Approximate Top-k for Increased Parallelism

Oscar Key[1] **Luka Ribar**[2] **Alberto Cattaneo**[2] **Luke Hudlass-Galley**[2] **Douglas Orr**[2]
[1]Centre for Artificial Intelligence, University College London [2]Graphcore Research
oscar.key.20@ucl.ac.uk {lukar, albertoc, lukehg, douglaso}@graphcore.ai

Abstract

The nearest neighbour search problem underlies many important machine learning applications, including efficient long-context generation, retrieval-augmented generation, and knowledge graph completion. However, computing top-k exactly suffers from limited parallelism, making it inefficient for highly parallel machine learning accelerators. By relaxing the requirement that the top-k is exact, bucketed algorithms can dramatically increase parallelism by independently computing many smaller top-k operations. We explore the design choices for this class of algorithms using both theoretical analysis and empirical evaluation on downstream tasks. Our motivating examples are sparsity algorithms for language models, which often use top-k to select the most important parameters or activations. We also release a fast bucketed top-k implementation for PyTorch.

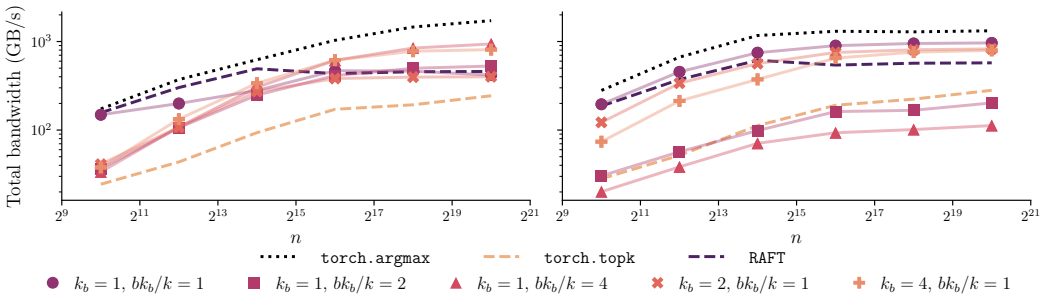

Figure 1: Our approximate top-k implementation, compared with exact top-k implementations from PyTorch and `RAFT`, and a bucketed top-k using `torch.argmax`, tested in `float32` on an H100 PCIe GPU with batch size $m = 128$. Total bandwidth is the minimum number of bytes transferred by top-k, divided by runtime. *Left:* Small fixed $k = 64$; it is faster to retrieve $k_b = 1$ element per bucket, varying the total number $b \cdot k_b$ of elements retrieved, where b is the number of buckets. *Right:* Large $k = n/4$; best to set $b \cdot k_b/k = 1$ and increase k_b.

1 Motivation and existing work

The top-k operation, which selects the k largest elements from a vector of length n, is common to most machine learning frameworks. It is an essential part of many algorithms, such as the k-nearest neighbours search (KNN), or top-k sampling in language models (Fan et al. 2018). In particular, as the capabilities and costs of state-of-the-art large language models (LLMs) grow, there is an increased requirement for efficient retrieval methods, be it from long textual sequences through in-context learning, or large document databases through retrieval-augmented generation. Moreover, sparsity methods can significantly reduce the computational costs by selecting only the parameters and activations with largest magnitudes through top-k operations, either during training, during generation (Sheng et al. 2023; Ribar et al. 2024), or both (Jayakumar et al. 2020). To this end, top-k

Workshop on Adaptive Foundation Models at NeurIPS 2024.

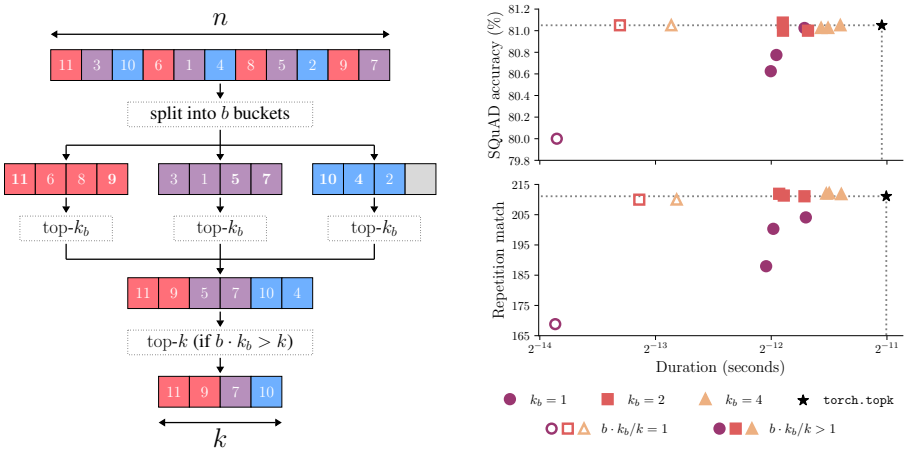

Figure 2: *Left:* An example of a bucketed top-k, with $n = 11$, $k = 4$, $b = 3$ and $k_b = 2$. In **Stage 1**, n elements are reduced to $b \cdot k_b$ elements via b independent top-k_b. An optional **Stage 2** takes final top-k. *Right:* The trade-off between top-k runtime duration and downstream task accuracy for SparQ Attention in SQuAD and a sequence repetition task (see Appendix B.2), when using different bucketed top-k settings with batch size $m = 1$. Good accuracy and speedups above $4\times$ are achieved with $k_b \in \{2, 4\}$.

may be called frequently and on long vectors. For example, Jayakumar et al. (2020) applies top-k to each weight matrix of a Transformer during all training iterations, with each matrix potentially containing millions of parameters. Similarly, Sheng et al. (2023) and Ribar et al. (2024) compute top-k over the sequence length, which might comprise tens of thousands of tokens, in each attention layer and for each token generated.

However, for larger n, computing top-k can be slow: in our experiments with Ribar et al. (2024), we found that top-k was the longest operation during attention and was not able to fully utilise compute resources. As the batch dimension is rarely large enough to saturate the processor, parallelism over n is necessary, but difficult. A common class of methods partitions the input vector into buckets, discards those buckets that fall outside the top-k, and recurses into buckets that are partially within the top-k. Parallelism is achieved by using many threads for the partitioning stage. Within this class, Alabi et al. (2012) introduce BUCKETSELECT and RADIXSELECT, as used by PyTorch (Paszke et al. 2019); see also Ribizel and Anzt (2019) and Zhang et al. (2023). Another approach is to use multiple threads to scan over the input while maintaining a shared priority queue of the k largest items (Johnson et al. 2021; Zhang et al. 2023). BITONIC TOP-K (Shanbhag et al. 2018) and DR. TOP-K (Gaihre et al. 2021) are hierarchical approaches, which split the input into buckets, perform selection in each bucket in parallel, and then, possibly recursively, merge the buckets until the final top-k is obtained. However, all these methods require substantial synchronisation between threads.

More parallelism can be achieved by computing an *approximate* top-k. This approach is particularly relevant for sparsity algorithms because top-k is generally already a heuristic, thus an additional approximation may not impact the overall performance. In this paper we consider methods which split the input into buckets (a common strategy for parallel reduction, see e.g. Triton (2024)), perform a smaller top-k_b in each bucket ($k_b \leq k$), and combine the outputs. This design is similar to the hierarchical exact algorithms, however – as the output no longer has to be exact – less or no synchronisation is required when combining the output of the buckets. An existing instance is TPU-KNN (Chern et al. 2022), implemented for TPUs as the `approx_max_k` operation in JAX and TensorFlow. Their design choices are tailored to KNN applications where $k \ll n$, for example $n \sim$ millions and $k < 100$. We find that these choices are not optimal for sparsity applications, where k might be 5% - 20% of n. While the `FAISS` library (Guzhva 2023) contains several GPU-efficient approximate KNN implementations, these mostly rely on an exact top-k. A single approximate top-k method is documented, which only supports CPU (and for which we haven't found any evaluations).

