Faster Approximate Top-K: Harnessing the Full Power of Two-Stages

Anonymous authors
Paper under double-blind review

Abstract

We consider the Top-K selection problem, which aims to identify the largest K elements in 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 that: (i) partitions the input array into equal-sized chunks and selects the top-1 element from each partition; and (ii) sorts the resulting *smaller subset* and returns the top K elements. In this paper, we generalize the first stage so that each partition selects the top K' elements (for $1 \leq K' \leq K$). Our contributions include: (i) an expression for the expected recall of this generalized algorithm under random partitioning, and a demonstration that choosing $K' > 1$ with *fewer partitions* in the first stage more effectively reduces the input size to the second stage while maintaining the same expected recall as the original algorithm; (ii) a bound on the expected recall of the original algorithm as a function of the algorithm parameters that is provably tighter by a factor of 2 than the bound reported by Chern et al. (2022); and (iii) an implementation of our algorithm on Cloud TPUv5e that achieves approximately an order of magnitude speedup over the original algorithm without sacrificing recall.

1 Introduction

Identifying the top-K elements in an array is an essential building block of many algorithms. Beyond its common applications in Maximum Inner Product Search (MIPS) or K-Nearest-Neighbors (KNN) (Chern et al., 2022; Li et al., 2023), it has recently become important for optimizing training and inference of large foundation models. Large language models (LLMs) use the Top-K operation to exploit the sparsity in several components, including 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 in retrieval augmented generation (Lewis et al., 2021; Borgeaud et al., 2022), sampling tokens (Shen et al., 2024), mixture-of-experts (He, 2024), and accelerating distributed training (Shi et al., 2019; Ruan et al., 2023).

Given the scale of these models, their training and inference are typically performed on accelerators such as TPUs and GPUs. However, computing Top-K on these devices can be expensive. 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, computing Top-K should be negligible compared to the matrix multiplications.

As a workaround, foundation models increasingly use *approximate* Top-K algorithms, which are generally robust to these approximations (Samaga B L et al., 2024; Key et al., 2024).

Chern et al. (2022) introduced a hardware-friendly approximate Top-K algorithm that operates in *two stages*. In the first stage, the input array is divided into a fixed number of equal-sized buckets, and the top-1 element

[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.

from each bucket is selected. In the second stage, these top-1 elements are sorted, and the first K elements are returned. They quantify the approximation error in terms of *expected recall* (Wang et al., 2014), 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, which is then used to *choose a number of buckets* that is sufficient to maintain a *user-specified average recall target*. We refer to this method as *the original algorithm*.

At the time of writing, this algorithm was implemented in `jax.lax.approx_max_k` for TPUs. In the earlier example of finding the top-2% of the hidden activations, this algorithm with a recall target of 95% still takes 9× more time than the matrix multiplication on TPUv5e.

As we explain in Sections 2 and 6, the second stage is typically the bottleneck, since the first-stage computation is relatively inexpensive and efficiently parallelizable. In fact, in tasks that require finding the top-K elements in each column or row of a matrix product, the first stage can be "fused"[2] (Snider & Liang, 2023) with the matrix multiplication, effectively hiding much of its cost. Therefore, improving the efficiency of this algorithm requires reducing the number of elements sorted in the second stage without sacrificing the expected recall, while ensuring that the first-stage computation remains inexpensive. Our main contribution is a new algorithm that achieves this, which we describe next.

We accomplish this by generalizing the first stage of the approximate Top-K algorithm of Chern et al. (2022) to select top-K' elements from each bucket (where $K' < K$) instead of restricting selection to the 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, it is possible to reduce the number of buckets B sufficiently that the smallest number of elements to sort in the second stage ($B \cdot K'$) is achieved by some $K' > 1$ while ensuring that the first stage does not become the bottleneck.*

Theoretically, we derive an expression relating K', K and B to the expected recall. Using this expression, we select the parameters K' and B for our algorithm that satisfy the user-specified average recall target. While the full potential is realized by choosing $K' > 1$, interestingly, even for $K' = 1$, the setting of Chern et al. (2022), our bound on the number of buckets is provably a factor of 2 tighter than theirs, which improves parameter selection and, in most cases, more than doubles the performance of the original algorithm. For $K' > 1$, the gains are even higher.

Empirically, we implement our generalized algorithm ($K' > 1$) for TPUs using Pallas and demonstrate an order of magnitude speedup in latency on TPUv5e chips compared to the original algorithm. We provide two implementations: (i) an unfused version that executes the two stages as two separate *kernels* (background on kernels is in Section 2); and (ii) a matmul-fused version that fuses the first stage with a matrix multiplication (background on fusion and related matmuls are in Sections 2 and 3). In the earlier example of identifying the top 2% of hidden activations, our implementations for $K' = 4$ makes the Top-K step 24× faster than `jax.lax.approx_max_k`. This reduces the time taken by approximate Top-K to below that of the corresponding matrix multiplication, resulting in an overall speedup of 6.7×. We share the code for our TPU implementations in Appendix Sections A.6 and A.7. Although our implementation and evaluation focus on TPUs, the underlying ideas, analysis and performance modeling framework are hardware-agnostic and would apply to GPUs as well. Key et al. (2024) validate the unfused version of the same algorithm on GPUs and demonstrate speedups with an open-source implementation.

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 compute is concentrated in two

[2]In this context, *fusion* refers to merging the top-K selection logic with the matrix multiplication logic into a single fused operation such that the top-K step can, in some cases, be performed at no additional cost. We discuss this in more detail in Section 2.

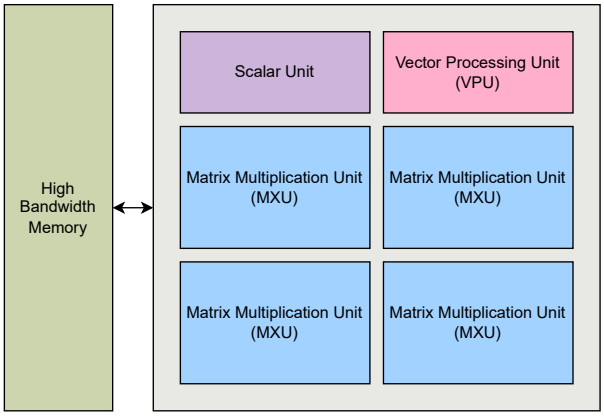

Figure 1: **Overview of TPUv5e subsystems.** The TPUv5e chip features four Matrix Multiplication Units (MXUs) dedicated to matrix-matrix multiplications along with a Vector Processing Unit (VPU) that performs general vector operations, such as activations and softmax. The chip also includes a scalar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

