FASTER APPROX. TOP-K: HARNESSING THE FULL POWER OF TWO STAGES

Yashas Samaga [1 *] **Varun Yerram** [2] **Spandana Raj Babbula** [2] **Prateek Jain** [2] **Praneeth Netrapalli** [2]

ABSTRACT

We consider the Top-K selection problem, which aims to identify the largest-K elements from an array. Top-K selection arises in many machine learning algorithms and often becomes a bottleneck on accelerators, which are optimized for dense matrix multiplications. To address this problem, Chern et al. (2022) proposed a fast two-stage *approximate* Top-K algorithm: (i) partition the input array and select the top-1 element from each partition, (ii) sort this *smaller subset* and return the top K elements. In this paper, we consider a generalized version of this algorithm, where the first stage selects top-K' elements, for some $1 \leq K' \leq K$, from each partition. Our contributions are as follows: (i) we derive an expression for the expected recall of this generalized algorithm and show that choosing $K' > 1$ with fewer partitions in the first stage reduces the input size to the second stage more effectively while maintaining the same expected recall as the original algorithm, (ii) we derive a bound on the expected recall for the original algorithm in Chern et al. (2022) that is provably tighter by a factor of 2 than the one in that paper, and (iii) we implement our algorithm on Cloud TPUv5e and achieve around an order of magnitude speedups over the original algorithm without sacrificing recall on real-world tasks.

1 INTRODUCTION

Identifying the top-K elements in an array is an essential building block of many algorithms. Beyond its common applications in performing Maximum Inner Product Search (MIPS) or K-Nearest-Neighbor (KNN) based tasks (Chern et al., 2022; Li et al., 2023), it has recently become particularly important for optimizing training and inference efficiency of large foundation models. Top-K operation has been used in large language models to exploit the sparsity in classification logits (Samaga B L et al., 2024), MLP blocks (Liu et al., 2023; Samaga B L et al., 2024; Alizadeh et al., 2024), and attention mechanisms (Roy et al., 2021; Madaan et al., 2023). It is also used for retrieval augmented generation (Lewis et al., 2021; Borgeaud et al., 2022), sampling tokens (Shen et al., 2024), mixture-of-experts (He, 2024), and for accelerating distributed training (Shi et al., 2019; Ruan et al., 2023).

Given the scale of these models, the training and inference are typically carried out on accelerators such as TPUs and GPUs. However, computing Top-K on these devices can be expensive and often becomes a bottleneck. On TPUv5e and A100, finding the top-2% of the hidden activations of Gemma 2 9B's (Team et al., 2024) feedforward blocks[1] during training using `jax.lax.top_k` takes 27× and 4.8× longer, respectively, than the corresponding matrix multiplication that generated those activations. Ideally, the time spent on Top-K must be negligible compared to the matrix multiplications.

As a workaround, there has been increasing use of *approximate* Top-K algorithms in foundation models. Research indicates that these models are generally robust to such approximations (Samaga B L et al., 2024; Key et al., 2024).

Chern et al. (2022) introduced a hardware-friendly approximate Top-K algorithm that works in *two stages*. In the first stage, the input array is partitioned into a fixed number of buckets, and the top-1 element from each bucket is selected. In the second stage, these top-1 elements are sorted, and the first K elements are returned. Chern et al. (2022) quantify the approximation error in terms of the *expected recall*, which is defined as the proportion of the actual top-K elements retrieved in the first stage averaged over all permutations of the inputs. They derive a closed-form expression that relates the expected recall to the number of buckets. Using this formula, they then *choose the number of buckets* in the first stage that is sufficient to maintain a *user-specified average recall target*. This algorithm was implemented in `jax.lax.approx_max_k` for TPUs at the time of writing this article. In the earlier example of finding the top-2% of the hidden activations, this algorithm still takes 9× more time than the matrix multiplication on TPUv5e.

*Work done while at Google DeepMind. [1]University of Washington, Seattle [2]Google DeepMind. Correspondence to: Yashas Samaga <syashas@cs.washington.edu>.

[1]This involves an einsum contraction between a 3D tensor of shape `[batch_size, seqlen, model_dims]` and a 2D weight matrix of shape `[model_dims, hidden_dims]`, contracting along the `model_dims` axis (i.e., `"bsm,mh -> bsh"`). Top-K is then applied along the `hidden_dims` axis.

As we explain in Sections 2 and 5, in most settings, the second stage is the bottleneck, since the first stage computation is relatively inexpensive and efficiently parallelizable. In fact, when the task requires finding the top-K elements in each column or row of a matrix product, the first stage can be "fused" (Snider & Liang, 2023) with the matrix multiplication, effectively hiding much of its cost. Therefore, to improve the efficiency of this algorithm, we need to reduce the number of elements sorted in the second stage without sacrificing the expected recall, and while ensuring that the first stage doesn't become too expensive – this is exactly the contribution of this paper.

We achieve this by generalizing the first stage of the approx. Top-K algorithm of Chern et al. (2022) to select top-K' elements from each bucket (where $K' < K$) instead of restricting to top-1. This increases the total number of elements sorted in the second stage to $B \cdot K'$, where B is the number of buckets. However, our key result shows that for a large set of values of K, array size N and recall targets, we can reduce the number of buckets B *sufficiently* that the optimal number of elements to sort in the second stage ($B \cdot K'$) is achieved by some $K' > 1$, all while ensuring that the first stage does not become the bottleneck.

Our main contributions are as follows. **Theoretically**, we derive an expression that connects K', K and B to the expected recall. Using this expression, we find the parameters K' and B for our algorithm that meet the user-specified average recall target. While the full potential of this result is realized by choosing $K' > 1$, it is interesting to note that even for $K' = 1$ (i.e., the setting of Chern et al. (2022)), our bound on the number of buckets is provably a factor of 2 tighter than that in Chern et al. (2022).

Empirically, we efficiently implement our improved algorithm ($K' > 1$) for TPUs using Pallas and demonstrate an order of magnitude improvement in latency on TPUv5e chips compared to the algorithm proposed in Chern et al. (2022). We provide two implementations: (i) an unfused version that executes the two stages separately as two kernels (see Section 2 for an explanation of kernel), and (ii) a matmul-fused version that fuses the first stage with the matmul operation (see Sections 2 and 3 for more details on fusion and related matmuls). In the earlier example of identifying the top 2% of the hidden activations, our implementations for $K' = 4$ make the Top-K step 24× faster than `jax.lax.approx_max_k` and reduce the time taken by approximate Top-k to less than the corresponding matrix multiplication, resulting in an overall speedup of 6.7×. Our implementations for TPUs are provided in the Appendix. For an unfused implementation of the same algorithm for GPUs, we refer you to the concurrent work by Key et al.

Paper organization: Section 2 provides a brief introduction to the background material on the organization of compute resources and the execution of programs on accelerators. Section 3 outlines the precise problem setup. In Section 4, we describe the original algorithm by Chern et al. (2022) in detail and highlight its features and limitations. In Section 5, we formally state our algorithm (Section 5.1), provide a theoretical analysis (Section 5.2), and discuss the details of the implementation (Section 5.3). In Section 6, we report our findings on the effectiveness of our algorithm in reducing the number of elements that must be sorted in the second stage (Section 6.1), followed by results on runtime performance of our implementation on TPUs (Section 6.2 for the unfused implementation and Section 6.3 for the matmul-fused implementation).

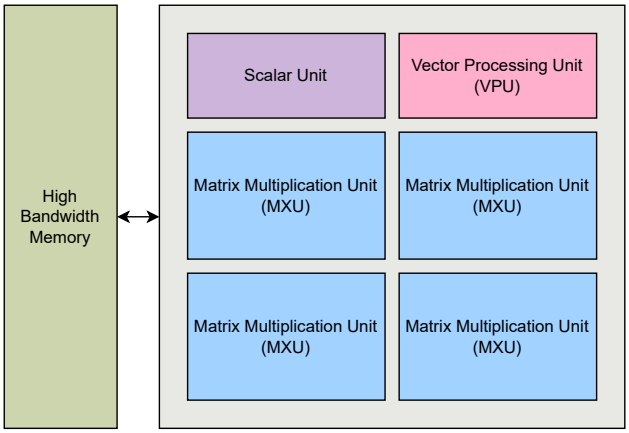

Figure 1. **Overview of the TPUv5e subsystems.** The TPUv5e chip features four Matrix Multiplication Units (MXUs) dedicated to matrix-matrix multiplications alongside a Vector Processing Unit (VPU) that performs general vector operations such as activations and softmax. It also has a scalar unit for calculating memory addresses, managing control flow, and tasks of similar nature.

2 Background

2.1 Organization of compute on accelerators

Compute resources in most accelerators are distributed across several distinct subsystems, each specialized for different types of operations (see Figure 1). On TPUs, the vast majority of the compute is concentrated in two compute subsystems: Matrix Multiply Unit (MXU) and Vector Processing Unit (VPU) (Norrie et al., 2020; Google Cloud, 2024b). MXUs account for more than 95% of the compute FLOPS (Chern et al., 2022; Google Cloud, 2024b), and therefore often only MXU-bound programs reach peak FLOPS utilization. Similarly, on GPUs, the compute resources are primarily spread across two subsystems: TensorCores for matrix multiplications and CUDA cores for scalar/vector computations. Like TPUs, most of the FLOPS are concentrated in TensorCores (NVIDIA, 2024).

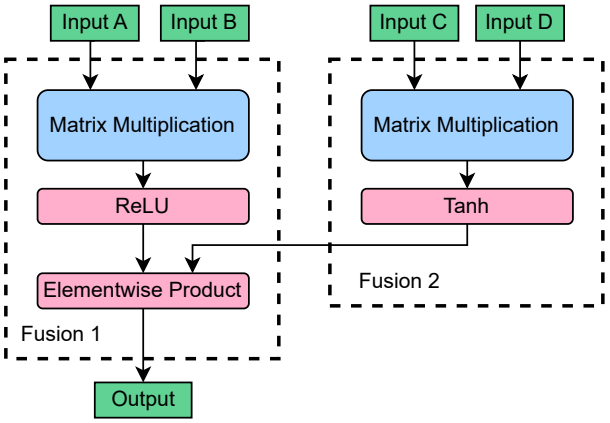

Figure 2. **Decomposing a program into smaller subprograms.** In this example, a program has been broken down into two subprograms with each containing several elementary operations. The subprograms are executed in an order that satisfies their dependencies, with Fusion 2 executing before Fusion 1. In Fusion 2, the scalar/vector units apply the $\tanh$ function on the output chunks as they arrive from the matrix unit before writing the final outputs to the memory. Similarly, in Fusion 1, the ReLU and the elementwise product execute on the scalar/vector units while the matrix multiplication executes on the matrix units.

2.2 Kernels and fusions

Programs for accelerators are typically decomposed into a series of smaller subprograms, known as kernels, which are executed atomically, and possibly concurrently, in some order on the hardware. Each subprogram may consist of many elementary operations and can simultaneously use all subsystems. For example, in the case of a matrix multiplication followed by bias addition and activation, the entire computation can be carried out in a single "fusion" (Snider & Liang, 2023; Google Cloud, 2024a), where the outputs from the matrix units are immediately processed by the scalar/vector units (for bias addition and activation) before writing to the memory. This minimizes the overhead associated with launching and terminating kernels, avoids multiple round-trips to the memory, and allows for more effective simultaneous use of the different compute resources. See Figure 2 for an illustration that summarizes this section.

2.3 Ridge-point analysis

Depending on the kernel, different subsystems of an accelerator are utilized to varying degrees, with one often becoming the bottleneck that dictates the overall runtime of the kernel. To accurately model performance and identify the bottleneck, we need to quantify the capabilities of each subsystem in the accelerator and how the kernel utilizes them.

We quantitatively characterize an accelerator's performance by measuring the peak throughput of each of its subsystems. For example, we measure the peak memory bandwidth for each memory subsystem and the peak operations per second for each compute subsystem. Depending on the subsystems utilized and their extent of saturation, we may choose to model only the relevant subsystems. For example, if a subsystem is not used or contributes negligibly to the runtime, we may omit it.

