Brute-Force k-Nearest Neighbors Search on the GPU

Shengren Li and Nina Amenta

University of California, Davis

Abstract. We present a brute-force approach for finding k-nearest neighbors on the GPU for many queries in parallel. Our program takes advantage of recent advances in fundamental GPU computing primitives. We modify a matrix multiplication subroutine in MAGMA library [6] to calculate the squared Euclidean distances between queries and references. The nearest neighbors selection is accomplished by a truncated merge sort built on top of sorting and merging functions in the Modern GPU library [3]. Compared to state-of-the-art approaches, our program is faster and it handles larger inputs. For instance, we can find 1000 nearest neighbors among 1 million 64-dimensional reference points at a rate of about 435 queries per second.

1 Introduction

Many important operations in data science involve finding nearest neighbors for each element in a query set Q from a fixed set R of high-dimensional reference points. The k-nearest neighbors problem takes sets Q and R as input, and a constant k, and returns the k nearest neighbors (kNNs) in R for every $q \in Q$. In this paper we consider the high-dimensional version of this problem and we give a state-of-the-art implementation of a brute-force GPU algorithm.

High-dimensional data may be structured data with many variables, but it also arises as long feature vectors derived from unstructured data such as text, images, video, time-series or shapes. Finding nearest-neighbors is the first step in using kernel and non-parametric regression to interpolate functions over the data [8, 26]. When learning classifiers, a nearest-neighbor algorithm [17] is often the most accurate predictor in practice, especially in well-designed feature spaces [13, 15]. An important topic of continuing research is using the nearest-neighbor algorithm with a distance function chosen specifically to improve the classification accuracy on a particular reference data set R. One successful group of algorithms in this area [21, 27, 47–49] chooses the distance function locally for each query q, based on a large set of nearest neighbors in R. Our brute-force algorithm would be particularly good in this situation, since it easily handles large values of k as well as large R.

High-dimensional nearest-neighbor search suffers from the "curse of dimensionality" [14]. This makes it impossible to construct index data structures of reasonable size on R that can answer a nearest-neighbor query exactly in time

sub-linear in $n = |R|$, not only in the worst case but also in many reasonable definitions of average case. Sub-linear solutions even to the approximate version of the problem are surprisingly difficult, and only in recent years have algorithms, most based on Locality Sensitive Hashing [19], provided provable worst-case sub-linear query times using polynomial-sized index structures. Thus brute-force approaches remain an important part of the solution space.

The GPU, with its massive SIMD parallelism, is well-suited to brute-force approaches, providing exact worst-case results at the rate of a couple of ms per query for moderately-sized problems (eg. a few million reference points). As GPU speed and memory size continues to increase - AMD recently released a 32GB GPU - the problem sizes appropriate for the GPU increase as well. Large problems will always have to be handled from disk, eg. [28], but even there, hybrid CPU-GPU implementations [32, 46] rely on the GPU to solve large subproblems by brute-force.

The efficiency of brute-force GPU implementations can themselves vary greatly, particularly with respect to the optimization of data movement through the memory hierarchy. Using better libraries for common operations such as sorting and matrix multiplication can easily improve performance by an order of magnitude over naive implementations. Our brute-force implementation makes heavy use of recent highly optimized CUDA libraries.

Any brute-force implementation consists of two steps. First, we compute a matrix $d^2(Q, R)$ giving the squared distance of each $q \in Q$ to each $r \in R$. To implement this step, we modify the inner loop of a well-optimized open source matrix multiplication kernel, the SGEMM kernel in MAGMA library [6]. In the second step, for each query, we search its row of the matrix to find the k smallest squared distances. There is considerable variation in how this step can be carried out in brute-force implementations. Our implementation uses the *merge-path* function from the Modern GPU library [3], which has proved to be very useful in other contexts, to implement a truncated merge sort, in which only the k smallest items move forward from one level of merging to the next.

Together, these two steps form a CUDA program, that, to the best of our knowledge, is currently the fastest kNN implementation on the GPU. Our code scales linearly in $m = |Q|$, $n = |R|$, the dimension d, and k, and unlike other codes, it handles large values of k (up to $k = 3000$). We compare our implementation to the two recent published algorithms for which code is available, `cuknns` [1] and `kNN CUDA` [2], and to an implementation with the segmented sort function in the Modern GPU library.