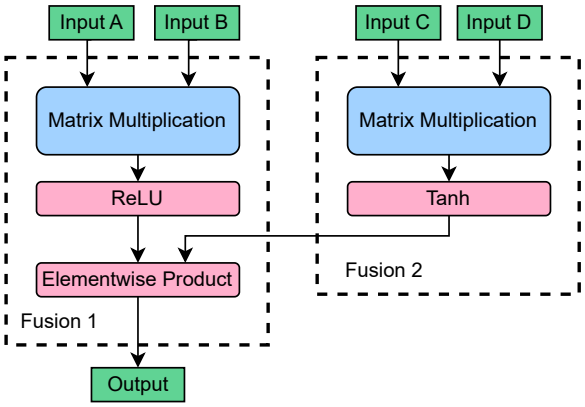

Figure 2: **Decomposing a program into smaller subprograms.** In this example, a program has been broken down into two subprograms, each containing several elementary operations. The subprograms are executed in an order that satisfies their dependencies, with Fusion 2 executing before Fusion 1.

subsystems: the Matrix Multiply Unit (MXU) and the Vector Processing Unit (VPU) (Norrie et al., 2020; Google Cloud, 2024b). MXUs account for more than 95% of compute FLOPS (Chern et al., 2022; Google Cloud, 2024b); consequently, only MXU-bound programs reach peak FLOPS utilization. Similarly, on GPUs, compute resources are primarily spread across two subsystems: TensorCores for matrix multiplications and CUDA cores for scalar/vector computations. As with TPUs, most FLOPS are concentrated in TensorCores (NVIDIA, 2024).

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 can consist of many elementary operations and can simultaneously use all subsystems. For example, for a matrix multiplication followed by bias addition and activation, the entire computation can be carried out in a single *fused* kernel (Snider & Liang, 2023; Google Cloud, 2024a), where matrix unit outputs are immediately processed by scalar/vector units (for bias addition and activation) before writing them to memory. This minimizes the overhead associated with launching and terminating kernels, avoids multiple round-trips to the memory, and enables more effective simultaneous use of the different compute resources. Figure 2 shows this process.

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 kernel's overall runtime. To accurately model performance and identify the bottleneck, we must quantify the capabilities of each accelerator subsystem and how the kernel uses it (Chern et al., 2022).

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 used 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, it can be omitted.

In this section, we restrict ourselves to TPUs and focus on three key subsystems: (i) HBM memory, (ii) the VPU and (iii) the MXU. We define the following parameters to quantify the throughput of each:

- 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, its total runtime is determined by the subsystem that requires the most time to complete its work.[3] 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 to be *memory-bound* if its memory subsystem cannot feed data sufficiently fast to keep 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 the utilization of non-bottleneck subsystems may *not* necessarily increase the kernel's runtime. To formalize this, we define *ridge points* of equation 1, which are configurations where the runtimes of any two subsystems are equal. For example, we can estimate as $\frac{\gamma}{\pi}$ the maximum number of VPU operations that can be performed per MXU operation to remain MXU-bound. Since the MXU has much higher throughput, 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 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. Ridge points thus provide a simple way to understand the balance between different subsystems and easily reason about how changes in a subsystem's utilization can impact overall performance (Williams et al., 2009). 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. Let U represent the set of actual top-K elements of y, and let V represent the top-K elements returned by an approximate algorithm. We define *expected recall* as the expected fraction of the true top-K elements retrieved by the algorithm, assuming that the top-K elements are placed randomly and uniformly in y:

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

Table 1: **Peak throughput and ridge points of different subsystems in accelerators.** The γ for TPUv4 was taken from Chern et al. (2022), and the γ for TPUv5e was estimated by timing VPU-bound kernels (see Appendix A.1). All other values derived directly from the official datasheets.

Device	β	γ (TFLOP/s)	π (TFLOP/s)	$\frac{\gamma}{\left(\frac{\pi}{256}\right)}$	$\frac{\gamma}{\left(\frac{\beta}{4}\right)}$
		fp32	bf16	ops per 128-d dot	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

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

Objective. The goal is to *minimize the time* required for this operation while *maximizing expected recall*, thereby improving the Pareto frontier describing the trade-off between latency and expected recall objectives.

4 Related Work

As of this writing, `jax.lax.top_k` is the only exact Top-K implementation available for TPUs. Chern et al. (2022) proposed a faster approximate Top-K algorithm, which is currently exposed in JAX as `jax.lax.approx_max_k`. To the best of our knowledge, these are the only Top-K implementations available for TPUs.

A large body of prior work focuses on exact Top-K algorithms for GPUs (Xie et al., 2024; Gaihre et al., 2021; Zhang et al., 2023; Li et al., 2024; Dashti et al., 2013; Alabi et al., 2012; Shanbhag et al., 2018; Ribizel & Anzt, 2019). Since the algorithmic framework introduced by Chern et al. (2022), which we generalize in this work, can transform any exact algorithm into a faster approximate variant, we do not compare with exact algorithms. We provide a detailed description of the Chern et al. (2022) algorithm in Section 5.

Key et al. independently developed the same generalized algorithm, implement it for GPUs in the unfused setting, and evaluate quality in several downstream tasks including SparQ attention Ribar et al. (2024), token sampling and link-prediction in knowledge graphs. Their work provides an API, `approx_top_k(array, K, K', B)`, that requires manual tuning of algorithm parameters. In contrast, our work provides a more rigorous theoretical analysis along with detailed runtime performance modeling, which together enable fast automatic parameter selection and a more user-friendly interface, `approx_top_k(array, K, recall_target)`. Beyond the unfused setting, we also implement a matmul-fused implementation that extracts non-trivial speedups in realistic workloads. The ridge-point analysis in Section 2.3, combined with our analysis of the implementation in Section 6.3, provides a principled framework for determining the available VPU headroom in a fusion and enables fusion-aware parameter selection, which could, in principle, be adapted into a cost model to support automatic compiler-driven fusion decisions.

5 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. 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. Approximations occur when multiple top-K elements fall into the same bucket since only one is selected and the rest are discarded. Increasing the number of buckets reduces the likelihood of such collisions and can be chosen to achieve a desired average recall target. Figure 3 presents the algorithm.

