August 2013

Efficient Computation of K-Nearest Neighbor Graphs for Large High-Dimensional Data Sets on GPU Clusters

Ali Dashti
University of Wisconsin-Milwaukee

EFFICIENT COMPUTATION OF K-NEAREST NEIGHBOR GRAPHS FOR LARGE HIGH-DIMENSIONAL DATA SETS ON GPU CLUSTERS

by

Ali Dashti

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Engineering

at

The University of Wisconsin-Milwaukee

August 2013

ABSTRACT
EFFICIENT COMPUTATION OF K-NEAREST NEIGHBOR GRAPHS FOR LARGE HIGH-DIMENSIONAL DATA SETS ON GPU CLUSTERS

by

Ali Dashti

The University of Wisconsin-Milwaukee, 2013
Under the Supervision of Professor Roshan M. D'Souza

The k-Nearest Neighbor Graph (*k*-NNG) and the related *k*-Nearest Neighbor (*k*-NN) methods have a wide variety of applications in areas such as bioinformatics, machine learning, data mining, clustering analysis, and pattern recognition. Our application of interest is manifold embedding. Due to the large dimensionality of the input data (< 15k), spatial subdivision based techniques such OBBs, *k*-d tree, BSP etc., are not viable. The only alternative is the brute-force search, which has two distinct parts. The first finds distances between individual vectors in the corpus based on a pre-defined metric. Given the distance matrix, the second step selects k nearest neighbors for each member of the query data set.

This thesis presents the development and implementation of a distributed exact k-Nearest

Neighbor Graph (k-NNG) construction method. The proposed method uses Graphics Processing Units (GPUs) and exploits multiple levels of parallelism for distributed computational systems using GPUs. It is scalable for different cluster sizes, with each compute node in the cluster containing multiple GPUs. The distance

computation is formulated as a basic matrix multiplication and reduction operation. The optimized CUBLAS matrix multiplication library is used for this purpose. Various distance metrics such as Euclidian, cosine, and Pearson are supported. For k-NNG construction, two different methods are presented. The first is based

on an approach called batch index sorting to build the k-NNG with three sorting operations. This method uses the optimized radix sort implementation in the Thrust library for GPU. The second is an efficient implementation using the latest GPU functionalities of a variant of the quick select algorithm. Overall, the batch index sorting based k-NNG method is approximately 13x faster than a distributed MATLAB implementation. The quick select algorithm itself has a 5x speedup over state-of-the art GPU methods. This has enabled the processing of k-NNG construction on a data set containing 20 million image vectors, each with dimension 15,000, as part of a manifold embedding technique for analyzing the conformations of biomolecules.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

The k-Nearest Neigbhor (k-NN) and the related k-Nearest Neighbor Graph (k-NNG) are important techniques used in a variety of fields such as path planning, bioinformatics, data mining, and geographic information systems. The k-NN search problem is rooted in the post-office problem first mentioned by Donald Knuth in The Art of Computer Programming [22]. The post-office problem is the task of finding the nearest post office(s) for each resident from a set of post offices in the area.

Formally, the (k-NN) problem is as follows: Given a set of reference data vectors $S = [\ v_1 \quad v_2 \quad ..v_k.. \quad v_n\ ]$, a distance metric $d : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ and a query vector $q \in \mathbb{R}^D$, the k-Nearest Neighbors to q are defined as the set $I(q) \subset S$ with

$$d(q, v_i) < d(q, v_j) | v_i \in I(q), v_j \in S - I(q)$$

As opposed to the k-NN search, in the k-NNG construction, every vector in the reference data set is also a query vector. For data sets with small dimensions, there are several efficient space partitioning- based techniques. These techniques use the reference data set to build query data structures such as kd-trees [19], BBD-trees [3] , random-projection trees (rp-trees) [10] or hashing based partitioning, such as locally sensitive hash [11]. These techniques are optimized for searches, in the sense that, when a query vector is produced, the k-NN can be quickly found. It is assumed that the reference data set is immutable. The k-NNG can be constructed by repeatedly invoking the k-NN on each member of the data set. Direct k-NNG construction has also been extensively investigated. All these methods have a computational complexity that is exponentially dependent on the vector dimension. For data sets consisting of high-dimensional vectors ($D > 100$), these techniques are computationally intractable. There are several approximate algorithms that are efficient in the regime of high-dimensional data sets. These typically reduce search space through recursive partitioning based on an accuracy measure, and then employ a brute force technique on the reduced data sets. For applications such as Manifold Embedding, accuracy is very important and therefore these approximate techniques are not viable. Consequently, the only alternative available is the brute-force technique, which has two important parts. The first consists of finding the distance matrix based on a pre-defined distance metric. The second part consists of using the distance matrix to find the nearest neighbors in the reference data set for each query. The computational cost of executing the brute-force search is substantial and cannot be handled by serial computing on single

desktop computers. In this thesis, the computational complexity is addressed through multi-level parallel execution on computing clusters with nodes containing graphics processing unit accelerators.

1.2 Contributions

In this thesis, a distributed multi-level parallel brute force k-NNG algorithm is presented. Efficient data partitioning and communication schemes were developed to partition the large input data set among individual nodes of a computing cluster for balanced execution. Brute force k-NNG consists of two main tasks: distance matrix calculation and k-NNG construction. Various distance metrics were formulated in terms of matrix multiplication and reduction operations. Efficient GPU libraries were used for computing distance matrices and two novel algorithms have been developed for k-NNG construction. The first algorithm, called batch index sorting, uses three sorting steps to build the k-NNG. The second algorithm is an efficient implementation of the Quick-Select algorithm and uses the latest GPU functionality to greatly speed up k-NNG construction. Overall, this work achieved a 6x speed up over an existing distributed multi-core parallel implementation of k-NNG implementation on a CPU cluster.

1.3 Manifold Embedding

Our target application of a k-NNG construction problem is manifold embedding. Manifold embedding is used in structural biology to recover the structure and conformations of a biomolecule from a large ensemble of noisy snapshots obtained from unknown orientations and conformations of the molecule [31] [5]. There are different methods for three-dimensional

reconstruction of an object from its two-dimensional snapshots. In the simplest case, when the object and orientations are known, standard tomographic methods [25] can be used to reconstruct the 3D view of object. Tomographic methods can be used even for unknown orientations and extremely noisy signals, and have been proposed for structure and recovery in heterogeneous datasets [31][14].

In addition to tomographic approaches, graph theoretic and differential geometric methods can be used for image reconstruction from datasets of unknown objects [15][13]. Graph theoretic approaches attempt to reduce the dimensionality of the data space by projecting into some low-dimensional manifold intrinsic to the geometric projection of the data, and hence are seen as nonlinear dimensionality reduction kernels. This is in contrast to to linear methods, such as principal component analysis (PCA)which essentially ignore the intrinsic geometry inherent to the data. However, there are some disadvantages associated with graph theoretic methods,such as computational costs and robustness against noise, to name a few.

The underlying assumption behind manifold embedding exploit the fact that the similarities between high-dimensional data points in data space reveal themselves as a low-dimensional hypersurface embedded in the high-dimensional data space. The embedded hyper surface, called a manifold from here on, contains the information about the system giving rise to the snapshots, for example, 3D structure of a biomolecule.

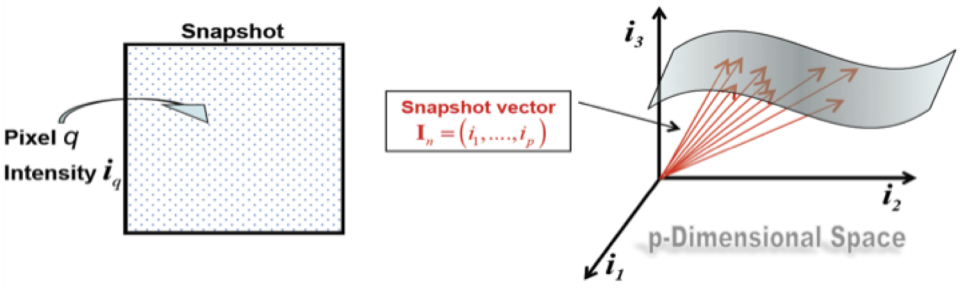


Figure 1.1 Schematic view of a manifold. Correlations in high dimensional space reveal themselves as a low-dimensional hypersurface (manifold) in data space. [5]

Manifold embedding measures the similarities or nonlinear correlations between clouds of snapshots by tracking their response to an operator and finding the orthonormal coordinates needed to describe the manifold. In our target application, we used the diffusion map operator, a technique inspired by hidden Markov models (HMM), for embedding. Another interesting feature of manifolds is their dimensionality. Dimensionality of a manifold has a direct relationship with the degrees of freedom the object has in reality. For example, since rotation of an object in space has three degrees of freedom and can be characterized by three values (for example, Euler angles), the manifold of snapshots form an object which rotates about any axis in a $3-$dimensional space (Figure 1.1).

Diffusion map manifold embedding is based on the so-called diffusion characteristics of the dataset. Specifically, it measures the distance between two points according to their Euclidean distance, but the path taken by the diffusion of heat. An affinity graph is defined as the exponential normalization of Euclidean distance between points with their neighbors in the k-nearest neighbor graph. Probability of diffusion from one point to another on a path in a

manifold is then defined by applying Laplace-Beltrami normalization to the affinity graph. A diffusion probability matrix contains all the probabilities of diffusion from one point to another on the manifold. Finally, eigenvectors of the diffusion probability matrix provide a description of the manifold in terms of a set of Euclidean coordinate (eigenvectors).

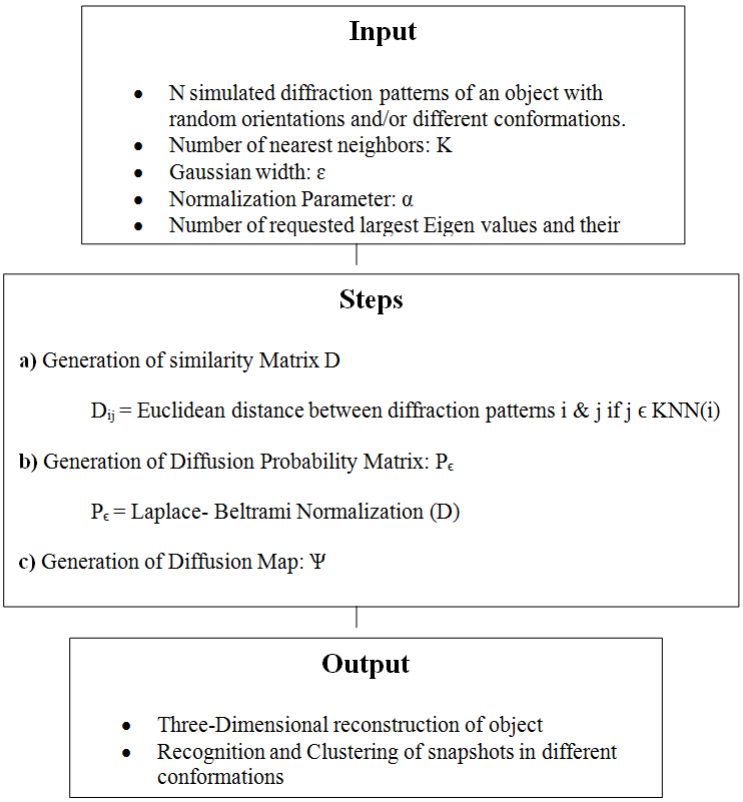


Figure 1.2 Flowchart of diffusion map embedding of manifold

Diffusion map embedding is a computationally efficient theoretical framework that contains the required information to reconstruct a 3D model of a rotating object from its scattering snapshots in random orientations on a 2D detector [31] [32]. The underlying information in a manifold captured in its Riemannian metric contains an object independent term and an object

specific signature [5].