We aim to make fast approximate top-k algorithms more readily available to practitioners by investigating the design choices of bucketed top-k, and by releasing our implementation as a fast PyTorch

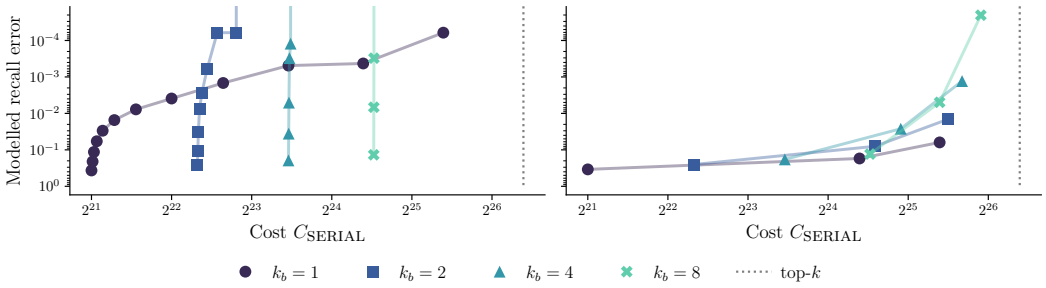

Figure 3: Theoretical trade-off curves, using the serial cost model, which computes the count of all operations executed in an abstract execution model (Appendix D), for $n = 1{,}048{,}576$ ($= 2^{20}$), with $k \ll n$ (*left*, $k = 256$) and $k \propto n$ (*right*, $k = n/8 = 131{,}072$). Points along the curves (from bottom to top) indicate increasing the $b \cdot k_b/k$ ratio, leading to a decreasing recall error. See Appendix E for the full set of trade-off curves with various cost models.

library[1]. We evaluate cost-quality trade-offs of our algorithm using theoretical analysis and empirical experiments on a sparse attention method for LLMs, LLM vocabulary sampling, and knowledge graph completion. We demonstrate substantial speedups of the top-k operation when compared to `torch.topk`, for example over $4\times$ on the sparse attention method task, with no degradation on downstream performance. However, when compared to a highly-optimised exact top-k implementation, the speedups are smaller. In the best case, for larger n, speedups of $2 - 4\times$ are still possible, but additional optimisation of our implementation is required to achieve them.

2 Algorithm design choices

We consider bucketed algorithms consisting of two stages (see Figure 2, left):

Stage 1 divide the n inputs into b interleaved buckets, select the largest k_b values in each bucket, and concatenate the $b \cdot k_b$ results;

Stage 2 if $b \cdot k_b > k$, use an exact top-k to select the k largest values.

The design parameters are $b \in \{1, \ldots, n\}$ and $k_b \in \{1, \ldots, \min(k, n/b)\}$, with the requirement that $b \cdot k_b \geq k$. TPU-KNN is an instance where $k_b = 1$ and **Stage 2** is implemented using a bitonic sort. The implementation in `FAISS` uses $b = k/k_b$, thus **Stage 2** is not required.

The quality of the approximation can be improved by increasing either b or k_b. We consider two regimes: $k \ll n$, k fixed to a small value (common in KNN applications); $k \propto n$, k is a significant proportion of n (common in sparsity applications). In $k \ll n$, we might expect that the computation time is dominated by **Stage 1**, thus our intuition is to improve quality by increasing b so that more parallelism is available. In $k \propto n$, the cost of **Stage 2** becomes significant, thus quality should be improved by increasing k_b and maintaining $b \cdot k_b = k$. Our analysis supports these intuitions, as shown in the following sections.

We could use any exact top-k algorithm in **Stage 1**. However, in our empirical evaluations we found that good approximation and efficiency can be achieved by capping $k_b \leq 4$. We can take advantage of this property to write a fast implementation that uses a small priority queue stored in registers and requires no or minimal communication between threads. See Appendix A.

3 Theoretical evaluation

As an initial evaluation of the design choices in an implementation-neutral manner, we use theoretical models to investigate the trade-off between the quality of the approximation and computational cost. We measure quality using recall, $\boldsymbol{R}(k, b, k_b) = \boldsymbol{Z}(k, b, k_b)/k$, where the random variable $\boldsymbol{Z}(k, b, k_b)$ counts the number of true top-k values returned by the approximate method, obtaining an upper

[1] `https://github.com/graphcore-research/pytorch-approx-topk`

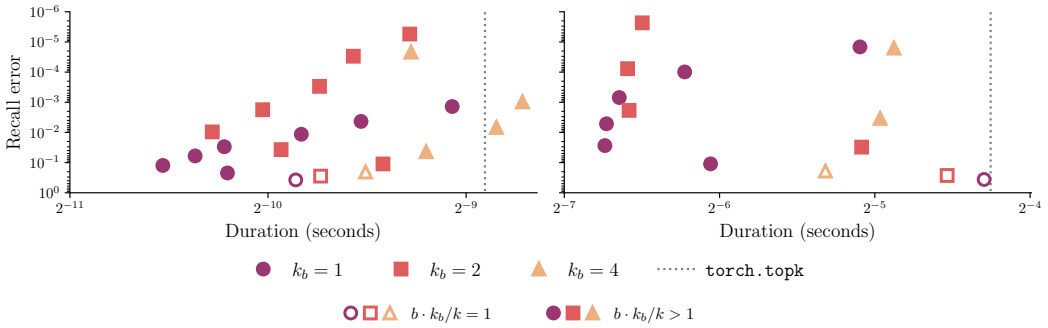

Figure 4: Bucketed top-k trade-off for LLM vocabulary sampling (*left*, $n = 128{,}256$, $k = 256$, $m = 64$) and for knowledge graph link prediction (*right*, $n = 2{,}653{,}751$, $k = 100$, $m = 128$). In both regimes, $k_b = 1$ gives peak performance, but $k_b = 2$ sacrifices some speed for sake of a lower error and is Pareto optimal when increasing b.

bound on the expected recall error, $1 - \mathbb{E}[\boldsymbol{R}]$, see Equation (1), and derive several theoretical cost models for different implementations based on operation counting, see Appendix D for details.

Figure 3 shows the trade-off between recall error and theoretical cost, as we vary b and k_b under the two regimes. The lowest point on each curve represents the case $b \cdot k_b = k$, and moving along the curve (from bottom to top) corresponds to increasing b further. Both regimes show that it is theoretically possible to achieve substantial speedups without much loss in recall. For $k \ll n$, we observe that increasing b alone is the most cost-effective way of improving recall. For $k \propto n$, larger values of $b \cdot k_b$ incur a substantial performance penalty due to the cost of **Stage 2**, thus it is better to increase recall by choosing a larger value of k_b and a smaller value of b.

4 Empirical evaluation

Our empirical evaluation consists of benchmarking the top-k runtime performance and exploring the cost-quality trade-off in downstream tasks. Further details are presented in Appendix B.

Benchmarking Figure 1 compares the memory bandwidth achieved by our implementation, under various configurations, with three baselines: `torch.argmax` (an approximate top-k implemented by bucketing the PyTorch argmax function), `torch.topk`, and the highly GPU-efficient exact method implemented in the `RAFT` library (Rapidsai 2022), which is based on AIR TOP-K and GRIDSELECT as developed by Zhang et al. (2023).

`torch.argmax` implements bucketed top-k for the case $b = k$, $k_b = 1$. If we assume that it is well optimised, it constitutes an upper bound for bucketed methods, showing huge opportunities for performance gains over `torch.topk`. The potential speedups over the highly-optimised `RAFT` are smaller, but still 2-4$\times$ for larger n. Our implementation for the case $b = k$, $k_b = 1$ exhibits good scaling, but would require further optimisation to reach the performance of the upper bound, especially for $k \ll n$. For most choices of parameters, our implementation offers substantial speedups over `torch.topk`, and can offer smaller speedups over `RAFT` for larger n.

Note that `RAFT` only supports `float32` values, thus this is the data type we use in Figure 1. Language modelling tasks commonly use lower precision data types, so we also compare the bandwidth for `bfloat16` in Figure 8, though the ranking of the methods is largely the same. The subsequent evaluations on downstream tasks are performed with `bfloat16`, thus we only compare against `torch.topk`.

Sparse generation in LLMs We consider SparQ Attention (Ribar et al. 2024), which speeds up generation for long sequences in Transformer models. Top-k is applied to the attention scores to select which values should be fetched from the KV cache. Here $k \propto n$, with $n/16 \leq k \leq n/8$, where n is the sequence length. Figure 2 shows the cost-quality curves for Llama 3 8B (Dubey et al. 2024) on two evaluation tasks. Consistent with our theoretical results, increasing k_b is the most

cost-effective way of improving task performance. In fact, for $k_b = 2$ and $b \cdot k_b/k = 1$, the cost of top-k is reduced by more than $4\times$, with almost no degradation in quality.

Small k tasks We consider two tasks where $k \ll n$. A common application of top-k in LLMs is to select the subset of most likely next tokens to sample from during generation. We again perform our evaluation on Llama 3 8B, which has a vocabulary size $n = 128{,}256$, and choose $k = 256$. Finally, we look at link-prediction on knowledge graphs, another very common application of top-k. We investigate tail predictions on PharMeBINet (Königs et al. 2022), a large biomedical knowledge graph with $n = 2.6 \times 10^6$ entities, and $k = 100$. Figure 4 shows the recall error (compared to exact top-k) for these two tasks. Consistent with the theoretical results, we observe that increasing k_b and $b \cdot k_b/k$ together can retain good recall while achieving speedups between $2\times$ and $4\times$.