For this section, we restrict ourselves to TPUs and focus on three key subsystems: HBM memory, VPU and the MXU. We define the following parameters to quantify the throughput of each subsystem:

- β: maximum HBM bandwidth in bytes per second.
- γ: maximum number of VPU operations per second.
- π: maximum number of MXU operations per second.

Similarly, we characterize a kernel by measuring its utilization of each subsystem in its lifetime.

- M: number of bytes transferred to/from the HBM.
- O_{VPU}: number of operations executed on the VPU.
- O_{MXU}: number of operations executed on the MXU.

Since the kernel can utilize all subsystems simultaneously, the total runtime of the kernel is determined by the subsystem that requires the most time to complete its work[2]. Therefore, we can estimate the total runtime of the kernel as:

$$\text{runtime} = \max\left(\frac{M}{\beta}, \frac{O_{\text{VPU}}}{\gamma}, \frac{O_{\text{MXU}}}{\pi}\right) \tag{1}$$

The bottleneck is the subsystem corresponding to the largest argument to $\max(\dots)$ in Equation 1. For example, a kernel is considered *memory-bound* if the memory subsystem cannot feed data fast enough to keep the compute subsystems busy, i.e., $\frac{M}{\beta} \geq \max(\frac{O_{\text{VPU}}}{\gamma}, \frac{O_{\text{MXU}}}{\pi})$. To minimize overall runtime, we must address the bottleneck subsystem until it is no longer the limiting factor.

A corollary of this model is that increasing utilization of the non-bottleneck subsystems may *not* necessarily increase the kernel's runtime. To aid in such comparisons, we can define "ridge points" of Equation 1, which are configurations where the runtime of any two subsystems are equal. For example, we can estimate the maximum number of VPU operations that can be performed per MXU operation to remain MXU-bound as $\frac{\gamma}{\pi}$. Since the MXU has much higher throughput,

[2]In some cases, there may be dependencies between subsystems that may cause the dominating subsystem to stall. However, in practice, most kernels do not suffer significantly from such dependencies and saturate at least one of the subsystems.

i.e., $\pi \gg \gamma$, it is often more convenient to use the number of VPU operations per d-dimensional dot product on the MXU, i.e., $\frac{\gamma}{\left(\frac{\pi}{2d}\right)}$. This reformulation helps keep the ratio as a small and interpretable integer. We can similarly define quantities such as $\frac{\gamma}{\left(\frac{\beta}{4}\right)}$ to denote the number of VPU operations that can be performed per 4-bytes of data transferred to/from the HBM. These ridge points help us understand the balance between different subsystems and easily reason how changes in a subsystem's utilization can impact the overall performance. For a list of values of these quantities on different accelerators, see Table 1.

3 Problem Setup

Given a matrix $W \in \mathbb{R}^{n \times d}$ and a vector $x \in \mathbb{R}^d$, the task is to approximately find the K largest elements of $y := Wx \in \mathbb{R}^n$.

Expected recall: For a given y, let U represent the set of actual top-K elements, and let V represent the top-K elements returned by our approximate algorithm. We define expected recall as the expected fraction of true top-K elements retrieved by the algorithm, assuming that the top-K elements are placed randomly and uniformly in y:

$$\mathbb{E}\left[\text{recall}\right] = \mathbb{E}\left[\frac{|U \cap V|}{|U|}\right]$$

Objective: The objective is to *minimize the time* required for this operation while *maximizing expected recall*. Specifically, we aim to expand the Pareto frontier for the trade-off between latency and expected recall objectives.

4 The Original Algorithm

Chern et al. (2022) designed an approximate Top-K algorithm that operates in two stages. In the first stage, the input array is partitioned by grouping elements separated by a fixed stride into buckets, and the top-1 element of each bucket is gathered to form a *smaller* array. In the second stage, this array is sorted using bitonic sort, and the top K elements are returned. The first stage reduces the size of the input array for the *expensive* second stage which improves performance. Mistakes occur when multiple top-K elements fall into the same bucket, as only one is selected and the rest are discarded. However, the number of buckets can be increased to reduce the likelihood of such collisions sufficiently to meet the recall target. Figure 3 illustrates the algorithm.

The algorithm accepts `recall_target` as a parameter, and the required number of buckets is calculated using a closed-form expression that relates the number of buckets to the expected recall for random inputs:

$$\#\text{buckets} \geq \frac{1}{1 - \mathbb{E}[\text{Recall}]^{\frac{1}{K-1}}} \approx \frac{K-1}{1-r}$$

The input often results from a matrix multiplication, e.g., maximum inner-product search (MIPS) or Top-K on key-query logits in attention. The first stage of the algorithm, which executes on the scalar/vector units, can often be fused with the preceding matrix multiplication, which executes on the matrix units. Hence, the fused first stage may incur little to no additional cost as it utilizes the otherwise idle scalar/vector compute units while the matrix units are busy.

To design their algorithm and implementation, they take a principled approach by modeling accelerator performance similar to the model described in Section 2.3. Their first stage uses a fixed budget of three operations per element to track the top-1 element (and its index) of each bucket. We argue that this leaves compute resources underutilized in many cases:

1. The analysis in their paper focuses on matrix multiplications with 128-dimensional dot products. However, we frequently work with larger dimensions where the available scalar/vector compute is nearly $\frac{\text{dims}}{128}$ times higher than the numbers estimated in their paper.
2. Even for 128-dimensional dot products, the first stage may not saturate the scalar/vector units on all hardware platforms.
3. In memory-bound computations, there is more scalar/vector compute available than would be expected from a matrix-multiplication-bound computation.

The additional compute available enables more sophisticated algorithms for the first stage, potentially improving the recall with fewer elements to process in the second stage. A more expensive first stage may still yield overall gains if gains in the second stage outweigh the increased cost of the first stage. Based on these insights, we generalize their algorithm to more *flexibly* utilize the available compute by selecting top-K' elements from each bucket instead of just the top-1.

We refer you to a concurrent work by Key et al. on implementing the same algorithm on GPUs and for a discussion of different Top-K algorithms. The differences between their work and ours are discussed in Appendix A.2.

5 Method

We describe our algorithm in Section 5.1 and provide an analysis in Section 5.2. In Section 5.3, we discuss the key ideas in our implementation of the algorithm.

Table 1. **Peak throughput and ridge-points of different subsystems in accelerators.** The γ for TPUv4 was taken from Chern et al. and the γ for TPUv5e was estimated by timing VPU-bound kernels (see Appendix A.1). All other quantities are in the public domain.

DEVICE	β	γ (TFLOP/S) FP32	π (TFLOP/S) BF16	$\frac{\gamma}{(\frac{\pi}{256})}$ VECTOR OPS PER 128-D DOT	$\frac{\gamma}{(\frac{\beta}{4})}$ VECTOR OPS PER 4 BYTES
A100 PCIE	1.935 TB/S	19.5	312	≈ 16	≈ 40
H100 SXM	3.35 TB/S	67	1,979	≈ 8	≈ 80
TPUV4	1.2 TB/S	4.3	275	≈ 4	≈ 14
TPUV5E	819 GB/S	≈ 6.14	197	≈ 8	≈ 30

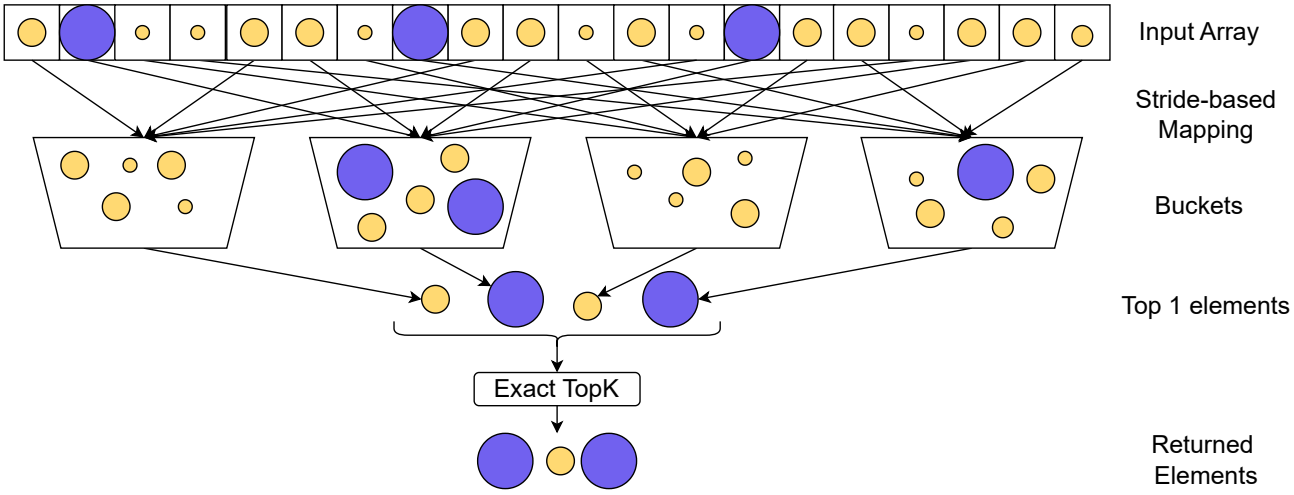

Figure 3. **Illustration of a two-stage approximate Top-K algorithm.** This example demonstrates the process of finding the approximate top three elements from an array of twenty elements using the algorithm by Chern et al. (2022). Ten buckets are required to guarantee an expected recall of 85%, but we use only four for illustration purposes. The size of the balls indicates their value, and the top three balls have been colored blue for visual clarity. The first stage groups elements separated by a fixed stride of four into buckets and selects the top-1 element from each bucket. An exact Top-K algorithm is applied on the selected elements to obtain the final result. In this example, two of the three actual top balls map to the same bucket, and one is dropped, resulting in an approximation error.

5.1 Algorithm

Given an array $A = [a_1, a_2, \ldots, a_N]$, number of buckets B, number of top elements to select per bucket K', number of top elements to find K, the algorithm proceeds as follows:

1. Partition A into buckets $G_1, G_2, \ldots, G_B$ by grouping elements separated by a stride of B into a bucket.

$$G_i \coloneqq \{a_{i+jB} \mid j \in \mathbb{Z}, i + jB \leq N, j \geq 0\} \quad \text{for} \quad i = 1, 2, \ldots, B$$

2. For each bucket G_i, select the top-K' elements.

$$T_{K'}(G_i) \coloneqq \text{Top-K'}(G_i)$$

3. Merge the selected elements to form the selected subset.

$$A_{\text{selected}} \coloneqq \bigcup_{i=1}^{B} T_{K'}(G_i)$$

4. Sort A_{selected} and return the first K elements:

$$T_K^{\text{approx}}(A) = \text{sorted}(A_{\text{selected}})[:K]$$

5.2 Analysis

Consider a scenario in which we have N balls, K of which are special balls, and B buckets. The N balls are evenly distributed in the B buckets. To model the distribution process, we can randomly order all the balls and then partition them into buckets: the first $\frac{N}{B}$ balls go to the first bucket, the next $\frac{N}{B}$ balls go to the second bucket, etc. In the context of our algorithm, the N balls correspond to the input elements, K special balls represent the top-K elements, and B buckets here correspond to the "buckets" in the algorithm.