Figure 1.2 shows a flowchart of a diffusion map embedding method. The main computational part of diffusion map embedding is the construction of a k-NNG for a high-dimensional data cloud that will be analyzed to generate a diffusion map in subsequent stages of the algorithm. The input data to the manifold embedding application contains upwards of 10^7 images, with each with 15k pixels. We require $k > 200$ for accurate results. The nature of high-dimensional data, in addition to accuracy requirements, makes a brute-force algorithm the only viable option for generating k-NNG.

1.4 Disrtributed parallel computing with GPU accelerators

The k-NNG implementation developed in this thesis uses multi-level parallel execution on computing clusters with GPU accelerators. In this section the various tools, techniques and programming models used in this implementation are described.

1.4.1 Message Passing Interface (MPI)

Message passing interface (MPI) is the most widely used application protocol interface (API) for distributed memory parallel execution on computing clusters. It is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation [28]. It provides language binding for C, C++ and FORTRAN, and currently is the de facto standard interface for high performance and high

throughput computing applications on distributed memory architecture. In the MPI programming model one node in the computing cluster is designated as the head node and controls the overall execution. Other nodes are designated as worker nodes and are responsible for the distributed execution. MPI provides point-to-point message passing for user-specified groups of exclusive processes. In programming with MPI, worker nodes access each others' data through the high-speed network connecting the cluster. In addition, there is an option for each node to broadcast its data to all other worker nodes in the cluster.

1.4.2 Open Multiprocessing (OpenMP)

In contrast to MPI, OpenMP is a shared memory architecture API. OpenMP enables the harnessing of multiple cores on modern CPUs through multi-threading. OpenMP follows a fork-join programming model, where a parent task/process can spawn multiple tasks that can execute in parallel. Parallel tasks can share data and synchronize. Tasks can execute entirely different programs on different data types. In the OpenMP task execution model each task can have independent access to memory and cache. OpenMP supports bindings for popular programming languages such as C, C++ and FORTRAN. In a cluster setup, MPI can be used for assigning the tasks to worker nodes and multithreaded task executions inside the nodes can be achieved with the help of OpenMP.

1.4.3 Graphics processing units (GPUs)

Graphics Processing Units were initially developed to handle computations related to graphics rendering. The need for specialized rendering routines led GPU vendors to provide user-defined functionality through shader programming [24]. Subsequently, researchers used shaders to essentially trick GPUs into performing scientific computations. This further led GPU vendors to develop extensions of the C language in order to directly access the parallel processing power for general purpose scientific computing. Examples of these APIs include CUDA [30], OpenCL,[21] and OpenGL [36].

1.4.3.1 Hardware architecture

The processing elements in a GPU are organized around several multi-processors (MPs). Each multi-processor, as the name suggests, has several serial processing units (SPUs). All SPUs in an MP have access on-chip to a user-controlled cache called shared memory. In addition, there is a register file that is distributed among all MPs. Shared memory is organized into memory banks. The off−chip RAM on a GPU is called global memory. While previous generation GPUs did not cache global memory, the latest generation GPUs have L1 and L2 caches. L1 cache is local to an MP while the L2 cache is shared among all MPs (Figure 1.3. A small portion of the global memory is designated as read-only constant memory. This memory that is automatically cached can be used to store constant values that are frequently accessed.

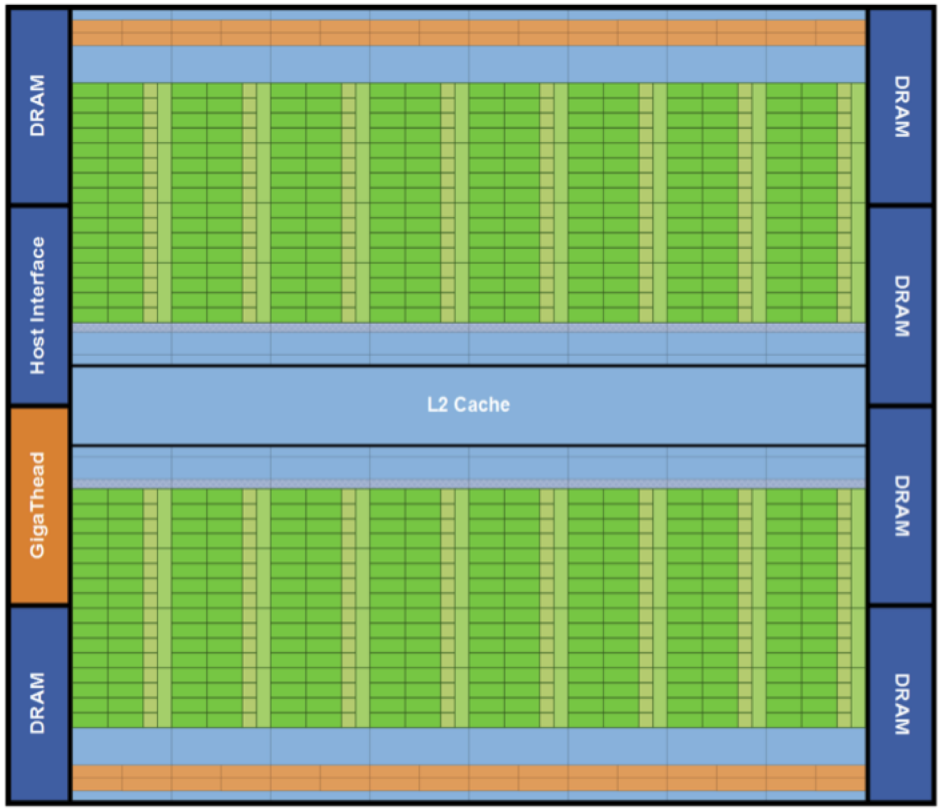


Figure 1.3 GPU Fermi architecture. 16 multiprocessors are positioned around a common L2 cache. Each orange portion is scheduler and dispatch, green portions are execution units and light blue portions are memory for register file and L1 cache [30]

.

1.4.4 Programming Model

GPUs generally follow the data-parallel programming model, where the same instruction is applied to elements of a data array. For GPUs developed by NVIDIA, there is a native API called Compute Unified Device Architecture (CUDA), which dramatically decreases the programming overhead involved in accessing the parallel computing power of GPUs.

The basic execution unit is a thread. Every thread executes the same program called a kernel. Threads are organized into logical partitions called thread blocks (TBs). During execution, all threads in a TB are assigned to a single MP. Threads in a TB can communicate via shared memory and can be synchronized. In a typical execution, the number of TBs far exceeds the number of MPs (Figure 1.4).

At the hardware level, threads are organized into warps. All threads in a warp execute in lock-step fashion. Warps are equivalent to threads in the symmetric multi-process context. It is therefore advisable to avoid thread divergence with a warp. If two or more threads in a warp access the same shared memory bank, this operation will cause bank conflicts and serialization. However, if all threads access a single shared memory location, then an efficient broadcast mechanism is used. In accessing global memory, all threads in a warp must access memory within a contiguous 128b segment. Non-compliant memory accesses are called un-coalesced accesses and are serialized.

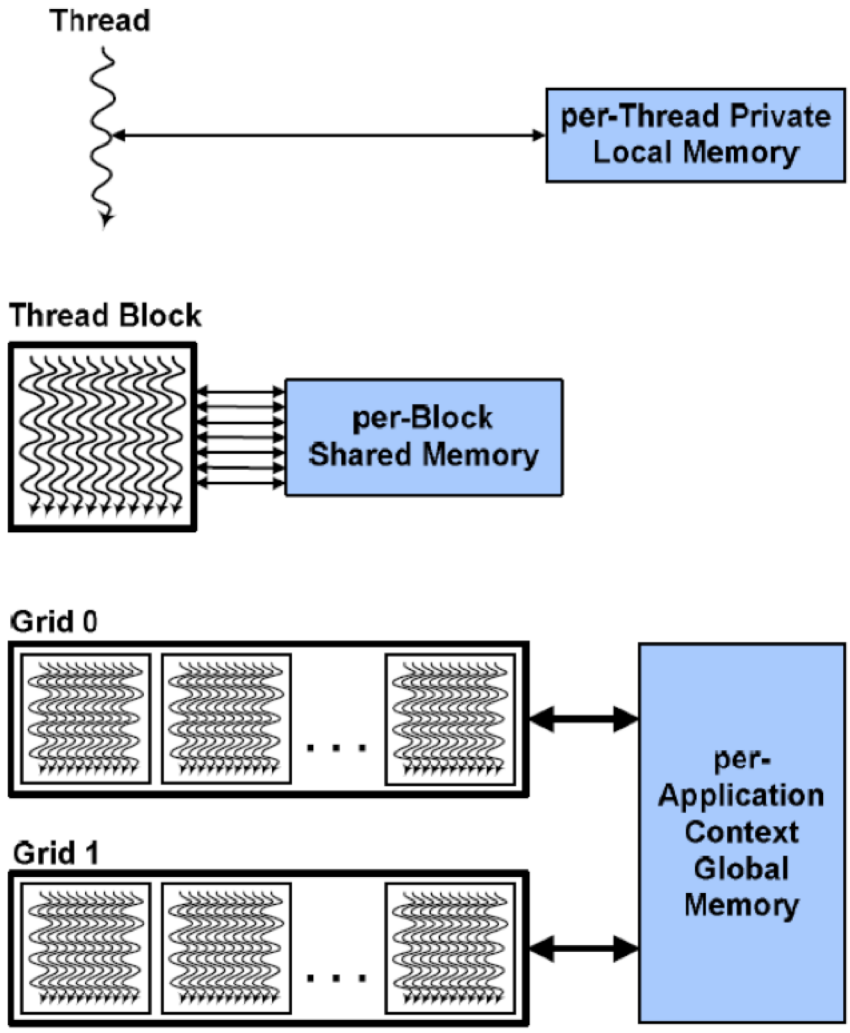


Figure 1.4 CUDA Model. Threads, blocks, and grids, with corresponding memory spaces for private per-threads, shared per-block, and global per-applications.[27]

Chapter 2

Literature Review

In this chapter, we discuss previous work on algorithms for k-nearest neighbor graph construction and k-nearest neighbor search. Broadly speaking, there are two categories of algorithms. One set of algorithms provides the exact k-NNs while the second set of algorithms sacrifices accuracy to a certain degree to gain on performance. For low dimensional data sets, there are several efficient algorithms based on space partitioning data structures. For intermediate dimensional data sets, there are approximate algorithms based on hashing[11]. For large dimensional data sets where exact computation of k-NN is a must, the only alternative is a brute force search that is computationally quite expensive. Recently, there has been much research into addressing the computational complexity of a brute force search by using novel architectures such as graphics processing units.

2.1 Exact k-NN/k-NNG algorithms

In very low dimensional spaces (2D or 3D) graph-based searching methods such as those using Voronoi diagrams and proximity graphs are very efficient and achieve super-linear speedup. For higher dimensions, space partitioning methods such as k-D Trees[19] , B-Trees [3], R-Trees[10] and Metric-Trees are most efficient. For these methods, there is a pre-processing

step where the reference data set is used to build the search data structure. Once this data structure is built, the search for k-NN is quite cheap. However, building the search data structure itself is quite expensive . Therefore these methods are most useful if the reference data set is static. For large data dimensions and large values of k, the search process is no better than the brute-force technique.

