MASSIVELY PARALLEL ALGORITHMS FOR POINT CLOUD BASED OBJECT RECOGNITION ON HETEROGENEOUS ARCHITECTURE

Linjia Hu
Michigan Technological University, linjiah@mtu.edu

MASSIVELY PARALLEL ALGORITHMS FOR POINT CLOUD BASED OBJECT RECOGNITION ON HETEROGENEOUS ARCHITECTURE

By

Linjia Hu

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2017

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Saeid Nooshabadi*
Committee Member:	*Dr. Timothy Havens*
Committee Member:	*Dr. Scott Kuhl*
Committee Member:	*Dr. Stefaan De Winter*
Department Chair:	*Dr. Min Song*

Dedication

To my loved mother Yuxiang Duan and my father Huazhong Hu, whose affection, love and encouragement make me able to hew out of the mountain of despair a stone of hope; my brilliant, gorgeous and supportive wife Xiaoting Cao; my sweet, exuberant and kind-hearted little boy Andrew Cao Hu.

Contents

List of Figures

List of Tables

Preface

This dissertation presents my research work in pursuing the PhD degree in Computer Science Department at Michigan Technological University. This dissertation includes previously published articles in Chapter 1, Chapter 2, Chapter 3 and Chapter 4. All the research works described herein were conducted under the supervision of my advisor Professor Saeid Nooshabadi.

Chapter 2 contains articles and proceedings *Parallel 3D Local Descriptor on* NVIDIA GPU *published in GPU Technology Conference 2014* [1], *A 3D local descriptor SHOT on massively parallel processors* in *IEEE International Conference on Consumer Electronics (ICCE) 2015* [2], *G-SHOT: GPU Accelerated 3D Local Descriptor for Surface Matching* in *Journal of Visual Communication and Image Representation (JVCI) 2015* [3].

Chapter 3 and Chapter 4 include articles and proceedings *Massively Parallel KD-Tree Construction and Nearest Neighbor Search Algorithms* in *IEEE International Symposium on Circuits and Systems (ISCAS) 2015* [4], *Parallel Randomized KD-Tree Forest on GPU Cluster for Image Descriptor Matching* in *IEEE International Symposium on Circuits and Systems (ISCAS) 2016* [5], *Massive parallelization of approximate nearest neighbor search on KD-tree for high-dimensional image descriptor matching* in *Journal of Visual Communication and Image Representation 2017* [6].

Chapter 4 and Chapter 5 contains articles *Parallelization of Buffered Approximate Nearest Neighbor Search on Forest of Randomized KD-trees for Image Descriptor Matching* submitted to *ACM Transactions on Parallel Computing* and *High-Dimensional Image Descriptor Matching Using Highly Parallel KD-tree Construction and Approximate Nearest Neighbor Search* submitted to *Journal of Parallel and Distributed Computing*. These two papers are still in review.

Chapter 1 and Chapter 6 involve all the published and the submitted papers. The permission to reuse all the published attached in Appendix A.

As the first author of all these papers and manuscript, with the guidance from my adviser, I completed the proposal, implementation, and analysis of the algorithms and methodologies. These articles were completed by me with the help from advisor.

Acknowledgments

First and foremost, I would like to thank my advisor Professor Saeid Nooshabadi for his support and guidance throughout my Ph.D. program at Michigan Technological University.

I would like to thank all my committee members, Professor Timothy Havens, Professor Scott Kuhl and Professor Stefaan De Winter for their time and invaluable advice.

I would like to thank my family and relatives. Without the love and support from my parents, it was impossible for me to become who I am today. Even though they are unschooled in the rural area of China, they never stopped encouraging me to pursue high education. My parents have always been a steady source of encouragement and support when I needed. My wife sacrificed a lot to support our family and bring up my son during my Ph.D. program. I couldn't compensate her and my son enough for the years of sacrifice they made to enable my success. Finally, there is a special appreciation to my cousin, Mr.Li Yude. He encouraged and supported me to pursue Ph.D. overseas. He always has been my advisor at each hard stage in my life. His kindness and insight inspired me to move forward.

Abstract

With the advent of new commodity depth sensors, point cloud data processing plays an increasingly important role in object recognition and perception. However, the computational cost of point cloud data processing is extremely high due to the large data size, high dimensionality, and algorithmic complexity. To address the computational challenges of real-time processing, this work investigates the possibilities of using modern heterogeneous computing platforms and its supporting ecosystem such as massively parallel architecture (MPA), computing cluster, compute unified device architecture (CUDA), and multithreaded programming to accelerate the point cloud based object recognition. The aforementioned computing platforms would not yield high performance unless the specific features are properly utilized. Failing that the result actually produces an inferior performance. To achieve the high-speed performance in image descriptor computing, indexing, and matching in point cloud based object recognition, this work explores both coarse and fine grain level parallelism, identifies the acceptable levels of algorithmic approximation, and analyzes various performance impactors. A set of heterogeneous parallel algorithms are designed and implemented in this work. These algorithms include exact and approximate scalable massively parallel image descriptors for descriptor computing, parallel construction of k-dimensional tree (KD-tree) and the forest of KD-trees for descriptor indexing, parallel approximate nearest neighbor search (ANNS) and buffered ANNS (BANNS) on the KD-tree and the forest of KD-trees for descriptor matching. The results show that the proposed massively parallel algorithms on heterogeneous computing platforms can significantly improve the execution time performance of feature computing, indexing, and matching. Meanwhile, this work demonstrates that the heterogeneous computing architectures, with appropriate architecture specific algorithms design and optimization, have the distinct advantages of improving the performance of multimedia applications.

Chapter 1

Introduction

Object recognition is a fundamental and active research topic in computer vision. Typical applications include robotics, automation, surveillance, remote sensing, navigation and medical treatment [7] [8]. In the past few decades, the methods of two-dimensional (2D) object recognition have been extensively explored and became mature technologies. Compared to 2D images, three-dimensional (3D) images exhibit a lot of advantages. The feature descriptor for 3D images encodes much more invariant geometrical information, which makes it robust under clutter and occlusion. Moreover, the increasing availability of low-cost 3D sensors promotes migration toward the processing of 3D information [9] [10] [11]. With the advent of cheap commodity hardware, point cloud based 3D object recognition, aiming to identify objects in a point cloud correctly, has become progressively popular [12] [13]. Although many algorithms have been proposed in the area of 3D object recognition, it is still very challenging to recognize objects in complex scenes in real-time. Good recognition accuracy typically incurs high computational complexity.

The emerging heterogeneous architecture system and its developing ecosystems, such as multi-core central processing unit (CPU), massively parallel architecture (MPA) such as graphical processing unit (GPU) and computing cluster, offer new opportunities for 3D point cloud based object recognition. The extensive computation in 3D point cloud processing can be accelerated by performing them in parallel on the heterogeneous architectures, leading to significant runtime reduction. However, designing efficient architecture-specific algorithms require deep analysis of the recognition pipeline and the exploration for parallel processing in the compute-extensive stages in the pipeline.

1.1 Point Cloud Based Object Recognition

1.1.1 Point Cloud

A point cloud is a data structure for the representation of a multi-dimensional collection of points. In a 3D point cloud, a point on the surface of an object is represented by its x, y and z coordinates [12] [13]. An extension to 3D point cloud is achieved by adding the color information. The typical sources for point cloud datasets are stereo cameras sensors, 3D scanners, or time-of-flight cameras. They are also generated synthetically from a computer model. Figure 1.1 is a classic Stanford Bunny generated by a 3D point cloud editor [14] and Figure 1.2 is an example of Granite Dells LiDAR point cloud [1].

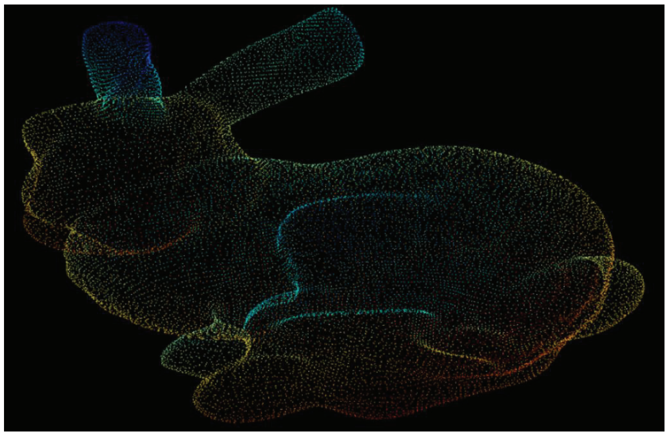


Figure 1.1: Point cloud of bunny

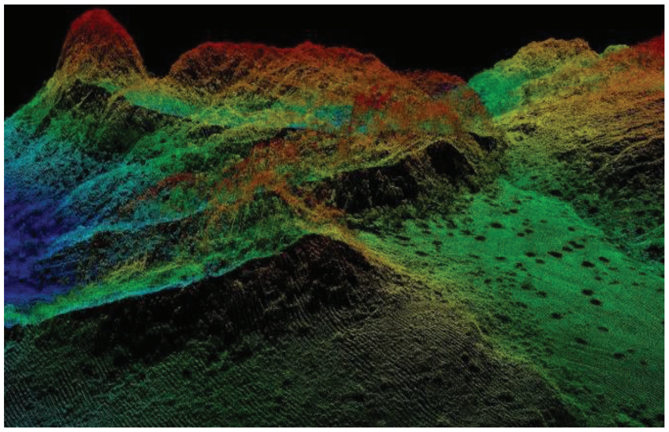


Figure 1.2: Point cloud of mountains

[1]This figure is from the open topography website: http://www.opentopography.org/.

1.1.2 Object Recognition Pipeline

Object recognition involves recognizing and determining the pose of the model (user chosen object) in a scene such as a photograph or range scan. Typically, an example of the object to be recognized is presented to a computer vision system in a controlled environment, and then for an arbitrary input such as a video or image stream, the system locates the previously presented object [15]. The approaches for recognizing a 3D object depends on the properties of the object in the presence of significant clutter and occlusion [16]. The point cloud based 3D object recognition approaches, employing the surface feature of objects, can be classified into two categories: global feature based recognition and local feature based recognition [13] [17].

The global feature based recognition use low-level object surface information to locate the object in the scene. The whole surface of the object is represented by a single descriptor. Typically, it does not take the 3D geometric constraints of the object into account during the matching, nor handle the clutter and occlusion [17] [18]. In the recognition procedure, the scene point cloud needs to be segmented to extract individual object instance in the clutters and occlusions situation. This approach is mainly investigated in object shape classification and model retrieval [19] [20].

The local feature based recognition, in contrast, encodes the geometric information of each key point extracted from the model, which makes it more robust to clutter and occlusion [21] [22] [23]. It first identifies some key points in a scene and then computes a feature descriptor for each key point as local image descriptors. These feature descriptors of the scene are finally matched against the feature descriptors of the model for association pairing in recognition. This approach works well for objects which have distinctive features. Objects which have good edge features or blob features can be successfully recognized [24] [25] [26].

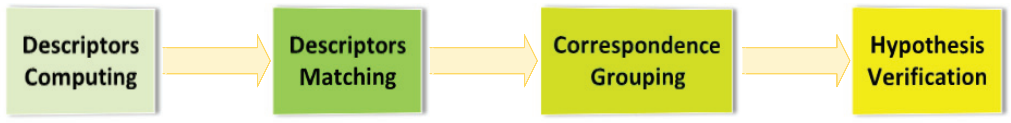


Figure 1.3: 3D object recognition pipeline

This work concentrates on the local feature based object recognition through surface matching. An object can be recognized in a scene by comparing a scene surface to a model surface stored in a database. When the model surface is matched to the scene surface, an association is made between the model and the scene. The conventional 3D object recognition pipeline comprises four stages as shown in Figure 1.3. In the first stage, descriptor computing, we explore an efficient and compact local representation of 3D objects in the scene and model, known as local 3D descriptors. The descriptors, localized descriptions of the global shape of the object, are invariant to rigid transformations. In the second stage, descriptor matching, through correlation of images, point correspondences between the model and the scene descriptors are established. In the third stage, correspondence grouping, geometric consistency is used to group the correspondences from which plausible rigid transformations that align the model with the scene are calculated. In the last stage, the transformations are then refined and verified using a modified iterative closest point algorithm [27] [28] [29]. We concentrate on the first two stages in this work, since the first two stages consume most of the recognition time and some of the algorithms can be reused in the works in the last two stages.

1.1.2.1 Descriptors Computing

Before computing the descriptors of model and scene, the key points should be detected and extracted from the model and the scene datasets. Key point detection aims to identify a set of interest points that are distinctive and repeatable under some variations including occlusion, clutter, viewpoint changes, noising and so on. There are several detectors proposed in the literature addressing these issues [30] [31] [32]. The two primary characteristics of the key point detector are repeatability and distinctiveness. With good repeatability, a detector can extract the same key points in a variety of noisy conditions. With super distinctiveness, a detector extracts key points that can be easily described, matched and classified, and therefore highly characterizing a surface [33]. Moreover, the distinctiveness of detector depends on the local descriptor of the key point. A set of key points can be salient depending on the traits of the local descriptors applied on them. To conduct a fair comparison among local descriptors, in this work, we did not adopt any existing detector technologies. Instead, we use a uniform sampling of key points over the point cloud datasets.

Once key points have been detected and extracted from a point cloud, a set of local descriptors will be associated with them. A local descriptor projects the local surface features around the key point into a proper feature space. A common trait of the local descriptors is the definition of a local support which is used to determine the subset of neighboring points around each key point. These neighbor points will be used to compute the descriptor of the key point [34] [35], [36]. Descriptiveness and robustness are two critical qualifications for a local feature descriptor [37]. The higher descriptiveness of the descriptor, the more distinctive it becomes in classification and matching. The robustness is used to characterize the invariability of a descriptor in the presence of noise, detection errors, clutter, occlusion, and geometric deformations. A good descriptor can provide a good trade-off between the descriptiveness and the robustness. $Recall$ versus $1 - Precision$ is a generally used criteria for local image descriptor evaluation. It calculates the feature recall and precision under different thresholds for feature matching [38] [39] [40].

In the past two decades, there has been active research interest in local image descriptors. The proposed descriptors include structural indexing [41], point signature [42], $3D$ point fingerprint [43], exponential mapping [44], spin images [45], local surface patches [46], shape index [47], 3D shape context [48], and intrinsic shape signatures [49]. The computation of a local descriptor depends on the local reference of each key point, with respect to a normal surface vector. However, in all these proposals, the choice of local reference for each descriptor is ambiguous or not unique. The work [50] has analyzed the repeatability and robustness of the existing local descriptors and divided them into two major categories, *viz.*, signature and histogram. To leverage on benefits of both categories, a novel local descriptor for 3D point cloud named signature of histogram of orientations (SHOT) was proposed in [38]. It combines the merits of signature and histogram descriptors. In this work, we concentrate on the parallelization of this novel specified local image descriptor on the MPA.

1.1.2.2 Descriptors Matching

Descriptors of the scene and model will be similar, but not exactly the same due to variations in surface sampling, noise from a different view and other environmental factors. Once the image descriptors are computed for the scene and each model in the point cloud, descriptor matching is launched to generate point-to-point correspondences. The closeness of a pair of descriptors is measured through the Euclidean distance in this work. A matching threshold in the Euclidean distance is commonly used to remove descriptor pairs that are far apart in the descriptor space. Finally, the correspondences are built up between each scene descriptor and its nearest neighbor in the model descriptor database [51] [52][53].

Figure 1.4 illustrates how the descriptor matching works. For a key point P_i in the model of a bunny, its image descriptor is d_i^m, where $d_i^m \in R^D$, with D features for each descriptor. The matching algorithm is performed to search for the closet neighbor in the descriptor dataset of the scene $\{d_0^s,\ d_1^s,\ d_2^s\ \ldots\ d_N^s\}$. It computes the distance of this image descriptors of model (d_i^m) to each descriptor of the scene (d_j^s). During the search, the P nearest neighbors of point P_i index and their associated distance are kept as the search results.

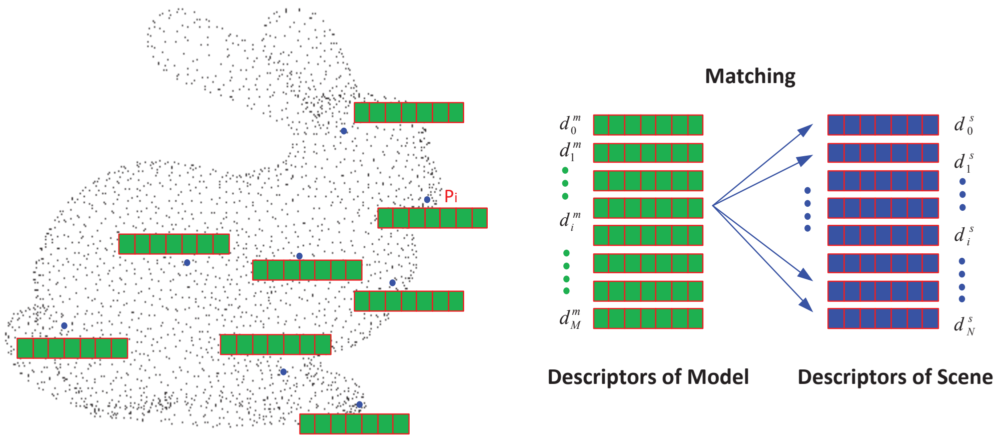


Figure 1.4: Model and scene descriptors matching

There has been a large body of work in image descriptor matching, exploring the efficient indexing and nearest neighbor search (NNS) in a point cloud. A brute force P-NNS compares M query points with all the N points in the search set, to obtain their P nearest neighbors. It results in the time complexity of $\mathrm{O}(MN)$ [54]. However, the search can be made more efficient by using spatial data structures, such as R-tree, B-tree, quad-tree, binary space partitioning (BSP) tree, K-means tree and K-dimensional tree (KD-tree). These structures subdivide the space containing all the points into smaller spatial regions, where a hierarchy is recursively imposed on the smaller regions. The NNS on this hierarchical spatial data structure is generally more efficient since it can prune large portions of target dataset.

In 3D point cloud object recognition, the NNS require fast performance [55] [56]. Unlike the typical applications with single point query [57], the NNS in these point cloud applications involves batch processing a large number of query points to match them against the points in the model object. To increase the feature descriptiveness, the image descriptors, typically, require high dimensionality [58] [59] [60] [3] [61] [62] [63] [64]. However, feature matching in high-dimensional space demands extremely high computational workload. To mitigate the computational workload associated with high-dimensional digital image descriptor matching, in this work, we propose a suite of massively parallel algorithms for indexing during the tree construction and the approximate NNS (ANNS) on the GPU and the GPU cluster.

1.1.2.3 Correspondence Grouping

In 3D object recognition, the stage followed by descriptor matching is correspondence grouping. As the result of the descriptor matching, point-to-point correspondences are built up by associating pairs of descriptors between model and scene in the descriptor space [65] [66]. However, only one correspondence between model and scene cannot be used to compute a transformation between them because a descriptor encodes less than six necessary degrees of freedom. Two or more oriented point correspondences are needed to compute a transformation between model and scene if the position and normals are encoded in the descriptors. If many correspondences are grouped, the resulting transformation will be more robust than the one computed from only a few correspondences [67] [68] . To avoid combinatoric explosion problem, geometric consistency is used to determine groups of correspondences. Some correspondences are discarded by enforcing geometrical consistency between them [69]. With the refined correspondences, the plausible transformations between model and scene can be computed [70] [71].

The common approach used in point cloud based object recognition is an iterative algorithm. Assuming the initial transformation between the model and the scene is rigid, the set of correspondences related to each model is grouped into subsets, each one holding the consensus for a specific rotation and translation of that model in the scene. If the consensus of the subsets of correspondences is too small, it will be removed. All correspondences are grouped into subsets that are geometrically consistent [69]. Starting from a seed correspondence $c_i = \{d_i^m, d_i^s\}$, where d_i^m and

d_i^s are the descriptors of key points in model and scene in correspondence c_i, then scanning all the un-grouped correspondences, the correspondence $c_j = \{d_j^m, d_j^s\}$ is added to the subset computed by c_i if it satisfies the following threshold,

$$|\|d_i^m - d_i^s\| - \|d_j^m - d_j^s\|| < \varepsilon \tag{1.1}$$

with ε being a parameter of this method, intuitively representing the consensus set dimension. Figure 1.5 demonstrates how the correspondence grouping works. We assume that there are seven correspondences between the model and the scene ($c_k = \{d_k^m, d_k^s\}$, $k \in \{0, 1, 2...6\}$). Starting from $\{c_0\}$, we check all the remaining correspondences. Since correspondences c_2, c_3, c_5 and c_6 satisfy 1.1 with respect to c_0, we put them in a single group $\{c_0, c_2, c_3, c_5, c_6\}$, and ignore c_1 and c_4.

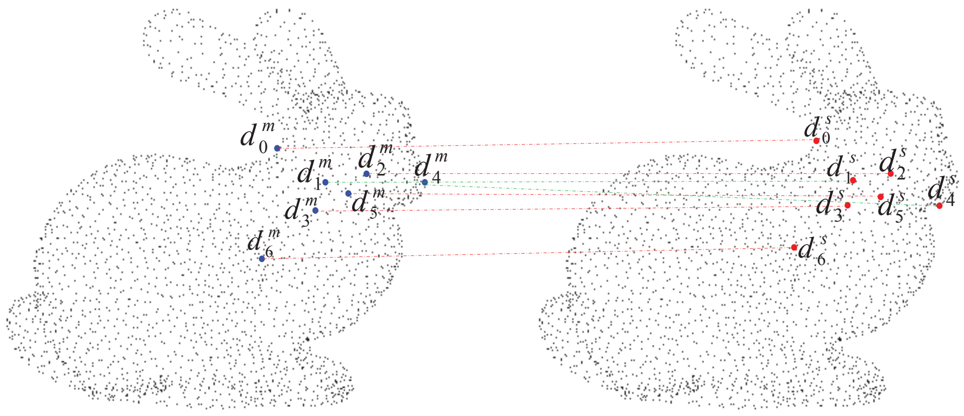


Figure 1.5: Model and scene correspondence grouping

1.1.2.4 Hypothesis Verification

The geometric consistency in correspondences grouping prunes a large number of inconsistent correspondences. However, it cannot ensure that all the existing correspondence pairs in each cluster are consistent with unique six degrees of freedom (like rigid rotation and translation of the model over the scene). The purpose of hypothesis verification is to find the best transformation of the model to the scene by eliminating transformations that are inconsistent when all of the scene data are compared to all of the model data. The general verification algorithm is the iterative closest point algorithm that can handle partially overlapping point sets and arbitrary transformations [72] [73]. During verification, point correspondences are spread over the surfaces of the scene and the model from the initial correspondences cluster. If many correspondences are established through spreading, a match between model and scene is validated [74].

Verification starts with an initial group of correspondences which are pruned through geometric consistency in correspondence grouping stage. With this group of correspondences, the transformation of the model to the scene is computed. Next, this transformation is applied to the model point cloud. The other correspondences are spread from the initial correspondence cluster as follows: for each scene point in an initial correspondence group, its closest scene points in the surface point cloud are turned into correspondences with their closest model points if the distance between the scene and closest model points is less than a threshold. This process is applied to each of the correspondences just added until no more correspondences can be created in a recursive fashion [75] [76] [77]. Figure 1.6 illustrates how initial correspondences, established by matching point cloud image descriptors, are spread over the surfaces of a model of the bunny point cloud.

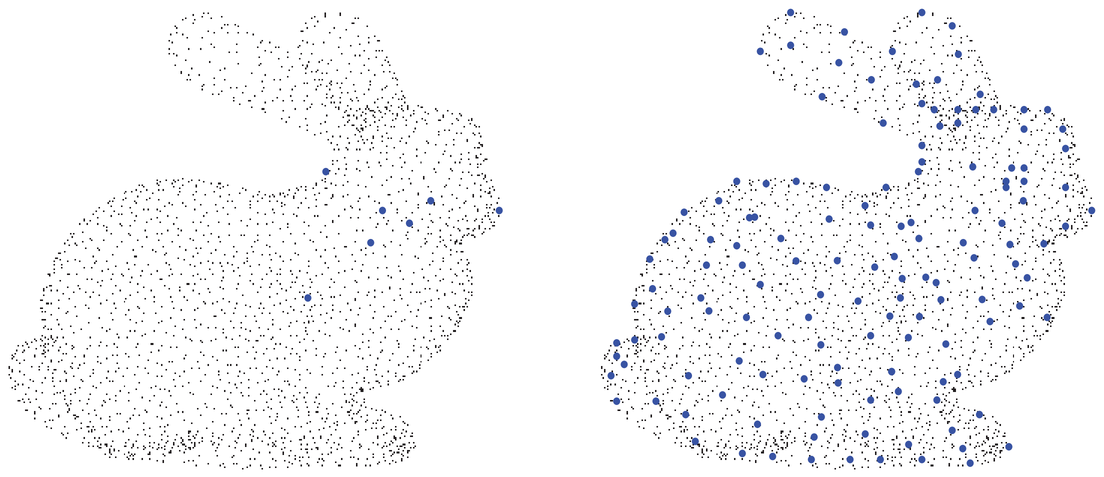


Figure 1.6: Initial and final correspondences of hypothesis verification

1.1.3 Point Cloud Library (PCL)

The point cloud library (PCL) is an open source project for large-scale point cloud based 2D and 3D image processing. This open source library contains various state-of-the-art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation. These general algorithms can be used to remove outliers from noisy datasets, integrate 3D point clouds, segment parts of a scene, extract key points, compute descriptors based on their geometric features, recognize objects, reconstruct surfaces from point clouds and visualize point clouds [12] [13]. The massively parallel algorithms designed and implemented in this work are based on the commonly used serial algorithms in this library.

