LM-Offload: Performance Model-Guided Generative Inference of Large Language Models with Parallelism Control

Jianbo Wu
University of California, Merced
Merced, USA
jwu323@ucmerced.edu

Jie Ren
William & Mary
Williamsburg, USA
jren03@wm.edu

Shuangyan Yang
University of California, Merced
Merced, USA
syang127@ucmerced.edu

Konstantinos Parasyris
Lawrence Livermore National Laboratory
Livermore, USA
parasyris1@llnl.gov

Giorgis Georgakoudis
Lawrence Livermore National Laboratory
Livermore, USA
georgakoudis1@llnl.gov

Ignacio Laguna
Lawrence Livermore National Laboratory
Livermore, USA
lagunaperalt1@llnl.gov

Dong Li
University of California, Merced
Merced, USA
dli35@ucmerced.edu

ABSTRACT

Large language models (LLMs) have achieved remarkable success in various natural language processing tasks. However, LLM inference is highly computational and memory-intensive, creating extreme deployment challenges. Tensor offloading, combined with tensor quantization and asynchronous task execution, provides a feasible and cost-effective solution by utilizing host memory to enable large-scale LLM inference with a limited number of GPUs. However, existing approaches struggle to fully utilize all available computational and memory resources due to a lack of consideration for (1) whether to use quantization and how to apply it effectively, and (2) managing thread-level parallelism within and across tasks. As a result, these approaches provide suboptimal solutions. In this paper, we introduce LM-Offload, a framework that addresses the above challenges by leveraging performance modeling and parallelism control. Experimental results demonstrate that LM-Offload outperforms FlexGen and ZeRO-Inference, two state-of-the-art systems for LLM inference, by up to 2.95× (2.34× on average) and 2.88× (1.57× on average) respectively in inference throughput.

1 INTRODUCTION

Deploying large language models (LLMs) is memory consuming. For example, the inference of LLaMA-2 model with 70-billion parameters [22] using FP16 consumes about 168GB GPU memory. Such a large memory consumption easily goes beyond the memory capacity of common GPUs (e.g., NVIDIA H100 with 80GB). Although using multiple GPUs can solve the memory capacity problem, that significantly increases the production cost.

To address the memory capacity problem, tensor offloading [5–9, 12–16, 18–20, 24, 28] provides a feasible and cost-effective solution. This technique uses CPU memory and hard drive disk (HDD) as an extension of GPU memory. Tensor offloading moves the contents of unused tensors to the CPU or HDD and moves them back to the GPU memory when needed. As such, tensor offloading reduces the used GPU memory and enables large LLM deployment in single GPU. To avoid frequent tensor migration between GPU and CPU, the recent tensor offloading solutions offload attention computation [20, 21] and the Adam optimizer [14, 16] to CPU, besides offloading tensors.

Tensor offloading faces challenges on I/O cost. Tensor movement between GPU and CPU must go through the interconnect between CPU and GPU, which can be 64 GB/s (using 16-lane PCIe5), an order of magnitude slower than HBM on GPU. In addition, the tensor size to offload can be large. For example, the size of KV cache (a type of tensors) reaches 157GB, and the size of weights (another type of tensors) reaches 55GB when running OPT-30B (a large LLM with 30-billion parameters) with the prompt length of 64, generation length of 128, and GPU batch size of 64.

To reduce the I/O cost, tensor quantization [2, 20, 21] and asynchronous task execution [14, 16, 20] are two common techniques. With tensor quantization, the elements in a tensor use lower precision (e.g., 4 bits or 16 bits) instead of 32 bits, which reduces the I/O volume. Asynchronous task execution uses parallel tasks to concurrently execute different operations, such as I/O operations and attentation computation operations. Despite the benefits of both of these approaches, we identify the following challenges:

Challenges. The benefit and overhead of quantization are highly correlated with whether to offload computation, the frequency of update of the tensor, and the size of the tensor. With the computation offloading, some tensors permanently reside in CPU memory, and hence avoid the necessity of quantization for I/O cost reduction. However, the computation offloading to CPU can slow down the computation because of the limited thread-level parallelism offered by CPU. Hence using quantization faces a tradeoff between the computation slowdown and reduction of I/O cost. Furthermore, different tensors have different update frequencies, thus causing different quantization overhead. For example, during LLM inference, weights are initialized, quantized once and then read only afterward. In contrast, the KV cache is updated throughout token generation and quantized at each transformer layer. The frequency of tensor update affects the benefit and overhead of quantization. As a result, different tensors should use different quantization strategies.

Second, asynchronous task execution causes challenges on parallelism management. In particular, a single task (e.g., the attention computation) can involve a set of operators without dependency, and multiple independent tasks can run in parallel. Each operator (e.g., AddmmBackward) can also use multi-threading. We observe that using different thread-level parallelism for each operator and across operators causes large performance variance (up to 40%). Using the default thread-level parallelism in PyTorch does not lead to the best performance. Hence we must have a mechanism to determine thread-level parallelism.

Because of the above challenges, it is difficult to navigate the space of whether and how quantization should be applied while also selecting how to dedicate the available CPU parallelism to the underlying operations. Without a systematic approach to explore these options, the space is infeasible to navigate due to the combinatorial nature of the problem.

Solutions. To address the above challenges, we introduce LM-Offload. To address the first challenge, LM-Offload introduces analytical performance models to quantify the benefit and overhead of quantization of weights and KV cache (the largest tensors to offload). The performance models consider the impact of offloading attention computation and how tensors are updated, hence providing an efficient method to quickly compare the end-to-end performance of various offloading and quantization strategies.

To address the second challenge, we introduce an algorithm to effectively explore thread-level parallelisms and find the optimal parallelism configurations for all tasks. In addition, we bundle small operators when throttling parallelism to avoid cache thrashing.

We summarize the major contributions as follows.

- We reveal the performance tradeoff when applying quantization to major tensors in LLMs, and reveal the impact of thread-level parallelism on the LLM inference performance. This is unprecedented.
- We build the performance models and an algorithm to address the major challenges faced by quantization and asynchronous task execution (two major techniques used for tensor offloading to reduce I/O cost).
- Evaluating with multiple LLMs (including LLaMA and OPT) with up to 66-billion parameters, we show that LM-Offload outperforms FlexGen [20] and ZeRO-Inference [2] from Microsoft DeepSpeed, two state-of-the-art tensor-offloading frameworks, by up to 2.95× (2.34× on average) and 2.88× (1.57× on average) respectively.

