KV-Runahead: Scalable Causal LLM Inference by Parallel Key-Value Cache Generation

Minsik Cho [1] Mohammad Rastegari [2] Devang Naik [1]

Abstract

Large Language Model or LLM inference has two phases, the prompt (or prefill) phase to output the first token and the extension (or decoding) phase to the generate subsequent tokens. In this work, we propose an efficient parallelization scheme, KV-Runahead to accelerate the prompt phase. The key observation is that the extension phase generates tokens faster than the prompt phase because of key-value cache (KV-cache). Hence, KV-Runahead parallelizes the prompt phase by orchestrating multiple processes to populate the KV-cache and minimizes the time-to-first-token (TTFT). Dual-purposing the KV-cache scheme has two main benefits. First, since KV-cache is designed to leverage the causal attention map, we minimize computation and computation automatically. Second, since it already exists for the extension phase, KV-Runahead is easy to implement. We further propose context-level load-balancing to handle uneven KV-cache generation (due to the causal attention) and to optimize TTFT. Compared with an existing parallelization scheme such as tensor or sequential parallelization where keys and values are locally generated and exchanged via all-gather collectives, our experimental results demonstrate that KV-Runahead can offer over $1.4\times$ and $1.6\times$ speedups for Llama 7B and Falcon 7B respectively.

1. Introduction

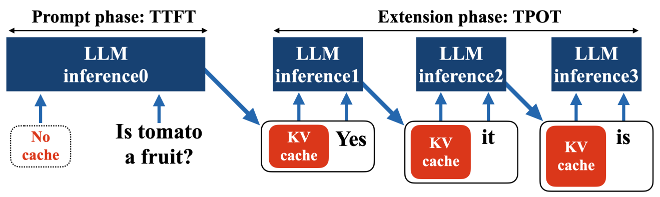
Large language models or LLMs, and especially Generative Pre-trained Transformer (GPT) models have shown excellent performance on many complex language tasks (Ouyang et al., 2022; Zhang et al., 2022a). However, the decoder architecture and autoregression execution in LLMs pose two challenges for efficient inferences: **a) Time-to-first-token or TTFT**: consuming potentially a long user context and generate the first token **b) Time Per Output Token or TPOT**: generating the subsequent tokens fast (Liu et al., 2023a). The second challenge is known to be a memory-bound problem, and a large body of research has been done (Pope et al., 2022), including sparsification, quantization, or weight clustering (Frantar et al., 2023; Lin et al., 2023; Cho et al., 2023; Liu et al., 2023b) or speculative decoding (Leviathan et al., 2023; Chen et al., 2023). But, the first challenge for a long user context is largely a compute-bound problem (Liu et al., 2023a; NVidia-LLM, 2023) and critical for favorable user experience with retrieval augmentation (Ram et al., 2023), in-context learning (Dong et al., 2023), summarization (Zhang et al., 2023b), story generation (Zhang et al., 2023a), and so on.

Since TTFT for a long context is compute-bound, one solution is to use more computing power in the form of parallelization. The current SOTA in LLM parallelization inlcudes tensor and sequential parallelization (Patel et al., 2023; Li et al., 2023; Korthikanti et al., 2022; NVidia-LLM, 2023) where the key and value computations are distributed over multiple processes and subsequently exchanged, aiming to compute the attention map perfectly in parallel. The methods above are generic enough to drive LLM inference (Vaswani et al., 2017), but not specialized enough for scalable LLM inference, as the causality in attention is not fully leveraged, resulting in up to $2\times$ overhead in terms of both computation and communication over the ideal case.

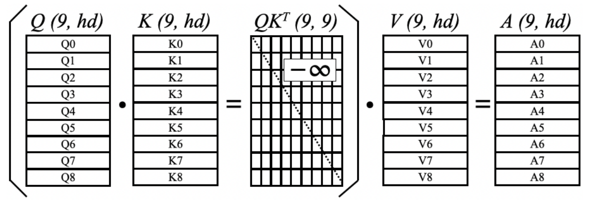
Therefore, we propose a novel yet effective parallelization technique tailed for LLM inference, KV-Runahead to minimize TTFT. By re-purposing the key-value cache or KV-cache (NVidia-LLM, 2023) mechanism (which exists anyway for subsequent token generation), our proposed KV-Runahead uses other processes to populate KV-cache for the last process with context-level load-balancing. Since KV-cache assumes causal attention computation, KV-Runahead reduces the computation and communication costs and offers lower TTFT over the existing methods. Further, KV-Runahead requires minimal engineering costs, as it simply makes the KV-cache interface dual-purposed. In detail, our contributions are the following:

[1] Apple. USA [2] Meta. USA (the work done while being with Apple). Correspondence to: Minsik Cho <minsik@apple.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).



(a) LLM inference consists of two phases: the prompt phase for the first token and the extension phase for the next tokens.



(b) Causal attention computation of one layer on one process: the upper triangle of QK^T will be masked out with $-\infty$.

Figure 1. LLM inference begins with the prompt phase to generate the KV-cache and the first token which drive the extension phase as in (a). Inside each layer of the LLM as in (b), a causal attention map (QK^T) is built to compute the attention A from query, value, and key s (Q, K, V). Computing attention thus has $O(C^2)$ complexity where C is the user context.

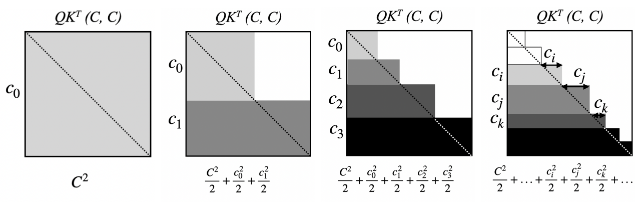
- **We demonstrate that KV-cache scheme can be dual-purposed to parallelize LLM inference for low TTFT**. Since the KV-cache is built on the causal decoder and gets populated in parallel, KV-Runahead can offer considerable compute and communication savings over tensor/sequential parallelization.
- **We show that using KV-cache for parallelization enables asynchronous communication**. Thus, KV-Runahead replaces global synchronization with point-to-point asynchronous communication, and provides robustness against network bandwidth fluctuation.
- **We highlight that context-level partitioning can load-balance parallel LLM inference.** Asymmetric computations and communication rise from KV-cache and its dependency chain across parallel processes. Yet, we can mitigate the negative effects on TTFT with the proposed context-level load-balancing.
- **We propose that hierarchical grid search for efficient context-partitioning**. Such search results contribute to a lookup table from which a TTFT-minimizing partitioning can be interpolated for various context lengths.

2. Related Works

LLM Inference: Generative LLM inference consists of two steps as in Fig. 1 (Patel et al., 2023). Once the user context is received, all the input tokens are consumed to generate the first token, which is called the prompt phase. At the same time, the computed key and value embeddings are saved as KV-cache (Park et al., 2020; Liu et al., 2023a) and fed to all subsequent token generations to expedite the extension phase. Accordingly, KV-cache grows as more tokens are generated, because the next token generation needs to attend to all previous tokens, including the user context. While the critical metric for the extension phase is time-per-output-token or TPOT, the prompt phase needs to deliver the first token fast which is measured as time-to-first-token or TTFT.

