU, significantly impacting overall latency, as illustrated in Figure 3b.

Although DejaVu [28] leverages activation sparsity to accelerate LLM inference, this approach, originally designed for data center environments, faces challenges when applied to consumer-grade GPUs incapable of hosting full-scale LLMs. The key challenge with DejaVu in such contexts stems from the need to frequently transfer activated neurons from the CPU to the GPU during runtime. For LLMs like OPT-30B that exceed GPU memory limits, DejaVu[2], albeit reducing the computational load on the GPU, is constrained by the data transfer procedure (Figure 3a). Consequently, as shown in Figure 3a, DejaVu experiences significant inference latency, comparable to that of FlexGen.

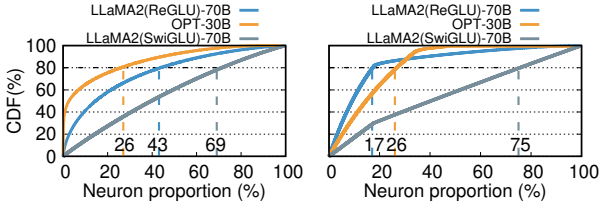
Hybrid offloading distributes model parameters between GPU and CPU, splitting them at the Transformer layer level (Figure 2b), with llama.cpp [17] as an example. The CPU processes its layers first, then sends intermediate results to the GPU for token generation. This offloading method reduces inference latency to around 600ms (Figure 3a) by minimizing data transfer and mitigating slow PCIe bandwidth. However, compared to the 45ms latency of the 30B model on A100, the speed is still too slow.

[2]Since DejaVu only works for GPU, we modified it by using NVIDIA Unified Memory (UM) [36] to fetch parameters from CPU memory.

Hybrid offloading still faces the locality mismatch issue, leading to suboptimal latency. Each inference iteration accesses the entire model, resulting in poor locality for hierarchical GPU-CPU memory structures. GPUs, while computationally powerful, are constrained by memory capacity. For instance, a 30B-parameter model on a 24GB NVIDIA RTX 4090 GPU means only 37% of the model is on the GPU, shifting most computational tasks to the CPU. The CPU, with higher memory but lower computational power, ends up handling 98% of the total computational time(Figure 3b).

3 Insights into Locality in LLM Inference

3.1 Insight-1: Power-law Activation



(a) (b)

Figure 4. Cumulative distribution function (CDF) of neuron activation in OPT-30B, LLaMA2(ReGLU)-70B and LLaMA2(SwiGLU)-70B. (a) CDF in a single MLP block. (b) CDF across the entire model. The X-axis shows neuron proportion. The Y-axis represents the CDF of neuron activation.

LLM inference exhibits a high degree of locality, indicating that a consistent group of neurons is *frequently* activated. Notwithstanding the input dependence of LLM activation sparsity, a power-law distribution is evident among activated neurons. We profile from wikipedia [13] dataset to collect the statistics of the activation sparsity about 1M tokens. Figure 4a reveals that in the MLP blocks of OPT-30B (using ReLU), LLaMA2 (ReGLU)-70B and LLaMA2 (SwiGLU)-70B, 26%, 43% and 69% of neurons respectively are responsible for 80% of total activations. This indicates that these neurons are frequently activated, which we termed as *hot-activated* neurons. Conversely, the activation of the remaining 74%, 57% and 31% of neurons is input-dependent, classifying them as *cold-activated* neurons. This distribution is particularly pronounced in ReLU-based models compared to those using other activation functions due to its higher sparsity.

Furthermore, this high locality is not confined to a single MLP block but extends throughout the entire model. As illustrated in Figure 4b, approximately 17% of neurons in OPT-30B, 26% in LLaMA2 (ReGLU)-70B, and 75% in LLaMA2 (SwiGLU)-70B are responsible for 80% of the total activations across all layers. Figure 5a clearly demonstrates that in the OPT-30B model, each transformer layer contains a small portion of hot neurons, while the majority are cold neurons. Notably, in the initial 24 layers, OPT-30B exhibits exceptionally low activation, with less than 1% of neurons becoming

active, resulting in a minimal presence of hot neurons within these layers.

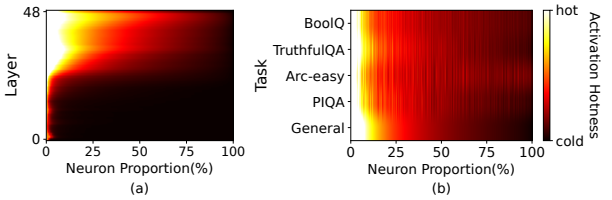
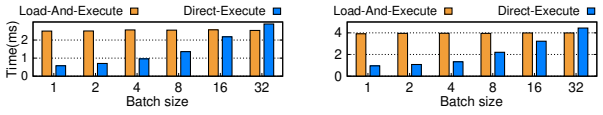


Figure 5. Activation frequency in OPT-30B. (a) The activation frequency of neurons in different layers. The Y-axis represents the layer id. (b) The activation frequency of neurons in different tasks for 30th layer. The Y-axis represents the tasks. The X-axis shows neuron proportion.

Interestingly, the same group of hot neurons consistently remains active across various downstream tasks. To demonstrate this, we profiled neuron activation across diverse datasets, including knowledge-based, truthfulness assessment, and reasoning tasks. Figure 5b illustrates that these hot neurons are consistently activated consistently across different tasks, indicating a stable pattern of power-law activation. Our analysis revealed that there is over 90% overlap in the top 20% most frequently activated neurons across these varied domains. This remarkable consistency suggests that the power-law distribution of neuron activations is an intrinsic property of the model architecture rather than being dataset-specific.

3.2 Insight-2: Fast In-CPU Computation



(a) MLP block **(b)** Attention block

Figure 6. Comparison of execution time for load-then-execute versus direct-execute methods when 10% and 60% neuron weights of one MLP and attention block in OPT-30B are CPU-resident. The X-axis shows input batch sizes, and the Y-axis measures execution time (ms). Load-then-execute involves transferring these neuron weights to GPU memory for computation, whereas direct-execute computes them directly on the CPU.

If activated neurons reside in CPU memory, computing them on the CPU is faster than transferring them to the GPU, especially with the small number of activated neurons and the small batch sizes typical in local deployments. Modern CPUs with vector extensions can efficiently handle such smaller matrix computations.

We compared the time to load and compute 10%[3] sparisty of the MLP block and 60% of attention block's CPU-side neurons on the GPU versus direct CPU execution in OPT-30B. Results in Figure 6 indicate that for batch sizes under 32, the time taken to transfer the weights of these neurons and compute them on the GPU (NVIDIA RTX 4090) exceeds the time required for calculation directly on the CPU using the AVX2 vector extension.

[3]While Insight-1 indicates that 43% of neurons account for 80% of the total activations in a single MLP block, it is typically found that only about 10% of its neurons are activated during an individual inference iteration.

4 PowerInfer Overview

We present PowerInfer, a low-latency LLM inference system deployed in a PC equipped with a single consumer-grade GPU. PowerInfer proposes a neuron-aware offloading strategy and an inference engine by fully leveraging the high locality insights described in §3. It utilizes both GPU and CPU for weight storage, accommodating LLMs of various sizes. This offloading approach, based on *Insight-1*, effectively exploits the power-law distribution of LLM inference. Specifically, PowerInfer preloads the GPU with weights for neurons that activate frequently, while less active neurons' weights are kept on the CPU.

To reduce inference latency, the inference engine computes only neurons predicted as active by online predictors, skipping most inactive ones. Moreover, the preloading strategy enables PowerInfer to allocate the bulk of inference tasks to the GPU, given that hot-activated neurons that have been loaded on the GPU constitute a major fraction of activations. For cold-activated neurons not in GPU memory, PowerInfer executes their computations on the CPU, eliminating the need for weight transfers to the GPU (*Insight-2*).