5 Conclusions

Motivated by sparsity algorithms for LLMs, we have shown that additional parallelism can be introduced to the top-k operation with a bucketing approach without affecting downstream task performance. Notably, different strategies are optimal depending on the ratio k/n: sparsity applications are better served by top-k_b ($k_b > 1$) per bucket, whereas if $k \ll n$ increasing the number of buckets is more cost-effective. While substantial speedups over `torch.topk` are possible, the performance improvements are smaller when compared to the highly-optimised exact top-k in RAFT. Future work could investigate the performance of bucketed methods in distributed settings, where the top-k is performed by multiple accelerators. Here approximate bucketed methods may have a larger advantage due to the limited communication required between buckets. Finally, we hope that the PyTorch library that we release will make it easier for researchers to start using approximate top-k algorithms in their work.

Acknowledgments and Disclosure of Funding

OK acknowledges support from the Engineering and Physical Sciences Research Council with grant number EP/S021566/1.

References

Alabi, T., Blanchard, J. D., Gordon, B., and Steinbach, R. (2012). "Fast k-selection algorithms for graphics processing units". In: *ACM J. Exp. Algorithmics* 17. URL: https://doi.org/10.1145/2133803.2345676.

Cattaneo, A., Justus, D., Mellor, H., Orr, D., Maloberti, J., Liu, Z., Farnsworth, T., Fitzgibbon, A., Banaszewski, B., and Luschi, C. (2022). "BESS: Balanced Entity Sampling and Sharing for Large-Scale Knowledge Graph Completion". In: *arXiv preprint*. URL: https://arxiv.org/abs/2211.12281.

CCCL Development Team (2023). *CCCL: CUDA C++ Core Libraries*. URL: https://github.com/NVIDIA/cccl.

Chern, F., Hechtman, B., Davis, A., Guo, R., Majnemer, D., and Kumar, S. (2022). "TPU-KNN: K Nearest Neighbor Search at Peak FLOP/s". In: *Advances in Neural Information Processing Systems*. Vol. 35. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/639d992f819c2b40387d4d5170b8ffd7-Abstract-Conference.html.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. (2024). "The Llama 3 Herd of Models". In: *arXiv preprint*. URL: https://arxiv.org/abs/2407.21783.

Fan, A., Lewis, M., and Dauphin, Y. N. (2018). "Hierarchical Neural Story Generation". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 889–898. URL: https://aclanthology.org/P18-1082/.

Forsythe, G. E. (1964). "Algorithm 232: heapsort". In: *Commun. ACM* 7.6, pp. 347–349. URL: https://doi.org/10.1145/512274.512284.

Gaihre, A., Zheng, D., Weitze, S., Li, L., Song, S. L., Ding, C., Li, X. S., and Liu, H. (2021). "Dr. Top-k: delegate-centric Top-k on GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA. URL: https://doi.org/10.1145/3458817.3476141.

Gray, A. (2019). *Getting Started with CUDA Graphs*. URL: https://developer.nvidia.com/blog/cuda-graphs/ (visited on 09/13/2024).

Guzhva, A. (2023). *FAISS CPU Approximate Top-K*. URL: https://github.com/facebookresearch/faiss/blob/c418b30f756d56952d90f47b4e378985c85e608b/faiss/utils/approx_topk/approx_topk.h.

Hoare, C. A. R. (1961). "Algorithm 65: find". In: *Commun. ACM* 4.7, pp. 321–322. URL: https://doi.org/10.1145/366622.366647.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. (2020). "Open graph benchmark: Datasets for machine learning on graphs". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 22118–22133.

Jayakumar, S. M., Pascanu, R., Rae, J. W., Osindero, S., and Elsen, E. (2020). "Top-KAST: Top-K Always Sparse Training". In: *Advances in Neural Information Processing Systems*. Vol. 33. URL: https://proceedings.neurips.cc/paper/2020/hash/ee76626ee11ada502d5dbf1fb5aae4d2-Abstract.html.

Johnson, J., Douze, M., and Jégou, H. (2021). "Billion-Scale Similarity Search with GPUs". In: *IEEE Trans. Big Data* 7.3, pp. 535–547. URL: https://doi.org/10.1109/TBDATA.2019.2921572.

Königs, C., Friedrichs, M., and Dietrich, T. (2022). "The heterogeneous pharmacological medical biochemical network PharMeBINet". In: *Scientific Data* 9.1, p. 393.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in Neural Information Processing Systems*. Vol. 32.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). "SQuAD: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250*.

Rapidsai (2022). *Rapidsai/raft: RAFT contains fundamental widely-used algorithms and primitives for data science, Graph and machine learning.* URL: https://github.com/rapidsai/raft.

Ribar, L., Chelombiev, I., Hudlass-Galley, L., Blake, C., Luschi, C., and Orr, D. (2024). "SparQ Attention: Bandwidth-Efficient LLM Inference". In: *Forty-first International Conference on Machine Learning*. URL: https://openreview.net/forum?id=OS5dqxmmtl.

Ribizel, T. and Anzt, H. (2019). "Approximate and Exact Selection on GPUs". In: *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pp. 471–478. URL: https://doi.org/10.1109/IPDPSW.2019.00088.

Shanbhag, A., Pirk, H., and Madden, S. (2018). "Efficient Top-K Query Processing on Massively Parallel Hardware". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 1557–1570. URL: https://doi.org/10.1145/3183713.3183735.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. (2023). "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU". In: *International Conference on Machine Learning*. Vol. 202, pp. 31094–31116. URL: https://proceedings.mlr.press/v202/sheng23a.html.

Triton (2024). *Layer Normaliazation - Triton Documentation*. URL: https://triton-lang.org/main/getting-started/tutorials/05-layer-norm.html.

Wang, Q., Mao, Z., Wang, B., and Guo, L. (2017). "Knowledge Graph Embedding: A Survey of Approaches and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* 29.12, pp. 2724–2743.

Yang, B., Yih, S. W.-t., He, X., Gao, J., and Deng, L. (2015). "Embedding Entities and Relations for Learning and Inference in Knowledge Bases". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Zhang, J., Naruse, A., Li, X., and Wang, Y. (2023). "Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA. URL: https://doi.org/10.1145/3581784.3607062.

A Algorithm and implementation details

We release a CUDA implementation of bucketed top-k. The implementation has two modes to ensure enough parallelism is exposed: (1) 1 thread per bucket, (2) 64 threads per bucket. In the benchmarks presented in the paper, for each specific configuration of n, k, b, k_b we always select the mode, between the two, giving the best performance. The implementation also offers a heuristic to select the mode: use mode (1) if

- number of threads $\geq$ number of lanes on the device,
- or, buckets contain < 64 elements.

In mode (1), each thread selects the top-k_b values from each bucket and writes them back to memory. In mode (2), each thread selects the top-k_b values from a subsection of a bucket, the resulting values are sorted within each group of 64 threads, and the final top-k_b values are written to memory. The merge is performed using the BlockMergeSort primitive from the CUB library (CCCL Development Team 2023). This is a cheap operation because it can be performed in shared memory and only takes place once. In both modes the implementation of each thread is the same: the thread iterates over the input while inserting each value into a max priority queue of size k_b. As $k_b \leq 4$, the priority queue can be cheaply and simply implemented using insertion sort in per-thread registers.

Mode (1) is the same as the per-thread top-k described by Shanbhag et al. (2018). Mode (2) is similar to BLOCKSELECT (Johnson et al. 2021), except we can take advantage of $k_b \leq 4$ to only merge the per-thread queues once, as opposed to periodically having to do it during the computation. Note that both of these previous implementations store the queue in fast but small memories, namely the registers or shared memory, thus they are limited to values of $k < 1024$. The bucketed approach can support much larger values of k while still storing the queues in fast memories.

A.1 Bucket assignment

When splitting n inputs into b buckets ($b \leq n$), we investigate two different ways of assigning elements along the sequence to buckets: *contiguous* and *interleaved*. Let $\mathcal{B} : \{0, \ldots, n-1\} \to \{0, \ldots, b-1\}$ be the assignment function, so that input x_i is assigned to bucket $\mathcal{B}(i)$.

- In *contiguous* mode, we set $\mathcal{B}(i) := \lfloor \frac{bi}{n} \rfloor$.
- In *interleaved* mode, we set $\mathcal{B}(i) := i \pmod b$.