1.2 Heterogeneous Parallel Computing

Heterogeneous computing refers to applications running on heterogeneous computing architecture (HCA) with two or more processor types. The HCA platform we employed encompasses a mix of general-purpose processors and massively parallel processors. General purpose processors like the CPUs provide powerful features like multiple execution units, branch prediction, floating point operations, and elaborate caching schemes to enhance the performance. Massively parallel processors, such as the GPU, being single instruction multiple data style processor, are used to reduce computing cost on the parallel data stream. These architectures require the customized data-parallel algorithm to mitigate response time to critical events [78][79][80]. Typically, the HCA utilizes multiple CPUs and GPUs as shown in Figure 1.7, and allows the developer writing applications that can seamlessly integrate the resource of the CPUs and GPUs [81] [82] [83]. The proposed works in this dissertation focus on this HCA.

The GPUs, apart from its well-known graphics rendering capabilities, have been extended to perform general intensive computations on big datasets in parallel, while the CPUs can run the operating system and perform traditional serial tasks. The general purpose parallel processing on the GPU is supported by standard APIs and tools such as CUDA and OpenCL [84][85].

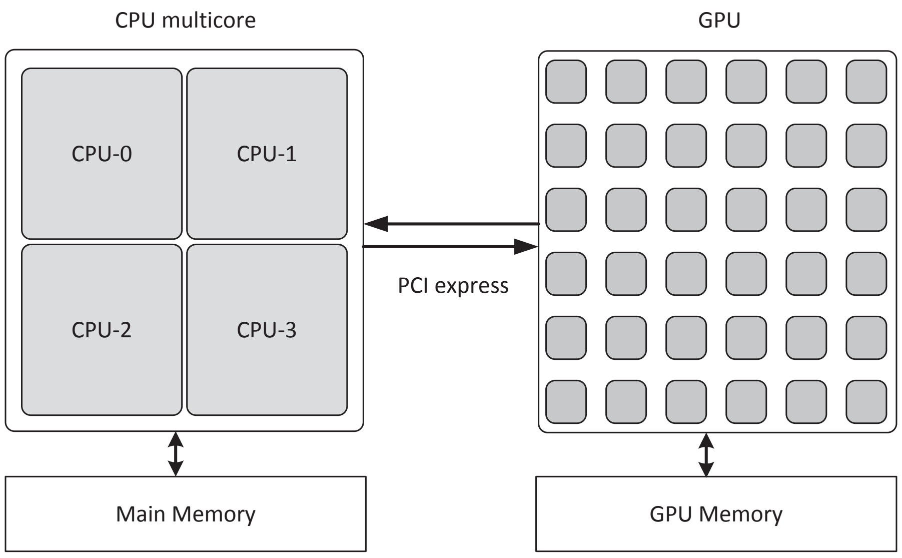


Figure 1.7: Heterogeneous computing architecture (HCA)

1.2.1 GPU Hardware Architecture

All algorithms in this work have been designed and implemented on the NVIDIA Fermi platforms GTX 570 or GTX 660, so we present a general review of the Fermi architecture here.

As shown in Figure 1.8, the Fermi architecture features up to 512 accelerator cores

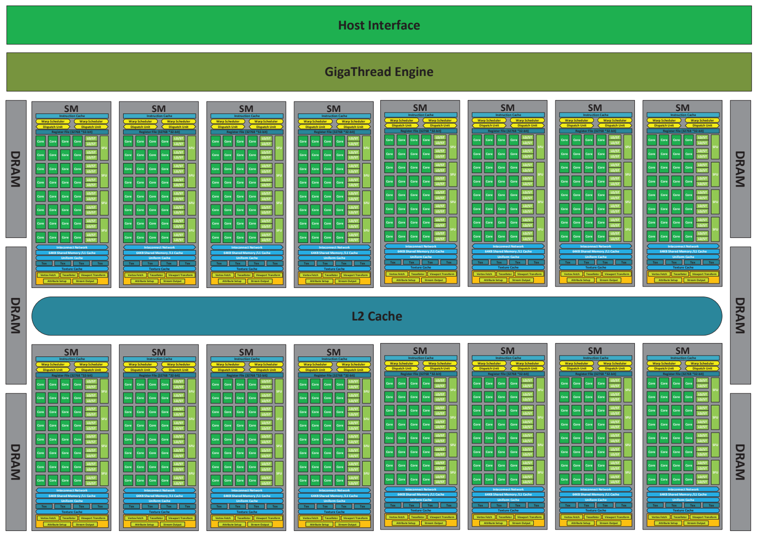


Figure 1.8: Fermi GPU architecture

called CUDA cores, or streaming processor (SP). The CUDA cores are organized in 16 streaming multiprocessors (SMs), each with 32 CUDA cores. Each CUDA core, as shown in Figure 1.9, has a fully pipelined integer arithmetic logic unit (ALU) and a floating point unit (FPU) that executes one integer or floating point instruction per clock cycle.

Fermi also includes a unified 768 KB L2 cache that is shared across all 16 multiprocessors. It also has a 384bit GDDR5 DRAM memory interface supporting up to 6 GB on-board memory. The host interface connects the GPU to the CPU via peripheral component interconnect express (PCIe) bus. The GigaThread global scheduler distributes thread blocks to multiprocessor thread schedulers. This scheduler handles concurrent kernel execution and out of order thread block execution. Each multiprocessor has 16 load/store units, allowing source and destination addresses to be

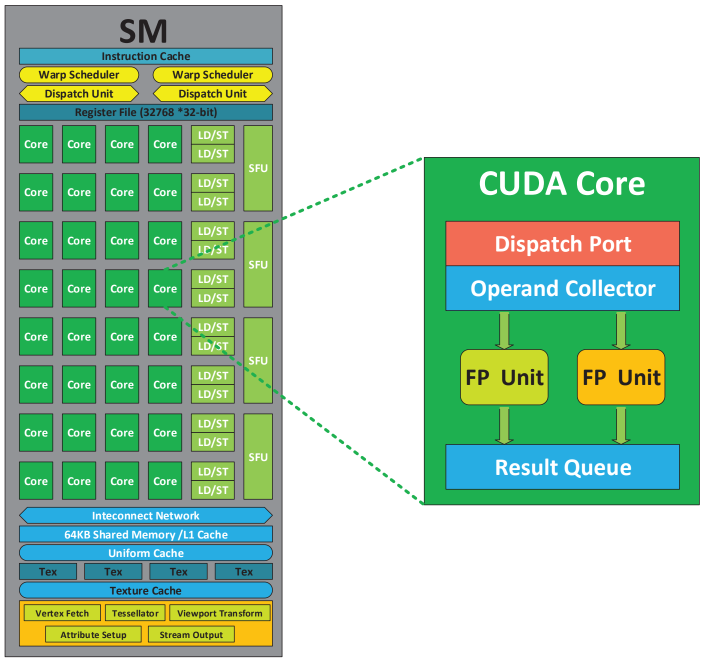


Figure 1.9: Streaming multiprocessor architecture

calculated for 16 threads per clock cycle. Special function units (SFUs) execute intrinsic instructions such as sine, cosine, square root, and interpolation. Each SFU executes one instruction per thread, per clock. The multiprocessor schedules threads in groups of 32 parallel threads called warps. Each multiprocessor features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The Fermi dual warp scheduler selects two warps, and issues one instruction from each warp to a group of 16 CUDA cores, 16 load/store units, or four SFUs. The multiprocessor has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. A traditional critique of GPUs has been their lack of IEEE compliant floating point operations and error correcting code (ECC) memory.

However, these shortcomings have been addressed by NVIDIA, and all of their recent GPUs offer fully, $IEEE754$ compliant single and double precision floating point operations, in addition to the ECC memory [86] [87] [88].

1.2.2 CUDA Programming Model and Memory Model

In CUDA, parallel programs are encapsulated in kernel functions written in C and $C{+}{+}$. The applications on GPU use single program multiple data (SPMD) computing paradigm. The computing model in CUDA is shown in Figure 1.10. All copies of the parallel program, named threads, execute the same set of instructions but operate on different data. The threads, then, are further grouped into a thread block. The threads in a thread block have access to a common shared memory. Thread blocks, in turn, are arranged in a grid with a common access to the global DRAM memory and cache. The thread blocks are executed on the GPU multiprocessors. As each multiprocessor has 32 computing cores, the threads in a thread block execute in units of 32 threads as a thread warp.

The CUDA memory hierarchy is depicted in Figure 1.11. A thread executing on the GPU can access the global DRAM, and on-chip memory through 6 different memory spaces; registers, local memory, shared memory, global memory, constant memory, and texture memory.

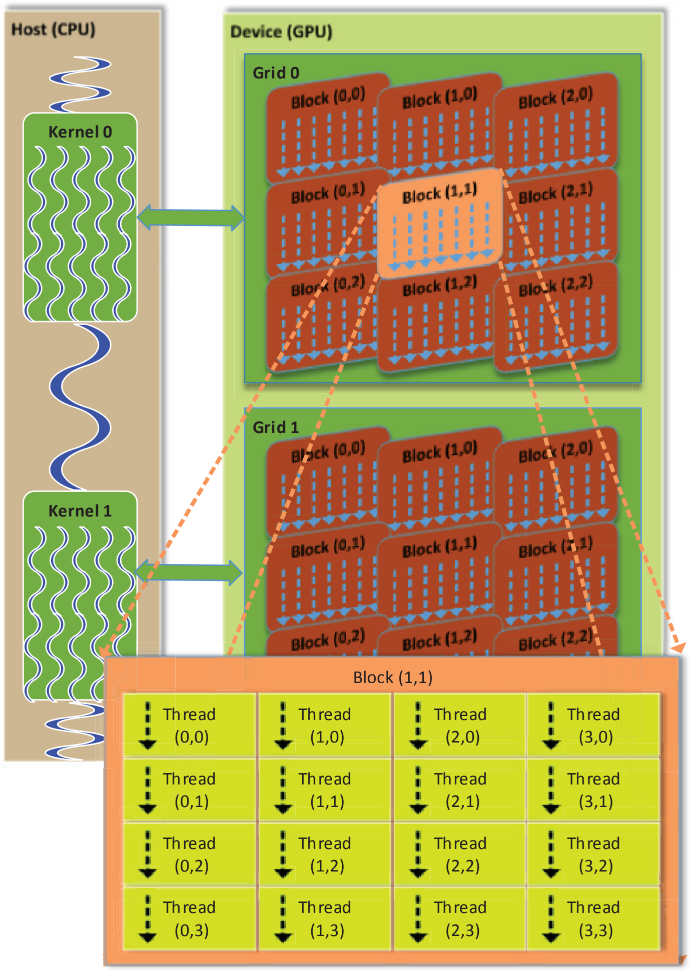


Figure 1.10: CUDA programming model

The global memory is used to exchange data between the CPU and the GPU and transferring data between the threads from different blocks. As an off-chip memory, global memory has a high latency. Shared memory, being on-chip, has much faster access time, especially if bank conflicts between the threads in a thread block are avoided. However, per thread block allocated shared memory is limited (64 KB on Fermi GPU). Another type of memory is a cache enabled constant memory for read-only data with high-speed access time if high hit rate can be guaranteed. Like

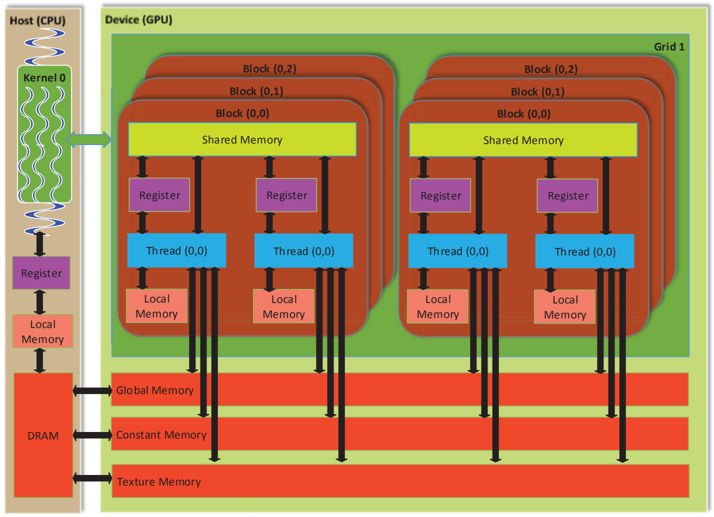


Figure 1.11: CUDA memory model

constant memory, texture memory is cached on-chip, so in some situations, it can provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality.

1.3 Scope of This Work

The goal of this work is to investigate and develop highly efficient algorithms and optimization techniques for the point cloud based 3D object recognition by exploiting a variety of parallel technologies and pruning less effective computations in local image descriptor computing, indexing, and matching on heterogeneous architecture system.

1.3.1 Parallelization of Image Descriptor Computing

Compact local feature descriptor of the 3D object that relies on local invariant features is a key in surface matching. The descriptor SHOT as a new 3D object local descriptor can achieve a good balance between descriptiveness and robustness [38]. However, its computation workload is much higher than the other 3D local descriptors. To make it usable for real-time applications, we investigate the development of suitable massively parallel algorithms on the GPU for computation of high density and large-scale 3D object local descriptors. We design two alternative parallel algorithms (G-SHOT); one exact and one approximate, on the GPU to speed up the original serial SHOT. Experimental results show both algorithms exhibit outstanding speed performance.

1.3.2 Parallelization of the ANNS

To overcome the high computing cost associated with high-dimensional digital image descriptor matching, we present a massively parallel ANNS on the KD-tree on the modern MPA [89] [90]. The proposed algorithm is of comparable quality to the traditional sequential counterpart on the CPU. However, it achieves a high speedup factor when applied to high-dimensional real-world image descriptor datasets. The algorithm is also studied for factors that impact its performance to obtain the optimal runtime configurations for various datasets. The performance of the proposed parallel ANNS algorithm is also verified on typical 3D image matching scenarios. The implementation in this work will potentially benefit the real-time image descriptor matching in high dimensions.

1.3.3 Parallelization of the Construction of KD-tree and the BANNS

To mitigate the image descriptor indexing, we present a parallel KD-tree construction. Moreover, to reduce the cost of descriptor matching, we propose a buffered ANNS (BANNS) on KD-tree on the MPA. To improve the runtime performance of the ANNS, we design an efficient sliding window for a parallel BANNS on KD-tree to mitigate the high cost of global memory accesses. When applied to high dimensional real-world image descriptor datasets, the proposed KD-tree construction and the BANNS algorithms are of comparable quality to the traditional sequential counterparts on the CPU, while outperforming their serial CPU counterparts by significant speedup factors. Moreover, we verify the features of the parallel algorithms on typical 3D image matching scenarios.

1.3.4 Parallel and Distributed BANNS on the Forest of Randomized KD-trees

To further address the computational challenges of KD-tree construction and the ANNS to real-time processing, we present parallel and distributed algorithms for the construction of the forest of randomized KD-trees and the BANNS on a cluster equipped with the MPA devices of the GPU [91] [64]. To utilize the GPU cluster platform, we design distributed randomized KD-tree forest for the BANNS to alleviate the backtracking cost on single KD-tree. Additionally, the algorithms are studied for the performance impact factors to obtain the optimal runtime configurations for various datasets. When applied to high-dimensional real-world image descriptor datasets, the proposed algorithms for KD-tree forest construction and the BANNS on the GPU cluster are of comparable matching quality to the coarse grain parallel counterparts on the CPU cluster with message passing interface (MPI), while outperforming counterparts by significant speedup factors. Moreover, we verify the features of the parallel algorithms on typical 3D image matching scenarios.

1.4 Overview of Chapters

This dissertation consists of five chapters. Chapter 2 presents the design of two novel parallel SHOT local image descriptors, one exact G-SHOT and the other approximate G-SHOT, to significantly reduce image descriptor computing time. Chapter 3 describes the design and implementation of a parallel ANNS on the MPA which can accelerate the descriptor matching greatly. Chapter 4 presents algorithms for parallel KD-tree construction and a massively parallel BANNS on the GPU, which can accelerate the indexing and matching significantly. Chapter 5 describes the distributed ANNS on the randomized forest of KD-tree on the GPU cluster. Chapter 6 concludes the major contributions.

Chapter 2

Massively Parallel 3D Local Image Descriptor G-SHOT

Surface matching is one of the core techniques for 3D object recognition and surface registration in computer vision. Compact local feature descriptor of 3D object that relies on local invariant features is a key in surface matching. The SHOT as a novel 3D object local descriptor can achieve a good balance between descriptiveness and robustness. However, its computation workload is much higher than the other 3D local descriptors. To make it usable for real-time applications, it requires parallel search algorithms that can run on a common massively parallel processor such as the GPU. This chapter investigates the development of suitable massively parallel algorithms on the GPU for computation of high density and large-scale 3D object local descriptors. The chapter presents the design of two alternative parallel algorithms (G-SHOT); one exact, and one approximate, on the GPU to speed up the original serial SHOT. Experimental results show both algorithms exhibit outstanding speed performance. The exact massively parallel G-SHOT descriptor comes at no cost to the descriptiveness with a speedup factor of up to 40.70, with respect to the serial SHOT on the CPU. The approximate parallel achieves a speedup factor of up to 54 with minor degradation in descriptiveness, with respect to its serial counterpart. The chapter also analyzes the descriptiveness of both parallel G-SHOTs algorithms through a set of recall-precision curves at two noise levels. The proposed algorithms are integrated into the PCL, an open source project for image and point cloud.

2.1 Introduction

Surface matching is a key tool in the 3D object recognition that locates model objects in a scene through building local correspondences between the model and the scene. It has found its way in numerous areas such as computer vision, robotics, automation, remote sensing, and perception. The most common method for surface matching is to explore effective and compact local representations of the point cloud of 3D objects, known as local 3D descriptors, and establish correspondences by matching those descriptors. In the past 20 years, there has been strong research interest in local descriptors. The techniques proposed include structural indexing [41], point signature

[42], 3D point fingerprint [43], exponential mapping [44], spin images [45], local surface patches [46], shape index [47], 3D shape context [48], and intrinsic shape signatures [49]. The computation of a local descriptor depends on local reference of each key point, with respect to a normal surface vector. However, in all these proposals, the choice of local reference for each descriptor is ambiguous and not unique.

Most recent work [38] [50] has analyzed the repeatability and robustness of existing local descriptor techniques and divided them into two major categories, *viz.*, signature and histogram. The signature-based descriptor describes the 3D neighborhood of a given key point by defining an invariant local reference frame (LRF), according to the local coordinates of points in the neighborhood point set. For each point in the neighborhood point set, one or more geometric measurements are encoded. The histogram based descriptor describes the key point by accumulating local geometrical or topological measurements into histogram bins according to a specific quantized domain which requires the definition of either a reference axis or reference frame. Broadly, signature descriptors are potentially highly descriptive due to the use of spatially well-localized information, whereas histograms descriptors trade-off descriptive for robustness by compressing geometric structure into bins. To leverage on benefits of both categories, a novel local 3D descriptor named SHOT [38] combines the merits of signature and histogram descriptors. Due to its repeatable LRFs, the SHOT descriptor exhibits a better descriptive power and robustness. However, its benefits come at a significant increase in the computational complexity.

A point cloud is a data structure for the representation of a multi-dimensional collection of points. In a 3D point cloud, for example, a point on the surface of an object is represented by its x, y and z coordinates. The typical sources for point cloud datasets are stereo camera sensors, 3D scanners, or time-of-flight cameras. They are also generated synthetically from a computer model. Figure 2.3 is the classic Stanford Bunny generated by a 3D point cloud editor.

SHOT as an effective 3D descriptor has already been integrated into the PCL, a large-scale, open source project for 2D and 3D image and point cloud processing [12]. PCL framework contains numerous state-of-the-art algorithms that can be used to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract key points and compute descriptors to recognize objects in the scene based on their geometric appearance, create surfaces from point clouds and visualize them.

To overcome the performance bottleneck of the SHOT descriptor in the PCL framework, this chapter proposes two alternative highly efficient parallel algorithms that target the massively parallel architecture of the GPU. We have targeted implementation on the GPU platform, as it is finding its way beyond graphics processing into general purpose computing. It offers massively data-parallel architecture alternative to the CPU through a large number of computing cores. In particular, the GPU has been widely employed for fast and real-time implementation of 3D image and point cloud processing algorithms [92] [93] [94] [95] [96]. The potential for the implementation of the surface matching algorithm on the GPU comes from the fact that descriptor computations for key points are independent of each other. It is well suited for parallelization on a massively parallel programming paradigm of GPUs[1]. This work presents two efficient GPU accelerated parallel SHOT descriptors named G-SHOTs.

This chapter is organized as follows. Section 2.2 briefly outlines the mathematical model of the SHOT descriptor. Section 2.3 describes the complexity of the SHOT descriptor and presents the exact and approximate parallel alternative implementations of the SHOT on GPU (G-SHOT) as a library component in the PCL framework. Section 2.4 presents the experimental results of both algorithms. Section 2.5 evaluates the trade-off between the speedup and descriptiveness of two parallel SHOT descriptors. Section 2.6 concludes the chapter.

2.2 Mathematical Model of the SHOT Descriptor

The strength of the SHOT descriptor is based on two features. First, the SHOT is a 3D descriptor and has a repeatable and robust LRF. Second, it combines the

[1]In this chapter, we use NVIDIA's compute unified device architecture (CUDA)[97] computing paradigm. The GPU used in the experiment in this work is the NVIDIA GTX 570 with 15 streaming multiprocessor (SM), with each SM having 32 stream processor (SP) cores [98] [99]

merits of signature and histogram categories of descriptors to create a more effective descriptor.

2.2.1 Repeatable Local Reference Frame (LRF)

To facilitate the following mathematical derivation of LRF, we assume that the radius of the neighborhood sphere for key point $\mathbf{p}$ is R, there are K nearest neighbors $\mathbf{p}_i$ in the sphere, the covariant matrix of the K points in the neighborhood sphere is $\mathbf{M}$, and the distance between key point $\mathbf{p}$ and neighbor point $\mathbf{p}_i$ is d_i.

The repeatable LRF is based on the estimation of the normal direction of key point on a surface [50]. This estimation involves the total least squares (TLS) of the normal direction computed by eigenvalue decomposition (EVD) of the covariant matrix $\mathbf{M}$ of the K points in the neighborhood sphere. TLS of normal direction is represented by the eigenvector corresponding to the smallest eigenvalue of $\mathbf{M}$. To increase the repeatability of the LRF, in the SHOT algorithm, smaller weights are assigned to distant points in the sphere and bigger weights are assigned to nearby points. Also, to improve the robustness, all the K points laying within the sphere that will be used to calculate the descriptor are included in formation of the covariant matrix as,

$$\mathbf{M} = \frac{1}{\sum\limits_{i:d_i \leq R}^{K} (R - d_i)} \cdot \sum_{i:d_i \leq R}^{K} (R - d_i)(\mathbf{p}_i - \mathbf{p})(\mathbf{p}_i - \mathbf{p})^\top \tag{2.1}$$

To uniquely determine the sign direction of the normal at the key point the methodology described in [38] for sign disambiguation for EVD is employed. The methodology is to change the sign of each singular value or reorient each eigenvector to make it consistent with the majority of the vectors used for the computation of the normal. The sign of a normal along a local axes $s \in \{x, y, z\}$ is determined to be as s^+ or s^- in the opposite direction as,

$$\mathbf{S}_{\mathbf{s}}^{+} \doteq \{i : d_i \leq R \wedge (\mathbf{p}_i - \mathbf{p}) \cdot \mathbf{s}^{+} \geq 0\} \tag{2.2}$$

$$\mathbf{S}_{\mathbf{s}}^{-} \doteq \{i : d_i \leq R \wedge (\mathbf{p}_i - \mathbf{p}) \cdot \mathbf{s}^{-} \geq 0\} \tag{2.3}$$

2.2.2 Descriptor Organization

Inspired by the well-established 2D feature descriptor, scale invariant feature transform (SIFT) [100], the SHOT relies on a set of local histograms that compute on a specific subset of points defined by a regular grid superimposed on the key point patch. For each key point, the SHOT technique uses an isotropic spherical grid partitioned along the radial, azimuth and elevation axes. The coarse partitioning of the spatial grid produces a small cardinality of the descriptors. The choice 32 spatial volumes are proven to be adequate, resulting in eight azimuths, two radial and two elevation divisions [38] [50]. Figure 2.1 shows the formation of eight azimuths and the two radial divisions, and Figure 2.2 exhibits the formation two elevation divisions.