2 BACKGROUND

2.1 Large Language Model

Basic mechanisms. Transformer-based LLMs are typically composed of a set of transformer layers, each featuring a self-attention layer and a feedforward neural network known as the Multi-Layer Perceptron (MLP). The attention layer measures importance or relevance of each token in the input sequence to the current focused token. Figure 1 depicts computation and memory usage of self-attention. The self-attention begins by computing query, key, and value vectors (Q, K, and V) for each input token (e.g., a word in an input sentence). These vectors are then used to update the key-value cache (KV cache), a caching mechanism that saves computation by storing the previously calculated key and value vectors. The KV cache is updated by concatenating the current token's key and value vectors with the existing ones, resulting in a *linear* increase in the cache size and a *quadratic* increase in computation scales as the sequence length grows. After updating the KV cache, the self-attention calculates an attention score for each pair of input tokens using the formula $QK^T/\sqrt{d_k}$, where d_k is the dimension of K. A `softmax` function is then applied, as shown in Figure 1.

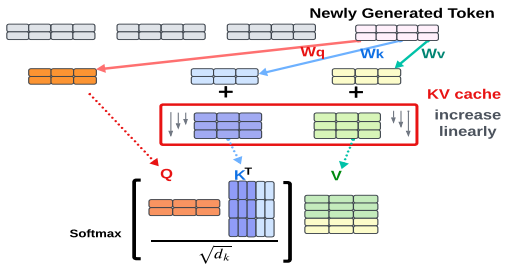

Figure 1: Self-attention calculation in a transformer layer. The size of KV cache increases linearly with the number of tokens being generated.

After the attention layer, there is a MLP layer where there are two linear transformations with a non-linear activation function in between. MLP introduces non-linearity to the attention output, enhancing the model's capacity to learn complex relationships.

LLM inference contains two phases: the *prefill* phase, which processes the input tokens present at the start of the inference, and the *decode* phase, which autoregressively generates output tokens. During the prefill phase, all input tokens are processed in parallel, and the model initializes the KV cache by computing and storing the key and value vectors that capture the relationships between input tokens, which are later retrieved during the decode phase.

In the decode phase, the output tokens are generated one by one in a sequential manner. Specifically, the model utilizes information in the KV cache along with the previously generated token to generate the next output token. The KV cache stores key and value vectors for *all* generated tokens, enabling the model to leverage both precomputed context and real-time information to create coherent and contextually relevant outputs. By storing these vectors, the KV cache allows the model to focus on calculating the attention for the new token, reducing the computational complexity.

2.2 LLM inference with Tensor Offloading

Handling large memory footprint in LLM is challenging, as both LLM parameters and KV cache are stored in memory. To enable LLM inference under strict GPU memory capacity constraints, existing works such as ZeRO-Inference [2] and FlexGen [20] explore the tensor offloading technique which leverages CPU memory as an extension of GPU memory by offloading tensors that are not immediately required for computation from GPU memory to CPU memory. FlexGen uses a zig-zag block schedule to decide computation order of inference batches. With this schedule, multiple batches traverse the LLM layers together as a block, and the block

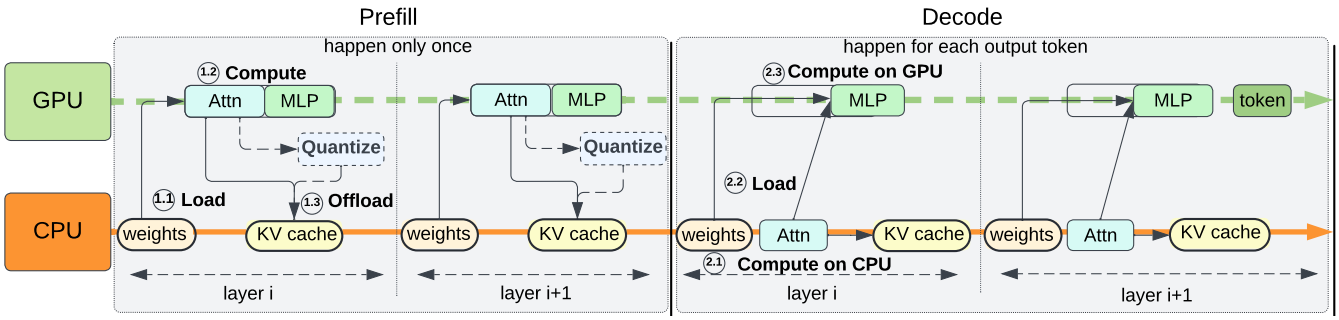

Figure 2: The workflow of LLM inference with FlexGen, which uses tensor offloading to save GPU memory.

size is equal to the inference batch size multiplied by the number of batches in the block. We depict FlexGen as follows and in Figure 2.

During the **prefill phase**, FlexGen (1.1) loads weights from CPU memory to GPU memory, and (1.2) performs attention and MLP computations on the GPU layer by layer. After each attention layer, FlexGen (1.3) offloads the KV cache to CPU memory, with an option to store the KV cache in either compressed or uncompressed format. The **decode phase** is an iterative process that generates output tokens sequentially. Within each iteration, FlexGen performs (2.1) attention computation and updates the KV cache on the CPU to avoid frequent tensor-transfers and enable handling long input/output sequence. It then (2.2) loads weights and activations (i.e., the output of attention computation) to the GPU and (2.3) performs MLP computation on the GPU to leverage its massive parallelism.

Policy search. FlexGen formulates the tensor offloading problem as an optimization problem. The objective is to maximize LLM inference throughput (tokens/sec) by determining the percentage of weights, KV cache, and activations on the GPU. FlexGen leverages linear programming to search for efficient patterns to store and access tensors. However, due to the lack of modeling quantization overhead and benefits, as well as inaccurately estimating the performance impact of asynchronous execution of computation and tensor movement, FlexGen provides an offloading policy that cannot fully utilize all available computational and memory resources in the system, leading to inferior performance.

3 PERFORMANCE MODELING FOR QUANTIZATION

The primary objective of this section is to investigate and quantify the impact of quantization techniques on LLM inference performance. We use the terms "quantization" and "compression" interchangeably in the following discussion.

3.1 Motivation

Evaluation setup. We use a single NVIDIA A100 GPU with 40GB global memory and 240GB CPU memory. Table 4 gives more hardware information. As a motivation study, we use OPT-30B (a transformer with 30 billion parameters). The prompt length (i.e., the input sequence length) is 64, the generation length is 128 (i.e., generating 128 tokens for each prompt), the inference batch size is 64, and the zig-zag block size is 640. We focus on offline inference and measure the throughput of LLM inference.

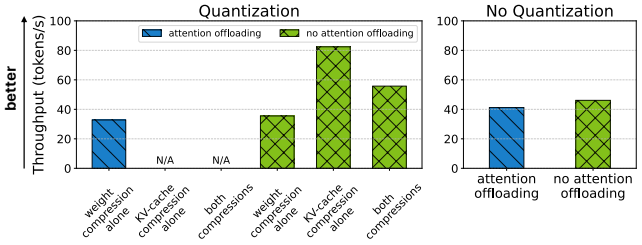

Figure 3: Performance comparison with various offloading and quantization strategies.