The effectiveness of PowerInfer is directly correlated with the model's activation sparsity. ReLU-family LLMs, exhibiting over 90% sparse activations in their FFNs, are ideal candidates for PowerInfer. While LLMs with other activation functions typically show around 50% sparsity, resulting in less pronounced acceleration, PowerInfer still offers some performance gains. Encouragingly, there is a growing trend on enhancing LLM sparsity without compromising performance [44, 46], or directly training LLMs with ReLU-family activations, as seen in NVIDIA Nemotron [1] and MiniTron [48]. This trend is expected to extend PowerInfer's applicability across a more diverse spectrum of LLMs.

4.1 Architecture and Workflow

Figure 7 presents an architectural overview of PowerInfer, comprising both offline and online components. Due to the variation in locality properties among different LLMs, the offline component should profile LLMs' activation sparsity, differentiating between hot and cold neurons. In the online phase, the inference engine loads two types of neurons into both GPU and CPU, serving LLM requests with low latency during runtime.

LLM Profiler and Policy Solver (Offline): Based on *Insight-1*, the neuron's activation pattern can be collected enough from general datasets, so we take an offline profiler that collects activation data from inference processes using

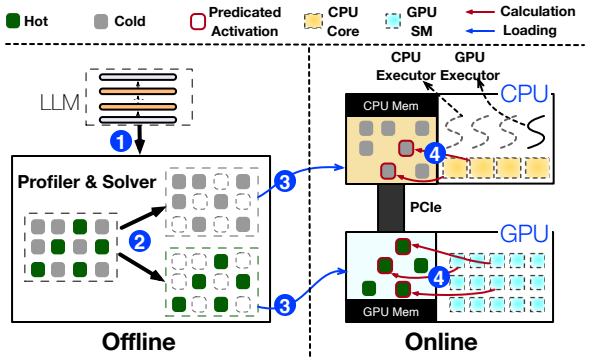


Figure 7. The architecture and inference workflow of PowerInfer.

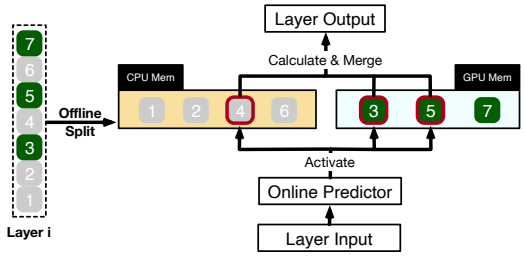


Figure 8. An illustrative example shows how PowerInfer calculates different neurons for one LLM layer.

requests derived from general datasets (e.g., C4 [39]). It monitors neuron activation across all layers (Step ①), followed by a policy solver categorizing neurons as hot or cold. The solver aims to allocate frequently activated neurons to the GPU and others to the CPU. It uses a neuron impact metric and hardware specifications to balance the workload, using integer linear programming to maximize the GPU's impact metric for neurons (Step ②).

Neuron-aware LLM Inference Engine (Online): Before processing user requests, the online engine assigns the two types of neurons to their respective processing units (Step ③), as per the offline solver's output. During runtime, the engine creates GPU and CPU executors, which are threads running on the CPU side, to manage concurrent CPU-GPU computations (Step ④). The engine also predicts neuron activation and skips non-activated ones. Activated neurons preloaded in GPU memory are processed there, while the CPU calculates and transfers results for its neurons to the GPU for integration. The engine uses sparse-neuron-aware operators on both CPU and GPU, focusing on individual neuron rows/columns within matrices.

4.2 Single Layer Example

Figure 8 illustrates how PowerInfer coordinates GPU and CPU in processing a layer's neurons. It classifies neurons based on offline data, assigning hot-activated ones (e.g., indices 3, 5, 7) to GPU memory and others to CPU memory. Upon receiving an input, a predictor identifies which neurons in the current layer are likely to be activated. For instance, it predicts activation for neurons 3, 4, and 5. It is crucial to note that hot-activated neurons, identified through offline statistical analysis, may not consistently match the runtime activation behaviors. For example, neuron 7, though labeled as hot-activated, is forecasted to be inactive in this case.

Both the GPU and CPU concurrently process their respective predicted active neurons, while efficiently bypassing inactive ones. Specifically, the GPU computes neurons 3 and 5, leveraging its parallel processing capabilities, while the CPU simultaneously handles neuron 4. Upon completion of neuron 4's computation on the CPU, its output is swiftly transferred to the GPU. The GPU then performs a final integration step, combining the results from all activated neurons (3, 4, and 5) using an optimized add operator.

5 Neuron-aware Inference Engine

5.1 Adaptive Sparsity Predictors

The online inference engine in PowerInfer reduces computational loads by only processing those neurons that are predicted to be activated. This method was also used in DejaVu [28], which advocates for training a set of fixed-size MLP predictors. Within each Transformer layer, DejaVu utilizes two separate predictors to forecast the activation of neurons in the self-attention and MLP blocks. Consequently, the inference computation is confined to neurons predicted to be active.

However, designing effective predictors for local deployments with limited resources is challenging, balancing prediction accuracy and model size. These predictors, frequently invoked for neuron activation prediction, should be stored in GPU memory for fast access. Yet, the considerable memory requirements of numerous fixed-size predictors can encroach upon the space needed for storing LLM parameters. For example, predictors for the OPT-175B model require around 27GB of GPU memory, surpassing an NVIDIA RTX 4090 GPU's capacity. On the other hand, naively reducing predictor size impairs accuracy; a decrease from 480MB to 320MB in predictor size dropped its accuracy from 92% to 84%, further adversely affecting the overall LLM accuracy (e.g., winogrande [41] task accuracy from 72.77% to 67.96%).

We have observed that the size of predictors is influenced by two main factors: the sparsity of LLM layers and their internal skewness. As shown in Figure 9, layers with higher activation sparsity simplify the task of identifying activated neurons, allowing for smaller predictor models. In contrast, layers with lower activation sparsity necessitate larger models with more parameters, as accurately pinpointing activated neurons becomes increasingly challenging. Additionally, in cases of high skewness, where activations are heavily concentrated in a few neurons, even a compact predictor can achieve high accuracy.

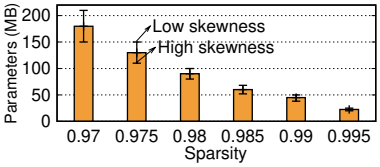


Figure 9. Correlation between predictor parameter size and layer sparsity at a guaranteed 95% accuracy level for OPT-175B. The X-axis represents sparsity, and the Y-axis represents the predictor parameter size. The bar indicates the average parameter size for the model in the corresponding sparsity, while the error bar reflects fluctuations in the predictor parameter size due to skewness within the layer.

To optimize for these factors, We design an iterative training method for **non-fixed-size** predictors for each Transformer layer. The process begins by establishing a baseline model size based on the layer's sparsity profile (Figure 9). Subsequently, the model size is iteratively adjusted, considering the internal activation skewness to maintain accuracy. An MLP predictor typically comprises a input, a single hidden, and an output layers. Since the dimensions of the input and output layers are determined by the Transformer layer's structure, modifications primarily target the hidden layer.

In training predictors, model perplexity on the WikiText-2 dataset served as the benchmark for the predictive accuracy. Initially, the predictor's dimensionality was set in accordance with the sparsity level of the layer. We then engaged in an iterative process to tune the size of the hidden layer. This process was continued until the perplexity of the model with the integrated predictor closely approximated that of the baseline model, achieving a discrepancy of less than 0.1%. Through this approach, PowerInfer effectively limits predictor parameters to a mere 6% of the total LLM parameters.