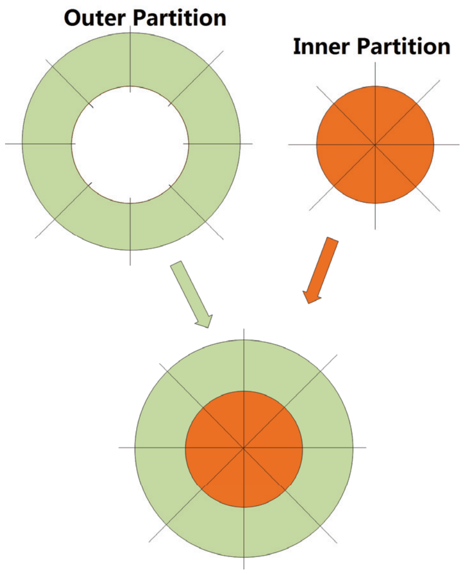


Figure 2.1: Azimuth/radial partition

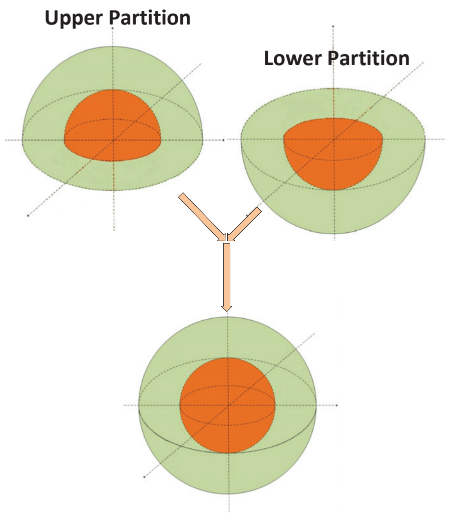


Figure 2.2: Elevation partition

Each segment within sphere in Figure 2.3 encodes a descriptive entity represented by its local histogram. The formation of the local histogram is shown in Figure

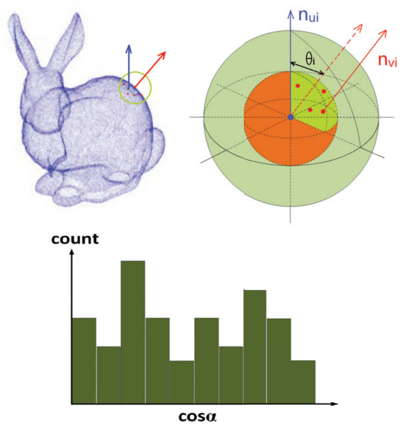


Figure 2.3: Local histogram for each volume in key point sphere

2.3, for a key point in the point cloud of Stanford Bunny, with one neighbor sphere encompassing the point (light green). For the local histograms of this segment, we accumulate point counts for each of the 32 segments into bins according to function $\cos\theta_i$, with θ_i the angle between the normal at each point $\mathbf{p}_i$ within the spherical grid segment (n_{v_i}), and the normal at the key point (n_{u_i}). Choice of binning using $\cos\theta$ has the advantage that it can be easily computed by the dot production as $cos\theta_i = n_{v_i} \cdot n_{u_i}$. Further, an equally spaced bin on $cos\theta_i$ is equivalent to a spatial varying spaced bins on θ_i. This has a significant advantage that coarser bins are created for directions close to the reference normal direction and finer ones for the orthogonal directions. For each of 32 volumes in the neighborhood sphere of the key point, there are local histograms with 10 bins. So, there is a total of 320 bins for each key point descriptor. Since the descriptor is based upon a set of local histograms, to avoid boundary effects for each point being accumulated into a specified local histogram bin, SHOT perform quadrilinear interpolation between the bin in the local histogram and the bins having the same index in the local histograms corresponding to the neighbor spherical segments within the neighborhood sphere of the key point under consideration.

2.3 Implementation of the G-SHOT Algorithms

The SHOT descriptor consists of two key parts: one the calculation of unique LRF for each key point, and the other computation of the descriptor histogram. The data profiling results of SHOT descriptor on the CPU show that these two parts of the algorithm together consume more than 90% of the total computing time. Since key points can be processed independent of each other, both parts can be performed in parallel on the GPU to achieve a significant speed advantage through massive parallel programming model of CUDA.

2.3.1 Exact G-SHOT Algorithm

The process of parallelization of key point calculations is carried out in six phases, that include two CUDA kernels, two memory copy operations and two host functions as show in Algorithm 1.

Before the launch of Kernel I for the computation of LRF and Kernel II for the histogram, for execution on GPU, we need to copy the data structure (key and surface points data) for the descriptor from the CPU host memory to GPU device memory. After the completion of execution of Kernel II end result is transferred from the device memory to host memory. Prior to the launch of kernel II, the irregular key points must be removed.

2.3.2 Approximate G-SHOT Algorithm

The G-SHOT Algorithm I even though achieves high speedup with respect to the existing serial CPU SHOT, suffers from multiple bottlenecks limiting the speedup performance. The reason is that Algorithm I provides for massively parallel processing of key points, but the workload associated with key points (or the GPU threads), is highly uneven. The thread block execution time is determined by the CUDA thread with the longest runtime that associated with key point with the largest number of neighbors. Unfortunately, the number of key point neighbors varies greatly in most typical datasets. For example, in a point cloud with 66053 key points and 307200 surface points, the number of neighbors for key points ranges from 6 to 1172, with the median of only 90. If we restrict neighbors for each key point to less than 400, 300, 200 and 90, the speedup performance of SHOT can be improved by 5.44%, 8.72%, 11.13% and 15.54% accordingly. To even the processing workload, in the approximate Algorithm II, each CUDA thread truncates the neighbor points of a key point so that it does not exceed the median. In Kernel I, each thread only process up to a median number of neighbor points to compute the unique LRF. The similar pruning work is done in Kernel II to compute the descriptor feature.

Table 2.1
Experimental platform details

OS Type	**Ubuntu 12**
Kernel Version	$2.6.34.969.fc13.x86\_64$
CPU Type	**Intel** $i7-960$
CPU Clock Speed	3.20GHz
Number of Core	4
Number of Thread	8
GPU Type	**GeForce GTX** 570
Global Memory Size	2559MBytes
Multiprocessor Number	15
CUDA cores/Multiprocessor	32
GPU Clock Speed	1.46GHz
Memory Clock Rate	1900.00Mhz
Memory Bus Width	320bit

Table 2.2
Profiling results of the parallel SHOT on the GPU

Operations	**Time(s)**	**Percentage**
Data Structure Conversion	0.83	2%
H2D Memory Copy	2.90	7%
GPU Kernels	36.91	89%
D2H Memory Copy	0.82	2%

2.3.3 Performance Optimizations

Performance of SHOT algorithm on the GPU is determined by a composition of several factors, including access coalescing in global memory, bank conflicts in shared memory, branch divergences, thread synchronization overhead and the organization of thread blocks [97].

To ensure that global memory accesses on the GPU are coalesced, *i.e.* threads accesses to memory are combined into a single transaction into an aligned and contiguous block of global memory, we perform a preprocessing, prior to copying the input data from host to device. we restructure data employing the structure of array (SOA) instead of array of structure (AOS) in the representation of key and surface points data.

Second, the multiprocessor's 64 KB shared memory/cache [98] is not large enough to buffer the intermediate variables even if it is configured as shared memory with the largest possible size of size 48 KB. Unfortunately, kernel profiling results show the workload for each thread causes a great deal of register spilling that lands in the L1 cache. In this case, any performance improvement should come through the configuration of this memory as L1 cache. Therefore, 64 KB shared memory/cache is configured to a maximum allowable L1 cache size of 48 KB (and 16 KB of shared memory.)

Third, in the design of this work, the read-only data are placed in texture and surface memory spaces to boost the performance of reading accesses. These memory spaces reside in the device memory and are cached in the texture cache.

Fourth, there are multiple iterations of loops in the LRF and histograms computing.

We improve performance by loop unrolling through the use of *#pragma unroll* directive in CUDA `C/C++`. Loop unrolling, of course, results in register pressure, which is alleviated through increase in the size of the L1 cache. There are four large for loops in the parallel SHOT algorithm that are candidates for loop unrolling. Unrolling these four loops can result in 20% speedup performance improvement.

Table 2.3
Speedup of the serial SHOT over the parallel SHOT

Model/Scene Dataset	N_{Surf}	N_{key}	$T_c(s)$	$T_{ga}(s)$	$T_{ga}(s)$	S_{gp}	S_{ga}
Milk Box Model	13704	13704	1.05	0.24	0.21	4.38	5.08
Office Chair Model	18815	18715	3.33	0.37	0.30	9.00	11.07
Stanford Bunny Model	204800	40251	6.51	0.83	0.65	7.84	10.04
Chicken Model	135142	135142	12.42	1.09	0.83	11.39	14.92
Stanford Dragon Model	313260	121550	15.75	1.06	0.88	14.86	17.98
Happy Buddha Model	614560	487951	97.27	2.36	1.81	40.70	53.72
Office Scene	145511	145505	13.67	1.23	0.97	11.12	14.12
Table Scene	307200	66053	12.53	0.91	0.73	13.76	17.06
Five people Scene	307198	241407	41.46	1.71	1.29	24.25	32.25

2.4 Experiment Results and Discussion

In this section, we provide experimental validation of the GPU accelerated G-SHOT algorithm. The serial SHOT algorithm has already been integrated into the PCL. We have integrated the GPU parallel G-SHOT algorithm into the PCL, with experimental platform given in Table 2.1.

Table 2.4
Physical limits for CUDA compute capability 2.0

Parameters	**Size**
CUDA driver version	4.0
CUDA capability version	2.0
Maximum local memory/thread	512 KB
RAM/SM	64KB
Shared memory/SM	16KB
L1 cache/SM	48KB
Shared memory allocation unit size	128
Threads/warp	32
Maximum warps/SM	48
Maximum threads/SM	1536
Maximum thread blocks/SM	8
Number of 32-bit registers/SM	32768
Register allocation unit size	64
Register allocation granularity	warp
Maximum registers/thread	63
Warp allocation granularity	2
Maximum thread block Size	1024

Table 2.2 presents the profiling results of the parallel G-SHOT on the GPU for one typical point cloud scene model from around ten models we studied. The input size in this experiment is 307200 surface points and 206775 key points. The memory copy operations from host to device (H2D Memory Copy) and vice versa (D2H Memory Copy), consume 7% and 2% of the total SHOT descriptor computation time, respectively. Moreover, the cost of preprocessing the data prior to transfer from host to the device is 2%. The computation time is dominated by two GPU kernels which contribute to 89% of overall time.

Table 2.5
Kernel CUDA occupancy

Kernels	**Kernel I**	**Kernel II**
Register	~ 48	~ 48
Shared Memory	0	0
Thread layout	[256,1]	[128,1]
Block layout	$[N_p/256+1]$	$[N_p/128+1]$
Active Thread	512	640
Active Warp	16	20
Active Block	2	5
SM Occupancy	33%	42%

The runtime performances of the serial SHOT on the CPU and two parallel G-SHOT alternatives on the GPU are compared in Table 2.3 for six selected 3D point cloud models and three scenes which are commonly used in computer graphics and computer vision. All these models are available for download from the PCL official website. In this table, we list the number of surface point (N_{surf}), the number of key-points (N_{key}), the runtime of serial SHOT on the CPU (T_c), exact G-SHOT on the GPU (T_{ge}), approximate G-SHOT on the GPU (T_{ga}), the speedup of the exact G-SHOT runtime with respect to the serial SHOT on the CPU (S_{ga}) and the speedup of the approximate G-SHOT runtime with respect to the serial SHOT on the CPU (S_{ga}) respectively. The speedup of the exact parallel SHOT (S_{ge}) marked in red ranges from 4.38 to 40.70, and that of the approximate parallel SHOT (S_{ga}) marked in blue varies from 5.08 to 53.72, with minor degradation in the descriptiveness. Compared to the exact parallel SHOT, the runtime improvement of the approximate algorithm is between 16% and 33%.

The computing time of descriptors for each model or scene depends on multiple factors: number of surface points, number of key points, point density and the contour outline. Larger number of surface points results in larger number of neighbor points to be computed for each key point. Similarly, more key points leads to computation of more descriptors. We only list the sizes of surface point set and key points set in Table 2.3, as they are the primary factors in determining the computational complexity. For the third model (Stanford Bunny Model), although the number of surface points is more then 10 times, and the number of key points are more then 20 times larger then that of the second model (Office Chair Model), the speedup decreases from 9 to 7.8. This is due to the irregular point density around the key points in Stanford Bunny Model. So, the workload of a few CUDA threads is 10 to 100 times larger then other CUDA threads, resulting in a performance bottleneck. This issue will be discussed further in the next sections.

Next, we illustrate the limiting factors influencing the performance of this parallel algorithm across big data-sets with a large number of key and surface points. For this discussion, we refer to information in Table 2.4 for the GPU platform that we used. For the many-core architecture of the GPU, a common measurement criterion is multiprocessor occupancy [101]. It is a measure of the number of parallel program thread warps that are actively running on multiprocessor cores. Ideally, we like the number of active thread warps to be equal to the the maximum number defined by CUDA compute capability version (48 for compute capability 2.0). However, due to hardware resource limitation (shared memory, and register) and the organization of thread blocks (number of threads per block), it cannot reach its peak value. The allocation of hardware resource for the best choice of number of threads per block is shown in Table 2.5. From the data, the occupancy for both Kernel I and II (33% and 42%, respectively) are limited by the number of registers allocated per thread (~ 48) in the program.

2.5 Comparative Performance Evaluation of the Two Parallel G-SHOT Algorithms

In this subsection, we provide experimental results of both exact and approximate parallel G-SHOT algorithms in terms descriptiveness in presences of noise. The quantitative evaluation of G-SHOT algorithms has been carried out in a typical surface matching scenario, where the aim is to establish a correspondence between a set of feature extracted from a scene and those present in a model. We used all the models and scenes listed in Table 2.3 for the evaluation. We created up to 40 scenes by randomly rotating and translating different subsets of the model set in Table 2.3; then. The models with average mesh resolution were corrupted with Gaussian random noise with standard deviation of 10%(σ_1) and 20%(σ_2). The precision-recall curves in Images 2.4 and 2.5 demonstrate that the approximate parallel G-SHOT on GPU loses no more than 5% descriptiveness compared to exact G-SHOT. This comes at the speedup performance improvement by a factor of 15% to 30% for the datasets listed Table 2.3.

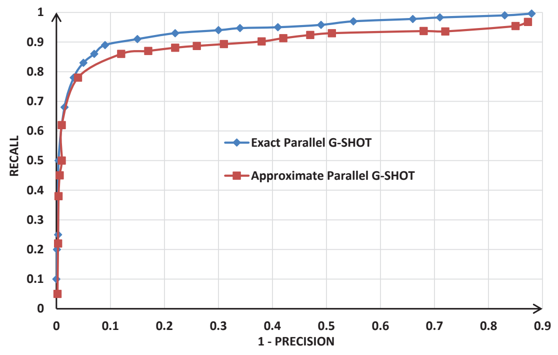


Figure 2.4: Precision-recall curves with noise levels $\sigma_1 = 10\%$

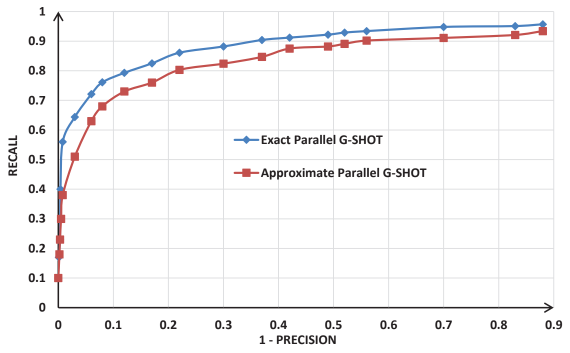


Figure 2.5: Precision-recall curves with noise levels $\sigma_2 = 20\%$

2.6 Conclusion

In this chapter, we have analyzed the latest local descriptor algorithm SHOT and designed two suitable alternative parallel G-SHOT descriptor algorithms on the GPU for large and high density point cloud models and scenes. Both parallel G-SHOT descriptors can achieve high speedup and competitive descriptiveness. The speedup of exact parallel G-SHOT can reach up to 40.7, while the approximate parallel G-SHOT reaches higher to 53.72 with minor matching performance deterioration. We conclude that a low cost many-core parallel programming platform such as GPU is a suitable programming environment for accelerating the computation of 3D point cloud local descriptor. These speedups will benefit the real-time recognition of 3D objects in complicated scenes, for robotics and other machine vision applications.

Algorithm 1 The GPU Accelerated SHOT Algorithm

```
Input:    Key-points coordinates,
          Surface points coordinates,
          Near neighbor points coordinates.
Output:   SHOT descriptor histograms of key-points.

procedure G-SHOT
    $threadsPerBlock \leftarrow 0$;
    $blocksPerGrid \leftarrow 0$;
    [Host] Compute the $k$ NNPs for each key-point, and preprocess the data
    structure of key-point's surface points coordinates.

    [Host→Device] Copy coordinates of surface points and key-points, distances
    and coordinates information of NNPs of each key-point from host to device.

    //compute block and grid layout for Kernel I
    $threadsPerBlock \leftarrow 256$;
    $blocksPerGrid \leftarrow number\_keypoint/256$;
    [Kernel I]This kernel is to compute the local reference frame (LRF) for all
    the key points. One thread is responsible for computing the LRF of one
    key-point. The workload for each thread in this kernel includes the covariance
    matrix computation for the NNP of one key-point, eigenvalue decomposition
    of the covariance matrix, ambiguity resolution.

    [Host] Base on the LRF computing results, remove irregular key points.

    //compute block and grid layout for Kernel II
    $threadsPerBlock \leftarrow 128$;
    $blocksPerGrid \leftarrow number\_keypoint/128$;
    [Kernel II] This kernel is to compute descriptors for all the key points. One
    thread is in charge of calculating the descriptor histogram of one key-point.
    The major overhead contains vector multi-plication,multiple bin value
    interpolations and normalization.

    [Device→Host] Copy all the SHOT descriptor histograms of key-points to
    host.
end procedure
```

Chapter 3

Massive Parallelization of the ANNS on the KD-tree

To overcome the high computing cost associated with high-dimensional digital image descriptor matching, this chapter presents a massively parallel ANNS on the KD-tree on the modern MPA. The proposed algorithm is of comparable quality to the traditional sequential counterpart on the CPU. However, it achieves a high speedup factor of 121 when applied to high-dimensional real-world image descriptor datasets. The algorithm is also studied for factors that impact its performance to obtain the optimal runtime configurations for various datasets. The performance of the proposed parallel ANNS algorithm is also verified on typical 3D image matching scenarios. With the classical local image descriptor SHOT, the parallel image descriptor matching can achieve a speedup of up to 128. The implementation in this work will potentially benefit real-time image descriptor matching in high dimensions.

3.1 Introduction

Point descriptors have become popular for obtaining an image to image correspondence for 3D reconstruction and object recognition. Search for the image point descriptors that are similar to the query, is one of the core techniques in object recognition and surface registration. To increase the feature descriptiveness, the image descriptors, typically, require high dimensionality [58] [59] [60] [3] [61] [62] [63] [64]. However, feature matching in high-dimensional space demands extremely high computational workload.

There has been a large body of work in image descriptor matching, exploring the efficient indexing and the NNS in the point cloud. A brute force P-NNS compares M query points with all the N points in the search set, to obtain their P nearest neighbors. It results in the time complexity of $\mathrm{O}(MN)$ [54]. Search can be made more efficient by using spatial data structures, such as R-tree, B-tree, quad-tree, binary space partitioning (BSP) tree, K-means tree and the KD-tree. These structures subdivide the space containing all the points into smaller spatial regions, where a hierarchy is imposed on the smaller regions in a recursive fashion. The NNS on this hierarchical spatial data structure is generally more efficient since it can prune large portions of target dataset.

In 2D/3D point cloud object recognition and perception, the NNS require fast performance [55] [56]. Unlike the typical applications with single point query [57], the NNS in these point cloud applications involves batch processing a large number of query points to match them against the points in the model object.

The current computing trends favor flexibility of heterogeneous programming model that combines multi-core CPU and many-core GPU. The GPU, as a typical MPA complement through a large number of computing cores, is finding its way into general purpose computing, where fine-grain parallelism is needed. The CUDA and OpenCL standards exemplify this paradigm [102] [103]. In particular, the GPU has been widely employed for fast and real-time implementation of 3D image processing algorithms [3], [104], [105], [106], [2], [4]. The inherent massive-parallelism in the NNS algorithm can be exploited for implementation on any computing platform that supports fine-grain parallelism.

To mitigate the computational workload associated with high-dimensional digital image descriptor matching, in this chapter, we propose a massively parallel approximate P-NNS (P-ANNS) on GPU to accelerate image descriptor matching. The parallel P-ANNS in all stages is fine-grain parallelized for high-dimensional image descriptor datasets. We employ a hybrid technique which combines non-linear and linear search features. For backtracking we use a priority queue which records the distances to the axis aligned bounding box (AABB). In trading off the efforts in tree traversal and backtracking (due to branch divergence) with the effort in linear search (due to leaf node size or the KD-tree height), the technique used in this work finds the near-optimal performance point. Moreover, set an upper bound on the number of backtracks for all query points, to reduce the impact of query outliers. Further, all factors that impact the performance are evaluated for the a near optimal configuration of the P-ANNS. The chapter is organized as follows. Section 3.2 presents the basic concepts of KD-tree construction, the NNS on the KD-tree, and programming model of CUDA. Section 3.3 briefly outlines the related works and highlights the innovations in the proposed work. Section 3.4 presents the design and implementation details of the massively parallel P-ANNS on the GPU. Section 3.5 discusses the experimental results. Section 3.6 concludes this chapter.

3.2 Background

3.2.1 KD-tree

The KD-tree is a hierarchical spatial partitioning data structure for organizing elements (points) in K-dimensional space $\mathbb{R}^K$. It provides the structure to perform

P-NNS, with the average and best time complexity of $\mathrm{O}\big(N \log N\big)$ and $\mathrm{O}\big(N\big)$, respectively, and a space complexity of $\mathrm{O}\big(N\big)$ [90]. The KD-tree partitions the points in the dataset into axis-aligned cells in a hierarchical fashion, with each cell represented by a node in the tree. Starting at the root of KD-tree, the cells are partitioned into two halves by a cutting hyperplane orthogonal to a chosen partition dimension. Typically, the dimension with the maximum span is selected as the partition dimension, and the split value is chosen as the median. Alternatively, the midpoint between the extreme points in that dimension is chosen as the split mark. Each of the two split cells from the root is then recursively split, in the same manner, into other cells. The recursive branching terminates when the number of points that are contained in a cell is no more than a given upper bound. For a KD-tree with leaf nodes containing only a single point, the height is $log_2 N$.

3.2.2 NNS on the KD-tree

In the NNS problem, given are set S of N searchable reference points, set Q of M query points, and a distance metric (*e.g.*, Euclidean, Manhattan, and Mahalanobis) in K dimensions. In a P-NNS, the purpose is to search for the P closest points in S for each point q in Q.

For high-dimensional feature matching, the most promising approximate indexing structures and the NNS algorithms including the KD-Tree, K-means tree, and locality sensitive hashing (LSH), are evaluated in [64].

The P-NNS on KD-tree can be more efficient since large portions of search region are quickly pruned. Starting from the root node, the search moves down the tree using depth first search (DFS). Once the search reaches a leaf node, P points within this node with the shortest distance to the query point are selected as the initial P nearest neighbor candidates. However, the initial candidates may not necessarily be the nearest neighbors to the query point. This requires a further search for the best candidates in the neighborhood of this initial cell. In the standard search, implemented through backtracking, a closer subtree is visited prior to visiting the more distant subtree [107]. To avoid visiting the unproductive nodes, in this work we replace the normal queue with a priority queue.

In high-dimensional space, the efficiency of exact search on the KD-tree is no better than the brute force technique, as most nodes need to be visited [108] [107] [89]. Therefore, practical KD-tree based applications perform the ANNS [89], by simply setting an upper bound on the number of leaf nodes that can be visited. The ANNS can perform an order of magnitude faster, with a relatively small number of errors.

3.3 Related Work and Proposed Innovation

3.3.1 Related Work

There has been a great deal of work on employing parallel architecture to accelerate the NNS in a broad range of areas. These works can be classified into two categories: linear and non-linear searches.

3.3.1.1 Linear NNS

Linear search algorithms use brute force approach in which the distances between the query point in Q and the reference points in S are computed in parallel. Then, a sequence of a parallel scan on the GPU is deployed to locate the point with the shortest distance. The parallel implementations of these linear search algorithms on the GPU are straightforward. The expected time complexity of these parallel algorithms is $O(N^2/C)$, where C is the number of available cores that can execute in parallel. The work in [109] applied a parallel linear search method for photon mapping to locate the nearest photons in the grid and compute the radiance estimation at any surface location in the scene. In [110] points were stored as textures on the GPU memory, and three program fragments were used to compute Manhattan distances, and then perform reductions to find the minimum distance. The work in [111] implemented a bucket sort on the GPU to partition 3D points into cells. A parallel linear search was used to find the best matches for query points in the buckets.

The work in [112] used an octree and proposed to deploy shifted sorting to sort both query and reference points on the GPU. It used Morton codes to order the octree cells [113] [114]. The NNS was implemented through multiple iterations of shifted sorting on the GPU. These works focus on the domain of the computer graphic applications with three-dimensional datasets. The data structures employed were adapted to specific needs of the applications.