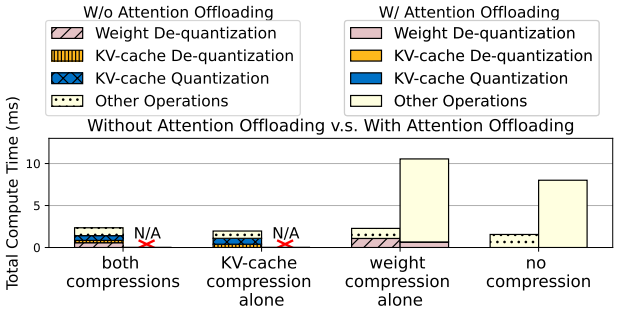

Figure 4: Performance breakdown to study quantization.

For the inference of OPT-30B, the total memory consumption is 214GB, among which the parameters take 55GB and the KV cache takes up to 157GB. Without tensor offloading, our evaluation platform cannot be used for model inference. We study multiple strategies using various offloading and quantization strategies.

Evaluation results. Figure 3 shows the results. Table 1 and Figure 4 have detailed analysis. We have two observations.

Observation 1. Attention offloading can impact the effectiveness of quantization.

In our evaluation, *with attention offloading*, using quantization always performs worse than without using quantization: without quantization, the throughput is 41 tokens/s, while with quantization, the best throughput is 32 tokens/s, which is 20% less. In contrast, *without attention offloading*, we have the opposite results: without quantization, the throughput is 46 tokens/s, while with quantization, the best throughput is 82 tokens/s, which is 78% more.

Table 1: I/O traffic for all layers for *one token generation* with and without offloading attention computation.

	Direction	I/O Traffic	
With attention offloading	From CPU to GPU	Weights	16.32 GB
		KV cache	0
		Activation	0.38 GB
	From GPU to CPU	Weights	None
		KV cache	0
		Activation	0.38 GB
Without attention offloading	From CPU to GPU	Weights	38.88 GB
		KV cache (old)	78.72 GB
		Activation	0.38 GB
	From GPU to CPU	Weights	0
		KV cache (new)	0.8 GB
		Activation	0.38 GB

Three reasons account for the above results. *First*, offloading attention computation reduces I/O traffic, providing fewer opportunities for quantization to improve performance. In particular, with attention offloading, the KV cache does not need to be transferred from the CPU to the GPU, which reduces I/O traffic by 78.72GB per token generated. Although the attention offloading scheme must upload the activation from CPU to GPU, that is much smaller than the KV cache (99.5% less). Table 1 depicts the difference in I/O traffic between attention offloading and no attention offloading.

Second, attention offloading allows more weights to be constantly resident on GPU, compared with no offloading of attention computation, which reduces the (de)quantization overhead paid for weights offloaded to CPU. Table 1 shows that the attention offloading saves 22.56 GB GPU memory per token generated, compared to no offloading of attention computation, which leads to a reduction of the (de)quantization overhead by 58%.

Third, offloading attention computation or not determines the necessity of KV cache (de)quantization. With attention offloading, KV cache is not needed to be sent to GPU and hence there is no need of (de)quantization. To support this conclusion, we break down the inference time to quantization, dequantization, and other operations (including attention and MLP). Figure 4 shows the results. With attention offloading, the (de)quantization overhead is zero.

Observation 2. Compressing different types of tensors can cause different impacts on inference throughput.

Figure 3 shows that without attention offloading, the inference throughput for the cases of applying quantization on weights alone, on KV cache alone, and on both weights and KV cache are 35 tokens/s, 82 tokens/s, and 55 tokens/s respectively. Applying quantization to the KV cache alone leads to the best performance.

There are two reasons that account for the above results. *First*, the compression of the weights and KV cache happens with different frequencies, which leads to different compression overhead. Weight compression happens during the weight initialization and hence happens only once. Since the weights are read-only, compression does not need to occur repeatedly. On the other hand, the KV cache is updated from one token to another, and therefore the compression occurs during each token generation.

Second, weights quantization and KV cache quantization have different overhead. During a token generation, the old KV cache from previous token generation is consumed, and then the new KV cache is generated and appended to the old KV cache. During a token generation, the new KV cache is compressed, and the old KV cache only needs to be decompressed. As the tokes are generated one by one, such (de)compression overhead continuously increases. In contrast, the weights only need to be de-compressed during the decode phase, and de-compression overhead is constant. As the (de)quantization overhead is related to the tensor size, the (de)quantization overhead for the KV cache changes across token generations.

Based on above observations, we conclude that the (de)quantization overhead is not trivial, and is related to attention offloading, tensor update frequency, and tensor size. We introduce performance modeling to predict the (de)quantization overhead such that we can determine whether applying quantization is beneficial in any scenario (i.e., attention offloading vs. no attention offloading) and which tensor (i.e., weights vs. KV cache) should be compressed.

Table 2: Notation for the performance modeling.

Notation	Description
bls	zig-zag block size
cpu_flops	CPU flops per second
cpu_freq	CPU frequency
cpu_mem_bdw	CPU memory bandwidth
gpu_flops	GPU flops per second for matrix multiplication
gpu_freq	GPU frequency
gpu_mem_bdw	GPU HBM bandwidth
h_1	hidden size
h_2	hidden size of the second MLP layer
l	number of layers in the LLMs
n	output sequence length
s	prompt sequence length
T	inference latency
T_{pf}	inference latency for one layer in prefill stage
T_{gen}	inference latency for one layer in decode stage
wc	percentage of weights on CPU

3.2 Performance Modeling

Table 2 list the notation used in performance modeling. In general, we want to minimize T/bls, where T is the end-to-end generative time and bls is the block size. In FlexGen, T is modeled in Equation 1

$$T = T_{init} + T_{pf} \cdot l + T_{gen} \cdot (n-1) \cdot l \tag{1}$$

where T_{init} is the cost of loading weights from hard drive to CPU memory (see (1.1) in Figure 2); T_{pf} and T_{gen} are the inference latency for one transformer layer during the prefill phase and decode phase respectively. Given a transformer model with l transformer layers and the output sequence length n, $T_{pf} \cdot l$ and $T_{gen} \cdot (n-1) \cdot l$ represent the prefill time and decode time respectively.

