
ACCELERATING INFERENCE FOR SPARSE EXTREME MULTI-LABEL RANKING TREES

A PREPRINT

Philip A. Etter* **Kai Zhong** **Hsiang-Fu Yu** **Lexing Ying**
Stanford Amazon Amazon Stanford
paetter@stanford.edu kaizhong89@gmail.com rofu.yu@gmail.com lexing@stanford.edu

Inderjit Dhillon
UT Austin & Amazon
inderjit@cs.utexas.edu

ABSTRACT

Tree-based models underpin many modern semantic search engines and recommender systems due to their sub-linear inference times. In industrial applications, these models operate at extreme scales, where every bit of performance is critical. Memory constraints at extreme scales also require that models be sparse, hence tree-based models are often back-ended by sparse matrix algebra routines. However, there are currently no sparse matrix techniques specifically designed for the sparsity structure one encounters in tree-based models for extreme multi-label ranking/classification (XMR/XMC) problems. To address this issue, we present the *masked sparse chunk multiplication* (MSCM) technique, a sparse matrix technique specifically tailored to XMR trees. MSCM is easy to implement, embarrassingly parallelizable, and offers a significant performance boost to any existing tree inference pipeline at no cost. We perform a comprehensive study of MSCM applied to several different sparse inference schemes and benchmark our methods on a general purpose extreme multi-label ranking framework. We observe that MSCM gives consistently dramatic speedups across both the online and batch inference settings, single- and multi-threaded settings, and on many different tree models and datasets. To demonstrate its utility in industrial applications, we apply MSCM to an enterprise-scale semantic product search problem with 100 million products and achieve sub-millisecond latency of 0.88 ms per query on a single thread — an 8x reduction in latency over vanilla inference techniques. The MSCM technique requires absolutely no sacrifices to model accuracy as it gives exactly the same results as standard sparse matrix techniques. Therefore, we believe that MSCM will enable users of XMR trees to save a substantial amount of compute resources in their inference pipelines at very little cost. Our code will be made publicly available at <https://github.com/amzn/pecos>.

1 Introduction

Tree-based models are the workhorse of many modern search engines and recommender systems [9, 18, 12, 10, 15, 17, 5]. Nonetheless, performing inference with these models on the scale demanded by modern applications quickly becomes intractable, requiring an unacceptable amount of compute time and memory to generate useful results. It is no surprise, then, that engineers devote a considerable amount of energy to optimizing their tree-based inference pipelines.

In the interest of pursuing tractable computation for practical applications, we dedicate this paper to examining how to make eXtreme Multi-label Ranking (XMR) and eXtreme Multi-label Classification (XMC)¹ more time and memory efficient. In particular, we propose herein a set of optimizations for sparse XMR tree models that can drastically accelerate inference speed.

*Work done during an internship at Amazon

¹We consider XMC as a subset of XMR. Everything in this paper regarding XMR applies equally well to XMC.

Most XMR tree models use a form of beam search to make inference tractable; the core idea of our optimizations is to take full advantage of the sparsity structure of this beam search to improve performance. This key idea manifests in our principal contribution, *Masked Sparse Chunk Multiplication* (MSCM). MSCM is a new matrix data structure and accompanying multiplication algorithm that uses the aforementioned structured sparsity to drastically reduce unnecessary traversal and cache misses in the underlying memory. We observe that this technique can improve inference time usually anywhere from 2 to 20 times over vanilla inference.

The data structures and algorithms presented herein generalize to many different types of linear XMR tree models. In our performance benchmarks, we present an in-depth exploration of many variations of this technique implemented on top of our generic linear XMR tree implementation.

We now quickly outline this paper: in Section 2, we describe previous related work in the literature. In Section 3, we give an overview of a generic XMR tree model. Section 4 includes all the details of our contribution to the literature. We present our performance benchmarks and evaluation in Section 5. In our benchmarks, we compare the results of our methods to a baseline implementation without MSCM — and demonstrate substantial improvements across a massive range of different settings, i.e., tree topologies, datasets, thread count, etc. We give advice on maximizing performance gain from MSCM in Appendix A.1. To demonstrate the usefulness of our method in an industrial setting, in Section 6 we apply our methods to an enterprise-scale semantic product search problem with 100 million products and use MSCM to achieve a sub-millisecond latency of 0.83 ms per query on a single thread — an 8x improvement over the baseline. Finally, we conclude our study in Section 7.

2 Related Work

XMR tree models are widely used throughout the literature — though a comprehensive terminology surrounding these related techniques has yet to fully emerge. In short, we define an XMR tree model as a tree where every node is associated with a ranker function, every label corresponds to a leaf of the tree, and the ranking for any given label is obtained by combining all ranker functions on the path from the label leaf to the root of the tree.

This definition captures several prior works — for example, the probabilistic label tree (PLT) that originates from [8]. This model in turn takes inspiration from previous work on label trees [3, 2], and there has since been a large body of follow-up work, spawning many state-of-the-art XMR models such as PARABEL [12], BONSAI [10], EXTREME TEXT [15], ATTENTIONXML [17], NAPKINXC [9], and PECOS [18]. Note that EXTREME TEXT and ATTENTIONXML are designed only for dense features. The remaining models support sparse features and thereby require masked sparse matrix times sparse vector multiplication routines (see eq. (6)). We present MSCM as a way to accelerate these routines.

We turn now from the models themselves to existing optimizations for sparse inference. Unlike dense matrix multiplication, the random nature of memory access in sparse matrix algebra can be harder to optimize. Despite that, computational science researchers have devoted considerable energy to optimizing sparse matrix times vector products. Notable sparse matrix techniques relevant to this paper include *cache blocking*, where the internal data of a matrix is reorganized into blocks to encourage memory locality. Many sparse kernel libraries make use of this idea to better tailor sparse matrix calculations to the underlying hardware. Notable examples include the SPARSITY framework [7], a framework that can automate the optimization of sparse kernels — making use of both register and cache level optimizations. Other examples include work by [14] specifically targeting multi-core platforms, and [11], who develop an analytic model to better help predict optimal sparse kernel reorganization. Unfortunately, these techniques all target sparse matrix times dense vector calculations, as these are most common in computational science.

In comparison, sparse matrix times sparse vector (SpMSPV) multiplication is an under-served area. However, there are a number of emerging techniques for sparse matrix operations with underlying graph structure, including [13, 1, 16]. Unfortunately, none of these methods are tailored to XMR tree inference, where beam search induces a very well-structured mask over matrix outputs. They are also significantly more heavy-weight than the method we present. The scale of XMR problems therefore necessitates the development of new masked sparse matrix methods specifically tailored to XMR trees.

3 XMR Tree Models

In this section, we present a generic formulation of the tree model for eXtreme Multi-label Ranking (XMR) problems. To provide the necessary context, we will give a brief overview of the inference algorithm. We omit training details because they are not directly relevant to the techniques in this paper once a model is trained; but we recommend that readers see any of the aforementioned XMR tree model papers e.g. [12, 10, 18] for an overview of training.

3.1 Overview

An XMR problem can be characterized as follows: given a query \mathbf{x} from some embedding \mathbb{R}^d and a set of labels \mathcal{Y} , produce a model that gives an (implicit) ranking of the relevance of the labels in \mathcal{Y} to query \mathbf{x} . In addition, for any query $\mathbf{x} \in \mathcal{X}$, one must be able to efficiently retrieve the top k most relevant labels in \mathcal{Y} according to the model — noting that d is typically very large and \mathbf{x} very sparse.