The work in [57] used an R-tree and proposed a traversal algorithm for multi-dimensional range query that converts recursive tree node access to sequential access. A parallel brute force scan on the GPU was adopted to achieve fast single query response time. The work in [115] designed a parallel brute force linear search on GPU, with high 96-dimensional synthetic datasets, and a reported speedup of up to 35 compared with the serial counterpart on CPU. However, when applied to real image descriptor datassets with the dimensionality of 27, the speedup is no more than 10. The limit of 96 dimensions, however, is far short of what is needed for high-dimensional image matching descriptors such as scale-invariant feature transform (SIFT) and the SHOT in [58] [60].

3.3.1.2 Non-linear NNS

Non-linear techniques use data structures like the KD-tree to alleviate the search complexity by pruning the dataset. There have been some recent attempts towards the implementation of non-linear NNS on the GPU. The work in [116] built a three-dimensional KD-tree on CPU with the linked list first, and then relocated it to the GPU to accelerate the NNS queries. It targeted parallel ray-tracing on the GPU and reported a speedup factor of 8 over the serial recursive counterpart on the CPU. The work in [105] adopted a similar method for 3D registration problem as [116]. It constructed an array based KD-tree on the CPU first, and then migrated it to the GPU for parallel NNS. Moreover, it used a small fixed length priority queue to reduce the backtracking, thereby, producing approximate query results. In 3D registration experiment with 68229 three-dimensional points, it reported a speedup factor of up to 88 with respect to the serial counterpart. However, the KD-tree node size is only one, which will increase the probability of backtracking, resulting in a loss of the speed performance.

The work in [117] employed buffers to hold query points at leaf nodes of KD-tree before launching three kernels on the GPU to perform parallel brute force searches on the leaf nodes. It yielded significant speedup of up to 50 over the sequential counterpart on the CPU for a 12-dimensional problem. It does an exact NSS backtracking all the way to the root. The work in [118] using a similar procedure to [117], designed a random ball cover (RBC) data structure which subdivides and prunes the dataset. The NNS is completed through two rounds of parallel brute force scans. The first round searches for the candidate subset, and the second round searches for the nearest neighbors in the subset. However, it is hard to ascertain the accuracy of the obtained results.

3.3.2 Proposed Innovation

Building upon the works so far, we propose a massively parallel P-ANNS algorithm for high-dimensional image descriptor matching on the MPA of the GPU. In the algorithm of this work, all stages of the P-ANNS are fine-grain parallelized.

We propose a hybrid P-ANNS technique that combines non-linear and linear search features. We perform the bulk of the work using linear search, by choosing a suitably sized leaf node. In the backtracking the KD-tree we employ a priority queue, adequately sized, to terminate some of the backtracking threads early with no loss of accuracy. In addition to priority queue, we also set an upper limit on the number of backtracks, (similar to other implementations *e.g.*, PCL), to mitigates the loss of speed performance, by sacrificing some accuracy. Through extensive experimentation, we studied the effects of number of points in the leaf node, and the number of backtracks on the speedup performance. It was observed that the best choices for these parameters vary with the size and dimensionality of the datasets.

The hybrid approach designed in this work, with the right choice of parameters, has the advantage of increasing the probability of locating P nearest neighbors, through the use the linear search, in the first candidate node, hence reducing the need for excessive backtracking that leads to the undesirable thread branch divergence. This is made possible through the parallel fabric of the GPU that allows further enlargement of the leaf node size (lowering of the tree height) for a fast linear search.

Moreover, instead of using cell boundaries, we use the AABB, which is much simpler to test for the boundaries in the branching operations in the NNS and reduces the search efforts. Further, to obtain a relatively balanced thread workload, in high-dimensional space, we only use the mean value of the extreme points in the chosen dimension during the KD-tree construction. Further, points in S are all located in the leaf nodes and internal nodes only include the split information. These steps collectively lead to an efficient implementation of the P-ANNS on the MPA of the GPU.

We have applied the implementation to 320-dimensional descriptor SHOT matching for 3D object recognition, where the massive parallelization approach in this work exhibits excellent performance.

3.4 Massive Parallel Implementation

3.4.1 Parallel Algorithm Design

The breath first search (BFS) based construction technique in the previous work [4] builds an array representation of KD-tree on the MPA of the GPU. In this section, we exploit the hierarchical structure of streaming multiprocessor of GPU to design a fast massively parallel P-ANNS using the hybrid linear, non-linear technique with a priority queue. The algorithm performs P-ANNS for all M query points in the query set Q on an N point dataset S in parallel.

The priority queue is very effective mean in early termination of unproductive backtracks on the KD-tree [107] [89]. With the priority queue, starting the DFS at the root node, the children nodes that are not visited are inserted into the queue at each branch point. Upon reaching a leaf node where the P nearest neighbors to the query point are most likely to be found, a brute force linear search is initiated. However, some or all of the nearest points may reside outside of this node. Therefore, a backtracking is performed for the other possible candidate points in the neighboring leaf nodes.

Priority queue as a dynamic data structure, is updated at each visited node. During the backtracking as internal nodes are extracted from the top of the queue (dequeued), the indices of their unvisited children nodes are enqueued at each branch point during the DFS. A node priority is determined based on the closeness of the distance between the query point and the AABB of the cell with unvisited node. These distances are computed easily in an incremental fashion during the branching in DFS [107]. The search terminates when the priority queue is emptied, or as soon as the distance between the query point to the bounding box of the cell extracted from the top of the queue is greater than the shortest distance between the query point and the best candidate so far, or when the backtracking has reached its upper bound. The illustration of the NNS on the KD-tree is shown in Figure 3.1 and 3.2.

The size of the priority queue for single query point is set to be slightly larger than the height of the KD-tree. Setting the size of the priority queue below the height of the tree, potentially increases the level of approximation in the ANNS. A suitable upper bound for the number of backtracks, for the desired level of accuracy, is obtained from the profiling. The following steps explain the iterative P-ANNS on the KD-tree, using the priority queue.

† The workload of one query point is assigned to one CUDA thread. The thread descends to one the leaf nodes through the DFS. At each branch point, the priority queue is updated with the node index and the distance between the query point and AABB of the child node to which it does not branch.

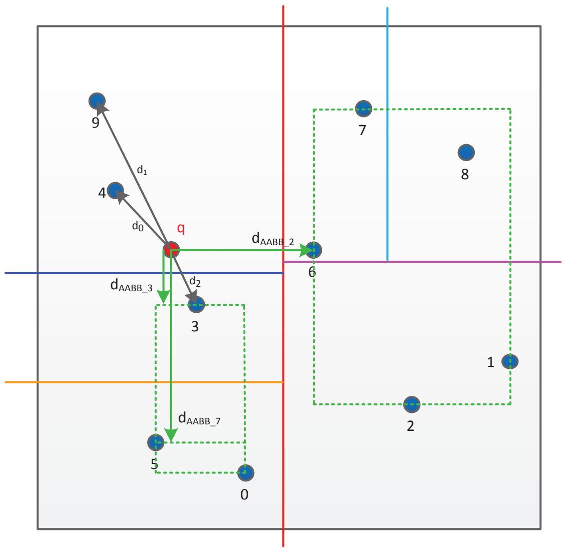


Figure 3.1: Two-dimensional KD-tree point layout and partitions

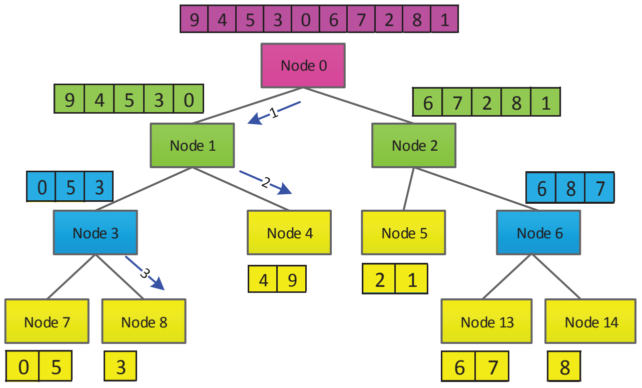


Figure 3.2: NNS on the two-dimensional KD-tree

† On the entry to a leaf node, the P shortest distances between the query point and the points in the leaf node are computed. In addition, the entries in the priority queue whose distance are greater than the current P^{th} shortest distance are purged from the queue, thereby, eliminating the unproductive search paths.

† In the next step backtracking starts by dequeuing the node from the top of

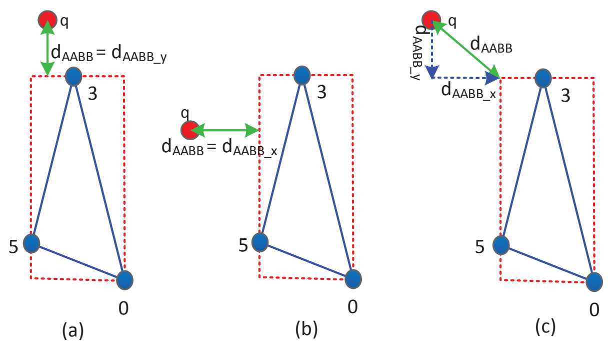


Figure 3.3: Computing of distance from the query point to the AABB

priority queue, and launching another DFS to a leaf node in search for other candidates. As soon as the priority queue becomes empty, or the backtracking search reaches the root node, or the backtracking counter reaches its upper bound limit, the search terminates.

The distance from the query point to the AABB of a node is computed as follows. During KD-tree construction, the minimum and maximum values at each dimension are recorded as part of the AABB array. For a given query point, if its projection along a dimension remains outside the minimum and maximum limits of the given AABB in that dimension, then the orthogonal distance between the query point and the AABB at that dimension is recorded as the partial distance between the query point and the AABB. The squared distance from the query point to the AABB is the accumulation of the partial squares of orthogonal distances in all dimensions. Figure 3.3 demonstrates three scenarios of computation of distance from query point to the AABB in a two-dimensional space as,

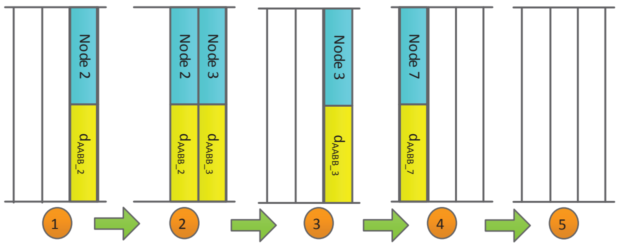


Figure 3.4: Priority queue of the P-NNS

† Figure 3.3 (a), $min_x < q_x < max_x$, and $q_y > max_y$, so $d_{AABB\_x} = 0$, and $d^2_{AABB} = d^2_{AABB\_y}$;

† Figure 3.3 (b), $min_y < q_y < max_y$, and $q_x > max_x$, so $d_{AABB\_y} = 0$, and $d^2_{AABB} = d^2_{AABB\_x}$;

† Figure 3.3 (c), $q_y > max_y$, and $q_x < min_x$, so, $d^2_{AABB} = d^2_{AABB\_x} + d^2_{AABB\_y}$.

Note that by the associating one thread per query point, the P-ANNS for all points in the query set Q are performed concurrently independent of each other. The detail of the implementation is shown in Algorithm 2. We use a `while` loop to convert the recursion in the traditional DFS algorithm to an iterative algorithm that can work better on the GPU. The backtracking is implemented through iterative exchange of three pointers ($leftchild$, $bestchild$ and $otherchild$), and the priority queue.

To better understand the P-ANNS, we demonstrate the key procedures of 2-NNS on the two-dimensional KD-tree with ten points. As depicted in Figures 3.1 and 3.2, the red point marks the query point, and points 1 to 9 are the points in the constructed

KD-tree. Further, each leaf node contains two points at most.

First, the search traverses down to the leaf node 4. As it passes through intermediate nodes (0 and 1), it inserts records of nodes 2 and 3 which are not branched to, and distances to their corresponding AABBs (d_{AABB_2} and d_{AABB_3}) into the priority queue in a sorted manner (Figure 3.4, steps 1 and 2). This is completed through two runs of the `while` loop in Algorithm 2 (lines 30 to 50). Upon reaching the leaf node 4, distances between the red query point and two points in the leaf node (point 4 with distance d_0 and point 9 with distance d_1) are computed and placed in the search result list of 2-NNS (one run of the `while` loop, lines 22 to 28). In the same run of the of `while` loop, entry (*node* 2, d_{AABB_2}) in the priority queue with an AABB distance (d_{AABB_2}) greater than the longest distance (d_1) in the 2-NNS result list is removed (Figure 3.4, step 3). In the next run of the `while` loop (lines 53 to 63), node 3 at the head of the priority queue is extracted as the backtrack start node. At the end of this run, the priority queue becomes empty (Figure 3.4, step 4). In the following iteration of the `while` loop (lines 30 to 50) the search descends to leaf node 8, and entry (*node* 7, d_{AABB_7}) is placed on the priority queue (Figure 3.4, step 5). On the later run of the `while` loop (lines 22 to 28) point 3, the only point in node with the distance $d_2 < d_1$, is placed on the 2-NNS result list, replacing node 4 as the nearest point. Point 4, in turn, replaces point 9. Since the $d_{AABB\_2} > d_0$ the queue is purged (Figure 3.4, step 5). In the final run of the `while` loop (lines 53 to 56) the priority queue is found empty and the search terminates. At the end of the whole process, the 2-NNS produces an ordered result list of $\{3, 4\}$ and a corresponding distance list of $\{d_2, d_0\}$.

3.4.2 Performance Optimization

Performance of parallel P-ANNS algorithm on the GPU is influenced by several factors, including the memory access coalescing in the global memory, bank conflicts in the shared memory, branch divergences, local and global synchronization overhead and the organization of thread blocks [119].

First, as mentioned in [4], the KD-tree is constructed using the SOA. The GPU SIMD (or SPMT) architecture can process vectors more efficiently than the non-linear data structure such as a tree. Second, to ensure that global memory accesses on GPU are coalesced, *i.e.* threads accesses to memory are combined into a single transaction, into an aligned and contiguous block of global memory, we perform a preprocessing, prior to copying the input data from the CPU host to the GPU device. We restructure data employing the SOA instead of the AOS in the representation of data structures. Third, in this design, the read-only data are placed in the texture and surface memories as much as possible to boost the performance of reading accesses. The texture and surface memories reside on the device and are cached in the texture cache, and therefore, presents a better alternative to accessing the global memory.

Algorithm 2 NNS on the KD-tree

```
1: **Input**:   node data structure (parent, child, splits), aabbMin,
2:              aabbMax, query, $P$, $bk\_bound$.
3: **Output**: results data structure
4: **procedure** $P$-NNS-POINTS
5:     allocate memory and copy $M$ query points to GPU;
6:     launch $P$-NNS kernel with one thread for each query point;
7:
8:     [$P$-**NNS Kernel on GPU**]
9:     //initialization for search from root node
10:    $backtrack \leftarrow false$;
11:    $currentnode \leftarrow 0$, $bk\_counter \leftarrow 0$;
12:    create a priority queue $Q_p$;
13:
14:    //iterative search for P-NNS
15:    **while** $true$ **do**
16:        //initialize flags
17:        $leftchild = child[currentnode]$;
18:
19:        //backtracking or travel down to leaf node and examine it
20:        **if** (!$backtrack$) **then**
21:            **if** $leftchild == -1$ **then**
22:                //process leaf node in this branch
23:                compute the distances from query node to all
24:                    points in this leaf node and store $P$ points
25:                    with the 1shortest distances in $results[i]$;
26:                remove entries with distances to their AABBs
27:                    greater than $restults[i].index[P-1]$ in $Q_p$;
28:                $backtrack \leftarrow true$, $bk\_counter{+}{+}$;
29:            **else**
30:                //process intermediate nodes in this branch
31:                //initialize flags
32:                $bestchild = leftchild$;
33:                $otherchild = leftchild$;
34:
35:                //check candidate node in left or right child node
36:                $split \leftarrow splits[currentnode]$;
37:                $delta \leftarrow query[i].[split.dim\_val] - split.split\_val$;
38:                **if** $delta < 0$ **then**
39:                    $otherchild{+}{+}$;
40:                **else**
41:                    $bestchild{+}{+}$;
42:                **end if**
43:
44:                //compute distance from query point to AABB
45:                $d_{AABB} \leftarrow$ compute the distance from query point
46:                     to bounding box AABB of $otherchild$;
47:                $enqueue(Q_p) \leftarrow (otherchild, d_{AABB})$;
48:
```

Algorithm 2 NNS on the KD-tree (Continued)

```
49:                 //prepare for the next search iteration
50:                 currentnode ← bestchild;
51:             end if
52:         else
53:             //search terminate condition
54:             if (length($Q_p$) == 0) ∥ (bk_counter ≥ bk_bound) then
55:                     break;
56:             end if
57:
58:             //extract node with mindist from $Q_p$
59:             (currentnode, mindist) ← dequeue($Q_p$);
60:             //examine candidate sub-tree in next iteration
61:             if (mindist ≤ restults[i].index[P − 1]) then
62:                 backtrack ← false;
63:             end if
64:         end if
65:     end while
66: end procedure
```

Fourth, to improve the performance, we try to reduce the branch conditions in loops in the algorithm. We achieve this by loop unrolling through the use of `#pragma unroll` directive in CUDA. Loop unrolling, of course, results in register pressure, which is alleviated through increase in the size of the L1 cache. There are multiple large `for` loops in the parallel algorithms that are candidates for loop unrolling. Finally, branch divergence effect of parallel P-ANNS on the KD-tree is minimized by making the size of the leaf node/(tree height) sufficiently large/(small).

3.4.3 Memory Usage

The usage of per-thread private memory is rather high resulting in spill over to L1 cache and global memory. The usage of global memory increases rapidly with the dimensionality. With 4 GB DRAM, we were limited to the dimensionality of 512. If the dataset is small, shared memory can be used to hold priority queue. However, for large and high-dimensional datasets we have to use global memory. The memory required to hold the AABB dimensional limits is also high.

3.5 Experiments and Results

In this section, we provide experimental performance validations for the GPU accelerated P-ANNS algorithm [1]. Unlike the previous work in [4], we use real-world image descriptor datasets with a wide range of sizes and dimensionality which were generated from Winder and Brown image dataset [120] [121], as well as datasets with high-dimensional SHOT feature descriptors, extracted from typical point clouds. We perform multiple sets of experiments to measure the speedup performance and accuracy of the P-ANNS (the fraction of the correct neighbors matched by the ANNS compared with that matched by the exact brute force technique). We also explore the performance impact factors including the dimensionality, node size, the upper bound on the number of backtracks, and the number of nearest neighbors for single query point (P).

To compare the algorithm in this work with the previous arts, the serial P-ANNS on

[1] The platform contains a 4-core, 3.2 GHz Intel i7-970 processor, with Ubuntu 12.04 OS, with 1.14 GHz, GeForce GTX 760 GPU with 4 GB RAM with 7 Streaming multiprocessors

KD-Tree algorithm in the PCL is used as the baseline. The ANNS implementation in the PCL uses three pointers, and a single counter to track and bound the number of backtracks that controls the approximation. Further, the leaf node size of the KD-Tree in the serial PCL P-ANNS is one. So the major differences between the proposed parallel P-ANNS on GPU and the serial PCL version are that we place no restriction on the size of leaf nodes, and we use a priority queue. To make a fair speed performance comparison, we modified the serial PCL counterpart to have no restriction on the size of leaf nodes. In the following experiments, the input parameters including dataset size, leaf node size, upper bound on the number of backtracks and the number of nearest neighbors (P) are set to be identical to the serial counterpart in the PCL and the parallel algorithm in this work.

Table 3.1
Comparison with the related works

Work	**NNS**	**DIM**	**Q Size (M)**	**Target Applications**	S_{NNS}	A_{NNS}
[115]	linear, GPU	27	1200	entropy, KL divergence	10	exact
[118]	linear, GPU	78	50k	machine learning	10	inexact
[112]	linear, GPU	3	2M	3D points, photo mapping	7	>90%
[56]	nonlinear, GPU	3	3000	ray tracing, photon mapping	13	–
[117]	nonlinear, GPU	12	10M	machine learning in astronomy	89	exact
[64]	nonlinear, cluster	128	100900	high-dimensional descriptors	10	>90%
this work	hybrid, GPU	512	100900	high-dimensional descriptors	121	>90%

3.5.1 Performance Evaluation on Real-world Image Descriptors

In the experiments with real-world image descriptors, we have used the approach in [120] [121] to generate the points in the datasets of various dimensionality, by random sampling from Trevi Fountain image patch suites[2]. The suites include up to 100900 key point descriptors. We set the query set to be identical to the reference set that was used to construct the KD-tree ($Q = S$). This is justified as in the image matching every query point descriptor has to be matched. To make sure the query accuracy is more than 90%, we have adjusted the upper bound on the number backtracking according to the dataset size and dimensionality.

3.5.1.1 Performance Comparison with Related Works

Table 3.1 provides a brief comparison of this work with the related works from multiple aspects including the dimensionality (DIM), number of query points (M), the ANNS speedup (S_{ANNS}) and the accuracy of the ANNS (A_{ANNS}). As seen, the parallel P-ANNS presented in this work provides the highest performance when applied to

[2]Data from: `http://phototour.cs.washington.edu/patches/`. The data is taken from Photo Tourism reconstructions from Trevi Fountain (Rome). Each dataset consists of a series of corresponding patches, which are obtained by projecting 3D points from Photo Tourism reconstructions back into the original images.

higher dimensions of up to 512 where the dataset size is large. We also achieve an accuracy of above 90%.

3.5.1.2 Performance Comparison Between the Serial and the Parallel Algorithms

Next, we compare the performance of parallel P-ANNS with respect to its serial counterpart. We generated seven N-point datasets and set $Q = S$, with $M = N$ range from 2560, to 100900 to cover a wide range of reference and query dataset sizes. To evaluate the results for a high dimension we set $K = 512$. We set the number of nearest neighbors $P = 4$. Table 3.2 presents the results, where parameters T_{cnns} and T_{gnns} represent the execution times for serial and parallel P-ANNS on the CPU and the GPU, respectively. Parameter S_{nns} denotes the speedup factor, T_{cnns}/T_{gnns} . As shown the speedup of parallel P-ANNS on the KD-tree increases with the data size reaching to a high value of 121.

Table 3.2
High-dimensional (d=512) P-ANNS ($P = 4$) execution times (in ms) and speedups

dataset size	T_{cnns}	T_{gnns}	S_{nns}
2560	9817	554	17.7
5120	37534	773	48.6
10240	121807	2194	55.5
20480	391551	6369	61.5
40960	1201617	18908	63.6
81920	4498158	48014	93.7
100900	8330679	68701	121.3

3.5.1.3 Effect of Divergence

The parallel P-ANNS algorithm in this work is based on *braided parallelism* [122] technique (mix of data and task parallelism), where multiple queries are performed as a set of parallel tasks on GPU, with group of tasks assigned to a thread block and each task assigned to a thread. Each task traverses down the KD-tree to one of the leaf nodes using an independent path. However, tasks are executed on GPU hardware in units of warps (32 threads). Any thread divergence within a warp cost execution clock cycles. However, since the KD-tree is a binary tree, there can only be a maximum of one divergence within a warp at each stage of the tree. Therefore, it costs no more than one clock cycle per warp per KD-tree stage. With a suitable choice of the KD-tree height (leaf node size) the effect of the divergence can be minimized.

To evaluate the impact of divergence, we launched M queries with the same query point, and then perform P-ANNS and measured the runtime. In the parallel 4-ANNS test with 512-dimensional 100900 image descriptors, the warp divergences resulted in the total search execution time increase of 44%. ing processors in a batch sequential mode.

Fortunately, for the image descriptor datasets we studied here, the query points are not indexed randomly. The query points in the same neighborhood have a high probability of being assigned to the same warp. This significantly reduces the effect of the divergence. Furthermore, with a large query size, the workload imbalance between the thread blocks has no effect on the performance. That is because the thread blocks are scheduled to the streaming processors in a batch sequential mode. To study the impact of index ordering of the query points, we performed a random shuffle on all the query points and repeated the test with the same parameters ($M = N = 100900$, $K = 512$, and $P = 4$). The random shuffle caused 42% increase in the search time.

3.5.1.4 Speedup Impact Factors

This section studies the impact on the P-ANNS execution time, from several factors including dataset size, leaf node size, the upper bound on the number of backtracks, and the number of nearest neighbors for a single query (P).

