OPEN ACCESS Freely available online

PLOS ONE

Efficient Computation of k-Nearest Neighbour Graphs for Large High-Dimensional Data Sets on GPU Clusters

Ali Dashti, Ivan Komarov, Roshan M. D'Souza*

Department of Mechanical Engineering, Complex Systems Simulation Lab, University of Wisconsin-Milwaukee, Milwaukee, Wisconsin, United States of America

Abstract

This paper presents an implementation of the brute-force exact k-Nearest Neighbor Graph (k-NNG) construction for ultra-large high-dimensional data cloud. The proposed method uses Graphics Processing Units (GPUs) and is scalable with multi-levels of parallelism (between nodes of a cluster, between different GPUs on a single node, and within a GPU). The method is applicable to homogeneous computing clusters with a varying number of nodes and GPUs per node. We achieve a 6-fold speedup in data processing as compared with an optimized method running on a cluster of CPUs and bring a hitherto impossible k-NNG generation for a dataset of twenty million images with 15 k dimensionality into the realm of practical possibility.

Citation: Dashti A, Komarov I, D'Souza RM (2013) Efficient Computation of k-Nearest Neighbour Graphs for Large High-Dimensional Data Sets on GPU Clusters. PLoS ONE 8(9): e74113. doi:10.1371/journal.pone.0074113

Editor: Attila Gursoy, Koc University, Turkey

Received January 10, 2013; **Accepted** August 1, 2013; **Published** September 23, 2013

Funding: This work is partially funded by the National Science Foundation (NSF) through the grants CNS-0968519 and CCF-1013278. No additional external funding was received for this study. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

* E-mail: dsouza@uwm.edu

Introduction

K-Nearest neighbor graphs have a variety of applications in bioinformatics [1,2], data mining [3], machine learning [4,5], manifold learning [6], clustering analysis [7], and pattern recognition [8]. The main reason behind the popularity of neighborhood graphs lies in their ability to extract underlying information about structure and governing dimensions of data clouds. The k-NNG problem is similar to the k-NN problem and a k-NNG can be built by repeatedly applying the k-NN query for every object in the input data once a convenient search indexing data structure has been built. Such search data structures include kd-trees [9], BBD-trees [10], random-projection trees (rp-trees) [11], and hashing based on locally sensitive hash [12]. These method focus on optimizing the k-NN search, i.e., finding k-NNs for a set of query points w.r.t. a set of points with which the search data structure is built, ignoring the fact that every query point is also a data point. These methods are generally less efficient compared with one that focuses on k-NNG construction directly.

The problem of constructing an exact k-NNGs has been investigated extensively to avoid the $O(n^2)$ complexity of the brute force method. An $O(n\log^{d-1}n)$ was presented in [13]. [14] presented a $O(c^d\log n)$ algorithm and [15] presented a worst case $O((c'd)^d n\log n)$. In [16] two practical algorithms for k-NNGs based on recursive partitioning and pivoting have empirically shown time complexities $0.685e^{0.23d}n^{1.48}$ and $0.858e^{0.11d}n^{2.15}$, respectively. All these algorithms have a time complexity that is exponential in the dimension d of the data. The same exponential dependence on dimension of time complexity is seen in methods based on space filling curves such as the Hilbert's curve [17] and Morton's curve [18]. There are several approximate methods that can handle moderately high dimensional data, typically with a trade-off between speed and accuracy. One set of techniques typically is based on a hybrid of spatial subdivision up to a threshold granularity and small scale brute force evaluation or heuristics for refinement [19–21]. These methods rapidly lose accuracy or speed or both when the dimension of the data exceeds 10^3.

The curse of dimensionality leads to a belief supported by many researchers that the most efficient method for finding k-NNGs for high-dimensional data clouds is in fact the brute force method [22]. The brute force algorithm breaks into two parts: distance calculation and comparison. In the distance calculation part, all distances between all points for graph construction are computed. This results in a $M\times N$ distance matrix, where M is the number of query points and N the number of data-base points. Next, each row of the matrix is sorted to get the nearest k neighbors to each of the query points. Recently, there have been several methods that accelerate brute force k-NN and k-NNGs on graphics processing units. They primarily differ in the manner of selecting k smallest elements in every row in the distance matrix. In [23,24] each row of the distance matrix is processed by one thread. Each thread uses an modified insertion sort algorithm to select the k nearest elements. The number of steps that each thread takes to process a given stage is not pre-determined. This can lead to branch divergence and loss of efficiency. Also, since the insertion sort data structure is stored in global memory, this can cause a significant loss in performance due to un-coalesced memory writes. In [25], every row of the distance matrix is processed by a thread block. A heap (one per row) maintains the nearest k smallest distances. Each thread in the thread block strides through the row reading and storing the element in a buffer if it is smaller than the largest element in the heap. When the buffer fills up, all threads synchronize and then push their elements on to the heap in a serialized manner. The last stage of this algorithm can be extremely expensive especially for large k. In [26], a thread block is used to process a single row. Each thread in the thread block

strides through the array storing the k smallest elements in a local heap maintained in global memory. Next, all the thread heaps are merged into 32 heaps in shared memory threads in a single thread warp. Finally a single thread merges the 32 heaps in shared memory to find the k nearest neighbors. Storing heaps in global memory has the same disadvantages of thread divergence and uncoalesced memory write as in [25]. Finally, for $k>192$ it may not even be possible to execute the method in share memory even on the latest GPUs. All these methods dramatically lose performance for large k. In [27] a radix sort based approach is used to select the k nearest neigbhbors. They claim that for large data sets, especially for a large number of queries, the selection process dominates. A simple complexity analysis suggests that this is quite impossible ($O(dmn)$ for distance calculation vs $O(mn\log n)$ for sorting where $d\approx 120$ is the data dimension, $m\approx 2000$ is the number of query points and $n\approx 30{,}000$ is the number of data points). A closer examination shows that they process each row of the distance matrix in a separate sort. For n that fit into GPU memory, this process underuses GPU resources.

Our target application is Manifold Embedding to recover structure and conformations from a large data set of images with high noise [28]. The underlying assumption behind manifold embedding states that a cloud of high dimensional data makes a low dimensional hyper-surface in high dimensional space. The generated hyper surface, called a manifold, contains the information about the individual objects, 3D structure of images, and the system that generated the data. The main computational part of manifold embedding is the construction of a k-NNG for the image data set that can be normalized to a diffusion map in subsequent stages of algorithm. There are upwards of 10^7 images with each image having $15k$ pixels. We require $k>200$ for accurate results. The nature of the data, coupled with the accuracy requirements, makes the brute-force algorithm the only viable alternative. In this paper we describe our parallelized brute-force k-NNG algorithm on a cluster of graphics processing units. We describe three levels of parallel distribution of tasks and data partitioning: between nodes, between multiple GPUs within a node, and finally within the GPU. We have also developed a novel algorithm based on sorting for the comparison and selection step of the brute force k-NNG routine. Our benchmarks show that our routine outperforms the best GPU-based comparison and selection methods. Overall, we achieve a $6\times$ gain in performance over a distributed solution on CPU clusters using the fastest libraries that are available. This bring a hitherto impossible k-NNG generation for a dataset of twenty million images with $15k$ dimensionality into the realm of practical possibility.