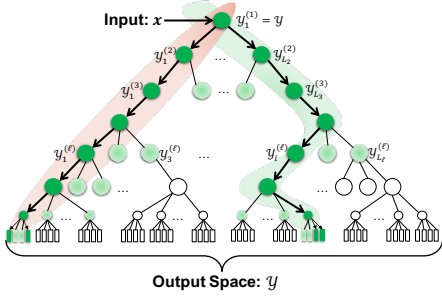


Figure 1: A diagram of the structure of an XMR tree model. [18]

Algorithm 1 Linear XMR Tree Inference

- 1: **Input:** Query matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$.
- 2: **Output:** Beamed predictions $\tilde{\mathbf{P}} \in \mathbb{R}^{n \times L}$.
- 3: Initialize 1st layer predictions to 1: $\tilde{\mathbf{P}}^{(1)} = \mathbf{1} \in \mathbb{R}^{n \times 1}$.
- 4: **for** $l \in \{2, 3, \dots, \text{depth}\}$ **do**
- 5: Copy previous layer predictions: $\tilde{\mathbf{P}}^{(l)} \leftarrow \tilde{\mathbf{P}}^{(l-1)} \mathbf{C}^{(l-1)T}$.
- 6: Get computation mask: $\mathcal{M}^{(l)} \leftarrow \text{bool}(\tilde{\mathbf{P}}^{(l)} \neq 0)$.
- 7: Conditional prediction step: $\hat{\mathbf{P}}^{(l)} \leftarrow \sigma(\mathcal{M}^{(l)} \odot (\mathbf{X} \mathbf{W}^{(l)}))$.
- 8: Combine with prev. layers: $\tilde{\mathbf{P}}^{(l)} \leftarrow \hat{\mathbf{P}}^{(l)} \odot \tilde{\mathbf{P}}^{(l-1)}$.
- 9: Beam search by selecting top b entries of each row: $\tilde{\mathbf{P}}^{(l)} \leftarrow \text{SelectTop}_b(\tilde{\mathbf{P}}^{(l)})$.
- 10: **end for**
- 11: **return** $\tilde{\mathbf{P}} = \tilde{\mathbf{P}}^{(\text{depth})}$

A linear XMR tree model is a hierarchical linear model that constructs a hierarchical clustering of the labels \mathcal{Y} , forming a tree structure. These clusters are denoted $\mathcal{Y}_i^{(l)}$, where l denotes the depth (i.e., layer) of $\mathcal{Y}_i^{(l)}$ in the model tree and i denotes the index of $\mathcal{Y}_i^{(l)}$ in that layer, see Figure 1. The leaves of the tree are the individual labels of \mathcal{Y} .

Every layer of the model has a ranker model that scores the relevance of a cluster $\mathcal{Y}_i^{(l)}$ to a query $\mathbf{x} \in \mathcal{X}$. This ranker model may take on different forms, but for this paper we assume that the model is logistic-like. This means that, at the second layer, for example, the relevance of a cluster $\mathcal{Y}_i^{(2)}$ is given by

$$f(\mathbf{x}, \mathcal{Y}_i^{(2)}) = \sigma(\mathbf{w}_i^{(2)} \cdot \mathbf{x}), \quad (1)$$

where σ denotes an activation function (e.g., sigmoid) and $\mathbf{w}_i^{(2)} \in \mathbb{R}^d$ denotes a very sparse vector of weight parameters.

At subsequent layers, rankers are composed with those of previous layers, mimicking the notion of conditional probability; hence the score of a cluster $\tilde{\mathcal{Y}} \subset \mathcal{Y}$ is defined by the model as

$$f(\mathbf{x}, \tilde{\mathcal{Y}}) = \prod_{\mathbf{w} \in \mathcal{A}(\tilde{\mathcal{Y}})} \sigma(\mathbf{w} \cdot \mathbf{x}), \quad (2)$$

where $\mathcal{A}(\tilde{\mathcal{Y}}) = \{\mathbf{w}_i^{(l)} \mid \tilde{\mathcal{Y}} \subset \mathcal{Y}_i^{(l)}, l \neq 1\}$ denotes all tree nodes on the path from $\tilde{\mathcal{Y}}$ to the root \mathcal{Y} (including $\tilde{\mathcal{Y}}$ and excluding \mathcal{Y}). Naturally, this definition extends all the way to the individual labels $\ell \in \mathcal{Y}$ at the bottom of the tree. We assume here for simplicity that the leaves of the tree all occur on the same layer, but the techniques described in this paper can be extended to the general case.

As a practical aside, the column weight vectors $\mathbf{w}_i^{(l)}$ for each layer l are stored in a $d \times L_l$ weight matrix

$$\mathbf{W}^{(l)} = \begin{bmatrix} \mathbf{w}_1^{(l)} & \mathbf{w}_2^{(l)} & \dots & \mathbf{w}_{L_l}^{(l)} \end{bmatrix}, \quad (3)$$

where L_l denotes the number of clusters in layer l . The tree topology at layer l is usually represented using a cluster indicator matrix $\mathbf{C}^{(l)}$. $\mathbf{C}^{(l)} \in \{0, 1\}^{L_{l+1} \times L_l}$ is defined as

$$\mathbf{C}_{ij}^{(l)} = \text{bool}(\mathcal{Y}_i^{(l+1)} \subset \mathcal{Y}_j^{(l)}), \quad (4)$$

i.e., it is one when $\mathcal{Y}_i^{(l+1)}$ is a child of $\mathcal{Y}_j^{(l)}$ in the tree. Here, $\text{bool}(\cdot)$ is 1 when the condition \cdot is true and 0 otherwise.

3.2 Inference

Throughout this paper, we will assume two inference settings:

1. *Batch Inference*: inference is performed for a batch of n queries represented by a sparse matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ where every row of \mathbf{X} is an individual query \mathbf{x}_i .
2. *Online Inference*: a subset of the batch setting where there is only one query, e.g., the matrix \mathbf{X} has only one row.

When performing inference, the XMR model f prescribes a score to all query-cluster pairs $(\mathbf{x}_i, \mathcal{Y}_j^{(l)})$. Hence, in the batch setting, one can define the prediction matrices,

$$\mathbf{P}_{ij}^{(l)} = f(\mathbf{x}_i, \mathcal{Y}_j^{(l)}) = \prod_{\mathbf{w} \in \mathcal{A}(\mathcal{Y}_j^{(l)})} \sigma(\mathbf{w} \cdot \mathbf{x}_i). \quad (5)$$

The act of batch inference entails collecting the top k most relevant labels (leaves) for each query \mathbf{x}_i and returning their respective prediction scores $\mathbf{P}_{ij}^{(l)}$.

However, the act of exact inference is typically intractable, as it requires searching the entire model tree. To sidestep this issue, models usually use a greedy beam search of the tree as an approximation. For a query \mathbf{x} , this approach discards any clusters on a given level that do not fall into the top $b \geq k$ most relevant clusters examined at that level. Hence, instead of $\mathbf{P}^{(l)}$, we compute *beamed* prediction matrices $\hat{\mathbf{P}}^{(l)}$, where each row has only b nonzero entries whose values are equal to their respective counterparts in $\mathbf{P}^{(l)}$. Pseudo-code for the inference algorithm is given for reference in Algorithm 1, where \odot denotes entry-wise multiplication.