3.5.1.4.1 Effect of the Dataset Size and the Dimensionality Figure 3.5 presents the plots of execution times of the parallel P-ANNS versus the number of query points for several different dimensions K. As seen the rate of growth in the execution time gradually increases with the dataset size. The reason for this is that in the parallel P-ANNS (Algorithm 2), with larger datasets the effects of execution divergence, uneven workload due to backtracking become more severe, leading to degradation in resource utilization and speed performance loss. Further, it was observed that for a given number of query points, the rate of increase in the runtime

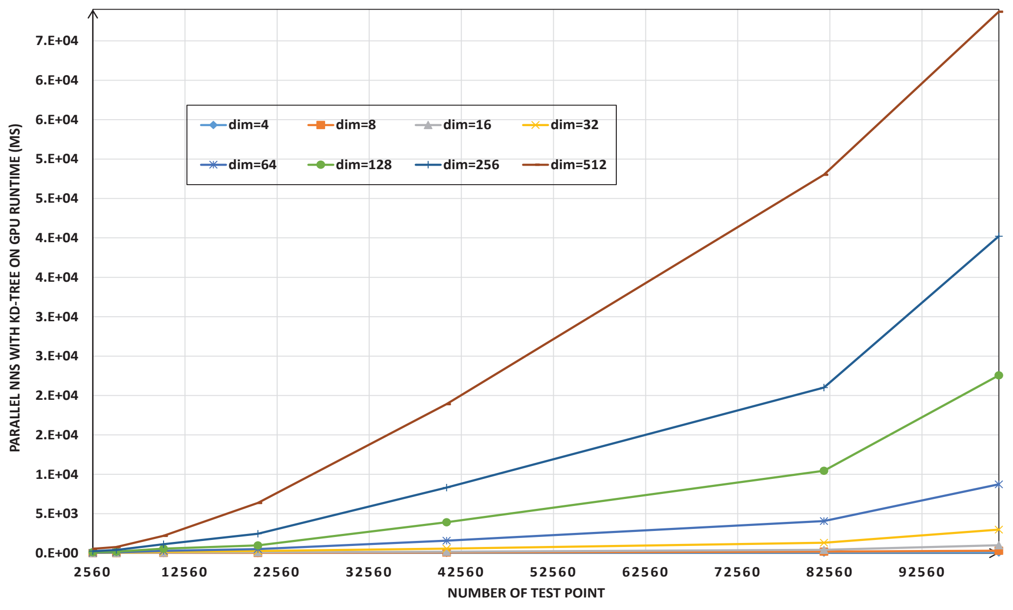


Figure 3.5: Runtime of parallel P-ANNS versus the dataset size with various dimensionality

slowly grows with the dimensionality K. This is because to maintain high accuracy, at higher dimensions we have to increase the number of backtracks. More frequent backtracking, enqueue and dequeue operations at the higher dimensions contribute to the increased rate in the execution time.

3.5.1.4.2 Effect of the Node Size The execution time of the parallel P-ANNS is impacted by the leaf node size. Searches with a large leaf node size are more linear-like, with an increase in the execution time, as the opportunity for pruning the KD-tree structure diminishes. On the other hand, with a leaf node size excessively reduced, the

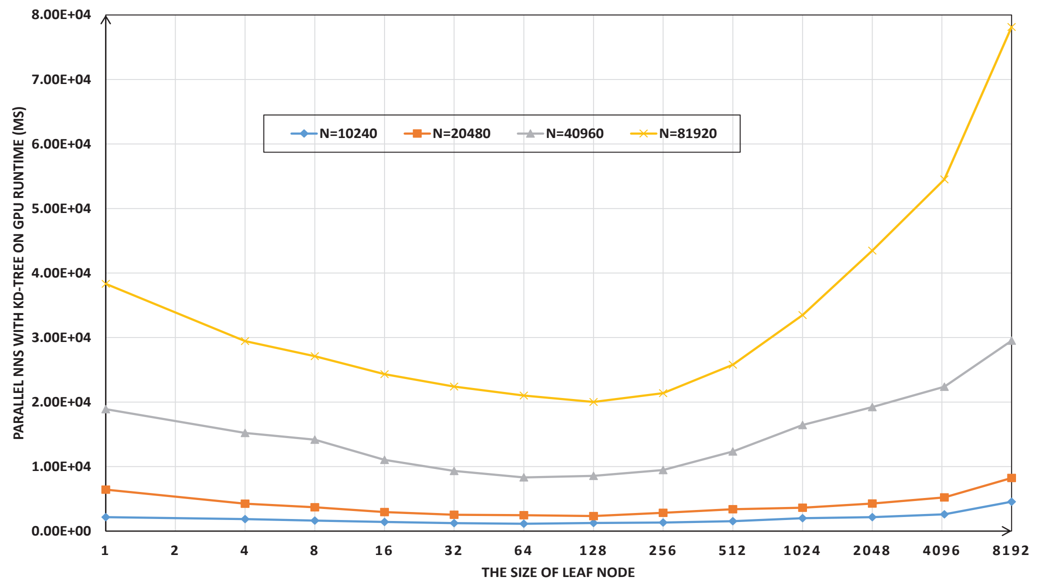


Figure 3.6: Runtime of parallel P-ANNS versus the leaf node size for 256-dimensional KD-tree

search encounters more frequent backtracking, leading to runtime degradation, due to thread divergence. Even though decreasing the tree height reduces the opportunity for pruning, GPU compensates for the increased linear search through parallel processing of the query points in the leaf node. Plots in Figure 3.6 present the effect of the leaf node size on the execution time of the parallel P-ANNS for the 256-dimensional KD-tree, where for the dataset size ranging from 10240 to 81920, the optimum leaf node size is between 64 to 256. Figure 3.7 shows the effect of the leaf node size on the execution time of the serial P-ANNS with the same configuration as the parallel counterparts. Figure 3.8 depicts the leaf node size impact to the speedup.

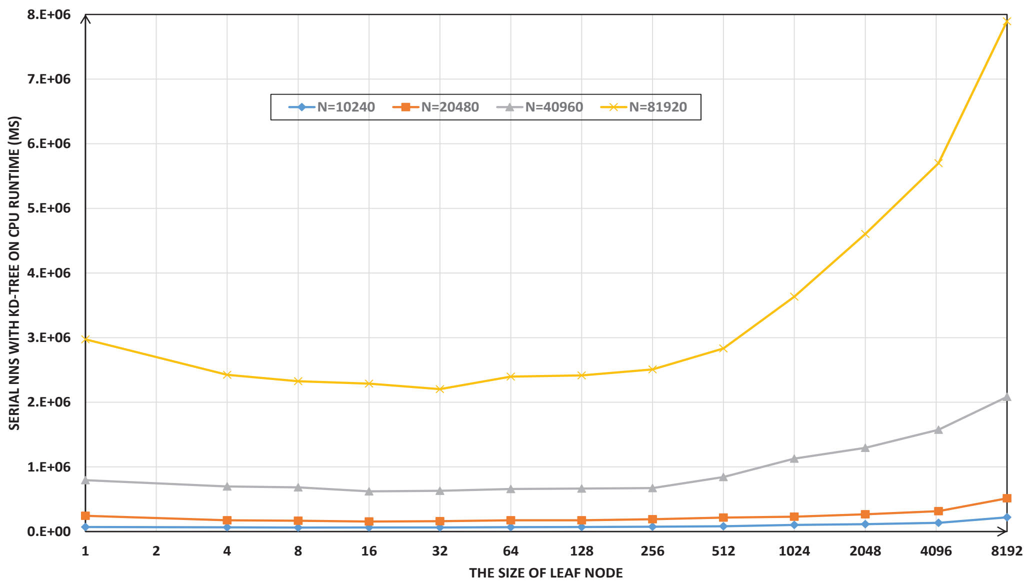


Figure 3.7: Runtime of serial P-ANNS versus the leaf node size for 256-dimensional KD-tree

3.5.1.4.3 Effect of the Backtracking In all experiments so far, the number of backtracks was chosen based on the dimensionality and size of image descriptor set, to achieve an accuracy of more than 90% for P-ANNS. An increase in the number of backtracks leads to higher search accuracy, since more leaf nodes will be inspected. Plots in Figure 3.9 present the effect of the number of backtracks on the execution times of the parallel P-ANNS on the dataset size of 81920 for various dimensionality choices of 4, 16, 64 and 256. The respective saturation points, when the node size is set to 64, are around 50, 200, 1500, and 2000, indicating to reach the same level of accuracy the number of leaf nodes that need to be visited increases with the

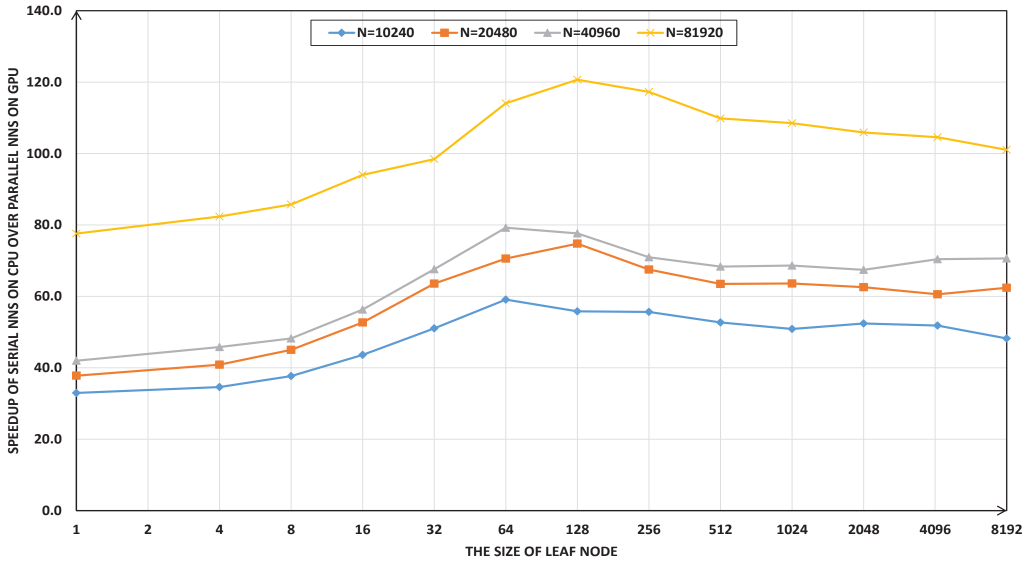


Figure 3.8: Speedup of parallel P-ANNS over the serial counterpart versus the leaf node size for 256-dimensional KD-tree

dimensionality.

3.5.1.4.4 Effect of the Number of Nearest Neighbors of Single Query Point (P) In all experiments so far, the number of nearest neighbors for single query point is set as $P = 4$ in all the P-ANNS tests. To study the effect of P, we evaluated the speed performance of P-ANNS for a range of P values for several 256-dimensional datasets. Plots in Figure 3.10 show that the execution time of parallel P-ANNS increases linearly first, but then slowly with the value of P.

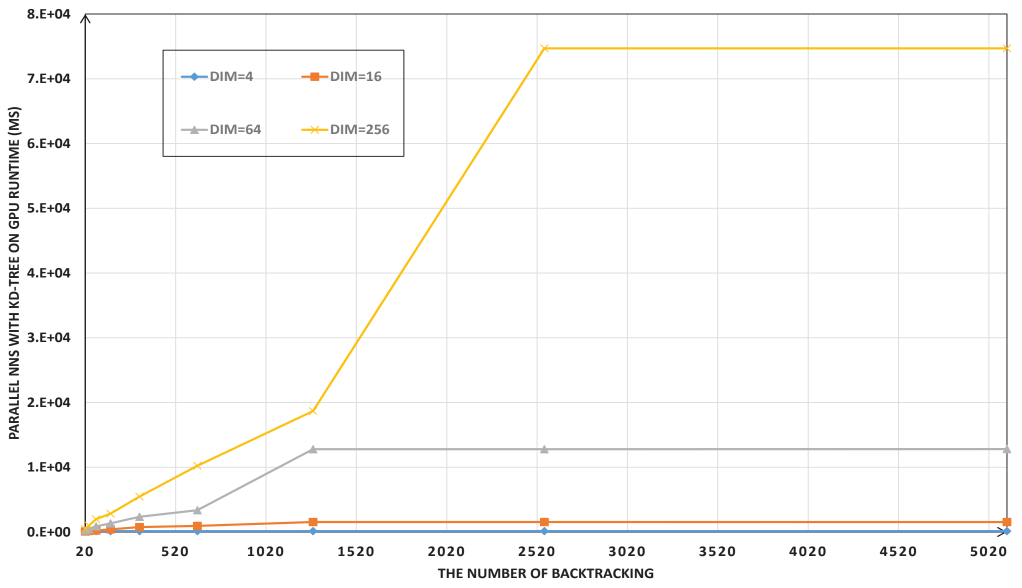


Figure 3.9: Runtime of parallel P-ANNS with 81920 test points versus the number of backtracks, with a node size of 64

3.5.2 Experimental Results for Local Descriptor SHOT Matching

To verify the performance of the massively parallel P-ANNS in high-dimensional space in this work, on real application, we conducted a series of matching experiments on nine real point cloud datasets. The sets chosen include six 3D point cloud models and three scenes, commonly used in computer graphics and computer vision. Datasets are available online for download [55]. The sets are shown in Table 3.3. For each

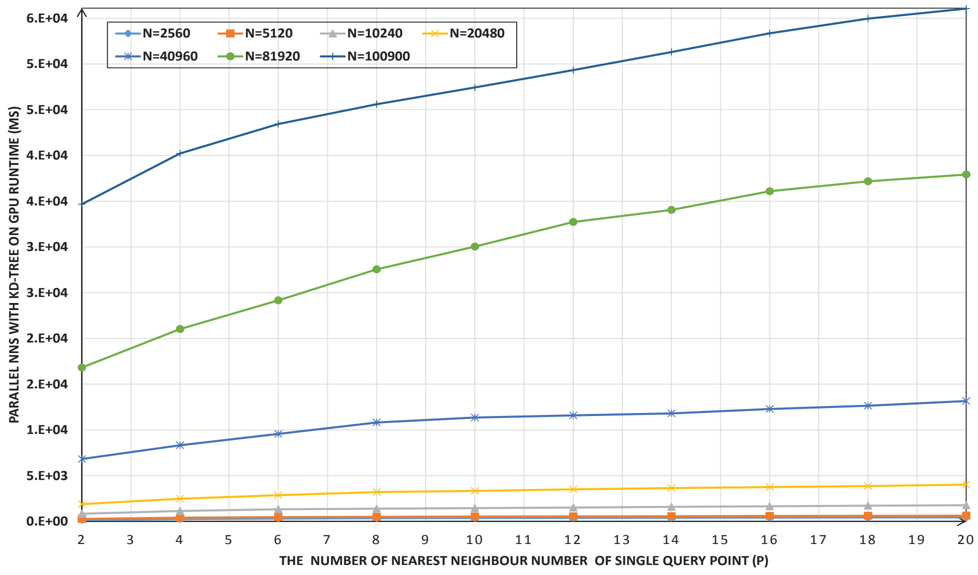


Figure 3.10: Runtime of parallel P-ANNS versus the value of P on a 256-dimensional KD-tree

point cloud dataset, we first sampled out the key points and then computed the SHOT local descriptors for each key point. We chose the SHOT because as a novel 3D object local descriptor it can achieve a good balance between descriptiveness and robustness. With the dimensionality 320 [59] [60], the regular SHOT descriptor also provides a good test case for the evaluation of this work. Next, we constructed a KD-tree with those 320-dimensional descriptors and then searched for $P = 4$ nearest neighbors for each key point on the tree ($Q = S$). The results are presented in Table 3.3, where the parameter N_{key}, denotes the number of key points. Further, T_{csrch} and T_{gsrch} denote the execution times of serial P-ANNS on the CPU and the parallel equivalent on the GPU, respectively. Parameters S_{srch} denotes the speedup factor, T_{csrch}/T_{gsrch}. As seen, the maximum speedup of parallel P-ANNS on the GPU reaches to 128.

Table 3.3
Matching runtime (in ms) and speedup of the parallel algorithm over the serial algorithm

Model/Scene Dataset	N_{key}	T_{csrch}	T_{gsrch}	S_{srch}
Milk Box Model	13704	135679	2993	45.3
Office Chair Model	18715	283344	4484	63.2
Stanford Bunny Model	20446	321337	7441	43.2
Chicken Model	85693	4198083	34021	123.4
Stanford Dragon Model	80047	3340781	30317	110.2
Happy Buddha Model	99614	6253534	61902	101.1
Office Scene	89031	5087316	43419	117.2
Table Scene	66053	1306562	17358	75.3
Five people Scene	91143	5734613	44913	127.7

3.6 Conclusion

In this chapter, we designed a massively parallel P-ANNS on the GPU for high-dimensional image descriptor matching. The proposed algorithm is of comparable quality to the traditional sequential counterpart on the CPU while achieving high speedup performance in a wide range of dimensions. The parallel algorithm was tested on real-world image descriptors datasets with varying dimensionality, as well as classical point cloud descriptors datasets in real applications. The speedup of P-ANNS reaches up 121 with real-world image descriptors with 512 dimensions. For the real application with SHOT descriptor datasets, the corresponding speedup raises up to 128.

Chapter 4

Highly Parallel KD-tree Construction for the BANNS

To overcome the high computational cost associated with the high-dimensional digital image descriptor matching, this chapter presents a set of integrated parallel algorithms for the construction of the KD-tree and the BANNS on the modern MPA. To improve the runtime performance of the ANNS, we propose an efficient sliding window for a parallel BANNS on KD-tree to mitigate the high cost of global memory accesses. When applied to high-dimensional real-world image descriptor datasets, the proposed KD-tree construction and the BANNS algorithms are of comparable quality to the traditional sequential counterparts on the CPU, while outperforming their serial CPU counterparts by speedup factors of up to 17 and 163, respectively. Moreover, we verify the features of the parallel algorithms on typical 3D image matching scenarios.

4.1 Introduction

Point descriptors have become popular for obtaining an image to image correspondence for 3D reconstruction and object recognition. Searching for the image point descriptors that are similar to the query descriptor is one of the core techniques in object recognition and surface registration. To increase the feature descriptiveness, the image descriptors, typically, require high dimensionality [58] [59] [60] [3] [61] [62] [63] [64]. However, feature matching in high dimensions demands extremely high-computational workload.

There has been a large body of research work in image descriptor matching, exploring the efficient indexing and search algorithms for that NNS that find the closest point descriptors to a specified number of query point descriptors. A brute force NNS compares a query point to all the N points in the reference set and results in the time complexity of $\mathrm{O}\big(N^2\big)$ [54]. However, its search time performance can be made more efficient by using spatial data structures, such as R-tree, B-tree, quad-tree, BSP tree, K-Means tree and KD-tree. These data structures subdivide the space containing all the points into smaller spatial regions, where a hierarchy is imposed on each smaller region in a recursive fashion. The NNS on this hierarchical spatial data structure is generally more efficient since it can prune a large portion of target dataset.

In 2D/3D point cloud object recognition and perception, indexing of image descriptors and the NNS of them require fast performance [55] [56]. In these applications, a captured scene changes in a dynamic fashion, and hence, indexing of descriptors in the new scene through the KD-tree construction becomes time critical. Moreover, unlike the typical applications with single point query [57], the NNS in these point cloud applications involves a batch of a large number of query points for matching with the points in the model object.

The current trends favor flexibility of heterogeneous computing model that combines multi-core CPU and many-core GPU. As a typical MPA complement to the CPU, the GPU is finding its way beyond graphical processing into general purpose computing. The CUDA and OpenCL standards exemplify these features [102] [103]. The GPU has been widely employed for fast and real-time implementation of 3D image processing algorithms [3], [104], [105], [106], [2], [4]. The inherent massive-parallelism in the KD-tree construction and the NNS algorithms can be exploited for implementation on any computing platform that supports fine-grain parallelism.

To mitigate the computational workload associated with high-dimensional digital image descriptor matching, in this chapter, we propose two massively parallel algorithms on the GPU to accelerate both KD-tree construction and the BANNS. The chapter is organized as follows. Section 4.2 presents the basic concepts of the KD-tree construction and the NNS. Section 4.3 briefly outlines the related works and highlights the innovations of this work. Section 4.4 presents the design and implementation details of the massive parallelization on the GPU. Section 4.5 describes the performance optimization considerations. Section 4.6 presents experimental results. Section 4.7 concludes this chapter.

4.2 Background

4.2.1 *P*-NNS on the KD-tree

In the NNS, given are set S of N searchable points, set Q of M query points, and a distance metric in K dimensions, such as Euclidean, Manhattan or Mahlanobis distance. In a P-NNS, the purpose is to search for the P closest points in S for each point q in Q. This search can be performed efficiently by using the structure of KD-tree to quickly prune large portions of the search space. Starting from the root node, the search moves down the tree using the DFS. Once the search reaches a leaf node, the point within this node that has the shortest distance to the query point is selected as the initial nearest neighbor candidate. However, the initial candidate may not necessarily be the nearest neighbor to the query point. This requires a further search for the best candidate in the neighborhood of this initial cell. However, in high dimensions, the efficiency of exact search on the KD-tree is not better than the brute force technique, as most nodes need to be visited [108] [107] [89]. To overcome the problem, practical KD-tree based applications use the ANNS algorithms [89] that can perform an order of magnitude faster, often with a relatively small error.

For matching high-dimensional features the works in [64] and [123] evaluate the most promising approximate indexing structures and the ANNS algorithms in literature including KD-Tree, K-means tree and the LSH. It was shown in [64] that the KD-tree is one the most efficient structures that can work well with high-dimensional image feature matching where the descriptors are correlated. However, these tree structures do not work well with randomly distributed descriptors [124] [125].

The work in [107] presents an efficient ANNS algorithm using a balanced KD-tree with a priority queue to avoid unproductive search paths, and find the nearest neighbor point with high probability in an approximate way. A priority queue is used to restrict the search to a fixed number of cells that are most likely to contain the nearest neighbor point [4], [107], [89]. With the priority queue, starting with the root node, the children nodes that are not visited are inserted into the queue at each branch point during the DFS. The priority of a node and its subtree is determined based on the closeness of the distance between the query point and the bounding box of the cell corresponding to the unvisited node. These distances are computed easily in an incremental fashion as each node is visited [4] [107]. After finding the candidate in a leaf node, other nodes in the queue are visited according to their priorities in the queue. The algorithm terminates when the priority queue becomes empty, or as soon as the distance between the query point to the bounding box of the cell corresponding to the point with the highest priority is greater than the shortest distance between the query point and the best candidate so far.

4.3 Related Work and Proposed Innovation

As discussed, KD-tree is one of the most efficient structures for high-dimensional image descriptor matching. Therefore, in this chapter we primarily address the massive parallelization of KD-tree construction and the ANNS for high-dimensional image descriptor matching.

4.3.1 Related Work

Large-scale parallelization of the KD-tree construction, using nonlinear algorithms, requiring tight synchronization among the execution threads, is a challenging task. A further challenge is the requirement for a design that is efficient for the ANNS [118]. A parallel design of 3D KD-tree construction on the GPU in a breadth-first search (BFS) manner was first introduced in [56], and applied to ray-tracer using the dynamic scenes. The input is limited to geometric primitives in a mesh where triangles instead of a general points are the objects of interest. This work proposed a strategy for fine-grain parallelism in the partitioning of nodes at the upper tree levels where their corresponding cells are larger. The approximate split metric used for partitioning combines empty space and median split technique using either the surface area heuristic (SAH) or the voxel volume heuristic (VVH). The speedup factor of this parallel KD-tree construction is about 9 to 13 with respect to the serial counterpart on the CPU.

Heuristic partitions in [56], however, are not suitable for extension to high-dimensional spaces due to the limited shared memory resource, and highly uneven workload among the threads when extended to high dimensions. The work in [126] proposed a fast architecture sensitive tree (FAST) for search. It takes the underlying memory access patterns (cache, TLB and page access) into account for the optimization of the algorithm. The speed improvement, however, is less than 2. Other related works [105], [116] and [117] build the KD-trees on the CPU and transfer the tree to the GPU for the NNS. To the best of our knowledge, there has been no work to deploy the GPU to accelerate high-dimensional KD-tree construction.

4.3.2 Proposed Innovation

Building upon the works so far, we propose highly parallel KD-tree construction and the BANNS algorithms for high-dimensional image descriptor matching on the GPU. All stages of the KD-tree construction and BANNS are fine-grain parallelized for high-dimensional datasets. To obtain a relatively balanced thread workload, in high-dimensional space, we use the midpoint of the extreme points in the chosen dimension during the KD-tree construction. Further, points in S are all located in the leaf nodes and internal nodes only include the split information. The new technique of this work is a hybrid nonlinear and linear search which with the right choice of parameters (size of leaf nodes, and the number of backtracks) has the advantage of increasing the probability of locating the nearest neighbor in a small number of backtracks. Through extensive experimentation, we studied the effects of KD-tree height, and the number of backtracks on the runtime and accuracy performance. It was observed that best choice for these parameters varies with the dimensionality and the size of the datasets. Moreover, to improve the speed performance of P-ANNS, we design a novel sliding window for the parallel P-BANNS on the KD-tree (Sec. 4.4.2), to mitigate the global memory access latency. Further, instead of using cell boundaries, we use axis aligned bounding box (AABB) (Sec. 4.4.1), which is much simpler to test for the boundaries in the branching operations in the NNS. We have applied the massive parallelization to 320-dimensional descriptor SHOT matching in 3D object recognition.

4.4 Massive Parallel Implementation

This section, describes a scalable massively parallel technique to construct a KD-tree from N points in set S, and perform a BANNS for all the M query points in the query set Q. We exploit the hierarchical structure of streaming multiprocessor of the GPU to achieve high speedup. To facilitate the development of the KD-tree construction with minimal programming effort, we use basic general parallel algorithms and data structures from the `Thrust` library that comes with high-level abstraction interfaces. The for a common comparison benchmark we have used the serial construction of KD-tree and the NNS algorithms in the PCL [55].