Methods

Given a set of vectors $V=\{v_1,v_2....v_N\}$ with $v_i\in\mathbb{R}^D$, k-NNG finds for each vector $v_i\in V$ a subset of k nearest vectors $I(v_i)\subset V$. Proximity is defined using a metric d. In this work, we are primarily concerned with the Euclidian distance metric given by:

$$d(v_i,v_j)=\sqrt{\|v_i-v_j\|^2}$$

The square of the distance metric can be written as:

$$\begin{aligned} d^2(v_i,v_j) &= \|v_i-v_j\|^2 \\ &= (v_i-v_j)^T(v_i-v_j) \\ &= v_i^Tv_i+v_j^Tv_j-2v_i^Tv_j \\ &= \|v_i\|^2+\|v_j\|^2-2v_i^Tv_j \end{aligned}$$

Note that D dimensional vector v_i is represented as

$$v_i=\{v_{i1},v_{i2}.....v_{iD}\}^T$$

Defining matrices $A^{N\times D}$, $B^{N\times N}$ be given by

$$A=\begin{pmatrix} v_1^T \\ v_2^T \\ .. \\ v_N^T \end{pmatrix}$$

$$B=\begin{pmatrix} \|v_1\|^2 & \|v_1\|^2 & .. & \|v_1\|^2 \\ \|v_2\|^2 & \|v_2\|^2 & .. & \|v_2\|^2 \\ .. & .. & .. & .. \\ \|v_N\|^2 & \|v_N\|^2 & .. & \|v_N\|^2 \end{pmatrix}$$

Defining the matrix of squared distances S as

$$S=\begin{pmatrix} d^2(v_1,v_1) & d^2(v_1,v_2) & .. & d(v_1,v_N) \\ d^2(v_2,v_1) & d^2(v_2,v_2) & .. & d(v_2,v_N) \\ ... & .. & .. & .. \\ d^2(v_N,v_1) & d^2(v_N,v_2) & .. & d(v_N,v_N) \end{pmatrix}$$

We can now write the following equation

$$S=B+B^T-2(AA^T).$$

Finding the set of k nearest neighbors for vector v_i involves sorting the i^{th} row of S and picking the column indices corresponding to the k smallest distances. In our application of interest, $N=10^6$ to 10^7. Therefore, computing and storing S will require 1.8–18 terabytes of memory. Therefore, our approach is to compute k nearest neighbors in parts. As illustrated in Figure 1., the computation of the squared distance matrix S is split into $P\times P$ partitions. Consequently, each portion $S_{(I,J)}$ is computed as:

$$S_{(I,J)}=B_I+B_J^T-2(A_IA_J^T)$$

where A_I,B_I $I=1,2...P$ are partitions of A,B respectively.

Other metrics such as Cosine and Pearson distances can also be incorporated in the k-NNG algorithm. The Cosine distance is given by:

$$S_{(v_1,v_2)}=\frac{v_1^Tv_2}{\|v_1\|\|v_2\|}$$

The Pearson distance is given by:

$$S_{(v_1,v_2)}=\frac{(v_1-\mu)^T(v_2-\mu)}{\|v_1-\mu\|\|v_2-\mu\|}$$

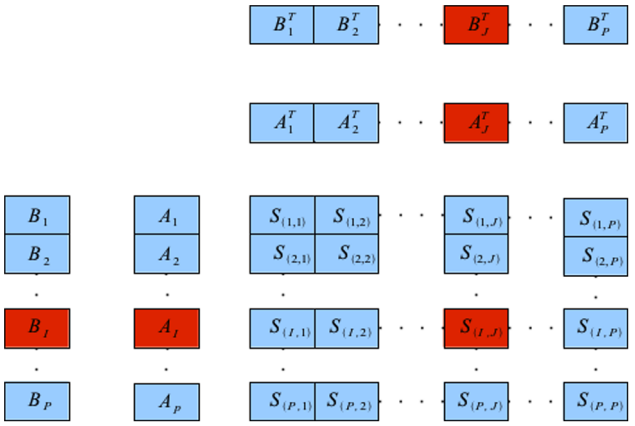

Figure 1. Data partitioning for distributed parallel execution. The squared distance matrix S is split into $P \times P$ partitions where P is an integral multiple of Q, the number of nodes. The computations of the partitions and the subsequent computations of the k-NNs are distributed between different nodes.
doi:10.1371/journal.pone.0074113.g001

where μ is the average vector of the dataset. The Pearson distance is essentially a centered Cosine distance and therefore an additional pre-processing step for centering is necessary. Clearly, these distance metrics can be formulated in terms of matrix multiplication ($v_i^T v_j$) and operations using vector norms. Therefore, the same decomposition and task partitioning applicable in Euclidian case can be used.

Distribution of Data and Tasks between Computing Nodes

Computing clusters typically have several nodes that are connected by high-speed interconnects. One of the nodes is designated as the head node. The head node typically co-ordinates the tasks between different worker nodes. Each node has its own hard disk. In addition, there is a large shared disk accessible through parallel (I/O) by all nodes that typically hold input data and results. The worker nodes, with smaller local disk space, copy input data as and when required from the shared disk.

Table 1 illustrates the steps involved in computing the k-NNG. For load balancing, we distribute computing of the partitions of S in a block cyclic manner. Since S is symmetric, we only need to compute the upper triangular portion, i.e., all sub-matrices $S_{(I,J)} | J \geq I$. Figure 2 illustrates the task partitioning between

Table 1. Parallel k-NNG algorithm.

1 Split A into P partitions $A_1, A_2..A_P$ where P is a multiple of Q
2 Each node q reads $A_I \mid I : I\%Q = q$
3 Each node builds the vector $\hat{B}_I$
4 All nodes build the whole vector $\hat{B}$ using an all_gather opreation
5 **for** $I = 1$ ***to*** $I = P$ **do**
6 Each node reads a portion of A_I from the shared disk in parallel
7 A_I is reconstructed at each node using an all_gather operation
8 **for** $K = I$ **to** $K < P$ **do**
9 **for** $J = K$ **to** $J = K + Q$ *in parallel* **do**
10 Read A_J from local memory of node $q = J\%Q$
11
12 Compute $S_{(I,J)}$
13 Compute the block row and block column k-NNs
14 Merge block column k-NNs
15 **end**
16 Merge block row k-NNs
$K = K + Q$
17 **end**
18 All nodes communicate merged block row k-NNs to node $q = I\%Q$
19 Node q merges all the received block row k-NNs with its block column k-NNs for column I to get global k-NNs for block row I
20 **end**

doi:10.1371/journal.pone.0074113.t001

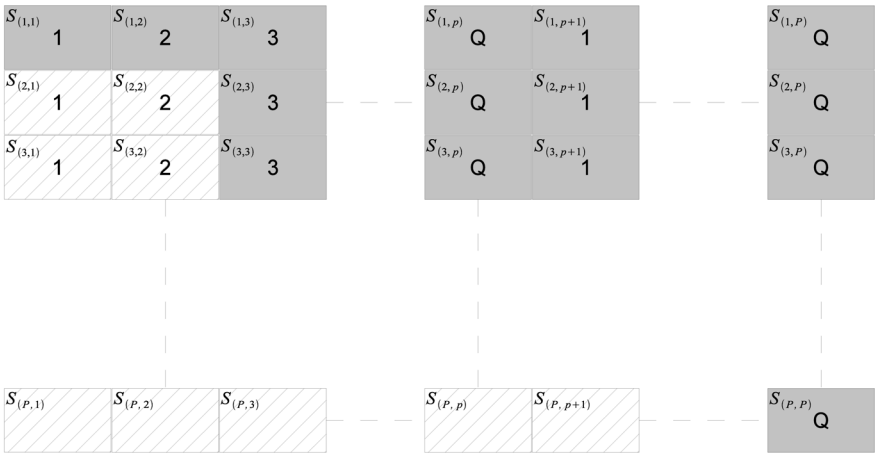

Figure 2. Node assignments for processing partitions of S. Due to symmetry, $S_{(I,J)}=S^T_{(J,I)}$. Therefore, only $S_{(I,J)}\ |I\leq J$ have to be computed. $S_{(I,J)}$ is processed by node q where $J\%Q=q$.
doi:10.1371/journal.pone.0074113.g002

different nodes. For example, in block row I, node q will process all sub-matrices $S_{(I,J)}\ J|J\%Q=q,J\geq I$. In the beginning, the input data matrix A is split into P parts. Each node q then reads portion A_J where $q=J\%Q$ into its local drive. The calculation of the distance sub-matrix $S_{(I,J)}$ and the calculation of associated k-NNs is handled by node q, where $q=J\%Q$. The computation proceeds in a block-row by block-row manner (lines 5–20 in Table 1). Any sub-matrix $S_{(I,J)}$ requires inputs A_I,A_J,B_I,B_J. Of these A_I,B_I are used by all nodes that are processing the I^{th} block row. Each node q also requires $B_J,A_J\ |J:J\%Q=q$. For each block row I, the partition A_I is read in parts in parallel by all nodes from the shared drive. All nodes then share the parts that they each read through an 'all_gather' operation [29] to build a local copy of A_I in memory. Each node q then reads $A_J\ |J:J\%Q=q$ from its local drive. Since the local disk read operation is slower, we use a memory buffer and overlap computation and disk read operation to completely hide latency. We do not actually build the matrix $\boldsymbol{B}$. Instead, the vector $\hat{\boldsymbol{B}}=\{\|v_1\|^2,\|v_2\|^2,.....\|v_N\|^2\}^T$ is computed in advance and stored in the RAM of each node. Even for $N=10^7$ the size of $\hat{\boldsymbol{B}}$ ($\sim$10 MB) is quite small compared to the RAM in each node (48 GB). Each node q computes the vector norms for all vectors in the partitions $A_J\ |J\%Q=q$ resident on its disk space