2 Related work

There are many GPU approaches to brute-force kNN, applying different strategies for the two major components of the algorithm.

Squared distance matrix: The two main existing approaches to computing the squared distance matrix are to implement it directly with a custom kernel [9,

23,24,29,30,33,35–37], or to derive the distances from an already well-optimized matrix multiplication routine [10, 18, 24, 31, 45]. Custom direct implementations are typically optimized by *tiling*, which divides the distance matrix into equal-sized submatrices (or tiles) and then assigns a thread block to each. The tile size is set so that a group of query and reference points can be accommodated in the fast shared memory and reused by threads within the same block.

The matrix multiplication approach to computing the squared distance matrix $d^2(Q, R)$ is based on the equation

$$d^2(Q, R) = N_Q + N_R - 2Q^T R, \tag{1}$$

where the elements of the ith row of N_Q are $\|Q_i\|^2$, and the elements of the jth column of N_R are $\|R_j\|^2$. These can be computed using custom CUDA kernels [18, 24] or Thrust library [7] primitives [31]. A matrix multiplication routine from a highly optimized library, e.g., cuBLAS [4] calculates the more expensive third term, and the speed of the highly optimized library routine compensates for the additional arithmetic operations.

Selecting nearest neighbors: The approaches for selecting nearest neighbors are more diverse. Kuang and Zhao [33] simply sort all the distances to each query using GPU radix sort; this relies on the speed of modern sorting libraries. Dashti et al. [18] use radix sort as well, but on the entire matrix. The candidate distances are first sorted all together and then stably sorted by query index to separate the results for each query. Kato and Hosino [29,30] build a max-heap for each query and parallel threads push new candidates to the heap using atomic operations. Beliakov and Li [12] calculate the kth smallest distance to each query directly using a GPU selection algorithm [11] based on Kelley's cutting plane method, a convex optimization technique.

Many approaches divide the distances to each query into blocks. Liang et al. [35–37] find the local kNN within each block by testing each distance against all the others in parallel; a single thread per query then merges the lists. Arefin et al. [9] maintain an unsorted array of size k for each query and a pointer to the largest element in the array. A single thread maintains this structure at each level with a linear scan.

Several other approaches use a parallel reduction pattern, that is, a hierarchical pattern of comparisons. Barrientos et al. [10] create multiple heaps for each query and then merge the heaps at each level. Miranda et al. [39] choose the kNN at each level using quickselect. Komarov et al. [31] also use quickselect, implemented with the CUDA warp vote function `__ballot()`, bit count function `__popc()` and bit shift operations.

Truncated sort was introduced by Sismanis et al. [45]. Elements are discarded from the sort when it is clear that they cannot belong to the smallest k. They describe several algorithms, and show that their truncated bitonic sort has outstanding performance on the GPU. Garcia et al. [23,24] use a truncated insertion sort.

Besides brute-force approaches, some of the asymptotically more efficient approximate kNN algorithms have been implemented on the GPU. Pan et al. [42–44] and Lukac et al. [38] construct variants of Locality Sensitive Hashing. The running times of these methods are competitive with existing brute-force implementations, but they return approximate results; like the brute-force approach, the main bottleneck is the selection of the kNNs from a large set of candidates from R, so our techniques may be useful in implementing these approaches as well. There are also heuristic techniques that use various kinds of filtering to try and avoid computing the entire squared distance matrix $d^2(Q, R)$ [16, 20, 46].

3 Implementation

Let $m = |Q|$ be the size of the list Q of query points and let $n = |R|$ be the number of reference points. In the input, the query and reference lists are organized as $d \times m$ and $d \times n$ matrices, where d is the dimension. These matrices are stored as row-major 1D arrays, so that, for each dimension i, the ith components of all the points are contiguous; this facilitates coalesced access to global memory.

In the squared distance matrix $d^2(Q, R)$, we represent each of the distances as a 64-bit integer, as follows. The high 32 bits contain the floating point distance between the reference and the query, and the low 32 bits contain the integer index of the reference point. When merging lists of kNNs for a particular query, this composite representation allows us to swap the positions of two candidate distances by swapping two 64-bit integers instead of swapping both the distances and indices.

3.1 Computing the squared distance matrix

We leverage the efficiency of GPU matrix multiplication, which is a very well-studied operation, to compute $d^2(Q, R)$. Listings 1.1 and 1.2 compare the computation of the squared Euclidean distance matrix and matrix multiplication. The only difference between them is in the innermost loop.