An ideal bucket assignment would break any correlations in the input data, so that each bucket can be expected to contain approximately the same number of top-k values as any other (uniform distribution assumption; see Appendix C). However, data is often highly correlated across the sequence, which can lead to poor recall when using contiguous assignment compared to exact top-k. We simulate this effect by repeatedly drawing a sequence of $n = 2048$ elements from a Multivariate normal distribution $\boldsymbol{x} \sim \mathcal{N}(0, \boldsymbol{\Sigma})$ such that $\Sigma_{i,j} = \rho^{|i-j|}$ with the correlation factor $\rho \in [0, 1)$. Applying the approximate bucket top-k algorithm with $k = 256$ for various k_b, we observe that, as the correlation between the elements is increased, the contiguous assignment leads to dramatic degradation in recall; on the other hand, interleaved assignment works very well in retaining recall, no matter the degree of correlation (Figure 5, left). In cases when permuting the data can be done cheaply, the performance can be alternatively retained by first shuffling the data (Figure 5, right).

All the real-world data that we tested in this work contains significant correlation between the ordered values – for example, Figure 6 showcases the performance degradation when using contiguous assignment for the LLM downstream task with SparQ attention. On the other hand, due to the contiguity of memory accesses, the contiguous assignment showcases runtime improvements compared to the interleaved assignment, as can be seen in Figure 7 using the same top-k settings as in the LLM downstream task experiments. Nevertheless, due to the data correlation issues, we assign the elements to buckets in the *interleaved* fashion throughout our experiments.

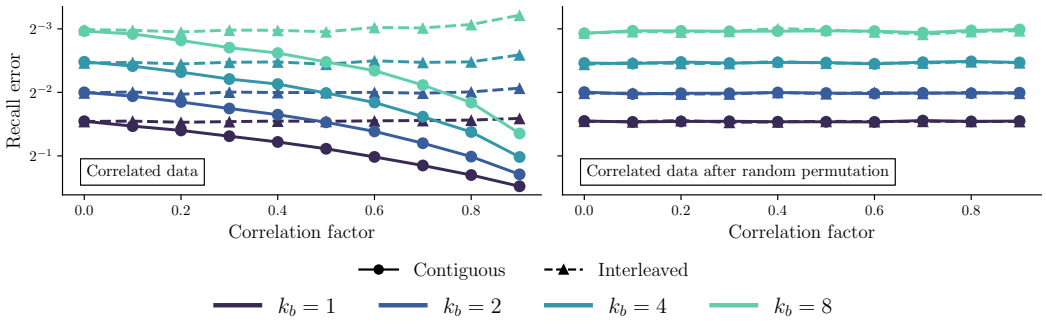

Figure 5: Comparison of achieved recall vs. exact top-k using interleaved and contiguous bucket assignments on normally distributed correlated data. *Left:* As the correlation between data points is increased, contiguous assignment showcases significant recall degradation, even as k_b is increased. *Right:* When the correlated data is randomly permuted, the contiguous assignment achieves the same recall as the interleaved assignment.

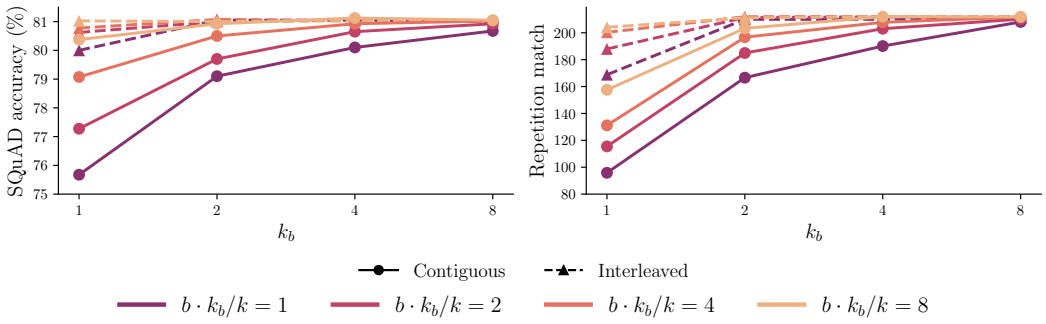

Figure 6: Comparison of LLM downstream task performance using interleaved and contiguous bucket assignment with SparQ attention. As textual sequences are highly correlated, there is a significant performance degradation when using contiguous assignment. *Left:* SQuAD task. *Right:* Repetition task.

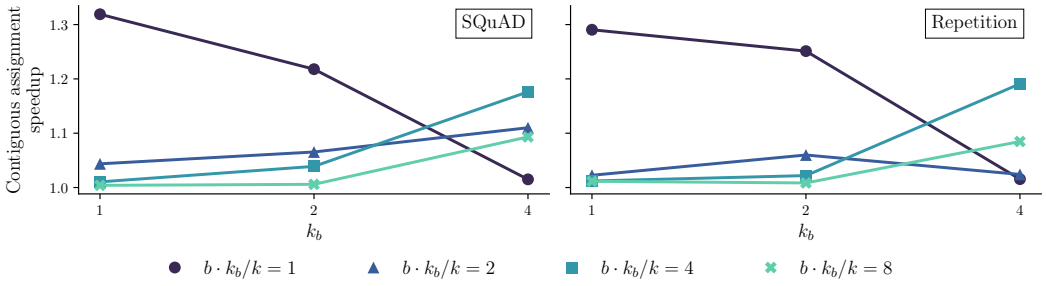

Figure 7: Runtime comparison of contiguous vs interleaved assignment on SQuAD and Repetition tasks, indicating the speedup of the contiguous assignment.

B Experiment details

B.1 Speed benchmarking

Benchmark results were all generated on a H100 PCIe GPU, using PyTorch 3.12 and CUDA 12.1. We generate a tensor of i.i.d. values from a unit normal distribution, of shape $(16, m, n)$, then create a CUDA graph (Gray 2019) for a tight loop of 16 iterations of the algorithm, with each iteration

consuming data from a different slice of the input. This helps to prevent small problems from fitting in L2 cache, while measuring only the kernel of interest. We execute this graph 16 times without measurement, for warmup, then for 16 timed executions, measuring each individually using CUDA events. Before each graph execution, but outside the scope of timing, the random inputs are regenerated.

We find that this setup yields very stable timings. Over all the benchmarking runs presented in this work, maximum ratio of standard error to mean duration is 5%, and the average ratio is 0.1%.

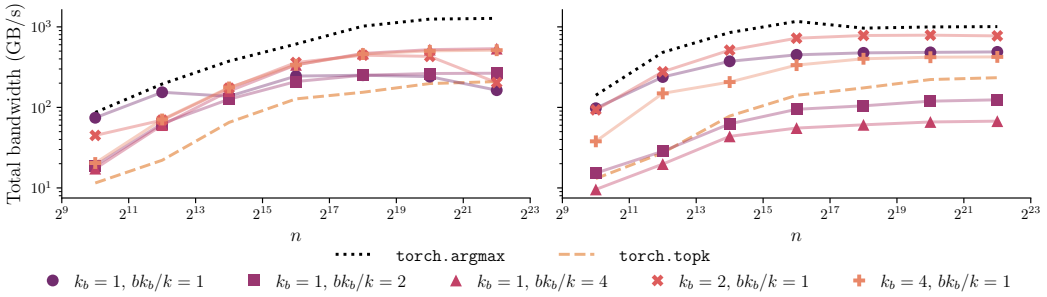

Figure 8: Bandwidth comparison for `bfloat16` values, in the same configurations used for Figure 1. *Left:* Small fixed $k = 64$. *Right:* Large $k = n/4$.

B.2 Downstream task details

B.2.1 LLM generation with SparQ Attention

SparQ Attention (Ribar et al. 2024) is an attention-approximating algorithm for speeding up long-sequence transformer generation. It consists of two main steps. Firstly, an approximate dot-product of the current query and the cached keys is conducted, generating approximate scores across the sequence length. Secondly, top-k approximate scores are selected, and attention is approximated using only these k key-value pairs.

We examine the effect of using approximate top-k for selecting top key-value pairs within **Step 2** of the SparQ Attention algorithm. The top-k algorithm is applied to the approximate attention scores, and k is generally chosen to be $1/16$ - $1/8$ of the sequence length. We test the effectiveness of the approximation by evaluating its effect on the downstream task performance on two representative tasks taken from (Ribar et al. 2024): question answering on SQuAD dataset and text repetition. For the former, we use a modified version of SQuAD v1.1 (Rajpurkar et al. 2016) where each provided context (required to answer the question) is augmented with seven additional unrelated contexts sampled from other examples. For the latter, the model is prompted to repeat a piece of text from its context, and the task performance is measured by the length of the exact match of the answer in characters. We perform both evaluations using the Llama 3 8B model (Dubey et al. 2024).