5.2 Neuron Management

When the offline solver determines a neuron placement policy, the online inference engine of PowerInfer loads the model into the CPU and GPU memory as per the policy. For each layer, which may consist of multiple weight matrices, PowerInfer assigns each neuron to either the GPU or CPU based on whether the neuron is hot-activated.

Ensuring the accurate computation of these segmented neurons in their proper sequence is vital for precise results. To this end, PowerInfer creates two neuron tables, one located in the CPU and the other in the GPU memory. These tables correlate each neuron to its original position in the matrix. During the process of multiplying with an input tensor, each neuron interacts with its corresponding tensor value, guided by the mappings in the neuron tables. The additional memory required for these neuron tables is relatively minimal, totaling only about 9MB for an LLM like OPT-175B, which needs 350GB of storage.

5.3 GPU-CPU Hybrid Execution

PowerInfer implements a GPU-CPU hybrid execution model where both units independently compute their respective activated neurons. This approach balances the computational workload between GPU and CPU, leveraging the strengths of each unit while minimizing transfer time inefficiencies. The GPU processes preloaded hot neurons, while the CPU handles cold neurons, with results combined on the GPU.

Before inference, PowerInfer constructs a computationally directed acyclic graph (DAG) with each node representing a computational LLM inference operator and stores it in a global queue in the CPU memory. Each operator in the queue is tagged with its prerequisite operators. During inference, two types of executors, pthreads created by the host OS, manage calculations on both CPU and GPU. They pull operators from the global queue, check dependencies, and assign them to the appropriate processing unit. The GPU and CPU use their neuron-aware operators, with the GPU executor launching GPU operators using APIs like cudaLaunchKernel, and the CPU executor coordinating unoccupied CPU cores for calculations. Before executing an operator, the CPU executor also determines the necessary thread count for parallel computation. To manage operator dependencies, especially when a parent node of a CPU operator is processed on the GPU, a *barrier* ensures GPU computations are complete before the CPU starts its operator.

In scenarios where activated neurons are split between GPU and CPU, synchronization between these processing units also becomes crucial. After one unit finishes its neuron calculations, it waits for the other to merge results. As GPU neurons are activated more frequently, PowerInfer assigns merging operations to the GPU.

5.4 Neuron-aware Operator

Considering the activation sparsity in LLMs, matrix multiplication operations can bypass inactive neurons and their weights, necessitating the use of sparse operators. However, current sparse matrix multiplication tools, including state-of-the-art sparse-aware compilers like SparTA [64] and Flash-LLM [54], as well as libraries like cuSPARSE [37], fall short in this regard. They either support only static compilation of sparse-aware kernels or require dynamic conversion of sparse matrices into dense formats, leading to significant performance overhead, especially with the dynamic sparsity in our scenario. Additionally, the dynamic JIT compiler PIT [63], though efficient for general sparse matrix multiplication on GPUs, is not suited for CPU-GPU hybrid execution where CPU computational capabilities are limited.

To overcome these limitations, PowerInfer introduces neuron-aware operators that directly compute activated neurons and their weights on both GPU and CPU without the need for runtime conversion to dense format. These operators differ from traditional ones as they focus on individual

row/column vectors within a matrix rather than the entire matrix. They first determine a neuron's activation status and then process it if predicted to be active, alongside the corresponding row or column of the parameter matrix.

Neuron-aware Operators for GPU: Despite vector-vector calculations being less efficient than matrix-vector calculations on GPU, neuron-aware operators based on vector-vector computation are advantageous when the batch size is small. They avoid unnecessary computations and memory operations associated with inactive neurons and do not need costly matrix conversions. Furthermore, these operators allow all thread blocks to concurrently check neuron activations and compute corresponding vectors if activated. Notably, PowerInfer assigns independent computational tasks to different thread blocks on the GPU. Each thread block processes a distinct set of neurons, eliminating the need for synchronization or divergence across blocks.

Neuron-aware Operators for CPU: Neuron-aware operators are particularly beneficial for CPUs, which generally have lower parallelism. The CPU executor assigns a neuron-aware operator to multiple cores, dividing neurons into smaller batches for concurrent activation checking. Each core processes only the activated neurons in its batch, optimizing vector-vector calculations with hardware vector extensions like AVX2.

6 Neuron Placement Policy

To fully harness the computational power of GPUs and CPUs, a locality-aware policy is crucial for optimizing performance. If we randomly place neurons on GPU and CPU, the GPU cannot fully utilize its powerful computation capability because some hot neurons are placed on CPU. Further, simply assigning the hotest neurons to the GPU may lead to excessive data transfers between the CPU and GPU, undermining efficiency. To tackle this challenge, PowerInfer's offline component devises a placement policy using a solver that determines the allocation of each neuron to GPU or CPU.

6.1 Offline Profiling

Before determining the placement of each neuron, it is important to profile the activation information of each neuron. Based on *Insight-1* that the hot neurons in general corpus are also activated frequently across different scenarios, we achieve the profile with an offline profiler, which deploys the LLM to handle requests generated from multiple general datasets, such as C4 [39] and Wikipedia [13]. To accurately measure activation information, the profiler inserts a monitoring kernel after each block within a Transformer layer. Additionally, it builds a neuron information table on the GPU, designed to track the activation count of each neuron.

6.2 Neuron Impact Metric

The neuron impact metric measures each neuron's contribution to the LLM's overall inference outcome, crucial for GPU neuron allocation. We calculate this metric effectively by leveraging the fact that profiled activation frequency mirrors runtime behavior accurately, provided the profiling involves a substantial amount of input data. As Equation 1 shows, this metric for a neuron is defined by its activation frequency obtained during profiling.

Table 2. Terminology for ILP formulation. The Par represents the parameters gathered from the profiler or the expressions used to define constraints, none of which need to be solved by the solver. The Var refers to the constraint and objective variables that emerge from the modeling process, which need to be solved by the solver.

Symbol	Type	Description
$\mathbb{L}$	Par	All layers
$\mathbb{N}$	Par	All neurons
$\mathbb{U}$	Par	CPU and GPU
f_i	Par	Activation frequency of neuron i
N_i	Par	Neuron in layer i
v_i	Par	Neuron impact for neuron i
M_i	Par	The memory size for neuron i
$MCap_j$	Par	The memory size for processing unit j
$Bandwidth_j$	Par	The memory bandwidth for processing unit j
T_{sync}	Par	The time required for one synchronization between the CPU and GPU
K	Par	A large positive number
a_{in}	Var	Whether neuron n is placed on processing unit i
T_l^j	Var	The time for computing one neuron in layer l on processing unit j
C_l	Var	The minimum number of neurons required to be allocated on the GPU when the solver opts to split neurons in layer l
y_l	Var	Binary auxliary variable for layer l to facilitate the modeling of conditional constraints

$$v_i = f_i \qquad \forall i \in \mathbb{N} \tag{1}$$

6.3 Modeling of Neuron Placement

Based on the neuron impact metric, PowerInfer utilizes a solver to optimize the total impacts of all neurons in the GPU. This cumulative impact is formulated as the objective function, as defined in Equation 2. This function is then input into an integer linear programming framework to identify a specific solution that maximizes the objective function. The binary variable a_{in}, defined in Equation 3 indicates whether the neuron n is placed on processing unit i.

$$Maximize \quad t_i = \sum_{e \in \mathbb{N}} a_{ie} * v_e \forall i \in \{GPU\} \tag{2}$$

$$\sum_{i \in \mathbb{U}} a_{in} = 1 \quad \forall n \in \mathbb{N} \tag{3}$$

When maximizing the objective function, the solver also needs to consider two sets of constraints. First, minimize the **communication overhead** between processing units. Second, ensure the GPU memory is fully utilized.

