Copyright
by
Brady Beida Zhou
2018

The Report committee for Brady Beida Zhou
certifies that this is the approved version of the following report:

GPU Accelerated K-Nearest Neighbor Kernel for Sparse Feature Datasets

APPROVED BY

SUPERVISING COMMITTEE:

George Biros, Supervisor

Philipp Krähenbühl

GPU Accelerated K-Nearest Neighbor Kernel for Sparse Feature Datasets

by

Brady Beida Zhou

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE in Computational Science, Engineering, and Math

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

Dedicated to my parents.

Acknowledgments

The research I've worked on in my time here at the University of Texas could not have been possible without the help and support from the following people.

- Prior to this work, my knowledge on efficient computing algorithms was close to none. Chenhan Yu provided mentorship throughout this entire project and was essential to the completion of this work.
- Earlier in my undergraduate studies, Etienne Vouga helped me develop my interests in computer graphics. He inspired me to pursue graduate education and never fails to have something interesting to think about.
- Haley Owen for helping me keep my sanity in a haze of coffee and problem sets in my first four years at UT.

GPU Accelerated K-Nearest Neighbor Kernel for Sparse Feature Datasets

Brady Beida Zhou, M.S.C.S.E.M.
The University of Texas at Austin, 2018

Supervisor: George Biros

In this report, we present an efficient library for computing k-nearest neighbors (kNN) on datasets with sparse features (that is, most of the features per database entry are zero). Our work uses advances in parallel computing and optimized GPU routines. This GPU implementation utilizes highly parallel routines that exploit the sparsity property and in cases of extreme sparsity, we are able to achieve over 100x speedup in time compared to other state-of-the-art approaches designed for more general (dense) features.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **ix**

List of Figures **x**

Chapter 1. Introduction **1**

1.1 Problem Statement 1

1.2 Applications 2

Chapter 2. Background **3**

2.1 Prerequisites 3

2.1.1 Sparse Matrix 3

2.1.2 Dense Matrix Storage 4

2.1.3 Sparse Matrix Storage 4

2.2 Related Work 5

Chapter 3. Method **6**

3.1 Distance Computation 6

3.1.1 Vector Norms 7

3.1.2 Inner Product 7

3.2 Sort and K-Select 7

3.2.1 Sort 8

3.2.2 K-Select 9

Chapter 4. Evaluation **10**
4.1 Experimental Setup 10
4.2 Results 10
4.2.1 Synthetic Dataset 10
4.2.2 Real Dataset 12

Chapter 5. Conclusion **14**

Bibliography **15**

List of Tables

3.1 Comparison of commonly used sorting algorithms. 8

4.1 Evaluation on the Netflix movie rating dataset. 13

List of Figures

3.1 Illustration of the sorting network [1]. 9

4.1 Benchmarks on very sparse data. 11
4.2 Differences in performance with respect to s. 11
4.3 K-Select when varying k. 12

Chapter 1

Introduction

In this work we provide a library for the k-nearest neighbors kernel tailored towards sparse features. We use the CUSPARSE for the sparse matrix-matrix multiplication in the distance computation, and implement the rest of the kernel subroutines from scratch in CUDA. All code is available open-source at `github.com/bradyz/sparse_knn_kernel`. Our contributions can be summarized by the following

- A software package for computing k-nearest neighbors.
- Analysis and benchmarks for running kNN on real and synthetic datasets.

1.1 Problem Statement

Let Q be the set of query points in $\mathbb{R}^{d \times m}$, and R be the set of reference points in $\mathbb{R}^{d \times m}$. To be explicit, m, n denote the number of query and reference points, respectively, and d denotes the number of dimensions that each feature has. For each query point, we need to find the k nearest points in the set of references points, and return them in sorted order. Both Q and R refer to sets of features, where each column is a feature. Depending on the dataset, these

vectors can be dense, or sparse. We denote the average sparsity, or number of non-zero elements divided by the total number of elements, by s.

1.2 Applications

The k-nearest neighbors algorithm has seen uses throughout inference problems of all kinds. In this section, we will give some example use cases of this work in machine learning applications.

Consider a database of users on Netflix with a user being represented as a characteristic vector (0-1 vector) of the shows or movies they have watched. Clearly, this set of features would be extremely high-dimensional, as the total number of shows and movies is easily in the thousands, but the number actually viewed per person would be a very small fraction of the total shows available. An example inference problem would be to find users with similar interests as a query user to provide suggestions. In this situation, $m = 1$, n is the size of a subset of all active users (on the order of thousands), and d is the total number of movies and shows available (on the order of thousands). We would compute the Euclidean distance from the query user's vector to each of the reference points, and sort them. This would be very time-expensive using existing methods, but we can exploit the sparsity property and the effectiveness of GPU computation simultaneously.