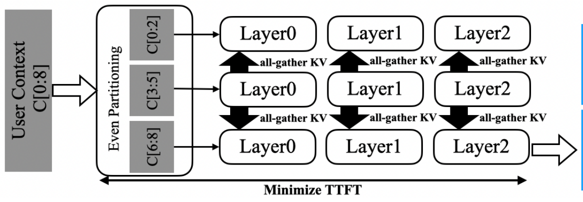
TTFT Optimization: Minimizing TTFT, especially for long context requires two efforts: efficient KV-cache management and fast attention map computation. PagedAttention (Kwon et al., 2023) facilitates the exchange of data including KV-cache between different memory subsystems to handle long contexts. Infinite-LLM (Lin et al., 2024) suggests distributed KV-cache management system at the cloud scale to adaptively handle extremely long context lengths. CacheGen (Liu et al., 2023a) proposes compressing KV-cache for pre-computed contexts to lower TTFT. SplitWise (Patel et al., 2023) proposes to use two different platforms, one with high computing capacity for the prompt phase and the other with low computing capacity for the extension phase by transferring the LLM states, including KV-cache from the first to the second platforms.

LLM Inference Parallelization: Since TTFT optimization is compute-bound, one can employ parallel DNN inference. Pipeline parallelism shards the layers of a model across multiple processes, splitting the model into several stages or layers (Huang et al., 2019; Narayanan et al., 2021a; Agrawal et al., 2023). Tensor Parallelism is one of the popular parallel methods from (HuggingFace-TensorParallelism; Shoeybi et al., 2020; Narayanan et al., 2021b) where a large matrix multiplication is scattered and then the partial output matrices are gathered, and is known to be superior to pipeline parallelism (Patel et al., 2023). Sequence parallelization (NVidia-LLM, 2023; Li et al., 2023) is a novel

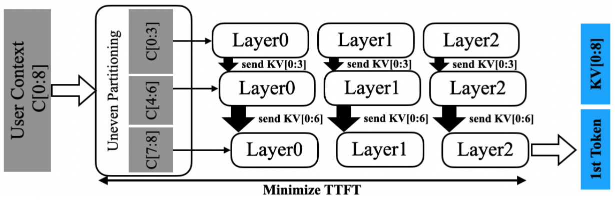

QK^T (C, C)
c_0
C^2
c_1
C^2/2 + c_0^2/2 + c_1^2/2
c_2
c_3
C^2/2 + c_0^2/2 + c_1^2/2 + c_2^2/2 + c_3^2/2
c_i
c_j
c_k
C^2/2 + ... + c_i^2/2 + c_j^2/2 + c_k^2/2 + ...

(a) 1 process (b) 2 processes (c) 4 processes (d) p processes

Figure 2. QK^T computation coverage using BLAS matrix-matrix multiplication: by linking each context partition to KV-cache, we can closely approximate the lower triangular part and minimize unnecessary dot products. Note the upper triangular part of the attention will be masked out to enforce causality.



(a) Tensor+Sequence Parallel Inference: With even context partitioning, each process will perform *all-gather* over K, V. All computations in the layers are symmetric and globally synchronized, and the user context can be evenly partitioned. (See Fig. 4).



(b) KV-Runahead Inference: Since the later processes wait for the KV-cache to be ready, the layers will asynchronously communicate, and the user context is unevenly partitioned (for load-balancing) to minimize TTFT. (See Fig. 5).

Figure 3. Comparing the existing tensor+sequence parallel scheme with the proposed KV-Runahead for parallel LLM inference.

data parallel algorithm (by evenly partitioning the input sequences over multiple processes) coupled with a distributed ring attention algorithm. By deploying the ring topology over all the devices, each process exchanges the key and value embedding with neighbors and builds a full attention map locally.

Both tensor and sequence parallelizations in LLM are mathematically similar in a sense that **a)** one of two matrices (i.e., either activations or parameters) in multiplication will be sharded over multiple devices, **b)** both require collective communication to merge the partial outcomes. Hence, both are popular for parallel LLM inference (Korthikanti et al., 2022), yet not specialized enough for causal attention, leading to excessive computation and communication overheads.

3. Causal LLM Scalability and Motivation

In this section, we will discuss the lower bound of the scalability of a causal attention-based LLM for a sufficiently long user context C over parallel p processes. Assume that the user context C is partitioned into $C = \{c_0, c_1, c_2, ..., c_{p-1}\}$ for p processes, and each process is exclusively mapped to one compute fabric (e.g., GPU). The minimum compute over p to generate the first token, $TTFT(p)$ with perfect load-balancing is as follows:

$$TTFT(p) \geq \alpha \left[\frac{\frac{1}{2}C^2 + \frac{1}{2}(\sum_0^{p-1} c_i^2)}{p} \right] \tag{1}$$

$$\geq \alpha \left[\frac{\frac{1}{2}C^2 + \frac{1}{2}p(\frac{C}{p})^2}{p} \right]$$

$$= \alpha \left[\frac{C^2}{2}(\frac{1}{p} + \frac{1}{p^2}) \right]$$

$$= \frac{TTFT(1)}{2}(\frac{1}{p} + \frac{1}{p^2}) \tag{2}$$

$$= \mathbf{TTFT}^*(p) \tag{3}$$

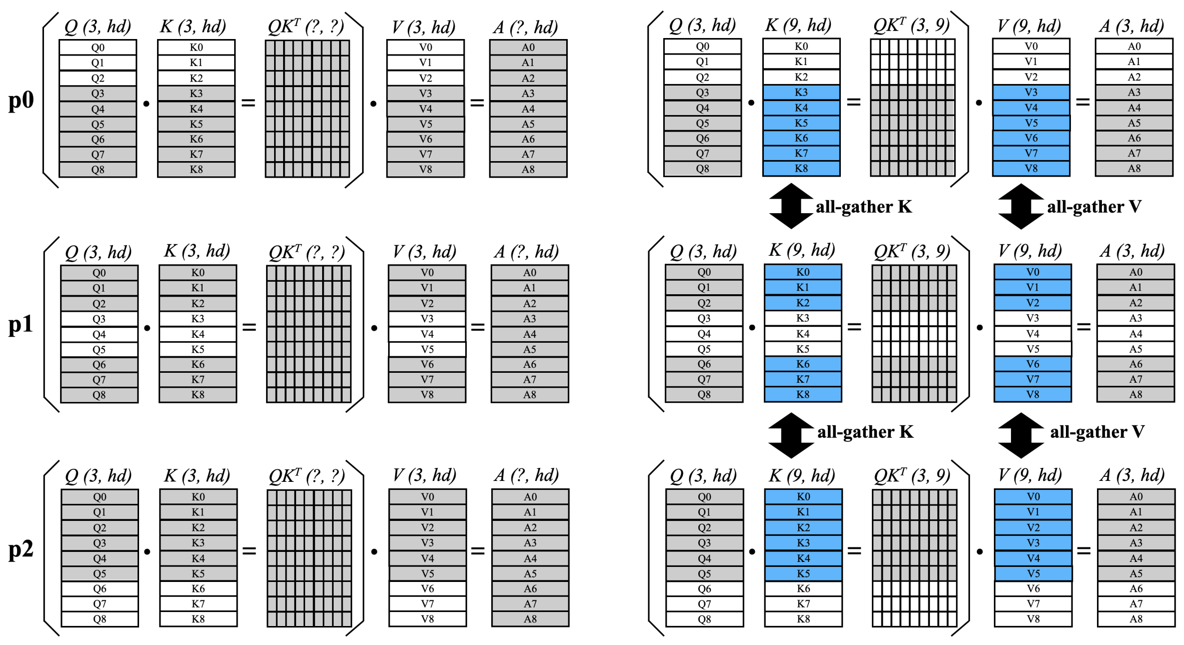
where α is a fitting coefficient such that $TTFT(1) = \alpha C^2$ (single process performance) (Dao et al., 2022), and $\mathbf{TTFT}^*$(p) is the lower bound of TTFT over p. The significance of $\mathbf{TTFT}^*(p)$ is that for a very long user context, there exists super-linear scalability (i.e., more than $2\times$ speedup with 2 processes) with causal LLM in the ideal setup, such as perfect load-balancing, zero communication costs, and so on. Please see the super-linear scalability of KV-Runahead reported in Fig. 8 (d).