2.2 Approximate algorithms

In many applications, especially with large data dimensions, search accuracy constraints are not very stringent. In such cases, methods have been developed based on hashing to bin objects based on the object hash that is a function of its vector coordinates. Popular hashing schemes include the Locality Sensitive Hashing (LSH)[11], Z-Morton Curve based hashing [9], and Hilbert space filling curve hashing [4] . Essentially, hashing schemes convert a D dimensional problem to a 1-D problem through a hash function that preserves proximity of the input data set. The most challenging part of this scheme is the definition of a good hashing function. There are other approximate algorithms that use a hybrid approach. For example,

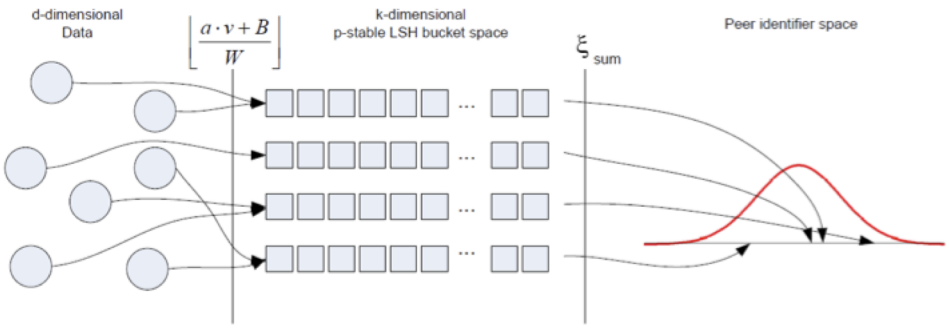


Figure 2.1 Schematic view of locality sensitive hashing methods. Two level hashing from d-dimensional data space to peer identifier space. [17]

in [9], a disk−based quadtree data structure is initially used to partition the reference data set. The k-NN search is conducted in two phases. A Z-order-based approximate proximity measure is used to find the approximate k-NN. Next, a recursive correction algorithm is used to improve the accuracy. Another set of techniques is based on a hybrid of spatial subdivision up to a threshold granularity and small scale brute force evaluation or heuristics for refinement [8] [12] [35]. Some techniques take advantage of the intrinsic dimensionality of the data set to project the data set into a low dimensional space that preserves proximity. These techniques use the results of the Johnson-Lindenstrauss theorem that says for any n point subset of Euclidean space can be embedded in $k = O(\log(n\,\epsilon^2)$ dimensions without distorting the distances between any pair of points by more than a factor of $1 + \epsilon$ for any $0 < \epsilon < 1$. This reduces the complexity of the search. Examples of such work include techniques using random projections as in [29].

2.3 Brute force k-NN search

For practical purposes, when one insists on having linear or near linear space requirements, the best performance time per query for a random input point cloud is bounded to $\min(2O(d), d \times n)$, which is essentially equal to a brute force search. In other words, the complexity of algorithms is linearly related to the dimension and data number, even for moderate dimensions. Exponential dependence of time or space on the dimension in a k-NN search is termed the curse of dimensionality and has been observed in practical experiments. Many well known structures exhibit a linear time search with linear or near linear storage even with

moderate dimensions (10-20). 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 [18].

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. That results in an $M \times N$ distance matrix, where M is the number of query points and N is 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. Fortunately, due to their simplicity, brute force methods are highly parallelizable and can be processed by computational clusters, clouds and high throughput parallel processors such as Graphical Processing Units (GPU).

2.3.1 Brute force implementation of k-NNG on GPU

Recently, there have been several methods that accelerate brute force k-NN and k-NNGs on graphics processing units. The complexity of the brute force algorithm brings the need for the development of algorithms and implementation on massive parallel processors such as GPUs. There are some GPU implementations of the brute force algorithm both for k-NN search and k-NNG construction. Garcia et al. in [16] proposed an algorithm for a k-NN search by devising two kernels for distance calculation based on cuBLAS, [34] an optimized matrix multiplication library on the GPU, and a comparison kernel based on a parallel insertion sort, and achieved a 100-fold increase in speed compared with the approximate nearest neighbor

(C++ ANN library) [16] . This method works on multiple queries simultaneously, with each thread handling a single query. If k is small, then the data structure for an insertion sort can be stored in a fast on-chip shared memory and this method can be quite efficient. For a large k the insertion sort data structure spills into the main memory and causes a dramatic loss of efficiency because of uncoalesced memory transactions. In fact, for a large k, the selection is much slower than a simple sort operation.

In [20] a multi-GPU brute force k-NNG algorithm is described. Data are partitioned and distributed among 3 GPUs on a single computing cluster node. They use symmetry of the distance matrix to compute only half the entries. For each data partition called a grid. In the second phase, a heap-based selection is employed. Each row of the distance matrix is processed by a thread block. There is a per thread block heap that stores the k smallest/largest elements. Each thread maintains a local buffer that stores the thread elements that are smaller/larger than the smallest/largest element in the heap. There is a thread synchronization step where each thread successively pushes elements in its buffer into the thread block heap. This last step is completely serialized and has a significant amount of uncoalesced memory access patterns especially if k is large, which necessitates that the heap be stored in global memory. As a final step all the heaps in all GPUs are sent to the CPU for merging to get the global k-NNG. This gains a serial step with significant memory transfer from GPU to CPU memory.

Work presented in [20] uses a slightly different approach to finding k-NN given the distance matrix. Here a single thread block handles each row of the distance matrix. Each thread stores

a heap of k elements that records the local k-NN. Each thread strides through the given row of the distance matrix in a coalesced manner to find the local k-NN. At the end of this process, all threads in the block have their own heaps. In the next step, a single warp is used to build 32 k-NN heaps, one for each thread. In the final step, the first thread of the first warp reduces the 32 k-NN heaps to get the final k-NN. The second and third steps of this algorithm lose a large amount of parallelism and therefore underutilize GPU resources. Finally, this method works well only if k is small such that the thread heaps can be stored in an on-chip shared memory. Otherwise, just like the work in [16], the heap is stored in global memory that necessitates un-coalesced global memory reads.

In [23] a radix sort-based approach is used to select the k nearest neighbors. The authors 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) A closer examination shows that the approach process each row of the distance matrix in a separate sort. For an n that fits into GPU memory, this process underuses GPU resources.

In another implementation of a brute force k-NN search algorithm, [33] proposed the truncated bitonic sort (TBiS), a selection method based on the Biotonic sort. One of the main characterizations of the proposed method is having a low synchronization cost achieved by using synchronous memory operations. The TBiSort uses recursion to break down input arrays

all the way to the base level and then goes upwards, merging the values. In the merging step, minimum and maximum values of each element pairs of two lists are detected and assigned to two minimum and maximum monotonic sublists. Having truncation, only the k elements of the minimum sublist are gathered in each step and the maximum sublist is set free in each step upwards. Truncation starts and continues from a minimum sublist with k elements. While this method performs significantly faster than a plain sort and selects for small values of k and n, it rapidly loses efficiency as k and n grow. While it is 16x faster than a plain radix sort and selects for $k = 2$ and $n = 2^{17}$, for $k = 2^8$ and $n = 2^{20}$ it is no better than a plain radix sort and select.

Chapter 3

Methods

In this chapter, an expandable method for the construction of nearest neighbor graphs for very large high-dimensional data clouds is presented using the brute force method. The method presented in this thesis relies on three levels of parallelism, is designed to execute on computing clusters with GPU accelerators in the nodes, and addresses the computational complexity of the brute force method. The three levels of parallelism are: distributed memory parallelism between the nodes of the cluster; shared memory parallelism between the cores of the CPUs in the node; and finally, data-parallelism within the GPU accelerators in each node. The main contributions of this thesis are the following: (a) A novel scheme for data partitioning and management for load balancing between the nodes of the cluster and efficient communication. (b) Two algorithms for selecting k-NN on GPUs. The first is based on indexed sorting and relies on the optimized radix sort algorithm available in the Thrust library [6]. The second is an efficient implementation of the quick-select algorithm [6] by using latest GPU functionalities. To our knowledge, the last algorithm is the fastest multi-query k-NN select algorithm to date.

3.1 Problem scale

The implementation performed as part of this thesis is a part of a manifold embedding pipeline for recovering the structure and conformation of biomolecules using a large database of high-noise images obtained through various imaging techniques. A typical data set consists of $n = 10^7$ images with dimensions on the order of $D = 10^4$. The process requires exact k-NN results and therefore approximate methods cannot be used. Furthermore, the dimensions of the input vectors render other techniques that are efficient for low-dimensional data (sub-linear growth), essentially intractable. The only option is brute force. The sheer size of the data means that the input data are on the order of 3.6TB in single precision. Furthermore, in the brute force method, if one were to compute the entire distance matrix, it would require 36000 TB. This kind of memory capacity is beyond the capability of any kind of cluster computer. The input data itself can only be stored in the head node of the average computing cluster. Therefore, processing this massive data set requires clever partitioning and interleaving of distance calculation and k-NN selection and merging of results.

3.2 Distributed k-NNG generation on GPU clusters

Brute force methods have two primary tasks, namely, generation of the distance matrix between input vectors, and selection of k-NNs. Both these tasks are computationally expensive. Due to the massive input data size and the even more massive intermediate results (distance matrix), we use a distributed approach with interleaving of distance calculation and k-NN selection along with merging of results. In this section, we begin by describing the distance

calculation. Next, we describe our data partitioning approach to handle the massive memory footprint as well as to balance the computation. Finally, we describe the two new algorithms for k-NN selection.

3.2.1 Distance calculation

In this thesis we focus on calculating the Euclidian distance. A similar approach can be taken for other distance metrics such as Cosine or Pearson distance. Given two D dimensional vectors v_i, v_j, the Euclidian distance is given by:

$$d(v_i, v_j) = sqrt\|v_i - v_j\|^2 \quad [3.1]$$

For k-NN selection, we may use the squared distance instead of the plain distance. The square of the distance metric can be written as:

$$d^2(v_i, v_j) = \|v_i - v_j\|^2 \quad [3.2]$$

$$= (v_i - v_j)^T (v_i - v_j) \quad [3.3]$$

$$= v_i^T v_i + v_j^T v_j - 2v_i^T v_j \quad [3.4]$$

$$= \|v_i\|^2 + \|v_j\|^2 - 2v_i^T v_j \quad [3.5]$$

Now consider a set $V = [\ v_1 \quad v_2 \quad .. \quad .. \quad v_n\ ]$. Further consider a squared distance matrix that contains the mutual distance between all vectors in V

$$S = \begin{pmatrix} d^2(v_1, v_1) & d^2(v_1, v_2) & .. & d(v_1, v_N) \\ d^2(v_2, v_1) & d^2(v_2, v_2) & .. & d(v_2, v_N) \\ ... & .. & .. & .. \\ d^2(v_N, v_1) & d^2(v_N, v_2) & .. & d(v_N, v_N) \end{pmatrix} \quad [3.6]$$

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

$$A = \begin{pmatrix} v_1^T \\ v_2^T \\ .. \\ v_N^T \end{pmatrix} \quad [3.7]$$

$$B = \begin{pmatrix} \|v_1\|^2 & \|v_1\|^2 & .. & \|v_1\|^2 \\ \|v_2\|^2 & \|v_2\|^2 & .. & \|v_2\|^2 \\ .. & .. & .. & .. \\ \|v_N\|^2 & \|v_N\|^2 & .. & \|v_N\|^2 \end{pmatrix} \quad [3.8]$$

We can now write the following equation

$$S = B + B^T - 2\left(AA^T\right) \quad [3.9]$$

Therefore, the calculation of the squared distance matrix can be formulated in terms of vector reductions, vector additions, and dense matrix multiplication. All of these are BLAS routines and have very efficient libraries on GPUs. Note that the S is symmetric and therefore, it is enough to compute only elements $S_{(i,j)}\ |i \geq j$. Finding the set of the 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. To handle the large data size, our approach is to compute the k nearest neighbors in parts. As illustrated in Figure 3.1, the computation of the squared distance matrix S is split into $P \times P$ partitions. Consequently, each portion $S_{(I,J)}$ is computed as:

$$S_{(I,J)} = B_I + B_J^T - 2\left(A_I A_J^T\right)$$

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

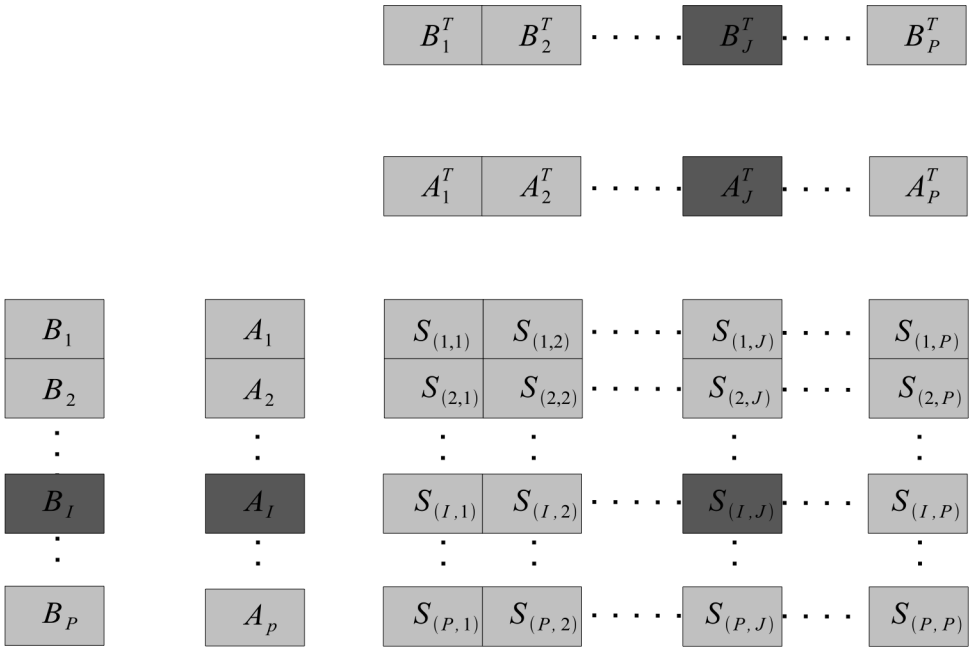


Figure 3.1 Data partitioning

3.2.2 Distribution of data and tasks between computing nodes

Computing clusters typically have several nodes connected by high-speed interconnects. One of the nodes is designated as the head node, which 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 by all nodes through parallel (I/O) that typically holds input data and results. The worker nodes, with smaller local disk space, copy input data from the shared disk as required .