4.4.1 Parallel Construction of the KD-tree

We employ the BFS to fully exploit the fine-grain parallelism of streaming multiprocessor architecture of the GPU in all stages of construction of the KD-tree. The quality of the proposed parallel KD-tree construction algorithm on the GPU is comparable to the serial counterpart on the CPU. At each BFS step in the parallel implementation in this work, each of the nodes with the same distance from the root spawns a new CUDA thread, with number of threads doubling with each step. Following the conventional construction of KD-tree, the algorithm in this work can be described in the following major steps.

† index all the K-dimensional N points in set S.

† sort points in each dimension, and store the results in the index array of the respective dimension.

† compute the AABB of each intermediate node, its split dimension and value based on the AABB.

† split nodes iteratively in each level of the tree.

The details are shown in Algorithms 3 and 4. Before launching the GPU kernel, we allocate global memory on the GPU for the needed data structures for each node, and each point within a node. For each node, we define the node data structure with `struct` and `union`. Prior to splitting a node, we store indices of the leftmost and rightmost points in the sub-array for the current node in the `Split` structure. After the split, we also store the split dimension and its value.

To avoid using the AOS that have inefficient uncoalesced global memory accesses on the GPU [119], we allocate the following arrays on the GPU global memory; array of points, arrays of dimensional values (one array per dimension), array of pre-allocated parent nodes, children nodes, array of owners and split node indices, array of bounding boxes for all nodes, left and right binary marks for all points, and so on.

Algorithm 3 Construct KD-tree on the GPU

```
1: Input:   Set $S$ with $N$ $K$-dimensional points, and $node\_size$
2:          (maximum number of point in a leaf node)
3: Output: KD-tree on the GPU
4: procedure KD-TREE-CONSTRUCTION-GPU
5:     //allocate global memory on GPU
6:     $M \leftarrow N/node\_size$;
7:     for all $M$ pre-allocated nodes do
8:         allocate global memory for children nodes array;
9:         allocate global memory for parent nodes array;
10:        allocate global memory for current nodes array;
11:        allocate global memory for split array ;
12:        allocate global memory for AABB array;
13:    end for
14:
15:    for all $N$ $points \in S$ do
16:        allocate global memory for all points index array;
17:        allocate global memory for all owners index array;
18:    end for
19:
20:    for all $N$ point in each of the $K$ dimensions do
21:        allocate global memory for points array;
22:        allocate global memory for points index array;
23:        allocate global memory for owner nodes array;
24:        allocate global memory for left and right marks array;
25:    end for
26:
27:    //point preprocessing
28:    assign indices (0 to $N-1$) to $N$ points;
29:    for for each of $K$ dimensions do
30:        assign indices (0 to $N-1$) to $N$ points in index array;
31:        sort $N$ points along the dimension and update index array;
32:    end for
33:
34:    //prepare for the split at root node
35:    compute AABB for root node according to the minimal and
36:     maximal value at each dimension;
37:
38:    //split nodes of KD-tree
39:    NODE-SPLITS;
40: end procedure
```

To benefit from the coalesced global memory accesses we performed the preprocessing through the SOA that significantly improves the efficiency of accessing arrays of set S, temporary points, child nodes, parent nodes, and left and right binary marks. To compute the AABB for each node's cell, we first sort all points along all dimensions.

Sorts are performed by multiple GPU kernel launches (one launch per dimension). After the sorts, the maximum and minimum values in each dimension are stored in the AABB array.

The split operation on the GPU is also implemented with parallel reduction kernels. One CUDA thread works on one node split. The number of nodes involved in the split doubles with each iteration. There are two major steps in each split iteration. The first step as shown in Algorithm 4 (lines 4 to 67) computes the indices of the parent and children of the split node, as well as the split value and dimension. In the event of a node undergoing a split, its associated thread first checks to see if enough memory space has been allocated for the addition of new nodes. If the number of points in the current node falls below the threshold of the number of points in a leaf node, no further split will be undertaken and the node will be marked as a leaf node. Otherwise, the left and right split nodes indices for the current split node i are computed as $(2i + 1)$ and $(2i + 2)$. The `Split` information of current split node is also updated with the new split value and dimension. After the split, all the CUDA threads are synchronized to ensure completion of all split operations at a given level from the root of the tree. Finally, at the end of this step, a check is made to see if any node remains that requires split in this iteration. If no nodes are left the loop breaks out and the procedure terminates. Otherwise, we will prepare for the next split.

The second major step as shown in Algorithm 4 (lines 68 to 139) focuses on the redistribution of points to the children nodes once all the split related information for children and parent nodes are computed. The algorithm launches N CUDA threads to process N groups of K-dimensional points. Each group contains K points with each point from a sorted list of N values in ascending order in one dimension. Each thread projects the dimensional values in each group to the split dimension. Each point in each group is placed in one of the children nodes on the left or right through a comparison of the projected value with the split value. The results are stored in the left and right marks, and the owner arrays for each node. Next, the points in the newly created nodes are sorted in all dimensions through three *Thrust* library functions working on the left and right mark flags; `exclusive scan`, `transform` and `scatter`. These operations are performed on all the K dimensions for all points in the dataset S. The sorted lists in each node are used to compute the AABB on the left and right children for the next iteration of the split.

To better understand the parallel KD-tree construction on the GPU in this work, we demonstrate the key procedure of the first split through an example with ten points in a two-dimensional space. Figure 4.1 presents the arrangement of points in this space. Along the x dimension the list sorted in ascending order is $\{9, 4, 5, 3, 0, 6, 7, 2, 8, 1\}$. Similarly, the sorted list along the y dimension is $\{0, 5, 2, 1, 3, 6, 4, 8, 7, 9\}$. In this example the threshold of the number of points in one node is set to 2. The first split is performed along the x dimension and the split value chosen as the midpoint of the

Algorithm 4 Node split and points re-distribution

```
Input:   indices of points in all $K$ dimensions
Output: parent node, child nodes, split info update
procedure NODES-SPLIT
    //Initialization for first split
    $node\_count \leftarrow 1$;
    $M \leftarrow N/node\_size$;
    $out\_space \leftarrow false$;
    $last\_node\_count \leftarrow 1$;

    while $true$ do
        //launch (last_node_count) threads for this kernel
        [GPU Kernel]split_check
        $split\_enable \leftarrow false$;
        shared $new\_nodes\_to\_add$;
        shared $allocated\_enough$;
        if ($threadIdx == 0$) then
            $new\_nodes\_to\_add \leftarrow 0$;
        end if
        synchronization;

        //check if any node in this round undergoes split, and
        //and if so, compute its children node numbers
        if ($child[threadIdx] == -1$) and (($splits[threadIdx].right$
            $-splits[threadIdx].left) > node\_size$) then
            $split\_enable \leftarrow true$;
            atomicAdd($new\_nodes\_to\_add$, 2);
        end if
        synchronization;

        //check the total number of nodes split sofar and
        //check if enough number of nodes are pre-allocated
        if ($threadIdx == 0$) then
            atomicAdd($node\_count$, $new\_nodes\_to\_add$);
            $allocated\_enough \leftarrow (node\_count < m)$;
            if (!$allocated\_enough$) then
                atomicAdd($node\_count$,$-new\_nodes\_to\_add$);
            end if
        end if
        synchronization;

        //split current node and update split/child/parent info
        if ($split\_enable$) and ($allocated\_enough$) then
            $left \leftarrow 2 * threadIdx + 1$
            $splits[threadIdx].split\_dim \leftarrow$
                compute the split dimension with maximal span;
            $splits[threadIdx].split\_value \leftarrow$
                compute the split value with mean;
            $child[threadIdx] \leftarrow left$;
            $parent[left] \leftarrow threadIdx$;
            $parent[left + 1] \leftarrow threadIdx$;
        end if
```

Algorithm 4 Node split and points re-distribution (Continued)

```
        [**CPU coordinate**]while loop termination and node reallocation.
        //no node deserving split, break out while loop
        if (($last\_node\_count$ == $node\_count$) and ($allocated\_enough$)) then
            break;
        end if
        $last\_node\_count \leftarrow node\_count$;

        //resize pre-allocated node size
        if (!$allocated\_enough$) then
            double pre-allocated node size;
            update new nodes' split/child/parent info;
            $M \leftarrow 2 * M$;
            continue;
        end if
        //launch N threads for this kernel, one thread
        //works for one column of dimensional values;
        [**GPU Kernel**]L_R_mark
        $owner \leftarrow owner[threadIdx]$;
        $leftchild \leftarrow child[owner]$;

        //leaf node does not deserve split
        if ($leftchild$ == -1) then
            return;
        end if

        //compute split dimension and split value
        $split\_dim \leftarrow splits[owner].dim$;
        $split\_value \leftarrow splits[owner].value$;

        //projection of points at each dimension to split dimension
        for $j = 0$ to $K - 1$ do
            $project[j] \leftarrow$ [**projection**]($split\_dim$, $point\_array[j, threadIdx]$);
        end for

        //update owner and left/right marks at each dimension.
        for $i = 0$ to $K - 1$ do
            $L\_R\_mark[i, threadIdx] \leftarrow (project[i] > split\_value)$;
        end for
        $owner[threadIdx] \leftarrow leftchild + L\_R\_mark[0, threadIdx]$;
        synchronization;
```

Algorithm 4 Node split and points re-distribution (Continued)

```
        //launch $N$ threads in the following three Thrust kernels,
          //to sort points in each leaf node.
        [GPU Kernels]distribute_points_to_children_nodes
        for $i = 0$ to $K - 1$ do
            $L\_R\_temp[i, threadIdx]$ ← [exclusive_scan]
                                    ($L\_R\_mark[i, threadIdx]$;)
            $L\_R\_map[i, threadIdx]$ ← [transform]
                                    ($L\_R\_temp[i, threadIdx]$);
            $sorted\_index[i, threadIdx]$ ← [scatters]
                ($L\_R\_map[i, threadIdx], sorted\_index[i, threadIdx]$);
        end for
        $owner[threadIdx]$ ← [scatters]
                    ($L\_R\_mark[0, threadIdx]$, $owner[threadIdx]$);

        //launch $N$ threads in the following Thrust kernels, to
          //compute unique labels and count.
        [GPU Kernel]point_unique_ownership_label
        $number\_of\_labels$ ← [unique_by_key_copy]
                    ($owner$, $count\_index$, $unique\_labels$,
                    $unique\_count\_index$);

        //launch $number\_of\_labels$ number of threads to compute
          //AABB for children nodes after the split.
        [GPU Kernel]compute_AABB_for_children_nodes
        //gather split info for current children nodes
        $index$ ← $unique\_lables[threadIdx]$;
        $left$ ← $unique\_count\_index[threadIdx]$;
        $splits[index].left$ ← $left$;
        if ($threadIdx < (number\_of\_labels - 1)$) then
            $right$ ← $unique\_count\_index[threadIdx + 1]$;
        else
            $right$ ← $N$;
        end if
        $splits[index].right$ ← $right$

        //compute AABB info for current children nodes
        for $k = 0$ to $K - 1$ do
            update $AABB[k]$ of current node according to left and
                right indices;
        end for
    end while
end procedure
```

extreme points in the sorted list along with that dimension. The process is easily understood from Figures 4.1 to 4.4.

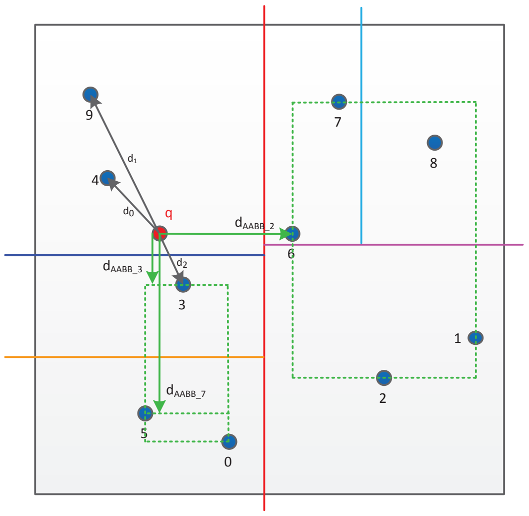


Figure 4.1: Two-dimensional KD-tree point layout and partitions

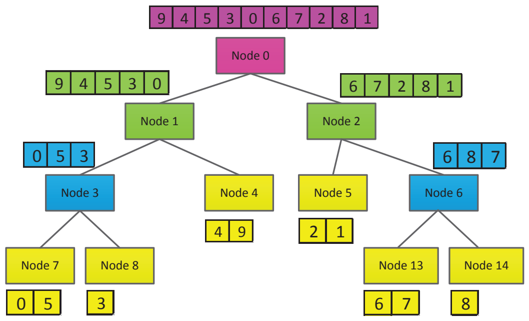


Figure 4.2: Two-dimensional KD-tree construction

Figure 4.3 shows the first iteration of node split. As seen, points are sequenced from 0 to $(N-1)$, with $N = 10$. At the start, points in set S are assigned three sequences as shown in table a. The sorting operations in the ascending order in the x and y dimensions, as shown in table b, are performed through two GPU kernels. So, two sequences in x and y dimensions are, respectively, computed as $\{9, 4, 5, 3, 0, 6, 7, 2, 8, 1\}$ and $\{0, 5, 2, 1, 3, 6, 4, 8, 7, 9\}$. This corresponds to preparation for node split in Algorithm 3 (lines 27 to 32). After sorting the points in two-dimensional sequences, the

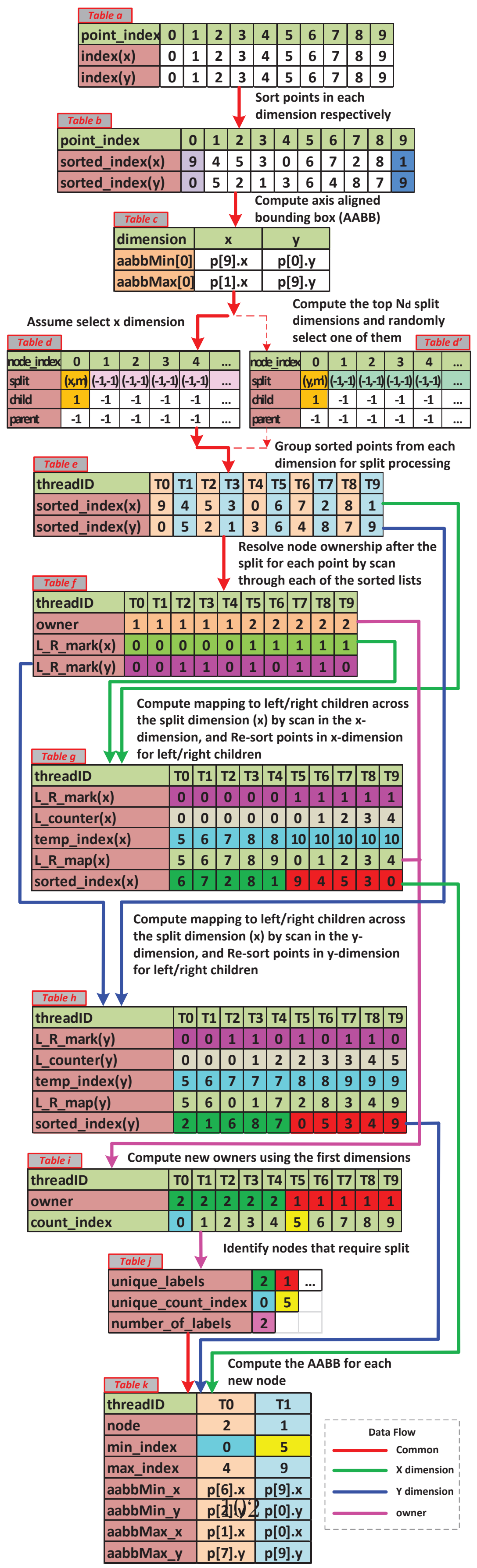


Table a

point_index	0	1	2	3	4	5	6	7	8	9
index(x)	0	1	2	3	4	5	6	7	8	9
index(y)	0	1	2	3	4	5	6	7	8	9

Sort points in each dimension respectively

Table b

point_index	0	1	2	3	4	5	6	7	8	9
sorted_index(x)	9	4	5	3	0	6	7	2	8	1
sorted_index(y)	0	5	2	1	3	6	4	8	7	9

Compute axis aligned bounding box (AABB)

Table c

dimension	x	y
aabbMin[0]	p[9].x	p[0].y
aabbMax[0]	p[1].x	p[9].y

Compute the top N_d split dimensions and randomly select one of them

Assume select x dimension

Table d

node_index	0	1	2	3	4	...
split	(x,m)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	...
child	1	-1	-1	-1	-1	...
parent	-1	-1	-1	-1	-1	...

Table d'

node_index	0	1	2	3	4	...
split	(y,m)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	...
child	1	-1	-1	-1	-1	...
parent	-1	-1	-1	-1	-1	...

Group sorted points from each dimension for split processing

Table e

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
sorted_index(x)	9	4	5	3	0	6	7	2	8	1
sorted_index(y)	0	5	2	1	3	6	4	8	7	9

Resolve node ownership after the split for each point by scan through each of the sorted lists

Table f

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
owner	1	1	1	1	1	2	2	2	2	2
L_R_mark(x)	0	0	0	0	0	1	1	1	1	1
L_R_mark(y)	0	0	1	1	0	1	0	1	1	0

Compute mapping to left/right children across the split dimension (x) by scan in the x-dimension, and Re-sort points in x-dimension for left/right children

Table g

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
L_R_mark(x)	0	0	0	0	0	1	1	1	1	1
L_counter(x)	0	0	0	0	0	0	1	2	3	4
temp_index(x)	5	6	7	8	8	10	10	10	10	10
L_R_map(x)	5	6	7	8	9	0	1	2	3	4
sorted_index(x)	6	7	2	8	1	9	4	5	3	0

Compute mapping to left/right children across the split dimension (x) by scan in the y-dimension, and Re-sort points in y-dimension for left/right children

Table h

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
L_R_mark(y)	0	0	1	1	0	1	0	1	1	0
L_counter(y)	0	0	0	1	2	2	3	3	4	5
temp_index(y)	5	6	7	7	7	8	8	9	9	9
L_R_map(y)	5	6	0	1	7	2	8	3	4	9
sorted_index(y)	2	1	6	8	7	0	5	3	4	9

Compute new owners using the first dimensions

Table i

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
owner	2	2	2	2	2	1	1	1	1	1
count_index	0	1	2	3	4	5	6	7	8	9

Identify nodes that require split

Table j

unique_labels	2	1	...
unique_count_index	0	5	
number_of_labels	2		

Compute the AABB for each new node

Table k

threadID	T0	T1
node	2	1
min_index	0	5
max_index	4	9
aabbMin_x	p[6].x	p[9].x
aabbMin_y	p[2].y	p[0].y
aabbMax_x	p[1].x	p[0].x
aabbMax_y	p[7].y	p[9].y

Figure 4.3: Array based KD-tree construction on the GPU

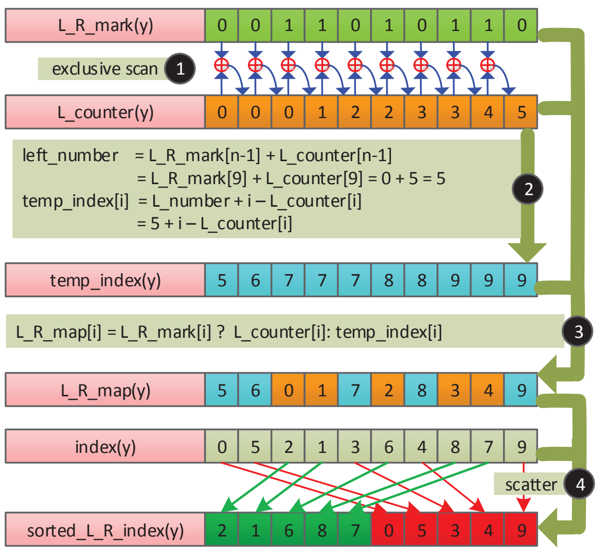


Figure 4.4: Operations in table g and table h in Figure 4.3

AABB of the root node is formed by recording the minimum and maximum values in the sorted arrays in two dimensions from table b. These extreme values defining the AABB are stored in table c. Through the AABB in table c, the split value in the x dimension is easily computed as $(p[9].x + p[1].x)/2)$. The initial computation for the AABB is shown in lines 34 to 36 of Algorithm 3.

Next, the node split is launched through the function `NODE-SPLITS` (line 39 in Algorithm 3). The details of function `NODE-SPLITS` are shown in Algorithm 4. In this algorithm all node splits are performed through multiple iterations of the `while` loop.

`NODE-SPLITS` in Algorithm 4 begins with some parameter initialization (lines 4 to 8). Notable is the pre-allocation of tree nodes (line 6). Prior to the split, a check is made to see if there are nodes at the current level of tree that deserve split, and if enough pre-allocated nodes are available for the pending splits (lines 12 to 39 in Algorithm 4). The results are updated in table d. Note that the function `atomicAdd` returns value of its first argument before the update.

Next step (lines 42 to 51 in Algorithm 4) determines the split dimensions for the nodes at the current level of the tree that undergo split (x dimension for node 0; the only node in the first iteration) and their corresponding mean values in their respective dimensions. In the first iteration, the left child of current node (root) is recorded as 1, with the right child having an implied value of 2. The parent node is fixed as -1. Further, the index for the parent of the current node's children is updated to 0. The entries for other nodes are initialized to -1. Table d only shows the columns for the first five nodes that have been pre-allocated.

After the split, information for nodes at the current level of the tree is updated, and some bookkeeping checks are performed on the CPU to prepare for the point distribution. First, a check is made to see if there are any nodes left to be split. If no node needs splitting, the algorithm will break out of the `while` loop and the KD-tree construction ends (lines 55 to 58). Next, another check is made to see if enough nodes have been pre-allocated, and the size will be doubled if more allocation is required(lines 62 to 67).

After computing the split dimension and value, assignment of points in a node to the descendant nodes is carried out (lines 70 to 95). Table e presents the assignment of workload to each thread in the first iteration. Each thread processes a group of two points (K points in general), from each of the sorted lists in x and y dimensions in table b. For each point in both lists, a check is made to resolve the left or right descendant in the split dimension. Assignments of points to descendant nodes are shown in table f, where each node is identified by its owner and a marker. As an example, the first element in owner array in dimension x is 1, indicating point 9 encountered in the x-sorted list belongs to node 1, along with the split dimension x. The L_R_mark for the first element is similarly set to 0 indicating ownership of the left partition node (node 1).

Next step is the distribution of points to the descendant nodes and sorting of points along all dimensions for each pair of descendant nodes (lines 100 to 108 in Algorithm 4). Three `Thrust` library functions; `exclusive_scan`, `transform` and `scatters` perform these tasks. Table g presents the partitioning and sequencing subsequent to the split in dimension x. The first and second halves of $sorted\_index(x)$ are the sorted sub-sequences in the x dimension for the left and right children after the split. Note that the left and right children and owners have been switched; a fallout from the use of `Thrust` library functions. The variable $count\_index$ assigns a count index to each point in the sorted sub-sequences.

Steps in table h are identical to those in table g, and demonstrate the process of re-sorting the sub-sequences in the left and right split nodes along the non-split dimension y without two explicit sorts. This is achieved through an exclusive scan of row $L\_R\_mark(y)$ in table h and recording the results in $L\_counter(y)$. The last element in $L\_counter$ corresponds to the number of points in the left child ($5 = 4 + 1$). Rows $temp\_index(y)$ and $L\_R\_map(y)$ in table h provide a mapping mechanism for the sort in the y dimension into the two sub-sequences. The details of the mapping through the `scan` and `scatter` operations in `Thrust` is shown in Figure 4.4. The last row in table h lists the indices of sorted lists in the left and right nodes in the y dimension subsequent to the split. Steps associating with tables g or h are repeated for each non-split dimension. Next, using the $L\_R\_mark(y)$ and $sorted\_index(y)$ saved in the previous split iteration, new $sorted\_index(y)$ are computed as shown in table i (lines 109 to 110).

The computations of the AABB for the left and right children of current node are shown in tables j to k. The first row in table j keeps record of unique nodes that have been split so far in the current round. The second row maintains the record of the starting count index in each sequence after the split. The last row keeps the record of the number of nodes generated in this iteration. This part of algorithm is performed through the `Thrust` function `unique_by_key_copy` (lines 114 to 117). Information in table j, the updated $stored\_index(x)$ in table g, and updated $stored\_index(y)$ in table i are used to compute the AABBs of left and right children. Since the nodes have already been sorted in each dimension, we can calculate the bounding box through the first and last elements in each dimension, in each of the left and right children nodes, through a launch of a kernel (lines 121 to 137).

The procedure detailed in tables a to k corresponds to the first split iteration. If any node requires further split, additional iterations are performed through steps in tables d to k. In this example, the node splits terminate at the sixth iteration.