6.3.1 Communication Constraint The number of neurons preloaded onto the GPU is limited by layer communication overheads, constrained by hardware PCIe bandwidth. Preloading too few neurons negates the GPU's computational benefits. Consequently, the solver must determine a

minimum neuron allocation for optimal GPU processing efficiency, as detailed in Inequality 4. In this inequality, C_l is the minimum count of neurons that must be assigned to the GPU for layer l.

When solving Inequality 4, it is essential to profile both the computation time for an individual neuron and the intra-layer communication overhead, T_{sync}. In LLM inference, particularly with smaller batch sizes, the limiting factor is memory bandwidth. Hence, a neuron's computation time is roughly equal to the duration required to access its weights once, as shown in Equation 5. And the extent of intra-layer data transfer tends to be consistent across layers, leading to a uniform synchronization cost. Consequently, we describe T_{sync} as the profiled overhead for a single instance of intra-layer communication.

$$C_l \cdot T_l^{GPU} + T_{sync} \leq C_l \cdot T_l^{CPU} \forall l \in \mathbb{L} \tag{4}$$

$$T_i^j = M_i / Bandwidth_j \quad \forall j \in \mathbb{U}, \forall i \in \mathbb{L} \tag{5}$$

6.3.2 Memory Constraint Neuron placement is further constrained by the memory capacities of the processing units, as defined in Inequality 6. Moreover, the solver ensures that when allocating neurons of a layer to the GPU, it either assigns at least the minimum number of neurons specified in Inequality 4 to offset communication costs or opts not to allocate any neurons from that layer to the GPU. Specifically, the number of neurons for layer l on the GPU must either exceed C_l or be equal to zero.

To model this constraint, we introduce an auxiliary binary variable, y_l, which can be either 1 or 0. This variable determines whether any neurons are assigned to the GPU for layer l. For computational convenience, a sufficiently large number K is also introduced. Inequalities 7 and 8 are formulated to model this constraint. When y_l is 1, indicating neuron placement on the GPU for this layer, and given that K is adequately large, these two inequalities effectively become $C_l \leq \sum_{e \in N_l} a_{ie} \leq K$. Conversely, if y_l is set to 0, signifying no neuron placement on the GPU for layer l, the inequalities reduce to $\sum_{e \in N_l} a_{ie} = 0$.

$$\sum_{n \in N} a_{jn} \cdot M_n < MCap_j \quad \forall j \in \mathbb{U} \tag{6}$$

$$\sum_{e \in N_l} a_{ie} \geq C_l \cdot y_l \quad \forall l \in \mathbb{L}, \forall i \in \{GPU\} \tag{7}$$

$$\sum_{e \in N_l} a_{ie} \leq K \cdot y_l \quad \forall l \in \mathbb{L}, \forall i \in \{GPU\} \tag{8}$$

6.3.3 ILP Optimization The solver utilizes Integer Linear Programming (ILP) to maximize the objective function, conforming to all the constraints from Equation/Inequality 3 to 8. Given that ILP problems are inherently NP-complete, directly solving them for an LLM with hundreds of billions of parameters poses a considerable computational challenge. To expedite the process and achieve an approximate solution, the primary strategy involves aggregating neurons within each layer into batches for collective placement analysis. Specifically, the solver groups 64 neurons with similar impacts from a layer into a single batch. This batching strategy dramatically reduces the total neuron count, N, from several millions to roughly tens of thousands, thereby significantly decreasing the time to solve the ILP problem to approximately 10 seconds.

7 Implementation

The online inference engine of PowerInfer has been implemented by incorporating 4,200 lines of C++ and CUDA code into llama.cpp [17], a state-of-the-art open-source LLM inference framework designed for PCs. The extensions made by PowerInfer include modifications to the model loader for distributing an LLM across GPU and CPU, following the guidance from the offline solver's outputs. We have also optimized the inference engine for GPU-CPU hybrid execution and introduced 10 neuron-aware operators for both processing units. All other components and functionalities of llama.cpp remains unchanged. For instance, the KV cache continues to reside in CPU memory, allowing more GPU memory for hot-activated neurons. Furthermore, around 400 lines of Python code were added to the transformers framework [53], enabling it to function as an offline profiler and solver for PowerInfer.

The current implementation of PowerInfer supports a range of mainstream LLM families with varying parameter sizes. For these models, PowerInfer utilized general corpora such as Wikipedia [13] to train online activation predictors, Notably we deliberately avoided using any of the downstream task datasets in our predictor training process. Training predictors, though time-consuming (often several hours), is a one-time task that can be expedited with multiple GPUs.

8 Evaluation

8.1 Experimental Setup

Hardware. All experiments were conducted on two distinct PC configurations, representing both high-end and low-end hardware scenarios:

- **PC-High**: Intel i9-13900K processor (eight 5.4GHz cores), 192GB host memory (bandwidth of 67.2 GB/s), an NVIDIA RTX 4090 GPU (24GB), and PCIe 4.0 interface (64GB/s bandwidth).
- **PC-Low**: Intel i7-12700K processor (eight 4.9GHz cores), 64GB host memory (bandwidth 38.4 GB/s), an NVIDIA RTX 2080Ti GPU (11GB), and PCIe 3.0 interface (32GB/s bandwidth).

Models. Table 3 shows the LLMs evaluated in this section, including average activation sparsity for various LLMs in the MLP blocks. These LLMs include OPT [58] models with

Table 3. All LLMs used for evaluation and their average activation sparsity in the MLP blocks. The sparsity is measured using the same method in Table 1.

Model	Activation Function	Sparsity
Bamboo-7B	dReLU	90%
OPT-7B/13B/30B/66B/175B	ReLU	96%-98%
Falcon(ReLU)-40B	ReLU	95%
LLaMA2(ReGLU)-7B	ReGLU	70%
LLaMA2(ReGLU)-13B	ReGLU	78%
LLaMA2(ReGLU)-70B	ReGLU	82%
Qwen1.5-4B	SwiGLU	40%
LLaMA2(SwiGLU)-13B	SwiGLU	43%
Yi-34B	SwiGLU	53%

parameters from 13B to 175B, Bamboo 7B [45], as well as Falcon(ReLU)-40B [3] and LLaMA(ReGLU)-70B [47] models. In addition to LLMs using ReLU-family activation functions, we also evaluate SiLU-family LLMs, with sparsity levels of approximately 50%. For our experiments, all models in our experiments use FP16 and INT4 quantized parameters, with intermediate activations in FP32, consistent with recent LLM research practices [15, 57].

Workloads. The workloads for our experiments are derived from chatbot arena [62] ChatGPT prompts [35] and Alpaca [50] datasets. covering a wide spectrum of language model uses. These datasets are the most representative examples of real LLM services. ChatGPT prompts include user interactions with ChatGPT [38], and Alpaca features instruction sets generated by GPT3.5 through self-instruction.

Baseline System. We compare PowerInfer with llama.cpp [17] and SpecInfer [33], state-of-the-art local LLM inference frameworks. llama.cpp is the most widely used LLM inference framework for local scenarios. For a fair comparison, we extended llama.cpp to support the OPT model, as it does not natively do so. SpecInfer is representative speculative decoding framework, that utilizes smaller draft models to generate tokens and subsequently verifies these tokens in batches using the original model. While other alternatives like FlexGen [43] and DejaVu [28] exist, they exhibit higher latency in the latency-sensitive scenarios discussed in this paper, as analyzed in §2.2.