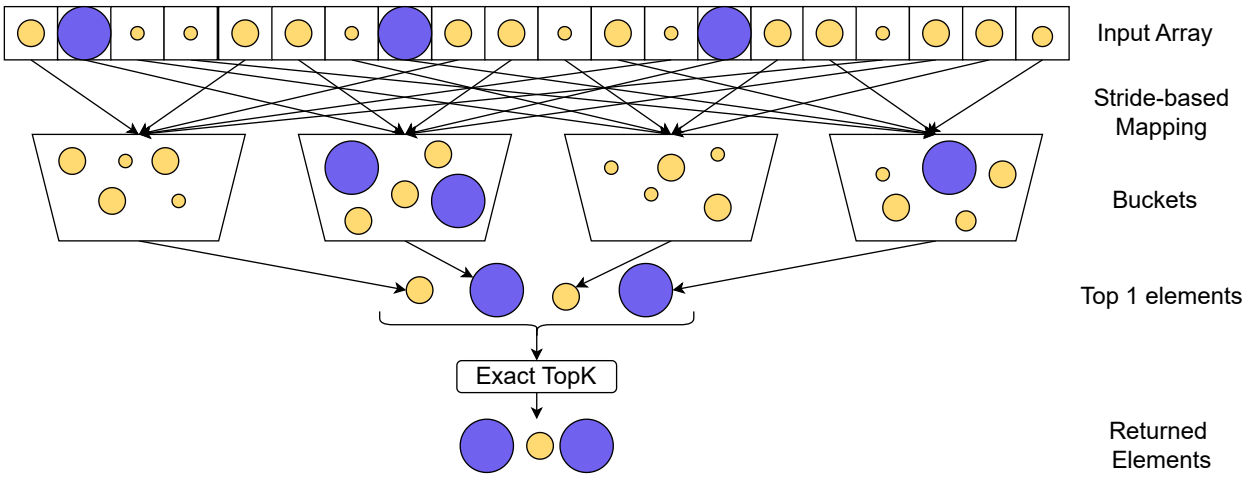

Figure 3: **The two-stage approximate Top-K algorithm by (Chern et al., 2022).** This example demonstrates the process of finding the approximate top three elements from an array of twenty elements using the algorithm by Chern et al. (2022). Ten buckets are required to guarantee an expected recall of 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.

The algorithm accepts `recall_target` as a parameter, denoted by r, which specifies the desired expected recall. The required number of buckets, denoted by B, is calculated using a closed-form expression that relates B to the expected recall under a model in which the positions of the top-K elements are independently and uniformly distributed over the input array:

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

The input often results from a matrix multiplication, e.g., a maximum inner-product search (MIPS) or the Top-K on key-query logits in attention. The algorithm's first stage, 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 might incur little to no additional cost since it utilizes the otherwise idle scalar/vector compute units while the matrix units are busy.

To design their algorithm and implementation, the authors adopt 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, as detailed below:

1. Their analysis focuses on matrix multiplications with 128-dimensional dot products, which on most hardware provide only 4-8 vector operations per output element (each 128-d dot product), consistent with their budget of three operations per element. However, we frequently work with larger dimensions, where the available scalar/vector compute is nearly $\frac{\text{dims}}{128}$ times higher than the numbers they estimate.

2. Even for 128-dimensional dot products, the first stage may not saturate the scalar/vector units on all hardware platforms. See Table 1.

3. In memory-bound computations, more scalar/vector compute is 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 might 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 *the top-K' elements from each bucket instead of just the top-*1.

6 Method

We describe our algorithm in Section 6.1 and provide an analysis in Section 6.2. In Section 6.3, we discuss the key ideas in our implementation of the algorithm.

6.1 Algorithm

Step 1: Partition into buckets

Given $A = [a_1, \ldots, a_N]$, partition into B buckets:

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

Step 2: Select Top-K' per bucket

Select the top-K' elements from each bucket:

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

Step 3: Merge selections

Merge the top-K' elements from all buckets:

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

Step 4: Return approximate top-K

Sort the merged set and return the top-K elements:

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

6.2 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, the K special balls represent the top-K elements, and B buckets correspond to the "buckets" in the algorithm.

Let X_b be a random variable that denotes the number of special balls in 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.

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

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 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 because the distribution of special balls in the first bucket must be the same as in all other buckets by symmetry, and it is apparent that the

distribution of special balls in the first bucket must follow 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 follows.

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

In Appendix A.2, 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 count only 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:* $\mathbb{E}[\textit{Recall}] = 1 - \frac{B}{K} \times \sum_{r=K'+1}^{\min(K,N/B)} (r - K') \frac{\binom{K}{r}\binom{N-K}{\frac{N}{B}-r}}{\binom{N}{\frac{N}{B}}}$. *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 $K' = 1$, the bound is a factor of 2 tighter than that in Chern et al. (2022).

We provide the proof in Appendix A.3.

6.3 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 since 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, we replace the smallest element (and its index) with the new element (and its index).

2. We 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 `TopKPrimeUpdate`

1: **Input:** input, index, values[K'], indices[K']
2: **Output:** values[K'], indices[K']
3: **Precondition:** values sorted in descending order
4: **if** input $\geq$ values[K'] **then** /* one compare */
5: values[K'] = input /* one select */
6: indices[K'] = index /* one select */
7: **end if**
8: **for** $k = K'$ **to** 2 **do**
9: **if** values[k] > values[k-1] **then** /* one compare */
10: swap(values[k], values[k-1]) /* two selects */
11: swap(indices[k], indices[k-1]) /* two selects */
12: **end if**
13: **end for**

Algorithm 2 Vectorized Top-K' Update

1: **Input:** input array of size N, number of lanes L, number of buckets B, top-K' values (-∞ initialized) and indices lists of shape [B, K']
2: **Output:** top-K' values and indices lists
3: $num_chunks \leftarrow N/L$
4: **for** $in_chunk \leftarrow 0$ **to** $num_chunks - 1$ **do**
5: $out_chunk \leftarrow in_chunk \bmod (B/L)$
6: **Load** inputs corresponding to in_chunk
7: **Load** current top-K' lists for out_chunk
8: **Compute** updated top-K' lists using trivially vectorized `TopKPrimeUpdate`
9: **Store** updated top-K' list at out_chunk
10: **end for**

Since the `values` list is stored in descending order, an input element larger than the k-th value in the list is also larger than all subsequent values. This property allows the comparison in Line 9, Algorithm 1 to be done using the input element as the LHS, which eliminates a loop-carried dependency.