Listing 1.1: Squared Euclidean distances

```
for i = 0 to m-1
  for j = 0 to n-1
    distance[i,j] = 0
    for k = 0 to d-1
      diff = Q[k,i] - R[k,j]
      distance[i,j] += diff * diff
```

Listing 1.2: Dot products

```
for i = 0 to m-1
  for j = 0 to n-1
    product[i,j] = 0
    for k = 0 to d-1
      product[i,j] += Q[k,i] * R[k,j]
```

Our computation of $d^2(Q, R)$ is a modification of a very efficient CUDA matrix multiplication kernel [6, 22, 34, 40], replacing the internal loop with the squared Euclidean distance computation, and then combining the resulting squared distance with the index of the reference point $r \in R$ to generate the 64-bit candidate representation described above.

This distance computation inherits a number of optimizations from the matrix multiplication kernel. The most important is tiling. The squared distance

matrix $d^2(Q, R)$ is divided into tiles of size $m_{blk} \times n_{blk}$. The input for computing a tile is a $d \times m_{blk}$ stripe of Q and a $d \times n_{blk}$ stripe of R. Each tile is processed by a block of threads. These data chunks are loaded into shared memory in a coalesced fashion and reused by threads within the same thread block. The tile size is tuned to the Fermi architecture.

Since the introduction of the Fermi architecture, accessing data in registers is much faster than accessing data from shared memory. To take advantage of this, one more level of tiling is employed at the thread level. Each thread computes a $m_{thd} \times n_{thd}$ matrix with stride m_{blk}/m_{thd} and n_{blk}/n_{thd}. For each dimension, n_{thd} values are loaded from shared memory to registers and reused to compute all $m_{thd} \times n_{thd}$ partial results.

The kernel also uses loop unrolling and double buffering [34]. Loop unrolling replaces a loop with a single block of straight-line code. Not only is the cost of looping eliminated, but also more instruction level parallelism can be obtained by the compiler. Double buffering takes advantage of the Fermi GPU's dual-issue architecture. It overlaps the arithmetic operations of the current iteration with the memory operations of the following iteration.

Other brute-force kNN search implementations [10, 18, 24, 31, 45] take advantage of fast GPU matrix multiplication, but they use it as a subroutine as described in Section 2. Clearly, our approach saves both memory and computation time. The only drawback is that we can only use it with open source matrix multiplication codes. Fortunately, MAGMA [6] is competitive with proprietary matrix multiplication kernels (see Section 4).

3.2 Selecting nearest neighbors

Overview: A naive approach to finding the k-nearest neighbors for each query would sort the n candidates by distance and then return the first k. Following Sismanis et al. [45], we use a truncated sorting algorithm instead, which discards candidates as it becomes clear that they cannot belong to the top k.

The truncated merge sort is designed to use the GPU shared memory efficiently. In the first stage, we divide the n candidates of a query into chunks of size at least k that fit into shared memory, and sort each chunk in parallel with a block of threads. In the second stage, we iteratively merge pairs of sorted chunks and discard the larger half of each pair, so that the number of sorted chunks in play decreases by a factor of two at each iteration (notice that this property gives an $O(n)$ running time, if we consider the chunk size to be constant). The second stage stops when only one chunk is left, which contains the k-nearest neighbors. In both the sorting and the merging stages, the operations for different queries are executed in parallel, the sorts and merges on the different chunks of each query are executed in parallel, and the chunk-level sorts and merges are themselves parallel operations.

Merge Path: We use the Merge Path algorithm [3, 25, 41] for both sorting and merging. In this section we briefly describe Merge Path and why it is so

efficient. Let a and b be the two input arrays, sorted from smallest to largest; for simplicity assume all elements are unique. Let $s(i, j)$ denote the set consisting of the first i items from a and the first j items from b: $a[0] \dots a[i-1]$ and $b[0] \dots b[j-1]$. Finally, define the list S_p of possible choices of $s(i, j)$ such that $i + j = p$, ordered by i. For instance, if $a = [2, 5, 11, 13]$ and $b = [3, 8, 12, 17]$, we get $S_3 = [[3, 8, 12], [2, 3, 8], [2, 3, 5], [2, 5, 11]]$. The correct first p elements of the output has to be one of the elements of S_p, ($[2, 3, 5]$ in the example); call this s_p.