8.2 End-to-End Performance

We first compare the end-to-end inference performance of PowerInfer , llama.cpp and SpecInfer with a batch size of one, the typical setting for local deployments [7]. Given real-world dialog input/output length variability [23], we sample prompts from Alpaca and ChatGPT datasets, ranging from 8 to 128 tokens and measure the generation speed.

Figure 10 illustrates the generation speeds for various models and input-output configurations on a PC-High equipped with an NVIDIA RTX 4090. On average, PowerInfer achieves a generation speed of 8.32 tokens/s, reaching up to 16.06 tokens/s, significantly outperforming llama.cpp and SpecInfer

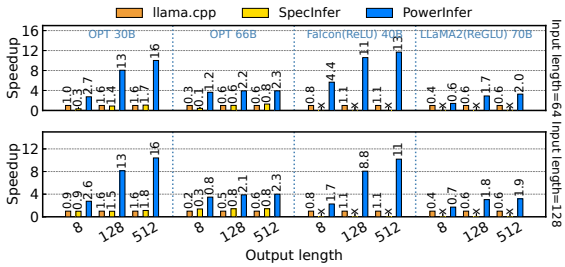


Figure 10. *Speedup of various models on PC-High in FP16 format.* The X axis and Y axis indicate the output length and speedup compared with llama.cpp. The number above each bar indicates the end-to-end generation speed (tokens/s). The up and down figures represent input length of 64 and 128.

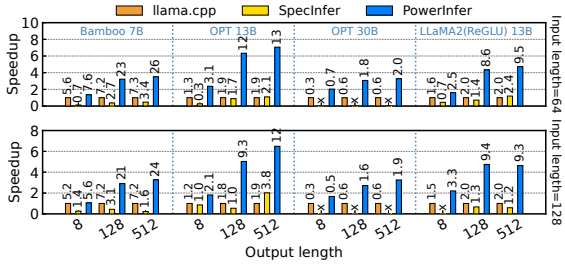
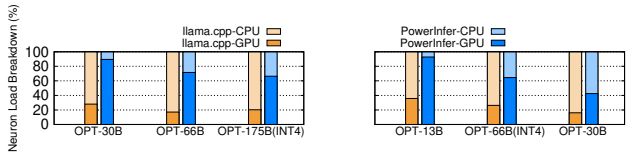


Figure 11. *Speedup of various models on PC-Low in FP16 format.* The up and down figures represent input length of 64 and 128, respectively.

with average speedups of 7.23× and 6.19×, and for Falcon-40B, up to 11.69× compared with llama.cpp. Although SpecInfer utilizes speculative decoding, since the large model size exceeds the GPU's capacity, there is still a significant amount of swapping during the verify phase, accounting for over 95% of the time. As a result, SpecInfer does not significantly outperform llama.cpp. The performance superiority of PowerInfer becomes more pronounced as the number of output tokens increases since the generation phase plays a more significant role in the overall inference time. In this phase, a small number of neurons are activated on both CPU and GPU, leading to fewer unnecessary computations compared to llama.cpp. For example, in the case of OPT-30B, only around 20% of neurons are activated for each token generated, with the majority processed on the GPU, a benefit of PowerInfer's neuron-aware inference engine.

Figure 11 shows that on a lower-end PC (PC-Low), PowerInfer still attains considerable performance enhancement over llama.cpp and SpecInfer, averaging a speedup of 4.71×, 5.97× and peaking at 7.06× and 7.47×. However, these improvements are smaller compared to those on a higher-end PC (PC-High), primarily due to the 11GB GPU memory limitation of PC-Low. This limitation affects the number of neurons that can be allocated to the GPU, particularly for models with around 30B parameters or more, leading to a greater dependence on the CPU for processing a larger number of activated neurons.



(a) PC-High **(b)** PC-Low

Figure 12. *Neuron load distribution on CPU and GPU during inference.* The *yellow* block refers to llama.cpp, and *blue* block refers to PowerInfer.

Table 4. End-to-end latency comparison for different models with 1.5K input length and 256 output length on PC-High.

LLMs	llama.cpp (ms)	PowerInfer (ms)	Speedup
LLaMA(ReGLU)-13B-FP16	49.91	14.38	3.47×
Falcon(ReLU)-40B-FP16	321.63	56.48	5.69×
LLaMA(ReGLU)-70B-FP16	92.76	37.17	2.50×

Figure 12 presents the distribution of neuron loads between the CPU and GPU for both PowerInfer and llama.cpp. Neuron loads refer to the proportion of activated neuron computations carried out by each processing unit. Notably, on PC-High, PowerInfer significantly increases the GPU's share of neuron load, from an average of 20% to 70%. This indicates that the GPU processes 70% of activated neurons. However, in cases where the model's memory requirements far exceed the GPU's capacity, such as running a 60GB model on an 11GB 2080Ti GPU, the GPU's neuron load is reduced to 42%. This decrease is due to the GPU's limited memory, which is insufficient to host all hot-activated neurons, necessitating that the CPU compute a portion of these neurons.

Inference with different input length. In scenarios involving long input prompts with extremely short output lengths (less than 8 tokens), which are less common [35], PowerInfer demonstrates limited performance gains, ranging from 1.07× (Figure 11) to 4× (Figure 10). In such situations, the prompt phase, where a substantial number of tokens are processed simultaneously, becomes a crucial factor in determining inference speed. This results in each token activating a unique set of neurons, substantially diminishing activation sparsity. As a consequence, the CPU becomes the primary bottleneck in the inference process, tasked with processing a considerable number of cold-activated neurons but constrained by its computational capabilities.

For relatively long input sequences, PowerInfer still achieves a 3.47× to 5.69× speedup, as shown in Table 4. This is because when the input sequence is lengthy, all neurons are activated collectively by the input tokens. In this case, PowerInfer switches to dense GPU computation during the prefill stage, resulting in prefill latency comparable to llama.cpp. During the generation phase, where only one token is processed at a time, sparsity emerges in each inference step. Here, PowerInfer leverages its hybrid engine to achieve significant acceleration.

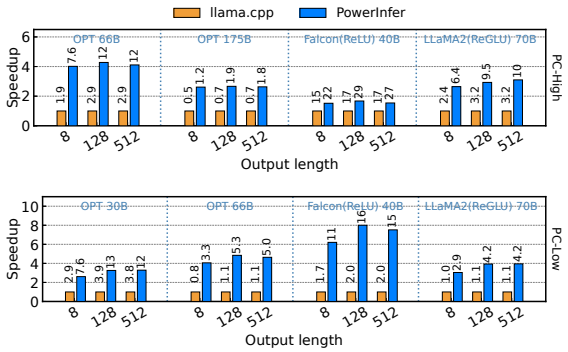


Figure 13. *Speedup of different models on PC-High and PC-Low in INT4 format.* The upper row of the figure presents performance on PC-High, while the lower row details those on PC-Low.

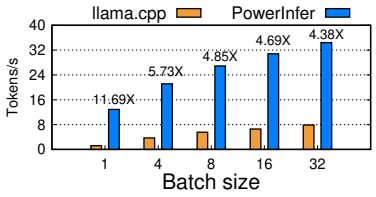


Figure 14. *Batch inference speedup of Falcon-40B on PC-High.* The X axis indicates the request batch size, the Y axis represents the end-to-end token generation speed (tokens/s). The number above each bar shows the speedup compared with llama.cpp.