B.2.2 Knowledge graph completion

Knowledge Graphs (KGs) encode domain knowledge in the form of subject-relation-object triples (h, r, t), with h, t in the set $\mathcal{V}$ of graph nodes (also known as *entities*) and r the edge type (or *relation type*). As KGs are often incomplete, inferring missing links is a classical ML application, which in turn can be framed in terms of triple completion, e.g. finding the most likely tail for a query $(h, r, ?)$. Knowledge Graph Embedding models learn embeddings for entities and relation types, which are used as inputs to a real-valued scoring function $f(\mathbf{h}, \mathbf{r}, \mathbf{t})$ whose output is interpreted as a likelihood score for the triple (h, r, t). Finding the set of k most likely completions for a query $(h, r, ?)$ becomes then a top-k problem: $k\text{-argmax}_{t \in \mathcal{V}} f(\mathbf{h}, \mathbf{r}, \mathbf{t})$. We refer to Wang et al. (2017) for more details.

In this task $n = |\mathcal{V}|$ can be very large (in the order of tens of millions for real-world KGs), while k is typically fixed and small ($k \leq 100$ in most practical applications and model benchmarks, see Hu et al. (2020)). For our experiments, we use PharMeBINet (Königs et al. 2022), a biomedical KG with $2{,}653{,}751$ entities, and consider the task of predicting tails on a random held-out test set of $55{,}000$ triples. As KGE model, we take the basic – but widely used (Hu et al. 2020) – DistMult scoring

function $f(\mathbf{h}, \mathbf{r}, \mathbf{t}) = \langle \mathbf{r}, \mathbf{h}, \mathbf{t} \rangle$ (B. Yang et al. 2015). Training and inference are performed using the BESS distribution framework (Cattaneo et al. 2022).

C Recall model

As in Section 3, let $\boldsymbol{Z}(k, b, k_b)$ be the number of true top-k values returned by the approximate method, and $\boldsymbol{R}(k, b, k_b) := \boldsymbol{Z}(k, b, k_b)/k$ be the recall. For the sake of modelling, we first assume that the input data is uniformly distributed, $k \geq k_b$ and $k_b \ll n/b$. Then:

$$\mathbb{E}[\boldsymbol{Z}(k, b, k_b)] = \mathbb{E}[\boldsymbol{Z}(k-1, b, k_b)]\phi_k + (\mathbb{E}[\boldsymbol{Z}(k-1, b, k_b)]+1)(1-\phi_k) = \mathbb{E}[\boldsymbol{Z}(k-1, b, k_b)]+1-\phi_k,$$

where we denote by ϕ_k the probability that the k-th top value falls in a bucket already containing at least k_b of the top-$(k-1)$ values. As top values are equally likely to be contained in any of the buckets, $\phi_k = 1 - F(k_b - 1; k-1, 1/b)$, where $F(x; m, p) := \Pr(X \leq x)$ is the cumulative function of the binomial distribution $X \sim \mathrm{Binom}(m, p)$. Moreover, $\mathbb{E}[\boldsymbol{Z}(k_b, b, k_b)] = k_b$, hence by recursion the **expected recall error** is:

$$1 - \mathbb{E}[\boldsymbol{R}(k, b, k_b)] = 1 - \frac{1}{k}\left(k_b + \sum_{i=k_b}^{k-1} F\left(k_b - 1; i, \frac{1}{b}\right)\right). \tag{1}$$

We point out that, when $k_b = 1$:

$$\mathbb{E}[\boldsymbol{R}(k, b, 1)] = \frac{1}{k}\left(1 + \sum_{i=1}^{k-1}\left(\frac{b-1}{b}\right)^i\right) = \frac{b}{k}\left(1 - \left(\frac{b-1}{b}\right)^k\right)$$

differs from the expected recall computed in the TPU-KNN paper (Chern et al. 2022): indeed, the authors of ibid. assume that only the values (among the top-k) that do not share a bucket with any other ones contribute to $\boldsymbol{Z}(k, b, 1)$, which is an overly-pessimistic hypothesis (if two top-k values fall in the same bucket, one of them is still retrieved by the approximate method; equivalently, $\boldsymbol{Z}(k, b, 1)$ coincides with the number of unique buckets containing a top-k value).

When dropping the assumption $k_b \ll n/b$, we observe that if a bucket contains an excess of top-(k-1) values compared to its peers, it will be less likely to contain the k-th top value too (differently from the previous analysis where all buckets were always considered equiprobable, based on the assumption that only a negligible number of values needed to be retrieved from each of them). As a consequence, in this more general setting Equation (1) can still be seen as a upper bound for the expected recall error.

If we also remove the hypothesis of uniformly distributed inputs, the worst-case scenario happens when all true top-k values are concentrated in the smallest possible number of buckets, that is $\lceil \frac{bk}{n} \rceil$. For this worst-case, the recall error is:

$$1 - \boldsymbol{R}_{\text{wc}}(n, k, b, k_b) = 1 - \frac{1}{k}\left(\left\lfloor \frac{bk}{n} \right\rfloor k_b + \min\left(k_b, k \pmod{\frac{n}{b}}\right)\right). \tag{2}$$

From experiments on real-world data, however, we find that the recall error achieved by our implementation (using interleaved bucket assignment to break correlations in the sequence, see Appendix A.1) is always sufficiently closely aligned with Equation (1), while Equation (2) gives too pessimistic bounds; therefore, we do not include the latter in our discussion.

D Cost models

To evaluate the efficiency of the approximate top-k algorithm in a general setting, independent of a specific hardware platform and implementation, we devise a set of abstract *cost models* for exact and approximate top-k algorithms. These serve as a guide — simplifying the complexities of real hardware and software platforms, they provide an understanding of the algorithmic factors influencing performance and the different regimes that we can expect to see in practical benchmarks. After describing the models in this section, we consider their agreement against GPU benchmarks in D.3.

Each cost model contains multiple algorithms, and the overall cost $C_{\mathcal{M}}(n, k, m)$ from a model $\mathcal{M}$ is the minimum across the algorithms $\mathcal{A}$ supported by that model, $C_{\mathcal{M}} = \min_{\mathcal{A}} C_{\mathcal{M},\mathcal{A}}$. Cost models for exact top-k are summarised in Table 1.

Given a cost model for exact top-k, we can form a model for the approximate top-k algorithm described in Section 2 as

$$\tilde{C}_{\mathcal{M}}(n, k, m, b, k_b) = C_{\mathcal{M}}(n/b, k_b, m \cdot b) + [b \cdot k_b > k] \cdot C_{\mathcal{M}}(b \cdot k_b, k, m),$$

where $[b \cdot k_b > k]$ is 1 when $b \cdot k_b > k$ and 0 otherwise. We assume here that costs are additive between **Stage 1** and **Stage 2**.

Table 1: Cost models and algorithms for exact top-k.

Model	Algorithm	Cost	Residual
Basic	-	$m \cdot n \cdot (\log_2 k + 1)$	$+\mathcal{O}(m \cdot k)$
Serial	PRIORITYQUEUE	$m \cdot n \cdot (3k - 1)$	$+\mathcal{O}(m)$
	RADIXSELECT	$m \cdot n \cdot (4 \log_2 n + 4)$	$+\mathcal{O}(m \log n)$
Parallel	SCANMAX	$k \cdot (2 \log_2 n + 3)$	$+\mathcal{O}(1)$
	RADIXSELECT	$\log_2 n \cdot (2 \log_2 n + 16)$	$+\mathcal{O}(1)$

D.1 Basic cost model

The basic cost model uses the asymptotic bound for top-k based on a linear scan through the data, maintaining a min-heap (Forsythe 1964) of the top-k of the prefix that has been scanned. Note that there are asymptotically faster top-k algorithms such as QUICKSELECT (Hoare 1961), however these are ill-suited to execution on highly parallel hardware, which is the focus of this work. Since this top-k algorithm is $\mathcal{O}(n \log k)$ and we note that top-1 is readily achieved with an $\mathcal{O}(n)$ scan through the data, we set the cost $C_{\text{BASIC}} = m \cdot n \cdot (\log_2 k + 1)$.

D.2 Serial and Parallel cost models

In these cost models, we go beyond asymptotic evaluation, since the combination of multiple algorithms based on the minimum cost depends on the constant factors associated with each algorithm. To do this, we describe a simple abstract execution model, with the following assumptions:

- Iterating through the input data linearly is free.
- Fixed-offset addressing is free.
- Bounds checking is free.
- If statements are free, but all branches are taken.
- Ignore small "residual" cost terms.
- Operations that cost 1: `==`, `<`, `>`, `+`, `&`, `|`, `<<`, `not`, `=` (arithmetic, logical, assignment).
- Operations that cost 2: `+=`, `&=`, `|=`, `^=` (read-modify-write).