For load balancing, we distribute computing of the partitions of S in a block cyclic manner. This means that node q computes all the partitions $S_{I,J}\,|J : J\%Q = q$. Now consider the case where the I^{th} block row of S is being processed. Any block $S_{I,J}$ 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 $B_J, A_J\,|J : J\%Q = q$. The disk space on the nodes restricts the number of partitions of A that can be saved locally. Therefore, in our setup, the vector data A is uploaded on the shared disk and divided into $A_1, A_2....A_P$ partitions. The partition A_I is read in parallel from the shared disk on the head node while partitions $A_J\,|J : J\%Q = q$ are stored locally on node q. A_I is then read in parallel by all nodes from the shared disk and then the portions are shared by using an asynchronous 'all gather' operation to build an image of A_I in each node′s RAM. Figure 3.2 illustrates this process. We use message passing interface (MPI) [28] to distribute the computational tasks as well as to communicate data between various nodes in the cluster.

We do not actually build the matrix B. Instead, the vector $\hat{B} = \{\|v_1\|^2, \|v_2\|^2, ...\|v_N\|^2\}^T$ is computed in advance and stored in the RAM of each node. Even for $N = 10^7$ the size of $\hat{B}$ ($\sim$ 10MB) is quite small compared with 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 locally. It then broadcasts the results to all other nodes.

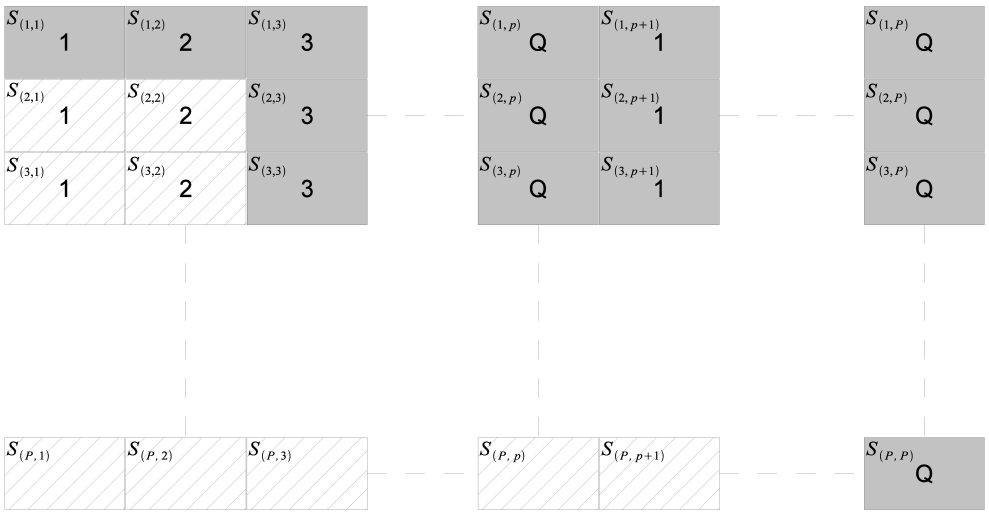


Figure 3.2 Load balancing

Once the partition $S_{(I,J)}$ is computed, the local $k-NN$s with respect to both the rows (k_R-NNs) and columns (k_C-NNs) are computed. Since the matrix S is symmetric, the local k_C-NNs w.r.t. partition $S_{(I,J)}$ are identical to the local k_R-NNs w.r.t. partition $S_{(J,I)}$. Therefore, each node q maintains one heap per 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 3.3, node 4 maintains one heap for each of the columns $4, Q+4, ...(P-Q+4)$. The heap for column 4 will contain the merged local 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 $S_{(1,Q+4)}, S_{(2,Q+4)}, ..., S_{(Q+4,Q+4)}$.

The node q is used to compute the global $k-NN$s for all vectors in $A_I\,|I\%Q=q$. The global $k-NN$s for all the vectors in A_I are generated by merging the local k_R-NNs w.r.t. partitions $S_{(I,J)}\ J=1,2,....,P$. However, the merged results of the local k_R-NNs w.r.t all

partitions $S_{(I,J)}$ $|J = 1, 2, ..., I$ are already available in node q from the local $k_R - NN$s computed previously. The local $k_R - NN$s w.r.t. all partitions $S_{(I,J)}$ $J = I + 1, I + 2, ..., P$ are cooperatively computed by different nodes. Each node maintains a heap to merge the results of finding the local $k_R - NN$s of the partitions that it processes. At the end, the merged results are communicated to the node processing the global $k - NN$s for the block row I for merging at the global level. For example, as illustrated in Figure 3.3 for $I = 4$, node 3 will compute local $k_R - NN$s w.r.t. all partitions $S_{(4,Q+3)}, S_{(4,2Q+3)}, ..S_{(4,Q-P+3)}$. 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 for computing the global $k - NN$s for all vectors in A_4.

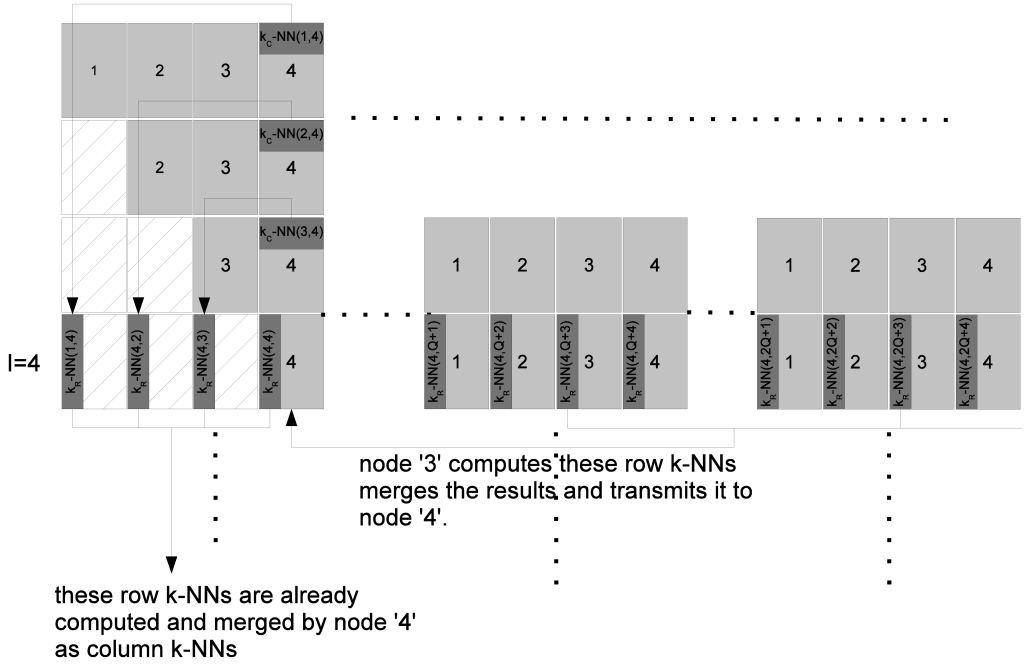


Figure 3.3 Computing k-NN

3.2.3 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 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. R is governed by the available GPU RAM. While reading A_I is quite fast (it is parallelized), reading A_J from the local disk is slow. We therefore hide this latency by reading the disk in parallel with computation.

Each GPU is given a partition of A_I denoted by $A_I(m)\,|m = 1, 2, ..., M$. A partition of the file A_J denoted by $A_J(r)|\ r = 1, 2..., R$ is read by all GPUs. 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$\hat{B}$ in its RAM. $\hat{B}$ 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 $k - NN$s as well as the row $k - NN$s are computed by using sorting. The column $k - NN$s are written back to CPU memory while the row $k - NN$s are kept on global memory to be merged. 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 $k - NN$s in the GPU memory without writing back to the CPU RAM. However, $S_{(I,J)}(m, r)\ m = 1, 2, ..M$ are being processed by different GPUs; therefore, the

column $k - NN$s are generated by different GPUs. The accumulated column $k - NN$s are then merged by one GPU per column. The final result of this operation is the local row and column $k - NN$s for the partition $S_{(I,J)}$. Figure 3.4 illustrates this process.

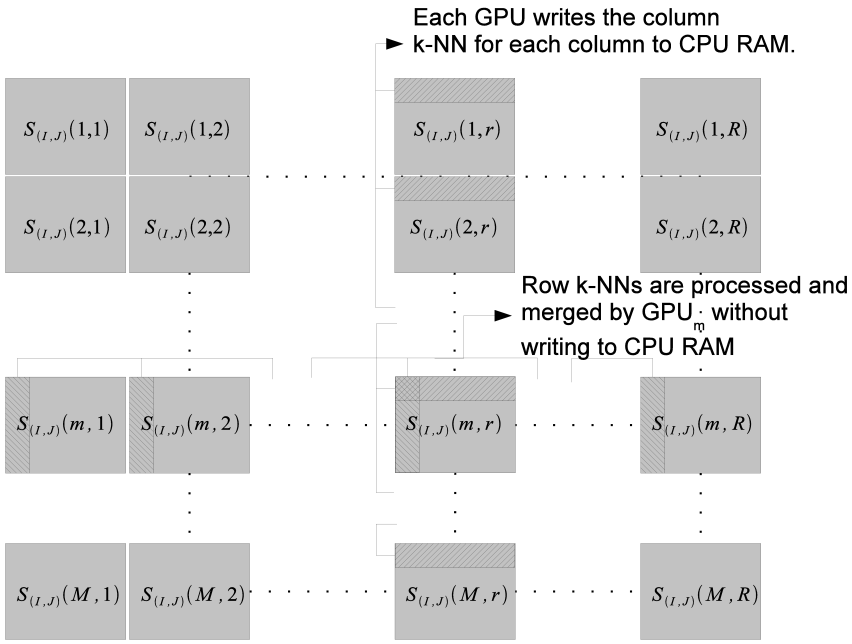


Figure 3.4 Task distribution within the nodes

We use OpenMP multi-threading [1]. 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.

3.2.4 Distribution of tasks and data within GPUs

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

- Finding vector $\hat{B}$ 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 $k - NN$s based on $S_{(I,J)}(m,r)$

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 [27] to code our kernels.

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 [26], using it will be inefficient since the vectors are relatively large in number (N=$10^6 - 10^7$) with a 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 the underuse of GPU resources by computing multiple vector norms in one kernel invocation. 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 by using atomic-add. Figure 3.5 illustrates this process.