Let X_b be a random variable that denotes the number of special balls in the bucket b. Approximation errors occur when more than K' special balls are placed in the same bucket. The total number of excess collisions is given by the sum of excess special balls in each bucket.

$$\mathbb{E}\left[\text{Excess-collisions}\right] = \mathbb{E}\left[\sum_{b=1}^{B} \max(0, X_b - K')\right] = \sum_{b=1}^{B} \mathbb{E}\left[\max(0, X_b - K')\right]$$

There exists a joint probability distribution that governs the set of X_b that satisfies the constraint that the total number of special balls in all buckets sums up to K, i.e., $\sum_{b=1}^{B} X_b = K$. However, the marginals X_b are all identically distributed as $X_b \sim \text{Hypergeometric}(N, K, \frac{N}{B})$. This arises from the fact that the distribution of special balls in the first bucket must be the same as in all other buckets by symmetry, and it is easy to see that the distribution of special balls in the first bucket must follow $\text{Hypergeometric}(N, K, \frac{N}{B})$. This is sufficient to simplify further:

$$\mathbb{E}\left[\text{Excess-collisions}\right] = B \times \mathbb{E}\left[\max(0, X_0 - K')\right]$$

The number of excess collisions is related to the recall as:

$$\mathbb{E}\left[\text{Recall}\right] = 1 - \frac{\mathbb{E}[\text{Excess-collisions}]}{K}$$

In Appendix A.3, we verify the accuracy of Monte Carlo evaluations of this expectation against the recall obtained from the simulated runs of the algorithm. Theorem 1 derives an algebraic expression for this expectation.

Chern et al. (2022) model their algorithm as randomly distributing K balls in B buckets, and relate it to the standard birthday problem. Based on this model, they derive a bound on the expected recall and invert the expression to obtain a formula for the number of buckets. They ignore the constraint that the number of balls in each bucket cannot exceed $\frac{N}{B}$, and they only count the non-colliding balls as correctly retrieved, even though one of the colliding balls is always correctly retrieved in a bucket with collisions. In Theorem 1, we derive a bound on the number of buckets based on our model that is provably tighter than theirs by at least a factor of two. We verify the quality of our bounds in Appendix A.4 and show that it closely approximates the exact expression with high fidelity.

Theorem 1. *Suppose N balls are randomly distributed into B buckets $G_1, \cdots, G_B$, each getting N/B balls. The recall of the top-K' balls across all the B buckets with respect to the top-K balls overall is given by: $\mathbb{E}[\text{Recall}] = 1 - \frac{B}{K} \times \sum_{r=K'+1}^{\min(K,N/B)} (r - K') \frac{\binom{K}{r}\binom{N-K}{\frac{N}{B}-r}}{\binom{N}{\frac{N}{B}}}$. Specifically, for $K' = 1$, and a target recall factor of at least r, the bound below implies that $B = \frac{K}{2\left(1-r+\frac{K}{2N}\right)}$ suffices to guarantee the target recall r.*

Remark: Note that for $K' = 1$, our bound above is a factor of 2 tighter than that in Chern et al. (2022).

Proof. (Proof of Theorem 1) Consider an arbitrary subset $S \subseteq \{1, \cdots, N\}$ such that $|S| = K$. Let $S_j := S \cap G_j$ for $j = 1, \cdots, B$. We now want to bound the number of elements in S_j greater than K' for some given K'.

$$m_j := \mathbb{E}\left[\max\left(0, |S_j| - K'\right)\right] = \sum_{r=K'+1}^{\min(K,N/B)} (r - K') \frac{\binom{K}{r}\binom{N-K}{\frac{N}{B}-r}}{\binom{N}{\frac{N}{B}}},$$

where each term in the summation refers to having $|S_j| = r$, $\binom{K}{r}$ refers to choosing r elements out of S, $\binom{N-K}{\frac{N}{B}-r}$ refers to the number of subsets where $\frac{N}{B} - r$ elements in G_j are chosen from outside of S, and $\binom{N}{\frac{N}{B}}$ refers to the total number of possible subsets that G_j can take. Finally, the recall (i.e., the expected number of elements in S eventually captured by the output of our algorithm) is given by:

$$\mathbb{E}\left[\text{Recall}\right] = 1 - \frac{B \cdot m_j}{K}.$$

We now show that the above expression is provably tighter than the expression obtained in (Chern et al., 2022) for $K' = 1$. Specifically, for $K' = 1$, note that:

$$\mathbb{E}\left[|S_j| - K'\right] = -1 \cdot \mathbb{P}\left[|S_j| = 0\right] + m_j \tag{1}$$

$$\Rightarrow m_j = \mathbb{E}\left[|S_j|\right] - 1 + \mathbb{P}\left[|S_j| = 0\right] \tag{2}$$

$$\Rightarrow m_j = \frac{K}{B} - 1 + \frac{\binom{N-K}{\frac{N}{B}}}{\binom{N}{\frac{N}{B}}} \tag{3}$$

$$\Rightarrow m_j \leq \frac{K}{B} - 1 + \left(1 - \frac{K}{N}\right)^{\frac{N}{B}} \tag{4}$$

$$\Rightarrow m_j \leq \frac{K}{B} - 1 + 1 - \frac{N}{B}\frac{K}{N} + \binom{\frac{N}{B}}{2}\left(\frac{K}{N}\right)^2 \tag{5}$$

$$\Rightarrow m_j \leq \frac{K^2}{2B}\left(\frac{1}{B} - \frac{1}{N}\right). \tag{6}$$

From the above, we see that the expected recall can be bounded as:

$$\mathbb{E}\left[\text{Recall}\right] \geq 1 - \frac{B}{K} \cdot \frac{K^2}{2B}\left(\frac{1}{B} - \frac{1}{N}\right) = 1 - \frac{K}{2}\left(\frac{1}{B} - \frac{1}{N}\right),$$

or equivalently, if we choose

$$B = \frac{K}{2\left(1 - r + \frac{K}{2N}\right)},$$

then, we will have $\mathbb{E}\left[\text{Recall}\right] \geq r$. On the other hand, (Chern et al., 2022) use $B = \frac{K}{1-r}$ to guarantee a recall of r, which is more than twice as large as required by our formula.

$$\underbrace{\frac{1}{2} \cdot \frac{K}{\left(1 - r + \frac{K}{2N}\right)}}_{\text{our formula}} < \frac{1}{2} \cdot \left(\frac{K}{1-r}\right) < \underbrace{\frac{K}{1-r}}_{\text{their formula}}$$

In Appendix Section A.4, we verify the tightness of our bound and show that expanding up to the quartic term in step 5 provides a near-perfect approximation of the exact expression that is practically indistinguishable. □

5.3 Implementation

In the first stage, we take an input array of shape `[batch_size, N]` and output two vectors: one for values and another for indices, both of which have the shape `[batch_size, B × K']`. Here, N is the total number of elements, B is the number of buckets, and K' is the number of top elements we select from each bucket.

We focus on identifying the top-K' elements of a single bucket, as supporting multiple buckets is a matter of running many *independent* instances of this subroutine. To create an effective fusible implementation, we track the top-K' elements in an online fashion as inputs continuously stream in from the matrix multiplication unit. We maintain two lists per bucket: one for the top-K' `values` and another for their corresponding `indices`. The `values` list is kept in descending order, and we ensure that each value's corresponding index is at the same position in the `indices` list. When a new element arrives, we update the lists in two steps:

1. If the new element is larger than the smallest element in the `values` list, replace the smallest element (and its index) with the new element (and its index).
2. Perform a single bubble sort pass over the lists to move the new element to its correct position.

Algorithm 1 contains the pseudocode for this subroutine. The first step requires one comparison and two selects for updating the value and index. The second step requires comparing adjacent elements (one comparison) and conditionally swapping elements (four selects) for each of the $(K' - 1)$ positions. In total, each input element requires $(5K' - 2)$ operations.

Algorithm 1 Bubble-sort based Top-K' Update Subroutine

```
1:  Input: input, index, values[K'], indices[K']
2:  if input ≥ values[-1] then            /* one compare */
3:      values[-1] = input                /* one select */
4:      indices[-1] = index               /* one select */
5:  end if
6:  for k = K' to 2 do
7:      if values[k] > values[k-1] then   /* one compare */
8:          swap(values[k], values[k-1])  /* two selects */
9:          swap(indices[k], indices[k-1]) /* two selects */
10:     end if
11: end for
```

Since the `values` list is stored in descending order, an input element larger than the k'th value in the list will also be larger than all subsequent values. This property allows the comparison in Line 7, Algorithm 1 to be done using the input element as the LHS, which eliminates a loop-carried dependency.

Once all inputs are processed, we obtain the final result by separately merging all the `values` lists to obtain the first stage values list and merging the corresponding `indices` lists to obtain the first stage indices list.

Since buckets group elements separated by a fixed stride, contiguous input elements map to different buckets. We can logically view the input array to have the shape `[batch_size, N / B, B]`. We store the top-K' lists with a physical layout of `[batch_size, K', B]` so that the minormost axis maps to the bucket axis, which aligns with the input's logical shape. The top-K' update subroutine (Algorithm 1) can be executed independently for each bucket and is trivially vectorizable along the bucket axis. To simplify the implementation, we restrict the number of buckets to a multiple of the vector width, denoted by L. We process contiguous L-sized chunks of the inputs and their corresponding L-sized top-K' lists in each iteration of a vectorized loop. The outline of the vectorized version is shown in Algorithm 2. Although this implementation appears to require $2K'$ loads and stores of the top-K' lists for each input that is read, we can schedule the iterations so that the input chunks corresponding to the same bucket are executed consecutively, allowing the top-K' lists to fully reside in the registers or the nearest cache depending on the choice of K' and the hardware.

Based on these insights, we implement our first stage kernel in Pallas, a JAX kernel language (Bradbury et al., 2018). We use `jax.lax.sort_key_val` and slice the top-K elements for the second stage. The Python code for our algorithm with detailed comments is provided in Listing A.6 for the unfused implementation and Listing A.7 for the matmul-fused implementation. To find the algorithm parameters

Algorithm 2 Outline of vectorization logic

```
num_chunks = input_size / #lanes
for in_chunk_idx = 0 to num_chunks - 1 do
    out_chunk_idx = in_chunk_idx % (#buckets / #lanes)
    Load inputs at in_chunk_idx
    Load top-K' lists at out_chunk_idx
    Do vectorized Top-K'-Update Subroutine
    Store updated top-K' lists at out_chunk_idx
end for
```

for a given input shape and recall target, we sweep through legal configurations and list those that meet the recall target. We then heuristically choose the configuration with the best performance. To calculate expected recall, we use Monte Carlo evaluations of the expectation expression derived in Section 5.2. The Python code to estimate expected recall and select algorithm parameters is shared in Listings A.8.

6 Results and Discussion

In Section 6.1, we also show that our algorithm can provide a substantially higher reduction in input size for the same expected recall compared to the improved baseline, which is the original algorithm by Chern et al. with our improved bound. Section 6.2 discusses the performance of our unfused Pallas implementation for TPUs. Finally, in section 6.3, we discuss the performance of our matmul-fused implementation for TPUs.

6.1 Expected recall of the first stage filtering

Table 2 shows the relationship between K', the number of buckets and the expected recall to select the top-1024 elements of an array of 262,144 elements. With a fixed number of output elements ($K' \times num\_buckets$), the expected recall increases significantly with K'. Keeping the expected recall fixed, even small values of K' (≤ 4) substantially reduce the number of output elements. For example, to achieve an expected recall of 95%, $K' = 1$ requires at least 16,384 output elements, while $K' = 4$ requires only 2,048 elements, an 8× reduction in the number of elements to process. Appendix Figure 10 plots expected recall versus the number of output elements for different values of K' to select the top-3,360 elements from an array of 430,080 elements. The expected recall improves rapidly with increasing K' as highlighted by the clear separation between the curves corresponding to our algorithm and the baseline.

Figure 4 shows the factor by which our variants of the algorithm, up to $K' = 4$, reduce the size of the inputs across a wide range of configurations ($K \in \{0.1\%, \ldots, 25\%\}$ and array sizes $\in [256, 4e9]$). We also account for the implementation constraints necessary for simplicity and performance, and therefore, the numbers indicate *real realizable reductions* using our implementation. The figure demonstrates that our algorithm drastically reduces the number of elements in virtually all configurations, with a median reduction of 7×. It only does worse for small values of K ($K \leq 10$) due to an artifact of our implementation that requires the number of buckets to be a multiple of 128. We conclude that our algorithm is broadly applicable and effectively reduces the number of output elements, even for small values of K'.