Fig. 2 visualizes the concepts behind Eq. (1) which essentially divides an attention map, $QK^T(C, C)$ in the shaded regions over p processes. We need to practically compute multiple rectangle regions using matrix-matrix multiplication and mask out the upper triangle part (which is how most LLMs are implemented). Therefore, with more partitions, we can eliminate the wasted computation. Other equally good partitioning setups (i.e., using vertical rectangles to approximate the lower triangle) could exist, but the one in Fig. 2 (d) is LLM-friendly: easy to generate at the context level, and exactly aligned with KV-cache.

Hence, we can intuitively map the partitions in Fig. 2 (d) to p processes, which can be implemented by dual-purposing the already-existing KV-cache interface with minor efforts, leading to the motivation behind KV-Runahead. Also, as seen in Fig. 2 (d), each process will suffer from varying computation load, thus one may not effectively minimize TTFT. Yet, optimizing c_i alone may lead to global over-computation. Hence, we perform context-level partitioning for load-balancing and minimal TTFT in KV-Runahead.

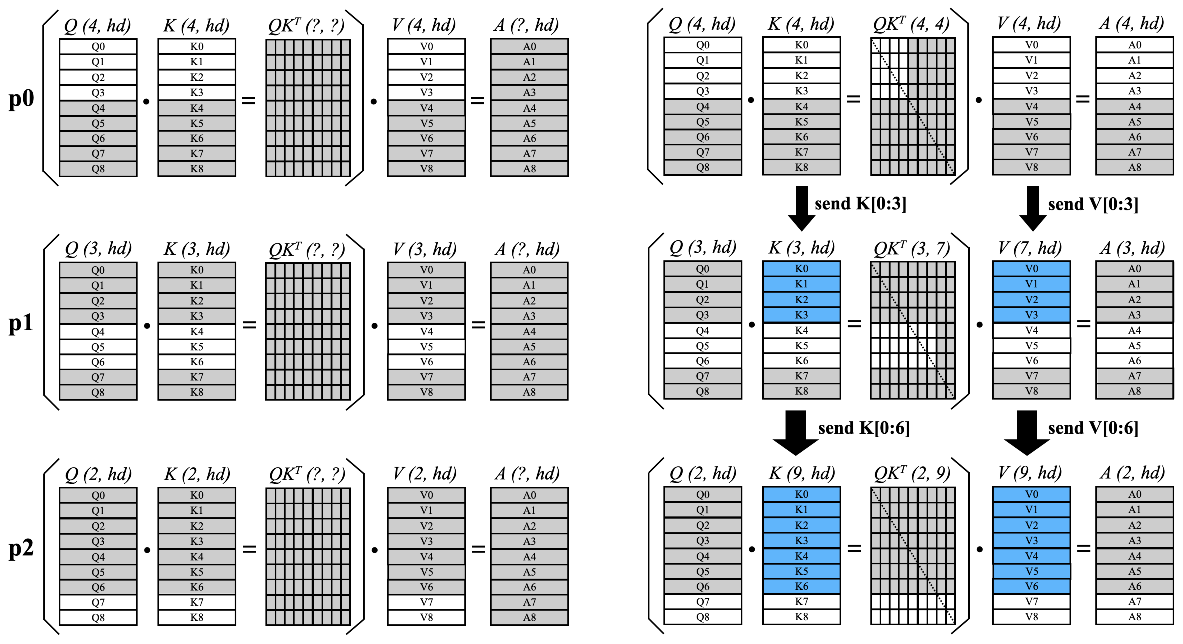
4. KV-Runahead Overview

In Fig. 3, the proposed KV-Runahead is illustrated and compared against Tensor/sequence parallel inference (or **TSP**), which characterizes both tensor parallelism (Shoeybi et al., 2020; Narayanan et al., 2021b) and sequence parallelization (Li et al., 2023). As in Fig. 3 (b), KV-Runahead starts with uneven context partitioning for load-balancing. The existing TSP parallelizes the forward computation itself, but KV-Runahead achieves parallel inference by utilizing multiple processes to populate KV-caches for the last process. Therefore, unlike TSP where computation is symmetric and evenly distributed (thus no need to balance out context partitioning), KV-Runahead needs good context partitioning



(a) Q, K, V are computed for the evenly partitioned context. (b) K, V are exchanged by *all-gather* to mimic Fig. 1 (b).

Figure 4. Tensor/sequence parallel inference over 3 processes $p_{\{0,1,2\}}$ within a layer to compute attention map (QK^T) and final attention A: Each process will compute the equal amount of (Q, K, V) in (a), and then globally share (K, V) using *all-gather* collectives to compute the equally sized partial QK^T (i.e., **27 dot-products** needed on each) and partial A. Such *all-gather* operations require global synchronization, and incur the **traffic** for **36** (K, V) entries (i.e., the number of blue rows in (K, V)).



(a) Q, K, V are computed for the uneven partitioning. (b) KV-caches are handed down from p_i to p_{i+1} by *send*.

Figure 5. KV-Runahead execution over 3 processes $p_{\{0,1,2\}}$ within a layer to compute attention map (QK^T) and final attention A: Each process will compute different amounts of (Q, K, V) in (a), and the maximum amount for QK^T is **21 dot-products** on p_1 (in contrast to **27** from Fig. 4 (b)). The locally computed (K, V) are passed down to the following processes as KV-cache using point-2-point one-way *send* (i.e., $p_0 \rightarrow p_1 \rightarrow p_2$). Our communication is much cheaper than global *all-gather* in Fig. 4 (b), as the **traffic** incurred in KV-Runahead is **22** (i.e., the number of blue rows in (K, V)), which is much lower than **36** from Fig. 4.

to balance out the KV-cache amount from each compute process and to minimize TTFT.

Once partitioning is complete, each process will run each layer, conditioning on the KV-cache from its precedent process. In detail, the current process must wait for the required KV-cache to arrive from the precedent (i.e., notice the layer misalignment in Fig. 3 (b)), forming a dependency chain via local peer-to-peer communication rather than global synchronization via *all-gather* (Thakur et al., 2005).

We will first elaborate on how KV-Runahead works inside each layer in terms of compute/communication saving in Section 4.1, and then discuss the context partitioning for load-balancing in KV-Runahead in Section 4.2. Finally, Section 4.3 briefly discuss implementing KV-Runahead.

