SparseInfer: Training-free Prediction of Activation Sparsity for Fast LLM Inference

Jiho Shin
University of Seoul
Seoul, South Korea
sjh010529@uos.ac.kr

Hoeseok Yang*
Santa Clara University
Santa Clara, CA, USA
hoeseok.yang@scu.edu

Youngmin Yi*
Sogang University
Seoul, South Korea
ymyi@sogang.ac.kr

***Abstract*—Leveraging sparsity is crucial for optimizing large language model (LLM) inference; however, modern LLMs employing SiLU as their activation function exhibit minimal activation sparsity. Recent research has proposed replacing SiLU with ReLU to induce significant activation sparsity and showed no downstream task accuracy degradation through fine-tuning. However, taking full advantage of it required training a predictor to estimate this sparsity. In this paper, we introduce *SparseInfer*, a simple, light-weight, and training-free predictor for activation sparsity of ReLU-fied LLMs, in which activation sparsity is predicted by comparing only the sign bits of inputs and weights. To compensate for possible prediction inaccuracy, an adaptive tuning of the predictor's conservativeness is enabled, which can also serve as a control knob for optimizing LLM inference. The proposed method achieves approximately 21% faster inference speed over the state-of-the-art, with negligible accuracy loss of within 1%p.**

***Index Terms*—Large language models (LLMs), Activation sparsity, Training-free**

I. Introduction

Following the remarkable success of Generative Pre-trained Transformer (GPT) models [1] based on the Transformer architecture [2], Large Language Models (LLMs) are being actively employed across a wide range of fields. The inference of LLMs based on the GPT architecture first performs an initial forward pass, also known as *prefill*, followed by an autoregressive *decoding* process that generates tokens sequentially, one at a time. Due to this sequential nature, memory accesses occur frequently in the decoding stage. This can lead to I/O bottlenecks, which in turn limits the overall computational performance. Well-known optimization methods such as pruning [3] or quantization [4] have been applied to LLMs to reduce this memory access overhead but can result in an accuracy loss.

Another approach for optimizing LLM inference is to take advantage of activation sparsity. Unlike pruning techniques, which either remove zero-valued weights from the model or set some weights to zero, activation sparsity exploits the zeros that occur dynamically during runtime, depending on the input. A major source of this activation sparsity comes from activation layers. While Sigmoid Linear Unit (SiLU) [5] or Gaussian Error Linear Unit (GELU) [6] is popularly used as activation layers in the Multi-Layer Perceptron (MLP) of recent LLMs, they exhibit significantly lower sparsity compared to Rectified Linear Unit (ReLU). However, recent studies [7], [8] have shown that replacing SiLU or GELU with the more sparse ReLU can maintain nearly the same level of accuracy with minor fine-tuning.

Although such *ReLU-fied* LLMs have successfully increased the degree of activation sparsity, it is still not feasible to know in advance which inner-product operations will ultimately result in zeros after passing through the activation layer, thus can be skipped, at design-time. Therefore, the LLM inference needs to be assisted by a method that can effectively predict this during runtime. A recent study [9] proposed a method that involves training an additional neural network, which consists of a couple of fully connected layers, per each MLP block to predict activation sparsity. However, this approach has the drawback of requiring a separate predictor to be trained. In addition, since the predictor needs to reside in memory throughout inference, the impact of this additional neural network on memory footprint, as well as its computational overhead, cannot be ignored.

In this paper, we propose *SparseInfer*, which accelerates General Matrix-Vector Multiplication (GEMV) operations in the MLPs of ReLU-fied LLMs by leveraging activation sparsity. Unlike previous methods, the proposed approach efficiently predicts the sparsity pattern without requiring additional predictor training, making it easily applicable to legacy pre-trained LLMs. Furthermore, this technique is portable across various types of hardware, including both commercial off-the-shelf (COTS) processors such as GPUs and CPUs, as well as customized hardware accelerators, as it only requires a simple lookup of partial information about the elements involved in the inner product of GEMV operations. We validate our implementation using a mobile-grade GPU.

The proposed activation sparsity prediction method is based on approximating the sign of the input to ReLU without calculating the exact result. Specifically, the sign of each element in the two vectors involved in the inner product is compared with the corresponding element's sign in the other vector using XOR, predicting the sign of each element-wise multiplication result. Then, assuming the absolute values of the elements in the two vectors follow a normal distribution, we compare the number of negative and positive results before accumulation to predict whether the input to ReLU will be positive or negative.

*Youngmin Yi and Hoeseok Yang are corresponding authors.

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00498)

978-3-9826741-0-0/DATE25/© 2025 EDAA

If more element-wise products are predicted to be negative, the final result is likely to be negative. Since a negative value would result in zero after passing through ReLU, this is assumed to contribute to activation sparsity.