6.2 Improved latency of finding TopK elements

Table 2 presents the latency of the two stages of the unfused implementation of our algorithm to identify the top-1024 elements from an array of 262,144 elements. To achieve a recall target of 99%, the baseline requires around 305µs, while our algorithm with $K' = 4$ takes only 27µs, resulting in 11× reduction in latency.

The cost of the first stage remains nearly constant from $K' = 1$ to $K' = 6$, which we attribute to the memory-bound nature of the first stage. According to the performance model in Section 2.3 and the numbers in Table 1, the first stage must be memory bound until we exceed 30 VPU operations per 4-byte element. This transition occurs around $K' = 6$ for our algorithm, according to the operational intensity formula in Section 5.3. Therefore, we expect the latency of the first stage to be independent of K' until we reach $K' = 6$.

6.3 Fusing Top-K with matrix multiplication

Many real-world applications require identifying the Top-K results from the outputs of matrix multiplications. One prominent example is maximum inner-product search (MIPS), where for a given query vector, the task is to retrieve the top-K vectors from a large database that have the highest inner products.

In this section, we evaluate our algorithm on a MIPS workload consisting of a database of one million 128-dimensional vectors and 1024 queries. Table 3 reports the runtime of our fused algorithm on TPUv5e for this task. When using `jax.lax.top_k`, an exact Top-K algorithm, the second stage takes nearly 80× (587ms) more time than the matmul (7.32ms). `jax.lax.approx_max_k` (Chern et al., 2022) reduces the cost but the second stage still takes 13× (118ms) more time than the matmul.

In our unfused implementation for $K' = 1$, which uses the improved bounds of the expected recall to choose the number of buckets, the second stage (50ms) reduces to roughly 6× the cost of the matmul.

Moving to $K' = 4$, we reduce the cost of the second stage (3.51ms) to slightly less than half of the cost of the matmul. At this point, the matrix multiplication (7.31ms) and the

Table 2. **(Left) Expected recall versus K' for selecting top-1024 elements from an array of 262,144 elements.** `#num_elements` refers to the number of output elements from the first stage, which is $B \times K'$. Smaller `#num_elements` will lead to better performance in the second stage. **(Right) The runtime of our algorithm on TPUv5e for a batch size of 8.** The `jax.lax.approx_max_k` rows present the performance of the official JAX implementation (which only supports $K' = 1$), while the $K' = 1$ rows present the performance of our implementation.

Algorithm Parameters		Algorithmic Performance		Runtime Performance		
K'	buckets	num_elements	$\mathbb{E}$[recall]	Stage 1	Stage 2	Total
jax.lax.approx_max_k	131,072	131,072	0.998 ± 0.000	12us	649us	661us
jax.lax.approx_max_k	65,536	65,536	0.992 ± 0.001	13us	292us	305us
jax.lax.approx_max_k	32,768	32,768	0.987 ± 0.004	13us	131us	144us
1	65,536	65,536	0.992 ± 0.001	13us	313us	326us
1	32,768	32,768	0.987 ± 0.004	14us	141us	155us
1	16,384	16,384	0.972 ± 0.005	13us	64us	77us
1	8,192	8,192	0.942 ± 0.007	13us	30us	42us
2	4,096	8,192	0.991 ± 0.003	15us	30us	45us
2	2,048	4,096	0.968 ± 0.006	13us	14us	27us
3	2,048	6,144	0.996 ± 0.002	16us	32us	48us
3	1,024	3,072	0.977 ± 0.005	12us	11us	23us
4	1,024	4,096	0.996 ± 0.002	13us	14us	27us
4	512	2,048	0.963 ± 0.007	12us	8us	20us
5	512	2,560	0.989 ± 0.004	13us	9us	22us
6	512	3,072	0.997 ± 0.002	14us	11us	25us
6	256	1,536	0.951 ± 0.008	14us	8us	22us
8	512	4,096	0.992 ± 0.004	16us	14us	30us
10	256	2,560	0.999 ± 0.000	19us	9us	28us
12	128	1,536	0.984 ± 0.006	23us	8us	31us
16	128	2,048	0.999 ± 0.001	29us	8us	37us

Table 3. **The runtime of our algorithm on TPUv5e to identify top-1024 elements from a database of 1M 128-dimensional vectors with 99% recall for 1024 query vectors.** The `jax.lax.top_k` row represents the performance of exact top-K. `jax.lax.approx_max_k` row presents the performance of the official JAX implementation for the $K' = 1$ setting. The remaining rows represent the performance of our implementation.

Algorithm	Matmul	Stage 1	Stage 2
top_k	7.32ms	-	587ms
approx_max_k	9.06ms	fused	118ms
$K' = 1$	7.32ms	6.58ms	50.0ms
$K' = 1$	9.03ms	fused	50.0ms
$K' = 4$	7.31ms	10.80ms	3.51ms
$K' = 4$	6.55ms	fused	3.51ms

first stage (10.80ms) dominate the runtime. By fusing the first stage with the matmul, we eliminate the cost of the first stage and also improve the matmul's performance (6.55ms). While we see modest improvements to matmul performance in this setting, the gains can be significant in many practical MIPS applications. To understand the source of these gains in the matmul step, we study the performance characteristics of the matmul operation.

The MIPS task involves multiplying two matrices of the shape `[B, D]` with `[D, N]`, where B is the number of queries, D is the vector size, and N is the number of database entries. In practice, B and D are often much smaller than N. Let E denote the number of bytes per element. The arithmetic intensity of the matrix multiplication operation – defined as the ratio of number of FLOPS to number of bytes transferred from memory – is given by:

$$\begin{aligned}\text{arithmetic intensity} &= \frac{2BDN}{E\left(BD + DN + BN\right)} \\ &\approx \frac{2BDN}{E\left(DN + BN\right)} \\ &= \frac{2BD}{E\left(B + D\right)} \\ &\leq \frac{2}{E}\min(B, D)\end{aligned}$$

In regimes where $B \gg D$, the matmul may be memory-bound if D is not large enough. This is common in large-scale deployments that serve millions of queries per second (QPS) with D in the low hundreds. In such cases, the output tensor is the largest and would dominate the memory traffic. By fusing the first stage with the matmul, we avoid having to write it to memory and increase the arithmetic intensity. This shifts the matmul closer to being compute-bound if not compute-bound outright. As a result, fusion not only

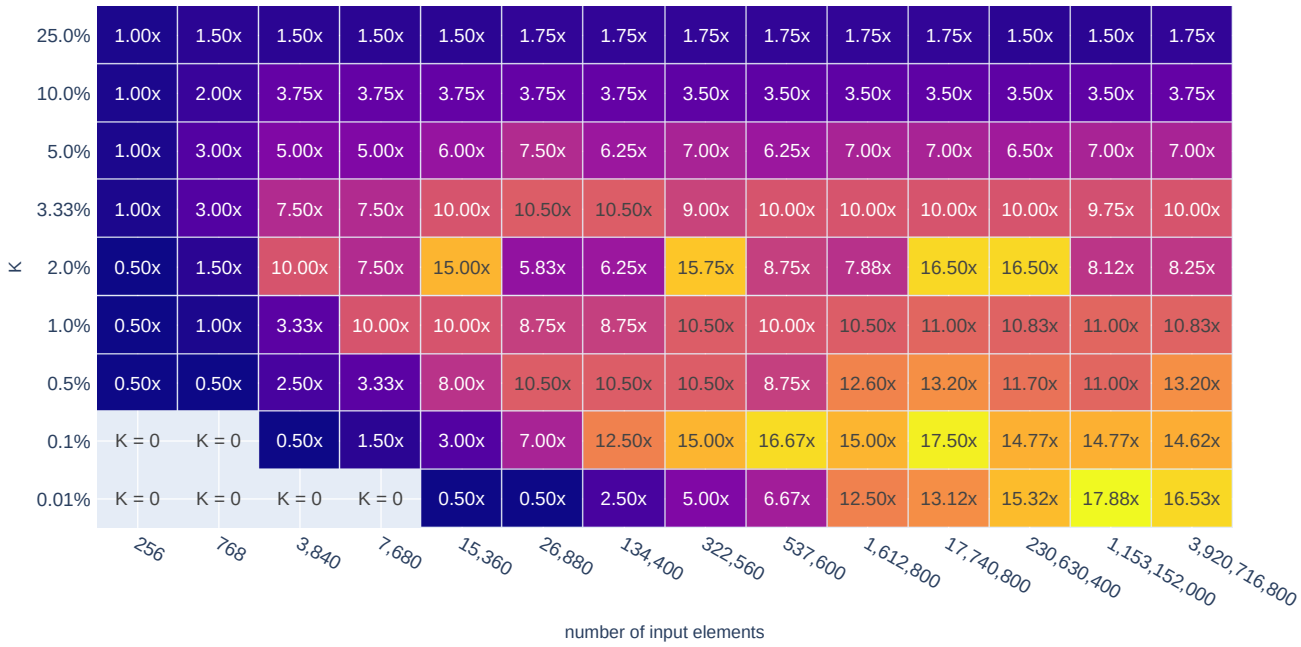

Figure 4. **Factor of additional reduction in output elements over the baseline ($K' = 1$) for 99% expected recall target.** The heatmap shows the factor by which our algorithm with $2 \leq K' \leq 4$ reduces the number of elements in the first stage over the reductions provided by the baseline, i.e., a value of $2\times$ indicates that our algorithm outputs two times fewer elements compared to the $K' = 1$ baseline. Our implementation constrains the number of buckets to be a multiple of 128 for simplicity and performance, which is accounted for in this figure. Even though $K' > 1$ would require fewer buckets compared to $K' = 1$, rounding the number of buckets to a multiple of 128 may sometimes lead to more output elements than required by $K' = 1$, as visible in the bottom left corner of the figure.

eliminates the cost of the first stage but can also improve the performance of the matmul in many cases.

Acknowledgments

The authors thank Lubo Litchev for helpful discussions and Ethan Shen for their feedback on the paper. We also express our gratitude to the reviewers of our submission at MLSys 2025 for their valuable comments, which helped improve the quality of our work.

References

Alizadeh, K., Mirzadeh, I., Belenko, D., Khatamifard, K., Cho, M., Mundo, C. C. D., Rastegari, M., and Farajtabar, M. Llm in a flash: Efficient large language model inference with limited memory, 2024. URL https://arxiv.org/abs/2312.11514.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J.-B., Damoc, B., Clark, A., de Las Casas, D., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. Improving language models by retrieving from trillions of tokens, 2022. URL https://arxiv.org/abs/2112.04426.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chern, F., Hechtman, B., Davis, A., Guo, R., Majnemer, D., and Kumar, S. Tpu-knn: K nearest neighbor search at peak flop/s, 2022. URL https://arxiv.org/abs/2206.14286.

Google Cloud. Cloud TPU Performance Guide, 2024a. URL https://cloud.google.com/tpu/docs/performance-guide.

Google Cloud. TPU System Architecture, 2024b. URL https://cloud.google.com/tpu/docs/system-architecture-tpu-vm.

He, X. O. Mixture of a million experts, 2024. URL https://arxiv.org/abs/2407.04153.

Key, O., Ribar, L., Cattaneo, A., Hudlass-Galley, L., and Orr, D. Approximate top-k for increased parallelism.

In *Adaptive Foundation Models: Evolving AI for Personalized and Efficient Learning*, 2024. URL https://openreview.net/forum?id=UonuElM9kV.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

Li, Y., Liu, K., Satapathy, R., Wang, S., and Cambria, E. Recent developments in recommender systems: A survey, 2023. URL https://arxiv.org/abs/2306.12680.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., and Chen, B. Deja vu: Contextual sparsity for efficient llms at inference time, 2023. URL https://arxiv.org/abs/2310.17157.

Madaan, L., Bhojanapalli, S., Jain, H., and Jain, P. Treeformer: Dense gradient trees for efficient attention computation, 2023. URL https://arxiv.org/abs/2208.09015.

Norrie, T., Patil, N., Yoon, D. H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N. P., and Patterson, D. Google's training chips revealed: Tpuv2 and tpuv3. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 1–70, 2020. doi: 10.1109/HCS49909.2020.9220735.