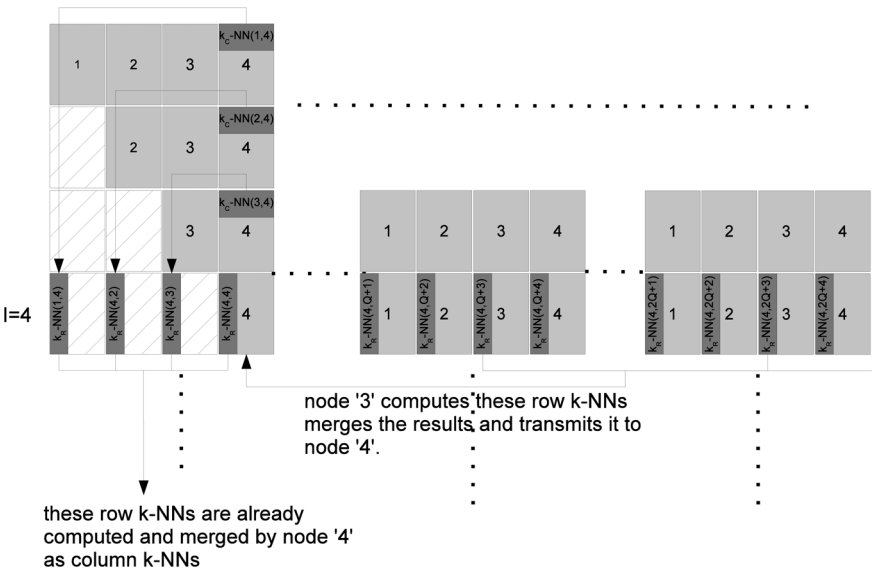

Figure 3. Processing global k-NNs. In this figure node $q=4$ is responsible for calculating the global k-NNs of all vectors that are in A_4. This is done by computing and merging local row k-NNs of $S_{(4,1)},S_{(4,2)},...S_{(4,p)},.....S_{(4,P)}$. Note that the local row k-NNs w.r.t. $S_{(4,J)}\ J=1,2,3$ have already been calculated when node 4 calculated local block-column k-NNs w.r.t. $S_{I,4}\ I=1,2,3$. The merged results are stored in a heap. The k-NNs w.r.t $S_{(4,J)}|\ J>4$ are cooperatively computed by all nodes. For example, node $q=3$ successively computes and merges k-NNs w.r.t. $S_{(4,Q+3)},S_{(4,2Q+3)},...,S_{(4,P-Q+3)}$. It then transmits the results to node $q=4$, which receives results from other nodes and does a global merge.
doi:10.1371/journal.pone.0074113.g003

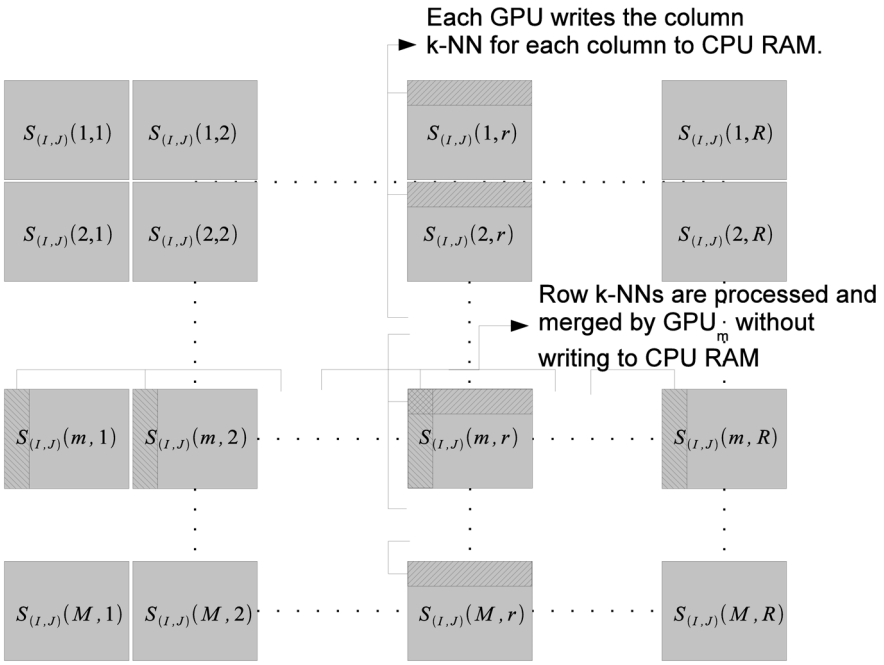

Figure 4. Processing local k-NNs within nodes. The sub-problem assigned to a node is finding the row and column k-NNs w.r.t $S_{(I,J)}$. $S_{(I,J)}$ is divided into $M \times R$ partitions. All partitions $S_{(I,J)}(m,1),S_{(I,J)}(m,2),\ldots,S_{(I,J)}(m,R)$ are processed by GPU m. The row k-NNs are processed within GPU memory and the merged results are written to CPU RAM. The column k-NNs are written to CPU RAM. Later, each of the local column k-NNs are merged by a single GPU.
doi:10.1371/journal.pone.0074113.g004

locally. Using the 'all_gather' operation, the nodes can then share data among themselves and build a complete local copy of $\hat{\boldsymbol{B}}$.

Once the partition $S_{(I,J)}$ is computed, the local block k-NNs with respect to both the rows (k_R-NNs) and columns (k_C-NNs) are computed (lines 9–14 in Table 1). Since the matrix S is symmetric, the local block k_C-NNs w.r.t. partition $S_{(I,J)}$ are identical to the local block k_R-NNs w.r.t. partition $S_{(J,I)}$. Therefore, each node q maintains one heap per block column $J|J\%Q=q$ of S that it processes. Each of these heaps contains the merged local k_C-NNs w.r.t. partitions $S_{(I,J)}\,|I=1,2,..J,\ J\%Q=q$. For example, as shown in Figure 3, node 4 maintains one heap for each of the columns $4,Q+4,\ldots(P-Q+4)$. The heap for column 4 will contain the merged block k_C-NNs for $S_{(1,4)},S_{(2,4)},S_{(3,4)},S_{(4,4)}$. The heap for column $Q+4$ will contain the merged local k_C-NNs for partitions $S_{(1,Q+4)},S_{(2,Q+4)},\ldots\ldots,S_{(Q+4,Q+4)}$.

The node q is used to compute the global k-NNs for all vectors in partition $A_I\ |I\%Q=q$. The global k-NNs for all the vectors in A_I are generated by merging the local block k_R-NNs w.r.t. partitions $S_{(I,J)}\ J=1,2,\ldots,P$. However, the merged results of the local block k_R-NNs w.r.t all partitions $S_{(I,J)}\ |J=1,2,\ldots,I$ are already available in node q from the local block k_C-NNs computed when processing rows $1,2\ldots I$. The block k_R-NNs w.r.t. all partitions $S_{(I,J)}\ J=I+1,I+2,\ldots\ldots,P$ are cooperatively computed by different nodes. Each node maintains a heap to merge the results of finding the local k_R-NNs of the partitions that it processes (line 15 in Table 1). At the end, the merged results are communicated to the node processing the global k-NNs for the block row I for merging at the global level (line 18 in Table 1). For example, as illustrated in Figure 3, for $I=4$, node 3 will compute block k_R-NNs w.r.t. all partitions $S_{(4,Q+3)},S_{(4,2Q+3)},\ldots S_{(4,Q-P+3)}$. The results will be merged and stored in a heap. At the end of the computation, the results in the heap will be communicated to node 4. Node 4 will have computed all k_R-NNs w.r.t. $S_{(4,1)},S_{(4,2)},S_{(4,3)}$ and merged them while processing block rows $I=1,2,3$. These results and the results communicated to it by all other nodes processing parts of block row $I=4$ will be merged by node 4 to compute global k-NNs for all vectors in A_4.