4.4.2 Parallel P-BANNS on the KD-tree

This section explores the hybrid linear, nonlinear, highly parallel BANNS on the KD-tree in this work. We combine the benefits of the BANNS [117], priority queue [107] and brute force search [54] to develop a very fast algorithm with little compromise on the quality of results. For P-BANNS to work, each query point uses the DFS to traverse down the tree until it reaches a leaf node. Once there, the search is made for P nearest neighbor point candidates in the leaf node. Some or all of the P nearest points, however, may reside outside of this leaf node. We, therefore, need to backtrack the tree to search for the other possible candidate points in the neighboring nodes. One possible approach is to invoke multiple P-BANNS queries in parallel in an uncoordinated manner [4] [91]. However, path divergences, uncoalesced memory accesses, and uneven numbers of backtracks among the parallel execution threads severely compromises the performance. We partially alleviate the uneven workloads by letting the threads working on the P-BANNS queries to follow divergent paths to reach the leaf nodes. However, the threads that reach at the same leaf node coordinate to perform time-consuming P-BANNS queries. Figure 4.5 depicts the major steps of the process as described below.

† In Step 1 in Figure 4.5 P-BANNS queries are placed in the leaf nodes' buffers. As soon the buffers are full or all the query points have been scattered into buffers, the distance computation for finding the P nearest neighbors begins. The scattering of query points to the buffers is handled through a CUDA kernel, with each thread responsible for the traversal of a single query point from set Q to a leaf node. As a thread descends towards a leaf node, at each intermediate node, the priority queue is updated with the node index and the distance between the query point and the AABB of the child node which is not branched to.

† Once a buffer in a leaf node is full or all the involved query points have been inserted into buffer, a second CUDA kernel is launched to compute the node-local P nearest neighbors for these query points (Step 2 in Figure 4.5). To compute the P shortest distances we employ a *sliding window*. In the same step, the entries in the priority queue whose distance are greater than the current P^{th} shortest distance will be removed.

† In Step 3 in Figure 4.5, threads perform status check. If the priority queue

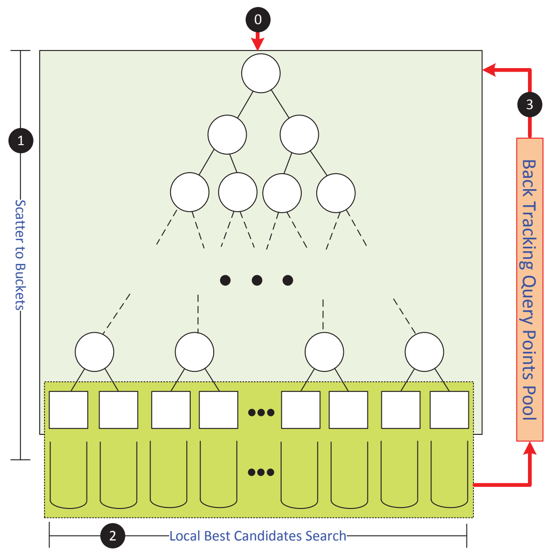


Figure 4.5: Major steps for P-BANNS on the KD-tree

for a query point has become empty, or the number of backtracks has reached its upper bound before reaching the root node, we will terminate the search procedure for this query point. Otherwise, it will be inserted into the backtrack pool for further search in the next iteration.

† As the number of backtracks of the query point from the leaf nodes drops below a threshold we terminate the process. The residual P-BANNS queries are performed through a parallel brute force search CUDA kernel [54]. Since each backtrack iteration requires the launch of two time-consuming CUDA kernels, it is more efficient to do the search for these remaining query points employing the parallel brute force.

The distance from the query point to the AABB of a node is computed as follows.

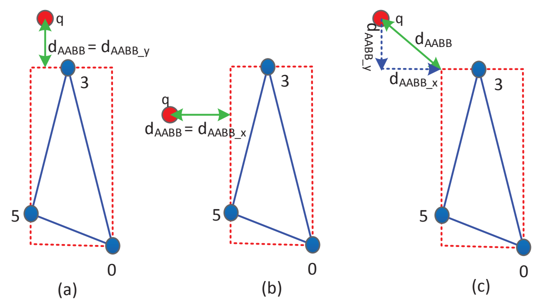


Figure 4.6: Computing of distances from the query point to the AABB: (a) $min_x < q_x < max_x$, and $q_y > max_y$, so $d_{AABB\_x} = 0$, and $d_{AABB} = d_{AABB\_y}$; (b) $min_y < q_y < max_y$, and $q_x > max_x$, so $d_{AABB\_y} = 0$, and $d_{AABB} = d_{AABB\_x}$; (c) $q_y > max_y$, and $q_x < min_x$, so, $d^2_{AABB} = d^2_{AABB\_x} + d^2_{AABB\_y}$.

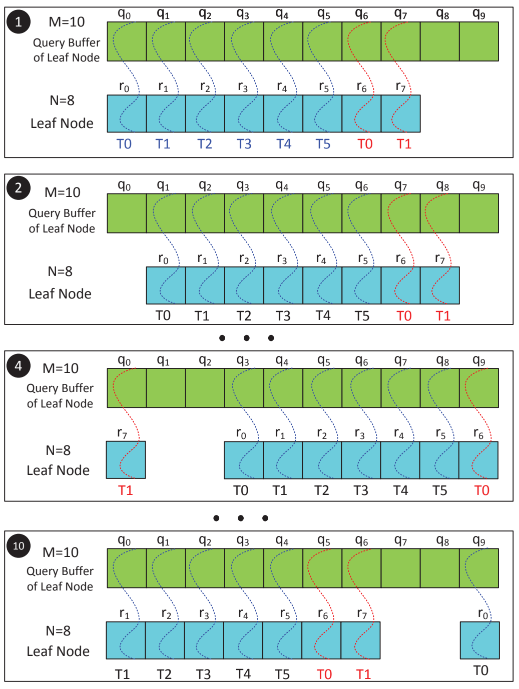


Figure 4.7: Sliding window for distance computing

During KD-tree construction, the minimum and maximum values at each dimension are recorded as part of the AABB array. For a given query point, if its projection along a dimension remains outside the minimum and maximum limits of the given the

AABB in that dimension, the orthogonal distance between the query point and the AABB at that dimension is recorded as partial distance between the query point and the AABB. The squared distance from the query point to the AABB is accumulation of partial squares of orthogonal distances in all dimensions. Figure 4.6 depicts three different scenarios in a two dimensional space.

Unlike the work in [117], the use of sliding window in Step 2 of Figure 4.5 for the computation of the P nearest neighbor distances ensures that all global memory accesses are coalesced. Figure 4.7 depicts the working of the threads in the sliding window where each thread access one pair of query point and leaf node to compute the distance between them. After a sequence of slides, the node-local P shortest distances for all the query points stored in the node buffer are computed. In the depiction of Figure 4.7 with a leaf node with eight points, ten query points in its associated buffer, and assignment of eight threads for the work, the process takes 10 slides of the window.

Algorithm 5 presents the implementation details, where the recursion in the traditional DFS is converted to an iterative `while` loop that works better on the GPU. The backtracks are implemented through iterative exchange of three pointers ($leftchild$, *bestchild* and *otherchild*), and the priority queue. The work is spread across three kernels. In the first kernel, the concurrent threads following divergent paths steer the query points into the leaf node buffers (one query point per thread). The second kernel associates a thread block to one leaf node and uses the sliding window to compute the distances for the P nearest neighbors local to the node. The update of priority queue is also performed by this kernel. When the number of residual query points, left over from the backtrack iterations, falls below a threshold, the third kernel performs a brute force parallel P-BANNS.

4.5 Performance Optimizations

Performance of KD-tree construction and the BANNS algorithms on the GPU are influenced by several factors, including the global memory access coalescing, shared memory bank conflicts, branch divergences, local and global synchronization overhead and the organization of thread blocks [119].

First, as mentioned before, the KD-tree is constructed using the linear SOA. The GPU SPMT architecture can process vectors more efficiently than the nonlinear data structure such as a tree. Second, to ensure that global memory accesses on the GPU are coalesced, *i.e.* threads accesses to memory are combined into a single transaction, into an aligned and contiguous block of global memory, we perform a preprocessing, prior to copying the input data from the CPU host to the GPU device. We restructure data employing the SOA instead of the AOS in the representation

Algorithm 5 *P*-BANNS on the KD-tree

```
1: **Input**:   node data structure (parent, child, splits), aabbMin,
2:              aabbMax, query points, $P$
3: **Output**: results data structure
4: **procedure** $P$-NNS-POINTS
5:     Allocate device memory and copy $M$ query points to the GPU;
6:     Allocation device memory for each leaf node buffer to cache
7:       query points according query point data size;
8:
9:     //initialization for each query point
10:    $backtrack \leftarrow false$;
11:    $currentnode \leftarrow 0$;
12:    create a priority queue $Q_p$;
13:    $query\_pool\_empty \leftarrow false$;
14:
15:    **while** (!$query\_pool\_empty$) **do**
16:        **[GPU Kernel:Scatter Query Points into Leaf Buffers]**
17:        $leftchild \leftarrow child[currentnode]$;
18:        **if** $leftchild == -1$ **then**
19:            $bestchild \leftarrow leftchild$;
20:            $otherchild \leftarrow leftchild$;
21:            //check candidate node in left or right child node
22:            $split \leftarrow splits[currentnode]$;
23:            $delta \leftarrow query[i].[split.dim\_val] - split.split\_val$;
24:            **if** $delta < 0$ **then**
25:                $otherchild{+}{+}$;
26:            **else**
27:                $bestchild{+}{+}$;
28:            **end if**
29:
30:            //compute distance from query point to AABB
31:            $d_{AABB} \leftarrow$ compute the distance from query point to
32:              bounding box AABB of $otherchild$;
33:            $enqueue(Q_p) \leftarrow (otherchild, d_{AABB})$;
34:            //prepare for the next search iteration
35:            $currentnode \leftarrow bestchild$;
36:        **else**
37:            insert the query point index in the buffer associated with
38:              this leaf node;
39:        **end if**
40:
41:        **[CPU Coordinate:Boundary Computing]**
42:        Check the status of each query point in the query pool. If
43:          the priority queue with it is empty or the next search target
44:          is root, remove this points in the pool; then, compute the
45:          index boundary of rest query points in the buffer and train
46:          points in leaf nodes; finally, copy these information to GPU.
```

Algorithm 5 *P*-BANNS on the KD-tree(Continued)

```
47:         [GPU Kernel:Local Candidates Search by Sliding Window]
48:         //Apply Sliding Window Technology to compute local NN.
49:         if $train\_point > query\_point$ then
50:             Query points slide on train points, update the NN for each
51:             query point;
52:         else
53:             Train points slide on query points, update the NN for each
54:             query point;
55:         end if
56:
57:         //extract node with mindist from $Q_p$
58:         $(currentnode, mindist) \leftarrow dequeue(Q_p)$;
59:         //examine candidate sub-tree in next iteration
60:         if $(mindist \leq restults[i].index[P-1])$ then
61:             $backtrack \leftarrow false$;
62:         end if
63:
64:         [GPU Kernel:Residual Query Points Brute Force Search]
65:         if $num\_res\_query\_point > res\_threshold$ then
66:             all the residual query points will scan all the train points
67:               in brute force method;
68:             $query\_pool\_empty \leftarrow true$;
69:         end if
70:     end while
71: end procedure
```

of data structures. Third, breaking of Algorithm 5 into three kernels allows us to implement the most time-consuming part of the algorithm as linear vectors, resulting in very high performance. Forth, in the design of this work, the read-only data are placed in the texture and surface memories to boost the performance of reading accesses. The texture and surface memories reside on the device and are cached in the texture cache, and therefore, presents a better alternative to accessing the global memory. Fifth, to improve the performance, we try to reduce the branch conditions in loops algorithms. We achieve this by loop unrolling through the use of `#pragma unroll` directive in CUDA. Loop unrolling, of course, results in register pressure, which is alleviated through increase in the size of the L1 cache. There are multiple large `for` loops in the parallel algorithms that are candidates for loop unrolling.

4.6 Experiments and Results

In this section, we provide experimental performance validations of the GPU accelerated parallel KD-tree construction and the P-BANNS algorithms[1]. We adopted real-world image descriptor datasets with a wide range of sizes and dimensions from Winder and Brown dataset [120] [121], as well as datasets from high-dimensional SHOT feature descriptors, extracted from typical point clouds [55]. We performed multiple sets of experiments to evaluate the performance of the parallel KD-tree construction and the P-BANNS in this work. We also explored the major P-BANNS performance impact factors, such as the size for reference (S) and query (Q) datasets, dimensionality, tree height, the number of backtracks, and the number P in the P-BANNS.

[1]The experimental platform contains a 4-core, 3.2 GHz Intel $i7-970$ processor, with Ubuntu 12.04 OS, with 1.14 GHz, GeForce GTX 680 GPU with 4 GB RAM with 7 Streaming multiprocessors

Table 4.1
Comparison with related works

Work	**KD-tree**	**NNS**	**Dim (d)**	**Query Size (Q)**	**P-NNS (P)**	**Target Applications**	S_{kdtree}	S_{NNS_kdtree}	S_{NNS_BF}	A_{NNS}
[115]	–	linear, GPU	96	38400	20	entropy, KL divergence	–	137	35	exact
[118]	–	linear, GPU	78	100k	1	machine learning	–	–	21	$\sim$ 90%
[117]	CPU	nonlinear, GPU	12	10M	10	machine learning	–	89	–	exact
[64]	CPU	nonlinear, cluster	128	100k	1	image descriptors	–	–	10	>90%
This Work	GPU	hybrid, GPU	96	38400	20	image descriptors	8	84	93	>90%
			96	100900	4		12	133	136	
			128	100900	4		16	138	86	
			256	100900	4		17	163	95	
			512	100900	4		14.4	128	118	

4.6.1 Evaluation on Real-world Image Descriptors

In these experiments, we have used the library of real-world images to sample image descriptors of different dimensionality from Trevi Fountain image patches [120]. We have employed the approach in [120] [121] to generate real-world image descriptors with varying dimensions. Moreover, the query sets are the same as the reference sets that were used for the construction of the KD-tree ($S = Q$). This is justified as in the image descriptor matching, every point descriptor has to be matched. To make sure the search accuracy is beyond 90%, we have adjusted the number of backtracking steps according to the size and dimensionality of the dataset and the number of query points involved.

4.6.1.1 Performance Comparison with Related Works

Table 4.1 provides a brief comparison of this work with the some related works in higher dimensions. The speedup factors($S_{KD-tree}$) is with respect to the construction runtime using the PCL library [55]. S_{NNS_kdtree} and S_{NNS_BF} are the speedups of parallel algorithm on GPU/cluster over the sequential counter part on the CPU, and over the serial brute force linear search on the CPU, respectively. We adopted the sequential linear brute force the NNS algorithm in the PCL as the common benchmark reference for a fair speedup comparison between various schemes. A_{NNS} represents the accuracy of the P-BANNS.

As shown, the algorithms presented in this work provide the highest performance for the large dataset having a high dimensionality of up to 512, for both KD-tree construction and P-ANNS. We also achieve an accuracy of more than 90% for P-ANNS similar to works in [64] and [118]. Comparing S_{NNS_kdtree} and S_{NNS_BF} of this work, it can be seen that the parallel-by-design P-ANNS algorithm has no efficient counterpart serial implementation on the CPU.

4.6.1.2 Comparison Between the Serial and Parallel Algorithms

Next, we compare the performance of parallel P-BANNS with respect to its serial counterpart. We generated seven target N-point sets S, with $N = 2560$, 5120, 10240, 20480, 40960, 81920, and 100900 to cover a wide range of KD-tree sizes. The maximum dataset size of the real-world image in this work is up to 100900 descriptors. We chose a high dimension case of $K = 512$. Table 4.2 presents the results. Parameters T_{ckdt}/T_{gkdt} represent runtimes for serial and parallel KD-tree construction on CPU/GPU. Parameters T_{cnns}/T_{gnns} represent runtimes for serial and parallel P-ANNS on the CPU/GPU. Parameter S_{kdt}/S_{nns} denotes speedup factor of parallel algorithm on the GPU over serial counterpart on the CPU for the KD-tree construction. As seen the speedup of parallel KD-tree construction and the P-BANNS on the

KD-tree can reach up to 14.4 and 128.5, respectively.

Table 4.2
High-dimensional ($d = 512$) KD-tree and the P-BANNS runtimes (in ms) and speedups

dataset size	**CPU**		**GPU**		**Speedup**	
	T_{ckdt}	T_{cnns}	T_{gkdt}	T_{gnns}	S_{kdt}	S_{nns}
2560	87	13047	92	898	0.9	14.5
5120	185	45642	109	1066	1.7	42.8
10240	350	152025	120	2796	2.9	54.4
20480	843	478258	136	7982	6.2	59.9
40960	1731	1486132	188	22793	9.2	65.2
81920	3884	5590777	299	57521	13.0	97.2
100900	4612	10991734	321	85539	14.4	128.5

4.6.2 Speedup and Accuracy Impact Factors

The runtimes of the construction of KD-tree and the P-BANNS, and accuracy of the P-BANNS are impacted by multiple factors including dimensionality, dataset size, tree height, the number of backtrack iterations and nearest neighbors for a single query point(P). Next, we study the impact of these factors.

4.6.2.1 Effect of the Dataset Size and the Dimensionality

Figure 4.8 plots the runtimes of the construction of KD-tree versus the number of points for several dimensions K for the parallel construction algorithm in this work. The plots demonstrate the power of the MPA where the massive parallelism works

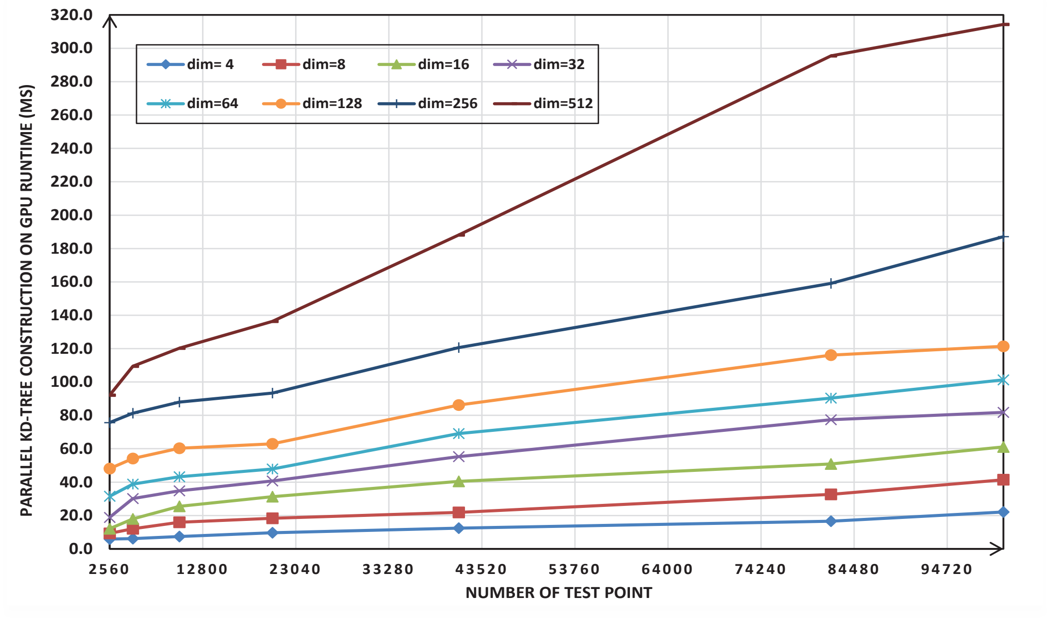


Figure 4.8: Runtime of parallel KD-tree construction versus the number of points in set S for various dimensions

best for large datasets when thread operations are coordinated (*i.e.* `sort, scatter`, *etc*); a 40-fold increase in the data size results in a only 3-fold increase in the runtime. Further, a closer observation of Figure 4.8 reveals that for a given number of reference points the runtime slowly increases with the dimension K (especially for smaller values of K). This is due to optimization steps in Section 4.5; the loop unrolling, using the SOA, and efficient use of the L1 cache.

Figure 4.9 plots the runtimes of the P-BANNS versus the number of points for several dimensions K using the parallel algorithm. The rate of increase in the runtime in the case of P-BANNS is much higher than the KD-tree construction in Figure 4.8. The

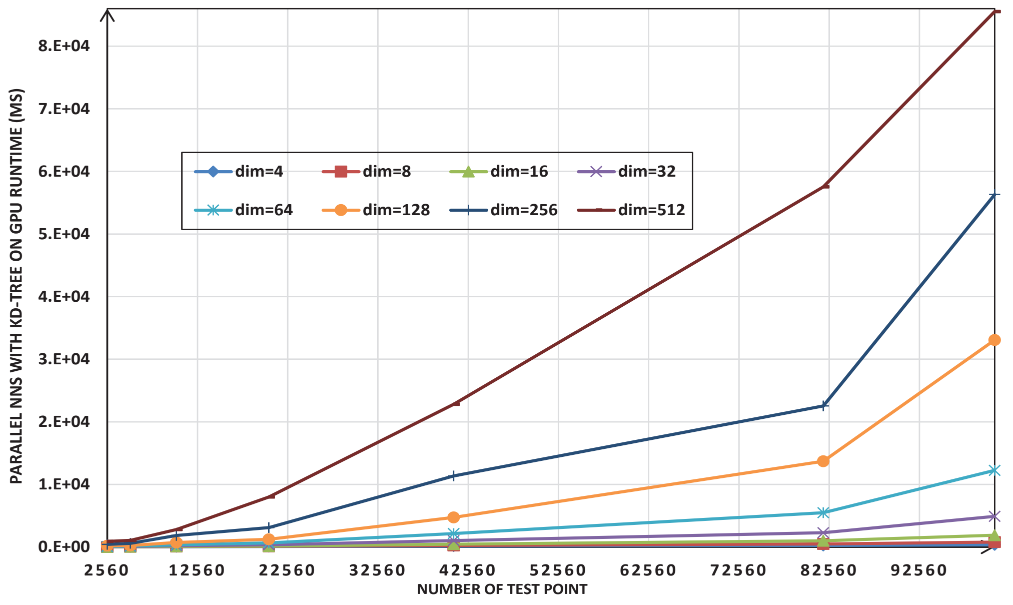


Figure 4.9: Runtime of parallel P-BANNS ($P = 4$) versus the number of points in set S for various dimensions

reason for this is the GPU architecture is ill-suited for nonlinear tree operations.

4.6.2.2 Effect of the KD-tree Height

Plots in Figure 4.10 present the effect of the KD-tree height on the runtime of the parallel P-BANNS for the 256-dimensional KD-tree. As seen, for the number of query points less than 20000, the optimum KD-tree height lies between 6 to 8. The optimum height ranges from 10 to 12 for the number of query points between 40960 to 100900. With a smaller tree height, the P-BANNS is more a linear-like search, resulting in an

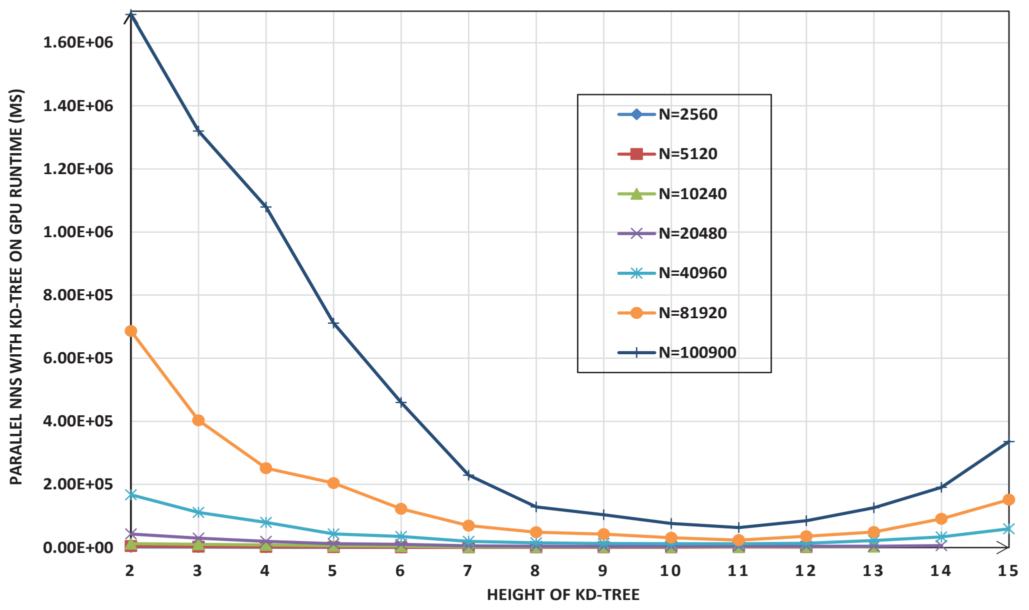


Figure 4.10: Runtime of parallel P-BANNS ($P = 4$) versus the tree height for a 256-dimensional KD-tree, for several dataset sizes S.

increase in the runtime. On the other hand, a large tree height yields higher number of backtracks, degrading the runtime performance. So, in the experiments of this work, the tree heights were chosen in accordance with the image descriptor datasets dimensions and sizes.

4.6.2.3 Effect of the Number of Backtracks

As discussed, an increase in the number backtracks yields a higher search accuracy, as more nodes are inspected. Plots in Figure 4.11 present the effect of the number of backtracks on the runtime of the parallel P-BANNS for the dataset size of 81920 with maximum leaf size of 64, for dimension choices of 4, 16, 64 and 256. The runtimes saturate after a certain number of the backtracks, depending on the dimension of the descriptor in the dataset. For 4, 16, 64 and 256-dimensional KD-tree, the saturation points are around 50, 200, 1500, and 2000, respectively. In the experiments of this work, with the optimum tree height (Figure 4.10), to achieve an accuracy of more than 90% for P-BANNS, the number of backtracks was chosen based on the dimensionality of the descriptor.