Now consider mapping the function $f(i, j) = b[j-1] < a[i]$ over S_p, where we define $f(i, -1) = \text{True}$. In the example, we get $[12 < 2 = \text{False}, 8 < 5 = \text{False}, 3 < 11 = \text{True}, \text{True}]$. And in fact, $f(i, j)$ is always False to the left of s_p and True at and to the right of s_p [41]. So we can find s_p by binary search on the Boolean array $f(S_p)$, computing only the elements of $f(S_p)$ that we need to evaluate. Once we know s_p, we can break the problem of merging a and b into two independent parts, one merging the first p output elements and the other merging the rest.

In fact, we break the problem into several independent parts. Assuming that there are r processors, we evenly divide the output array c into non-overlapping segments of size $l = \frac{|a|+|b|}{r}$. Processor x finds s_{lx} and then generates the output between positions lx and $l(x + 1) - 1$. Each processor works independently of the others, except for the synchronization after the binary search.

Merge Path works well on the GPU because it divides the work into roughly balanced subtasks. Choosing the size of the subtasks is the key tuning parameter; choosing r too large increases the number of subproblems and allows the binary search to dominate, while choosing r too small allows the sequential merges to dominate and fails to create enough work for all the processors.

Using MGPU: Modern GPU (MGPU) [3] is a library of high-performance CUDA primitives, including Merge Path, that takes advantage of parallelism at both the kernel and thread block levels. We demonstrate that the MGPU primitives, particularly Merge Path, leads to a very efficient nearest-neighbors selection algorithm.

In the first (sorting) stage of the selection algorithm, each thread block loads a chunk of nearest neighbor candidates into shared memory and then calls `mgpu::CTAMergesort` to sort them. In `mgpu::CTAMergesort`, each thread first sorts a small number of candidates in registers. Next, the sorted arrays are merged (still in shared memory) using a parallel reduction pattern. Each merge operation is done with Merge Path. In the first step of the reduction, two threads work on merging each pair of arrays. As the array length doubles, so does the number of cooperating threads per array, so that each sequential merge operation ends up handling the same number of items (determined by the parameter r, above).

At each iteration of the second (merging) stage, each thread block loads two of the sorted chunks into shared memory and then uses the Merge Path algorithm.

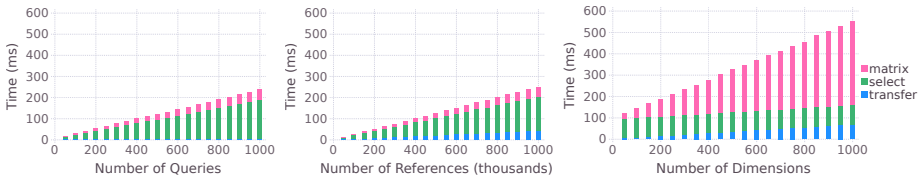

Fig. 1: Total running time, with the proportions of computing squared distance matrix (denoted by *matrix*), selecting nearest neighbors (denoted by *select* and with $k = 1000$) and transferring data (denoted by *transfer*) as we vary the number of queries, references and dimensions, respectively.

Just the smaller half of the output c will be stored back to global memory, so we only need to assign threads to construct the first half of the output array.

The chunk sizes we use depend on the choice of k, when k is large; for $k < 500$, we use the chunk size for $k = 500$ since making it smaller does not improve the running time.

4 Results

Our experimental environment employs CUDA Toolkit 6.5 [5] and a GeForce GTX 460 graphics card, which uses the Fermi architecture and has 1023 MB global memory.

Since our implementation is brute-force, the distribution of input data does not influence the performance of our program, so we use test data composed of uniformly distributed random numbers between -1 and 1.

The size of input is determined by the number of queries (m), references (n) and dimensions (d). We generated three test datasets to demonstrate the influence of each of these factors on the running time:

- $m \in [50, 1000]$, $n = 100000$ and $d = 64$.
- $m = 90$, $n \in [50000, 1000000]$ and $d = 64$.
- $m = 500$, $n = 100000$ and $d \in [50, 1000]$.