NVIDIA. NVIDIA H100 Tensor Core GPU, 2024. URL https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

Roy, A., Saffar, M., Vaswani, A., and Grangier, D. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Ruan, M., Yan, G., Xiao, Y., Song, L., and Xu, W. Adaptive top-k in sgd for communication-efficient distributed learning, 2023. URL https://arxiv.org/abs/2210.13532.

Samaga B L, Y., Yerram, V., You, C., Bhojanapalli, S., Kumar, S., Jain, P., and Netrapalli, P. Hire: High recall approximate top-k estimation for efficient llm inference, 2024. URL https://arxiv.org/abs/2402.09360.

Shen, E., Fan, A., Pratt, S. M., Park, J. S., Wallingford, M., Kakade, S. M., Holtzman, A., Krishna, R., Farhadi, A., and Kusupati, A. Superposed decoding: Multiple generations from a single autoregressive inference pass, 2024. URL https://arxiv.org/abs/2405.18400.

Shi, S., Chu, X., Cheung, K. C., and See, S. Understanding top-k sparsification in distributed deep learning, 2019. URL https://arxiv.org/abs/1911.08772.

Snider, D. and Liang, R. Operator fusion in xla: Analysis and evaluation, 2023. URL https://arxiv.org/abs/2301.13062.

Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., Tafti, P., Hussenot, L., Sessa, P. G., Chowdhery, A., Roberts, A., Barua, A., Botev, A., Castro-Ros, A., Slone, A., Héliou, A., Tacchetti, A., Bulanova, A., Paterson, A., Tsai, B., Shahriari, B., Lan, C. L., Choquette-Choo, C. A., Crepy, C., Cer, D., Ippolito, D., Reid, D., Buchatskaya, E., Ni, E., Noland, E., Yan, G., Tucker, G., Muraru, G.-C., Rozhdestvenskiy, G., Michalewski, H., Tenney, I., Grishchenko, I., Austin, J., Keeling, J., Labanowski, J., Lespiau, J.-B., Stanway, J., Brennan, J., Chen, J., Ferret, J., Chiu, J., Mao-Jones, J., Lee, K., Yu, K., Millican, K., Sjoesund, L. L., Lee, L., Dixon, L., Reid, M., Mikuła, M., Wirth, M., Sharman, M., Chinaev, N., Thain, N., Bachem, O., Chang, O., Wahltinez, O., Bailey, P., Michel, P., Yotov, P., Chaabouni, R., Comanescu, R., Jana, R., Anil, R., McIlroy, R., Liu, R., Mullins, R., Smith, S. L., Borgeaud, S., Girgin, S., Douglas, S., Pandya, S., Shakeri, S., De, S., Klimenko, T., Hennigan, T., Feinberg, V., Stokowiec, W., hui Chen, Y., Ahmed, Z., Gong, Z., Warkentin, T., Peran, L., Giang, M., Farabet, C., Vinyals, O., Dean, J., Kavukcuoglu, K., Hassabis, D., Ghahramani, Z., Eck, D., Barral, J., Pereira, F., Collins, E., Joulin, A., Fiedel, N., Senter, E., Andreev, A., and Kenealy, K. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

A APPENDIX

A.1 Estimating peak VPU throughput of TPUv5e

We used two test programs with a controllable parameter that allows us to vary the number of vector operations per element. We run the programs on a large `s32[4096, 4096]`-shaped array with different parameters and time the kernels. We verify that the compiler fuses the operations into a single kernel. We assume that addition and multiplication are instructions in the TPU's ISA. Given the large size of the inputs, we assume that these programs saturate the vector processing unit.

```
@partial(jax.jit, static_argnames='n')
def fibonacci(x, y, n):
  for i in range(n):
    c = x + y
    x = y
    y = c
  # We expect the compiler to optimize the snippet to the following series of
  # operations:
  # r1 = x
  # r2 = y
  # r3 = r1 + r2
  # r1 = r2 + r3
  # r2 = r3 + r1
  # r3 = r1 + r2
  # ...
  #
  # For every two elements read from memory, we perform 'n' additions.
  return y

@functools.partial(jax.jit, static_argnames='steps')
def fast_exponentiation(x: jax.Array, steps: int):
  z = x
  for _ in range(1, steps):
    z = z * z
  return z
```

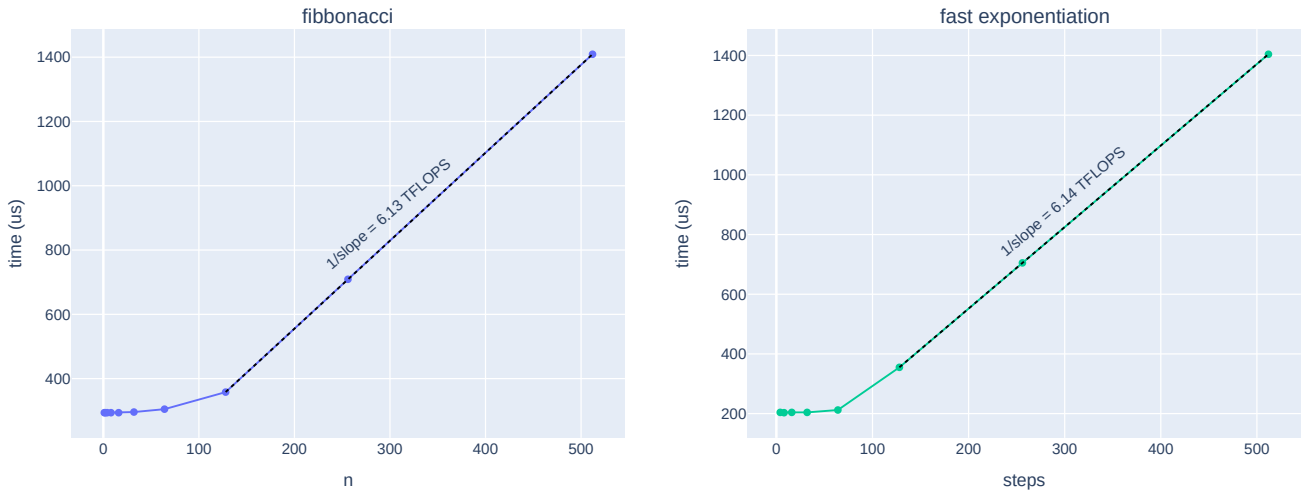

Figure 5. **Estimating the throughput of the VPU on TPUv5e.** We expect the kernels to be memory-bound (constant line) initially and then be vector compute bound (linear scaling). We fit a line to the points in the linear region with the following model: $time = num\_ops \times \frac{1}{throughput} + overhead$. The inverse of the slope is an estimate of the peak throughput of the VPU.

A.2 Comparison with concurrent work on GPU implementation of the algorithm

A concurrent work by Key et al. (2024) presents the same algorithm as ours but targets GPUs instead of TPUs. Unlike our implementation, which is explicitly designed with fusibility in mind, their approach consists of two separate, unfused stages for the algorithm. We place a strong emphasis on the ability to fuse parts of the algorithm with preceding operations, such as matrix multiplication, which is not addressed in their work. We use a theoretical performance model to guide the design and analyze the algorithm's performance in both fused and unfused settings. Using this model, we are able to accurately reason about the runtimes for various choices of algorithm parameters. For example, we account for the free vector compute available in memory-bound or matmul-bound computations that we can exploit. In contrast, their model only models the compute requirements by counting operations and do not account for the interaction with other subsystems. On the theoretical side, we analyze the algorithm for our extension to the $K' > 1$ case, and additionally provide an improved analysis of the baseline $K' = 1$ setting from Chern et al. (2022), leading to substantial performance improvements.

While our paper focuses on the theoretical analysis and the systems-level aspects of the algorithm, their work provides a broader contextualization of the work. They offer an in-depth discussion of related Top-K algorithms, both exact and approximate, and explore a wide array of practical applications such as sparse attention in transformers and knowledge graph completion. These evaluations demonstrate the end-to-end impact of the algorithm in real-world scenarios. Our paper does not cover these aspects in detail, and we cite theirs for such discussions. Even though the algorithm is shared, the two papers are complementary in focus and scope.

A.3 Empirical verification of the quality of Monte Carlo estimates for expected recall

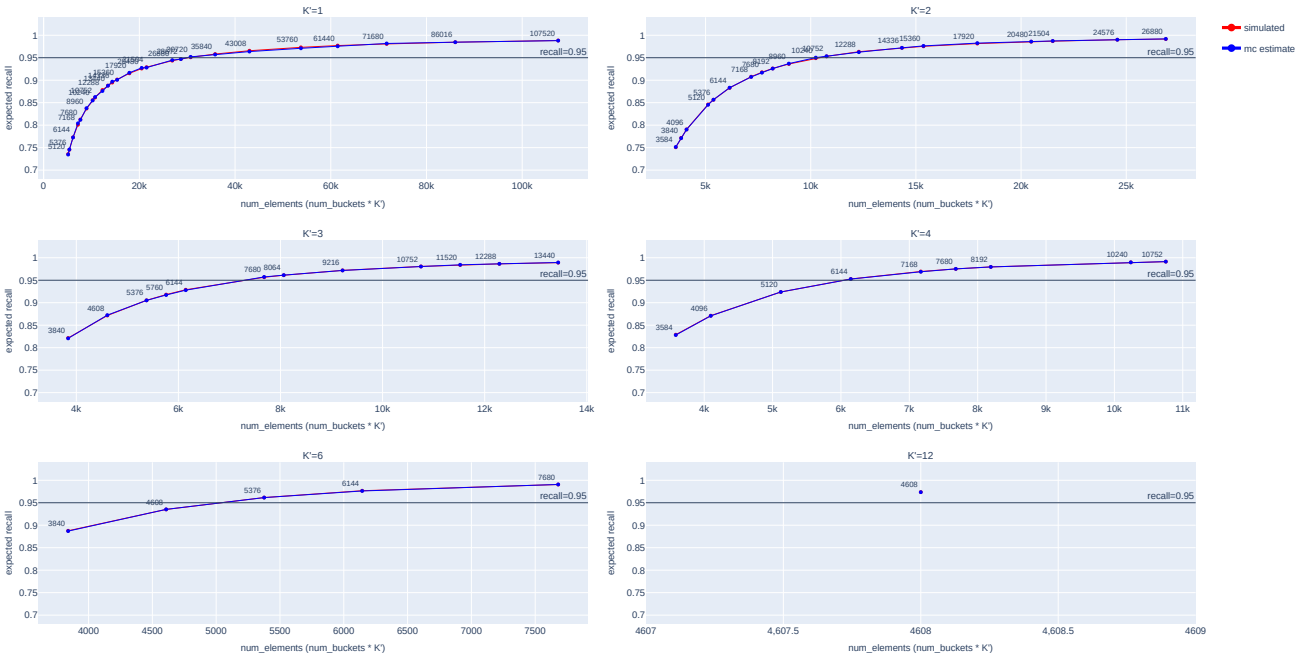

Figure 6. **Verification of Monte Carlo estimates of expected recall against simulated runs of the algorithm for finding top-3360 ($\approx$ 0.8%) elements from an array of size 430,080.** The simulated estimates were obtained from 1024 runs of the algorithm on randomly generated integers and the Monte Carlo estimates were obtained from 262144 samples of the expectation expression derived in Section 5.2.