4.1. Forward Execution

The causal attention computation on a single process is shown in Fig. 1 (b), which is to be parallelized in this section. For a given context, once (Q, K, V) are computed, QK^T or attention map is computed for A. Although only the lower triangular part of QK^T is needed due to the causality, the entire QK^T is commonly computed via dense matrix-matrix multiplication first, then a mask is added in general (HuggingFace-Transformers), because no good mapping to BLAS-L3 exists or writing a custom kernel is expensive (NVidia-cuBLAS).

One SOTA way to enable parallel inference for LLM (e.g., GPT-3, Llama, and BLOOM), would be to utilize tensor and sequence parallelizations (Li et al., 2023; Patel et al., 2023; Shoeybi et al., 2020; Korthikanti et al., 2022), Tensor/sequence parallelization or **TSP** in Fig. 4 where the focus is on parallelizing the single process behavior from Fig. 1 (b). In TSP, for a given evenly partitioned context, (Q, K, V) are independently computed on each process as in Fig. 4 (a). Then, the collective operation *all-gather* is performed to exchange K and V to all processes so that QK^T can be evenly distributed as shown in Fig. 4 (b). Although TSP faithfully follows the single process case in Fig. 1 (b), it does not take advantage of the causality in LLM inference.

In our KV-Runahead, we start with a given yet unevenly partitioned context, and (Q, K, V) are independently computed on each process as in Fig. 5 (a). Thus, each process computes a different number of entries in (Q, K, V). Then, KV-Runahead simply populates the KV-cache from each process and hands over to the next process in chain, mimicking the extension phase in Fig. 1 (a). As a result, only the last process will have the full (K, V), but still each process can output the A in the same shape as Q, driving the next layer. Since KV-cache itself is built upon the causality, KV-Runahead can automatically minimize the computation of the upper triangle and reduce the number of dot-products for QK^T. For example, 27 dot-products are needed on all the processes in TSP as in Fig. 4 (b), but KV-Runahead requires 21 (max out of $\{p_0 : 16, p_1 : 21, p_2 : 18\}$) as in Fig. 5 (b). This also highlights the motivation behind uneven context partitioning to minimize the largest QK^T computation.

KV-Runahead also removes the global synchronization points and reduces the total traffic volume exchanged among processes. *All-gather* operations in Fig. 4 (b) force all the processes to stop and secure the full (K, V) (Thakur et al., 2005), while KV-Runahead shares only the local KV-cache with the next process via point-to-point *send* operations. As a result, TSP in Fig. 4 (a) requires to share 36 (K, V) entries to get to the state in Fig. 4 (b), but KV-Runahead only needs 22 to transit to Fig. 5 (b). Such a dependency chain from KV-cache introduces a longer wait time for the later processes, but KV-Runahead can outperform TSP even with such overheads.

In theory, with a sufficient number of parallel processes and a sufficiently long user context (i.e., QK^T dominates the runtime), KV-Runahead can offer up to 2× speed up over TSP, because both total QK^T computation and network traffic among processes in KV-Runahead are half of those in TSP. It could be possible to handcraft a custom/expensive BLAS kernel for TSP to avoid over-computation. Even with a tailored custom kernel, however, the communication involved in TSP remains suboptimal as it still uses All-gather to exchange (K, V) The proposed KV-Runahead avoids both over-computation and wasted network traffic seamlessly, by dual-purposing the LLM-specific KV-cache scheme (which already exists for the extension phase).

Additionally, the same computational savings achieved with the custom GPU kernel, can also be applied to KVR. From Fig 5 (b), we can still see some wasted computation. Hence, a custom kernel would save such waste to further enhance the performance of KVR. Yet, the benefit from a custom kernel would diminish with more GPUs in parallel, as the nature of KV-cache allows our technique to approximate the unmasked lower triangle more accurately with more processes, as illustrated in Figure 2 (b) and (d).

For simplicity, assume that a user context C is even partitioned for KV-Runahead and TSP over p processes. Then, the total TSP traffic $\mathbf{Net}_{tsp}$ can be written as follows:

$$\mathbf{Net}_{tsp}(C, p) = p(p-1)\frac{C}{p} \tag{4}$$

$$= (p-1)C \tag{5}$$

which is essentially the total number of (K, V) entries from other processes. The total KV-Runahead traffic $\mathbf{Net}_{kvr}$ is the sum of the total KV-cache put into the network.

$$\mathbf{Net}_{kvr}(C,p) = \frac{C}{p} + \frac{2C}{p} + \frac{3C}{p} + ... \quad (6)$$

$$= \sum_{i=1}^{p-1} \frac{iC}{p} = \frac{(p-1)}{2}C \quad (7)$$

The 2× reduction is over the total computation and network traffic, not for each individual process. Therefore, it is critical to perform load-balancing to maximize the gain over TSP and minimize TTFT, and KV-Runahead accomplishes it by context-level load-balancing in Section 4.2.

4.2. Context-Level Load-Balancing

As discussed in Section 3, KV-Runahead needs load-balancing for low TTFT. We propose running off-line search for the best partitioning, and then store the results in a partitioning lookup table. For example, we pre-compute the optimal partitioning of user contexts at various lengths for a given number of processes off-line by measuring TTFT on the target hardware, and then contribute the search results to a lookup table. During inference, we can predict the best partitioning by interpolating the two nearest known entries in the lookup table. For the example of 10k prompt, we can interpolate from the known partitioning configurations from 8k and 12k in the lookup table.

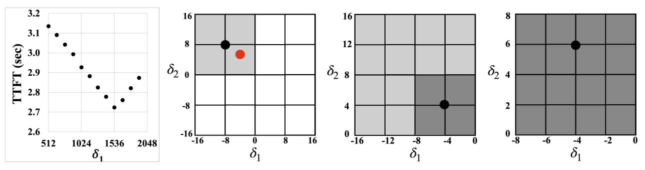
Finding the best partitioning configuration for a given user context, although one-time off-line overhead, can be expensive. Hence, we propose a hierarchical grid search for acceleration. From the nature of KV-Runahead, it is straightforward to see that finding the TTFT-optimal partitioning has two conflicting objectives.

- The partitions for the earlier processes must be small, otherwise the later processes will wait too long for the earlier ones to populate KV-caches and send them over.
- The partitions for the later processes need to be small, otherwise the later processes will the bottleneck in generating the first token.

For two processes, we can use a binary search to find out the best partitioning. Fig. 6 (a) shows how TTFT changes as we grow the partition for the p_0 for a 16k context where the partitioning is $C[0, 8192 + \delta_1, 16384]$. As δ_1 grows, it bottoms at the partition of $[0, 9728, 16384]$ (i.e., $\delta_1 = 1536$, thus p_0 takes C[0:9728] and p_1 takes in C[9728:16384]).