4 Our Method

Our contribution is a method of evaluating masked sparse matrix multiplication that leverages the unique sparsity structure of the beam search to reduce unnecessary traversal, optimize memory locality, and minimize cache misses. The core prediction step of linear XMR tree models is the evaluation of a masked matrix product, i.e.,

$$\mathbf{A}^{(l)} = \mathcal{M}^{(l)} \odot (\mathbf{X}\mathbf{W}^{(l)}), \quad (6)$$

where $\mathbf{A}^{(l)} \in \mathbb{R}^{n \times L_l}$ denotes ranker activations at layer l , $\mathcal{M}^{(l)} \in \{0, 1\}^{n \times L_l}$ denotes a dynamic mask matrix determined by beam search, $\mathbf{X} \in \mathbb{R}^{n \times d}$ is a sparse matrix whose rows correspond to queries in the embedding space, $\mathbf{W}^{(l)} \in \mathbb{R}^{d \times L_l}$ is the sparse weight matrix of our tree model at layer l , and \odot denotes entry-wise multiplication. Note the mask matrix $\mathcal{M}^{(l)}$ is only known at the time of reaching the l -th layer. We leave out the application of σ because it can be applied as a post processing step. We have observed that this masked matrix multiplication takes up the vast majority of inference time on various data sets — between 90% and 98% depending on model sparsity — and hence is ripe for optimization.

For readability, we will suppress the (l) superscript for the rest of the section. Recall that both \mathbf{W} and \mathbf{X} are highly sparse. One typically implements the above computation by storing the weight matrix \mathbf{W} in compressed sparse column (CSC) format and the query matrix \mathbf{X} in compressed sparse row (CSR) format², facilitating efficient access to columns of \mathbf{W} and rows of \mathbf{X} , respectively. Then, to perform the operation in Equation (6), one iterates through all the nonzero entries $\mathcal{M}_{ij} \neq 0$, extracts the i -th query \mathbf{x}_i and the ranker weights \mathbf{w}_j for cluster j , and computes the inner product $\mathbf{A}_{ij} = \mathbf{x}_i \cdot \mathbf{w}_j$ via a sparse vector dot product routine. While one may use different dot product routines depending on the relative sparsities of \mathbf{x}_i and \mathbf{w}_j , an obvious method is to use a progressive binary search in both \mathbf{x}_i and \mathbf{w}_j to find all simultaneous nonzeros. We give pseudo-code for this baseline in Algorithm 4.

While for general sparse matrix multiplication, this is a reasonable method of computation, there are a few special properties about the matrices \mathcal{M} and \mathbf{W} that this method ignores, namely:

1. The sparsity pattern of the mask matrix \mathcal{M} is determined by prolongating the search beam from layer $l - 1$ to layer l . That is, if cluster $\mathcal{Y}_j^{(l)}$ in layer l is a child of a cluster in the search beam of query i , then $\mathcal{M}_{ij} \neq 0$. This means that, if the nodes on a layer are grouped by their respective parents, the nonzero entries of \mathcal{M} come in contiguous single row blocks corresponding to siblings in the model tree. See Figure 2, right, for an illustration. **This suggests that one may improve performance by evaluating all siblings simultaneously.**
2. Columns in \mathbf{W} corresponding to siblings in the model tree often tend to have *similar sparsity patterns*. We give an illustration of this fact in Figure 2, left. Together with the previous observation, **this suggests storing the matrix \mathbf{W} in such a way that entries of \mathbf{W} are contiguous in memory if they both lie on the same row and belong to siblings in the model tree.**

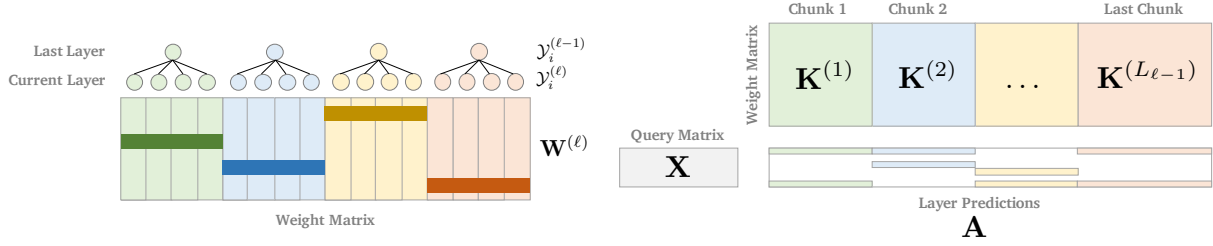


Figure 2: **Left:** A simplified pictorial representation of the property discussed in Item 2. The darkened region indicate non-zeros and the light regions indicate zeros. **Right:** A visual representation of the chunked matrix multiplication routine. The colored blocks of \mathbf{A} represent possible nonzeros. For each colored block in the output, MSCM evaluates the entire block simultaneously. Blocks are evaluated in order of their color.

These considerations lead us naturally to the **column chunked matrix** data structure for the weight matrix \mathbf{W} . In this data structure, we store the matrix $\mathbf{W} \in \mathbb{R}^{d \times L_l}$ as a horizontal array of *matrix chunks* $\mathbf{K}^{(i)}$,

$$\mathbf{W}^{(l)} = [\mathbf{K}^{(1)} \quad \mathbf{K}^{(2)} \quad \dots \quad \mathbf{K}^{(L_{l-1})}], \quad (7)$$

where each chunk $\mathbf{K}^{(i)} \in \mathbb{R}^{d \times B}$ (B is the branching factor, i.e. number of children of the parent node) and is stored as a vertical sparse array of some sparse horizontal vectors $\mathbf{v}^{(j,i)}$,

$$\mathbf{K}^{(i)} = [\mathbf{0} \quad \dots \quad (\mathbf{v}^{(r_1,i)})^T \quad \dots \quad (\mathbf{v}^{(r_{s_i,i})})^T \quad \dots \quad \mathbf{0}]^T. \quad (8)$$

We identify each chunk $\mathbf{K}^{(i)}$ with a parent node in layer $l-1$ of the model tree, and the columns of the chunk $\mathbf{K}^{(i)}$ with the set of siblings in layer l of the model tree that share the aforementioned parent node in layer $l-1$. As we will see, this data structure therefore makes use of both ideas Item 1 and Item 2 simultaneously.

To see why this data structure can accelerate the masked matrix multiplication, consider that one can think of the mask matrix \mathcal{M} as being composed of blocks,

$$\mathcal{M} = \begin{bmatrix} \mathcal{M}^{(1,1)} & \mathcal{M}^{(1,2)} & \dots & \mathcal{M}^{(1,L_{l-1})} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{M}^{(L_l,1)} & \mathcal{M}^{(L_l,2)} & \dots & \mathcal{M}^{(L_l,L_{l-1})} \end{bmatrix}, \quad (9)$$

where the block column partition is the same as that in Equation (7), and every block has one row and corresponds to a single query. As per the observation in Item 1 above, every block $\mathcal{M}^{(j,i)}$ must either be composed entirely of zeros or entirely of ones.