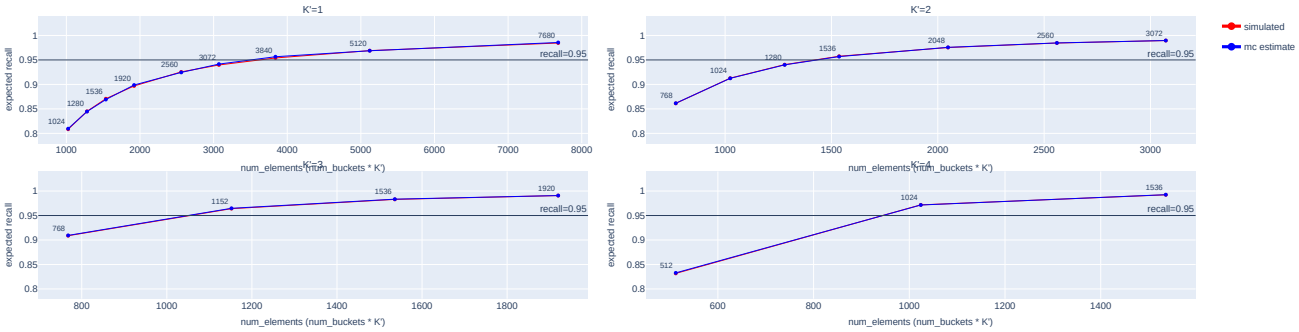

Figure 7. **Verification of Monte Carlo estimates of expected recall against simulated runs of the algorithm for finding top-480 ($\approx$ 3%) elements from an array of size 15,360.** The simulated estimates were obtained from 1024 runs of the algorithm on randomly generated integers and the Monte Carlo estimates were obtained from 262,144 samples of the expectation expression derived in Section 5.2.

A.4 Quality of the theoretical bounds on expected recall for $K' = 1$ setting

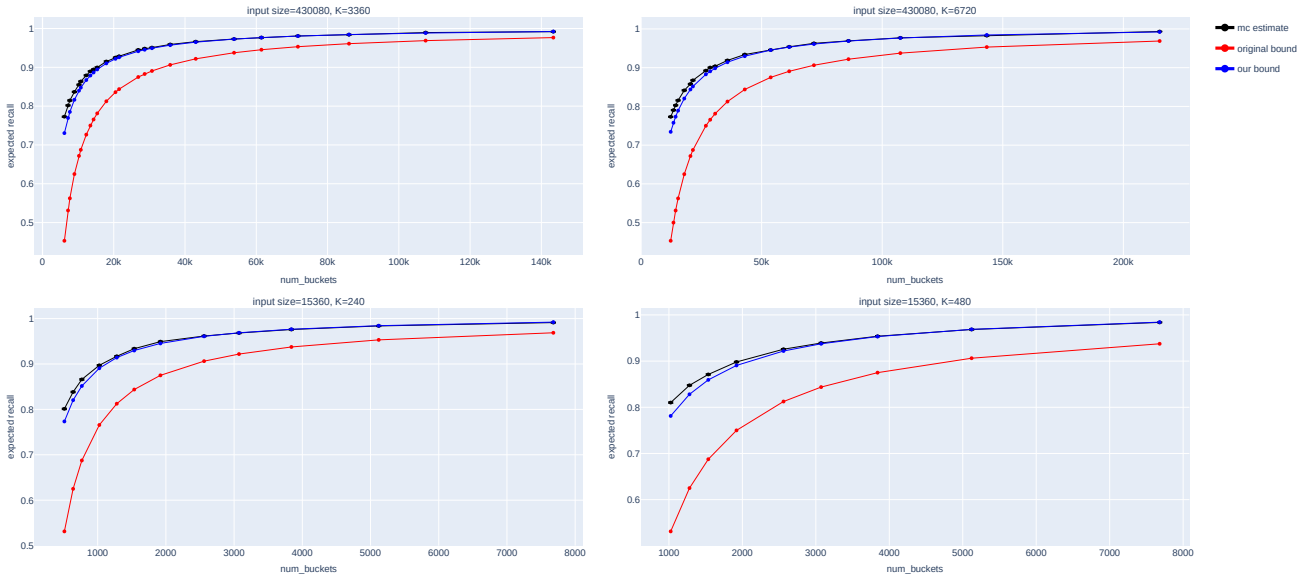

Figure 8. **Tightness of our theoretical bound on expected recall for $K' = 1$ setting compared to the original bound derived in Chern et al.** See Section 5.2 for the derivation of our bound ($\mathbb{E}\left[\text{Recall}\right] \geq 1 - \frac{K}{2B}$) which is tighter than the original bound ($\mathbb{E}\left[\text{Recall}\right] \geq 1 - \frac{K}{B}$).

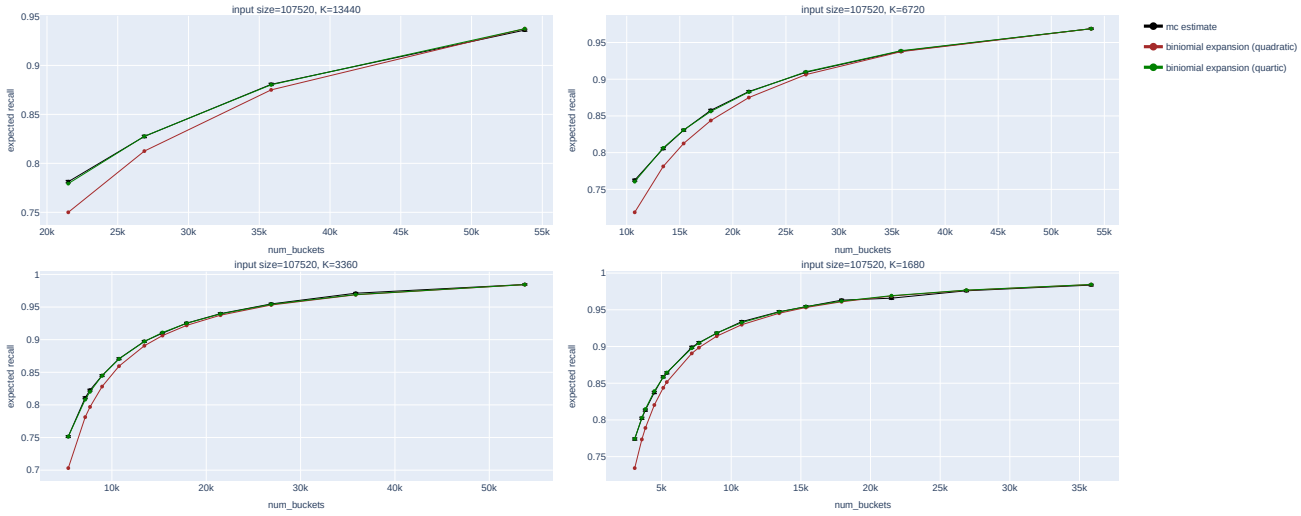

Figure 9. **Expanding the binomial expression in Step 5 to quartic terms in Theorem 1 accurately approximates the expected recall.**

A.5 Packages and utility functions for the code listings

```
import jax
import jax.numpy as jnp
from jax.experimental import pallas as pl

import numpy as np

def get_all_factors(n):
  small_factors = [i for i in range(1, int(np.ceil(np.sqrt(n)))) if n % i == 0]
  pair_factors = [n // factor for factor in small_factors]
  return set(small_factors + pair_factors)
```

A.6 Unfused implementation of our algorithm

```
def generalized_partial_reduce(inputs, local_K, num_buckets, tunable_params={}, **kwargs):
  """ApproxTopK with generalized partial reduce for minor-axis reductions.

  Note: Input elements separated by a fixed stride form a bucket.

  Args:
    inputs: jax.ShapeDtypeStruct-compatible object of the input array of the
      shape [batch_size, reduction_dims].
    local_K: number of top elements to keep track of per bucket
    num_buckets: number of buckets
    tunable_params: These are hardware-specific (auto)tunable parameters. See
      the uses in the function body for more information.
    *kwargs: forwarded as is to `pallas_call`.

  Note: The default tunable params selection does not check if the choices are
    meaningful (e.g., VMEM OOMs). Please tune the choices if they don't work.

  Note: The procedure does not allow arbitrary input shapes for performance
    reasons and triggers assertions in case of incompatible shapes. Please pad
    the input shapes accordingly.

  Returns:
    A unary function implementing the algorithm that takes the input array and
    returns a tuple of TopK values and indices with the shape ([batch_size,
    num_elements], [batch_size, num_elements]) where num_elements is `local_K *
    num_buckets`.
  """

  # Pallas imposes constraints on block specifications. Refer to the docs.
  PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE = 8
  PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE = 128

  input_shape = inputs.shape
  batch_size, reduction_dims = input_shape
  num_elements = num_buckets * local_K
  output_shape = (batch_size, num_elements)

  batch_tile_size = tunable_params.get('batch_tile_size', None)
  if batch_tile_size is None:
    factors = get_all_factors(batch_size)
```

```
  legal_factors = {
      f for f in factors
        if f % PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE == 0
        or f == batch_size
  }
  # Higher tile size provide more opportunities for instruction-level
  # parallelism.
  batch_tile_size = max({ f for f in legal_factors if f <= 2048 })
assert(batch_size % batch_tile_size == 0)

reduction_tile_size = tunable_params.get('reduction_tile_size', None)
if reduction_tile_size is None:
  # In each subprogram, we would want to process sufficient number of inputs
  # to cover all the buckets. We would also want to have several passes over
  # the buckets in each subprogram so that the compiler can schedule the loop
  # iterations in a way that state loads/stores to the same buckets run
  # consecutively, and the state information is cached in registers or in the
  # nearest cache.
  factors = set(get_all_factors(reduction_dims))
  legal_factors = {
      f for f in factors
        if f % num_buckets == 0 and
           f % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0
  }
  # We want to pick sufficiently large blocks so that the overheads are
  # amortized. However, we don't want the blocks to be too large to a point
  # that we have too few pipelined iterations and the head and tail latencies
  # make up a substantial portion of the runtime. These numbers vary from chip
  # to chip and need to be tuned.
  reduction_tile_size = max({
      d for d in legal_factors if d <= max(32_768, num_buckets)
  })
assert(reduction_dims % reduction_tile_size == 0)

# For simplicity, we restrict the tile sizes to be such that all the buckets
# are processed equal number of times in each subprogram.
assert(reduction_tile_size % num_buckets == 0)

input_transform_indices_fn = lambda i, j: (i, j)
input_tile_shape = (batch_tile_size, reduction_tile_size)
iteration_bounds = [
    asz // tsz for asz, tsz in zip(input_shape, input_tile_shape)
]
assert(len(input_tile_shape) == len(input_shape))

# Pallas currently does not allow non-consecutive grid points to write to the
# same slices of output. We therefore do not block the output along the
# reduction axis.
output_transform_indices_fn = lambda i, j: (i, 0)
output_tile_shape = (batch_tile_size, num_elements)
assert(len(output_tile_shape) == len(output_shape))

# At the time of writing, the Mosaic compiler does not have a rule for
# lowering comparisions between types that are not 32-bits wide. Hence, we
```

```
# explicitly promote inputs to their wider 32-bit type.
assert(inputs.dtype.itemsize <= 4)
if jnp.issubdtype(inputs.dtype, jnp.floating):
  compute_type = jnp.float32
elif jnp.issubtype(inputs.dtype, jnp.signedinteger):
  compute_type = jnp.int32
elif jnp.issubtype(inputs.dtype, jnp.unsignedinteger):
  compute_type = jnp.uint32
else:
  assert "Unknown data type"

def _kernel(inputs_ref, values_ref, indices_ref):
  assert(values_ref.shape == indices_ref.shape)

  # b = batch axis, r = reduction axis
  tile_b, tile_r = pl.program_id(0), pl.program_id(1)

  # On TPUs, we are guaranteed a sequential grid execution and we use the
  # first run for each batch to initialize the outputs.
  @pl.when(tile_r == 0)
  def initialize_outputs():
    # We don't have to initialize the indices as non-strict comparators for
    # selection guarantee that the indices will be updated.
    values_ref[...] = jnp.full_like(values_ref, -jnp.inf)

  # The loop count may be large but we explicitly want to unroll to eliminate
  # a lot of state loads/stores. Rewriting the loop as two nested loops where
  # the unrolled inner loop explicitly reuses the state load/stores and the
  # non-unrolled outer loop runs over different sets of buckets may lead to
  # faster compilation.
  num_iterations_over_outputs = reduction_tile_size // num_buckets
  for iter_idx in range(num_iterations_over_outputs):
    # Note that inputs are already tiled by pallas and we use local offsets.
    inputs = inputs_ref[
        :, pl.ds(start=iter_idx * num_buckets, size=num_buckets)
    ]
    inputs = inputs.astype(compute_type)

    iota = jax.lax.broadcasted_iota(indices_ref.dtype, inputs.shape, 1)
    iota += tile_r * reduction_tile_size + iter_idx * num_buckets
    assert(inputs.shape == iota.shape)

    # Load state information for the current chunk.
    values_by_k, indices_by_k = [], []
    for k in range(local_K):
      values = values_ref[
          :,
          pl.ds(
              start=k * num_buckets,
              size=num_buckets
          )
      ].astype(compute_type)
      indices = indices_ref[
          :,
```

```
            pl.ds(
                start=k * num_buckets,
                size=num_buckets
            )
        ]
        assert(values.shape == indices.shape)
        values_by_k.append(values)
        indices_by_k.append(indices)

      # Compute the new state information for the current chunk.
      pred = inputs >= values_by_k[-1]
      values_by_k[-1] = jax.lax.select(pred, inputs, values_by_k[-1])
      indices_by_k[-1] = jax.lax.select(pred, iota, indices_by_k[-1])
      for k in reversed(range(1, local_K)):
        # Note that the commented line and uncommented line are algorithmically
        # equivalent, but the uncommented version has one less loop-carried
        # dependency.
        # pred = values_by_k[k] > values_by_k[k - 1]
        pred = inputs > values_by_k[k - 1]

        values_to_shift = values_by_k[k]
        values_by_k[k] = \
          jax.lax.select(pred, values_by_k[k-1], values_to_shift)
        values_by_k[k-1] = \
          jax.lax.select(pred, values_to_shift, values_by_k[k-1])

        indices_to_shift = indices_by_k[k]
        indices_by_k[k] = \
          jax.lax.select(pred, indices_by_k[k-1], indices_to_shift)
        indices_by_k[k-1] = \
          jax.lax.select(pred, indices_to_shift, indices_by_k[k-1])

      # Write the new state information for the current chunk.
      for k in range(local_K):
        values_ref[
            :,
            pl.ds(
                start=k * num_buckets,
                size=num_buckets
            )
        ] = values_by_k[k].astype(values_ref.dtype)
        indices_ref[
            :,
            pl.ds(
                start=k * num_buckets,
                size=num_buckets
            )
        ] = indices_by_k[k]

  def wrapper(inputs):
    pr_values, pr_indices = pl.pallas_call(
        _kernel,
        in_specs=[
            pl.BlockSpec(input_tile_shape, input_transform_indices_fn),
```

```
        ],
        out_shape=[
            jax.ShapeDtypeStruct(output_shape, inputs.dtype),
            jax.ShapeDtypeStruct(output_shape, jnp.int32),
        ],
        out_specs=[
            pl.BlockSpec(output_tile_shape, output_transform_indices_fn),
            pl.BlockSpec(output_tile_shape, output_transform_indices_fn)
        ],
        grid=iteration_bounds,
        compiler_params=pltpu.TPUCompilerParams(
          dimension_semantics=("parallel", "arbitrary")
        ),
        **kwargs
    )(inputs)
    return pr_values, pr_indices
  return wrapper

def make_generalized_approx_topk(operand, num_buckets, local_K, global_K, **kwargs):
    partial_reduce_fn = \
      generalized_partial_reduce(operand, local_K, num_buckets, **kwargs)

    def wrapper(operand):
      bucket_values, bucket_indices = partial_reduce_fn(operand)
      values, indices = \
        jax.lax.sort_key_val(bucket_values, bucket_indices, is_stable=False)
      values = jnp.flip(values[..., -global_K:], axis=-1)
      indices = jnp.flip(indices[..., -global_K:], axis=-1)
      return values, indices
    return wrapper
```