Despite a high level of accuracy, this predictor is inevitably subject to a certain degree of inaccuracy. Experimental results indicate that this inaccuracy is relatively more noticeable in the early layers. To overcome this, two additional measures have been implemented. The first measure is to consider additional sparsity identified during MLP execution, which the predictor initially missed, as *actual* sparsity (as opposed to *predicted* sparsity). Secondly, we extract a tunable parameter to adaptively control the level of conservativeness in predictions. With this, we can explore the trade-off between inference speed and potential accuracy degradation. Also, this can serve as an important control knob for design space exploration (DSE) in optimizing LLM inference, given the target platform, the model, and the downstream task.

SparseInfer was implemented on an open source LLM inference engine, *llama.cpp* [10]. The proposed XOR-based activation sparsity prediction method was implemented as a CUDA kernel to enable parallel processing on the GPU. Based on this prediction, the MLP operations were optimized to exploit activation sparsity. To further enhance the performance, kernel fusion was applied to the MLP and activation layers, reducing memory access overhead. Experiments on NVIDIA Jetson Orin AGX 64GB show that the overhead of the predictor was reduced by a factor of four compared to the existing activation sparsity prediction method, and the inference speed per token with the Llama2 13B and 7B models improved by approximately 21% and 19% compared to the state-of-the-art, while maintaining the accuracy loss within 1%p.

II. Related Work

Activation Sparsity-induced LLM: Mirzadeh et al. [7] proposed a technique called ReLUfication which replaces activation functions in LLM, such as SiLU, with ReLU. They showed ReLUfication can induce significant activation sparsity while accuracy loss is negligible after fine-tuning. *ProSparse* [8] significantly induced activation sparsity by progressively applying sparsity regularization after Relufication. It also sets a positive value instead of zero for a ReLU threshold and achieves larger sparsity using a technique called *FATReLU* [11]. It induced a 22%p higher sparsity compared to *ReluLLaMa* [12], an open-source ReLU-based LLM fine-tuned from *Llama2*, achieving 66% sparsity. Despite a slight performance drop in some common sense benchmarks, the resultant model generally maintained comparable performance on average and even performed slightly better in certain cases, achieving up to a $4.65\times$ speedup in inference time with *PowerInfer* [13]. A similar approach, *TurboSparse* [14], was proposed for Mixtral models.

Another approach is to employ the SiLU activation function as is but set an input distribution-based threshold to induce sparsity [8], [15]. While *CATS* [8] only sparsified the Feed-Forward Network (FFN) layers, *TEAL* [15] applied the method to the Attention layer as well. Compared to ReLUfication, this approach does not require fine-tuning but shows lower sparsity at the comparable performance; *CATS* reported 15% speedup in inference.

Activation Sparsity Prediction: *DEJAVU* [9] proposed utilizing activation sparsity in the Attention and MLP blocks of a Transformer layer, leveraging a sparsity predictor composed of two fully connected layers. While this approach is argued to be less computationally intensive than a nearest-neighbor-based predictor, it still incurs considerable overhead. Furthermore, the predictor necessitates additional training. Also, it is not easy to determine the training hyper-parameters, such as the dimension size of predictors in a layer, which may impact the prediction performance. It reduced the inference latency for *OPT-175B* model by $2\times$ compared to *FasterTransformer* on A100 GPUs. *PowerInfer* [13] adopted *DEJAVU*'s activation sparsity predictor and achieved a large speedup of up to $11.6\times$ over *llama.cpp* when the GPU memory could not load the entire model. The breakdown of speedup showed that the speedup due to exploiting sparsity was about twofold.

Novelty of the proposed method: Most existing methods for utilizing LLM activation sparsity, if not all, rely on predictors that require training. This approach has a fundamental drawback in that retraining is necessary whenever the model changes or its quantization is modified. The proposed method, in contrast, predicts sparsity dynamically and approximately without the need for training, distinguishing it from previous methods. The proposed predictor not only eliminates the need for additional training but also reduces memory and computational overhead. Although similar prediction methods have been explored in Convolutional Neural Networks (CNNs), such as in SeerNet [16], it has not been applied to LLMs to the best of our knowledge.

III. Background: Gate-Based MLP

In this section, we illustrate the operation of the MLP block, which we aim to optimize using activation sparsity and where the majority of time is spent during the LLM decoding process.[1] Due to the ease of presentation and space constraints, we focus our explanation on the Gate-based MLP used in the Llama model [17]. However, it is important to note that the proposed technique is not limited to this specific model and can be applied, without loss of generality, to any models that can be ReLU-fied. As demonstrated in Mirzadeh et al. [7], ReLUfication can be effectively applied to different types of MLP blocks found in models such as *Falcon* [18] and *OPT* [19].