Therefore, since \mathbf{A} and \mathcal{M} share the same sparsity pattern, the ranker activation matrix \mathbf{A} is also composed of the same block partition as \mathcal{M} ,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}^{(1,1)} & \mathbf{A}^{(1,2)} & \dots & \mathbf{A}^{(1,L_{l-1})} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{(L_l,1)} & \mathbf{A}^{(L_l,2)} & \dots & \mathbf{A}^{(L_l,L_{l-1})} \end{bmatrix}, \quad (10)$$

$$\mathbf{A}^{(j,i)} = \mathcal{M}^{(j,i)} \odot (\mathbf{x}_j \mathbf{K}^{(i)}).$$

Hence, for all mask blocks $\mathcal{M}^{(j,i)}$ that are 1, we have

$$\mathbf{A}^{(j,i)} = \mathbf{x}_j \mathbf{K}^{(i)} = \sum_{k \in S(\mathbf{x}_j) \cap S(\mathbf{K}^{(i)})} x_{jk} \mathbf{v}^{(k,i)}, \quad (11)$$

where $S(\mathbf{x}_j)$ and $S(\mathbf{K}^{(i)})$ denote the indices of the nonzero entries of \mathbf{x}_j and the nonzero rows of $\mathbf{K}^{(i)}$ respectively. The above equation says that for all entries of \mathbf{A} in the same block, the intersection $k \in S(\mathbf{x}_j) \cap S(\mathbf{K}^{(i)})$ only needs to be iterated through *once per chunk*, as opposed to *once per column* as is done in a vanilla implementation. Of course, it is theoretically possible that $S(\mathbf{x}_j) \cap S(\mathbf{K}^{(i)})$ is substantially larger than the intersections $S(\mathbf{x}_j) \cap S(\mathbf{w}_i)$ in our baseline, but this tends not to be the case in practice because of the observation in Item 2 that the columns of $\mathbf{K}^{(i)}$

²For details of the well-known CSR and CSC formats, see [6].

tend to have similar support. Moreover, the actual memory locations of the values actively participating in the product Equation (11) are physically closer in memory than they are when $\mathbf{K}^{(i)}$ is stored in CSC format. This helps contribute to better memory locality.

We take a moment to pause and remark that the only task remaining to fully specify our algorithm is to determine how to efficiently iterate over the nonzero entries x_{jk} and nonzero rows $\mathbf{K}^{(k,i)}$ for $k \in \mathcal{S}(\mathbf{x}_j) \cap \mathcal{S}(\mathbf{K}^{(i)})$. This is essential for computing the vector-chunk product $\mathbf{x}_j \mathbf{K}^{(i)}$ efficiently. There are number of ways to do this, each with potential benefits and drawbacks:

1. **Marching Pointers:** The easiest method is to use a marching pointer scheme to iterate over x_{jk} and $\mathbf{K}^{(k,i)}$ for $k \in \mathcal{S}(\mathbf{x}_j) \cap \mathcal{S}(\mathbf{K}^{(i)})$. In this scheme, we maintain both $\mathcal{S}(\mathbf{x}_j)$ and $\mathcal{S}(\mathbf{K}^{(i)})$ as sorted arrays. To iterate, we maintain an index k_x in $\mathcal{S}(\mathbf{x}_j)$ and an index k_K in $\mathcal{S}(\mathbf{K}^{(i)})$. At any given time, we either have $k_x = k_K$, in which case, we emit x_{jk} and $\mathbf{K}^{(k,i)}$; if $k_x < k_K$, we increment k_x ; and if $k_x > k_K$, we increment k_K .
2. **Binary Search:** Since \mathbf{x} can be highly sparse, the second possibility is to do marching pointers, but instead of incrementing pointers one-by-one to find all intersections, we use binary search to quickly find the next intersection. This mirrors the implementation of our baseline Algorithm 4. We maintain an index k_x in $\mathcal{S}(\mathbf{x}_j)$ and an index k_K in $\mathcal{S}(\mathbf{K}^{(i)})$. At any given time, we either have $k_x = k_K$, in which case, we emit x_{jk} and $\mathbf{K}^{(k,i)}$; if $k_x < k_K$, we use a binary search to find the index k'_K in $\mathcal{S}(\mathbf{K}^{(i)})$ where $\mathcal{S}(\mathbf{x}_j)[k_x]$ would be inserted into $\mathcal{S}(\mathbf{K}^{(i)})$ and set $k_K \leftarrow k'_K$; and we handle $k_x > k_K$ similarly.
3. **Hash-map:** The third possibility is to enable fast random access to the rows of $\mathbf{K}^{(i)}$ via a hash-map. The hash-map maps indices i to nonzero rows of $\mathbf{K}^{(i)}$. One can iterate over x_{jk} for $k \in \mathcal{S}(\mathbf{x}_j)$ and perform a hash-map lookup for each k to retrieve $\mathbf{K}^{(k,i)}$ if nonzero. This method is implemented by NAPKINXC [9] for online inference. However, in NAPKINXC, it is implemented on a per-column basis, which introduces a massive memory overhead. Matrix chunking significantly reduces this memory overhead.
4. **Dense Lookup:** The last possibility is to accelerate the above hash-map access by copying the contents of the hash-map into a dense array of length d . Then, a random access to a row of $\mathbf{K}^{(i)}$ is done by an array lookup. This dense array is recycled across the entire program, so afterwards, the dense array must be cleared. This is the method implemented by PARABEL [12] and BONSAI [10].

Algorithm 2 Sparse Vector Chunk Product

- 1: **Input:** Sparse row vector $\mathbf{x} \in \mathbb{R}^d$ and chunk $\mathbf{K} \in \mathbb{R}^{d \times s}$
 - 2: **Output:** The dense product $\mathbf{x}\mathbf{K} \in \mathbb{R}^s$
 - 3: Initialize dense result vector: $\mathbf{z} \leftarrow \mathbf{0} \in \mathbb{R}^s$
 - 4: *Note:* in the following loop, use one of the iterators as described in items 1 through 4.
 - 5: *Note:* For vector \star , $\mathcal{S}(\star)$ denotes the array of indices of nonzeros in \star
 - 6: *Note:* For chunk \star , $\mathcal{S}(\star)$ denotes the array of indices of nonzero rows in \star
 - 7: **for** scalar x_i , row \mathbf{K}_i , where $i \in \mathcal{S}(\mathbf{x}) \cap \mathcal{S}(\mathbf{K})$ **do**
 - 8: **for** All nonzeros in chunk row: $k \in \mathcal{S}(\mathbf{K}_i)$ **do**
 - 9: $\mathbf{z}_k \leftarrow \mathbf{z}_k + x_i \mathbf{K}_{i,k}$
 - 10: **end for**
 - 11: **end for**
 - 12: **return** \mathbf{z}
-