Chapter 2

Background

In this section, we provide the reader with introductory knowledge about matrix types, storage, and algorithms. Additionally, we review some past works on the kNN kernel and some of the advantages and disadvantages of each approach.

2.1 Prerequisites

We go over the different types of matrices, dense vs. sparse, and the differences of how they are stored as well as the implementations of basic linear algebra operations - particularly matrix-matrix multiplication.

2.1.1 Sparse Matrix

The "sparsity" of a matrix A refers to the number of non-zero (denoted by $\text{nnz}(A)$) elements in the matrix in comparison to the size. Consider a matrix $A \in \mathbb{R}^{n \times n}$. In general, we say the matrix A is sparse, if the number of non-zero elements of A is small compared to the total size. A simple rule of thumb is that the $\text{nnz}(A)$ is $O(n)$, that is, the number of non-zero elements grows *linearly* with respect to the height and width of the matrix.

Sparse matrices have applications and show up naturally in a variety of fields in computer science. For instance, in computer graphics, it is common to compute the Laplacian ($L = D - A$) of a given triangle mesh, where D is the degree matrix, and A is the adjacency graph. These triangle meshes are almost always sparse in terms of connectivity, and as a result the Laplacian is also sparse. Another example of sparse matrices is in machine learning, where often times the design matrix, also known as the data matrix, has features that are mostly 0 as presented in the Netflix example in Section 1.2.

2.1.2 Dense Matrix Storage

Let $A \in \mathbb{R}^{m \times n}$ be a dense matrix with m rows and n columns. The most general method of storing this matrix is simply in a contiguous block of memory of size mn, where the matrix is stored in memory either in row or column major order.

2.1.3 Sparse Matrix Storage

Let $A \in \mathbb{R}^{m \times n}$ be a sparse matrix with m rows and n columns with c elements, where c is a linear function of m and n, that is $c = \alpha m + \beta n$, where α, $\beta \in \mathbb{R}^+$ are constants. Consider the case if $m = n$ and A was the identity matrix. There would only be n nonzero elements and storing the matrix naively would be quadratic in memory. This can be infeasible if n is large and is addressed by storing the matrix in sparse matrix formats. There are many sparse matrix formats, but we will cover one of the most popular -

compressed sparse row (CSR).

In the CSR format, the matrix $A \in \mathbb{R}^{m \times n}$ with r nonzero values is represented by three arrays. The actual values are stored row-major order in V of size r. Next, I is an array of size $m+1$ where $I_0 = 0$ and $I_i = I_{i-1} + r_i$, where r_i is the number of nonzeros in row i. Finally, J is an array of size r that holds the column indices of each value. Using this matrix format, we have fast access to each row, which allows for fast matrix-matrix multiplication in $O(r)$ time, compared to $O(mn^2)$ for dense matrices. Storage is also much more efficient, being $O(r)$ compared to $O(mn)$.

2.2 Related Work

In this section we present a few other works on the brute force algorithm for the kNN kernel for high dimensional, dense features. To our knowledge, there are no other methods that exploit the sparsity of the features. C. Yu *et al.*[4], introduces a CPU implementation of the kNN kernel optimized for x86 architectures that merges the distance computation and sort. V. Garcia *et al.*[2] is the closest work to ours, that similarly uses a CUDA implementation of the kNN kernel to use GPU acceleration for the distance computation, followed by either a comb-sort or insertion-sort depending on the magnitude of k. However, their method does not consider the sparsity of the data and always performs a dense matrix-matrix multiplication. In a follow up work, V. Garcia *et al.*[3] use a combination of a CUDA and CUBLAS, a CUDA implementation of the BLAS library to gain a 4x speedup over their work.

Chapter 3

Method

The k-nearest neighbors kernel can be split into two main routines consisting of the distance matrix computation, and the top-k selection. All subroutines can be efficiently parallelized and performed on the GPU, resulting in significant speedups compared to existing CPU implementations. In this section, we describe the implementation in detail, along with some additional prerequisite information.

- Squared Distance Computation
  - Compute norms $||q_i||_2^2$, $||r_j||_2^2$, $\forall q_i \in Q, r_j \in R$.
  - Compute dot products $-q_i^\intercal r_j$, $\forall q_i \in Q, r_j \in R$.
- K-Select
  - Sort the distance matrix using bitonic merge sort.
  - Select the first k elements of each row.

3.1 Distance Computation

We seek to construct a squared distance matrix $D \in \mathbb{R}^{m \times n}$ such that $D_{i,j} = ||q_i - r_j||_2^2$, $\forall q_i \in Q, r_j \in R$, that is, the squared distance between the

ith query point and the jth reference point.

3.1.1 Vector Norms

Expanding $||q_i - r_j||_2^2 = ||q_i||_2^2 - 2q_i^\intercal r_j + ||r_j||_2^2$, we can see that the two terms $||q_i||_2^2$, and $||r_j||_2^2$ can be efficiently computed in $O(d)$ time for each of the m, and n terms.