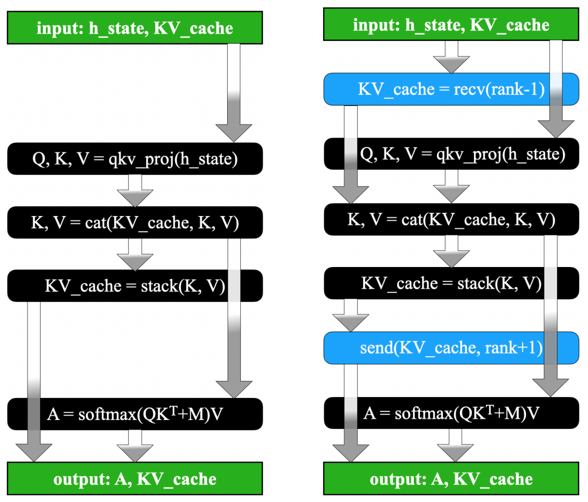
By generalizing the binary search into a hierarchical grid search for multiple processes (Zhang et al., 2022b), we can find a high-quality partitioning fast for a given user context length. Figs. 6 (b-d) depict the proposed search process for a user context length of 96 over 4 processes, which is to find the optimal (δ_1, δ_2) for the partitioning of $C[0, 32+\delta_1, 64+\delta_2, 96]$. In the first level, we set the search stride as 8 and measure the TTFTs on each grid. Once we find the best performing (δ_1, δ_2) pair, we limit the search to the gray grid and reduce the search stride to 4 to perform another scan as in Fig. 6 (c). We repeat the same process recursively until the minimum stride is applied, leading to the final search as in Fig. 6 (d). The best partitioning is then $[0, 28, 70, 96]$ and marked as a red dot in Fig. 6 (b).

A comprehensive partitioning lookup table will enable efficient partitioning as in Fig. 3 (b) for effective load-balancing. For a given user context, we will interpolate and predict the best partitioning from two closest entries. Therefore, having a dense and large table would be advantageous at the cost of one-time search overhead. Our results also show that even at the 4k intervals between entries, the predicted partitioning can yield excellent TTFT (see Fig. 10).



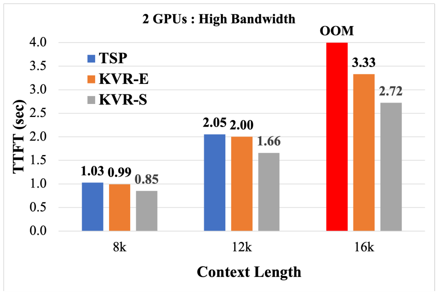
(a) 2 partitions (b) stride 8 (c) stride 4 (d) stride 2

Figure 6. Finding two partitions can be done quickly by binary search as shown in (a) where δ_1 is the variable for additional context the p_0. We can extend such binary search into hierarchical grid search for multiple processes as in (b, c, d) for the example of $C[0, 32 + \delta_1, 64 + \delta_2, 96]$.

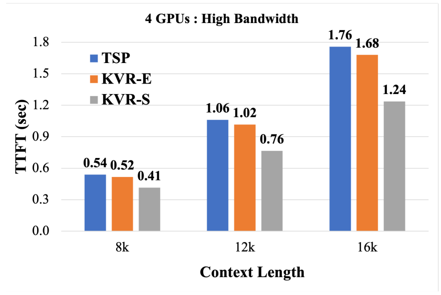


(a) Without KV-Runahead. (b) With KV-Runahead.

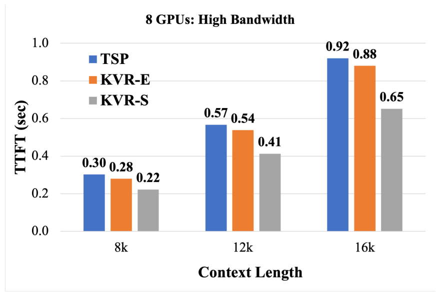
Figure 7. KV-Runahead can be easily implemented in LLM with existing KV-cache support (e.g., most of public LLMs) by simply inserting *recv/send* operations (in the blue boxes). Note that M is the causality mask.



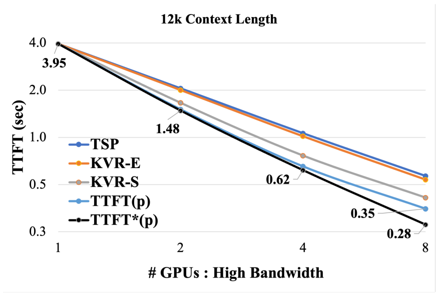
(a) **TSP** hits out-of-memory error for 16k.



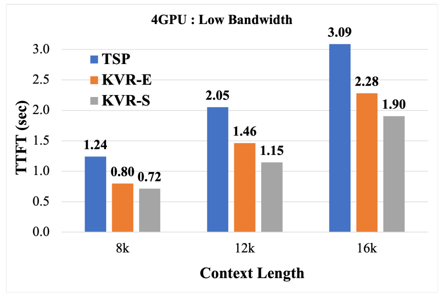
(b) **KVR-S** is 1.42× faster for 12k than **TSP**.



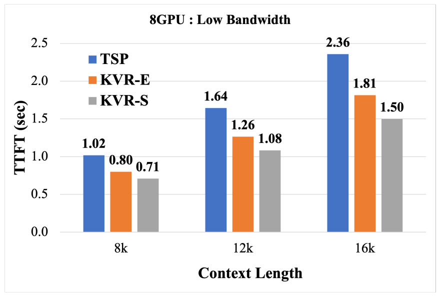
(c) **KVR-S** is 1.41× faster for 16k than **TSP**.



(d) **KVR-S** shows the best scalability.

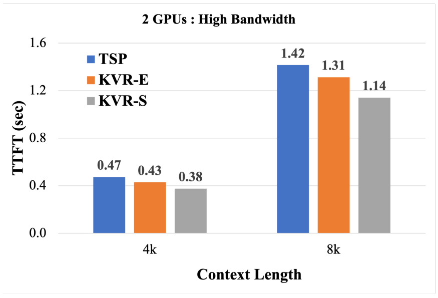


(e) **KVR-S** is 1.79× faster for 8k than **TSP**.

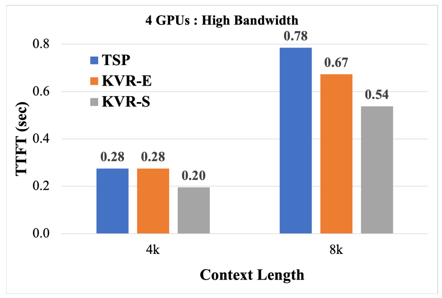


(f) **KVR-S** is 1.57× faster for 16k than **TSP**.

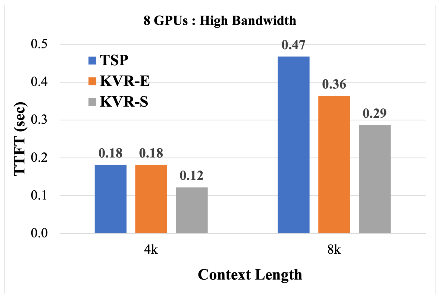
Figure 8. **Llama 7B:** while **KVR-E** already outperforms **TSP** in all the test cases, **KVR-S** further accelerates by 1.42× over **TSP** as in (a-c) with 300GB/s network. The speedups from **KVR-E** and **KVR-S** are even higher with 10GB/s network as in (e, f): 1.55× (4 GPUs and 8k) and 1.79× (4 GPUs and 12k) over **TSP**, respectively. **KVR-S** is the closest to the scalability lower bounds as in (d).



(a) **KVR-S** is 1.26× faster for 4k than **TSP**.



(b) **KVR-S** is 1.46× faster for 8k than **TSP**.



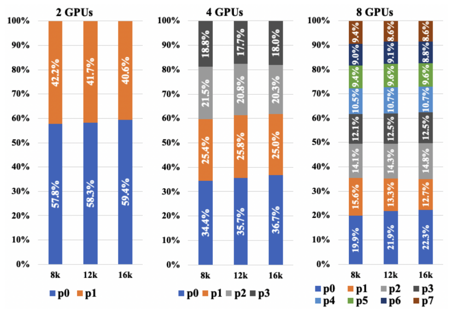
(c) **KVR-S** is 1.63× faster for 8k than **TSP**.