Algorithm 3 Evaluating Masked Matrix Products

- 1: **Input:** Mask matrix $\mathcal{M} \in \{0, 1\}^{n \times L_l}$ in CSR format, query matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ in CSR format, and weight matrix $\mathbf{W} \in \mathbb{R}^{d \times L_l}$ in chunked format.
 - 2: **Output:** Ranker activation matrix $\mathbf{A} \in \mathbb{R}^{n \times L_l}$ in CSR format.
 - 3: Allocate memory for \mathbf{A} using the sparsity pattern of \mathcal{M}
 - 4: Initialize $\mathbf{A} \leftarrow \mathbf{0}$.
 - 5: Collect nonzero blocks: $A \leftarrow \{(i, j) \mid \mathcal{M}^{(i,j)} \neq \mathbf{0}\}$
 - 6: **if** $n > 1$ **then**
 - 7: Sort nonzero blocks $(i, j) \in A$ by chunk index j
 - 8: **end if**
 - 9: **for** $(i, j) \in A$ **do**
 - 10: Compute $\mathbf{A}^{(i,j)} \leftarrow \mathbf{x}_i \mathbf{K}^{(j)}$ via Algorithm 2
 - 11: **end for**
 - 12: **return** \mathbf{A} .
-

There is one final optimization that we have found particularly helpful in reducing inference time — and that is evaluating the nonzero blocks $\mathbf{A}^{(j,i)}$ in order of column chunk i . Doing this ensures that a single chunk $\mathbf{K}^{(i)}$ ideally only has to enter the cache once for all the nonzero blocks $\mathbf{A}^{(j,i)}$ whose values depend on $\mathbf{K}^{(i)}$.

With these optimizations in place, we have found in our performance benchmarks that the hash-map iteration scheme tends to perform best on small to medium sized batches. On large batch sizes, dense lookup performs the best — this is because we incur the cost of copying the hash-map contents into a dense array only once per chunk if we perform

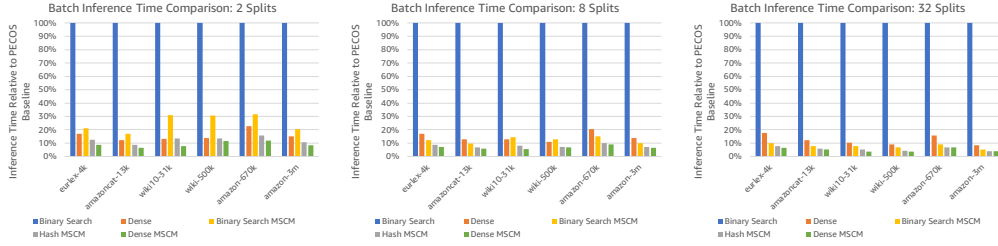


Figure 3: A comparison of batch inference times on different datasets and different model branching factors between the baseline algorithm and our different MSCM implementations as described in item 1 through item 6 of section 5.

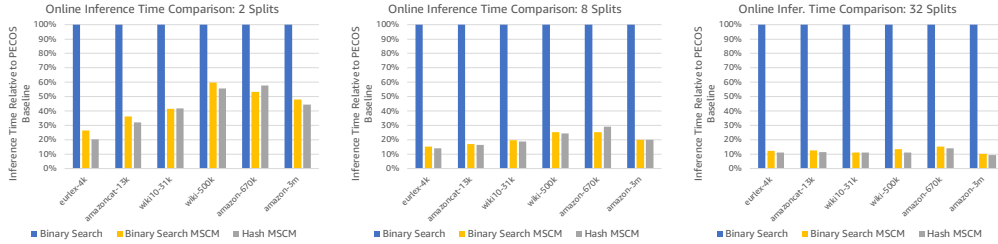


Figure 4: A comparison of online inference times (i.e., batch size one) on different datasets and different model branching factors between the baseline and our different MSCM implementations as described in item 1 through item 6 of section 5. We have omitted methods that were not competitive.

evaluations in chunk order. This cost is then amortized by the increase in the speed of random accesses to the rows of $\mathbf{K}^{(i)}$.

To help readers decide which implementation would suit them best, we present a rule of thumb to deciding which iterator to use in Appendix A.1. A more detailed examination of time complexity and memory overhead is given in Table 2.

With all of this taken into account, we present Algorithm 3, the final version of our algorithm for performing the masked sparse matrix multiplication in Equation (6). As one can see in our benchmarks in Figures 3 and 4, we have found that the optimizations above produce substantial speedups over the baseline implementation in Alg. 4. We will discuss this in Section 5. Generally, the larger the branching factor of the model tree, the more significant the performance boost due to the fact that our method aggregates computation over entire chunks. Moreover, we note that the **performance boost of this technique is essentially free** in that it gives exactly the same inference result as the baseline method, but runs significantly faster.

5 Benchmarks

To evaluate the extent to which the MSCM technique improves performance, we put all of the above variations of MSCM through a series of benchmarks on a set of models that we trained on six publicly available datasets [4]. We ran all benchmarks in this paper on r5 AWS instances. The following subsection discusses single-threaded results, while Appendix A.2 discusses multi-threaded results.

For each dataset, we trained multiple models with varying tree topologies — in our results we present benchmarks with tree branching factors of 2, 8, and 32 to cover a broad swath of performance under different model meta-parameter choices. Furthermore, we ran each of our methods in both the batch setting, where we pass all the queries to the model at the same time as a single matrix, as well as the online setting, where we pass the queries to the model individually one-by-one for evaluation. Both of these settings represent different use cases in industry and we believe it is important for our technique to work well in both settings. For each choice of model branching factor, setting, and dataset, we run the following algorithms:

1. **Binary Search (Baseline):** The baseline Alg. 4 is used to compute inner products between query rows and columns of the unchunked weight matrix.

2. **Dense Lookup** (*Dense*): An unchunked implementation of the dense lookup strategy Item 4 of Section 4 for computing the inner product between query rows and columns of the unchunked weight matrix. This method is an implementation of the algorithm that PARABEL [12] and BONSAI [10] use to perform inference; however, it is implemented in our code and hence, unlike PARABEL, also runs on models with non-binary tree topologies.
3. **Marching Pointers Chunked** (*March MSCM*): A chunked implementation of the Marching Pointers strategy in Item 1 of Section 4 for computing the product of a query row and a column chunk of the weight matrix.
4. **Binary Search Chunked** (*Binary Search MSCM*): A chunked implementation of the binary search strategy in Item 2 of Section 4. This is the chunked analogue of our baseline and provides an example of how reorganization of matrix data can accelerate compute time.
5. **Hash-map Chunked** (*Hash-map MSCM*): A chunked implementation of the hash-map strategy Item 3 of Section 4.
6. **Dense Lookup Chunked** (*Dense MSCM*): A chunked implementation of the dense lookup strategy in Item 4 of Section 4. This is the chunked analogue of the dense lookup algorithm.

For additional context, we also provide statistics about the sizes of each of our datasets in Table 1. The results of our benchmarks can be seen in tabular form in Table 3, Table 4, and Table 5. We also provide a visual graph of the inference times relative to our baseline in Figure 3 and Figure 4.

5.1 Discussion

It is evident from the results in Tables 3, 4, 5 and Figures 3 and 4 that the MSCM technique provides a substantial acceleration to any baseline inference technique. Moreover, in the batch setting with a single thread, we observe that chunked matrices with dense lookup always perform faster than every other technique, regardless of dataset or tree topology, usually giving a 10x to 20x speed boost over the baseline. Moreover, this speed boost grows more substantial as the branching factor of the model grows. This matches our expectations, because larger branching factors allow the chunked methods to shave off more of the unnecessary traversal costs and cache misses of the unchunked methods.