During token generation, FlexGen executes six tasks. Algorithm 1 depicts the six tasks in FlexGen. The six tasks include loading the next transformer layer's weights (`load_weight`), storing KV cache and activation of the previous batch (`store_cache` and `store_activation`), loading KV cache and activation of the next batch (`load_cache` and `load_activation`), and computing the current batch (`compute_layer`), as described in Lines 6, 8, 11, and 14 in Algorithm 1. The six tasks are performed for each token generation,

Algorithm 1 Six tasks during the decode phase.

```
1:  function GENERATION_LOOP_OVERLAP_MULTI_BATCH
2:      for i = 1 to generation_length do
3:          for j = 1 to num_layers do
4:              // Compute a block with multiple GPU batches
5:              for k = 1 to num_GPU_batches do
6:                  // Load the weight of the next layer
7:                  load_weight(i, j + 1, k)
8:                  // Store the cache and activation of the prev batch
9:                  store_activation(i, j, k − 1)
10:                 store_cache(i, j, k − 1)
11:                 // Load the cache and activation of the next batch
12:                 load_cache(i, j, k + 1)
13:                 load_activation(i, j, k + 1)
14:                 // Compute this batch
15:                 compute(i, j, k)
16:             end for
17:             // Synchronize all devices
18:             synchronize()
19:         end for
20:     end for
21: end function
```

at each LLM layer, and for each batch, which are manifested as a three-level nested loop in Algorithm 1. The six tasks are launched asynchronously and run in parallel. Hence, T_{gen} is modeled as follows.

$$T_{gen} = max(\texttt{load\_weight}, \texttt{load\_cache}, \texttt{load\_activation}, \texttt{store\_cache}, \texttt{store\_activation}, \texttt{compute}) \quad (2)$$

The six tasks have been modeled in FlexGen. In particular, among the six tasks, the load and store tasks are modeled by offloaded tensor size divided by the interconnect bandwidth. The offloaded tensor size is determined by the LLM model size, prompt length, generation length, and zig-zag block size. `compute` task includes attention and MLP. In FlexGen and LM-Offload, MLP occurs on GPU, and attention can happen on either CPU or GPU. Hence, the `compute` task in our performance models focuses on attention computation. The `compute` task time is in proportion to the generation length and zig-zag block size. Our modeling focuses on (de)quantization and how it is correlated with the six tasks.

Impacts on the six tasks. We include the (de)quantization overhead into the load and store tasks for weights and KV cache. We do not apply the (de)quantization to activation, because the activation size is small at the scale of KB and the load/store activation takes less than 1% of inference time, which is very small. The `compute` task is not impacted by the (de)quantization.

For the weights, the quantization occurs only once and its overhead is included into the initialization time, shown in Equation 3. After that, the weights only cause de-quantization overhead on GPU, shown in Equation 4.

$$T_{init} = T_{init} + quan_{pf\_wgt} \quad (3)$$

$$load\_weight = load\_weight + dequan\_wgt \quad (4)$$

The KV cache is pre-populated and quantized during the prefill phase, modeled in Equation 5. During the decode phase, in each token generation, the old KV cache (including the KV cache generated in the prefill phase and prior tokens generated) is uploaded to GPU memory, modeled in Equation 6. In each token generation, the newly generated KV cache is quantized and stored back to CPU memory, modeled in Equation 7.

$$T_{pf} = T_{pf} + quan_{pf\_cache} \quad (5)$$

$$load\_cache = load\_cache + dequan\_old\_cache \quad (6)$$

$$store\_cache = store\_cache + quan\_new\_cache \quad (7)$$

Quantization and attention offloading. We model the quantization overhead with and without attention offloading.

With attention offloading, Equations 3-4 remain the same for the weights. For the KV cache, `load_cache` = 0 and `store_cache` = 0.

Without attention offloading, the GPU memory space for loading weights and KV cache becomes smaller. This reduces the size of the largest transformer model that can be deployed on the GPU, compared with using attention offloading. To make a fair performance comparison between attention offloading and no attention offloading, we assume that both cases use the same transformer models. This means that without attention offloading, the load and store times become smaller, which are modeled in Equations 8 and 9. In the two equations, r is a ratio determined by the linear programming in FlexGen. The linear programming decides how to share the limited GPU space between tensors to maximize inference throughput. r is the ratio of offloaded weights (without attention offloading) to the original weights (with attention offloading).

$$load_{\_weights/cache} = load_{\_weights/cache} \times r \quad (8)$$

$$store_{\_weights/cache} = store_{\_weights/cache} \times r \quad (9)$$

Quantization modeling. FlexGen uses a group-wise quantization algorithm shown in Algorithm 2. Given tensors, this algorithm pads the tensor into a specific shape so that SIMD processing can be efficient (Lines 5-6). The algorithm then partitions all elements into groups along a certain dimension (the group size is defined by the user), and finds the minimum and maximum values within each group (Lines 9-10). Then, the algorithm normalizes each element to the range $(0, 2^b - 1)$ using Equation 10 (Lines 12 and 14) where b is the number of bits after quantization. The results are then packed and re-shaped to be aligned with the b-bit size (Lines 16 and 18).

$$x_{\text{quant}} = \text{round}\left(\frac{x - \min}{\max - \min} \times (2^b - 1)\right). \quad (10)$$

The quantization workload can be partitioned into four phases: pre-processing (Lines 5-6), finding min and max (Lines 9-10), normalization (Lines 8-12 and 14), and post-processing (Lines 16 and 18). We profile these phases, and find that the last three phases dominate the quantization time. For example, for OPT-30B with the prompt length of 64, generation length of 8, and zig-zag block size

Algorithm 2 Group-wise quantization

```
function Compress
    Input: tensor, target_bits, quantize_dim
    Output: compressed tensor
    // Pad
    get_new_shape
    pad_tensor_to_new_shape
    // Quantize
    // find min and max value of data within specific dimension
    min = min(data, quantize_dim)
    max = max(data, quantize_dim)
    // min-max normalization
    min-max-norm(data), see Equation 10
    // clamp into target_bits
    clamp(data, target_bits)
    // Pack
    pack_tensor
    // Reshape
    reshape_tensor
    Return compressed tensor
end function
```

of 640, these three phases account for 95% of the quantization time. We focus on these three phases for modeling.