Note that Algorithm 1 does not include an early return if the condition on Line 4 fails, nor does it exit the bubble sort loop early when the condition in Line 9 fails. This is required to vectorize the routine across buckets. An early return would theoretically reduce operations for individual buckets but would prevent vectorization across buckets.

Once all inputs are processed, we obtain the final result by separately merging all `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 as having 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 minor-most 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. Algorithm 2 outlines the vectorized version. 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 loop iterations so that the input chunks corresponding to the same bucket are executed consecutively; this lets the top-K' lists 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. We share the Python code for our algorithm with detailed comments in Listing A.6 for the unfused implementation and Listing A.7 for the matmul-fused implementation. To find the algorithm parameters 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 6.2. We share the Python code to estimate expected recall and select algorithm parameters in Listing A.8. The compute cost of this procedure is relatively cheap; see Appendix Section A.8.2 for a detailed discussion.

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 $B \times K'$. A smaller `#num_elements` leads 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 supports only $K' = 1$), 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

7 Results and Discussion

In Section 7.1, we show that our algorithm substantially reduces the input size for the same expected recall compared to the improved baseline, which is the original algorithm by Chern et al. (2022) with our improved bound. Section 7.2 discusses the performance of our unfused Pallas implementation for TPUs, and Section 7.3 does the same for our matmul-fused implementation.

7.1 Theoretical effectiveness 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 algorithm variants, up to $K' = 4$, reduce the size of the inputs across a wide range of configurations ($\frac{K}{N} \in \{0.1\%, \ldots, 25\%\}$ and array sizes $\in [256, 4e9]$). We also account for the implementation constraints necessary for simplicity and performance; therefore, the numbers indicate *real*

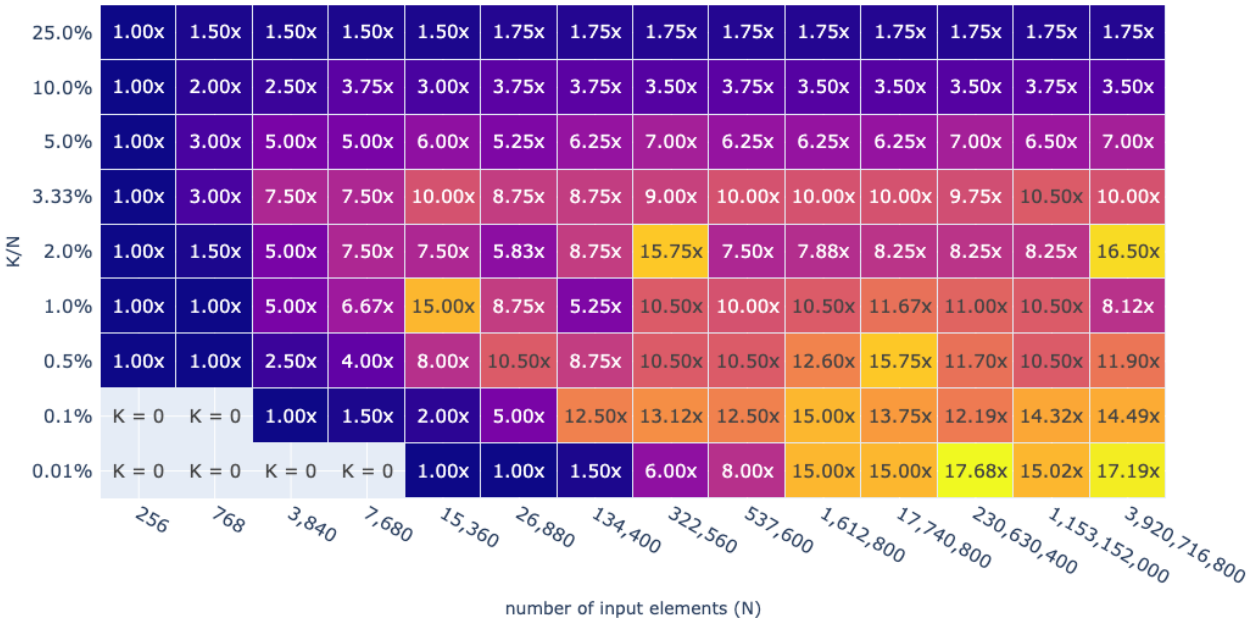

Figure 4: **Factor of reduction in output elements over the baseline ($K' = 1$) for 99% expected recall target.** The heatmap shows the factor by which our generalized algorithm with $1 \leq K' \leq 4$ reduces the number of elements in the first stage over the reductions provided by the baseline, i.e., a value of 2× 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 lead to more output elements than required by $K' = 1$; in such cases we default to $K' = 1$.

realizable reductions using our implementation. The figure demonstrates that our algorithm significantly reduces the number of elements in virtually all configurations, with a median reduction of 7×. It performs worse only 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'.

7.2 Improved latency of finding the Top-K elements

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

The first stage cost remains nearly constant from $K' = 1$ to $K' = 6$ due to its memory-bound nature. According to the performance model in Section 2.3 and Table 1, the first stage must be memory bound until we exceed 30 VPU operations per 4-byte element, which occurs around $K' = 6$, according to the operation count formula in Section 6.3. Therefore, we expect the latency of the first stage to be independent of K' until we reach $K' = 6$.

7.3 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.

Table 3: **The runtime of our algorithm on TPUv5e to identify the 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. The `jax.lax.approx_max_k` row presents the performance of the official JAX implementation for the $K' = 1$ setting. The remaining rows present the performance of our implementation.

Algorithm	Matmul	Stage 1	Stage 2	Total
`jax.lax.top_k`	7.32ms	-	587ms	594ms
`jax.lax.approx_max_k`	9.06ms	fused	118ms	127ms
$K' = 1$	7.32ms	6.58ms	50.0ms	64ms
$K' = 1$	9.03ms	fused	50.0ms	59ms
$K' = 4$	7.31ms	10.80ms	3.51ms	22ms
$K' = 4$	6.55ms	fused	3.51ms	10ms

We evaluate our algorithm on a MIPS workload of one million 128-dimensional vectors and 1024 queries. Table 3 reports runtimes on TPUv5e. Exact Top-K (`jax.lax.top_k`) takes 80× longer in the second stage than the matmul (587ms vs 7.32ms). `jax.lax.approx_max_k` reduces this to 13× (118ms), and our $K' = 1$ unfused implementation to 6× (50ms).