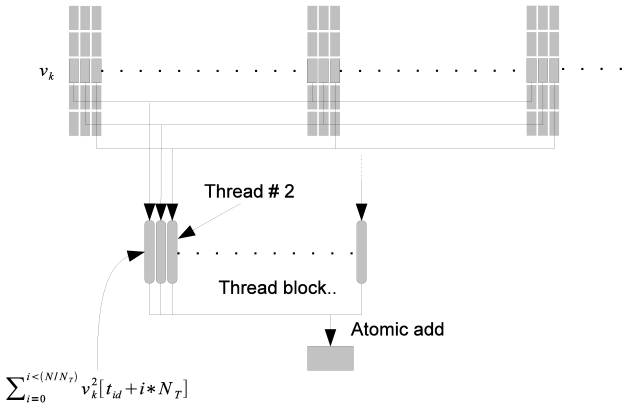


Figure 3.5 Vector norms

For dense matrix multiplication $A_I(m)A_J^T(m)$, we use the optimized library function from CUBLAS [26]. 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. 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 (Figure 3.6 (a)). 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 (Figure 3.6(b)). In reading an element $\hat{B}_I(m)$ by all threads in the thread block, we use the broadcast mechanism in GPUs.

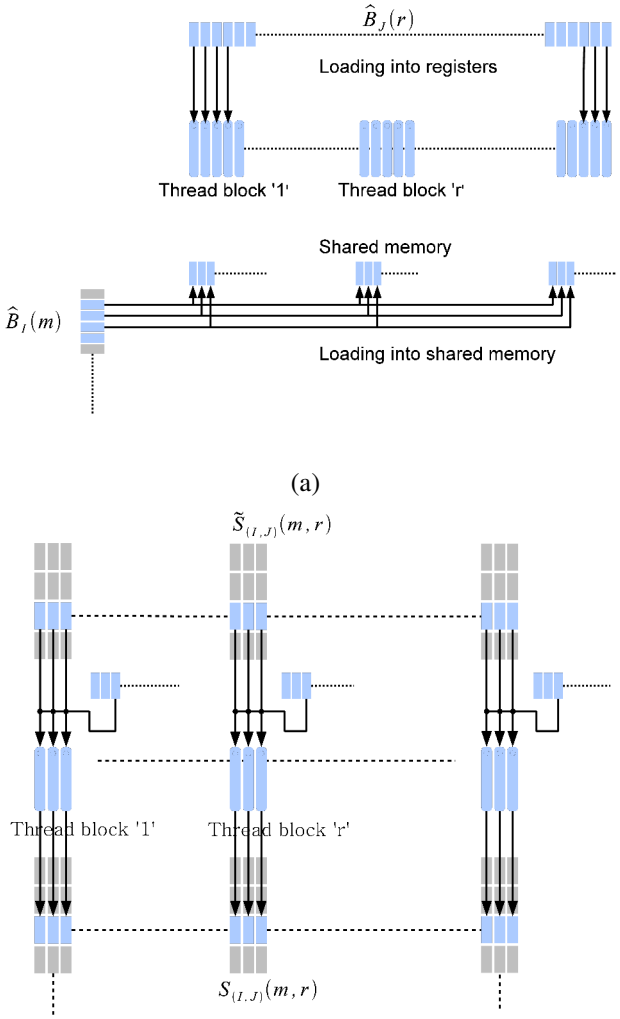


Figure 3.6 Addition kernel to find distance martix S

3.3 Finding k-NN using batch index sorting

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 the radix sort with the elements of $S_{(I,J)}(m,r)$ as key. In the next two steps, we execute an order preserving the sort results of the previous step with the column index as the key. Separately, we also execute an order preserving the 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 $k-NN$s for each row and column w.r.t. $S_{(I,J)}(m,r)$. Figure 3.7 illustrates this process.

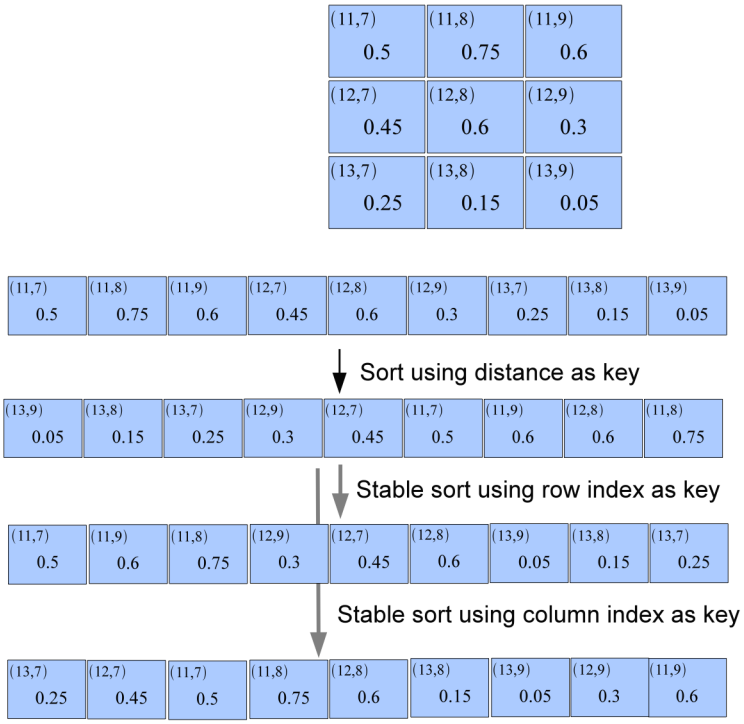


Figure 3.7 Batch index sort for finding k-NN

3.4 Quick select algorithm

The quick select algorithm is a variant of the quick sort algorithm. Given an array, just as in the quick sort algorithm, a random element is selected as the pivot. Next, the array is re-arranged with the elements less than the pivot being moved to the left of the pivot and the elements greater than the pivot being moved to the right side. This process is recursive on the right partition and the left partition until the partition size reaches two elements and these elements can be swapped. In the quick select algorithm, as soon as the partition is finished, the sizes of the left and right partitions, L and R respectively, are found. If $L > k$, then the right partition is discarded and the left partition is further recursively processed. If $L < k$, then the left partition is kept. Next we set $k = k - L$. Then the right partition is processed recursively. This algorithm has complexity $O(n) + k\log(k)$ for a sorted k-NN list and $O(n)$ for the unsorted k-NN list.

Our approach operates on multiple arrays simultaneously. Each array is handled by a single thread warp. Threads in a warp are executed simultaneously on a single multi-processor and therefore are synchronized by default. Partitioning of an array is done incrementally in 32-element wide segments. We use shared memory to ensure coalesced memory writes of the results of partitioning into the auxiliary array in global memory. Our approach uses the warp voting function `_ballot(`p`)` to partition the input without reading the input array twice and without executing the parallel-prefix sum. The ballot function `_ballot(`p`)` fills a 32-bit unsigned integer, one bit per thread in the warp, based on the evaluation of the predicate p.

Each thread in the warp first reads in an element into its register from global memory. Elements are then written to a 32-element wide shared memory array with elements greater than equal to the pivot being written from the right end and elements less than the pivot being written from the left end. To do this, each thread needs to know where in shared memory to off load its element. We execute the warp voting function based on a predicate that checks whether the element in the register is greater than or equal to the pivot or smaller than the pivot. Threads that have an element greater than or equal to the pivot set the corresponding bit to '1' and to '0' if the element is less than the pivot. This process is illustrated in Figure 3.8.

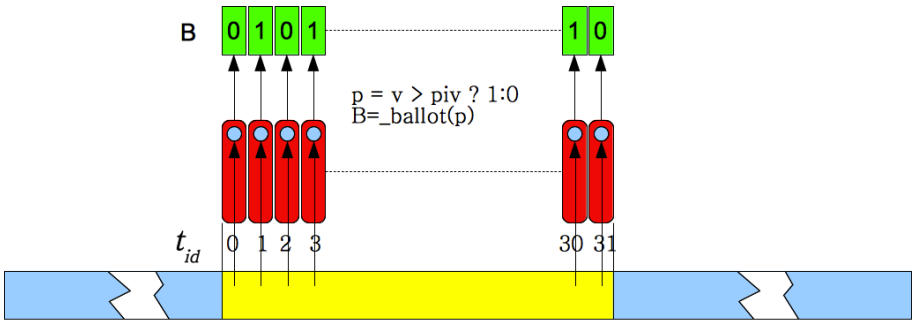


Figure 3.8 Read in process. Read in of the array is done incrementally in sets of 32 elements. As illustrated, the memory access is coalesced. The value is stored in a register. Simultaneously, the invocation of the warp voting function fills the bit array B based on the evaluation of the predicate that indicates if value in the register is greater than, equal to or less than the pivot.

If the element in the register is less than the pivot, then the thread needs to find how many of threads before it have elements less than the pivot and vice-versa. We use a combination of bit shift operations and the `_popc(x)` function on the integer result of the warp voting to accomplish this. The `_popc(x)` functions count the number of bits set to '1' in the input integer

'x'. For example, if the warp vote integer in binary is $B = 0101....$, then it is clear that threads 0,2 have elements less than the pivot and threads 1,3 have elements greater than or equal to the pivot. Each thread i computes the result $b = B >> (31 - i)$. When the _popc(b) function is applied to the result of this step, it will indicate the position in shared memory at which thread i will off-load its element that happens to be less than the pivot. Similarly, $[31 - \text{\_popc}(\tilde{b})]$ will indicate the position from the right side at which thread i will off-load its element that happens to be greater than or equal to the pivot. In the example, thread 3 will bit shift B to the right by 29 bits and the result will be $b = 0101$. Then _popc(b)=2. Therefore, thread 3 will off-load its element at the second location from the left in shared memory (Figure 3.9). Note that the total number of elements in shared memory that are less than the pivot is given by _popc(B). Two global counters, $g_<$ and $g_\geq$, keep track of the total number of elements less than the pivot and the total number of elements greater than or equal to the pivot, respectively, are also maintained. These two counters indicate the location in the auxiliary array at which the warp writes the incremental results of pivoting from shared memory. Next, the threads write the contents of the shared memory into the global auxiliary array with threads whose id is less than _popc(B) writing from the left side and other threads writing from the right side. This write process requires two coalesced writes, one for elements smaller than the pivot and one for elements greater than or equal to the pivot, into the auxiliary array (Figure 3.10).

Once a partition is complete, the output array (auxiliary array) has a left side of length L elements, each of whom is less than the pivot. The right side is of length R elements, each of

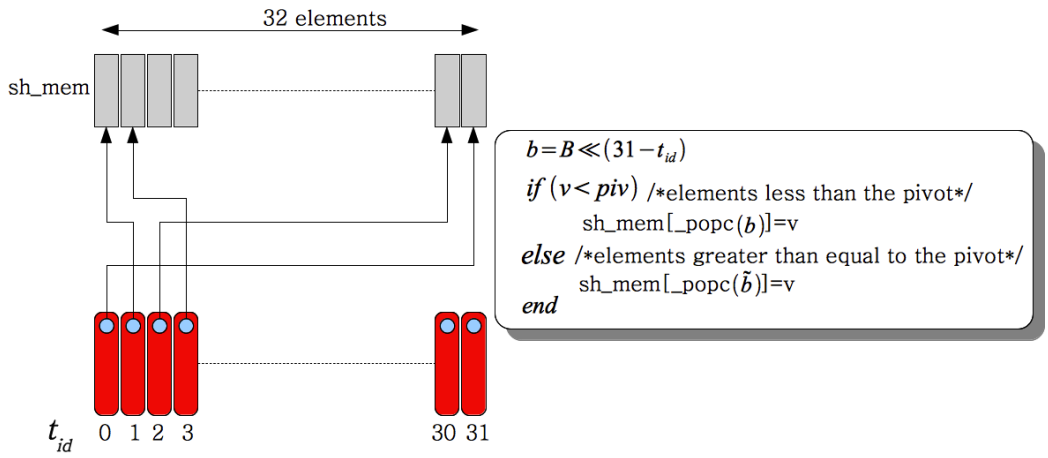


Figure 3.9 Pivot process. The pivot process is accomplished in shared memory. Each thread determines where in the shared memory the value has to be written. Values less than the pivot are accumulated on the left hand side and values greater than or equal to the pivot are accumulated on the right hand side. Since all threads write to different locations, there are no bank conflicts.

whom is greater than equal to the pivot. Suppose we need k nearest neighbors, and $k < L$; then we need to process only the left hand side. Suppose $k > L$; then we keep the left hand side as is, and partition the right hand side to find $k - L$ elements. Since the input and auxiliary arrays are swapped at the end of the partition process, in the second case ($k > L$), we would have to copy the left hand side from the auxiliary array to the input array. We can avoid copying the data by storing a stack of references that indicate the start and end indices and the arrays (auxiliary or input) where the partitions that form the k nearest neighbors are to be found. This significantly reduces the memory transactions needed for the operations.