Data partitioning parameters. Our tests reveal that compute time dominates communication time. For a variety of

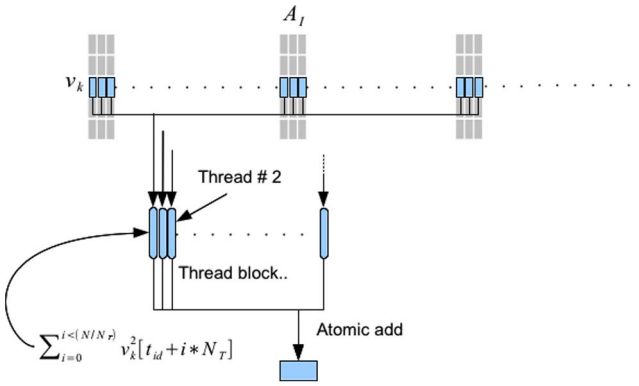

Figure 5. Finding vector norms. Each thread block is assigned to compute the norm of one vector in A_I. Each thread t strides through the vector and computes the sum $\sum v_k^2[i]\ |i\%N_T=t$, where N_T is the number of threads in a thread block. Finally, an atomic add operation is used to add all the sums within each thread into a location in global memory on the GPU.
doi:10.1371/journal.pone.0074113.g005

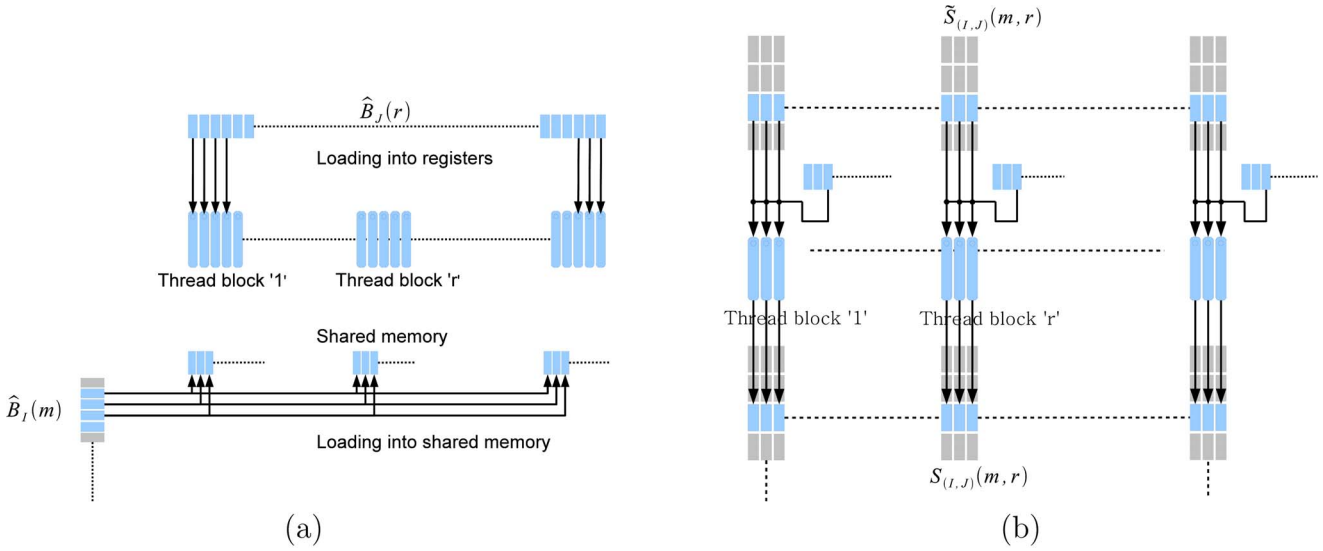

Figure 6. Summation kernel. Calculation of every row of $S_{(I,J)}(m,r)$ involves $\hat{B}_J(r)$ and one element of $\hat{B}_I(m)$ per row. Therefore, each thread loads an element of $\hat{B}_J(r)$ into a register. These data are reused to compute all rows of $S_{(I,J)}(m,r)$. Next, one thread per block reads the corresponding element of $\hat{B}_I(m)$ into shared memory. Next, each thread reads an element of $\tilde{S}_{(I,J)}(m,r)$ and adds to it the element of $\hat{B}_J(r)$, which is in the register and the $\hat{B}_I(m)$ into shared memory to generate the corresponding element of $S_{(I,J)}(m,r)$.
doi:10.1371/journal.pone.0074113.g006

decomposition sizes, we find that communication time is no more than 2% of compute time. Therefore, by making the decompositions as large as possible, any communication costs can easily be masked by overlapping with computing. The size of the partition is therefore governed solely by the amount of node RAM and is given by:

$$\begin{aligned}&\text{amount of RAM in each node}\\ &> \frac{(N)(d)(\text{size of float/double})}{P} + \frac{2(N)(d)(\text{size of float/double}}{(P)(R)}..\\ &+(Q)(k)((size\ of\ float/double\ +\ size\ of\ int)\left(\frac{N}{P}\right)\\ &+(k)(size\ of\ float/double\ +\ size\ of\ int)\left(\frac{N}{Q}\right)..\\ &+(k)(M)(size\ of\ float/double+\ size\ of\ int)\left(\frac{N}{(P)(R)}\right)+OS\end{aligned}$$

where, P is the smallest multiple of Q that satisfies the above relation, R in the size of the partition of A_J within the node and is dependent on GPU RAM size. Here the first term in the right hand side is the size of A_I, the second term is the size of a buffered portion of A_J^T (one buffer portion is for reading from the disk while the second is for feeding the GPUs. These buffers are swapped at the end of computation.), the third term is the size of the buffer to store Q row k-NNs (this data is used to compute the global k-NNs. The data structure is a tuple consisting of a float/double to represent the distance and a int for index.), the fourth term accounts for the block column k-NN storage for all columns being processed by a node, the fifth term represents the memory required for partial column k-NN generated by the M GPUs in each node, and the sixth term represents the memory requirements for the operating system. R is given by the smallest integer value such that the following relationship holds:

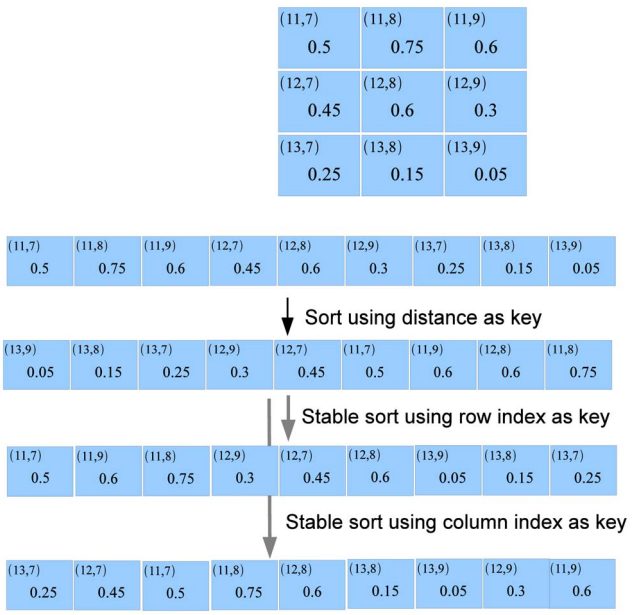

Figure 7. k-NN using sorting. Here we illustrate the k-NN search for 3 vectors. The distance matrix is stored along with the column and row indices in a row-major format. First, we sort the entire distance matrix with the distance as the key. The result is next sorted in a stable manner first with the column as the index and then as separately with the row as the index. We then pick the closest k distances both for the columns and the rows results.
doi:10.1371/journal.pone.0074113.g007