The gate-based MLP block consists of three linear layers (W_{gate}, W_{up}, and $W_{down} \in \mathbb{R}^{d\times k}$ in which $k > d$) with d being the model dimension and one activation layer ($\sigma(\cdot)$). It can be formulated as follows:

$$MLP(X) = (\sigma(X \cdot W_{gate}) \odot (X \cdot W_{up})) \cdot W_{down}^{T}, \quad (1)$$

where $\odot$ denotes an element-wise multiplication between two vectors. Specifically, the input vector $X \in \mathbb{R}^{1\times d}$ goes through 4 steps.

[1] In our profiling of the inference of *Llama2-13B* using *llama.cpp* on NVIDIA Jetson Orin AGX 64GB, we observed that during decoding, self-attention and MLP computations accounted for 38% and 62%, respectively.

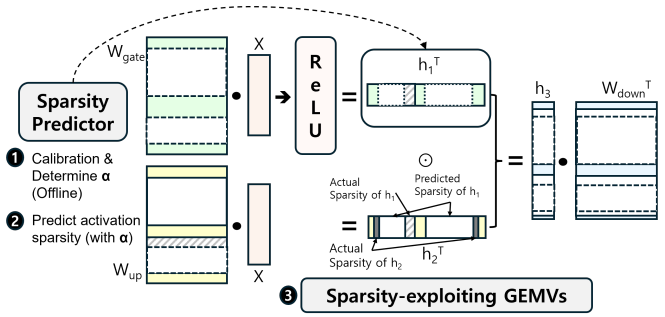

Fig. 1: Overview of SparseInfer approach

1) (gate computation) The input X is first used to compute the gate values. This is done by applying a linear transformation indicated by W_{gate} followed by the activation function, thus formulated as $h1 = \sigma(X \cdot W_{gate})$.
2) (input processing) Simultaneously, the input vector X undergoes a separate linear transformation by W_{up}, resulting in $h2 = X \cdot W_{up}$.
3) (gate application) The gate values computed in step 1 are then applied element-wise to the processed input from step 2. This is achieved through element-wise multiplication ($h3 = h1 \odot h2$).
4) (output generation) Finally, the gated vector is passed through another linear transformation using the weight matrix W_{down}, resulting in the final output as $h3 \cdot W_{down}^T$.

IV. Proposed Method: *SparseInfer*

As explained in Section III, ReLU is applied to the result of Matrix-Vector multiplication of W_{gate} and X, producing $h1$ vector. Note that the activation sparsity of each layer is approximately 90% [8], and the corresponding rows of W_{gate} matrix to the zero elements in $h1$ need not be loaded and computed. The same applies to W_{up} matrix in step 2 as $h2$ is used for element-wise vector-vector multiplication with $h1$. Furthermore, the activation sparsity of the $h3$ vector which should be the same or higher than that of the $h1$ vector is exploited to skip the corresponding rows of the transposed W_{down} matrix in step 3. These significantly reduce the inference time.

Fig. 1 illustrates the overview of how the proposed approach, *SparseInfer*, utilizes the activation sparsity. ① First, we set the coefficients for the predictor to efficiently predict the sparsity, which is done through an offline effort. ② Next, we predict the sparsity in $h1$ by only looking up partial information from W_{gate} and X. ③ Finally, based on the predicted results from step 2, we skip the memory loading and computations of steps 1-4 for the positions in $h1$ that are predicted to be zero.

Note that, step 2, which is found in *Llama* but not in *Falcon* and *OPT* models, can be executed in parallel with step 1, e.g., using Concurrent Kernel Execution (CKE) in CUDA. Alternatively, executing steps 1 and 2 sequentially offers two advantages. First, the steps can be implemented in a single kernel (i.e., kernel fusion), eliminating the store and the load of input vector X. A more significant advantage of executing steps 1 and 2 sequentially is the ability to compensate for the predictor's inaccuracy. The predictor relies solely on sign information, which can lead to inevitable inaccuracies. Therefore, to minimize accuracy loss, the prediction of activation sparsity, performed before step 1, is conducted conservatively, which will be explained in Section IV-A. After step 1, in addition to the predicted sparsity, the exact, actual sparsity that was identified can be utilized. In other words, when steps 1 and 2 are executed sequentially, it becomes possible to account for the sparsity missed by the initial predictor in subsequent steps. This activation sparsity to which we coined the term *actual sparsity*, in addition to the predicted sparsity, can be exploited in the subsequent steps in the MLP block. The actual sparsity of $h2$, as well as that of $h1$, is exploited in step 4.