We then define two cost model variants:

- *Serial*, where the total cost is the sum of all operation costs.
- *Parallel*, where the total cost is total cost of operations executed by a machine with infinite parallel threads.

We evaluate two algorithms for each cost model, one appropriate for small-k, another for large-k, with a rough correspondence to the priority queue and RADIXSELECT implementations that we have evaluated. Costs are outlined in Table 1 and explained using illustrative Python code in Figures 9 to 12.

D.3 Analysing cost models

We compare the scaling trends predicted by our abstract cost models and the practical benchmarking results observed for `torch.topk`, which is based on RADIXSELECT and our CUDA implementation of a PRIORITYQUEUE using insertion sort.

```
def topk_insertion(data, k):
    topk = sorted(data[:k])                                  # (ignore)
    for x in data:                                           # * n
        if x > topk[0]:                                      # | +1
            topk[0] = x                                      # | +1
        for j in range(1, k):                                # | * (k-1)
            if topk[j-1] > topk[j]:                          # | | +1
                topk[j-1], topk[j] = topk[j], topk[j-1]      # | | +2
    return topk
```

Figure 9: PRIORITYQUEUE (serial), with insertion sort, cost $m \cdot n \cdot (3k-1) + \mathcal{O}(m)$. The first sort can be merged into the loop.

```
def topk_radix_select(data, k):
    # Find kth value
    kth_value, mask, count_gt = 0, 0, 0
    for r in range(31, -1, -1):          # * log(n)
        r_mask = 1 << r
        kth_value |= r_mask
        mask |= r_mask
        count_1 = 0
        for x, _ in data:                # | * n
            if x & mask == kth_value:    # | | +2
                count_1 += 1             # | | +2
        if count_gt + count_1 < k:
            kth_value ^= r_mask
            count_gt += count_1

    # Collect topk
    topk = [None] * k
    i = 0
    for x in data:              # * n
        if x[0] >= kth_value:   # | +1
            topk[i] = x         # | +1
            i += 1              # | +2
    return topk
```

Figure 10: RADIXSELECT (serial), cost $m \cdot n \cdot (4 \log_2 n + 4) + \mathcal{O}(m \log n)$. Note that we assume a key length of $\log_2 n$, to uniquely identify n elements; in practical scenarios, the key length is separate from n. If the k^{th} element may be tied, a second "collect" step may be necessary to ensure that only these tied elements are discarded.

Figure 13 shows the performance of `torch.topk` as n and k vary, versus the corresponding cost model of Table 1. As expected, RADIXSELECT performance does not depend strongly on k, and behaves like the parallel cost model for small problems (small $m \cdot n$), then like the serial cost model for larger problems. This is to be expected of a GPU, which has resources to support a large finite number of parallel threads. When the available parallelism is under-utilised, implementations scale as if the number of threads is infinite. As the resources are exhausted, implementations scale based on the total amount of work.

We perform a similar exercise for our implementation of PRIORITYQUEUE in Figure 14. The serial cost model is generally a good predictor of runtime, but real runtime scales better than expected with k. This may be because the kernel is often bandwidth-bound w.r.t. reads from memory. In this case, there may be no additional cost to computing $k = 2$ versus $k = 1$, since the computation time is hidden by memory access.

```
def scan_argmax(data):
    a = list(range(len(data)))                    # +1
    for i in range(log2(len(data))):              # * log(n)
        a = [a[j]                                 # | +1
             if j+2**i >= len(data) \
                or data[a[j+2**i]] < data[a[j]]   # | +1
             else a[j+2**i]
             for j in range(len(data))]
    return a[0]

def topk_scan_max(data, k):
    data = data.copy()
    topk = [None] * k
    for i in range(k):          # * k
        j = scan_argmax(data)   # | +2*log(n) + 1
        topk[i] = data[j]       # | +1
        data[j] = (-inf, 0)     # | +1
    return topk
```

Figure 11: ScanMax (parallel), cost $k \cdot (2\log_2 n + 3) + \mathcal{O}(1)$.

```
def scan_cumsum(data):
    s = data.copy()                                      # +1
    for i in range(log2(len(s))):                        # * log(n)
        s = [s[j] + (s[j-2**i] if j-2**i >= 0 else 0)    # | +2
             for j in range(len(s))]
    return s

def topk_radix_select_parallel(data, k):
    # Find kth value
    kth_value, mask, count_gt = 0, 0, 0
    for r in range(31, -1, -1):                                  # * log(n)
        r_mask = 1 << r                                          # | +1
        kth_value |= r_mask                                      # | +2
        mask |= r_mask                                           # | +2
        count_1s = [x & mask == kth_value for x, _ in data]      # | +2
        count_1 = scan_cumsum(count_1s)[-1]                      # | +2*log(n) + 1
        if count_gt + count_1 < k:                               # | +2
            kth_value ^= r_mask                                  # | +2
            count_gt += count_1                                  # | +2

    # Collect topk
    in_topk = [x >= kth_value for x, _ in data]
    offset = scan_cumsum(in_topk)                                # +2*log(n) (+ 1)
    topk = [None] * k
    for i in range(len(data)):                                   # (in parallel)
        if in_topk[i]:
            topk[offset[i] - 1] = data[i]
    return topk
```

Figure 12: RadixSelect (parallel), cost $\log_2 n \cdot (2\log_2 n + 16) + \mathcal{O}(1)$.

E Trade-off curves

Combining our theoretical recall and cost models (Appendices C and D), we have made a broad sweep of trade-off curves to help guide hyperparameter selection of k_b and b. These are shown in Figures 16 to 18 for each cost model in turn. They motivate the following observations:

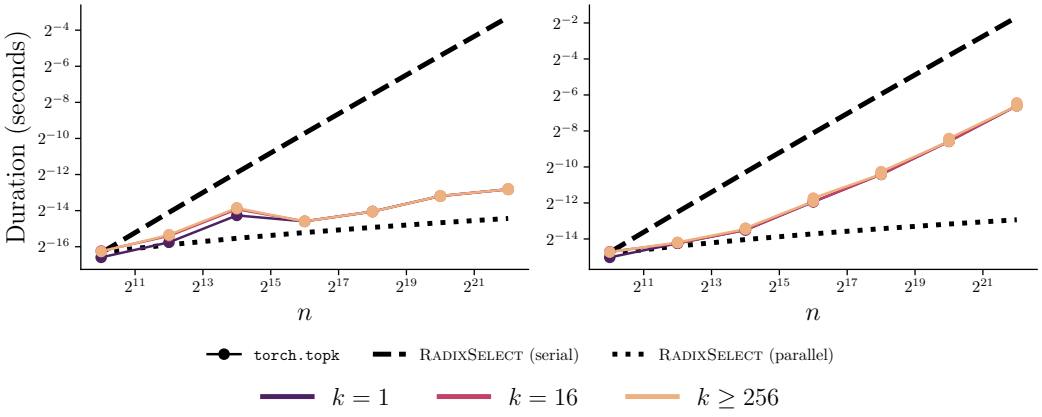

Figure 13: PyTorch top-k runtime as a function of n, k. *Left:* Batch size $m = 1$. *Right:* $m = 256$. The serial and parallel cost models have been aligned to match PyTorch at $n = 2^{10}$. Practical runtime does not depend strongly on k, and follows the parallel cost model when batch size m and input length n are small, but then trend toward the serial cost model.

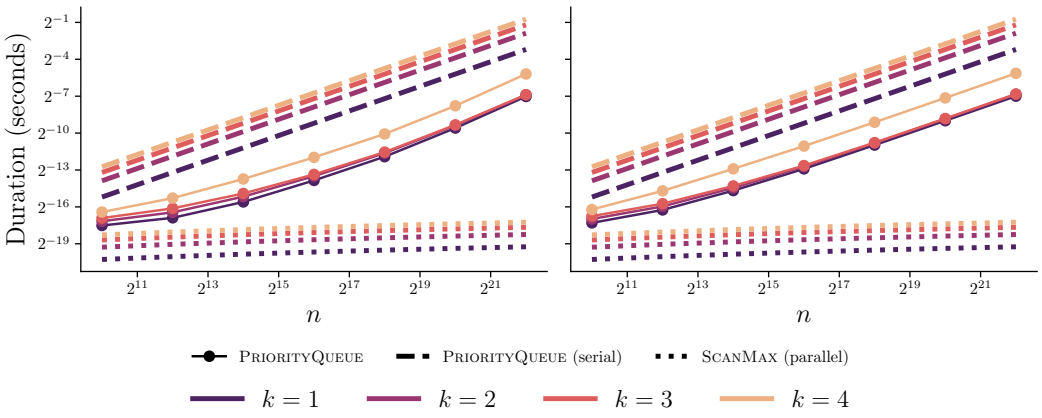

Figure 14: Our priority queue top-k runtime as a function of n, k. *Left:* Batch size $m = 1$. *Right:* $m = 256$. The serial and parallel cost models have been shifted vertically for sake of visual tracking, since they do not directly predict wall-clock time. Runtime scaling with k is better than the predicted linear scaling (Table 1), especially for $k \in \{2, 3\}$. At very small $n \cdot m$, runtime scaling with n is close to the parallel model, but otherwise it scales as per the serial model.