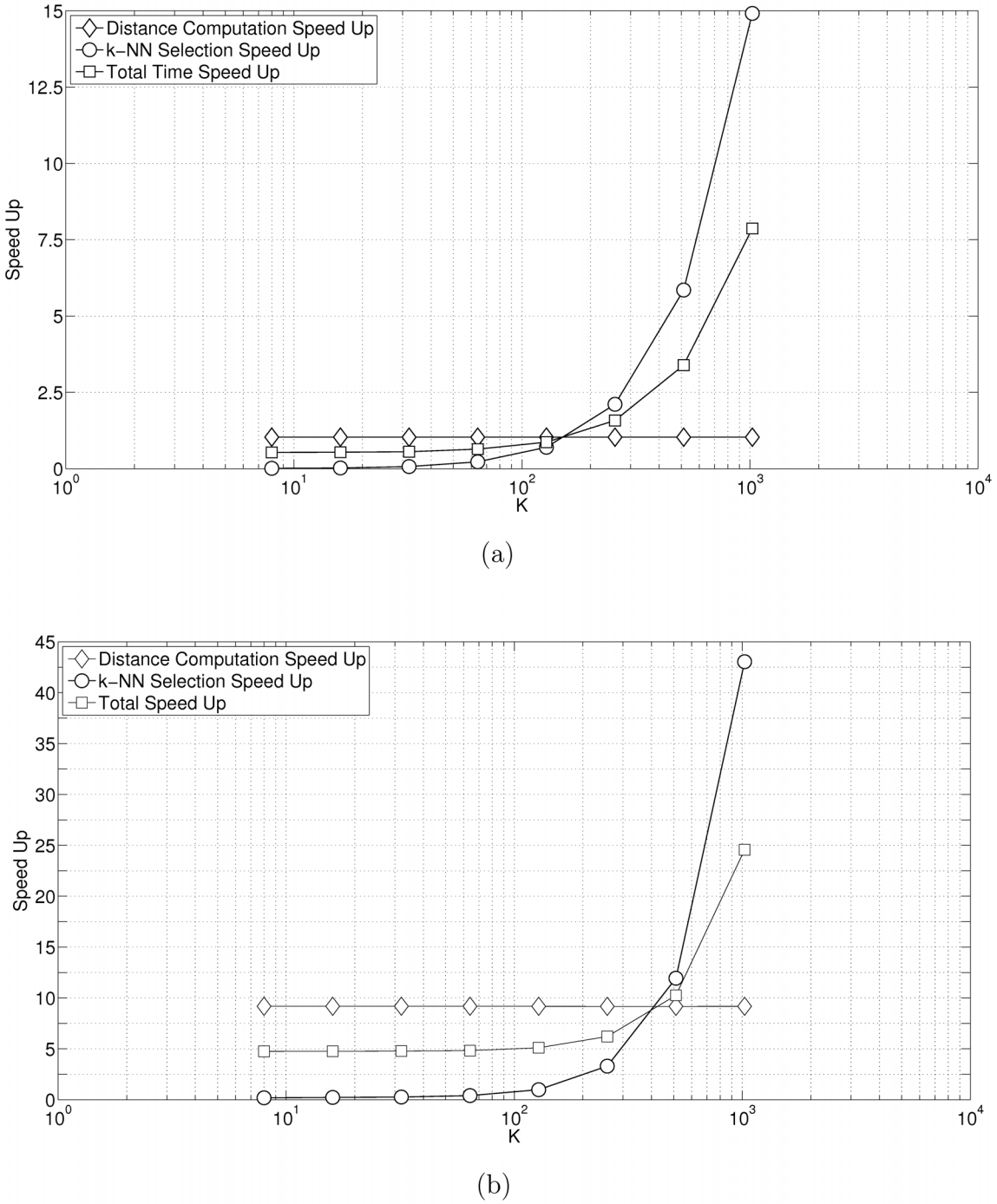

Figure 8. Performance benchmarks for varying *k*. In this test our input data has the dimension $d=4096$ and the number of input objects/vectors $n=16384$. Figure 7(a) shows the performance vs. [23]. Figure 7(b) shows the performance vs. [24].
doi:10.1371/journal.pone.0074113.g008

$$\text{amount of RAM on GPU} > \frac{(N)(d)\ (\text{size of float/double})}{PM} + \frac{(N)(d)\ (\text{size of float/double})}{(P)(R)} + (k)(R)\ (\ \textit{size of float/double} + \textit{size of integer})$$

the first term represents the storage for a portion of A_I, the second term represents the storage for a portion of A_J and the third term represents the storage for partial R row k-NNs which will be merged by the GPU.

Distribution of Tasks and Data Within Nodes

We assume that each node has M GPUs. In our current setup, $M=2$. As mentioned previously, each node is responsible for computing a partition $S_{(I,J)}$ of the squared distance matrix. A_Is

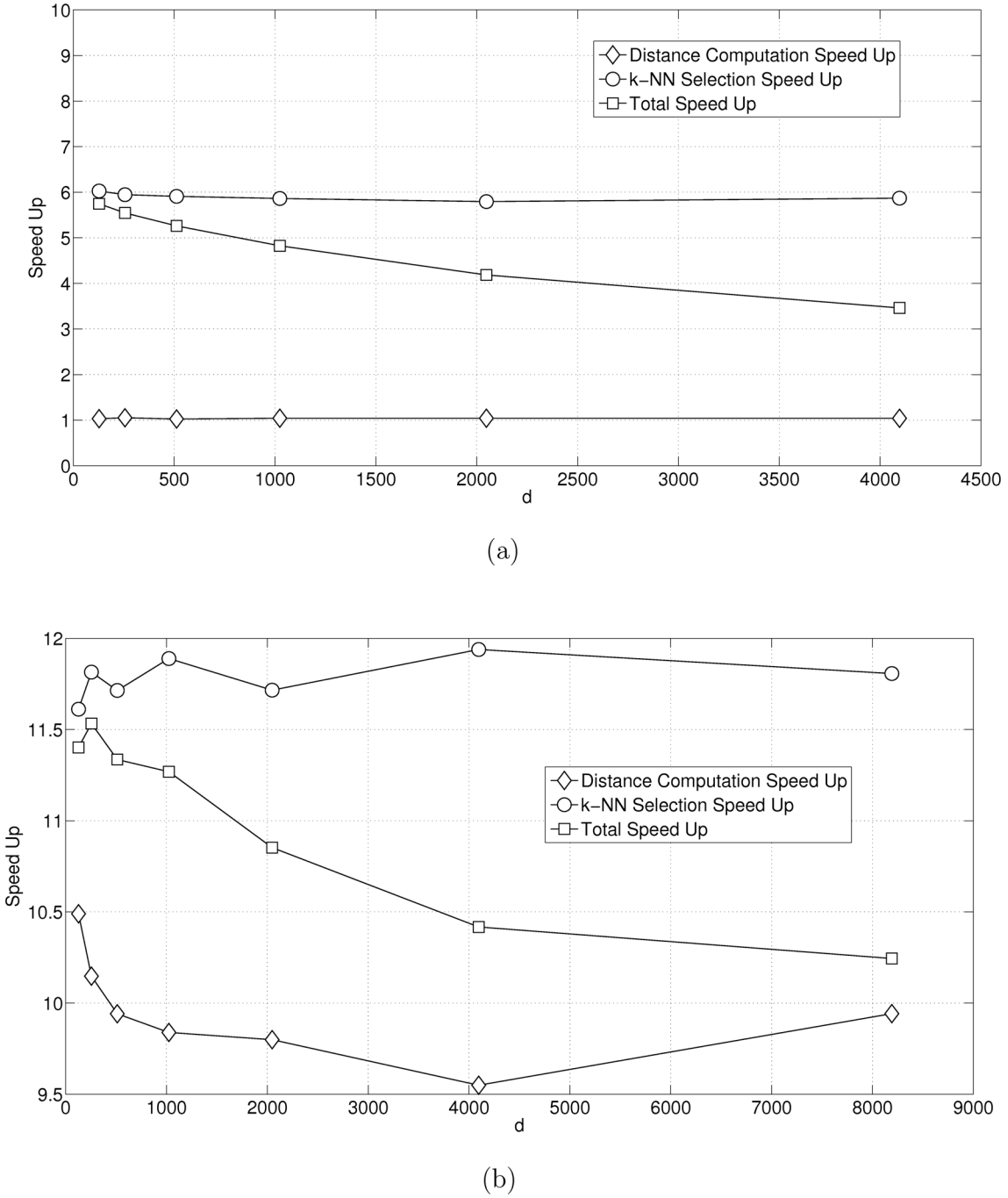

Figure 9. Performance benchmarks for varying d. In this test our input data has the number of closest neighbors $k=512$ and the number of input objects/vectors $n=16384$. Figure 7(a) shows the performance vs. [23]. Figure 7(b) shows the performance vs. [24].
doi:10.1371/journal.pone.0074113.g009

are read from the head node through parallel I/O. A_Js are read from the local disk. Within the node, A_I is divided into M equal partitions. A_J is divided into R partitions.