A. Training-free Sparsity Prediction

SparseInfer is designed to approximately predict activation sparsity without additional training. Since we target ReLU-fied LLMs, it is not necessary to predict the *exact* value of the inputs to the ReLU. Instead, we only need to determine whether it is positive or negative. To this end, we devise a technique that minimizes memory access by using only the sign information of W_{gate} and X. This allows us to approximate the sign information of the result of $X \cdot W_{gate}$ in step 1 (gate computation) will be positive or negative.

This approximation is relatively accurate because it relies on the assumption that the elements of X and row vector of W_{gate} ($W_{gate,i}$) follow independent Gaussian distributions with a mean equal to zero [15], [20]. It is known that the products of two variables following independent Gaussian distributions with zero means have a symmetric distribution with a mean equal to zero [21]. In this distribution, the ratio of positive to negative values is generally similar, meaning that the probability of individual products being positive or negative is almost equal. As a result, if there are more positive or negative values, the cumulative result is highly likely to have the sign of the more frequent values. This allows us to predict whether the result of $X \cdot W_{gate}$ will be positive or negative with relatively high accuracy, even without knowing the exact values.

Fig. 2 illustrates the distribution of values in a row vector of W_{gate} ($W_{gate,i}$) and X obtained from the *ProSparse-Llama2-13B* model [8] when performing 8-shot inference on a subset of the *GSM8K* dataset [22]. Due to space constraints, intermediate layers exhibiting similar distributions have been omitted.

Both X and W_{gate} exhibit distributions that closely resemble Gaussian distributions, with an approximately equal ratio of positive and negative values. Moreover, their products $Y = X \odot W_{gate,i}$ follow a symmetric distribution with a mean approaching zero, as previously hypothesized. This confirms the validity of our initial assumption regarding the distributions of X and W_{gate}, and consequently, the accuracy of our approximation method. In the early layers, the distribution of X values is dominated by near-zero negative and near-zero positive values, leading to a narrow distribution heavily concentrated around zero, rather than following a typical normal distribution. This characteristic is believed to contribute to prediction inaccuracies.

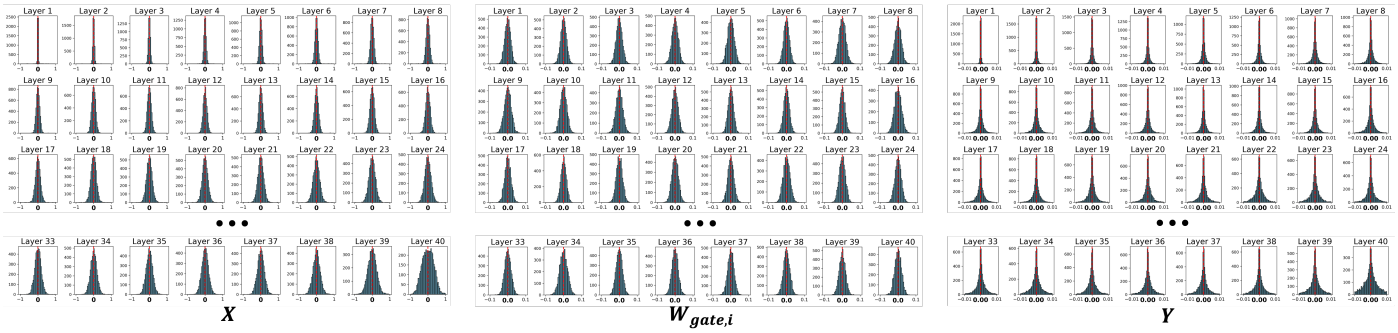

Fig. 2: Distribution of X, $W_{gate,i}$, and $Y = X \odot W_{gate,i}$ values obtained from *ProSparse-Llama2-13B* during 8-shot inference on the *GSM8K* dataset.

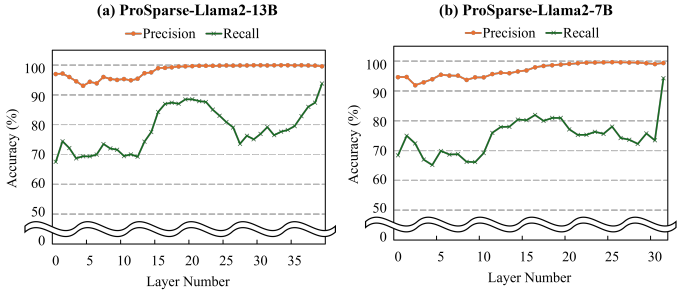

Fig. 3: Precision/Recall of *SparseInfer*'s activation sparsity prediction method for each decoding layer of *ProSparse-Llama2-7B* and *ProSparse-Llama2-13B*.