Moving to $K' = 4$, the second stage (3.51ms) falls below half the matmul cost, leaving the matmul (7.31ms) and first stage (10.80ms) as the bottleneck. Fusing the first stage with the matmul eliminates its cost and improves matmul performance (6.55ms). The gains from fusion can be significant in practice. The MIPS matmul multiplies `[B, D]` by `[D, N]` (queries, vector size, database size), giving arithmetic intensity $\frac{2}{E}\min(B, D)$ (derived in Appendix A.10), where E is the element size. In large-scale deployments with D in the low hundreds, the matmul is often memory-bound. Fusion avoids writing the large output tensor to memory, increasing arithmetic intensity and shifting the matmul closer to compute-bound.

8 Conclusion

We present a generalization of the approximate Top-K algorithm of Chern et al. (2022) that selects the top-K' elements per bucket in the first stage, instead of restricting to top-1. We theoretically analyze the expected recall of the generalized algorithm deriving an exact expression for it, and obtain a bound for $K' = 1$ which is provably tighter by a factor of 2 compared to that in Chern et al. (2022). For $K' > 1$, we demonstrate both theoretically and empirically that the algorithm substantially reduces the second-stage input size across a wide range of configurations, spanning $\frac{K}{N}$ ratios from 0.01% to 25%, with a median reduction of 7× over the $K' = 1$ baseline. We implement the algorithm for Cloud TPUv5e with a principled performance modeling framework that accurately predicts runtimes, enabling a user-friendly `approx_top_k(array, K, recall_target)` interface that requires no manual tuning. We additionally present a matmul-fused implementation that further eliminates the first-stage cost. While our kernel implementations are specific to TPUs, the algorithm and its analysis are general and we believe that it has applicability to other hardware platforms such as GPUs.

Limitations and Future Work. The empirical evaluation is focused on Cloud TPUv5e; extending empirical validation and the matmul-fused kernel to GPUs is an important open direction. The recall analysis assumes uniform random placement of top-K elements; efficient parallel randomization schemes that bring real-world inputs in line with this model would fully close the gap between theory and practice. Finally, integrating the algorithm into an optimizing compiler would enable automatic fusion decisions and fusion-aware parameter selection. Our generalization of the first stage represents the simplest step in a richer design space; more sophisticated first-stage selection schemes beyond the top-K' per bucket approach could push the performance envelope further.

References

Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russell Steinbach. Fast k-selection algorithms for graphics processing units. *ACM J. Exp. Algorithmics*, 17, 2012. URL https://api.semanticscholar.org/CorpusID:17158089.

Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory, 2024. URL https://arxiv.org/abs/2312.11514.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2022. URL https://arxiv.org/abs/2112.04426.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Felix Chern, Blake Hechtman, Andy Davis, Ruiqi Guo, David Majnemer, and Sanjiv Kumar. Tpu-knn: K nearest neighbor search at peak flop/s, 2022. URL https://arxiv.org/abs/2206.14286.

Ali A. Dashti, Ivan Komarov, and Roshan M. D'Souza. Efficient computation of k-nearest neighbour graphs for large high-dimensional data sets on gpu clusters. *PLoS ONE*, 8, 2013. URL https://api.semanticscholar.org/CorpusID:1075415.

Anil Gaihre, Da Zheng, Scott Weitze, Lingda Li, Shuaiwen Leon Song, Caiwen Ding, Xiaoye Sherry Li, and Hang Liu. Dr. top-k: Delegate-centric top-k on gpus. *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021. URL https://api.semanticscholar.org/CorpusID:237563148.

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.

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

Oscar Key, Luka Ribar, Alberto Cattaneo, Luke Hudlass-Galley, and Douglas Orr. 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.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

Yang Li, Kangbo Liu, Ranjan Satapathy, Suhang Wang, and Erik Cambria. Recent developments in recommender systems: A survey, 2023. URL https://arxiv.org/abs/2306.12680.

Yifei Li, Bole Zhou, Jiejing Zhang, Xuechao Wei, Yinghan Li, and Yingda Chen. Radik: Scalable and optimized gpu-parallel radix top-k selection. *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024. URL https://api.semanticscholar.org/CorpusID:270227092.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient llms at inference time, 2023. URL https://arxiv.org/abs/2310.17157.

Lovish Madaan, Srinadh Bhojanapalli, Himanshu Jain, and Prateek Jain. Treeformer: Dense gradient trees for efficient attention computation, 2023. URL https://arxiv.org/abs/2208.09015.

Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David Patterson. 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.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference, 2024. URL https://arxiv.org/abs/2312.04985.

Tobias Ribizel and Hartwig Anzt. Approximate and exact selection on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 471–478, 2019. doi: 10.1109/IPDPSW.2019.00088.

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

Mengzhe Ruan, Guangfeng Yan, Yuanzhang Xiao, Linqi Song, and Weitao Xu. Adaptive top-k in sgd for communication-efficient distributed learning, 2023. URL https://arxiv.org/abs/2210.13532.

Yashas Samaga B L, Varun Yerram, Chong You, Srinadh Bhojanapalli, Sanjiv Kumar, Prateek Jain, and Praneeth Netrapalli. Hire: High recall approximate top-k estimation for efficient llm inference, 2024. URL https://arxiv.org/abs/2402.09360.

Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. *Proceedings of the 2018 International Conference on Management of Data*, 2018. URL https://api.semanticscholar.org/CorpusID:44100892.

Ethan Shen, Alan Fan, Sarah M. Pratt, Jae Sung Park, Matthew Wallingford, Sham M. Kakade, Ari Holtzman, Ranjay Krishna, Ali Farhadi, and Aditya Kusupati. Superposed decoding: Multiple generations from a single autoregressive inference pass, 2024. URL https://arxiv.org/abs/2405.18400.

Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. Understanding top-k sparsification in distributed deep learning, 2019. URL https://arxiv.org/abs/1911.08772.

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

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey, 2014. URL https://arxiv.org/abs/1408.2927.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Xi Xie, Yuebo Luo, Hongwu Peng, and Caiwen Ding. Rtop-k: Ultra-fast row-wise top-k selection for neural network acceleration on gpus. In *International Conference on Learning Representations*, 2024. URL https://api.semanticscholar.org/CorpusID:272367162.

Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. Parallel top-k algorithms on gpu: A comprehensive study and new methods. *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2023. URL https://api.semanticscholar.org/CorpusID:264591510.