Task distribution within each node is described in Figure 4. Each GPU m is responsible for computing the sub-matrices $S_{I,J}(m,r)$ and the associated row and column k-NNs. This process requires portions of the input vector data matrices $A_I(m)$ and $A_J(r)$. Once again the computation proceeds in a block row manner. Therefore the portion $A_J(r)$ is read by all GPUs while as the portion $A_I(m)$ is read by the GPU m. When the computation of $S_{(I,J)}(1,r),S_{(I,J)}(2,r)$, ...,$S_{(I,J)}(m,r)$ is being done by the M GPUs, the node simultaneously reads the file partition $A_J(r+1)$ into the RAM. The node also has the norm vector $\hat{\boldsymbol{B}}$ in its RAM. $\hat{\boldsymbol{B}}$ is partitioned in two ways: one has M partitions with each of these partitions going to the M different GPUs and the other has R partitions, with each partition being read sequentially by all GPUs. When the computation of $S_{(I,J)}(m,r)$ is complete, the column k-NNs as well as the row k-NNs are computed using sorting. The column k-NNs are written back to CPU memory while the row k-NNs are kept on global memory to be merged.

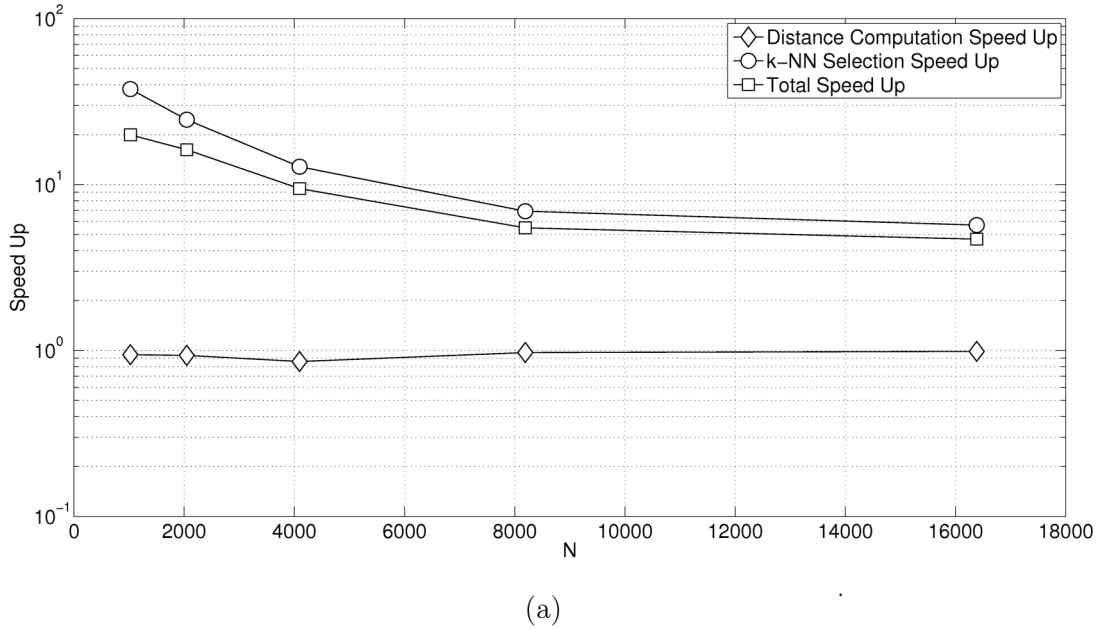

Figure 10. Performance benchmarks for varying n. In this test our input data have the dimension $d=1024$ and the number of input objects/vectors $n=16384$. Figure 7(a) shows the performance vs. [23]. Figure 7(b) shows the performance vs. [24]. As n increases, the total performance gain asymptotically approaches that of matrix multiplication because for large n, this computation dominates.
doi:10.1371/journal.pone.0074113.g010

Note that $S_{(I,J)}(m,r)$ $r=1,2..,R$ are being processed by the same GPU m, therefore, it makes sense to merge row k-NNs in the GPU memory without write back to CPU RAM. However, $S_{(I,J)}(m,r)$ $m=1,2,..M$ are being processed by different GPUs, therefore, the column k-NNs are generated by different GPUs. The accumulated column k-NNs are then merged by one GPU per column. The final result of this operation is the local row and column k-NNs for the partition $S_{(I,J)}$.

We use OpenMP multi-threading [30]. Each node runs $M+1$ CPU threads. M threads control the M GPUs while one thread is in charge of I/O from the local disk as well as the shared disk.

Distribution of Tasks and Data within GPUs

The tasks that are accomplished within each GPU include the following:

- Finding vector $\hat{\boldsymbol{B}}$ of input data norms
- Dense matrix multiplication to generate the result $2A(m)_I A_J^T(r)$
- Summation to find the result $S_{(I,J)}(m,r)=B_I(m)+B_J^T(r)-2A_I(m)A_J^T(m)$
- Finding local k-NNs based on $S_{(I,J)}(m,r)$

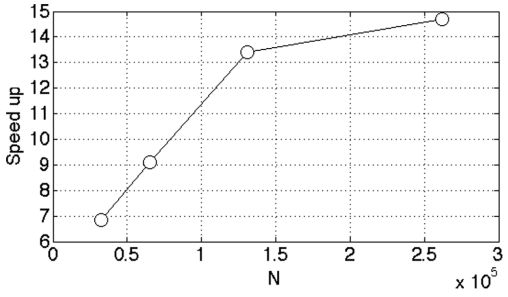

Figure 11. Performance benchmarks for multi-GPU execution. In this test we used 2 GPUs. For the implementation in [24] the 2 GPUs (Tesla 2050) were mounted on a single desktop machine. For our implementation, we use 2 nodes in our GPU cluster and opted to use only one GPU per node. The input data had the dimension $d = 16384$, and the number of closest neighbors $k = 512$.
doi:10.1371/journal.pone.0074113.g011

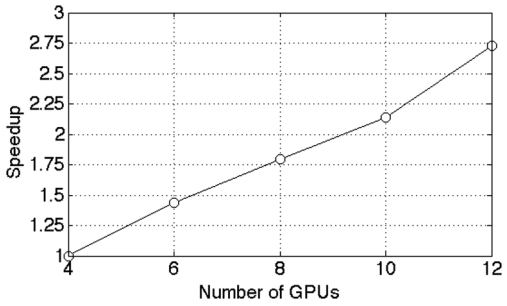

Figure 12. Performance scaling with number of GPUs. In this test we used a data set with $n = 1966080, d = 2048, k = 512$. We achieve a performance speedup that is slightly below linear in the number of GPUs.
doi:10.1371/journal.pone.0074113.g012

Each of these tasks is coded as *kernels*. In our clusters we have Tesla C2050 compute GPUs from NVIDIA. We use the CUDA programming environment [31] to code our kernels.