Evaluation and analysis: Figure 1 shows the running time of our program and each of its three major components, computing squared distance matrix, selecting nearest neighbors and transferring data between CPU and GPU, on the test data. The running time is indeed linear in each of the factors (m, n, d), although the overall running time is $O(mnd)$. In any fixed dimension, the running time for the nearest neighbors selection step increases more quickly as the input size grows, and eventually dominates the time required for the matrix multiplication. The number of dimensions is irrelevant to the performance of the nearest neighbors selection.

Because the optimal chunk size in the nearest neighbors selection phase is achieved at $k = 500$, choosing k smaller than that does not improve the running time by much. The running time increases linearly with k for $k > 500$, however.

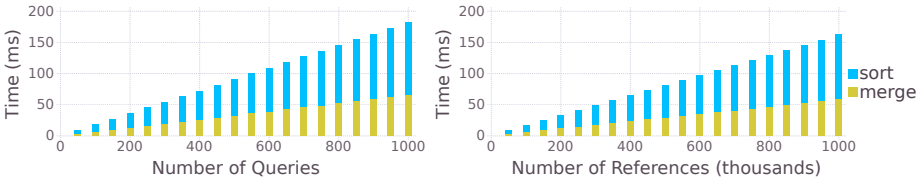

Fig. 2: Total running time of selecting nearest neighbors with $k = 1000$ and the proportions of sorting and merging candidate chunks (denoted by *sort* and *merge*, respectively) as we vary the number of queries and references, respectively.

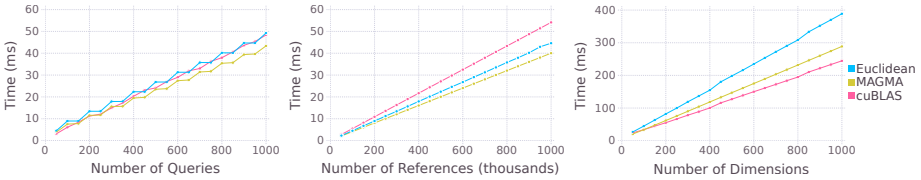

Fig. 3: Running time comparison of our squared distance matrix computation kernel (denoted by *Euclidean*) and the SGEMM subroutine in MAGMA [6] and cuBLAS [4] as we vary the number of queries, references and dimensions, respectively.

Next, we take a closer look at selecting nearest neighbors and its two kernels, sorting and merging candidate chunks, in Figure 2. We observe that the running time of the sorting step increases more quickly with input size.

Comparisons: To evaluate the performance of the kernel that computes our squared distance matrix $d^2(Q, R)$, we compare its performance to the two SGEMM (single precision general matrix-matrix multiply) implementations in MAGMA [6] and cuBLAS [4] (see Figure 3). Recall that both custom squared distance kernels and squared distance computations that use matrix multiplication as a subroutine are less efficient than the heavily optimized matrix multiplication subroutines.

Our implementation is modified from the SGEMM subroutine in MAGMA, but it is only marginally slower. The proprietary SGEMM matrix multiply implementation in cuBLAS performs better than MAGMA as the number of dimensions increases. In principle, any efficient matrix multiplication kernel can be modified to compute squared distances; we could not use cuBLAS only because it is not open source.

Finally, we evaluate the running time of our Merge Path nearest neighbors selection step with that of two recent nearest neighbor algorithms for which code is available. These are truncated insertion sort [2,24] and truncated bitonic sort [1,45]. We also compare against segmented sort applied to all n candidates for each query, as implemented in the Modern GPU library [3]. These compar-

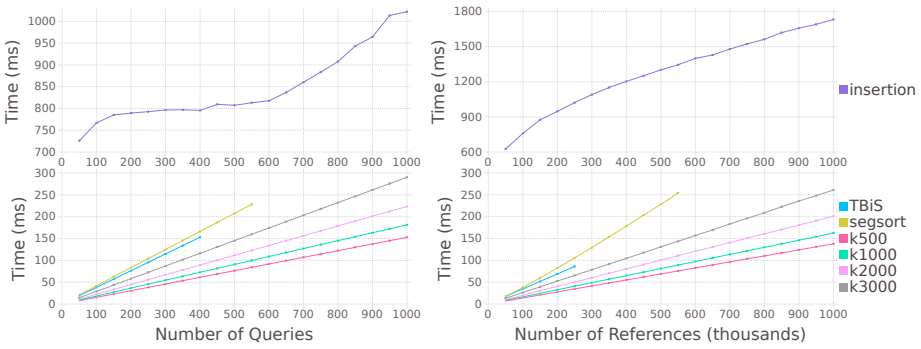

Fig. 4: Running time comparison of our nearest neighbors selection component for different k (denoted by *k500*, *k1000*, *k2000* and *k3000*, respectively), truncated insertion sort [2] (denoted by *insertion* and with $k = 100$), truncated bitonic sort [1] (denoted by *TBiS* and with $k = 500$) and segmented sort in the Modern GPU library [3] (denoted by *segsort* and with $k = n$) as we vary the number of queries and references, respectively.