The dequantization has the same three phases except the normalization shown in Equation 11.

$$x_{\text{dequant}} = \left(\frac{x_{\text{quant}}}{2^b - 1}\right) \times (\max - \min) + \min \tag{11}$$

Weights quantization. Equations 12 and 16 model the (de)quantization overhead in an attention layer where wc is the percentage of weights on CPU. The term $num\_weights$ in the two equations quantifies the total number of weights in one attention layer. $4h_1^2$ accounts for the weights in Q, K, V, and output projection; $2h_1 \cdot h_2$ accounts for the weights for two linear transformations.

$$num\_weights = 4h_1^2 + 2h_1 \cdot h_2$$

The weight quantization overhead ($quan_{pf\_wgt}$), modeled in Equation 12, includes the overheads of finding min and max (Equation 13), normalization (Equation 14), and post-processing (Equation 15).

$$quan_{pf\_wgt} = minmax + norm\_wgt + postprocess\_wgt \tag{12}$$

The overhead of finding min and max values is proportional to the number of weights on CPU, shown in Equation 13.

$$minmax = \frac{num\_weights \cdot wc}{cpu\_freq} \tag{13}$$

The overhead of normalization is modeled in Equation 14 according to Equation 10. In Equation 10, given that min, max, and $(2^b - 1)$ are constants, there are only 3 floating point operations per weights. Hence the numerators in Equation 14 quantify the total number of floating point operations.

$$norm\_wgt = \frac{num\_weights \cdot wc \cdot 3}{cpu\_flops} \tag{14}$$

The post-processing phase is mainly about memory copy, modeled in Equation 15.

$$postprocess\_wgt = \frac{num\_weights \cdot wc}{cpu\_mem\_bw} \tag{15}$$

The dequantization overhead is modeled in Equation 16, where there is no overhead for finding min and max, because it has been paid during the quantization.

$$dequan\_wqt = de\_norm\_wgt + de\_postprocess\_wgt \tag{16}$$

The normalization overhead for dequantization (de_norm_wgt) is modeled in the same way as in Equation 14, but replacing cpu_freq with gpu_freq. The post-processing overhead ($de\_postprocess\_wgt$) is modeled in the same way as in Equation 15, but replacing cpu_mem_bw with gpu_mem_bw.

KV cache quantization. Equations 20 and 24 model the (de)quantization overhead. The sizes of prefilled KV cache, old KV cache, and the newly generated KV cache in a transformer layer are quantified in Equations 17-19 where s is the prompt length and n is the generation length. Note that old_kv_cache changes from one token to another. To make the model of old_kv_cache general and simple, we model the average KV cache size in Equation 18.

$$pf\_kv\_cache = 2 \cdot (s + 1) \cdot h_1 \cdot bls \tag{17}$$

$$old\_kv\_cache = (2 \cdot (s + n/2) \cdot h_1 \cdot bls) \cdot n \tag{18}$$

$$new\_kv\_cache = 2 \cdot h_1 \cdot bls \cdot n \tag{19}$$

The overhead of KV cache quantization during the prefill phase ($quan_{pf\_cache}$), modeled in Equation 20, includes the same three types of overhead as in Equation 12 for weights.

$$quan_{pf\_cache} = minmax_{pf\_cache} + norm_{pf\_cache} + postprocess_{pf\_cache} \tag{20}$$

The overhead of finding min and max values is proportional to the KV cache size, shown in Equation 21.

$$minmax_{pf\_cache} = \frac{pf\_kv\_cache}{gpu\_freq} \tag{21}$$

$norm_{pf\_cache}$ is modeled in the same way as for weights shown in Equation 14, except that it happens on GPU instead of CPU.

$$norm_{pf\_cache} = \frac{pf\_kv\_cache \cdot 3}{gpu\_flops} \tag{22}$$

$postprocess_{pf\_cache}$ is mainly about memory copy, modeled in Equation 23.

$$postprocess_{pf\_cache} = \frac{pf\_kv\_cache}{gpu\_mem\_bdw} \tag{23}$$

For $quan\_new\_cache$ in Equation 7, we use the similar formulation as Equations 21-23 but replace pf_kv_cache with new_kv_cache.

The dequantization overhead is modeled in Equation 24, where there is no overhead for finding min and max, because it has been paid during the quantization.

$$dequan\_cache = de\_norm\_cache + de\_postprocess\_cache \quad (24)$$

The normalization overhead for dequantization (de_norm_cache) is modeled in the same way as in Equation 22, but replacing pf_kv_cache with old_kv_cache. The post-processing overhead ($de\_post process\_cache$) is modeled in the same way as in Equation 23 but replacing num_pre_cache with old_kv_cache.

How to use the models. The models include a set of parameters: (1) h_1 and h_2, which are related to the transformer model structure; (2) s, n, and bls, which are user-specified configurations; (3) wc and r, which are determined offline by FlexGen; and (4) gpu_flops, gpu_mem_bw, gpu_freq, cpu_flops, cpu_mem_bw and cpu_freq, which are hardware features. All of them can be easily determined.

The models can be used in the following scenarios.

- Determine whether weight quantization is beneficial. This is done by comparing `load_weight` without quantization with Equation 3 and Equation 4.
- Determine whether KV cache quantization is beneficial. This is done by comparing (`load_cache` + `store_cache`) with Equation 6 + Equation 7.
- Determine the benefit of attention offloading with quantization. This is done by comparing Equation 8 + Equation 9 with Equations 3-7.

4 PARALLELISM CONTROL

Attention computation (when offloaded to CPU) and the other five tasks (load and store tasks) are performed using thread-level parallelism. This indicates multiple facts: (1) multiple tasks can perform in parallel; (2) a single task (such as attention computation), if including multiple operations (e.g., `AccumulateGrad` and `ViewBackward`), can run these operations in parallel and each operation can use multiple threads.

4.1 Performance Characterization with Various Parallelisms

Changing thread-level parallelism to run the six tasks impacts the inference performance. We study the impact in this section.

Evaluation setup. We use two Intel(R) Xeon(R) Gold 6330 CPUs (28 cores each and 56 physical threads in total) and a 40GB NVIDIA A100 GPU (detailed in Table 4). We use OPT-30B, and the prompt length and generation length are 64 and 8 respectively. We use the NUMA first touch and the default setting in FlexGen (i.e., using attention offloading without quantization).

The thread-level parallelism in PyTorch is classified into inter-op parallelism and intra-op parallelism. Inter-op parallelism decides the maximum number of operations that can co-run at any moment, and intra-op parallelism decides the number of threads to run each operation. We use `torch.set_num_interop_threads()` to control inter-op parallelism and `torch.set_num_threads()` to control intra-op parallelism. When we change the number of intra-op parallelism, we use the default setting for inter-op parallelism (i.e., 112). Similarly, when we change the inter-op parallelism, we use the default setting for for intra-op parallelism (i.e., 56).