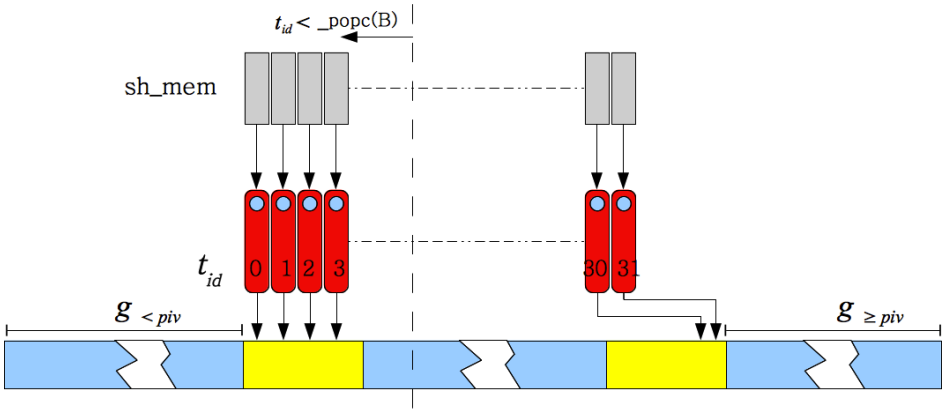


Figure 3.10 Write-out process: The thread id indicates (based on the computation `_popc(B)` whether a given thread is writing out an element less than the pivot or greater than or equal to the pivot. The values $g_{<}$ piv and $g_{\geq}$ piv that are maintained in shared memory and updated incrementally indicate the location in the global array the location of the last element that is less than the pivot and greater than or equal to the pivot. This operation involves at most two coalesced memory writes.

Chapter 4

Results

In this section we present the results of our benchmarks. Our implementation of task distribution and data partitioning is unique to the problem at hand; i.e., calculating one half of the distance matrix and generating k-NNG. While there are optimized algorithms for dense matrix multiplication on distributed computing systems, these methods are not suitable for our application. However, there are several comparable exact brute force k-NN implementations on GPUs [16, 20, 2, 4]. We chose to benchmark against [16] and [2], since the code is readily available. All implementations were compiled using C++ with appropriate compiler optimization flags. The implementations were run on a NVIDIA Tesla C2050. We used synthetic data sets for performance analysis. We provide benchmark results for distance computation, k-NN selection, and total calculation time.

4.1 k-NNG construction with Batch Index Sorting

This section presents performance analysis of kNNG with the batch-index sorting algorithm. First the results of single GPU benchmarks are presented. Performance analysis was

performed for varying different parameters, which showcase the behavior of our implementation for different scales. Benchmarks were obtained with various k (number of wanted nearest neighbors), n (number of data points) and D (dimension of each data vector). Our performance benchmarks show that our algorithm performance is superior to those of [16] and [2], even in single GPU execution. In addition, we benchmarked multi GPU performance analysis vs. that of [2], reaching better performance due to proper task distribution with a symmetric k-NNG structure.

4.1.1 Performance benchmarks of k-NNG construction algorithms in single GPU

Figure 4.1 compares the performance of our first k-NNG algorithm with that of [16] with varying k. In this test two constant parameters, data dimension and the number of samples, were set to d = 4096 and n = 16384, respectively. Figure 4.1 (a) shows the distance calculation, selection and the total k-NNG speedup versus [16]. The distance computation time is almost the same for both algorithms, because both formulate distance computation as a matrix-matrix multiplication and use the optimized CUBLAS library. Our version of the k-NN selection breaks even with the work of [16] 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 4.1(b) shows the speedup versus [2]. 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 $k = 1024$.

The degradation of performance of the k-NN selection algorithms in [16] and [2] occurs at large values of k, because the temporary list of k elements maintained for each thread does not fit into the fast shared memory on GPUs and is therefore maintained in global memory. Because of the nature of the insertion sort and the heap sort, this causes thread divergence and un-coalesced memory transaction that results in a huge drop in performance.

In order to show the performance of k-NNG with batch index sorting for different data dimensionalities, we performed a second set of benchmarks in which $n = 16384$ and $k = 512$ and d waqs varied. Figure 4.2(a) shows the comparison with [16]. As the complexity of distance calculation is linearly related to the number of data dimensions and the complexity of the selection part of k-NNG is not related to the data dimensionality, the speedup with respect to both distance and k-NN selection remains constant at $\approx 1\times$ and $\approx 6\times$, respectively. However,

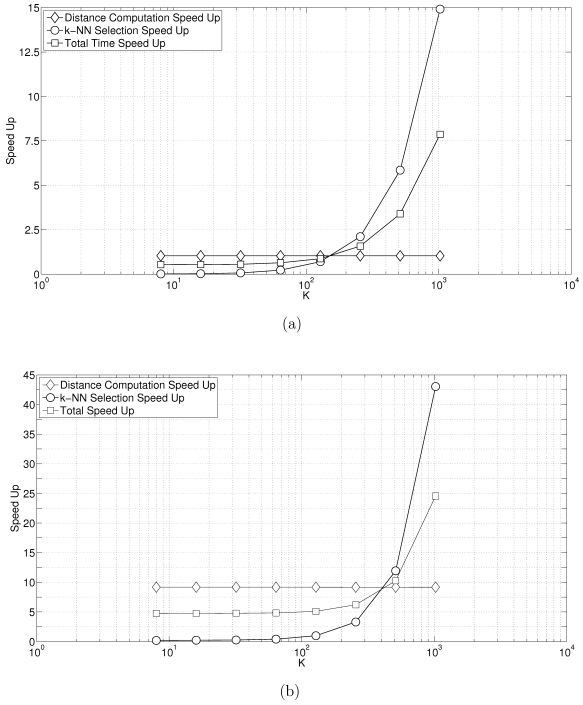


Figure 4.1 Performance analysis of k-NNG with batch index sorting. benchmark results for varying k in comparison with [16] and [2]. In this test our input data has the dimension d = 4096 and the number of input objects/vectors n = 16384. (a) shows the performance vs. [16]. (b) shows the performance vs. [2]

as the proportion of time for the distance computation increases linearly with d, the performance gain for the total time decreases from $5.7\times$ to $3.5\times$.

Figure 4.2(b) shows the speedup with respect to [2] . Once again, the speedup with respect to distance and with respect to k-NN selection remains constant at $\approx 9.5\times$ and $\approx 10\times$, respectively. As d increases, the proportion of time for the distance computation increases linearly with d. For the large d, the graph shows the stabilization of the overall speed up at $10.25\times$.

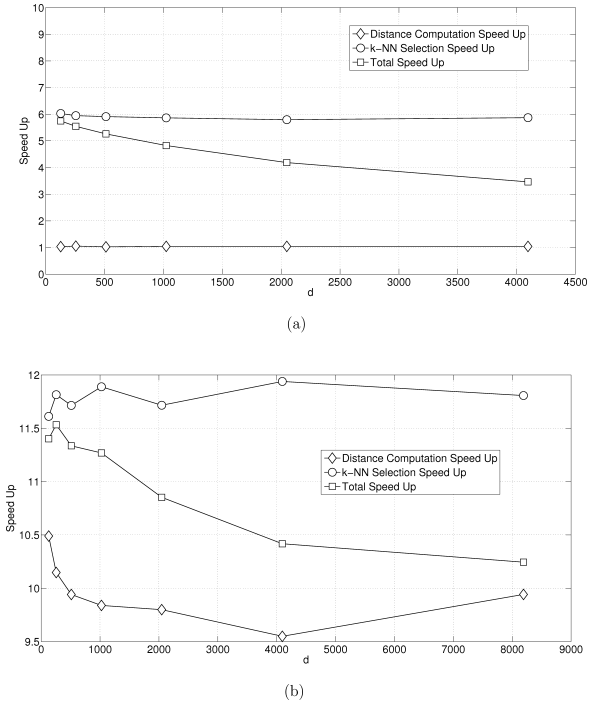
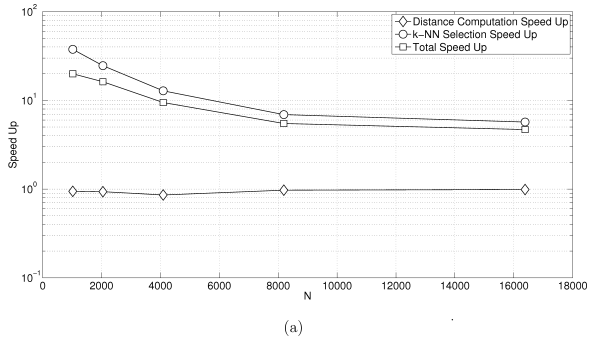


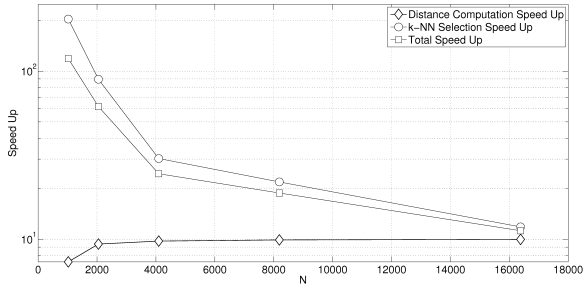
Figure 4.2 Performance analysis of k-NNG with batch index sorting. Benchmarks for varying d. In this test our input data has the number of closest neighbors $k = 512$ and the number of input objects/vectors $n = 16384$. (a) shows the performance vs. [16]. (b) shows the performance vs. [2]

Next we benchmarked the performance for different values of n. Specifically, we kept $d = 1024$ and $k = 512$ and varied n. Figure 4.3 shows the comparison with [16]. For a 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 4.3(b) shows the comparison with [2]. Once again, for a 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 [16] could only handle a model size up to $n = 16,384$. Note that our k-NN method grows

proportional to $n^2 \log(n)$ as opposed to $n^2 \log(k)$. However, for data with the ranges of n that fit into GPU memory, our k-NN method is still much faster.



(a)



(b)

Figure 4.3 Benchmarks for varying n. In this test our input data have the dimension $k = 1024$ and the number of input objects/vectors $d = 4192$. (a) shows the performance vs. [16]. (b) shows the performance vs. [2].

4.1.2 Performance analysis of Multi GPU k-NNG construction with batch index sorting

In order to show our algorithm's ability to exploit the symmetry of a k-NNG structure due to a proper task distribution, we tested our implementation vs. [2] for multi-GPU configuration. While the implementation in [2] requires all GPUs to be on a single computer (connected through a 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 by [2] on two GPUs on a single desktop, while 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 4.4 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 [2] 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 batch index sorting k-NN algorithm is not as efficient for small ks.