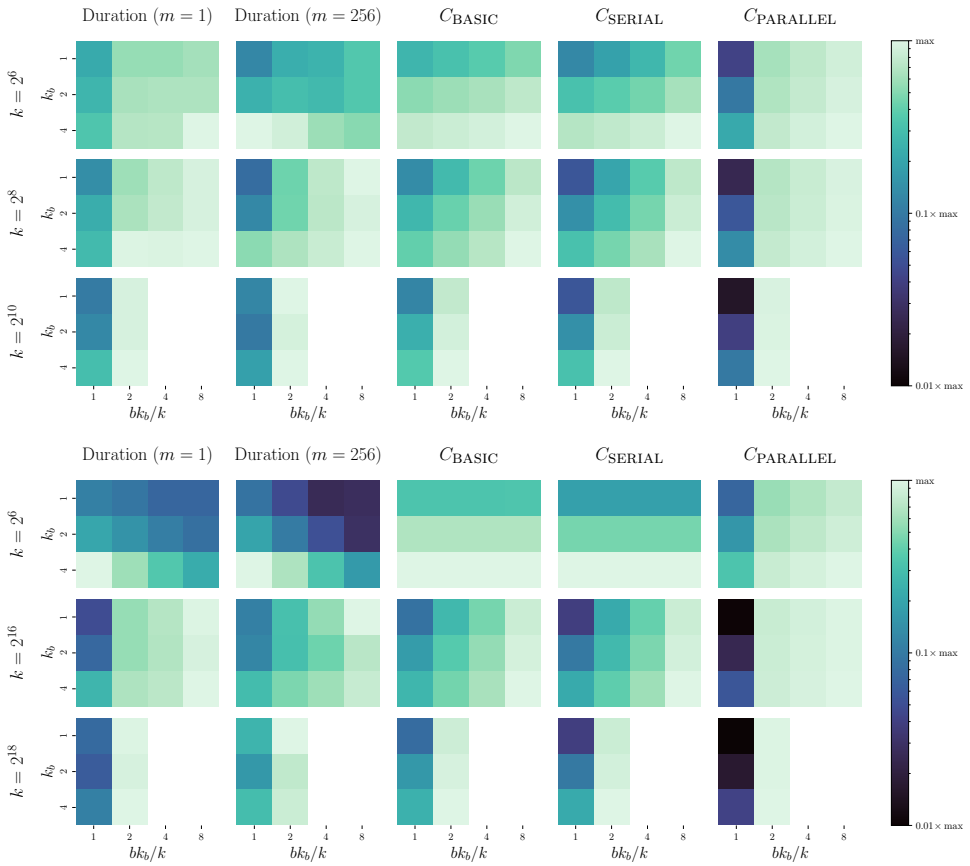

Figure 15: A comparison of the relative runtime of our approximate top-k implementation, against cost models C. *Top:* $n = 2^{12}$. *Bottom:* $n = 2^{20}$. These show that the basic or serial cost models are generally a reasonable match to practical runtime. One exception is that they generally underestimate the cost of having **Stage 2** (the step between $b \cdot k_b/k = 1$ and 2). The other notable exception is for $n = 2^{20}, k = 2^6$, where the runtime profile is inverted, with smaller $b \cdot k_b/k$ taking longer than larger $b \cdot k_b/k$.

- Low recall error and large speedups are available in the small-k regime, roughly when $k/n \leq 1/64$ (in both serial and basic cost models).
- In the small-k regime, it is best to control error by increasing the number of buckets such that $b \cdot k_b > k$, keeping $k_b = 1$. To achieve very low error, it eventually becomes optimal to increase $k_b > 1$ too.
- In the large-k regime, it is better to control error by increasing the per-bucket $k_b > 1$, while maintaining $b \cdot k_b = k$. This is because **Stage 2** can be very expensive, and it is often better to avoid it entirely.
- Substantial speedups are available in the parallel cost model, as long as **Stage 2** is not required. In this cost model, increasing $k_b > 1$ should always be prioritised over increasing $b \cdot k_b > k$.

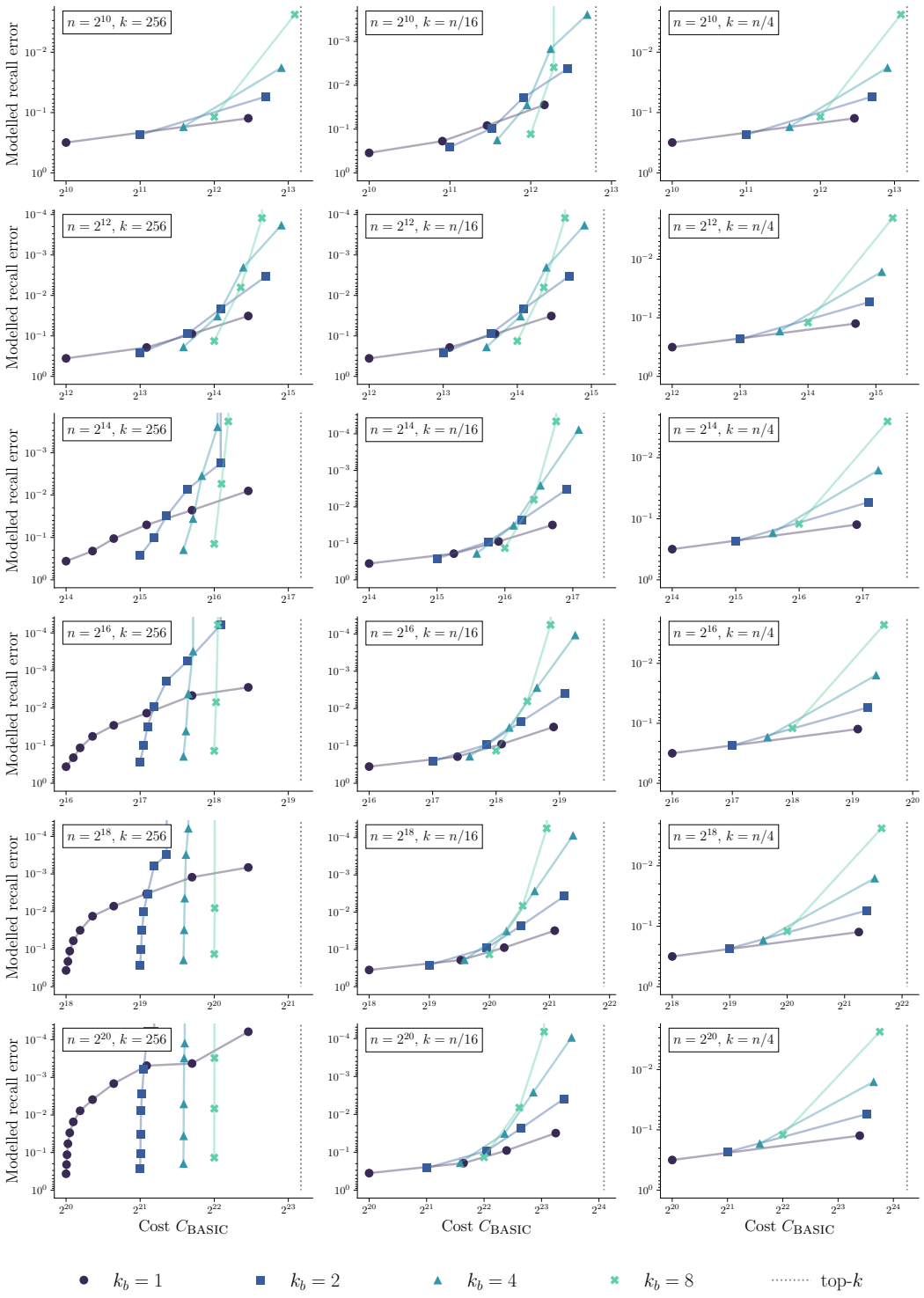

Figure 16: Theoretical trade-off curves under the basic cost model, as n, k, k_b and $b \cdot k_b/k$ are varied. Columns show three different regimes based on k. Points along the curves correspond to different $b \cdot k_b/k$ ratios. *Left:* small fixed $k = 256$. *Center:* moderate $k = n/16$. *Right:* large $k = n/4$. Rows show increasing n. Figures in the bottom left demonstrate the small-k regime, where $k_b = 1$ should be preferred, and b increased to reduce error. Figures in the center and right show that for large k, it is often better to increase $k_b > 1$. In general, the dependence on n is limited, given the ratio k/n.