In online mode, there is no longer always a clear winner in the performance benchmarks, but it seems that hash-map chunked matrices usually provide optimal or near-optimal performance among the algorithms that we have tested here, although there is more individual variation among the results than in the batch setting. We note that the dense lookup methods tend to perform worse in this setting because the cost of loading a hash-map into a dense vector is no longer amortized across a large number of queries. Furthermore, we also note that, as one would expect, larger branching factors give a more substantial performance increase. These performance gains persist in multi-threaded environments, see appendix A.2 for details about multi-threaded performance.

From these results, we can conclude that it is always beneficial to use a chunked MSCM matrix format over a column-by-column (e.g., CSC) format for the weight matrix.

6 Enterprise-Scale Performance

To end our discussion of the performance benefits of MSCM, we deploy our MSCM method on an enterprise-scale semantic search model with $L = 100$ million labels (products) and a feature dimension of $d = 4$ million. We see that with beam size 10, both binary search and hash-map MSCM deliver sub-millisecond latency per query on this exceedingly large search problem on 100 million products — a more than $8\times$ reduction in average and P95/P99 inference times compared to vanilla binary search algorithms. In particular, binary search MSCM with beam size 10 has average inference time of 0.884 ms while binary search without MSCM needs 7.282 ms. In terms of P99 latency, binary search MSCM gains more over binary search without MSCM — 1.416 ms vs 12.781 ms. More performance results are provided in Table 6 in the Appendix.

7 Conclusion

In this paper, we have presented the MSCM technique for accelerating inference for sparse XMR tree models. MSCM leverages the structure of the inference sparsity pattern to mitigate unnecessary traversal and improve memory locality. MSCM also comes **free-of-charge** as it exactly preserves inference results, and only requires modest changes to an existing inference pipeline. Moreover, we have examined different variations of the MSCM technique to compile guidelines for extracting the optimal performance. We believe that this technique will allow practitioners to save a substantial amount of compute resources in their inference pipelines, in both online and offline prediction settings. We

have already implemented it into our production pipeline and have been very pleased with the low latency our method enables, as demonstrated in section 6.

For future work, another avenue for optimization comes from the observation that a substantial part of our performance boost comes from sorting vector times chunk products by chunk id — thus better localizing our computation in memory space. It is possible reordering the queries in order to achieve a similar effect. We briefly investigated this, but were unable to obtain a performance boost in our exploration. Further, our exploration of MSCM techniques is not exhaustive, there may be additional ways to iterate through the intersection of query and chunk supports.

As for **limitations** of the work presented herein, we note that our technique explicitly requires that the XMR tree is composed of linear rankers — which means our method is not directly applicable to models that use nonlinear based rankers, such as neural networks. However, since most neural network architectures are repeated compositions of linear transformations and activation functions, our technique may be applicable in sparse settings. Also, we note that our technique as presented here is designed to run on the CPU, and one might gain additional performance by investigating GPU-based implementations.

References

- [1] A. Azad and A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 688–697. IEEE, 2017.
- [2] S. Bengio, J. Weston, and D. Grangier. Label embedding trees for large multi-class tasks. In *Advances in Neural Information Processing Systems*, pages 163–171, 2010.
- [3] A. Beygelzimer, J. Langford, and P. Ravikumar. Error-correcting tournaments. In *International Conference on Algorithmic Learning Theory*, pages 247–262. Springer, 2009.
- [4] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma. The extreme classification repository: Multi-label datasets and code, 2016. URL <http://manikvarma.org/downloads/XC/XMLRepository.html>.
- [5] W.-C. Chang, D. Jiang, H.-F. Yu, C.-H. Teo, J. Zhang, K. Zhong, K. Kolluri, Q. Hu, N. Shandilya, V. Ievgrafov, J. Singh, and I. Dhillon. Extreme multi-label learning for semantic matching in product search. In *Proceedings of the 27th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2021.
- [6] T. A. Davis. *Direct methods for sparse linear systems*. SIAM, 2006.
- [7] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.
- [8] K. Jasinska, K. Dembczynski, R. Busa-Fekete, K. Pfannschmidt, T. Klerx, and E. Hullermeier. Extreme F-measure maximization using sparse probability estimates. In *International Conference on Machine Learning*, pages 1435–1444, 2016.
- [9] K. Jasinska-Kobus, M. Wydmuch, K. Dembczynski, M. Kuznetsov, and R. Busa-Fekete. Probabilistic label trees for extreme multi-label classification. *arXiv preprint arXiv:2009.11218*, 2020.
- [10] S. Khandagale, H. Xiao, and R. Babbar. Bonsai: diverse and shallow trees for extreme multi-label classification. *Machine Learning*, pages 1–21, 2020.
- [11] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.
- [12] Y. Prabhu, A. Kag, S. Harsola, R. Agrawal, and M. Varma. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In *Proceedings of the 2018 World Wide Web Conference*, pages 993–1002, 2018.
- [13] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*, 2015.
- [14] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

- [15] M. Wydmuch, K. Jasinska, M. Kuznetsov, R. Busa-Fekete, and K. Dembczynski. A no-regret generalization of hierarchical softmax to extreme multi-label classification. In *Advances in Neural Information Processing Systems*, pages 6355–6366, 2018.
- [16] C. Yang, Y. Wang, and J. D. Owens. Fast sparse matrix and sparse vector multiplication algorithm on the gpu. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 841–847. IEEE, 2015.
- [17] R. You, Z. Zhang, Z. Wang, S. Dai, H. Mamitsuka, and S. Zhu. AttentionXML: Label tree-based attention-aware deep model for high-performance extreme multi-label text classification. In *Advances in Neural Information Processing Systems*, pages 5820–5830, 2019.
- [18] H.-F. Yu, K. Zhong, and I. S. Dhillon. PECOS: Prediction for enormous and correlated output spaces. *arXiv preprint*, 2020.

A Appendix

A.1 Selecting an Iteration Method

Using the results of the performance benchmark, as well as the contents of Table 2, we briefly provide a guide to choosing a version of the MSCM technique that performs well in a given setting. We always recommend using MSCM, and we suggest that the user consider their choice of iteration scheme in the following order:

1. **Dense Lookup MSCM:** Use this when batch sizes are sufficiently large and storing a dense vector of feature dimension d is not an issue. Conversely, when the batch size is small (i.e., online settings), we don’t recommend using dense lookup.
2. **Hash-map MSCM:** We recommend using this technique when the queries are significantly sparser than the MSCM chunks. However, this technique typically requires some memory overhead (in our implementation around 40% additional memory), because a hash-map of nonzero rows must be stored for every chunk.
3. **Binary Search MSCM:** We recommend using this if the extra memory requirements of hash-map MSCM are too demanding, as this seems to be a good alternative to hash-maps with only a slight loss in performance, according to our benchmarks. Also, we recommend using this if the weight matrices are significantly sparser than the queries.
4. **Marching Pointers MSCM:** We have not found any situations in our data where this outperforms the above two. However, there may be hypothetical cases where marching pointers can perform better than hash-maps or binary search. For example, if queries and chunks are equally sparse, then marching pointers could have the best complexity.