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
```

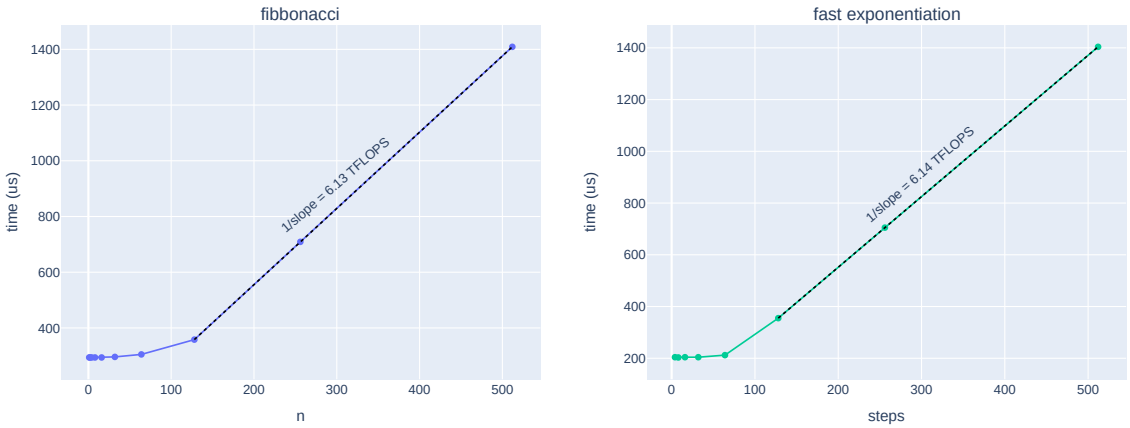

Figure 5: **Estimating the throughput of the VPU on TPUv5e.** We expect the kernels to be memory-bound (constant line) initially and then be vector compute bound (linear scaling). We fit a line to the points in the linear region with the following model: $time = num_ops \times \frac{1}{throughput} +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

A.2 Empirical verification of the quality of Monte Carlo estimates for expected recall

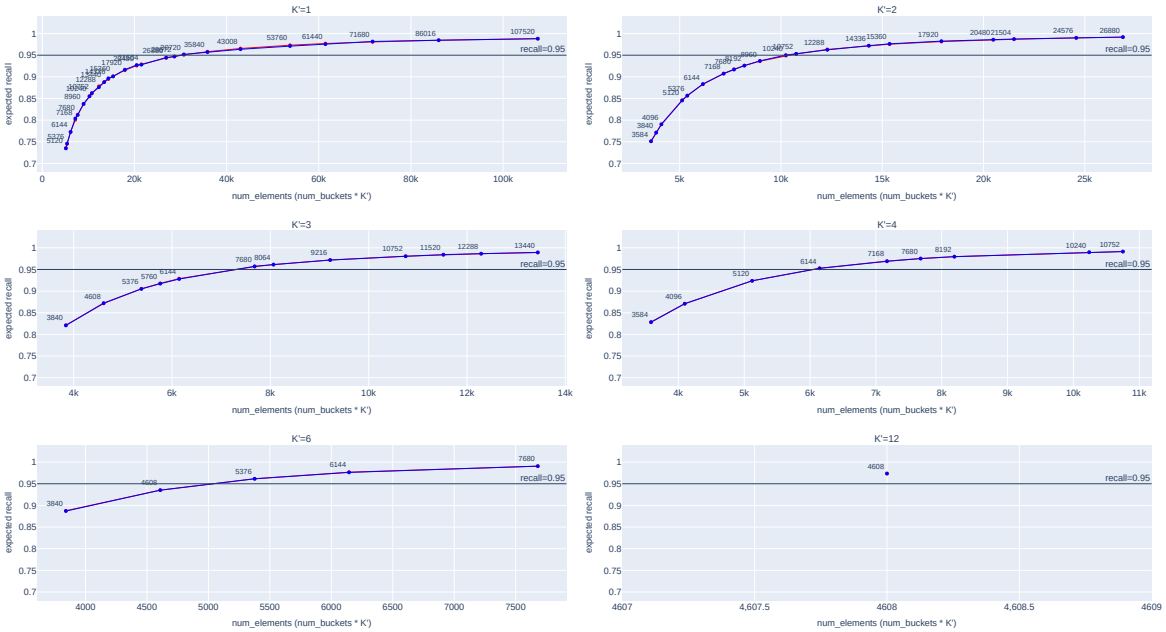

Figure 6: **Verification of Monte Carlo estimates of expected recall against simulated runs of the algorithm for finding top-3360 ($\approx$ 0.8%) elements from an array of size 430,080.** The simulated estimates were obtained from 1024 runs of the algorithm on randomly generated integers and the Monte Carlo estimates were obtained from 262144 samples of the expectation expression derived in Section 6.2.

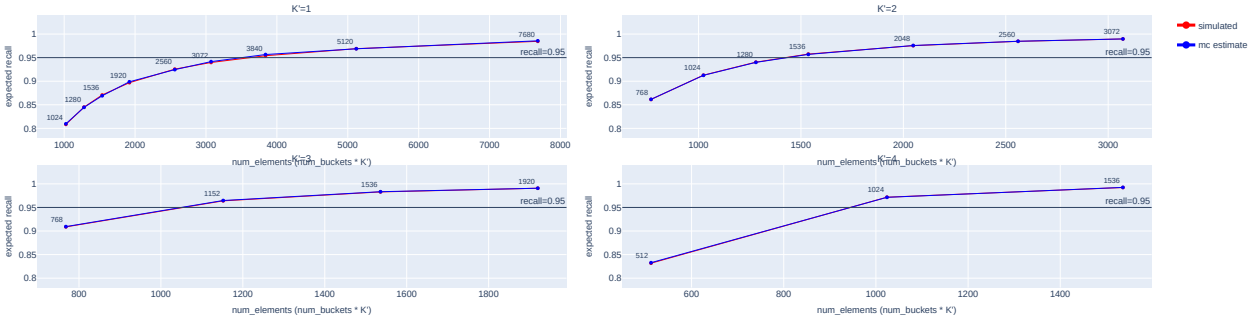

Figure 7: **Verification of Monte Carlo estimates of expected recall against simulated runs of the algorithm for finding top-480 ($\approx$ 3%) elements from an array of size 15,360.** The simulated estimates were obtained from 1024 runs of the algorithm on randomly generated integers and the Monte Carlo estimates were obtained from 262,144 samples of the expectation expression derived in Section 6.2.

A.3 Proof of Theorem 1

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':

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