The proposed prediction method estimates whether the result of the inner product between each row of W_{gate}, denoted as $W_{gate,i}$, and X will become zero after passing through the ReLU function. By looking up only the most significant bits (MSBs) of $W_{gate,i}$ and X, we can predict the sign of each element-wise product through an XOR operation. Specifically, if the result of the XOR operation is 0, the product is predicted to be positive; if it is 1, the product is predicted to be negative. Based on this, we can determine the number of positive elements (N_{pos}) and negative elements (N_{neg}) in the resulting vector from the inner product of $W_{gate,i}$ and X. If $N_{pos} < N_{neg}$, it is likely that the accumulated result will be negative, and the corresponding element in $h1$ will be likely to be zero after passing through the ReLU function.

Since this prediction is not based on exact values but on empirical observations and statistical hypotheses, it comes with inevitable inaccuracies. To compensate for the potential inaccuracies that this may introduce into the inference process, a tunable coefficient α can be added to the equation, allowing for the adjustment of the aggressiveness or conservativeness of the activation sparsity prediction. Specifically, the above inequality is refined as follows:

$$\alpha \cdot N_{pos} < N_{neg}. \tag{2}$$

In this case, $\alpha > 1.0$ results in a more conservative prediction of activation sparsity, whereas $\alpha < 1.0$ leads to a more aggressive prediction, increasing the degree of activation sparsity.

Fig. 3 shows the precision and recall of the proposed activation sparsity prediction for each layer in the *ProSparse-Llama2-7B* and *ProSparse-Llama2-13B* models. Here, precision represents the proportion of elements in $h1$ that the predictor classified as sparse and are indeed sparse (i.e., the likelihood that an element predicted to be sparse is actually sparse). On the other hand, recall indicates the proportion of truly sparse elements in $h1$ that the predictor successfully identified as sparse, relative to the total number of sparse elements (i.e., how well the predictor captures the overall sparsity). Overall, both models exhibited high precision values, reaching over 99% but a relatively lower precision was observed in the early layers compared to the later layers.

To compensate for the relatively lower accuracy in the early layers, the tunable parameter α can be applied differently across the layers. For the early layers where precision is relatively lower, where precision is relatively lower, α is set slightly above 1.0, while for the stabilized remaining layers, α is set to 1.0. The optimal value for α can be easily calibrated through test runs as the model changes, but empirically, setting α to a small value such as 1.01–1.03 for the early layers has shown good performance as will be shown in experiments.

The proposed prediction method is training-free, making it robust to various standard quantization methods, e.g., INT8 or FP16. As long as the sign bit, i.e., MSB, can be extracted, it can be applied directly, regardless of the quantization scheme used. In contrast, training-based prediction schemes such as *DEJAVU* [9] would require re-training of the predictor to maintain its prediction accuracy whenever different quantization levels or methods are employed.

It should be emphasized that, as the proposed prediction scheme is training-free and tunable, it is adequate for the DSE of LLM inference which can find the optimal (time, accuracy) configurations for the given target platform and model.

B. CUDA Implementation of SparseInfer

In this subsection, we present the implementation of the proposed method on a GPU using CUDA.

1) Pre-fetching and Packing Sign-Bit Information: Extracting the sign bits by loading W_{gate} at each layer during the decoding phase would result in unnecessary memory accesses. Instead, we pack the sign bits of 32 consecutive elements in W_{gate} into a 32-bit integer variable when the model is loaded into memory. This process incurs a one-time overhead, as it is executed only once before *llama.cpp* constructs the call graph.

For the input vector X, which is determined dynamically, such pre-fetching is not feasible. Therefore, we pack the sign bits of its elements into 32-bit integer variables during the decoding phase, using a method similar to that used for W_{gate}.

2) Sparsity Prediction: CUDA employs a two-level hierarchical parallel structure, where a single kernel execution (grid) is composed of multiple thread blocks, and each thread block contains multiple threads. In our implementation for sparsity prediction in $X \cdot W_{gate}$, each thread block is organized as a 2-dimensional 32×16 threads; 32 threads, or one warp, is assigned a row and 16 warps exist in a thread block. Consequently, the grid consists of $k/16$ thread blocks, where k is the total number of rows in a matrix. Note that, a warp in CUDA is an independent scheduling unit, and the synchronization, such as sum reduction, among the threads within a warp can be performed efficiently.

Listing 1 shows the CUDA kernel for activation sparsity prediction. Each thread retrieves the 32-bit packed sign information from its corresponding position within $sign\_W\_gate$ and $sign\_X$, then performs an XOR operation to predict the sign of the product result (lines 6-8). Subsequently, the CUDA built-in function $\_\_popc()$ is utilized to count the number of ones in the XOR result, representing the number of elements predicted as negative, and this count is accumulated in $count$. The sign information of a single row of W_{gate} is packed into a 32-bit format, requiring a total of $ncols = d/32$ XOR operations. As explained above, a single warp is assigned a row and it needs to repeat lines 6-8 $ncols/WARP\_SIZE$ times to complete the counting (line 5). Note that assigning more than one warp for the same row could reduce the number of iterations along the $ncols$, but it could require sum reduction among multiple warps. Finally, it predicts an element to be sparse, setting the $skip$ flag for the corresponding row to true if the number of negative products ($count$) is larger than that of the positive products ($ncols*32-count$). Note that, depending on a layer, a scale factor, α, can be applied differently.