A.7 Matmul-fused implementation of our algorithm

```
def matmul_fused_generalized_partial_reduce(
    lhs, rhs,
    local_K, num_buckets,
    tunable_params={}, *, **kwargs
):
  """Fused ApproxTopK with generalized partial reduce for minor-axis reductions.

  Note: Input elements separated by a fixed stride form a bucket.

  Args:
    lhs: jax.ShapeDtypeStruct-compatible object of LHS array with shape
      [batch_size, contracting_dims].
    rhs: jax.ShapeDtypeStruct-compatible object of RHS array with shape
      [contracting_dims, reduction_dims].
    local_K: number of top elements to keep track of per bucket.
    num_buckets: number of buckets.
    tunable_params: These are hardware-specific (auto)tunable parameters. See
      the uses in the function body for more information.
    **kwargs: forwarded as is to `pallas_call`.

  Note: The default tunable params selection does not check if the choices are
```

```
    meaningful (e.g., VMEM OOMs). Please tune the choices if they don't work.

  Note: The procedure does not allow arbitrary input shapes for performance
    reasons and triggers assertions in case of incompatible shapes. Please pad
    the input shapes accordingly.

  Returns:
    A binary function implementing the algorithm that takes the arguments to
    matmul and returns a tuple of TopK values and indices of lhs @ rhs with
    shapes of ([batch_size, num_elements], [batch_size, num_elements]).
    The `num_elements` is calculated as `num_buckets * local_K`.
  """

  # Pallas imposes constraints on block specifications. Refer to the docs.
  PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE = 8
  PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE = 128

  # This implementation maps buckets to the minormost axis.
  assert(num_buckets % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0)

  batch_size, contracting_dims = lhs.shape
  contracting_dims_rhs, reduction_dims = rhs.shape
  assert(contracting_dims == contracting_dims_rhs)
  assert(reduction_dims % num_buckets == 0)
  assert(num_buckets < reduction_dims)
  assert(lhs.dtype == rhs.dtype)

  num_elements = num_buckets * local_K
  output_shape = (batch_size, num_elements)

  # We will block the matrices for software pipelining as follows:
  # lhs: [batch_tile_size, contracting_tile_size]
  # rhs: [contracting_tile_size, reduction_tile_size]
  # result-scratch: [batch_tile_size, reduction_tile_size]
  #
  # Note that partial reduce computation can start only after the loop over the
  # contracting axis ends, as it requires fully accumulated sums to begin.
  # The VPU may idle waiting for the result tile in all but the last iteration
  # of the loop over the contracting axis.
  #
  # We can alleviate the problem by pipelining the computation into matmul and
  # TopK stages. We let the VPU processes the previous result tile while the new
  # result tile is being accumulated. We do not implement the idea here and
  # choose to use large tiling along contracting axis to minimize wasted cycles.

  # The tiles will be further subtiled automatically by the compiler to meet the
  # shape of the hardware matmul units. Since the subtiling loops will be fully
  # unrolled, the compiler would ideally generate code to run TopK on the
  # previous subtile while a new subtile is being accumulated.

  batch_tile_size = tunable_params.get('batch_tile_size', None)
  if batch_tile_size is None:
    # We want this to be as large as possible. This parameter controls the
    # arithmetic intensity of the blocked matmul operation. Therefore, we must
```

```
  # have a batch tile size that is high enough to ensure that each blocked
  # matmul operation is MXU-bound.
  factors = get_all_factors(batch_size)
  legal_factors = {
      f for f in factors
        if f % PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE == 0
        or f == batch_size
  }
  # We heuristically pick the largest legal tile size up to 2048. A larger
  # tile size may be more performant but carries the risk of exhausting VMEM.
  batch_tile_size = max({ f for f in legal_factors if f <= 2048 })
assert(batch_size % batch_tile_size == 0)

contracting_tile_size = tunable_params.get('contracting_tile_size', None)
if contracting_tile_size is None:
  # This tile size does not affect the arithmetic intensity of the matrix
  # multiplication. However, as mentioned earlier, we cannot start the TopK
  # computation without the fully accumulated result tile. To minimize VPU
  # idle time, we would like to have as large a tile size as possible for the
  # contracting axis so that there are as few iterations as possible where the
  # final result tile is only partially accumulated.
  factors = get_all_factors(contracting_dims)

  # This axis would be the minor axis for LHS and the major axis for RHS. It
  # must meet the multiple requirements for the LHS and RHS respectively or
  # must be equal to the axis size.
  legal_factors = {
      f for f in factors
        if (f % PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE == 0 and
            f % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0)
        or f == contracting_dims
  }

  # We heuristically pick the largest size up to 2048. A larger tile size
  # would minimize VPU idling for this implementation but increases the risk
  # of VMEM OOMs.
  contracting_tile_size = max({ f for f in legal_factors if f <= 2048 })
assert(contracting_dims % contracting_tile_size == 0)

reduction_tile_size = tunable_params.get('reduction_tile_size', None)
if reduction_tile_size is None:
  # There are two possiblities for picking reduction tile size:
  # 1. tile size > number of buckets
  # 2. tile size <= number of buckets
  #
  # Our implementation handles both cases. However, the first possibility is
  # preferred for performance reasons.

  # We need to ensure that we have sufficient VMEM to accomodate the large
  # tiles for matrix multiplication so that it remains MXU-bound. Let's cap
  # the tile size to 4096 to reduce the risk of exhausting VMEM.
  if num_buckets > 4096:
    assert(reduction_dims % num_buckets == 0)
    factors = set(get_all_factors(num_buckets))
```

```
    legal_factors = {
        f for f in factors
          if f % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0
    }
    reduction_tile_size = max({ d for d in legal_factors if d <= 4096 })
  else:
    # We want to pick sufficiently large tiles so that the load/store overhead
    # of the TopK lists is amortized. However, we don't want the blocks to be
    # too large to a point that we have too few pipelining iterations and the
    # head and tail latencies make up a substantial portion of the runtime.
    assert(reduction_dims % num_buckets == 0)
    factors = set(get_all_factors(reduction_dims))
    legal_factors = {
        f for f in factors
        if f % num_buckets == 0
        and f % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0
    }
    reduction_tile_size = max({ f for f in legal_factors if f <= 4096 })

assert(reduction_dims % reduction_tile_size == 0)
if reduction_tile_size > num_buckets:
  # For simplifying the implementation, we restrict the tile size to be
  # multiples of number of buckets.
  assert(reduction_tile_size % num_buckets == 0)
else:
  # For simplifying the implementation, we restrict the tile size to be
  # factors of number of buckets.
  assert(num_buckets % reduction_tile_size == 0)

lhs_transform_indices_fn = lambda i, j, k: (i, k)
lhs_tile_shape = (batch_tile_size, contracting_tile_size)
assert(len(lhs_tile_shape) == len(lhs.shape))

rhs_transform_indices_fn = lambda i, j, k: (k, j)
rhs_tile_shape = (contracting_tile_size, reduction_tile_size)
assert(len(rhs_tile_shape) == len(rhs.shape))

result_tile_shape = (batch_tile_size, reduction_tile_size)

iteration_bounds = [asz // tsz for asz, tsz in zip(
    [batch_size, reduction_dims, contracting_dims],
    [batch_tile_size, reduction_tile_size, contracting_tile_size]
)]
contraction_steps = iteration_bounds[2]

# Pallas currently does not allow non-consecutive grid points to write to the
# same slices of output. We therefore do not block the output along the
# reduction axis.
output_transform_indices_fn = lambda i, j, k: (i, 0)
output_tile_shape = (batch_tile_size, num_elements)
assert(len(output_tile_shape) == len(output_shape))

# At the time of writing, the Mosaic compiler does not have a rule for
# lowering comparisions between types that are not 32-bits wide. Hence, we
```

```
# explicitly promote inputs to their wider 32-bit type.
assert(lhs.dtype.itemsize <= 4)
if jnp.issubdtype(lhs.dtype, jnp.floating):
  compute_type = jnp.float32
elif jnp.issubtype(lhs.dtype, jnp.signedinteger):
  compute_type = jnp.int32
elif jnp.issubtype(lhs.dtype, jnp.unsignedinteger):
  compute_type = jnp.uint32
else:
  assert "Unknown data type"

def _kernel(lhs_ref, rhs_ref, values_ref, indices_ref, acc_ref):
  # b = batch axis, r = reduction axis, c = contracting axis
  tile_b, tile_r, tile_c = \
    pl.program_id(0), pl.program_id(1), pl.program_id(2)

  if contraction_steps > 1:
    @pl.when(tile_c == 0)
    def reset_accumulators():
      acc_ref[...] = jnp.zeros_like(acc_ref)

    # For each output tile, we reset the accumulators to zero.
    @pl.when(tile_c < contraction_steps - 1)
    def matmul_only_step():
      acc_ref[...] += jnp.matmul(
          lhs_ref[...], rhs_ref[...], preferred_element_type=jnp.float32
      )

  # When we've accumulated all the partial products, we update the top-K'
  # lists with the new elements.
  @pl.when(tile_c == contraction_steps - 1)
  def update_topk_state():
    assert(values_ref.shape == indices_ref.shape)

    @pl.when(tile_r == 0)
    def initialize_outputs():
      # We don't have to initialize the indices as non-strict comparators for
      # selection guarantee that the indices will be updated.
      values_ref[...] = jnp.full_like(values_ref, -jnp.inf)

    if contraction_steps == 1:
      acc_ref[...] = jnp.zeros_like(acc_ref)

    acc_ref[...] += jnp.matmul(
        lhs_ref[...], rhs_ref[...], preferred_element_type=jnp.float32
    )

    def update_state(inputs, iota, state_offset, state_size):
      """Update the top-K' lists with the new inputs."""

      assert(inputs.shape == iota.shape)
      assert(inputs.shape[-1] == state_size)
      assert(state_size <= num_buckets)
```

```
# Load state information for the current chunk.
values_by_k, indices_by_k = [], []
for k in range(local_K):
  values = values_ref[
      :,
      pl.ds(
          start=pl.multiple_of(k * num_buckets + state_offset, 128),
          size=state_size
      )
  ].astype(compute_type)
  indices = indices_ref[
      :,
      pl.ds(
          start=k * num_buckets + state_offset,
          size=state_size
      )
  ]
  assert(values.shape == indices.shape)
  values_by_k.append(values)
  indices_by_k.append(indices)

# Compute the new state information for the current chunk.
pred = inputs >= values_by_k[-1]
values_by_k[-1] = jax.lax.select(pred, inputs, values_by_k[-1])
indices_by_k[-1] = jax.lax.select(pred, iota, indices_by_k[-1])
for k in reversed(range(1, local_K)):
  # The commented line and uncommented line are algorithmically
  # equivalent, but the uncommented version has one less loop-carried
  # dependency.
  # pred = values_by_k[k] > values_by_k[k - 1]
  pred = inputs > values_by_k[k - 1]

  values_to_shift = values_by_k[k]
  values_by_k[k] = \
    jax.lax.select(pred, values_by_k[k-1], values_to_shift)
  values_by_k[k-1] = \
    jax.lax.select(pred, values_to_shift, values_by_k[k-1])

  indices_to_shift = indices_by_k[k]
  indices_by_k[k] = \
    jax.lax.select(pred, indices_by_k[k-1], indices_to_shift)
  indices_by_k[k-1] = \
    jax.lax.select(pred, indices_to_shift, indices_by_k[k-1])

# Write the new state information for the current chunk.
for k in range(local_K):
  values_ref[
      :,
      pl.ds(
          start=k * num_buckets + state_offset,
          size=state_size
      )
  ] = values_by_k[k].astype(values_ref.dtype)
  indices_ref[
```

```
            :,
            pl.ds(
                start=k * num_buckets + state_offset,
                size=state_size
            )
        ] = indices_by_k[k]

    if reduction_tile_size > num_buckets:
      assert(reduction_tile_size % num_buckets == 0)
      num_iterations_over_outputs = reduction_tile_size // num_buckets

      # The loop count may be large but we explicitly want to unroll to
      # eliminate a lot of state loads/stores.
      for iter_idx in range(num_iterations_over_outputs):
        # The inputs are already tiled by pallas and we use local offsets.
        inputs = acc_ref[
            :, pl.ds(start=iter_idx * num_buckets, size=num_buckets)
        ].astype(compute_type)

        iota = jax.lax.broadcasted_iota(indices_ref.dtype, inputs.shape, 1)
        iota += tile_r * reduction_tile_size + iter_idx * num_buckets
        assert(inputs.shape == iota.shape)

        update_state(inputs, iota, 0, num_buckets)
    else:
      assert(num_buckets % reduction_tile_size == 0)
      inputs = acc_ref[...].astype(compute_type)

      iota = jax.lax.broadcasted_iota(indices_ref.dtype, inputs.shape, 1)
      iota += tile_r * reduction_tile_size
      assert(inputs.shape == iota.shape)

      num_tiles_over_outputs = num_buckets // reduction_tile_size
      state_offset = (tile_r % num_tiles_over_outputs) * reduction_tile_size

      update_state(inputs, iota, state_offset, reduction_tile_size)

def wrapper(lhs, rhs):
  pr_values, pr_indices = pl.pallas_call(
    _kernel,
    grid_spec=pltpu.PrefetchScalarGridSpec(
      num_scalar_prefetch=0,
      in_specs=[
        pl.BlockSpec(lhs_tile_shape, lhs_transform_indices_fn),
        pl.BlockSpec(rhs_tile_shape, rhs_transform_indices_fn),
      ],
      out_specs=[
        pl.BlockSpec(output_tile_shape, output_transform_indices_fn),
        pl.BlockSpec(output_tile_shape, output_transform_indices_fn)
      ],
      scratch_shapes=[pltpu.VMEM(result_tile_shape, jnp.float32)],
      grid=iteration_bounds,
    ),
    out_shape=[
```

```
        jax.ShapeDtypeStruct(output_shape, compute_type),
        jax.ShapeDtypeStruct(output_shape, jnp.int32),
      ],
      compiler_params=pltpu.TPUCompilerParams(
        dimension_semantics=("parallel", "arbitrary", "arbitrary")
      ),
      **kwargs
    )(lhs, rhs)
    return pr_values, pr_indices
  return wrapper

def make_matmul_fused_generalized_approx_topk(
    lhs, rhs, num_buckets, local_K, global_K, **kwargs
):
    partial_reduce_fn = \
      matmul_fused_generalized_partial_reduce_v2(lhs, rhs, local_K, num_buckets, **kwargs)

    def wrapper(lhs, rhs):
      bucket_values, bucket_indices = partial_reduce_fn(lhs, rhs)
      values, indices = \
        jax.lax.sort_key_val(bucket_values, bucket_indices, is_stable=False)
      values = jnp.flip(values[..., -global_K:], axis=-1)
      indices = jnp.flip(indices[..., -global_K:], axis=-1)
      return values, indices
    return wrapper

def matmul_fused_generalized_approx_topk(lhs, rhs, *args, **kwargs):
  return make_matmul_fused_generalized_approx_topk(
      lhs, rhs, *args, **kwargs)(lhs, rhs)
```