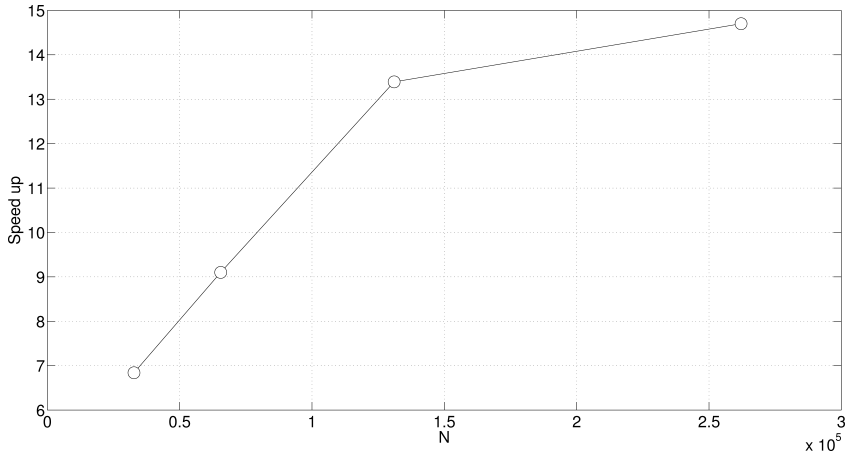


Figure 4.4 Performance analysis of Multi GPU k-NNG construction with batch index sorting. Benchmarks in comparison with [2]. In this test we used 2 GPUs. For the implementation of [2] algorithm, the 2 GPUs (Tesla 2050) were mounted on a single desktop machine. For our implementation, we use 2 nodes in our GPU cluster and opted to use only one GPU per node. The input data had dimension $d = 16384$, and the number of closest neighbors $k = 512$.

4.2 Benchmarks of k-NN selection with Quick-Select

In this section, we analyze the performance of the k-NNG construction with Quick-Select. Different tests were designed in order to showcase the performance of our algorithm with varying parameters, against k-NN algorithms presented in [33] and [16]. The work in [33]

uses a truncated bitonic sort for a k-NN search on GPUs. Finally, we benchmarked our code k^{th} element selection algorithm [7]. The k^{th} element algorithm selects the k^{th} largest/smallest values in a vector, and therefore is slightly different from the k-NN problem.

4.2.1 Performance analysis of single GPU Quick-Select against truncated bitonic sort and insertion sort

In this section we present performance benchmarks of our k-NNG construction with the quick select against those of [33] and [16]. In Figure 4.5 the comparison of quick select k-NN search algorithm is shown with that of [16]. A three-dimensional graph is chosen to represent the performance analysis of k-NN search algorithms with varying k and n. The data dimension is set to a constant value of $d = 128$ for all tests. The number of objects varies from $n = 1024$ to $n = 131072$, and the number of extracted nearest neighbors is varied from $k = 8$ to $k = 512$.

As the figure shows, the speedup grows exponentially with increasing n. For small k and n, the speedup is $\approx 3\times$. For small n values, increasing k leads to $\approx 250\times$ speedup. This speedup grows to up to $\approx 450\times$ for large values of n with increasing k.

Figure 4.6 shows the performance advantage of our algorithm when benchmarked against truncated bitonic sort by Sismanis. Our speedup ranges from 1.5x for $k = 2^3, n = 2^{17}$ to 5.3x for $k = 2^9, n = 2^17$. We could not go above $k = 2^9$ since the Sismanis implementation would crash. It is obvious that the speedup saturates for large n. This saturation point is further away as the size of k grows. This clearly shows that for $k > 2^9$, the speedup would grow even more. In Figure 4.7, we show the performance advantage when both the distance calculation and the

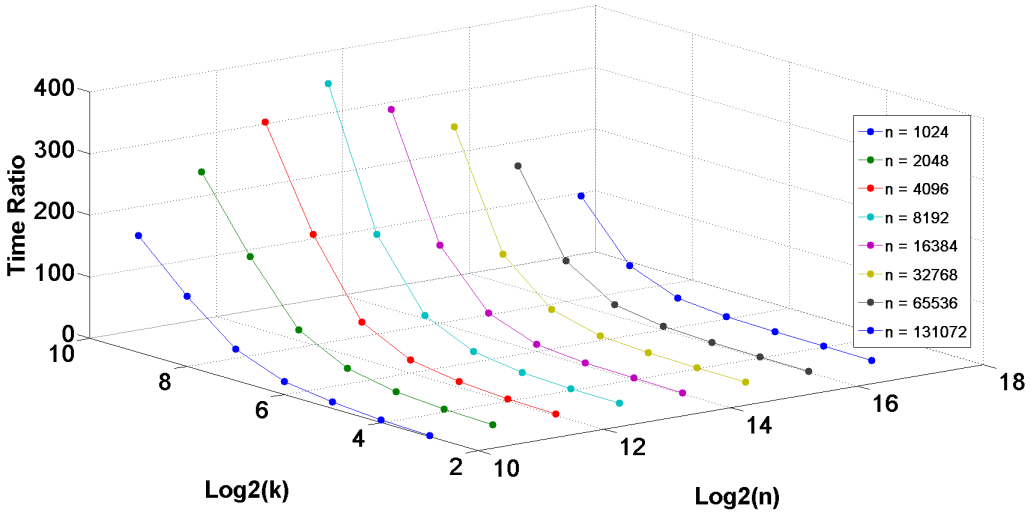


Figure 4.5 Quick-Select benchmarks against insertion sort [16]. In this set of tests the data dimension is set to be constant at $d = 128$ and k is doubled from $k = 8$ to $k = 512$. For each k, the performance graph is representing the timings for different n starting from $n = 1024$ to $n = 131072$.

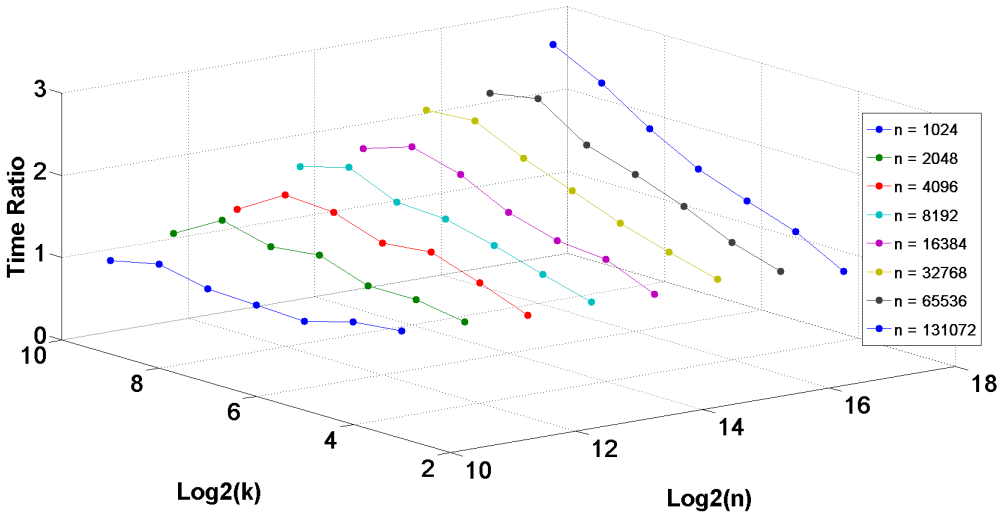


Figure 4.6 Performance analysis of a single GPU selection with Quick-Select. Benchmark results in comparison with that of TBiS is presented. In this set of tests the data dimension is set to be constant at d= 128 and k is doubled from k = 8 to k = 512. For each k, the performance graph is representing the timings for different n starting from n = 1024 to n = 131072.

k-NN search are included. The same general trend is observed. Since both algorithms use a similar method for distance calculation, the only advantage is due to our superior k-NN search method.

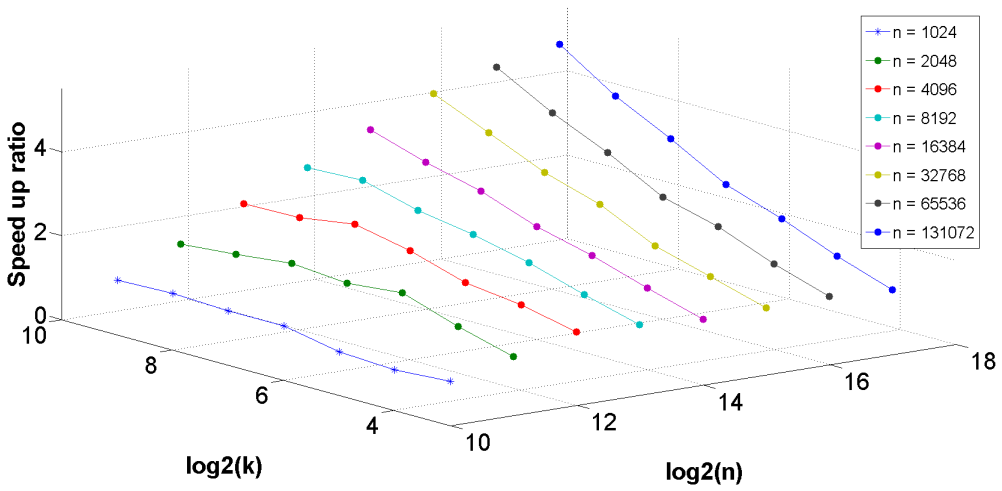


Figure 4.7 Performance analysis of quick select. Benchmark results in comparison with TBiS is presented. In this set of tests k is doubled from k = 8 to k = 512. For each k, the performance graph is representing the timings for different n starting from n = 1024 to n = 131072.

Our benchmarks against the MGPU Select was for the selection algorithm alone. Note that the MGPU Select works for one query at time. Moreover, the MGPU Select algorithm only selects the k^{th} largest/smallest element. We conducted multiple queries by first loading the distance matrix in global memory on the GPU and then running the MGPU Select in successive rows, one row at a time. Furthermore, while our algorithm finds the k smallest elements with indices, the MGPU Select algorithm only finds the k^{th} smallest element. For finding the k smallest elements with indices, the MGPU Select algorithm is no better than a plain sort and

selection [7]. Figure 4.8 shows the results. For small n, our algorithm has dramatic performance advantages ($\approx 100\times$). This is because the GPU is not saturated by the MGPU Select. With increasing n, we see a significant drop off and possible saturation of the performance gain at around 8x. We are not able to explore a larger n because the distance matrix does not fit into GPU memory.

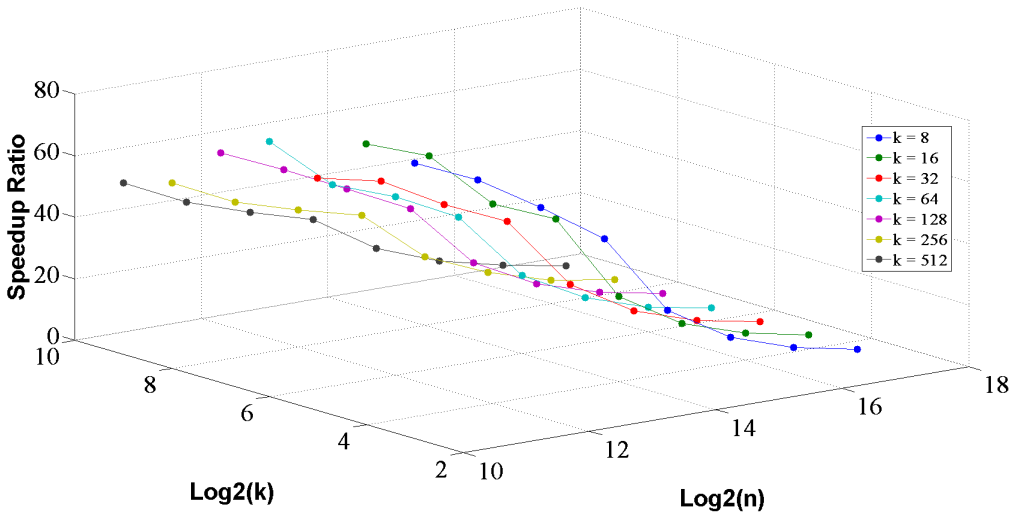


Figure 4.8 Performance analysis of quick select. Benchmark results in comparison with MGPU select is presented. In this set of tests k is doubled from $k = 8$ to $k = 512$. For each k, the performance graph is representing the timings for different n starting from $n = 1024$ to n = 131072.