Listing 1: Acitvation Sparsity Prediction in CUDA

```
1  __global__ void predict_activation_sparsity(const
       int32_t *sign_W_gate, const int32_t *sign_X, int
       *skip, int nrows, int ncols, int alpha) {
2      const int64_t row = (int64_t)blockIdx.x *
           blockDim.y + threadIdx.y;
3      const int tid = threadIdx.x;
4      int count = 0;
5      for (int i = 0; i < ncols / WARP_SIZE ; i++) {
6          int col = i * WARP_SIZE + tid;
7          int idx = row * ncols + col;
8          int32_t Xor = sign_W[idx] ^ sign_X[col];
9          count += __popc(Xor); }
10     count = warp_reduce_sum(count);
11     if (tid == 0) {
12         if (count * 100 - (ncols * 32 - count) *
               alpha > 0) skip[row] = 0;
13         else skip[row] = 1; }
14 }
```

3) Sparse GEMV: The sparse GEMV kernel takes as input the $skip$ flag for each row, which was predicted by Listing 1. The thread block size remains consistent with the Sparsity Predictor, set to 32×16. As before, each warp is assigned to handle one row and iteratively performs inner-product computations over $ncols$, accumulating the results. When the $skip$ flag is on, the warp immediately returns 0 without any computation. Since activation sparsity is determined at the row level and a sparse row results in the entire warp being skipped, there is no need for additional load balancing. Note that, the sparse GEMV performed in the steps other than step 1 uses adjusted skip flags, which is the union of the predicted sparsity or previous flags *and* the actual sparsity that can be obtained after step 1.

4) Kernel Fusion: Steps 1, 2, and 3 of the MLP operation described in Equation (1) may optionally be fused into a single CUDA kernel. Since steps 1 and 2 share the input vector X, this kernel fusion enhances memory reusability. Additionally, step 3 uses the outputs of steps 1 and 2, making it efficient to include it in the same kernel. The activation sparsity prediction is performed before step 1, generating a k-length vector where sparse elements are marked with 1 and non-sparse elements with 0. Through this kernel fusion, memory access is limited to one load for X and one write for the result of step 3, h_3. If each step were implemented as a separate kernel, as in the original llama.cpp implementation, X would need to be loaded twice, while h_1 and h_2 would each require one load and one write, and h_3 would require one write. This illustrates the efficiency of the fused kernel approach.

On the other hand, step 4 is implemented as a separate kernel. As W_{down} is transposed to skip a row, instead of a column, the sum reduction of products in an inner-product in a GEMV cannot be implemented as accumulation but should be performed through $atomicAdd()$ among different warps, which are assigned different rows. If the flag indicates that the row is sparse, the warp skips the product computation and $atomicAdd()$, simply by returning. Note that, W_{down} is transposed and stored as W_{down}^T during model loading.

V. Experimental Results

The hardware platform used in the experiments is Jetson Orin 64GB [23], installed with Ubuntu 22.04, JetPack 6.0 DP, CUDA 12.2. *SparseInfer* was extended from *llama.cpp*, implementing the activation sparsity prediction and exploitation explained in Section IV. Thus, the baseline for the comparison is *llama.cpp* [10], and we also compared *SparseInfer* with *PowerInfer* [13], which is the state-of-the-art inference engine that supports activation sparsity. It is also based on *llama.cpp* and employs the activation sparsity predictor of *DEJAVU* [9]. Assuming on-device environments, all the model weights were pre-loaded on GPU. *PowerInfer* performed inference on GPU, not with a CPU-GPU hybrid execution. Note that Orin SoC integrates Cortex CPU and Ampere GPU in a single chip and they share the same LPDDR DRAM. The models used are *ProSparse-Llama2-13B* and *7B*, which are ReLUfied Llama models. Note that *SparseInfer* can perform inference with any ReLUfied models.