Performance analysis. Figure 5 shows the results. We have two observations. *First*, when we increase the intra-op parallelism,

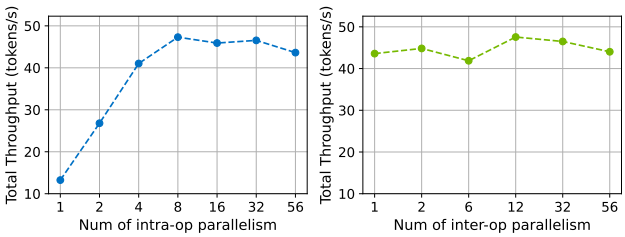

Figure 5: Performance of different thread-level parallelisms.

the performance increases but becomes stable when the number of threads is larger than 8. This is expected, as increasing the number of threads creates pressure on the cache hierarchy and memory bandwidth. This is especially true for those memory-intensive operations in attention (e.g., `AddmmBackward` and `BmmBackward`). *Second*, when we change the inter-op parallelism, the best performance is achieved when the inter-op parallelism is 12. As we further increases it, the performance drops down because of two reasons: (1) the cross-socket memory accesses become more often due to the NUMA effect; and (2) co-running operations creates conflicts on the cache hierarchy.

In conclusion, we need a systematic approach to decide inter-op and intra-op parallelism for high throughput.

4.2 Managing Thread-Level Parallelism

Overview. We introduce an algorithm to determine intra-op and inter-op parallelisms for all tasks, depicted in Algorithm 3. The algorithm decides the inter-op parallelism based on the analysis on the maximum concurrency level in the compute dependency graph of the `compute` task. The algorithm also enumerates the possible number of intra-op parallelism for operations in the `compute` task, and then assigns threads to the load and store tasks based on their data transfer volumes.

Algorithm details. To determine inter-op parallelism, the algorithm starts from the `compute` task, because this task is most time-consuming among the six tasks and the algorithm aims to provide it with enough parallelism. LM-Offload extracts the compute dependency graph of the `compute` task where the dependency between operations is shown (see Figure 6). Assuming that the execution time of each operation is known, we can decide the maximum concurrency level in the graph using the Kahn's algorithm for topological sorting [26] (Line 4 in Algorithm 3). This maximum concurrency level is the inter-op parallelism for the `compute` task (i.e., `inter_op_p_compute`).

To know the execution time of each operation, we must determine the intra-op parallelism for each operation. We apply the same intra-op parallelism to all operations in the `compute` task, because of two reasons: (1) frequently changing it across operations causes cache misses, and (2) the execution time of each operation is short (at the micro-second level), and the overhead of thread scheduling can easily kill the performance.

Algorithm 3 uses a loop to enumerate the intra-op parallelism (Line 3). In each iteration, once the inter-op parallelism for the `compute` task is determined, the algorithm calculates the number of remaining threads (Lines 5). The remaining threads are assigned

Algorithm 3 Thread-level parallelism management

```
1:  function GET_OPTIMAL_PARALLELISM_SETTING
2:      Output: optimal parallelism setting
3:      for intra_op_p_comp = 1 to max_thrs - 5 do
4:          Estimate inter_op_p_comp using the max concur. level
5:          free_thrs = max_thrs -
                (inter_op_p_comp × intra_op_p_comp)
6:          if free_thrs < 5 then
7:              continue
8:          else
9:              Decides the intra_op_p for load/store tasks
10:             Estimate current_throughput using performance modeling
                and offline profiling
11:         end if
12:         if current_throughput > optimal_throughput then
13:             optimal_throughput = current_throughput
14:             record current parallelism setting
15:         end if
16:     end for
17: end function
```

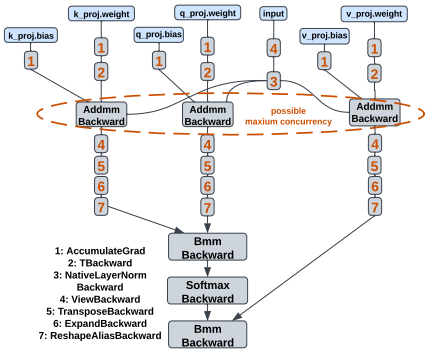

Figure 6: Operation dependency graph in attention computation (i.e., the `compute` task).

to the load/store tasks (five in total). There must be at least five remaining threads to run the load/store tasks, which indicates when exploring parallelisms for the `compute` task, the number of compute threads must be bounded (Line 3). In addition, each load or store task runs an operation, which indicates that the inter-op parallelism should be $inter\_op\_p = inter\_op\_p\_compute + 5$. Also, the intra-op parallelism for each load/store task is in proportion to the data transfer volume so that the larger data transfer has more threads.

To use the algorithm to determine parallelism, we do not need to run operations during the LLM inference to measure execution time. For those operations in the `compute` task, we use offline profiling and collect the execution times of those operations with various intra-op parallelism. This offline profiling happens only once, and the profiling results are repeatedly used during the online LLM inference. For the load/store tasks, their execution times are estimated based on the interconnect bandwidth and data transfer volumes.

5 EVALUATION

LM-Offload introduces quantization-awareness and thread-level parallelism. Building upon FlexGen [20], LM-Offload demonstrates the effectiveness of integrating these crucial techniques to achieve superior performance and efficiency in LLM inference.

5.1 Evaluation Methodology

Evaluation platform. Table 4 presents the hardware platforms used for evaluation. We use a single-GPU platform to demonstrate the effectiveness of LM-Offload in reducing memory consumption and improving inference throughput, and use a multi-GPU platform to showcase LM-Offload's ability to scale its performance seamlessly.

LLM for evaluation. We evaluate two SOTA transformer-based LLMs: LLaMA [23] with 30 and 65 billion parameters, and OPT [29] with 30 and 66 billion parameters. The length of the input prompts is standardized at 64 tokens. We vary the token generation length to assess the performance of LM-Offload under different settings.

Baseline. We compare LM-Offload with two state-of-the-art tensor offloading-based solutions: FlexGen[20] and ZeRO-Inference [2]. ZeRO-Inference does not support partial tensor-offloading (in other words, ZeRO-Inference either offloads the whole tensor or does not offload it at all), which significantly limits its ability to perform inference on large models with long sequences with constrained GPU resources. To infer LLMs with 30 billion weights, ZeRO-Inference must offload all weights and the KV cache to CPU memory, resulting in a substantial reduction in throughput. For ZeRO-Inference, we offload KV cache not weights to CPU while use 4-bit weights quantization (default setting) for reasonable throughput.

5.2 Overall Performance

We evaluate the end-to-end LLM inference throughput of LM-Offload across various token generation lengths and compare them against the ZeRO-Inference and FlexGen baselines. Table 3 presents the detailed LLM deployment configurations and results. LM-Offload demonstrates a significant improvement in throughput compared to both ZeRO-Inference and FlexGen for all tested configurations.

Compared with FlexGen, LM-Offload achieves the same or comparable inferenceable batch size while better utilizing GPU memory capacity by enabling more weights to be stored in GPU memory through effective quantization. By leveraging quantization, using the GPU for attention computation and parallelism control, LM-Offload outperforms FlexGen by up to 195% (134% on average) across the four LLM models.