Figure 9. **Falcon 7B: KVR-S** offers up to 1.63× speedup over **TSP**. Since 4k context length is relatively short, the benefit from **KVR-E** is canceled out by the overhead from KV-cache waiting time and unbalanced attention compute as in (b-c). However, with load-balancing, **KVR-S** delivers 1.37× and 1.47× speedup over **TSP** with 4 and 8 GPUs, respectively, which emphasizes the context-level load-balancing.

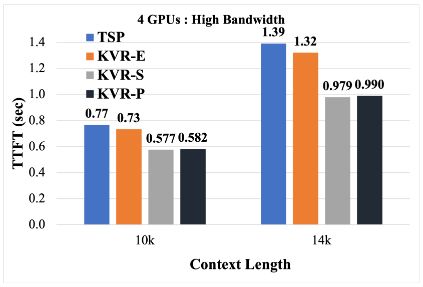
4.3. Implementation

Since KV-Runahead dual-purposes the KV-cache interface, which exists in most LLM implementations for faster subsequent token generations during the extension phase in Fig. 1 (a) (HuggingFace-Transformers), KV-Runahead is easy to implement. Fig. 7 shows the pseudocode/computational graph without and with KV-Runahead. Note that KV-cache is already in the input argument to the attention block. The only additions are two parts in the blue boxes: **a)** overwrite the KV-cache by receiving it from p_{i-1} before concatenating it with the local (K, V), and **b)** forward the updated KV-cache to p_{i+1} right after concatenation. We can make both *recv* and *send* asynchronous calls by overlapping with *qkv_proj* and *softmax* respectively, thanks to the nature of point-to-point connections. More details on the implementation and examples can be found in Appendix 5.

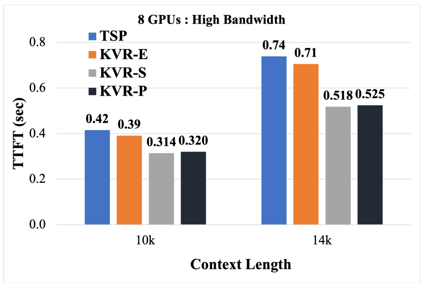
Both TSP and KV-Runahead require to have tensors in the contiguous memory space for efficient network communication, which is then about KV-cache for KV-Runahead: if KV-cache is physically fragmented, costly extra memory copy will be necessary. Therefore, the KV-cache management such as vLLM (Kwon et al., 2023; vLLM) needs to



(a) The partitioning breakdowns found by hierarchical grid search for **KVR-S** in Figs. 8 (a-c).



(b) **KVR-P** with predicted partitions is only 1.1% worse than **KVR-S**.



(c) **KVR-P** with predicted partitions is only 1.3% worse than **KVR-S**.

Figure 10. **KVR-P** with predicted partitions for 10k and 12k contexts interpolated from the searched breakdowns for Figs. 8 (a-c) has negligible TTFT degradations from **KVR-S**, supporting the fact that the proposed lookup method for context partitioning is effective.

support contiguous physical memory allocation during the prompt phase to work seamlessly with KV-Runahead.

5. Experimental Results

We used PyTorch 2.0 (Paszke et al., 2019) and NCCL 2.14 to enable KV-Runahead in Huggingface LLaMA 7B and Falcon 7B (Touvron et al., 2023; Almazrouei et al., 2023). All our experiments were done on a single node with 8× NVidia A100 GPUs, and under high (300GB/s) and low (10GB/s) bandwidth setups. Note that we turned off the high-speed CUDA-direct link (NVidia-NCCL, 2023) to configure the low bandwidth environments.

We used FP16 for the inference. We compared KV-Runahead with Tensor/Sequence Parallelization (**TSP**) (Li et al., 2023; Shoeybi et al., 2020; Patel et al., 2023). Note that KV-Runahead is applicable to any LLM with causal attention and does not alter any task accuracy. For ablation, we created a few variants of **KVR** as below.

KVR-E	with even context partitioning
KVR-S	with searched context partitioning
KVR-P	with predicted/interpolated context partitioning

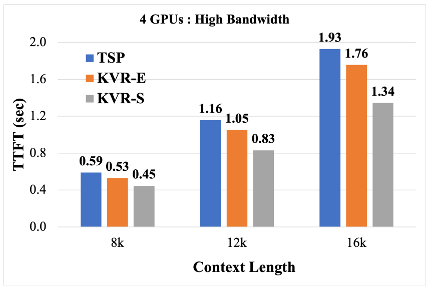
Acceleration: Our results are presented in Figs. 8 and 9 with multiple context lengths and GPU counts. From Figs. 8 (a-c), we can see **KVR-S** (even **KVR-E**) consistently outperforms **TSP**. And, **KVR-S** can deliver larger speed up (over 40%) with longer contexts and more GPUs, and the speedup gain is even higher on low bandwidth (10GB/s) network as in (e, f). Also, note that **TSP** hit the out-of-memory error for 16k contexts on 2 GPUs, apparently consuming more memory. Fig. 9 shows the similar results with 8k context lengths, but speedups are observed only with **KVR-S** for 4k context. Fig. 8 (d) compares the scalabilities of **TSP, KVR-E**, and **KVR-S** against two lower bounds: $\textbf{TTFT}(p)$ is the same as **KVR-S** without any communication (so practical lower bound), and $\textbf{TTFT}^*(p)$ is from Eq. (3) (so theoretical lower bound), which leads to the following observations:

- $\textbf{TTFT}^*(p)$ is very tight to $\textbf{TTFT}(p)$, until the non-parallelizable parts become dominant, like on 8 GPUs.
- **KVR-S** gets much closer than **TSP** to $\textbf{TTFT}(p)$.
- **KVR-S** is up to 17% away from $\textbf{TTFT}(p)$ in our tests.

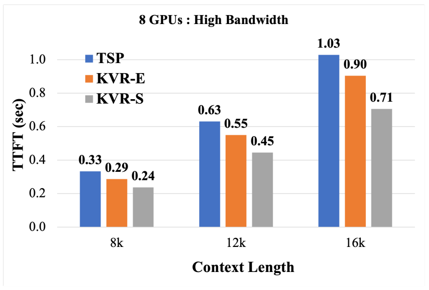
More results with other smaller/bigger LLMs and shorter/-longer contexts are available in Appendix A.

Context-level Partitioning: Fig. 10 (a) discloses the searched context partitioning for the cases in Figs. 8 (a-c). In general, we can see the earlier processes need to consume more contexts, and the later ones consume less, which implies that the wait time for the later processes is less of a concern for the configuration. We can use these breakdowns to build a partitioning lookup table, and linearly interpolate the partitionings for 10k and 14k contexts. For example, we can interpolate from the breakdowns of 8k and 12k to get the predicted partitioning for 10k on 4 GPUs, which results in $[0.350, 0.255, 0.210, 0.185]$ in terms of ratio. And. it can be done similarly for 12k user contexts as well on 4 and 8 GPUs. According to our results in Figs. 10 (b, c), even with 4k intervals, **KVR-P** with predicted partitioning from interpolation is within 1.3% of the **KVR-S** cases with the searched partition configurations and still outperforms **TSP**.