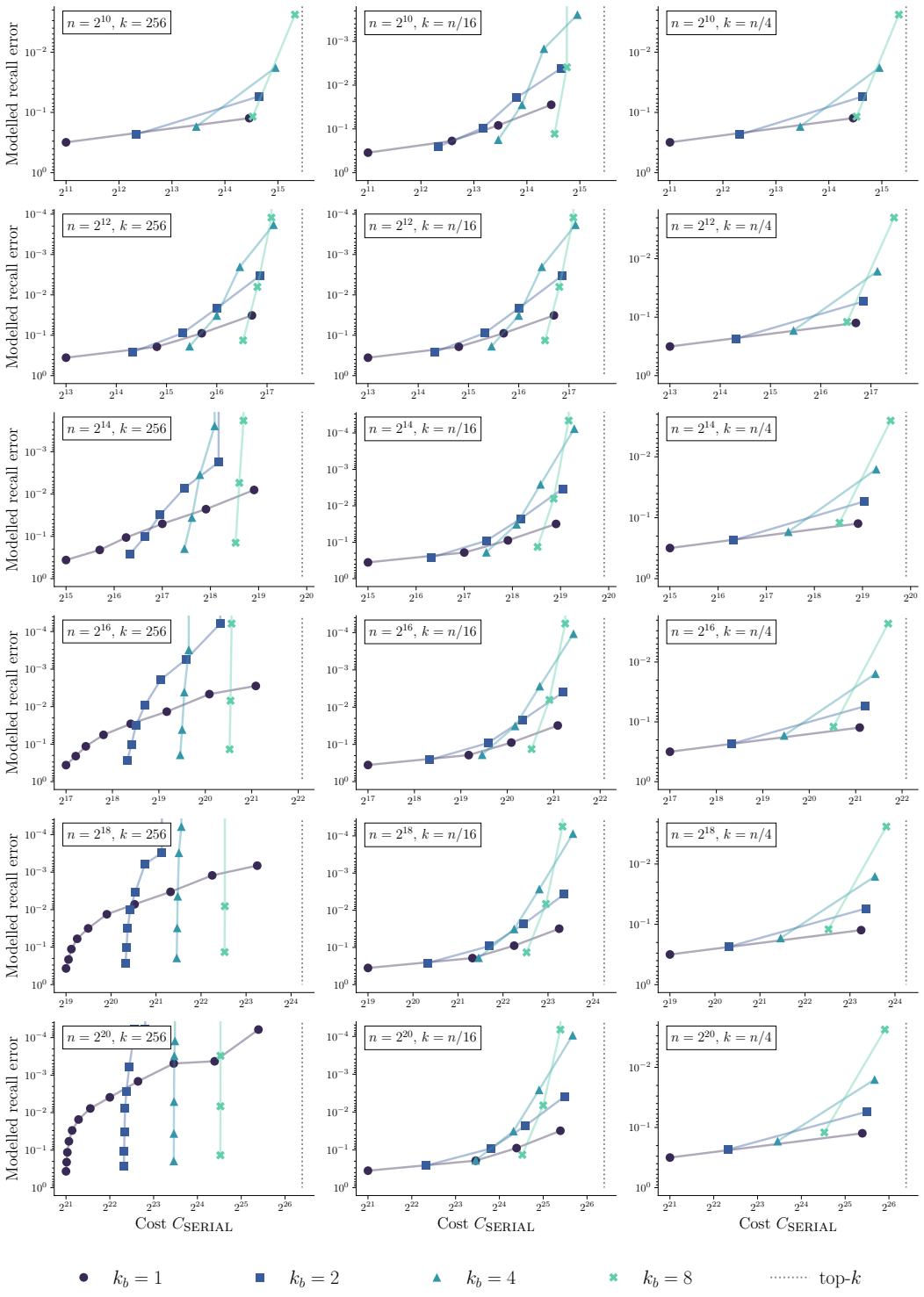

Figure 17: Theoretical trade-off curves under the serial cost model, as n, k, k_b and $b \cdot k_b/k$ are varied. We observe that approximate top-k in this model scales similarly to that of the basic model; all observations made for Figure 16 apply here too.

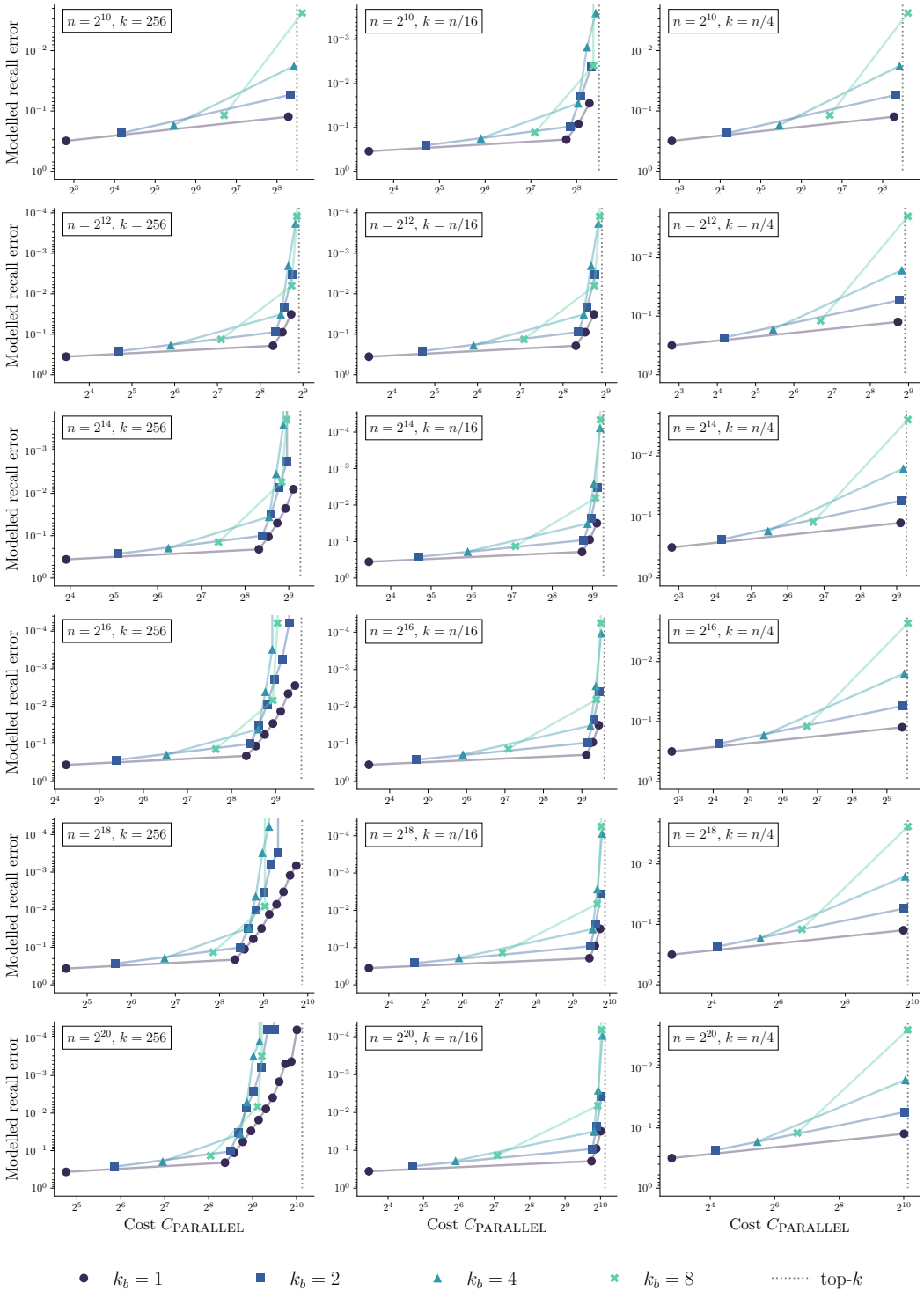

Figure 18: Theoretical trade-off curves under the parallel cost model, as n, k, k_b and $b \cdot k_b/k$ are varied. The parallel cost model places a premium on the execution of **Stage 2**, which is avoided when $b \cdot k_b = k$ (for the leftmost points of each line). It is therefore optimal under this model to use k_b to control error, even in the small-k regime (left column). We note that it is hard to achieve substantial speedups with very low error under the parallel cost model.