A.8 Algorithm Parameter Selection

A.8.1 Monte Carlo Estimation of Expected Recall

```
def expected_recall_mc(N, B, K_global, K_local, num_trials):
  assert(N % B == 0)
  bucket_size = N // B
  X_samples = np.random.hypergeometric(
      K_global,
      N - K_global,
      bucket_size,
      size=num_trials
  )
  num_collisions = B * np.maximum(X_samples - K_local, 0)
  recall = 1 - num_collisions / K_global
  expected_recall = np.mean(recall)
  std_error = np.std(recall, ddof=1) / np.sqrt(num_trials)
  return expected_recall, std_error
```

A.8.2 Parameter Sweep

```
def select_parameters(
    input_size, K,
    recall_target,
    allowed_local_K=[1, 2, 3, 4]
```

```
):
"""Finds a good set of algorithm parameters for the given configuration.

Args:
  input_size: size of the array
  K: number of top entries
  recall_target: minimum "expected" recall required
  allowed_local_K: list of local K to consider in the search space
"""

divisors = get_all_factors(input_size)
allowed_num_buckets = [ d for d in divisors if d % 128 == 0 ]

# For a fixed K, the expected recall decreases as the number of buckets
# decreases. Therefore, by sweeping through `num_buckets` in descending
# order, we can terminate the search early when we miss the recall target.
allowed_num_buckets = sorted(allowed_num_buckets, reverse=True)

# The best configuration selection logic only checks for the total number of
# elements using a strict comparision. We want to try local K in asecending
# order so that in the case of a tie, we pick the configuration with a smaller
# local K.
allowed_local_K = sorted(allowed_local_K)

best_config = None
best_num_elements = np.inf
for local_K in allowed_local_K:
  for num_buckets in allowed_num_buckets:
    if num_buckets * local_K < K:
      break

    if recall_target >= 0.995:
      warnings.warn(
          f"recall_target of {recall_target} too high"
          " for reliable selection of algorithm.",
          RuntimeWarning
      )

    num_trials = 4096
    recall, recall_err = \
      expected_recall_mc(input_size, num_buckets, K, local_K, num_trials)
    while recall_err * 3 > 0.005:
      num_trials *= 2
      recall, recall_err = \
        expected_recall_mc(input_size, num_buckets, K, local_K, num_trials)

    if recall < recall_target:
      break

    num_elements = num_buckets * local_K
    if num_elements < best_num_elements:
      best_config = (local_K, num_buckets)
      best_num_elements = num_elements
assert(best_config is not None)
```

```
    return best_config
```

A.9 Expected recall rapidly improves with increasing K'.

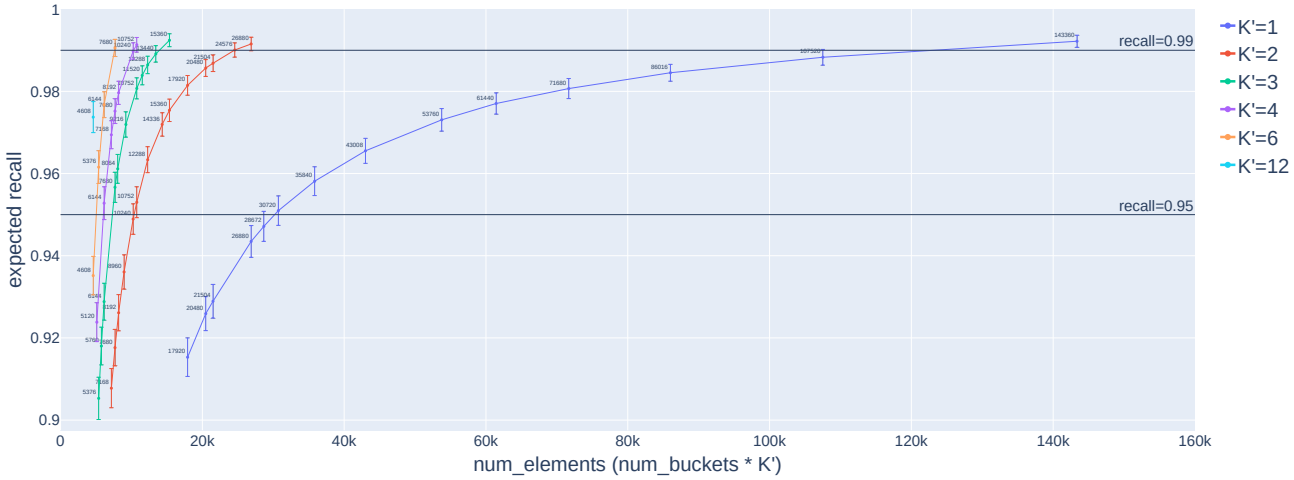

Figure 10. **Recall vs number of elements for finding top-3360 ($\approx$ 0.8%) elements from an array of size 430,080.** The data was obtained from simulated runs of the algorithm on randomly generated integers. The markers represent the sample mean and the error bars represent the sample standard deviation from 1024 trials. Each curve corresponding to a K' depicts the Pareto frontier for that K'. The ideal point is $(K, 1.0)$. Beyond a certain K', the first stage will become sufficiently expensive that the additional cost of the first stage exceeds the gains in the second stage. However, for small values of K', where the additional cost of the first stage is negligible, we note that the Pareto frontier improves as K' increases.