TABLE I: Number of Operations for Prediction and MLP Block

	Prediction	MLP Block
llama.cpp (dense)	0	2.123×10^8
PowerInfer	1.940×10^7	1.699×10^7
SparseInfer (proposed)	2.211×10^6	1.699×10^7

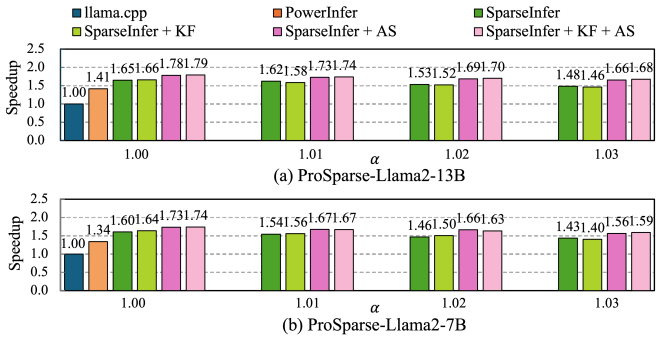

Fig. 4: Comparison of end-to-end latency for token generation.

TABLE II: Performance comparison for *ProSparse-Llama2-13B*

Method	α	GSM8K	BBH	Average
Baseline	-	30.71	44.80	37.76
SparseInfer	1.00	27.75 (-2.96)	42.91 (-1.89)	35.33 (-2.43)
	1.01	28.73 (-1.98)	43.57 (-1.23)	36.15 (-1.61)
	1.02	29.64 (-1.07)	44.43 (-0.37)	37.04 (-0.72)
	1.03	30.63 (-0.08)	44.34 (-0.46)	37.49 (-0.27)

TABLE III: Performance comparison for *ProSparse-Llama2-7B*

Method	α	GSM8K	BBH	Average
Baseline	-	13.42	35.80	24.61
SparseInfer	1.00	8.95 (-4.47)	27.37 (-8.43)	18.16 (-6.45)
	1.01	10.77 (-2.65)	33.70 (-2.10)	22.24 (-2.37)
	1.02	11.30 (-2.12)	35.51 (-0.29)	23.41 (-1.20)
	1.03	12.96 (-0.46)	35.60 (-0.20)	24.28 (-0.33)

A. Prediction Overhead

1) Latency: To compare the prediction overhead of *SparseInfer* with *PowerInfer*, the predictor latency with the first 10 samples in *GSM8K* dataset was measured on these inference engines. *PowerInfer* performs inference with *PowerInfer-ProSparse-Llama2-13B* model where the weights of the *DEJAVU* predictor trained with the rank size of 1024 is also included. Table I shows the number of operations in the *PowerInfer* predictor is even larger than the actual computation amount for MLP block. *SparseInfer* predictor has about an order of magnitude less number of operations than *PowerInfer*, and note that the operation is 32-bit bitwise XOR in *SparseInfer* while it is FP16 multiplication in *PowerInfer*. As a result, the predictor latency of *SparseInfe*r for one token generation per layer is only 70 usec on average, which is 3.66× faster than that of *PowerInfer*. The speedup is lower than the operation reduction ratio because FP16 multiplications in *PowerInfer* run on Tensor cores, while XORs run on CUDA cores.

2) Memory Usage: The hidden state dimension d in *ProSparse-Llama2-13B* is 5120 and W_{gate} size in an MLP block is $d \times k$ = 5120×13824, and the model has 40 MLP blocks. *PowerInfer* requires (5120×1024 + 1024×13824)×2(byte)×40 = 1480 MB, when the rank in *DEJAVU* predictor employed in *PowerInfer* is 1024. In contrast, *SparseInfer* only requires the sign bit of each element, which is packed in 32-bit variables ,leading to 13824×160×4(byte)×40 = 337.5 MB, which consumes 4.38× less memory usage.

B. End-to-End Speedup

We measured the inference time of the first 10 samples in the *GSM8K* dataset and compared the average end-to-end latency of token generation of *SparseInfer* with *PowerInfer*, as well as *llama.cpp*, varying α from 1.00 to 1.03. Figure 4 shows the comparison results for *ProSparse-Llama2-13B* and *7B*, where fourfold *SparseInfer* results were drawn depending on whether the kernel fusion was applied and whether the actual sparsity was exploited.

With α being 1.00, *SparseInfer* achieved 1.79× and 1.74× speedups over *llama.cpp* for 13B and 7B models, respectively, outperforming *PowerInfer* by 1.27× and 1.30×. As α gets larger, the speedups become slightly lower, due to the increased non-zero elements. With α being 1.03 for the 7B model, it leads to 1.59× over llama.cpp and 1.19× over *PowerInfer*. We applied $\alpha > 1.0$ only to the first 20 layers for both the 13B and 7B models.

The gain from the kernel fusion (+*KF*) turned out to be insignificant due to the relatively small size of the input vector compared to the large cache size. Utilizing *actual sparsity* (+*AS*), in addition to the predicted activation sparsity, contributes significantly to the speedup, especially when α gets larger as it conservatively predicts the sparsity.