3.1.2 Inner Product

The next step - the inner product, however, will dominate in terms of computation. By computing $C = -2Q^\intercal R$, each entry of this matrix, $C_{i,j} = -2q_i^\intercal r_j$. We can compute this inner product matrix with a simple matrix-matrix multiplication, and take advantage of general matrix-matrix multiplication (GEMM) routines on the GPU. Moreover, in our applications, we exploit the sparsity of Q and R and in particular, we use a sparse matrix-matrix multiplication using a routine using CUSPARSE.

3.2 Sort and K-Select

After we have populated the distance matrix D, where each row i is associated with single query q_i and the pairwise distances to each point in the reference set, we need to pick the smallest k elements out of each unsorted row. There are many selection algorithms, and we choose to perform the k-select by using a brute force method - doing a parallel bitonic mergesort on each row, then simply selecting the firth k elements out of each row. There are

Sorting Algorithms			
Method	Runtime	Space	Parallelizable
Mergesort	$O(n\ log(n))$	$O(n)$	no
Quicksort	$O(n\ log(n))$	$O(n)$	no
Heapsort	$O(n\ log(k))$	$O(k)$	no
Bitonic Mergesort	$O(n\ \log^2(n))$	$O(n)$	yes

Table 3.1: Comparison of commonly used sorting algorithms.

existing methods that combine this process [4], but the brute force method can be efficiently parallelized using the large number of threads on the GPU.

3.2.1 Sort

In order to fully utilize the efficiency of multiple threads, we use a bitonic mergesort to sort the distance array. This sorting method uses $O(n\ \log^2(n))$ comparison operations with a delay of $O(\log^2(n))$ by using sorting networks and only min/max operations.

First, define a sequence of values to be *binonic* if the sequence consists of two non-overlapping sequences - one that is increasing, and the other decreasing. Formally, a sequence A of length n is bitonic if $a_0 \leq a_1 \leq \ldots \leq a_s \geq a_{s+1} \geq \ldots \geq a_n$. In the bitonic mergesort, a series of sorting networks are created create bitonic sequences of length 4, then merge them to create bitonic sequences of length 8, and on.

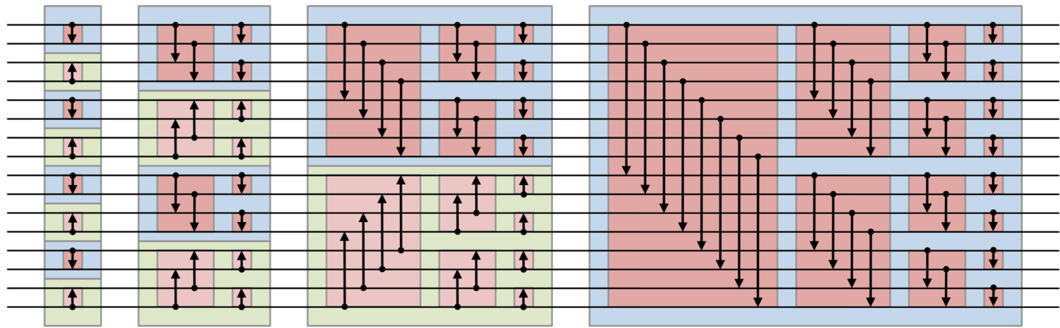

Figure 3.1: Illustration of the sorting network [1].

We can see that there are going to be $\log(n)$ total major steps with $\log(n)$ minor steps, where every step does comparisons on n items, giving us our total runtime of $n\log^2(n)$. On a single thread, this is suboptimal from other sorting algorithms like mergesort and quicksort with their average case runtimes of $n\log(n)$, but in the bitonic mergesort, we can parallelize each of the n comparisons, since they are all independent, to give a runtime of $\log^2(n)$.

3.2.2 K-Select

In the final step of our algorithm, we have a matrix of squared distances D with each row in sorted order and we need to retrieve the minimum k elements from each row. This is trivial and is implemented by extracting the left $m \times k$ submatrix.

Chapter 4

Evaluation

4.1 Experimental Setup

We test the effectiveness of our implementation vs [3] on systems equipped with Intel Core i5 6500 3.20 GHz Quad Core processor, 16 RAM, and a NVIDIA GTX 1070 GPU with 8 GB VRAM. We measure wall-clock time averaged over 10 runs on both synthetic datasets, as well benchmarks on a actual dataset. In both experiments, we vary the sizes of m query points, n reference points, d dimensionality, k neighbors. However, in the synthetic dataset, we are able to vary the average sparsity, s, of each feature which will allow us to better analyze the tradeoff of our method. In the synthetic dataset, we construct the query points Q and reference points R by creating a matrix where each entry is sampled from a unit gaussian if $rand() < s$, else 0.