Inference with Quantization. Figure 13 illustrates that PowerInfer effectively supports LLMs that are compressed using INT4 quantization. We fail to run SpecInfer with INT4 quantization due to the lack of support. On a high-end PC (PC-High), PowerInfer delivers responses at an average speed of 13.20 tokens/s, reaching a peak of 29.08 tokens/s. The average speedup achieved compared with llama.cpp is 2.89×, with a maximum of 4.28×. On a lower-end setup (PC-Low), the average speedup is 5.01×, peaking at 8.00×. The reduction in memory requirements due to quantization enables PowerInfer to more efficiently manage larger models. For instance, in our experiment with the OPT-175B model on PC-High, PowerInfer nearly reaches two tokens per second, surpassing llama.cpp by a factor of 2.66×.

Batching Inference. We also evaluate the end-to-end inference performance of PowerInfer with different batch sizes, as shown in Figure 14. PowerInfer demonstrates a significant advantage when the batch size is smaller than 32, achieving an average 6.08× improvement in performance compared with llama.cpp. As the batch size increases, the speed-up ratio offered by PowerInfer decreases. This reduction is attributed to the diminished sparsity of model joint activations. However, even with the batch size set to 32, PowerInfer still maintains a considerable speedup, achieving a 4.38× speedup.

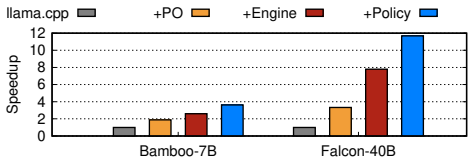


Figure 15. *Performance breakdown for each component of PowerInfer. The Falcon-40B is running on PC-High and Bamboo-7B is running on PC-Low.*

8.3 Ablation Studies

8.3.1 Performance Breakdown Figure 15 breaks down the contributions of each PowerInfer component to the overall performance speedup. Using a step-by-step integration method, we progressively incorporate PowerInfer features into llama.cpp. First, we add PowerInfer's predictors and neuron-aware operators into llama.cpp (labeled "+PO"), enabling computation of only activated neurons on both GPU and CPU. Yet, +PO still adheres to layer-wise computation, where each layer is processed entirely by either GPU or CPU.

Building on +PO, we introduce PowerInfer's hybrid inference engine (denoted "+Engine"), which allows neuron-aware operators to process neurons within the same layer simultaneously on both GPU and CPU. +Engine uses a naive neuron partitioning policy that assigns neurons randomly to the GPU. The final step involves integrating our optimized policy ("+Policy"), formulated by the offline solver as described in §6, into the +Engine setup, showcasing the full capabilities of PowerInfer.

The initial integration of +PO into llama.cpp yields performance boosts of 1.87× and 3.32× for Bamboo-7B and Falcon-40B, respectively, primarily by reducing unnecessary inactive neurons. +Engine further escalates these gains to 2.60× and 7.80×, thanks to precise neuron placement and intra-layer calculations that significantly increase the GPU's computational share. Finally, incorporating +Policy results in improvements of 3.62× and 11.69×. The enhancement achieved by our policy lies in its ability to finely balance the intra-layer communication overhead. The naive partitioning policy in +Engine overlooks the hotness of neurons and the GPU-CPU intra-layer communication, often offsetting the benefits of assigning high-frequency activation neurons to the GPU. Conversely, our policy in PowerInfer more adeptly balances processing loads and communication costs between the CPU and GPU.

8.3.2 Generation Latency Analysis In this section, we investigate the robustness of PowerInfer's speedup across different tasks and inter-token generation latency distritbution. We sample from the most representative tasks on the Hugging Face community, including STEM, coding, roleplaying and information extraction. We measure the inter-token generation latency distribution across different tasks. Table 5 presents the evaluation results of Bamboo-7B on PC-Low.

Table 5. Generation latency (ms) distribution on various Tasks.

		STEM	HumanEval	Roleplay	Table-Extraction
Bamboo-7B on PC-Low	Avg.	38.05	36.71	37.08	37.42
	P95	40.75	40.48	40.61	40.74
	P99	42.33	42.37	43.87	42.93

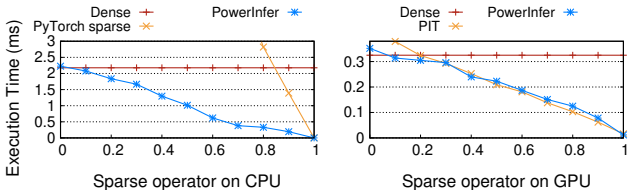


Figure 16. *Comparing Neuron-aware operator with different sparse operators on PC-Low.* The X axis indicates the sparsity level, the Y axis represents the execution time (ms).

PowerInfer exhibits a highly consistent inter-token generation latency across different tasks, with only a 10% difference between the P95 latency and the average. It is due to PowerInfer's policy that places the general hot neurons on GPU. This fluctuation is caused by the sparsity variation of different tokens from 80% to 86%. When encountering tokens with lower sparsity, the computation introduces 10% more latency compared with the average.

8.3.3 Neuron-aware Operator Performance This section evaluates the performance of PowerInfer's sparse operators on both CPU and GPU across various sparsity levels. We benchmark PowerInfer against leading sparse libraries: for CPU benchmarks, we use PyTorch sparse, the state-of-the-art sparse kernels within PyTorch, as our baseline. In GPU, PowerInfer is compared with PIT [63]. Given that the sparsity in LLMs is typically based on neuron granularity, our experiments are specifically designed to evaluate sparse matrices of this nature. We focus on sparse matrix-vector multiplication using a [4096, 4096] × [4096, 1] configuration, a common setup in local LLM inference [7]. To adjust sparsity, we introduce zero values to matrix rows.

Figure 16 shows that PowerInfer's operator achieves nearly linear acceleration with increasing sparsity levels, a stark contrast to dense matrix computations. On the CPU, traditional sparse operators do not outperform dense computation until sparsity surpasses 87%. However, PowerInfer's CPU operator outperforms dense matrix multiplication even at sparsity levels below 10%. For the GPU, PowerInfer matches PIT in performance. Its primary advantage, however, is its unified CPU-GPU framework. This design allows for flexible execution of sparse operators on both processing units, unlike PIT, which is optimized solely for GPU-based sparse matrix multiplication and does not support hybrid CPU-GPU environments.

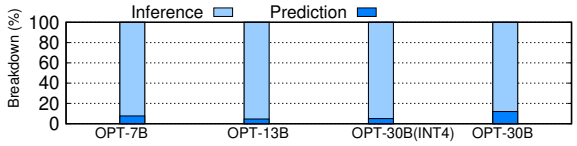


Figure 17. *End-to-end prediction overhead of PowerInfer on PC-Low.* The Y-axis displays the percentage breakdown between predictor overhead and LLM inference time.

Table 6. Predictor parameter sizes and ratios for various LLMs. The predictor ratio represents the percentage of predictor parameters relative to the original model parameters.

Model	OPT-13B	OPT-66B	Falcon(ReLU)-40B	LLaMA(ReGLU)-70B
Predictor-params	0.88B	3.23B	3.63B	5.66B
Predictor-ratio(%)	6.71%	4.88%	8.68%	8.08%

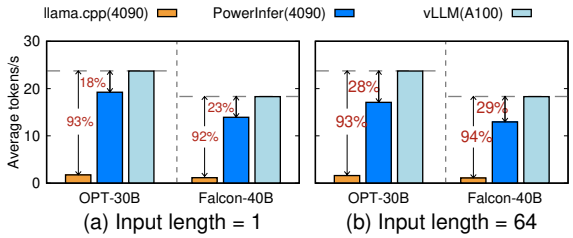


Figure 18. *Generation speed of NVIDIA RTX 4090 compared with single A100.* The X axis represents various models, while the Y axis represents generation speed (tokens/s) under various inference framework. The percentages within the arrows represent the slowdown relative to vLLM on the A100.