In comparison to ZeRO-Inference, LM-Offload enables an average of 24× larger batch sizes for inference, due to the use of quantization and partial tensor offloading, which reduces the memory footprint of the weights and KV cache. The larger batch sizes directly translate to higher inference throughput, with LM-Offload achieving up to 188% (57% on average) improvement compared to ZeRO-Inference. It is worth noting that LM-Offload performs slightly worse than ZeRO-Inference by 7% in OPT-30B with a token generation length of 128. This is because when the model size is relatively small but the token generation length is large, the overhead of offloading and retrieving weights from CPU memory becomes more pronounced compared to the computation time. However,

Table 3: Comparison between FlexGen, ZeRO-Inference, and LM-Offload. "len" represents the token generation length. "bsz" denotes the batch size. "wg", "cg", and "hg" indicate the percentage of weights, KV cache, and hidden activations stored on GPU memory, respectively. "mem" and "tput" refer to the total memory consumption (in GB) and LLM inference throughput (in tokens/sec), respectively. "norm tput" represents the normalized throughput, which is calculated by dividing the throughput of each framework by the throughput of LM-Offload for the given configuration. The best performance is highlighted.

		OPT-30B							OPT-66B							LLaMA-30B							LLaMA-65B						
Framework	**len**	**bsz**	**wg**	**cg**	**hg**	**mem**	**tput**	**norm tput**	**bsz**	**wg**	**cg**	**hg**	**mem**	**tput**	**norm tput**	**bsz**	**wg**	**cg**	**hg**	**mem**	**tput**	**norm tput**	**bsz**	**wg**	**cg**	**hg**	**mem**	**tput**	**norm tput**
FlexGen	8	1792	55	0	0	222	51	0.43	780	23	0	100	246	24	0.61	1536	48	0	0	226	35	0.36	1140	14	0	100	323	20	0.45
	16	1600	55	0	0	221	56	0.40	828	23	0	100	269	22	0.52	1408	48	0	0	229	38	0.35	1020	14	0	100	322	20	0.43
	32	1344	55	0	0	222	53	0.36	702	24	0	0	271	17	0.49	1152	48	0	0	226	37	0.33	616	16	0	100	266	23	0.59
	64	960	55	0	0	214	50	0.40	720	24	0	0	326	14	0.43	832	48	0	0	220	35	0.36	616	16	0	100	315	18	0.48
	128	640	55	0	0	214	41	0.40	480	24	0	0	326	11	0.45	576	48	0	100	226	31	0.35	392	16	0	100	306	15	0.47
ZeRO-Inference	8	64	100	0	100	61	94	0.80	32	100	0	100	127	28	0.71	64	100	0	100	66	34	0.36	32	100	0	100	126	19	0.44
	16	64	100	0	100	62	116	0.83	16	100	0	100	125	32	0.75	64	100	0	100	67	68	0.62	16	100	0	100	126	25	0.53
	32	64	100	0	100	63	113	0.78	8	100	0	100	124	20	0.59	64	100	0	100	69	73	0.65	8	100	0	100	135	39	0.98
	64	64	100	0	100	66	**126**	**1.00**	4	100	0	100	123	11	0.34	64	100	0	100	72	69	0.71	4	100	0	100	130	31	0.82
	128	64	100	0	100	71	**110**	**1.07**	4	100	0	100	124	10	0.41	64	100	0	100	78	63	0.70	4	100	0	100	135	31	0.96
LM-Offload	8	1792	55	0	0	222	**117**	**1.00**	780	23	0	100	246	**40**	**1.00**	1536	42	0	0	226	**95**	**1.00**	1140	14	0	100	323	**44**	**1.00**
	16	1600	55	0	0	221	**139**	**1.00**	828	23	0	100	269	**42**	**1.00**	1408	48	0	0	229	**109**	**1.00**	1020	14	0	100	322	**47**	**1.00**
	32	1344	55	0	0	222	**144**	**1.00**	702	24	0	0	271	**34**	**1.00**	1152	48	0	0	226	**111**	**1.00**	616	16	0	100	266	**40**	**1.00**
	64	960	70	0	0	214	**126**	**1.00**	720	24	0	0	326	**31**	**1.00**	832	48	0	0	220	**96**	**1.00**	616	16	0	100	315	**38**	**1.00**
	128	640	75	0	0	246	102	1.00	480	24	0	0	326	**25**	**1.00**	576	48	0	100	226	**89**	**1.00**	392	16	0	100	306	**32**	**1.00**

Table 4: Hardware details in evaluation platform.

		Specification
Single GPU	CPU	2× Intel Xeon Gold 6330 CPU 56 cores in total, 240 GB host memory
	GPU	1× NVIDIA A100 with 40GB GPU memory
	Interconnect	PCIe 4.0 x16, with a total bidirectional bandwidth of 64 GB/s
Multi-GPUs	CPU	2×IBM POWER9 44 cores in total, 280GB host memory
	GPU	4× NVIDIA V100 with 16GB GPU memory
	Interconnect	NVIDIA NVLink 2.0 with a total bidirectional bandwidth 300GB/s

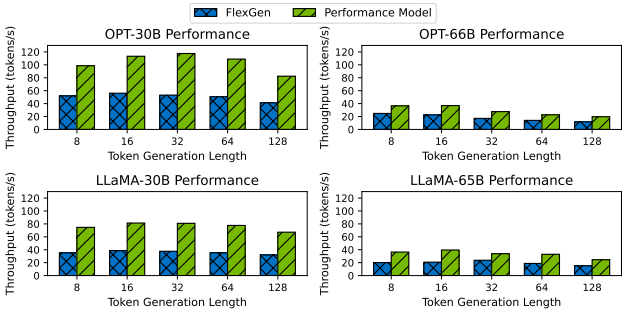

Figure 7: Effective Quantization.

this performance difference is minimal and only observed in this specific, uncommon configuration.

Overall, the evaluation results highlight LM-Offload's ability to strike a good balance between extending GPU capacity through offloading and achieving high performance. By intelligently applying quantization and parallelism control, optimizing the computation-offloading trade-off, LM-Offload outperforms state-of-the-art baselines in terms of both memory efficiency and inference throughput.

5.3 Evaluation of Performance Modeling

To quantify the effectiveness of performance modeling, we evaluate LM-Offload with thread-level parallelism control disabled. Figure 7 presents the results. We do not compare with ZeRO-Inference, because it does not support the quantization of KV cache and partial tensor offloading. Table 3 summarizes the offload policy. By accurately modeling the overhead and benefits of quantization, LM-Offload maximizes inference throughput by allocating more weights to GPU memory (compared to the cases without quantization-aware performance modeling): LM-Offload outperforms FlexGen by 90%-121% in all configurations for 30 billion parameter LLMs. Moreover, the performance benefits of LM-Offload remain consistent as the model size increases, demonstrating its scalability and effectiveness in optimizing inference for LLM.