A.2 Multi-Threaded MSCM

One of the benefits of binary search and hash-map MSCM techniques is that batch processing is embarrassingly parallelizable. Indeed, for binary search and hash-map MSCM, one can simply distribute all of the row-chunk operations in line (7) of Algorithm 2 across many different threads with a library like OpenMP, and no substantial additional work or synchronization is required. Dense lookup (both the baseline and MSCM variants) is harder to parallelize because each thread requires its own dense lookup; moreover, we observe that the performance of dense lookup and its MSCM variant are not competitive when parallelized, due to these subtleties that arise in implementation. Since there are no alternative parallel dense lookup tree-based XMR methods available at time of writing (PARABEL and BONSAI do not offer parallelization at this level of granularity), we leave this as possible future work and focus on binary search and hash-map MSCM.

We present the results of parallelizing MSCM in Figure 5, which clearly shows the performance benefits of MSCM clearly extend into the multi-threaded regime. In each of our three tests on WIKI-500K, AMAZON-670K, and AMAZON-3M, both binary search and hash-map MSCM are significantly faster than their non-MSCM counterparts.

| Dataset | eurlex-4k | amazoncat-13k | wiki10-31k | wiki-500k | amazon-670k | amazon-3m |
|-------------------|-----------|---------------|------------|-----------|-------------|-----------|
| Dimension (d) | 5K | 204K | 102K | 2M | 136K | 337K |
| Labels (L) | 4K | 13K | 31K | 501K | 670K | 3M |
| Train Data Size | 16K | 1M | 14K | 2M | 490K | 2M |
| Test Data Size | 4K | 307K | 7K | 784K | 153K | 743K |

Table 1: Size statistics for all datasets in our experiments

| Iteration Method | Per-Query Time Complexity | Extra Memory Overhead |
|-------------------|--|---------------------------|
| Marching Pointers | $O(\text{nnz}_x + \text{nnz}_K)$ | None |
| Binary Search | $O(\min(\text{nnz}_x, \text{nnz}_K) \cdot \log(\max(\text{nnz}_x, \text{nnz}_K)))$ | None |
| Hash-map | $O(h \cdot \text{nnz}_x)$ | $O(c \cdot \text{nnz}_K)$ |
| Dense Lookup | $O(\text{nnz}_x + \text{nnz}_K/n)$ | $O(d)$ |

Table 2: A chart of the per-query time complexity and memory overhead of all the different iteration methods provided in this section. Here h denotes the time it takes to perform a hash lookup, c denotes the number of chunks, $\text{nnz}_x = |\mathcal{S}(\mathbf{x})|$ and $\text{nnz}_K = |\mathcal{S}(\mathbf{K})|$.

| Branching Factor: 2 | eurlex-4k | amazoncat-13k | wiki10-31k | wiki-500k | amazon-670k | amazon-3m |
|--------------------------|-----------|---------------|------------|-----------|-------------|-----------|
| Batch | | | | | | |
| Binary Search (Baseline) | 2.690 ms | 2.880 ms | 7.760 ms | 10.500 ms | 2.320 ms | 3.670 ms |
| Dense Lookup | 0.461 ms | 0.356 ms | 1.030 ms | 1.450 ms | 0.530 ms | 0.557 ms |
| Marching Pointers MSCM | 0.549 ms | 3.170 ms | 2.140 ms | 16.900 ms | 2.760 ms | 8.400 ms |
| Binary Search MSCM | 0.569 ms | 0.485 ms | 2.400 ms | 3.230 ms | 0.365 ms | 0.748 ms |
| Hash-map MSCM | 0.334 ms | 0.254 ms | 1.040 ms | 1.430 ms | 0.365 ms | 0.391 ms |
| Dense Lookup MSCM | 0.231 ms | 0.188 ms | 0.611 ms | 1.230 ms | 0.279 ms | 0.311 ms |
| Online | | | | | | |
| Binary Search (Baseline) | 2.770 ms | 3.300 ms | 8.280 ms | 13.100 ms | 2.820 ms | 4.510 ms |
| Dense Lookup | 1.500 ms | 13.100 ms | 4.450 ms | * | * | * |
| Marching Pointers MSCM | 0.756 ms | 4.380 ms | 3.510 ms | 23.800 ms | 4.190 ms | 10.900 ms |
| Binary Search MSCM | 0.734 ms | 1.190 ms | 3.440 ms | 7.820 ms | 1.500 ms | 2.160 ms |
| Hash-map MSCM | 0.562 ms | 1.060 ms | 3.470 ms | 7.280 ms | 1.630 ms | 2.000 ms |
| Dense Lookup MSCM | 1.070 ms | 6.780 ms | 4.230 ms | * | * | * |

Table 3: Inference time per query in both the batch and online settings. The performance results were run on models trained with a branching factor of 2. (*) denotes that the benchmark did not terminate running within four days and therefore had to be terminated early.

| Branching Factor: 8 | eurlex-4k | amazoncat-13k | wiki10-31k | wiki-500k | amazon-670k | amazon-3m |
|--------------------------|-----------|---------------|------------|-----------|-------------|-----------|
| Batch | | | | | | |
| Binary Search (Baseline) | 2.710 ms | 2.820 ms | 8.200 ms | 12.800 ms | 2.720 ms | 4.280 ms |
| Dense Lookup | 0.463 ms | 0.365 ms | 1.060 ms | 1.410 ms | 0.557 ms | 0.595 ms |
| Marching Pointers MSCM | 0.349 ms | 1.700 ms | 1.290 ms | 10.600 ms | 1.500 ms | 3.880 ms |
| Binary Search MSCM | 0.332 ms | 0.277 ms | 1.180 ms | 1.630 ms | 0.412 ms | 0.423 ms |
| Hash-map MSCM | 0.238 ms | 0.191 ms | 0.673 ms | 0.905 ms | 0.275 ms | 0.303 ms |
| Dense Lookup MSCM | 0.190 ms | 0.166 ms | 0.448 ms | 0.862 ms | 0.247 ms | 0.281 ms |
| Online | | | | | | |
| Binary Search (Baseline) | 2.780 ms | 3.110 ms | 8.400 ms | 14.200 ms | 3.030 ms | 4.950 ms |
| Dense Lookup | 1.480 ms | 12.900 ms | 4.720 ms | * | * | * |
| Marching Pointers MSCM | 0.449 ms | 2.170 ms | 1.980 ms | 13.200 ms | 2.110 ms | 4.630 ms |
| Binary Search MSCM | 0.423 ms | 0.533 ms | 1.650 ms | 3.580 ms | 0.768 ms | 0.988 ms |
| Hash-map MSCM | 0.391 ms | 0.514 ms | 1.570 ms | 3.450 ms | 0.880 ms | 0.992 ms |
| Dense Lookup MSCM | 0.638 ms | 3.270 ms | 2.470 ms | * | * | * |

Table 4: Inference time per query in both the batch and online settings. The performance results were run on models trained with a branching factor of 8. (*) denotes that the benchmark did not terminate running within four days and therefore had to be terminated early.