8.3.4 Predictor Overhead The execution time of the online predictors for different models is also measured, as depicted in Figure 17. On average, the execution of predictors constitutes less than 10% of the total inference time in PowerInfer. The execution time of predictors correlates directly with the predictor sizes, which, as shown in Table 6, comprise only 7.09% of the model weights. The efficiency of these predictors stems from adaptive construction methods that minimize both size and computational load. Furthermore, PowerInfer integrates these predictors into its solver for neuron placement decisions, preferentially allocating them to GPUs. This strategy exploits the parallel processing capabilities of GPUs, further reducing the runtime overhead associated with prediction.

8.3.5 Performance Comparison with A100 In our study, we analyze the extent to which PowerInfer reduces the performance gap between a consumer-grade GPU and its top-tier server-grade counterpart. Therefore, we evaluate the generation speed of PowerInfer, deployed on PC-High, in comparison to the performance of llama.cpp and vLLM [23] executed on a single 80GB NVIDIA A100 GPU. We chose the OPT-30B and Falcon-40B models for comparison, considering their exact memory requirements matching precisely with the capacity of the A100 GPU. Our evaluation used input lengths of 1 and 64 to measure pure generation speed and conversational interactions, respectively.

Table 7. Comparison of LLM accuracy between PowerInfer-optimized models (termed as "model-PowerInfer") and their original counterparts. Arc-Challenge [11] is a dataset for evaluating AI systems' comprehension and reasoning in natural language. MMLU [21] benchmarks model performance across various domains. PIQA [5] and Winogrande [41] assess common sense reasoning and understanding of physical interactions in LLMs.

	PIQA	Winogrande	Arc-Challenge	MMLU	GSM8K	Average
OPT-7B	75.78%	65.19%	30.63%	24.95%	1.90%	39.69%
OPT-7B-PowerInfer	75.67%	65.51%	30.63%	24.73%	1.82%	39.67%
OPT-13B	76.01%	64.96%	32.94%	25.02%	2.12%	40.21%
OPT-13B-PowerInfer	76.28%	65.98%	33.19%	24.76%	2.20%	40.48%
LLaMA(ReGLU)-13B	76.44%	70.09%	36.52%	50.21%	25.40%	51.73%
LLaMA(ReGLU)-13B-PowerInfer	74.06%	69.93%	36.60%	49.47%	23.90%	50.79%
Falcon-40B	81.23%	75.45%	50.68%	51.78%	21.99%	56.23%
Falcon-40B-PowerInfer	81.01%	75.92%	50.68%	51.68%	20.45%	55.95%
LLaMA(ReGLU))-70B	82.01%	75.93%	52.39%	62.30%	62.30%	66.99%
LLAMA(ReGLU)-70B-PowerInfer	82.05%	75.53%	51.45%	61.90%	61.90%	66.57%

Table 8. Generation speed (tokens/s) comparison for SwiGLU and ReLU-based LLMs.

Setting	Model	PowerInfer	llama.cpp	Speedup
PC-High	Yi(SwiGLU)-34B	1.7	1.0	1.7×
PC-Low	LLaMA(SwiGLU)-2-13B	3.1	2.1	1.5×
PC-High	OPT-30B	12.0	1.1	10.9×

Figure 18a demonstrates that PowerInfer significantly narrows the performance gap between the NVIDIA 4090 and A100 in generation tasks with input length 1. On PC-High, llama.cpp lags behind vLLM on the A100 by 93% and 92% for OPT-30B and Falcon-40B, respectively, but PowerInfer reduces this to 18% and 23%. Figure 18b shows that despite reduced cumulative sparsity in the prompt phase, PowerInfer still reduces the performance gap to 28% and 29%. The remaining disparity mainly stems from the CPU's considerable computational load, which has become a bottleneck.

8.4 SiLU-based LLM Performance

While PowerInfer demonstrates remarkable speedups on ReLU-based models due to their high activation sparsity, it also shows effectiveness in accelerating SiLU-based models. Our evaluation of SiLU-based LLMs, as presented in Table 8, reveals that PowerInfer achieves a speedup of 1.47× to 1.7×. This performance gain, albeit less pronounced than for ReLU-based counterparts (like OPT-30B in Table 8), underscores the efficacy of PowerInfer's sparse computation mechanisms across different activation functions. The comparatively lower speedup can be attributed to the reduced sparsity in SiLU-based models, where a larger proportion of neurons remain active during inference. Consequently, CPU computation emerges as a potential bottleneck in these scenarios. The speedup observed in SiLU models indicates that the effectiveness of PowerInfer's acceleration is correlated with the extent of the model's sparsity.

Table 9. Performance comparison between SLM (Qwen-1.5-4B) and PowerInfer (Bamboo-7B).

Model	TBT(ms)	Average(%)	MMLU(%)	GSM8K(%)	ARC-C(%)
Qwen1.5-4B	10.83	52.23	55.26	53.9	47.53
Bamboo-7B-PowerInfer	11.85	65.65	62.26	70.54	64.16
Bamboo-7B-dense	18.54	65.49	62.46	70.28	63.74

8.5 LLM Accuracy

Since PowerInfer selectively omits neurons predicted to be inactive, we investigated whether this approach affects the inference accuracy of LLMs, focusing on models using ReLU or related activation functions. Table 7 compares the accuracy of models from the OPT, Falcon (ReLU), and LLaMA2 (ReGLU) families, both with and without differentiating activated/inactivated neurons, across a variety of representative downstream tasks. The results show that PowerInfer causes negligible loss in inference accuracy, regardless of the model size or type of task, consistent with previous research findings [28]. Although the predictors in each Transformer layer maintain an accuracy rate above 95%, they may occasionally miss some active neurons. As a result, there are minor fluctuations in LLM accuracy, leading to slight fluctuations in performance on specific downstream tasks.

We further analyze the nature and impact of mispredicted neurons on inference accuracy. Our investigation reveals that neurons mispredicted by the predictor typically have minimal influence on the layer's output. This characteristic makes these neurons more susceptible to misprediction, as their activation or non-activation has little effect on the overall result. To quantify this impact, we conducted an analysis on the OPT-7B model. By comparing the cosine similarity between outputs from original dense computations and our predictor-based computations, we found that mispredictions only affect results by approximately 0.4%. This minimal difference further supports our observation that PowerInfer maintains the accuracy of the original model, despite the occasional misprediction of neurons.

Comparison with Smaller Language Models. To better understand PowerInfer's performance gains, we compare it with a common acceleration approach: using smaller language models (SLMs). However, the SLM approach often comes at the cost of reduced model accuracy. To fully understand the benefits of PowerInfer, we compare its performance using a larger model to that of a state-of-the-art SLM, evaluating both speed and accuracy. Table 9 shows that the Bamboo-7B model outperforms the Qwen-1.5-4B (a state-of-the-art 4B model) across various tasks, and PowerInfer (Bamboo-7B-PowerInfer) maintains the original dense model's accuracy while achieving 4B-level decoding speed.

9 Related Work

LLM Weight Sparsity: Model pruning [19, 20, 31] reduces parameters by setting weights to zero, as seen in SparseGPT [14] and Wanda [49], achieving 50% sparsity. SparTA [64] leverages both sparse tensor and SIMT cores by dividing sparse matrices. Flash-LLM [54] introduces a "Load-as-Sparse and Compute-as-Dense" approach for tensor core SpMM. However, these methods, orthogonal to LLMs' intrinsic sparse activations, usually incur accuracy losses and wall-clock model acceleration challenges [34]. In contrast, PowerInfer uses natural sparse activations to maintain performance and efficiency.