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:

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

$$\Rightarrow m_j = \mathbb{E}\left[|S_j|\right] - 1 + \mathbb{P}\left[|S_j| = 0\right] \tag{3}$$

$$\Rightarrow m_j = \frac{K}{B} - 1 + \frac{\binom{N-K}{\frac{N}{B}}}{\binom{N}{\frac{N}{B}}} \tag{4}$$

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

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

$$\Rightarrow m_j \leq \frac{K^2}{2B}\left(\frac{1}{B} - \frac{1}{N}\right). \tag{7}$$

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

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

or equivalently, if we choose

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

we will then 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 6 provides a near-perfect approximation of the exact expression that is practically indistinguishable. □

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

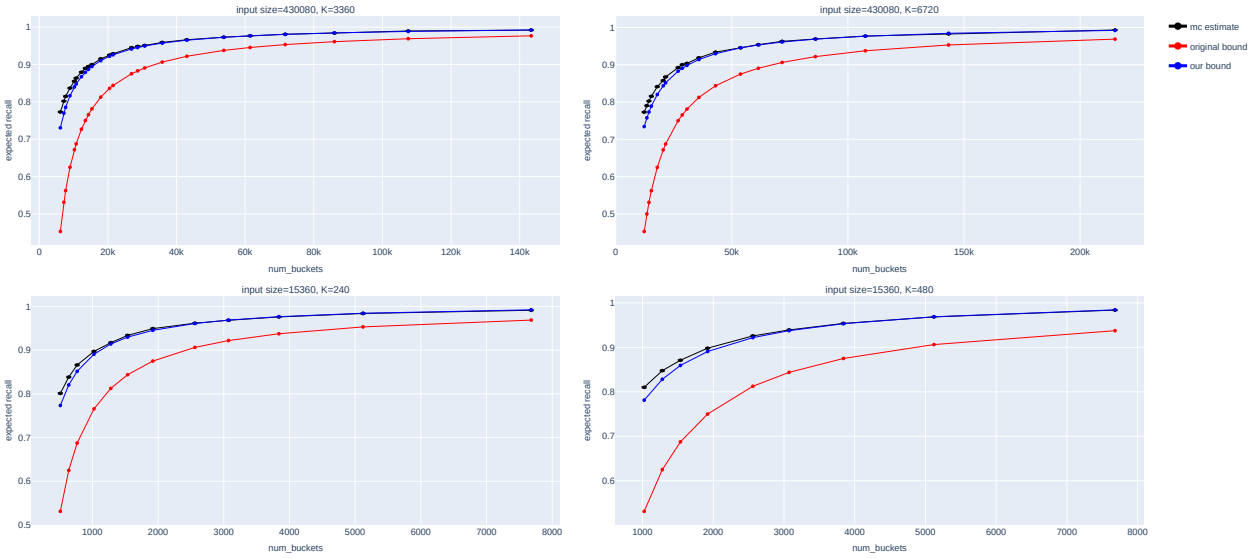

Figure 8: **Tightness of our theoretical bound on expected recall for $K' = 1$ setting compared to the original bound derived in Chern et al. (2022)** See Section 6.2 for the derivation of our bound ($\mathbb{E}[\text{Recall}] \geq 1 - \frac{K}{2B}$) which is tighter than the original bound ($\mathbb{E}[\text{Recall}] \geq 1 - \frac{K}{B}$).

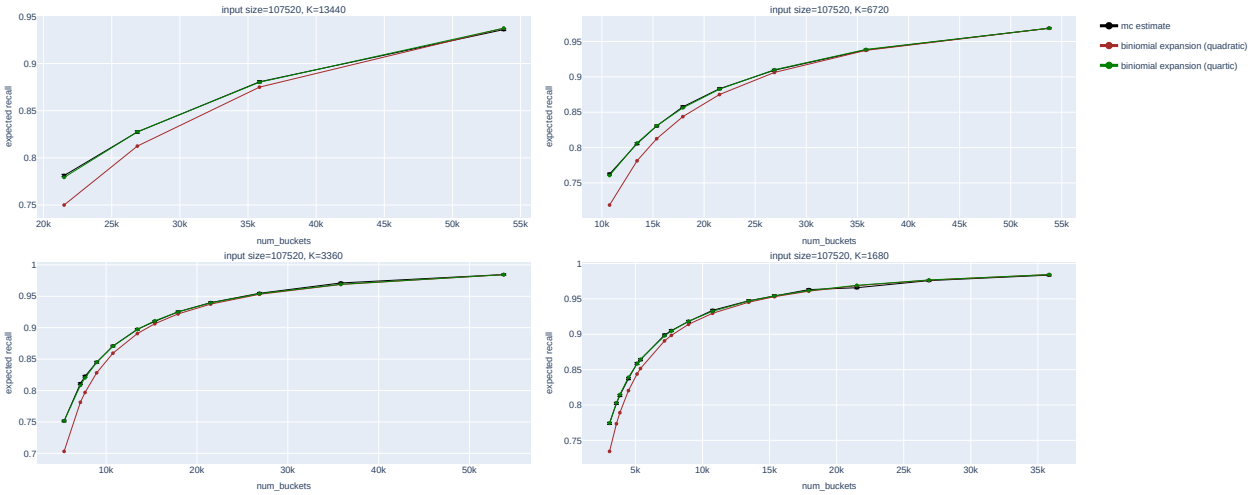

Figure 9: **Expanding the binomial expression in Step 6 to quartic terms in Theorem 1 approximates the expected recall that is nearly exact.**

A.5 Packages and utility functions for the code listings