isons are shown in Figure 4. The running time of the truncated insertion sort is shown in a separate graph because it is significantly slower, even for $k = 100$.

Our kernels are configured to find 500, 1000, 2000 and 3000 nearest neighbors per query, respectively. 3000 is the maximum size of candidate chunk that our kernels can handle (limited by the size of shared memory). In both graphs, the truncated bitonic sort works well up to a certain point, after which it stops producing correct results. Our program is twice as fast at $k = 500$, and only when we reduce k to 16 does TBiS become faster than our program with $k = 500$. Segmented sort [3] is robust at large input sizes, but it is slower and requires much more memory.

5 Conclusions

Finding ways to use highly-optimized GPU library functions is an effective way to achieve both speed and robustness in this important application. Our algorithm advances the state of the art for all but the smallest values of k. It is unique in its ability to handle large values of k, and large input datasets. The performance of our algorithm for very small values of k is limited mainly by the performance of the selection step. Possibly this could be improved by allowing one thread block to perform multiple truncated merge sorts in parallel. The drawback of this approach would be that it complicates the kernel.

Approximate kNN search approaches where nearest-neighbor candidates are filtered so that not all squared distances need to be computed could benefit from using our truncated merge sort to select the true nearest neighbors from the candidates. This is true for Locality Sensitive Hashing as well as for heuristic approaches.

Acknowledgments. We are grateful for NSF grant IIS-0964357, which supported this work.

References

1. cuknns: GPU accelerated k-nearest neighbor library. `http://autogpu.ee.auth.gr/doku.php?id=cuknns:gpu_accelerated_k-nearest_neighbor_library`, 2012.
2. kNN CUDA. `http://vincentfpgarcia.github.io/kNN-CUDA/`, 2013.
3. Modern GPU. `http://nvlabs.github.io/moderngpu/`, 2013.
4. cuBLAS in CUDA toolkit 6.5. `https://developer.nvidia.com/cuBLAS`, 2014.
5. CUDA toolkit 6.5. `https://developer.nvidia.com/cuda-toolkit-65`, 2014.
6. MAGMA 1.6.1. `http://icl.cs.utk.edu/magma/`, 2015.
7. Thrust. `https://developer.nvidia.com/Thrust`, 2015.
8. N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.
9. Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, and Pablo Moscato. GPU-FS-kNN: A software tool for fast and scalable kNN computation using GPUs. *PLOS ONE*, 7(8):e44000, August 2012.
10. Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto Matias, and Mauricio Marin. kNN query processing in metric spaces using GPUs. In *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 380–392. Springer, 2011.
11. G. Beliakov, M. Johnstone, and S. Nahavandi. Computing of high breakdown regression estimators without sorting on graphics processing units. *Computing*, 94(5):433–447, May 2012.
12. Gleb Beliakov and Gang Li. Improving the speed and stability of the k-nearest neighbors method. *Pattern Recognition Letters*, 33(10):1296–1301, 2012.
13. Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, April 2002.
14. Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Database Theory — ICDT'99*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin Heidelberg, 1999.
15. Oren Boiman, Eli Shechtman, and Michal Irani. In defense of nearest-neighbor based image classification. In *IEEE Conference on Computer Vision and Pattern Recognition, 2008. CVPR 2008.*, pages 1–8. IEEE, June 2008.
16. Lawrence Cayton. Accelerating nearest neighbor search on manycore systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 402–413. IEEE, May 2012.
17. T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.
18. Ali 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(9):e74113, September 2013.
19. Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG'04, pages 253–262. ACM, 2004.