LLM Attention Sparsity: PowerInfer leverages the sparse activation characteristics of MLP layers. It's worth noting that attention blocks also exhibit sparsity, which primarily stems from two sources: attention heads, where only some need computation for the current token [28], and sparsity within heads, where unimportant KV cache is pruned or offloaded to alleviate memory-bound bottleneck of LLMs serving, as in H2o [60] and InfiniGen [25]. The sparse activations used by PowerInfer are orthogonal to attention sparsity and can further reduce inference latency.

Speculative LLM Inference: Speculative inference [7, 8, 16, 52] can also be leveraged to serve models exceeding GPU memory, which uses a smaller model to pre-decode tokens, later validated by the main model in a batch. SpecInfer [33] effectively reduces the number of LLM decoding steps. While separate from our focus, integrating speculative inference into PowerInfer could further boost LLM inference speed.

LLM-Specific Serving Optimizations: There are many ML serving systems [18] for general ML models. The prominence of Transformers has led to specialized serving systems [9, 12, 22, 42, 65]. Orca [57] introduces iteration-level scheduling. vLLM [23] implements PagedAttention for token storage in varied GPU memory addresses, overcoming KV cache's continuous storage limit. These methods do not address the challenge of deploying models on PCs where the entire model cannot fit within the GPU memory.

10 Conclusion

PowerInfer is a fast inference system optimized for LLMs that exploits the locality property in LLM inference. It utilizes adaptive predictors and neuron-aware operators for neuron activation and computational sparsity. PowerInfer achieves up to 11.69× faster LLM inference compared to systems like llama.cpp, without compromising accuracy.

11 Acknowledgments

We sincerely thank our shepherd Shivaram Venkataraman and anonymous reviewers for their insightful suggestions. We are deeply grateful to Rong Chen and Yubin Xia for their comprehensive and constructive feedback, which greatly enhanced the quality of this paper. This work was partially supported by National Key R&D Program of China (2023YFB4503702) and NSFC (No. 62372287 and 61925206). Zeyu Mi (yzmizeyu@sjtu.edu.cn) is the corresponding author.

References

[1] Bo Adler, Niket Agarwal, Ashwath Aithal, Dong H Anh, Pallab Bhattacharya, Annika Brundyn, Jared Casper, Bryan Catanzaro, Sharon Clay, Jonathan Cohen, et al. 2024. Nemotron-4 340B Technical Report. *arXiv preprint arXiv:2406.11704* (2024).

[2] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[3] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. Falcon-40B: an open large language model with state-of-the-art performance. (2023).

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[5] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 7432–7439.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, and Tri Dao. 2023. Medusa: Simple Framework for Accelerating LLM Generation with Multiple Decoding Heads. https://github.com/FasterDecoding/Medusa.

[8] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. arXiv:2302.01318 [cs.CL]

[9] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547* (2023).

[10] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/1966445.1966473

[11] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv:1803.05457v1* (2018).

[12] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[13] Wikimedia Foundation. 2024. Wikimedia Downloads. https://dumps.wikimedia.org.

[14] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. *arXiv preprint arXiv:2301.00774* (2023).

[15] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323* (2022).

[16] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2023. Breaking the Sequential Dependency of LLM Inference Using Lookahead Decoding. https://lmsys.org/blog/2023-11-21-lookahead-decoding/

[17] Georgi Gerganov. 2023. ggerganov/llama.cpp: Port of Facebook's LLaMA model in C/C++. https://github.com/ggerganov/llama.cpp.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[19] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[21] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[22] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. 2023. SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. arXiv:2312.16733 [cs.DC]

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[24] Je-Yong Lee, Donghyun Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. 2024. CATS: Contextually-Aware Thresholding for Sparsity in Large Language Models. *arXiv preprint arXiv:2404.08763* (2024).

[25] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[26] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. 2022. The Lazy Neuron Phenomenon: On Emergence of Activation Sparsity in Transformers. In *The Eleventh International Conference on Learning Representations*.

[27] Peiyu Liu, Zikang Liu, Ze-Feng Gao, Dawei Gao, Wayne Xin Zhao, Yaliang Li, Bolin Ding, and Ji-Rong Wen. 2023. Do Emergent Abilities Exist in Quantized Large Language Models: An Empirical Study. arXiv:2307.08072 [cs.CL]

[28] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 22137–22176. https://proceedings.mlr.press/v202/liu23am.html

[29] Hanjia Lyu, Song Jiang, Hanqing Zeng, Qifan Wang, Si Zhang, Ren Chen, Chris Leung, Jiajie Tang, Yinglong Xia, and Jiebo Luo. 2023. LLM-Rec: Personalized Recommendation via Prompting Large Language Models. arXiv:2307.15780 [cs.CL]

[30] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. *arXiv preprint arXiv:2305.11627* (2023).

[31] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. In *Advances in Neural Information Processing Systems*.

[32] Iván Martínez Toro, Daniel Gallego Vico, and Pablo Orgaz. 2023. *PrivateGPT*. https://github.com/imartinez/privateGPT

[33] 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. 2023. SpecInfer: Accelerating Generative Large Language Model Serving with Speculative Inference and Token Tree Verification. arXiv:2305.09781 [cs.CL]

[34] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. arXiv:2310.04564 [cs.LG]

[35] MohamedRashad. 2023. https://huggingface.co/datasets/MohamedRashad/ChatGPT-prompts.

[36] NVIDIA. 2021. Unified Memory Programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[37] NVIDIA. 2023. cuSPARSE: Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs. https://developer.nvidia.com/cusparse.

[38] OpenAI. 2023. https://openai.com/blog/chatgpt.

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[40] Prajit Ramachandran, Barret Zoph, and Quoc V Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).

[41] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Commun. ACM* 64, 9 (2021), 99–106.

[42] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2023. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. arXiv:2311.03285 [cs.LG]

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. (2023).

[44] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, et al. 2024. ProSparse: Introducing and Enhancing Intrinsic Activation Sparsity within Large Language Models. *arXiv preprint arXiv:2402.13516* (2024).

[45] Yixin Song, Haotong Xie, Zeyu Mi, Li Ma, and Haibo Chen. 2024. Bamboo: Harmonizing Sparsity and Performance in Large Language Models. (2024).

[46] Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. 2024. Turbo Sparse: Achieving LLM SOTA Performance with Minimal Activated Parameters. *arXiv preprint arXiv:2406.05955* (2024).

[47] SparseLLM. 2024. ReLULLaMA-70B. https://huggingface.co/SparseLLM/ReluLLaMA-70B.

[48] Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. 2024. LLM Pruning and Distillation in Practice: The Minitron Approach. *arXiv preprint arXiv:2408.11796* (2024).

[49] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2023. A Simple and Effective Pruning Approach for Large Language Models. *arXiv preprint arXiv:2306.11695* (2023).

[50] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[52] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 233–248. https://doi.org/10.1145/3552326.3587438

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[54] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. arXiv:2309.10285 [cs.DC]

[55] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.

[56] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. *arXiv preprint arXiv:2406.06282* (2024).

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[58] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[59] Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. 2022. MoEfication: Transformer Feed-forward Layers are Mixtures of Experts. In *Findings of ACL 2022*.

[60] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[61] Zhengyan Zhang, Yixin Song, Guanghui Yu, Xu Han, Yankai Lin, Chaojun Xiao, Chenyang Song, Zhiyuan Liu, Zeyu Mi, and Maosong Sun. 2024. ReLU^2 Wins: Discovering Efficient Activation Functions for Sparse LLMs. arXiv:2402.03804 [cs.LG]

[62] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

[63] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. 2023. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 331–347. https://doi.org/10.1145/3600006.3613139

[64] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 213–232. https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin

[65] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 489–504. https://www.usenix.org/conference/atc22/presentation/zhou-zhe