```python
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

```python
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)
```

```python

    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

```python
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`.
  """
```

```
34
35    # Pallas imposes constraints on block specifications. Refer to the docs.
36    PALLAS_TPU_BLOCKSPEC_MAJOR_MULTIPLE = 8
37    PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE = 128
38
39    # This implementation maps buckets to the minormost axis.
40    assert(num_buckets % PALLAS_TPU_BLOCKSPEC_MINOR_MULTIPLE == 0)
41
42    batch_size, contracting_dims = lhs.shape
43    contracting_dims_rhs, reduction_dims = rhs.shape
44    assert(contracting_dims == contracting_dims_rhs)
45    assert(reduction_dims % num_buckets == 0)
46    assert(num_buckets < reduction_dims)
47    assert(lhs.dtype == rhs.dtype)
48
49    num_elements = num_buckets * local_K
50    output_shape = (batch_size, num_elements)
51
52    # We will block the matrices for software pipelining as follows:
53    # lhs: [batch_tile_size, contracting_tile_size]
54    # rhs: [contracting_tile_size, reduction_tile_size]
55    # result-scratch: [batch_tile_size, reduction_tile_size]
56    #
57    # Note that partial reduce computation can start only after the loop
        over the
58    # contracting axis ends, as it requires fully accumulated sums to begin.
59    # The VPU may idle waiting for the result tile in all but the last
        iteration
60    # of the loop over the contracting axis.
61    #
62    # We can alleviate the problem by pipelining the computation into matmul
         and
63    # TopK stages. We let the VPU processes the previous result tile while
        the new
64    # result tile is being accumulated. We do not implement the idea here
        and
65    # choose to use large tiling along contracting axis to minimize wasted
        cycles.
66
67    # The tiles will be further subtiled automatically by the compiler to
        meet the
68    # shape of the hardware matmul units. Since the subtiling loops will be
        fully
69    # unrolled, the compiler would ideally generate code to run TopK on the
70    # previous subtile while a new subtile is being accumulated.
71
72    batch_tile_size = tunable_params.get('batch_tile_size', None)
73    if batch_tile_size is None:
74      # We want this to be as large as possible. This parameter controls the
75      # arithmetic intensity of the blocked matmul operation. Therefore, we
          must
76      # have a batch tile size that is high enough to ensure that each
          blocked
77      # 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
```

The parameter sweep evaluates configurations in descending order of bucket count for each value of K', terminating early once the recall target is no longer met. This works since recall decreases monotonically with fewer buckets. For each configuration evaluated, we use an adaptive sampling procedure that draws hypergeometric samples until the recall estimate is within ± 0.005 at 3-sigma confidence. To give a sense of the cost, we evaluated eight representative configurations spanning array sizes from 16k to 917k and K from 128 to 3360 at 95% recall target; the parameter sweep evaluated between a few dozen and up to 115 configurations per (N, K, recall_target) combination, drawing between 930k and 5M hypergeometric samples in total, and completed in under a second on an AMD Ryzen 7 3700X desktop CPU. This cost is negligible relative to the typical compilation times for LLMs, which range from a few minutes to several hours. In practice, the selected parameters can be cached and reused across all calls with the same configuration, which is common in LLMs where repeated identical blocks of layers dominate the architecture. The implementation has not been optimized; straightforward improvements such as binary search over bucket counts or sharing random samples across configurations could further reduce the cost.

A.8.3 Discussion and Generalization

The parameter sweep is currently specific to TPUv5e. The model in Section 2.3 and Table 1 determines the maximum K' for which the first stage remains memory-bound and therefore costs approximately the same regardless of K' (confirmed empirically in Table 2, where Stage 1 latency is nearly flat from $K' = 1$ to $K' = 6$). Within this regime, minimizing the number of output elements ($B \times K'$) is equivalent to minimizing total latency, so the sweep correctly optimizes the right objective without explicitly consulting Table 1 at search time. On TPUv5e, the ridge point analysis gives $K' = 6$ as the crossover between memory-bound and compute-bound first-stage behavior; we conservatively restrict to $K' \leq 4$ to leave headroom. The constraint that the number of buckets be a multiple of 128, and a divisor of the input size, arises from TPUv5e-specific alignment requirements in our implementation.

The underlying methodology nonetheless generalizes. For any accelerator for which the quantities in Table 1 can be measured or obtained from the datasheets, one can: (i) compute the ridge point to determine the K' ceiling below which the first stage remains memory-bound; (ii) identify the implementation constraints; and (iii) run the same sweep to find the configuration minimizing second-stage input size. We can in principle extend this existing sweep routine to other TPUs directly using corresponding numbers from Table 1.

The benefits of hardware-aware parameter selection are most significant in fused settings. When operations such as ReLU, sigmoid, or elementwise products (e.g., in GLU-based MLP blocks) are fused alongside the Top-K first stage, they compete for VPU instructions and reduce the effective vector compute budget available for Top-K, thereby lowering the effective K' ceiling. Conversely, when fusing with matrix multiplications with large contracting dimensions, substantially more VPU compute becomes available than the baseline estimate from Section 2.3 suggests (as discussed in Section 5), potentially enabling K' well beyond 6. A complete fusion-aware parameter selection procedure would account for these competing demands when determining the K' ceiling, and could in principle be integrated into a compiler cost model to automate fusion decisions. This framework applies broadly, not just to matmul fusions, but to any kernel with which Top-K might be fused, including memory-bound kernels.

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

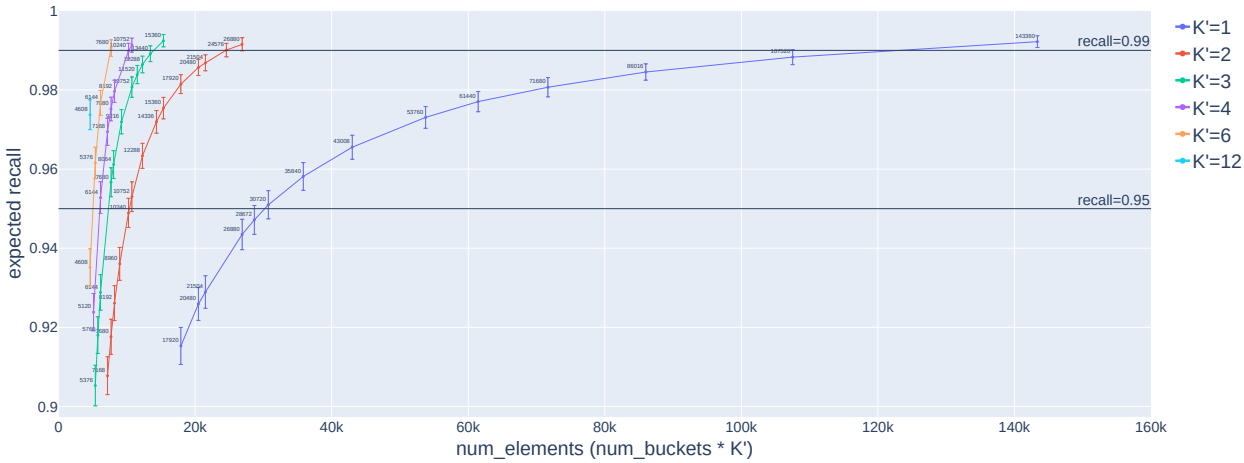

Figure 10: **Recall vs number of elements for finding top-3360 ($\approx$ 0.8%) elements from an array of size 430,080.** The data was obtained from simulated runs of the algorithm on randomly generated integers. The markers represent the sample mean and the error bars represent the sample standard deviation 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.

A.10 Arithmetic intensity of fused matmul operation

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