4.2 Results

4.2.1 Synthetic Dataset

We evaluate our method on the synthetic dataset to have full control over the sparsity properties of the features. This allows for more thorough evaluation and analysis of the benefits and shortcomings of our approach. In

this section, unless otherwise specified, $m = n$.

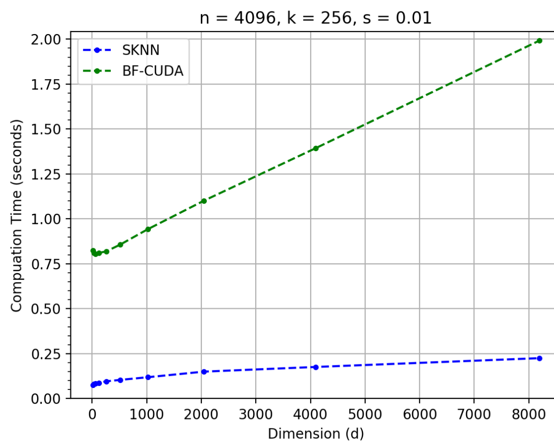

Figure 4.1: Benchmarks on very sparse data.

This case shows the advantage of using our method in sparse cases - achieving a 10x speedup across all tested dimensions ranging from 8 to 8192.

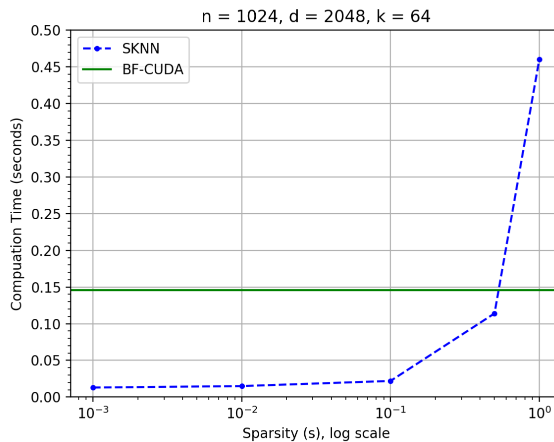

Figure 4.2: Differences in performance with respect to s.

This figure shows the cutoff of when the performance of our method degrades. Interestingly enough, at $s = 0.5$, we can still achieve comparable times as BF-CUDA [3].

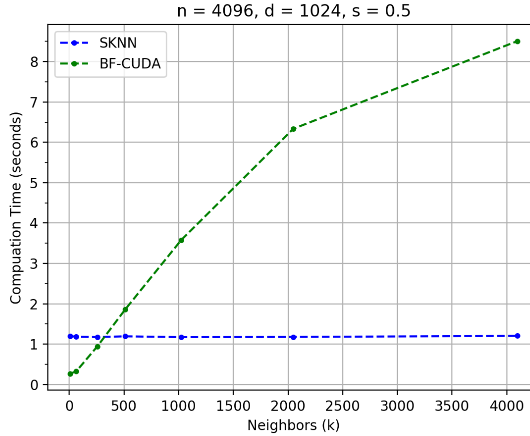

Figure 4.3: K-Select when varying k.

This plot demonstrates the effects of our using an untruncated bitonic mergesort. This algorithm does not consider k and sorts the entire array, so we do not benefit from smaller levels of k like other algorithms such as heapsort and insertion sort.

4.2.2 Real Dataset

Next, we evaluate our method on a real dataset - the Netflix rating dataset presented in a Kaggle competition. In this dataset, each user is associated with a vector, where each entry v_i is the rating for the ith movie. There are 4500 movies in this dataset, so $d = 4500$, and the average number of

n	k	BF-CUDA	SKNN (ours)
1024	64	0.138	**0.021**
1024	1024	0.141	**0.023**
2048	64	0.303	**0.068**
2048	2048	0.305	**0.075**

Table 4.1: Evaluation on the Netflix movie rating dataset.

movies rated per user is 2.25, which give us an average sparsity of $s = 0.0005$. The following figure gives times measured in seconds for varying parameters of kNN search ($m = n$).

Chapter 5

Conclusion

We have introduced an implementation of the kNN kernel tailored toward sparse features. We have presented benchmarks on synthetic as well as real dataset to show the performance of our algorithm on a variety of settings. On the appropriate datasets our approach can lead to massive speedups over existing libraries. In future works we can extend our approach to adaptively use different sorting algorithms based on various magnitudes of k (neighbors), and address usage with larger values of n and m (query and reference points), where the distance matrix does not fit in memory.

Bibliography

[1] Bitonic Sorter. Bitonic sorter — Wikipedia, the free encyclopedia, 2018. [Online; accessed April-2018].

[2] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[3] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3757–3760. IEEE, 2010.

[4] Chenhan D Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2015.