C. Accuracy

To evaluate the effect of the proposed approach on model performance, lm-harness [24], a widely used benchmark suite was executed. As *SparseInfer* utilizes activation sparsity only in the decoding phase, not in the prefill phase, log-likelihood-based benchmark evaluation is inadequate. We evaluated the model performance of *ProSparse-Llama2-7B*, *13B* with *GSM8K*, *BBH* [25] benchmarks that generate the answer. Since a set of widely used benchmarks do not yet directly support *llama.cpp*, we ported the *SparseInfer* predictor and sparse GEMV to the PyTorch-based Transformer inference engine. Additionally, we were unable to measure accuracy because *PowerInfer* could not be ported to the Transformer engine.

The performance results are shown in Table II and Table III. Without scaling for the earlier layers, the performance drop occurs, especially in *ProSparse-Llama2-7B*. However, it becomes negligible within 1%p when α = 1.03 is applied. Note that random selection with the 90% activation sparsity, instead of the prediction, resulted in 0% accuracy.

VI. Conclusion

This paper proposes an efficient yet sufficiently accurate prediction method for activation sparsity in LLMs, by which the memory and computation in the inference can be significantly reduced. Based on the observation of the input activation and weight distribution, the prediction method requires only sign-bit information of the weight matrix and the input vector, making the approach training-free and DSE-friendly. *SparseInfer*, a framework for fast LLM inference, employing the proposed predictor, allows designers to explore the different trade-offs of LLM inference, given the target architecture and the model. The experimental results show that *SparseInfer* outperforms *PowerInfer*, the state-of-the-art, by 21% while maintaining the minimal accuracy loss on Jetson Orin AGX.

REFERENCES

[1] A. Radford, "Improving language understanding by generative pre-training," 2018.

[2] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[3] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21 702–21 720, 2023.

[4] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 168–27 183, 2022.

[5] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[6] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[7] I. Mirzadeh, K. Alizadeh, S. Mehta, C. C. D. Mundo, O. Tuzel, G. Samei, M. Rastegari, and M. Farajtabar, "Relu strikes back: Exploiting activation sparsity in large language models," *arXiv preprint arXiv:2310.04564*, 2023.

[8] C. Song, X. Han, Z. Zhang, S. Hu, X. Shi, K. Li, C. Chen, Z. Liu, G. Li, T. Yang, and M. Sun, "Prosparse: Introducing and enhancing intrinsic activation sparsity within large language models," *arXiv preprint arXiv:2402.13516*, 2024.

[9] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re *et al.*, "Deja vu: Contextual sparsity for efficient llms at inference time," in *International Conference on Machine Learning*. PMLR, 2023, pp. 22 137–22 176.

[10] G. Gerganov, "llama.cpp," https://github.com/ggerganov/llama.cpp, 2023, accessed: September 18th, 2024.

[11] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5533–5543.

[12] ReluLLaMa, "ReluLLaMa," https://huggingface.co/SparseLLM/ReluLLaMA-7B.

[13] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," *arXiv preprint arXiv:2312.12456*, 2023.

[14] Y. Song, H. Xie, Z. Zhang, B. Wen, L. Ma, Z. Mi, and H. Chen, "Turbo sparse: Achieving llm sota performance with minimal activated parameters," *arXiv preprint arXiv:2406.05955*, 2024.

[15] J. Liu, P. Ponnusamy, T. Cai, H. Guo, Y. Kim, and B. Athiwaratkun, "Training-free activation sparsity in large language models," *arXiv preprint arXiv:2408.14690*, 2024.

[16] S. Cao, L. Ma, W. Xiao, C. Zhang, Y. Liu, L. Zhang, L. Nie, and Z. Yang, "Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 216–11 225.

[17] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[18] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Alhammadi, M. Daniele, D. Heslow, J. Launay, Q. Malartic *et al.*, "The falcon series of language models: Towards open frontier models," *Hugging Face repository*, 2023.

[19] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models, 2022," *URL https://arxiv. org/abs/2205.01068*, vol. 3, pp. 19–0, 2023.

[20] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: efficient finetuning of quantized llms," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[21] C. C. Craig, "On the frequency function of xy," *Annals of Mathematical Statistics*, vol. 7, pp. 1–15, 1936. [Online]. Available: https://api.semanticscholar.org/CorpusID:121925566

[22] C. Karl, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[23] L. S. Karumbunathan, "Nvidia jetson agx orin series," 2022.

[24] EleutherAI, "lm-evaluation-harness," https://github.com/EleutherAI/lm-evaluation-harness, 2024, accessed: September 23th, 2024.

[25] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou *et al.*, "Challenging big-bench tasks and whether chain-of-thought can solve them," *arXiv preprint arXiv:2210.09261*, 2022.