Finding vector norms. Finding vector $\hat{B}$ of input data norms is done once in the beginning at the same time that the input files A_J are communicated to each node. For example, if node q will receive all files $A_J | q = J\%Q$. When the file is received, it is partitioned into M partitions, one partition per GPU. Although there is a library function to calculate vector norms in CUBLAS [32], using it will be inefficient since the vectors are relatively large in number ($N = 10^6$ to 10^7) with much smaller dimension $D \approx 15000$. Finding the norm of vectors one at a time will not fully use GPU resources. We have written our own kernel that overcomes underuse of GPU resources by computing multiple vector norms in one kernel invocation.This process is illustrated in Figure 5. Every vector in partition $A_J(m)$ is processed by one thread block. All threads in the thread block cooperatively compute the vector norm. Each thread strides through the vector components adding the square of the entries with a stride length equal to the number of threads in the thread block. Finally, all threads in the thread block write to a single global memory location using atomic-add.

Dense matrix multiplication. For dense matrix multiplication $A_I(m)A_J^T(m)$, we use the optimized library function from CUBLAS [?]. The result of the dense matrix multiplication $\tilde{S}_{(I,J)}(m,r)$ is stored in global memory. For summation, we have written a special kernel to take advantage of the particular structure to minimize memory transactions. Figure 6 illustrates the summation kernel. As mentioned previously, we do not build the matrices $B_I(m)$ and $B_J^T(r)$ but simply use the portions of the vector of input data norms $\hat{B}_I(m)$ and $\hat{B}_J(r)$. We process $S_{(I,J)}(m,r)$ one row at a time. Every thread reads one element of $\hat{B}_J(r)$ into its register. Next, the thread block reads a section of $\hat{B}_I(m)$ into shared memory. Finally, all threads simultaneously update the entire row of $S_{(I,J)}(m,r)$. Each thread reads the corresponding element of row m of $\tilde{S}_{(I,J)}(m,r)$, adds to it the corresponding element of $\hat{B}_J(r)$ that is in its register, and an element $\hat{B}_I(m)$ that is in shared memory. In reading an element $\hat{B}_I(m)$ by all threads in the thread block, we use the broadcast mechanism in GPUs.

Batch index sort for k-NN. We could obviously sort each column and each row separately. However, this is not efficient because the resources on the GPU are not fully used. Also, for sorting according to columns, we will need to execute an expensive operation to rearrange the data in the column major format, Instead, we use a process we called Batch Index Sorting. Note that in GPU RAM, $S_{(I,J)}(m,r)$ is laid out as a linear array in a row-major format. Each element of $S_{(I,J)}(m,r)$ is also then associated with its row index and column index. We use radix sort with the elements of $S_{(I,J)}(m,r)$ as key. In the next two steps, we execute an order preserving sort of the result of the previous step with the column index as the key. Separately, we also execute an order preserving sort with the result of the first sort with the row index as the key. These two sorting operations generate the nearest neighbors for each column and row. We then execute a separate kernel to extract the k-NNs for each row and column w.r.t. $S_{(I,J)}(m,r)$. Figure 7 illustrates and example with 3 data vectors and 3 query vectors.

Results

We tested our implementation against that of [24] and [23]. All implementations were compiled using C++ using appropriate compiler optimization flags. The implementations were run on a NVIDIA Tesla C2050. We used synthetic data sets. We benchmarked distance computation, k-NN selection, and total time. Figure 8 shows the comparison when we varied k. In this test we set the data dimension to $d = 4096$ and the number of objects $n = 16384$. Figure 8(a) shows the speedup versus [23]. The distance computation is almost the same because both works formulate distance computation as a matrix-matrix multiplication and use the optimized CUBLAS library. Our version of k-NN selection breaks even with the work of [23] at about $k = 128$ and outperforms it by $15\times$ for $k = 1024$. Overall our implementation has a performance advantage of $7.87\times$. Figure 8(b) shows the speedup versus [24]. For this test we re-formulated the Pearson distance computation to enable the use of optimized matrix-matrix multiplication. Consequently, our distance implementation has a roughly $9\times$ performance advantage. The k-NN implementation is a per thread linear insertion sort with each thread handling one row of the distance matrix. Our implementation breaks even at $k = 128$ and ends up with a $42\times$ performance advantage when $k = 1024$. Overall, our implementation has a $24\times$ performance gain at $k = 1024$.

In the second sets of tests we kept $n = 16384$ and $k = 512$ and varied d. Figure 9(a) shows the comparison with [23]. Both the speedup with respect to distance and with respect to k-NN selection remains constant at $\approx 1\times$ and $\approx 6\times$, respectively. However, the proportion of time for distance computation increases linearly with d. Therefore, we see that the performance gain for total time decreases from $5.7\times$ to $3.5\times$. Figure 9(b)

shows the speedup with respect to [24]. Once again, the speed up with respect to distance and with respect to k-NN selection remains constant at $\approx 9.5\times$ and $\approx 10\times$, respectively. Once again as d increases, the proportion of time for distance computation increases linearly with d. For large d, the graph shows stabilization of the overall speedup at $10.25\times$.

In a third sets of tests we kept $d=1024$ and $k=512$ and varied n. Figure 10(a) shows the comparison with [23]. For small n, the speedup with respect to selection is $\approx 200\times$. As n increases, the performance gains taper off to $\approx 12\times$. Overall speedup starts off at $\approx 100\times$ and falls to $\approx 11\times$. Figure 10(b) shows the comparison with [24]. Once again for small n, the speedup with respect to selection is $\approx 37\times$. As n increases, the speedup tapers off to $\approx 5.6\times$. Overall speedup starts off at $\approx 20\times$ and tapers off to $\approx 4.7\times$. Finally, for the tests with varying d and n, while our implementation was able to handle a model size up to $n=32{,}767$, the implementations by [23] could only handle a model size up to $n=16{,}384$.

We also tested our implementation vs. [24] for multi-GPU configuration. While the implementation in [24] requires all GPUs be on a single computer (connected through PCI Express bus with OpenMP multi-threading), our implementation is designed for execution on GPU clusters, i.e., the scalability is much larger. In the tests, we ran the implementation in [24] on two GPUs on a single desktop and our implementation was run on two nodes of a cluster, with each node containing a single GPU. We used a combination of MPI and OpenMP for multi-GPU execution. Figure 11 shows the result. Here $d=16384$ and $k=512$. We achieve up to $15\times$ overall speedup. However, for data with a small dimension ($d<500$) and a small k ($k<64$), the implementation in [24] can be faster. In fact, for $n=1507328$, $d=294$ and $k=20$, our implementation is roughly $2.2\times$ slower. This is mainly because our k-NN algorithm is not as efficient for small ks. Our new k-NN algorithm to be published in a subsequent paper is much faster.

Figure 12 illustrates the scalability of the method w.r.t. the number of GPUs in the cluster. For this test we used a data set with $n=1966080$, $d=2048$ and $k=512$. As can be seen from the graph, we achieve a performance gain that is lightly below linear in the number of GPUs. Since each node contains 2 GPUs, we went from using 2 nodes to 6 nodes in the cluster.

To demonstrate the full capabilities of our implementations, we executed k-NNG construction for two datasets. The first data set contains two million images of simulated diffraction patterns of a randomly oriented Adenylate kinase (ADK) molecule. Each image has $126\times 126=15876$ pixels. The second dataset consists of twenty million images of simulated diffraction patterns of melting ADK in ten different molecular conformations. For more information about the structure of data sets, please refer to [28]. Unfortunately, the second data set on our CPU cluster would take approximately 3 months to compute, and therefore, we do not have actual timings.