Table 5: CPU last-level cache misses under LM-Offload and default threading.

Parallelism Control	Load Misses	Store Misses
Disable (Default)	10 Billion	19 Billion
Enable	6 Billion	12 Billion

5.4 Evaluation of Parallelism Control

We investigate the benefit of thread-level parallelism control on the single GPU evaluation platform using OPT-30B with a token generation length of 8. We measure the execution time of the six tasks during the decode phase (Algorithm 1) with asynchronous execution disabled and also report the end-to-end inference time with asynchronous execution enabled. By default, the system utilizes 56 threads for intra-op parallelism and 112 threads (all hyper-threads) for inter-op parallelism. In contrast, LM-Offload optimizes the thread allocation by using 12 threads for inter-op parallelism and 16 threads for intra-op parallelism.

Figure 8 presents the results of thread-level parallelism control with LM-Offload. The `compute` task benefits the most, with a 32% reduction in execution time compared to the default setting. LM-Offload successfully reduces thread contention, resulting in a 19%

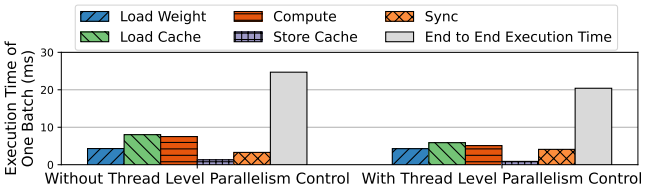

Figure 8: Effectiveness of thread-level parallelism control. Load/store activations have negligible execution time and are not shown in this figure.

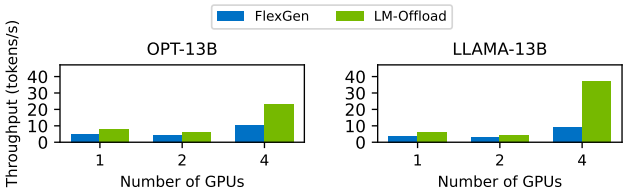

Figure 9: Effective Scalability.

average reduction in execution time across all tasks and a 38% reduction in end-to-end execution time.

To further understand the benefit of thread parallelism control, we measure the number of last-level cache misses during LLM inference. Table 5 shows the results. Compared with the default thread scheduling, LM-Offload reduces cache misses by 38% for both load and store operations.

5.5 Multi-GPU Evaluation

We evaluate LM-Offload on the multi-GPU platform (Table 4) using pipeline parallelism. For the evaluation, we use OPT-13B and LLaMA-13B models configured with a prompt length of 256 and a token generation length of 64. We conduct weak scaling tests by doubling the inference batch size as the number of GPUs increases. We compare LM-Offload with FlexGen but not ZeRO-Inference, because ZeRO-Inference does not support multi-GPU for inference very well. Figure 9 reveals that LM-Offload outperforms FlexGen in all cases, by up to 327% (112% on average). Moreover, LM-Offload demonstrates better scalability compared to FlexGen, as the performance gap between LM-Offload and FlexGen increases by up to 13.9 × as the number of GPUs increases from 1 to 4.

6 RELATED WORK

Tensor offloading. Various methods address memory limitations in deep learning training. ZeRO-Offload [16] and ZeRO-Infinity [14] use CPU memory to augment GPU memory during LLM training. Betty [27] focuses on GNN training, offloading tasks to CPUs. DyNN-Offload [17] guides tensor offloading for dynamic neural networks. LLM-in-a-flash [1] optimizes tensor computation and layout, while PowerInfer [21] allocates neurons to CPU or GPU for efficient inference.

LLM performance optimization. To speed up training and extend context in Transformers, methods like attention approximation have been proposed at algorithm levels. Such as Linformer [25] approximates self-attention with a low-rank matrix, reducing complexity to linear ($O(n)$) from $O(n^2)$. Longformer [3] employs sliding window attention to focus on a token subset. At the system level, vLLM [10] introduces paged attention to enhance batch size and throughput. SpecInfer [11] utilizes tree attention to eliminate redundant KV cache allocation. FlashAttention [4] improves memory efficiency with reordered attention computation.

7 CONCLUSIONS

We introduce LM-Offload, a framework for efficient LLM inference that utilizes CPU memory through tensor offloading, quantization, and thread-level parallelism control. LM-Offload employs quantization-aware offloading, which models the costs and benefits of quantization. LM-Offload incorporates a detailed thread-level parallelism control to maximize the benefits of asynchronous execution of tensor offloading, uploading, and CPU computation. LM-Offload demonstrates higher LLM inference throughput compared to state-of-the-art systems for LLM inference such as FlexGen and ZeRO-Inference. LM-Offload outperforms FlexGen and ZeRO-Inference by up to 2.95× (2.34× on average) and 2.88× (1.57× on average) respectively in inference throughput.

REFERENCES

[1] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL]

[2] 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.

[3] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150* (2020).

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

[5] Yubin Duan and Jie Wu. 2021. Computation offloading scheduling for deep neural network inference in mobile computing. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[6] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[7] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. 2021. Rim: Offloading inference to the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 80–92.

[8] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* (2018).

[10] 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*. 611–626.

[11] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2023. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781* (2023).

[12] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. 2020. Distributed inference acceleration with adaptive DNN partitioning and offloading. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 854–863.

[13] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[14] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *CoRR* abs/2104.07857 (2021).

[15] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2020. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[16] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *USENIX Annual Technical Conference*.

[17] Jie Ren, Dong Xu, Shuangyan Yang, Jiacheng Li, Zhicheng Zhang, Christian Navasca, Chenxi Wang, Guoqing Harry Xu, and Dong Li. HPCA, 2024. Enabling Large Dynamic Neural Network Training with Learning-based Memory Management. *In 30th International Symposium on High-Performance Computer Architecture* (HPCA, 2024).

[18] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*.

[19] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. 2021. Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 226–237.

[20] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[21] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2023. *PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU.* Technical Report. Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University.

[22] Sam Stoelinga. 2023. Calculating GPU Memory for LLM. https://www.substratus.ai/blog/calculating-gpu-memory-for-llm/. Accessed: 2023-12-09.

[23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[24] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[25] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768* (2020).

[26] Wikipedia. 2023. Topological sorting — Kahn's algorithm. https://en.wikipedia.org/wiki/Topological_sorting. Accessed: 2024-04-14.

[27] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. 2023. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. 2020. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th conference on embedded networked sensor systems*. 476–488.

[29] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]