4.3 k-NNG and manifold embedding

Our application of interest for the k-NN graph construction is manifold embedding. The basic idea behind manifold embedding is that a cloud of correlated high-dimensional data can be characterized with a low-dimensional hyper-surface that is embedded in the original high dimensional space. The manifold contains information about the individual objects and the

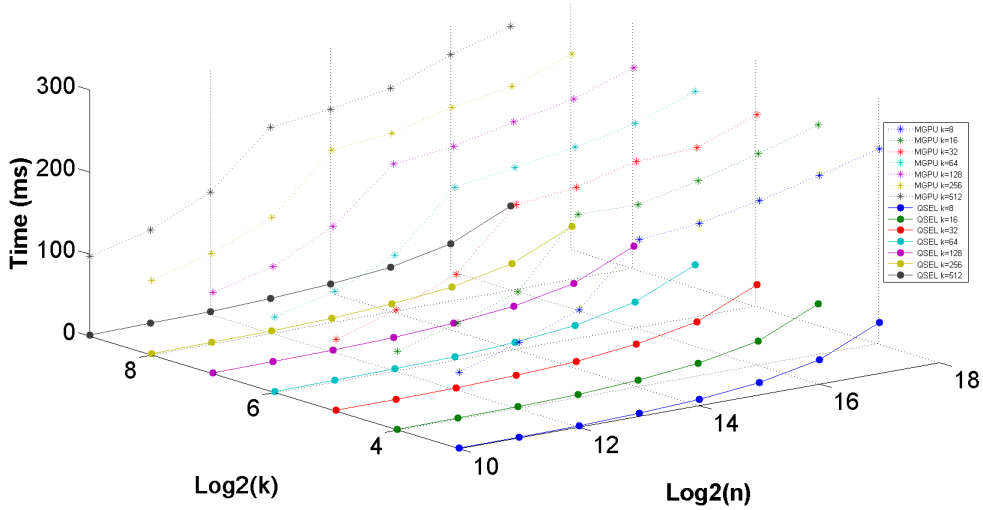


Figure 4.9 Timing benchmarks of quick select and MGPU select algorithms. In this set of tests k is doubled from k = 8 to k = 512. For each k, the performance graph is representing the timings for different n starting from n = 1024 to n = 131072.

system that generated the data. The main execution part of the manifold embedding is the generation of a neighborhood graph for an input data set. The k-NN graph can then be normalized and embedded in order to give the governing eigenfunctions of the low-dimensional manifold.

To evaluate and apply our algorithms in manifold embedding, we executed k-NNG with batch index sorting for two data sets. The first data set contained two million images of simulated diffraction patterns of a randomly oriented adenylate kinase (ADK) molecule. Each image has $126 \times 126 = 15876$ pixels; i.e., high dimensionality. The second dataset consisted of twenty million images of simulated diffraction patterns of denaturing ADK in ten different molecular conformations. (For more information about the structure of data sets, please refer to [32]). We evaluated the k-NNG algorithm with a previous implementation of a neighborhood

graph construction by using MATLAB technical computing language. The MATLAB implementation took 56 hours on an exclusive CPU cluster with 32 nodes for two million diffraction patterns with the use of a highly optimized ATLAS-BLAS library for multi-threaded Matrix-Matrix Multiplication in double precision. The cluster had one Xeon E5420 quad-core CPU per node with 16kB of L1 cache, 6144kB or L2 cache and 40GFLOPS of double precision computing power. Since the parallel MATLAB implementation did not take advantage of the symmetry of the distance matrix, one can assume that such an implementation would 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 3GB with 506GFLOPS of double precision computing power. There are 14 multi-processors sharing 720kB of L2 cache and each multi-processor having 48kB of user-configurable L1 cache/shared memory. In addition, each of the GPU nodes had two quad-core Xeon E5620s. Note that in our GPU cluster, the CPUs are used mostly for managing the GPUs and moving 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 single core of an Xeon E5420 and concluded that if all four cores of the CPU were active, we could achieve a roughly $7.7\times$ speedup using GPUs just for matrix multiplication alone. As shown earlier, our

complete implementation is slightly worse at a $6.6\times$ gain in performance.

Based on the complexity of the manifold embedding, the estimated execution time for a second dataset with twenty million snapshots on the CPU cluster was more than eight months. With the use of RME implementation on our GPU cluster, the execution time for twenty million snapshots was achieved in less than two weeks. Figure 4.10 shows the computational resources configuration and execution time of manifold embedding for each data ensemble.

	CPU #	GPU #	CPU	CPU RAM	GPU	GPU Global Memory	RME Total Time First dataset (Double precision)	RME Total time Second dataset
GPU Cluster	16	32	Intel(R) Xeon(R) E5620 2.4 GHz	48 GB	NVIDIA Tesla C2050	2.5 GB	4:23' Hrs	10 Days (Single Precision)
CPU Cluster	32	0	Intel(R) Xeon(R) E5620 2.4 GHz	48GB	–	–	56 Hrs	Approx. 233 days (Double Precision)

Figure 4.10 Clusters configuration and total timings for construction of neighborhood graph. number of GPU in each node m =2 and p_3 = 8

Chapter 5

Conclusions

In this thesis, we presented a distributed GPU-accelerated implementation of the brute force k nearest neighbor graph construction method. Our implementation runs on an exclusive access GPU cluster. It forms a central core of a software pipeline for data and compute intensive Manifold Embedding being used for structure and conformation recovery of biomolecules from a large data set of high noise images. The pipeline and individual algorithms have been benchmarked against a similar state-of-the-art system. Significant gains in overall performance demonstrated. As a result of this work, it is now possible to process an image data set with over 2×10^7 image vectors with dimensions exceeding 1.5^4 in a time span of 14 days compared with an estimated 180 days on a comparable CPU cluster.

5.1 Contributions

The contributions of this work are the following:

- A scheme for data partitioning and task assignment for efficient load-balanced execution of the brute force k-NNG method on a homogeneous cluster with GPU accelerated

nodes. This implementation uses multiple levels of parallelism (between nodes, between multiple cores in nodes, and on GPUs) along with parallel I/O.

- Two new GPU algorithms for finding k-NNG from a given distance matrix
  - The first called batch index sorting uses three sort operations to directly find the k-NNG without further manipulation of the distance matrix.
  - The second is an efficient GPU implementation of the quick select algorithm and requires the computation of the transpose of the distance matrix for k-NNG construction. This implementation is the fastest method in its class with a nearly 4x gain over the state-of-the art.

Overall, the implementation developed as part of this thesis has achieved a 6x performance gain over a comparable implementation running on a cluster of CPUs.

5.2 Discussions

There is room for further enhancements 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 percent use of GPU resources as opposed to 90 percent use by finely tuned CPU libraries. A better GPU matrix multiplication library would further enhance the performance of our approach.

The brute force implementation requires $O(n^2)$ distance calculations. This dominates the computational expense for a large n. One way of reducing this complexity will be to use a hybrid algorithm that approximately subdivides the data sets into overlapping sets to reduce the computations (distance computation and selection) for each input vector based on the set membership. While such methods exist, to our knowledge there are no parallel cluster implementations with GPU acceleration.

Finally, performance of the algorithms is significantly impacted by the nature of the data (data dimension as well as size) and the execution configuration parameters. For example, currently we manually select the number of data partitions P of the input matrix A and the size of sub-partitions of A_I within the nodes. The GPU quick select algorithm in particular is very sensitive to the number of simultaneous queries and the arrangement of execution resources (the number of warps per thread block). Currently, these execution parameters are manually set and adjusted through trial and error. This can be automated is the future.

5.3 Future work

Our future work will continues in three different areas. The first is the development of a stand-alone k-NN search library on GPU that outperforms all state of the art algorithms and that can be easily accessible and applicable to huge applications of k-NN search and k- NN graph construction. In order to achieve this goal, we are planning to create a user friendly

environment and a parameter space for different range of applications. The second possible target of future work in this area is to release a user friendly version of a k-NNG construction library for GPU distributed systems. The proposed library would use the expandability and flexibility of proposed algorithm with regards to computational system configurations and data characteristics respectively. The third area lies on the application side, especially manifold embedding. In this work, we achieved the computational capability to deal with data ensembles at least one order of magnitude larger than current datasets. However, with the growing pace of expansion in input data sizes (both in number of dimensions and reference data points) alongside technological advances in GPU hardware and software configurations, new strategies for data management, partitioning and algorithm development will be needed in the foreseeable future.

References

[1] Laksono Adhianto and Barbara Chapman. Performance modeling of communication and computation in hybrid mpi and openmp applications. *Simulation Modelling Practice and Theory*, 15(4):481–491, 2007.

[2] 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, 2012.

[3] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582. Society for Industrial and Applied Mathematics, 1994.

[4] 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*, pages 380–392. Springer, 2011.

[5] Mikhail Belkin and Partha Niyogi. Convergence of laplacian eigenmaps. *Advances in Neural Information Processing Systems*, 19:129, 2007.

[6] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems: Jade Edition*, pages 359–372, 2011.

[7] Sean Bexter. Mgpu select, 2012.

[8] Jie Chen, Haw-ren Fang, and Yousef Saad. Fast approximate k nn graph construction for high dimensional data via recursive lanczos bisection. *The Journal of Machine Learning Research*, 10:1989–2012, 2009.

[9] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *Visualization and Computer Graphics, IEEE Transactions on*, 16(4):599–608, 2010.

[10] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[11] 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*, pages 253–262. ACM, 2004.

[12] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011.

[13] Veit Elser et al. Reconstruction algorithm for single-particle diffraction imaging experiments. *Physical Review E*, 80(2):026705, 2009.

[14] Joachim Frank. Single-particle imaging of macromolecules by cryo-electron microscopy. *Annual review of biophysics and biomolecular structure*, 31(1):303–319, 2002.

[15] Russell Fung, Valentin Shneerson, Dilano K Saldin, and Abbas Ourmazd. Structure from fleeting illumination of faint spinning objects in flight. *Nature Physics*, 5(1):64–67, 2008.

[16] 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.

[17] Parisa Haghani, Sebastian Michel, Philippe Cudré-Mauroux, and Karl Aberer. Lsh at largedistributed knn search in high dimensions. In *International Workshop on Web and Databases (WebDB)*, 2008.

[18] Piotr Indyk. Nearest neighbors in high-dimensional spaces. 2004.

[19] Peter Wilcox Jones, Andrei Osipov, and Vladimir Rokhlin. Randomized approximate nearest neighbors algorithm. *Proceedings of the National Academy of Sciences*, 108(38):15679–15686, 2011.

[20] 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*, pages 769–773. IEEE Computer Society, 2010.

[21] Khronos Group. *The OpenCL Specification*, September 2010.

[22] Donald E. Knuth. *The art of computer programming*. Addison–Wesley Pub. Co., 2006.

[23] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. In *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pages 151–155, 2009.

[24] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. Gpgpu: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.

[25] Frank Natterer and Ge Wang. The mathematics of computerized tomography. *Medical Physics*, 29:107, 2002.

[26] nVidia. *CUBLAS Library User Guide*. nVidia, v5.0 edition, October 2012.

[27] CUDA Nvidia. Programming guide, 2008.

[28] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[29] Rodrigo Paredes, Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Experimental Algorithms*, pages 85–97. Springer, 2006.

[30] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[31] P Schwander, R Fung, GN Phillips Jr, and A Ourmazd. Mapping the conformations of biological assemblies. *New Journal of Physics*, 12(3):035007, 2010.

[32] Peter Schwander, Dimitrios Giannakis, Chun Hong Yoon, and Abbas Ourmazd. The symmetries of image formation by scattering. ii. applications. *Optics Express*, 20(12):12827–12849, 2012.

[33] Nikos Sismanis, Nikos Pitsianis, and Xiaobai Sun. Parallel search of k-nearest neighbors with synchronous operations. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6. IEEE, 2012.

[34] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008.

[35] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113. IEEE, 2012.

[36] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.