| Branching Factor: 32 | eurlex-4k | amazoncat-13k | wiki10-31k | wiki-500k | amazon-670k | amazon-3m |
|--------------------------|-----------|---------------|------------|-----------|-------------|-----------|
| Batch | | | | | | |
| Binary Search (Baseline) | 2.610 ms | 3.610 ms | 15.600 ms | 22.800 ms | 4.480 ms | 13.300 ms |
| Dense Lookup | 0.460 ms | 0.437 ms | 1.590 ms | 2.080 ms | 0.703 ms | 1.100 ms |
| Marching Pointers MSCM | 0.278 ms | 1.450 ms | 1.400 ms | 9.370 ms | 1.220 ms | 5.180 ms |
| Binary Search MSCM | 0.261 ms | 0.275 ms | 1.230 ms | 1.530 ms | 0.403 ms | 0.682 ms |
| Hash-map MSCM | 0.202 ms | 0.211 ms | 0.794 ms | 0.955 ms | 0.312 ms | 0.545 ms |
| Dense Lookup MSCM | 0.170 ms | 0.192 ms | 0.578 ms | 0.804 ms | 0.301 ms | 0.524 ms |
| Online | | | | | | |
| Binary Search (Baseline) | 2.650 ms | 3.910 ms | 16.000 ms | 23.600 ms | 4.810 ms | 13.900 ms |
| Dense Lookup | 1.390 ms | 17.100 ms | 7.260 ms | * | * | * |
| Marching Pointers MSCM | 0.353 ms | 1.820 ms | 2.010 ms | 10.900 ms | 1.640 ms | 5.830 ms |
| Binary Search MSCM | 0.330 ms | 0.496 ms | 1.800 ms | 3.220 ms | 0.732 ms | 1.440 ms |
| Hash-map MSCM | 0.292 ms | 0.450 ms | 1.760 ms | 2.610 ms | 0.681 ms | 1.320 ms |
| Dense Lookup MSCM | 0.499 ms | 2.690 ms | 2.650 ms | * | * | * |

Table 5: Inference time per query in both the batch and online settings. The performance results were run on models trained with a branching factor of 32. (*) denotes that the benchmark did not terminate running within four days and therefore had to be terminated early.

| Iteration Method | Average Time (ms/query) | P95 Time (ms/query) | P99 Time (ms/query) |
|----------------------|-------------------------|---------------------|---------------------|
| <i>Beam Size: 10</i> | | | |
| Binary Search MSCM | 0.884 | 1.201 | 1.416 |
| Hash-map MSCM | 0.956 | 1.377 | 1.616 |
| Binary Search | 7.282 | 10.555 | 12.781 |
| <i>Beam Size: 20</i> | | | |
| Binary Search MSCM | 1.629 | 2.225 | 2.630 |
| Hash-map MSCM | 1.799 | 2.603 | 3.058 |
| Binary Search | 14.323 | 20.675 | 24.812 |

Table 6: Performance results on an enterprise-scale semantic search model with 100 million labels in batch mode using a single thread on an AWS X1 instance. The model’s branching factor is 32. Dense lookup is not compared due to out of memory issue.

Algorithm 4 Sparse Vector Inner Product (Baseline)

```

1: Input: Sparse vectors  $\mathbf{x} \in \mathbb{R}^d$  and  $\mathbf{y} \in \mathbb{R}^d$ 
2: Output: The value of  $\mathbf{x} \cdot \mathbf{y}$ 
3: Initialize result variable:  $z \leftarrow 0$ .
4: Initialize nonzero entry indices:  $i_x \leftarrow 0$  and  $i_y \leftarrow 0$ 
5: Note: For vector  $\star$ ,  $\mathcal{S}(\star)$  denotes the array of indices of nonzeros in  $\star$ 
6: Note: The LowerBound( $L, i$ ) function finds the index of the first element in list  $L$  that is not less than  $i$ .
7: repeat
8:   Get index of  $i_x$ -th nonzero in  $\mathbf{x}$ :  $j_x \leftarrow \mathcal{S}(\mathbf{x})[i_x]$ 
9:   Get index of  $i_y$ -th nonzero in  $\mathbf{y}$ :  $j_y \leftarrow \mathcal{S}(\mathbf{y})[i_y]$ 
10:  if Collision check:  $j_x = j_y$  then
11:     $z \leftarrow z + x_{j_x} y_{j_y}$ 
12:    Increment  $i_x$  and  $i_y$ .
13:  else if  $j_x < j_y$  then
14:    Advance  $i_x$ :  $i_x \leftarrow \text{LowerBound}(\mathcal{S}(\mathbf{x}), j_y)$ 
15:  else if  $j_y < j_x$  then
16:    Advance  $i_y$ :  $i_y \leftarrow \text{LowerBound}(\mathcal{S}(\mathbf{y}), j_x)$ 
17:  end if
18: until  $i_x = \text{Length}(\mathcal{S}(\mathbf{x}))$  and  $i_y = \text{Length}(\mathcal{S}(\mathbf{y}))$ 
19: return  $z$ 

```

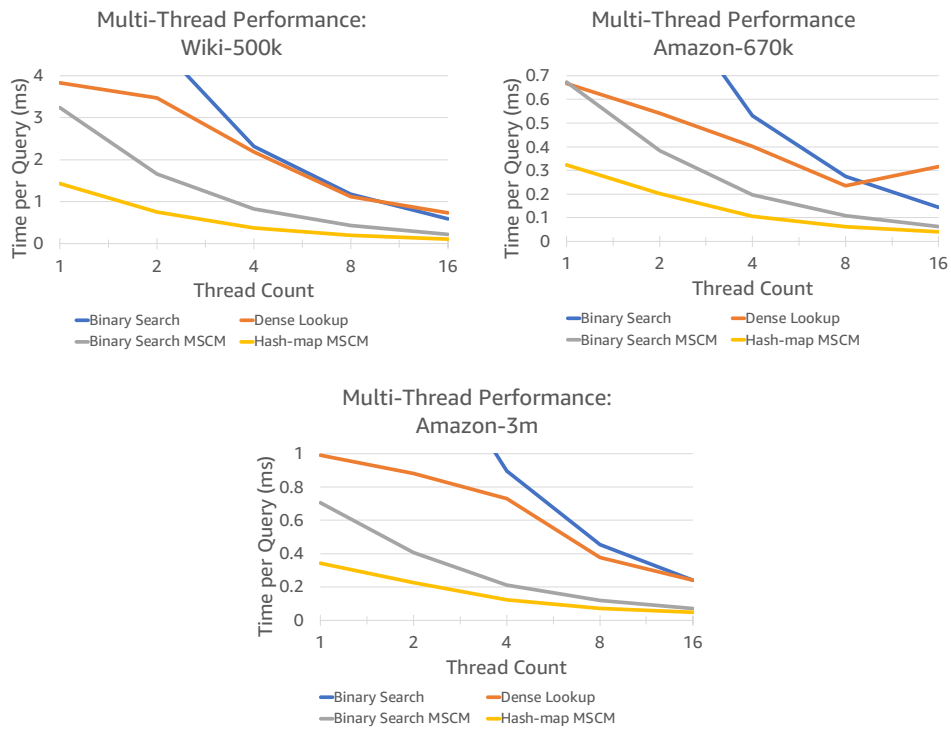


Figure 5: A figure demonstrating the easy parallelization of MSCM. Each graph compares the performance of binary search, dense lookup, binary search MSCM, and hash-map MSCM for several different thread counts. Branching factor here is set as 2.