We evaluated our method with a previous implementation of a neighborhood graph construction using MATLAB technical computing language. The MATLAB implementation took 56 hours on an exclusive CPU cluster with 32 nodes for two million images of diffraction patterns with the use of highly a optimized ATLAS-BLAS library [33] for multi-threaded Matrix-Matrix Multiplication in double precision. The cluster had one Xeon E5420 quad-core CPU per node with 16 kB of L1 cache, 6144 kB or L2 cache and 40 GFLOPS of double precision computing power. Comparison and selection was accomplished using the quick sort algorithm. Since the parallel MATLAB implementation did not take advantage of the symmetry of the distance matrix, one can assume that such an implementation would roughly take about 28 hours. Our GPU cluster had 16 nodes with each node equipped with two NVIDIA Tesla C2050 GPUs. Each Tesla C2050 GPU has a RAM of 3 GB with 506 GFLOPS of double precision computing power. There are 14 multi-processors sharing 720 kB of L2 cache and each multi-processor having 48 kB of user-configurable L1 cache/Share memory. In addition, each of the GPU nodes had two quad-core Xeon E5620. Note that in our GPU cluster, CPUs are used mostly for managing the GPUs and movement of data between nodes and not for computation. Our GPU cluster implementation took 4.23 hours, giving a roughly $6.6\times$ gain in performance.

To investigate the efficiency of our implementation we also benchmarked the most expensive part of the computation, i.e., matrix matrix multiplication, we tested both double and single precision matrix-matrix multiplication on a single GPU (Tesla C2050) vs. a multi-core optimized (with SSE instructions) on a Xeon processor. We achieved a roughly $7.7\times$ speed-up using GPUs just for matrix multiplication alone. As shown earlier, our complete implementation is slightly worse at a $6.6\times$ gain in performance. This slight degradation may be attributed to communication overhead while executing on the GPU cluster.

Discussion

We have successfully implemented a parallel brute-force k-NNG algorithm on a cluster of graphics processing units. We have used multiple levels of parallelism, task distribution, and data partitioning to achieve a roughly $6.6\times$ performance gain over an implementation using MATLAB on a comparable CPU cluster. There are several shortcomings in our implementation, as evidenced from comparing the raw float point processing power of the processors. The most expensive part of the brute force k-NNG is matrix multiplication. With the best tuned GPU libraries, we see that there is only 50% utilization of GPU resources as opposed to 90% use by finely tuned CPU libraries. A better GPU matrix multiplication library would go a long way toward addressing this deficiency. Another possible route to reducing the computational load is to recursively partition the input data set into sub-divisions with overlaps. This pre-processing step will reduce the computations (distance computation and selection) for each input vector based on set membership. While our current method from comparison and selection is faster than the state-of-the-art, it is clear that by grouping all rows in the distance matrix together and sorting, we are doing a significant amount of redundant work. For example, in sorting distances, there is no need to compare entries in a given row with entries in all other rows to get the smallest k values in the given row. We are currently working on an elegant solution based on quick-sort that will have a complexity of $O(n)$. With these improvements we will expect to gain a $30-40\%$ increase in overall performance.

Author Contributions

Conceived and designed the experiments: IK AD RMD. Performed the experiments: AD. Analyzed the data: RMD AD. Contributed reagents/materials/analysis tools: AD IK. Wrote the paper: AD RMD.

References

1. Roberts A, McMillan L, Wang W, Parker J, Rusyn I, et al. (2007) Inferring missing genotypes in large snp panels using fast nearest-neighbor searches over sliding windows. Bioinformatics 23: i401–i407.
2. Weston J, Elisseeff A, Zhou D, Leslie CS, Noble WS (2004) Protein ranking: from local to global structure in the protein similarity network. Proc Natl Acad Sci U S A 101: 6559–6563.
3. Zaki MJ, Ho CT, editors (2000) Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD, August 15, 1999, San Diego, CA, USA, revised papers, volume 1759 of *Lecture Notes in Computer Science*. Springer.
4. Maier M, Hein M, von Luxburg U (2009) Optimal construction of k-nearest-neighbor graphs for identifying noisy clusters. Theor Comput Sci 410: 1749–1764.
5. Liu W, He J, Chang SF (2010) Large graph construction for scalable semi-supervised learning. In: ICML. pp. 679–686.
6. Tenenbaum JB, Silva V, Langford JC (2000) A Global Geometric Framework for Nonlinear Dimensionality Reduction. Science 290: 2319–2323.
7. Fränti P, Virmajoki O, Hautamäki V (2006) Fast agglomerative clustering using a k-nearest neighbor graph. IEEE Trans Pattern Anal Mach Intell 28: 1875–1881.
8. Duda RO, Hart PE, Stork DG (2001) Pattern Classification. Wiley-Interscience, 2nd edition.
9. Jones PW, Osipov A, Rokhlin V (2011) Randomized approximate nearest neighbors algorithm. Proceedings of the National Academy of Sciences 108: 15679–15686.
10. Arya S, Mount DM, Netanyahu NS, Silverman R, Wu A (1994) An optimal algorithm for approximate nearest neighbor searching. In: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, SODA '94, pp. 573–582. Available: http://dl.acm.org/citation.cfm?id = 314464.314652.
11. Dasgupta S, Freund Y (2008) Random projection trees and low dimensional manifolds. In: Proceedings of the 40th annual ACM symposium on theory of computing. New York, NY, USA: ACM, STOC '08, pp. 537–546.
12. Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on computational geometry. New York, NY, USA: ACM, SCG '04, pp. 253–262.
13. Bentley JL (1980) Multidimensional divide-and-conquer. Commun ACM 23: 214–229.
14. Clarkson KL (1983) Fast algorithms for the all nearest neighbors problem. In: FOCS. pp. 226–232.
15. Vaidya PM (1989) An O(n log n) algorithm for the all-nearest-neighbors problem. Discrete & Computational Geometry 4: 101–115.
16. Paredes R, Chávez E, Figueroa K, Navarro G (2006) Practical construction of *k*-nearest neighbor graphs in metric spaces. In: WEA. pp. 85–97.
17. Chan TM (1998) Approximate nearest neighbor queries revisited. Discrete & Computational Geometry 20: 359–373.
18. Connor M, Kumar P (2010) Fast construction of k-nearest neighbor graphs for point clouds. IEEE Trans Vis Comput Graph 16: 599–608.
19. Chen J, Fang H, Saad Y (2009) Fast approximate *k*-nn graph construction for high dimensional data via recursive lanczos bisection. Journal of Machine Learning Research 10: 1989–2012.
20. Wang J, Wang J, Zeng G, Tu Z, Gan R, et al. (2012) Scalable k-nn graph construction for visual descriptors. In: CVPR. pp. 1106–1113.
21. Dong W, Moses C, Li K (2011) Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proceedings of the 20th international conference on World Wide Web. New York, NY, USA: ACM, WWW '11, pp. 577–586. doi:10.1145/1963405.1963487.
22. Indyk P (2004) Nearest neighbors in high-dimensional spaces. In: Goodman JE, O'Rourke J, editors, Handbook of Discrete and Computational Geometry, Boca Raton, FL: CRC Press LLC. 2nd edition.
23. Garcia V, Debreuve E, Nielsen F, Barlaud M (2010) K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In: ICIP. pp. 3757–3760.
24. Arefin AS, Riveros C, Berretta R, Moscato P (2012) Gpu-fs-*k*nn: A software tool for fast and scalable *k*nn computation using gpus. PLoS ONE 7: e44000.
25. Barrientos RJ, Gómez JI, Tenllado C, Prieto-Matias M, Marin M (2011) knn query processing in metric spaces using gpus. In: Euro-Par (1)'11. pp. 380–392.
26. Kato K, Hosino T (2009) Solving k-nearest vector problem on multiple graphics processors. CoRR abs/0906.0231.
27. Kuang Q, Zhao L (2009) A practical gpu based knn algorithm. In: Proceedings of the Second Symposium International Computer Science and Computational Technology (ISCSCT 09). pp. 151–155.
28. Schwander P, Fung R, Phillips GN, Ourmazd A (2010) Mapping the conformations of biological assemblies. New Journal of Physics 12: 035007+.
29. Pacheco PS (1996) Parallel programming with MPI. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
30. Chapman B, Jost G, Pas Rvd (2007) Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.
31. Sanders J, Kandrot E (2010) CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional. 1st edition.
32. nVidia (2012) CUBLAS Library User Guide. nVidia, v5.0 edition. Available: http://docs.nvidia.com/cublas/index.html.
33. Whaley RC, Dongarra JJ (1998) Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM). Washington, DC, USA: IEEE Computer Society, Supercomputing '98, pp. 1–27. Available: http://dl.acm.org/citation.cfm?id = 509058.509096.