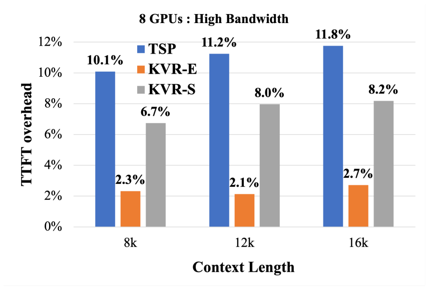
Point-to-point communication: To understand the benefit of point-to-point asynchronous communication of **KVR** over the *all-gather* operation in **TSP**, we added a noisy sidecar to generate the bidirectional network traffic between a random pair of adjacent GPUs (i.e., simulating dynamically changing non-uniform network bandwidth), averaged out multiple TTFTs for the 8k, 12k, and 16k context lengths, and then reported the results in Fig 11. We found that **KVR** is much more robust against non-uniform bandwidth among processes: while **TSP** degraded the TTFTs over 10% on average due to non-uniform effective bandwidth, **KVR** has up to 3.7% impact on TTFT, clearly demonstrating the benefits of the communication mechanism in KV-Runahead. Also,



(a) On a noisy network, **KVR-S** outperforms **TSP** even more (42.2% vs. 43.4%) than the quiet case in Fig. 8 (b).



(b) On a noisy network, **KVR-S** outperforms **TSP** even more (41.1% vs. 45.8%) than the quiet case in Fig. 8 (c).



(c) In terms of the TTFT overhead, **TSP** is most affected (up to 11.8%), and **KVR-E** is least influenced (up to 2.7%).

Figure 11. On a noisy high bandwidth network, **KVR-S** still outperforms **TSP** by even wider margin than the case in Figs. 8 (b, c), and shows high tolerance against the other noisy traffics. Yet, in terms of absolute tolerance, **KVR-E** appears to be the best.

KVR-S is tuned to the quiet environment, but still outperforms **TSP** thanks to the point-to-point communication.

6. Conclusion

In this work, we propose an efficient parallel LLM inference technique, KV-Runahead, to minimize the time-to-first-token. With the proposed techniques, we observed over 60% speedup in the first token generation over the existing parallelization schemes and higher robustness against a non-uniform bandwidth environment.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, E., Heslow, D., Launay, J., Malartic, Q., Noune, B., Pannier, B., and Penedo, G. Falcon-40B: an open large language model with state-of-the-art performance. 2023.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling, 2023.

Cho, M., Vahid, K. A., Fu, Q., Adya, S., Mundo, C. C. D., Rastegari, M., Naik, D., and Zatloukal, P. edkm: An efficient and accurate train-time weight clustering for large language models. 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 2022.

Dong, Q., Li, L., Dai, D., Zheng, C., Wu, Z., Chang, B., Sun, X., Xu, J., Li, L., and Sui, Z. A survey on in-context learning, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *arXiv*, 2023.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, 2019.

HuggingFace-TensorParallelism. https://huggingface.co/docs/transformers/v4.15.0/parallelism#tensor-parallelism.

HuggingFace-Transformers. https://huggingface.co/docs/transformers/main_classes/output.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models, 2022.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. In *Association for Computational Linguistics (ACL)*, 2023.

Lin, B., Peng, T., Zhang, C., Sun, M., Li, L., Zhao, H., Xiao, W., Xu, Q., Qiu, X., Li, S., Ji, Z., Li, Y., and Lin, W. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. *arXiv*, 2023.

Liu, Y., Li, H., Du, K., Yao, J., Cheng, Y., Huang, Y., Lu, S., Maire, M., Hoffmann, H., Holtzman, A., Ananthanarayanan, G., and Jiang, J. Cachegen: Fast context loading for language model applications, 2023a.

Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., and Chandra, V. LLM-QAT: Data-Free Quantization Aware Training for Large Language Models. *arXiv*, 2023b.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, 2021a.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021b.

NVidia-cuBLAS. https://docs.nvidia.com/cuda/cublas/.

NVidia-LLM. https://developer.nvidia.com/blog/mastering-llm-techniques-inference_optimization, 2023.

NVidia-NCCL. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html, 2023.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

Park, J., Yoon, H., Ahn, D., Choi, J., and Kim, J. OPTIMUS: optimized matrix multiplication structure for transformer neural network accelerator. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 2019.

Patel, P., Choukse, E., Zhang, C., Íñigo Goiri, Shah, A., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting, 2023.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference, 2022.

Ram, O., Levine, Y., Dalmedigos, I., Muhlgay, D., Shashua, A., Leyton-Brown, K., and Shoham, Y. In-context retrieval-augmented language models, 2023.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. In *arXiv*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

vLLM. https://docs.vllm.ai/en/latest/.

Zhang, H., Song, H., Li, S., Zhou, M., and Song, D. A survey of controllable text generation using transformer-based pre-trained language models, 2023a.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models. In *arXiv*, 2022a.

Zhang, T., Ladhak, F., Durmus, E., Liang, P., McKeown, K., and Hashimoto, T. B. Benchmarking large language models for news summarization, 2023b.

Zhang, Y., Liu, W., Wang, X., and Shaheer, M. A. A novel hierarchical hyper-parameter search algorithm based on greedy strategy for wind turbine fault diagnosis. *Expert Systems with Applications*, 202:117473, 2022b.

Network	Context	4 GPUs			8 GPUs		
		Method			Method		
	Length	TSP	KVR-S	SpeedUp×	TSP	KVR-S	SpeedUp×
Llama 7B	1k	0.107	0.097	1.11	0.117	0.098	1.19
	2k	0.111	0.100	1.11	0.117	0.103	1.14
	4k	0.20	0.17	1.17	0.12	0.11	1.14
	8k	0.54	0.41	1.30	0.30	0.22	1.36
	12k	1.06	0.76	1.39	0.57	0.41	1.37
	16k	1.76	1.24	1.42	0.92	0.65	1.41
Llama 13B	1k	0.140	0.126	1.12	0.150	0.129	1.16
	2k	0.143	0.131	1.09	0.153	0.131	1.17
	4k	0.32	0.29	1.12	0.19	0.16	1.17
	8k	0.87	0.68	1.27	0.49	0.36	1.35
	12k	1.71	1.25	1.36	0.91	0.66	1.37
	16k	2.89	2.05	1.41	1.46	1.05	1.39
Llama 30B	1k	0.21	0.20	1.08	0.23	0.19	1.19
	2k	0.28	0.26	1.06	0.24	0.20	1.19
Falcon 1B	1k	0.073	0.061	1.18	0.081	0.066	1.23
	2k	0.067	0.060	1.12	0.079	0.065	1.23
	4k	0.09	0.07	1.26	0.08	0.06	1.21
	8k	0.25	0.19	1.28	0.15	0.10	1.58
Falcon 7B	1k	0.107	0.095	1.12	0.119	0.096	1.24
	2k	0.117	0.103	1.13	0.118	0.099	1.20
	4k	0.28	0.21	1.30	0.18	0.12	1.47
	8k	0.78	0.54	1.46	0.47	0.29	1.63

Table 1. Top-1 accuracy with ImageNet1k: **KV-Runahead** outperforms other schemes with various pruning rates.