20. Patrick Diehl and Marc Alexander Schweitzer. Efficient neighbor search for particle methods on GPUs. In *Meshfree Methods for Partial Differential Equations VII*, volume 100 of *Lecture Notes in Computational Science and Engineering*, pages 81–95. Springer, 2015.
21. Carlotta Domeniconi, Jing Peng, and Dimitrios Gunopulos. Locally adaptive metric nearest-neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(9):1281–1285, September 2002.
22. Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical Computations with GPUs*, chapter 1, pages 3–28. Springer International Publishing, 2014.
23. Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08.*, pages 1–6. IEEE, June 2008.
24. Vincent Garcia, Éric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proceedings of 2010 IEEE 17th International Conference on Image Processing*, pages 3757–3760, September 2010.
25. Oded Green, Robert McColl, and David A. Bader. GPU merge path - a GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS'12, pages 331–340. ACM, 2012.
26. Wolfgang Härdle. *Applied nonparametric regression.* Number 19 in Econometric Society Monographs. Cambridge University Press, 1990.
27. Trevor Hastie and Robert Tibshirani. Discriminant adaptive nearest neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):607–616, June 1996.
28. Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, 2011.
29. Kimikazu Kato and Tikara Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID'10, pages 769–773. IEEE Computer Society, 2010.
30. Kimikazu Kato and Tikara Hosino. Multi-GPU algorithm for k-nearest neighbor problem. *Concurrency and Computation: Practice and Experience*, 24(1):45–53, 2012.
31. Ivan Komarov, Ali Dashti, and Roshan M. D'Souza. Fast k-NNG construction with GPU-based quick multi-select. *PLOS ONE*, 9(5):e92409, May 2014.
32. Martin Kruliš, Tomáš Skopal, Jakub Lokoč, and Christian Beecks. Combining cpu and gpu architectures for fast similarity search. *Distributed and Parallel Databases*, 30(3-4):179–207, 2012.
33. Quansheng Kuang and Lei Zhao. A practical GPU based KNN algorithm. In *Proceedings of the Second Symposium International Computer Science and Computational Technology (ISCSCT'09)*, pages 151–155. Citeseer, December 2009.
34. Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, November 2012.
35. Shenshen Liang, Ying Liu, Cheng Wang, and Liheng Jian. A CUDA-based parallel implementation of k-nearest neighbor algorithm. In *International Confer-*

ence on Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC'09., pages 291–296. IEEE, October 2009.
36. Shenshen Liang, Ying Liu, Cheng Wang, and Liheng Jian. Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. In *2010 IEEE 2nd Symposium on Web Society (SWS)*, pages 53–60. IEEE, August 2010.
37. Shenshen Liang, Cheng Wang, Ying Liu, and Liheng Jian. CUKNN: A parallel implementation of k-nearest neighbor on CUDA-enabled GPU. In *IEEE Youth Conference on Information, Computing and Telecommunication, 2009. YC-ICT'09.*, pages 415–418. IEEE, September 2009.
38. Niko Lukač and Borut Žalik. Fast approximate k-nearest neighbours search using GPGPU. In *GPU Computing and Applications*, chapter 14, pages 221–234. Springer, 2015.
39. Natalia Miranda, Edgar Chávez, María Fabiana Piccoli, and Nora Reyes. (very) fast (all) k-nearest neighbors in metric and non metric spaces without indexing. In *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 300–311. Springer, 2013.
40. Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, November 2010.
41. Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1611–1618. IEEE, May 2012.
42. Jia Pan, Christian Lauterbach, and Dinesh Manocha. Efficient nearest-neighbor computation for GPU-based motion planning. In *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2243–2248. IEEE, October 2010.
43. Jia Pan and Dinesh Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'11, pages 211–220. ACM, November 2011.
44. Jia Pan and Dinesh Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pages 378–389. IEEE, April 2012.
45. Nikos Sismanis, Nikos Pitsianis, and Xiaobai Sun. Parallel search of k-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–6. IEEE, September 2012.
46. George Teodoro, Eduardo Valle, Nathan Mariano, Ricardo Torres, Wagner Meira Jr, and Joel H. Saltz. Approximate similarity search for online multimedia services on distributed CPU—GPU platforms. *The VLDB Journal*, 23(3):427–448, June 2014.
47. Pascal Vincent and Yoshua Bengio. K-local hyperplane and convex distance nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, pages 985–992. MIT Press, 2002.
48. Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 10:207–244, December 2009.
49. Hao Zhang, Alexander C. Berg, Michael Maire, and Jitendra Malik. SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 2126–2136. IEEE, 2006.