A. Additional Experiments

In this section, we present additional results with a wider range of LLMs using both long and short contexts to confirm that KV-Runahead will generalize well across a broader spectrum of LLMs. We experimented with Falcon 1B, Llama 13B, and Llama 30B, (in addition to Llama 7B and Falcon-7B from Section 5) and summarize the results in Table 1 where we can observe the followings:

- **KVR-S** consistently outperforms **TSP** for all cases across 4 and 8 GPUs over high bandwidth network.
- The speedup from **KVR-S** is less with shorter inputs (as attention is less bottlenecked).

Also, we tested Llama 7B with Multi-Query-Attention (MQA) and Group-Query-Attention (GQA) (Ainslie et al., 2023) over high bandwidth network and report the results in Table 2. MQA and GQA (Ainslie et al., 2023) are techniques to share keys and values among queries so that the attention part can be computationally more efficient. Accordingly, (K, V) projection computation costs will be reduced for **TSP** and**KVR**, benefiting both. In detail, **TSP** has lower communication costs as it has a fewer K and V matrices to *all_gather*, and **KVR** will have lower communication costs with MQA or GQA, as it leads a smaller (K, V) cache.

Compared with Multi-Head-Attention cases from Fig. 8 (b-c), overall GQA8 and MQA reduce the TTFT universally. For example, the speedup is as large as 1.22x with MQA. **KVR** demonstrates marginally better speedup gains over **TSP** with GQA8 and MQA than with MHA. For the example of 8GPU and 16k context, the speedup over **TSP** was 1.41x with MHA (see Fig. 8 (c)), but it becomes 1.48x with MQA and 1.46x with GQA.

B. Parallel Inference Benefits

The benefit of parallel LLM inference depends on the input context size (which determines the parallelization gain) and the network bandwidth (which determines the parallelization cost). To understand when **KVR** (i.e., parallel LLM inference in

Network	Context	4 GPUs			8 GPUs		
		Method			Method		
	Length	TSP	KVR-S	SpeedUp×	TSP	KVR-S	SpeedUp×
Llama 7B MQA	1k	0.109	0.102	1.07	0.117	0.101	1.17
	2k	0.107	0.102	1.05	0.120	0.101	1.18
	4k	0.18	0.14	1.23	0.12	0.10	1.18
	8k	0.49	0.37	1.33	0.26	0.18	1.44
	12k	0.98	0.70	1.41	0.51	0.35	1.45
	16k	1.65	1.16	1.43	0.84	0.57	1.48
Llama 7B GQA8	1k	0.112	0.102	1.10	0.119	0.101	1.18
	2k	0.113	0.102	1.12	0.118	0.103	1.15
	4k	0.18	0.15	1.20	0.12	0.11	1.15
	8k	0.50	0.38	1.32	0.27	0.19	1.42
	12k	1.00	0.72	1.39	0.52	0.36	1.42
	16k	1.67	1.16	1.44	0.86	0.59	1.46

Table 2. Top-1 accuracy with ImageNet1k: **KV-Runahead** outperforms other schemes with various pruning rates.

Context	base	10GB/s		1GB/s	
Length	1 GPU	2 GPU	4 GPU	2 GPU	4 GPU
1k	0.10	0.10	0.10	0.11	0.19
2k	0.24	**0.16**	**0.19**	**0.21**	0.35
4k	0.65	**0.38**	**0.36**	0.84	0.93
8k	1.95	**0.99**	**0.72**	**1.31**	2.06
12k	3.95	**1.82**	**1.15**	**2.28**	**2.30**

Table 3. Top-1 accuracy with ImageNet1k: **KV-Runahead** outperforms other schemes with various pruning rates.

method	rank/gpu	partition	size
TSP	0	Antibiotics are a	3
	1	type of medication	3
	2	used to treat	3
	3	bacterial infections	2
KVR	0	Antibiotics are a type of	5
	1	medication used to	3
	2	treat bacterial	2
	3	infections	1

Table 4. Partitioning examples for Table 5.

general) does help or does not, we experimented with Llama 7B on a low-bandwidth setup (10GB/s) and a poor-bandwidth setup (1GB/s) and report the TTFT for each case in Table 3. The bold numbers are when it is beneficial to have multi-GPU inference over single-GPU inference in terms of TTFT. One can observe the followings:

- Parallel inference is helpful only when the bandwidth is good enough OR the input context is long enough. For example, the bold numbers which indicate when it is beneficial to have multi-GPU inference over single-GPU inference in terms of TTFT form a lower triangle in the table.
- Even for parallel inference, when the bandwidth is not high enough, using more GPUs is not always helping. For the example of 2K input and 10GB/s, TTFT is 0.16sec with 2GPUs, but it gets worse into 0.19sec with 4GPUs. Such a degradation is more pronounced with 1GB/s network.

All above imply that for a given the infrastructure bandwidth, the optimal system for LLM inference can be determined based on the input size distribution of the target application. A user request needs to dynamically be assigned to a system

```
input = 'Antibiotics are a type of medication used to treat bacterial infections'

if method=='tsp':
    context = even_context_partitioning(input, rank, world_size)
elif method=='kvr':
    context = kva_context_partitioning(input, rank, world_size)
else:
    context = input

def forward(context, mask, rank, world_size, method, KV_cache=None):

    if method=='kvr' and rank>0:
        KV_cache = net_recv(rank-1)

    if method=='tsp':
        Q = q_proj(context)
        local_K = k_proj(context)
        local_V = v_proj(context)
        K, V = net_all_gather(local_K, local_V)
    else: #kvr, base
        Q = q_proj(context)
        K = k_proj(context)
        V = v_proj(context)

    if KV_cache:
        K = cat(KV_cache[0], K)
        V = cat(KV_cache[1], V)

    KV_cache = stack(K, V)

    if method=='kvr' and rank < world_size-1:
        net_send(KV_cache, rank+1)

    attn_weights = softmax (matmul(Q, K.T) + mask)
    attn_output = matmul(attn_weights, V)
    attn_output = o_proj(attn_output)

    return attn_output, KV_cache
```

Table 5. Simplified Pseudo Code with KV-Runahead Integration

with the right number of GPUs based on the optimization metric (i.e., cost, latency, utilization, and so on).

C. Pseudo Code and Example

Table 5 shows the simplified pseudo code for KV-Runahead integration into an existing transformer implementation, which also contrasts it with **TSP**. Table 4 illustrates one plausible partitioning with **TSP** and **KVR** for the example in Table 5, underscoring its difference from **TSP**.

D. Lookup Table Generation Overhead

We analytically derive the cost to precompute a partitioning lookup table (which is a one-time job). Let us assume that there are N GPUs and a C context with size , and we will pick a stride size at each level such that there are 5 values to check for each as shown in Figure 6. Let the time taken for each forward pass to measure TTFT be T.

At each level, there are $(N-1)^5$ combinations to evaluate. Once the best combination is picked, we can zoom in and repeat the evaluation for $log_{5-1}C$ levels. Therefore, the time taken to precompute the lookup table will be $T(N-1)^5 log_{5-1}C$.

For instance, if we assume $T = 1$ sec, $N = 8$, and $C = 16$k for the case in Fig. 8 (c), it would take approximately 33 hours for an entry. Moreover, each entry can be searched for in parallel, if more GPUs are available. In practice, after a few entries, we can seed the search from the interpolated context partitioning with limited scopes for the expedition.