Next, we study the relation between the required search accuracy and the corresponding minimum number of backtracks. Plots in Figure 4.12 present the minimum number of required backtracks for search accuracy levels of 50%, 60%, 70%, 80% and 90%, for dataset size of 81920, for dimensions 4, 16, 64, 256 and 512. As expected for the higher dimensions the rate of increase in a number of backtracks with the accuracy is significantly higher.

4.6.2.4 Effect of the Number of Nearest Neighbors of Single Query Point (P)

In all experiments so far, a value of $P = 4$ has been assumed. To study the effect of number of nearest neighbor points, we evaluated the runtime performance of the P-BANNS in a range of P values for several 256-dimensional datasets. From the

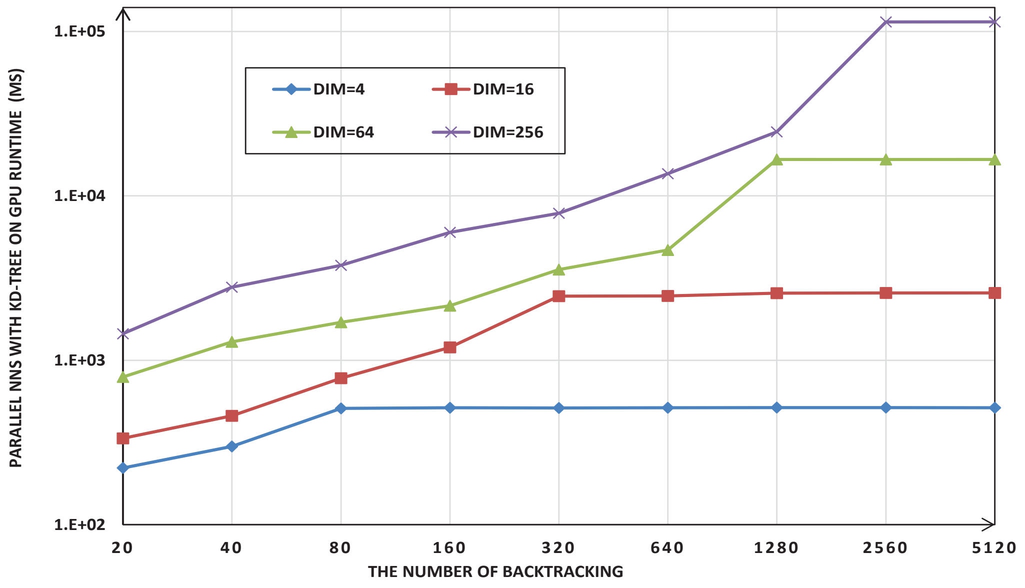


Figure 4.11: Runtime of parallel P-BANNS ($P = 4$) versus the number of backtracks for different dimensions with dataset $S = 81920$

plots in Figure 4.13 the runtime of the algorithm first increases linearly with P, and tends to saturate beyond a certain point. This is due to the effectiveness of sliding window for P-BANNS on KD-tree for larger values of P.

4.6.3 Evaluation of the Sliding Windows

Next, we evaluate the speed performance improvement of the sliding windows for P-BANNS on the KD-tree, with priority queue on the GPU. For evaluation we used the datasets of dimensions 256 and 512, with the number of the points ranging from

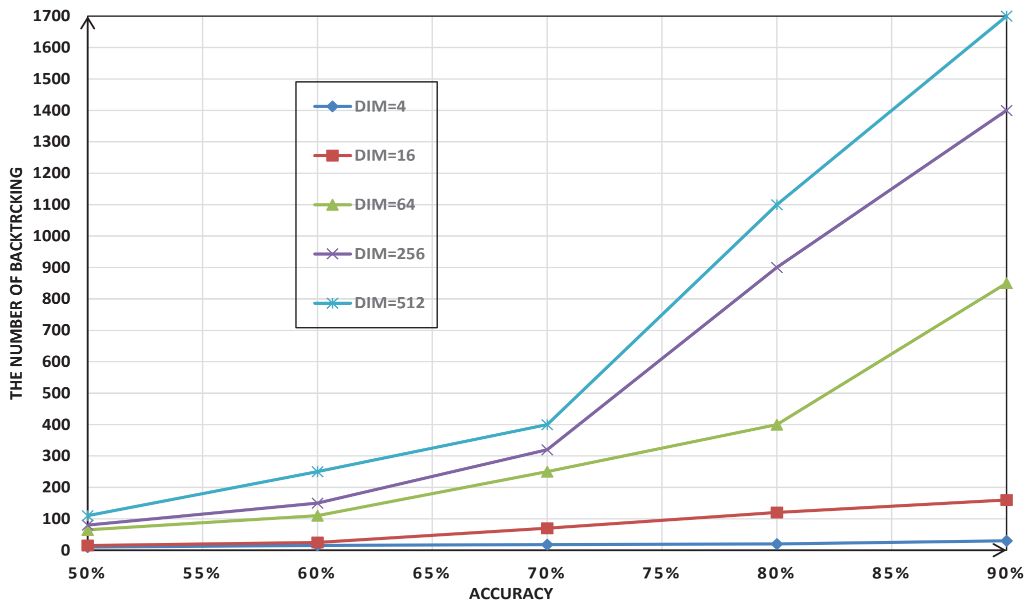


Figure 4.12: Minimum number of backtracks versus the accuracy for five different dimensions for dataset size of $S = 81920$ and $P = 4$

2560 to 100900. The runtimes of parallel P-BANNS ($P = 4$) with and without sliding window are presented in Table 4.3. Compared with the parallel P-BANNS on the GPU without the sliding window, the novel parallel P-BANNS on the GPU in this work can improve the runtime performance by up to 40% while maintaining the same accuracy.

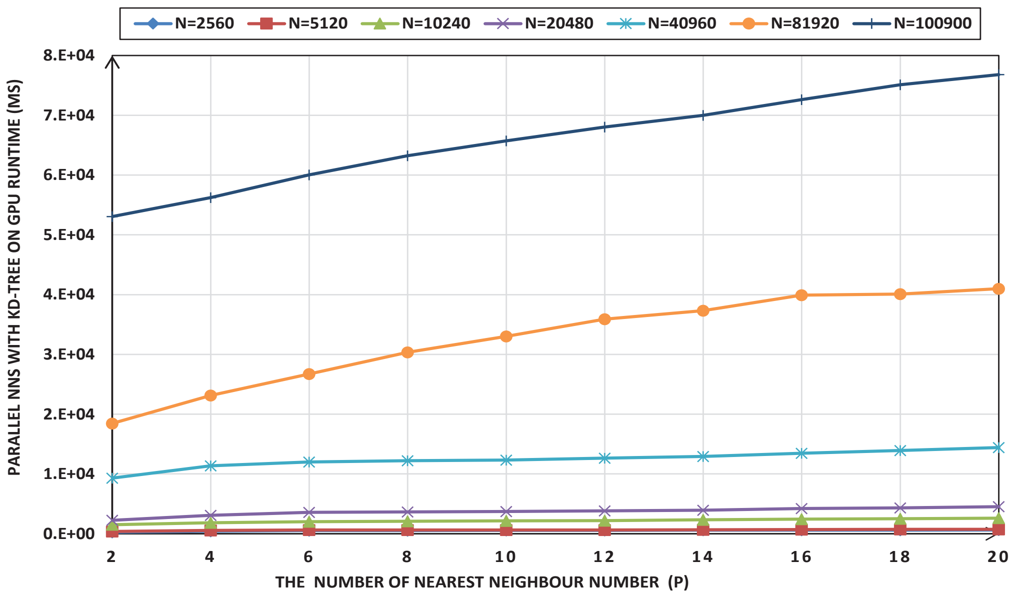


Figure 4.13: Runtime of parallel P-BANNS versus the value of P for a 256-dimensional KD-tree with for several datasets S.

Table 4.3
Sliding window runtime performance ($P = 4$) evaluation

dataset size	**W/O SW (ms)**		**W/ SW (ms)**		**Improvement**	
	d=256	d=512	d=256	d=512	d=256	d=512
2560	453	998	421	898	7.6%	11.1%
5120	641	1224	576	1066	11.3%	14.8%
10240	2061	3545	1829	2796	12.7%	26.8%
20480	3771	10449	3096	7982	21.8%	30.9%
40960	14747	30359	11353	22793	29.9%	33.2%
81920	29472	78586	22540	57521	30.8%	36.6%
100900	75376	120107	56298	85539	33.9%	40.4%

4.6.4 Experiments on the SHOT Matching

To verify the performance of the massively parallel high-dimensional KD-tree construction and the P-BANNS in this work, on another real-world application, we conducted a series of matching experiments on nine real point cloud datasets [55]. The sets chosen include six 3D point cloud models and three scenes, commonly used in computer graphics and computer vision. The sets are shown in Table 4.4. All models and scenes are available online for download [55]. For each point cloud dataset, we first sampled out the key points and then computed SHOT local descriptors for each key point. The SHOT as a novel 3D object local descriptor can achieve a good balance between descriptiveness and robustness. The dimensionality of the regular SHOT descriptor is 320 [59] [60]. Next, we constructed a KD-tree with those 320-dimensional descriptors and then search for $P = 4$ nearest neighbors for each key point on the KD-tree. In other words, the query descriptor sets are the same as the descriptor set for the KD-tree construction ($Q = S$). The results are presented in Table 4.4, where the parameter N_{key} denotes the number of key points. Further, parameters T_{ccnst} and T_{gcnst} denote runtimes of serial KD-tree construction on the CPU and its parallel counterpart on the GPU. Similarly, T_{csrch} and T_{gsrch} denote the runtimes of serial P-BANNS on the CPU and parallel equivalent on the GPU. Parameters S_{cnst} depicts the speedup of the parallel KD-tree construction on the GPU over the serial counterpart on the CPU. Also, S_{srch} demonstrates the speedup of the parallel P-BANNS on the GPU over the serial equivalent on the CPU. As seen, the maximum speedup of KD-tree construction on the GPU reaches to 11. The speedup of P-BANNS reaches to 138.

Table 4.4
Matching runtime (in seconds) and speedup of the parallel over serial algorithms

Model/Scene Dataset	N_{key}	T_{ccnst}	T_{csrch}	T_{gcnst}	T_{gsrch}	S_{cnst}	S_{srch}
Milk Box Model	13704	584	197327	223	8343	2.62	23.65
Office Chair Model	18715	753	410336	254	17267	2.96	23.76
Stanford Bunny Model	20446	781	461359	263	26351	2.96	17.51
Chicken Model	85693	3218	5987176	336	47021	9.58	127.33
Stanford Dragon Model	80047	3115	4791935	328	42317	9.50	113.24
Happy Buddha Model	99614	3914	8891631	351	74331	11.15	119.62
Office Scene	89031	3602	7294792	343	53332	10.50	136.78
Table Scene	66053	2329	1931697	311	31358	7.49	61.60
Five people Scene	91143	3869	8272439	345	59913	11.21	138.07

4.7 Conclusion

This chapter presented the design of high performance parallel construction of KD-tree, and the BANNS on the GPU for high-dimensional image descriptor matching. The proposed algorithms are of comparable quality to the traditional sequential counterparts on the CPU, while achieving high speedup performance in a wide range of dimensions. The massively parallel algorithms presented in this chapter were tested on real-world image descriptors with varying dimensionality, as well as classical point cloud descriptors in real applications. The speedups of KD-tree construction and the BANNS reach up to 17 and 163 with real-world image descriptors with varying dimensionality. For the real application with SHOT descriptor dataset, the corresponding speedups raise up to 11 and 138. The implementations in this work will benefit real-time 3D image registration in low-dimensional spaces, and image descriptor matching employing high-dimensional KD-tree.

Chapter 5

Parallel and Distributed BANNS on the Forest of Randomized KD-trees

Image descriptor matching plays a significant role in object recognition and surface registration. However, the computational cost is extremely high due to the data processing in high dimensional space. To address the computational challenges of real-time processing, we present parallel and distributed algorithms for randomized KD-tree forest construction and the BANNS on a cluster equipped with the MPA devices of the GPU in this chapter. To utilize the GPU cluster platform more fully, we design distributed randomized KD-tree forest for the BANNS to alleviate the backtracking cost on single KD-tree. Additionally, the algorithms are also studied for the performance impact factors to obtain the optimal runtime configurations for various datasets. When applied to high-dimensional real-world image descriptor datasets, the proposed KD-tree forest tree construction and the BANNS algorithms on GPU cluster are of a comparable matching quality to the coarse grain parallel counterparts on the CPU cluster with the MPI, while outperforming counterparts by speedup factors of up to 2.5 and 61.7, respectively. Moreover, we verify the features of the parallel algorithms on typical 3D image matching scenarios. With the classical local image descriptor SHOT datasets, the parallel KD-tree construction and image descriptor matching can achieve up to 2.4 and 62-fold speedups, respectively.

5.1 Introduction

Searching for the image point descriptors that are similar to the query descriptor, is one of the core techniques in object recognition and surface registration. To increase the feature descriptiveness, the image descriptors, typically, require high dimensionality [58] [59] [60] [3] [61] [62] [63] [64]. However, feature matching in high dimensions demands extremely high computational workload.

There has been a large body of research work in image descriptor matching, exploring the efficient indexing and search algorithms for the NNS that find the closest point descriptors to a specified number of query point descriptors. A brute force NNS compares a query point to all the N points in the reference set and results in the time complexity of $O(N^2)$ [54]. However, the NNS can be made to perform more efficiently by using the spatial data structures, such as R-tree, B-tree, quad-tree, BSP tree, K-Means tree and KD-tree. These data structures subdivide the space containing all the points into smaller spatial regions, where a hierarchy is imposed on each smaller region in a recursive fashion. The NNS on this hierarchical spatial data structure is generally more efficient since it can prune a large portion of target dataset. For a good coverage of spatial data structures, the readers can refer to [127].

In this work, we focus on the KD-tree and its variation, generalized binary tree, first introduced in [90], and its several subsequent improvements that use a balanced KD-tree with a priority queue to avoid unproductive search paths [107]. However, when working with high-dimensional image descriptors, there is no known exact NNS algorithm that has acceptable speed performance. To overcome the speed bottleneck, practical applications employ the ANNS algorithms to locate more than 90% percent of the correct neighbors. To improve the performance of the NNS, both in terms of accuracy and speed of search, the regular priority search in a single KD-tree can be extended to the forest of multiple randomized KD-trees [91]. By creating multiple randomized KD-trees from the same dataset and concurrently searching among these trees, the NNS performance can be improved significantly.

The current trends favor flexibility of heterogeneous computing model that combines multi-core CPU and many-core GPU. As a typical MPA complement to the CPU, the GPU is finding its way beyond graphical processing into general purpose computing. The CUDA and OpenCL standards exemplify these features [102] [103]. The GPU has been widely employed for fast and real-time implementation of 3D image processing algorithms [3], [104], [105], [106], [2], [4]. The inherent massive-parallelism in the construction of forest of randomized KD-trees and the NNS algorithms can be exploited for implementation on any computing platform that supports fine-grain parallelism.

In this chapter, we propose a multi-stage massively parallel construction of the forest of randomized KD-tree and parallel and distributed BANNS algorithms for high-dimensional image descriptor matching on a computing cluster via a hierarchy of grain-parallelism through the MPI on a cluster, multi-threading on a multi-core CPU, and massively parallelism on the GPU. The chapter is organized as follows. Section 5.2 presents the background on the NNS and the forest of KD-trees. Section 5.3 briefly outlines the related works. Section 1.1.2.3 presents the design and implementation details of the massively parallel algorithms on the GPU cluster. Section 5.5 explains the performance optimizations and considerations. Section 5.6 describes and discusses the experimental results. Section 5.7 concludes this chapter.

5.2 Background

5.2.1 NNS on the Forest of Randomized KD-trees

The forest of KD-tree consists of multiple KD-trees generated from the same datasets. The trees in the forest are built in a similar manner as the typical KD-tree in [89]. The difference is that the typical KD-tree algorithm splits data along a dimension with the highest variance, while in the randomized KD-tree the split is chosen randomly from the fixed number of dimensions with the highest variance. When query point is close to one of the split hyperplane, its nearest neighbor typically lies with equal probability on either side of the hyperplane. Search in multiple trees in the forest increases the probability of finding the nearest neighbor. Later in the next section, through an example, we will show the value of employing several coordinating randomized KD-trees.

Figure 5.1 illustrates the construction of the two randomized KD-tree trees in a ten-points and two-dimensional space. Along the x dimension, the points sorted by ascending order is $\{9, 4, 5, 3, 0, 6, 7, 2, 8, 1\}$. Similarly, the sorted list of points along the y dimension is $\{0, 5, 2, 1, 3, 6, 4, 8, 7, 9\}$. In this example, the leaf node size is set to be 2. The split dimension is randomly selected between x and y. Figure 5.1 shows

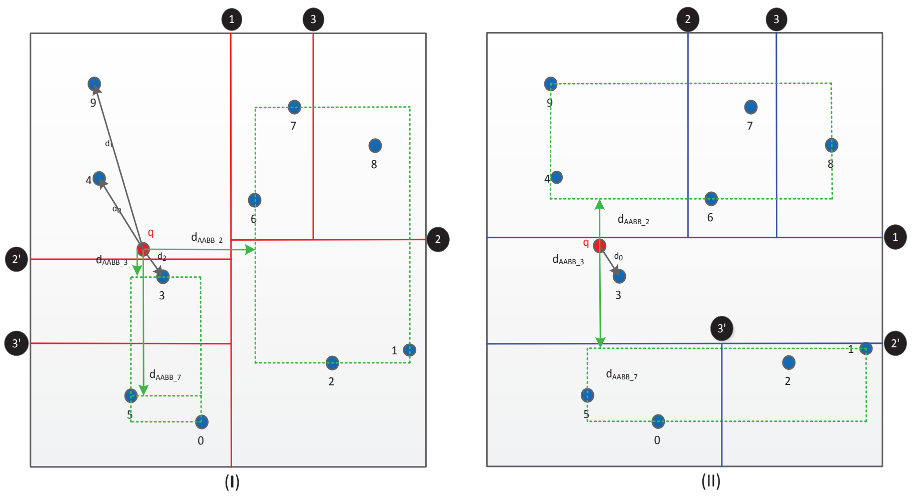


Figure 5.1: Construction of the forest of randomized KD-trees

the advantage of using a KD-tree forest. In Figure 5.1(I) the nearest neighbor is across the decision boundary from the query point, and the NNS needs to backtrack to a neighboring leaf node to do a further search. With the limit on the number of backtracks, the ANNS may miss this nearest neighbor altogether. However, the nearest neighbor in Figure 5.1(II) belongs to the same leaf node, and no backtrack is needed.

5.3 Related Work

There has been a great deal of work on employing parallel architectures to accelerate the KD-tree construction and the NNS. The works so far can be classified into two search categories: linear and non-linear techniques. Linear search algorithms use brute force approach in which the distances between the query point in Q and the target points in S are computed in parallel. Then, the NNS is followed by a parallel scan reduction to find the minimal distance between the query point and the neighbor points. The parallel implementations of these linear search algorithms on the GPU are straightforward. The expected time complexity of these parallel algorithm is $O(N^2/p)$, where p is the number of parallel cores. The work in [109] applied a parallel linear search method to the problem of photon mapping to locate the nearest photons in the grid and compute an estimate of the radiance at any surface location in the scene. In [110], points were stored as textures on the GPU, and three program fragments were used to compute Manhattan distances and perform reductions to find the minimum distance. The work in [111] implemented a bucket sort on the GPU to partition 3D points into cells. A parallel brute force technique was used to search in the cubic cell neighborhood for all the query points. It must be noted that all these referred parallel implementation and optimization algorithms for the NNS are for points in 3D space for applications in graphics. These techniques are not easily adaptable for the implementation of high-dimensional KD-tree construction and the NNS on the MPA such as the GPU. The implementation in [115] extended the brute force linear NNS to higher dimensions of up to 96 using CUDA. It first employs a sort and then applies a binary search to locate the p nearest points. This work reports a speedup of up to 400 compared with the equivalent serial brute force linear NNS on the CPU. The dimensionality limit of 96 is far short of what is needed for matching high-dimensional descriptors such as the SIFT and the SHOT in [58] [128].

Non-linear search techniques use data structures like the KD-tree to reduce the search complexity by pruning the target dataset. However, as mentioned before, neither of the KD-tree construction nor the NNS can be easily parallelized due to non-linear and recursive nature of operations that can not be implemented on the GPU directly. There have been some recent attempts towards the implementation of the KD-tree construction and the NNS on the GPU. The work in [116] builds a 3D KD-tree on the CPU with the linked list first, and then transfers the constructed tree to the GPU for accelerating NNS with parallel streaming processors. It targets parallel ray-tracing on the GPU hardware and reports a speedup factor of eight over the recursive serial implementation on the CPU. The work in [105] adopts a similar method for 3D registration problem. It constructs a 3D array based the KD-tree on the CPU first, and then migrates it to the GPU for the NNS. Moreover, it uses small fixed length priority queue to reduce the backtracking, thereby, producing approximate query results. In this implementation, the 3D registration with $68,229$ points on the GPU is 88 times faster than the serial counterpart on the CPU. The problem with work in [105] is that each leaf node contains only one point. This design results in increased probability of backtracking and speed performance loss. Further, this work only uses a single-element priority queue, an approximation that did not significantly deteriorate the quality of 3D registration. However, this limitation significantly reduces the quality of search if the search is extended to high-dimensional problems.

A parallel implementation of 3D KD-tree construction on the GPU in the BFS scheme was introduced in [56], and applied to ray-tracer using the dynamic scenes. The input is limited to geometric primitives in a mesh where triangles instead of general point as the object of interest. The work proposed a strategy for fine-grained parallelism in the partitioning of large nodes at upper tree levels. The approximate splitting metric used for partitioning combines empty space and median splitting using either the SAH or the VVH. The SAH based KD-tree accelerates ray-tracing, while the VVH KD-tree accelerates the NNS. The VVH based NNS was iterated using a range region search and by increasing the fixed radius of the search region on each iteration. The speedup of this parallel KD-tree construction is about 9 to 13 with respect to the serial counterpart on the CPU in some classic mesh datasets. The optimizations for the parallel KD-tree construction like heuristic partitions in [56], however, are not suitable to be extended to high-dimensional spaces. Extension to higher dimensions creates a highly uneven workload among the threads. Unfortunately, none of the published work on the parallelization of KD-tree goes beyond 3D applications such as photon mapping, ray-tracing and registration.

In order to scale to very large datasets, the work in [64] employed computer cluster and performed the ANNS on the forest of KD-trees using the MPI. The work outperforms most other ones in high-dimensional image matching. Further, the work in [4] presented two parallel algorithms for the construction of KD-tree and the NNS on the single MPA device. The work in [6] on the other hand proposed a parallel ANNS algorithm on a single MPA.

Building upon the work so far, we propose a massively parallel forest of KD-tree construction and the BANNS algorithms for high-dimensional image descriptor matching on the GPU cluster. In the parallel algorithm of this work, all stages of the construction of the forest of KD-tree and the BANNS employ both the coarse-grain and fine-grain parallelism for high dimensionality. Additionally, the BANNS is scalable, in the scene that the target searchable points are randomly sampled and scattered to different cluster nodes and the GPU devices on each node. It should be noted that the random sampling divides the N data points equally across the GPU devices. To order to mitigate the synchronization cost between cluster nodes and GPU devices, the search only synchronizes the global results only after L backtracking iterations, instead of each iteration.

For a balanced thread workload, in high-dimensional space, we only use mean of the points in the chosen dimension in the KD-tree construction. Further, points in S are all located in the leaf nodes. Internal nodes only include the splitting information. In this work, we employ a hybrid technique which combines the both non-linear and linear search features. we limit the leaf node size to more than 64 for high-dimensional spaces. This has the advantage of increasing the probability of locating the nearest neighbor in the first candidate node, through a linear search in the leaf nodes. It has also the advantage of reducing the backtracking. Moreover, in the BANNS search, we limit the size of priority queue to no more than 40, still a relatively large number. These steps lead to an efficient implementation of BANNS on the MPA architecture of the GPU. Further, we use the AABB, instead of cell boundaries, to compute the distance between the query point and the target node, which is much simpler to test the boundaries during the NNS. We have applied the implementation in this work to 320-dimensional descriptor SHOT matching for 3D object recognition. Overall, the approach in this work allows for massive parallelization, where the workloads across the threads have a good balance.

5.4 Massively Parallel Implementations on the GPU Cluster

This section describes a scalable massively parallel technique to construct a forest of KD-tree from N points in set S, and perform a BANNS for all the M query points in the query set Q. We exploit the hierarchical structure of the cluster of GPUs with streaming multiprocessor (Figure 5.2) to achieve high speedup. To facilitate the development of the construction of the forest of the KD-trees with minimal programming effort, we use basic general parallel algorithms and data structures from the `Thrust` library [129]. For a common comparison benchmark, we used the serial

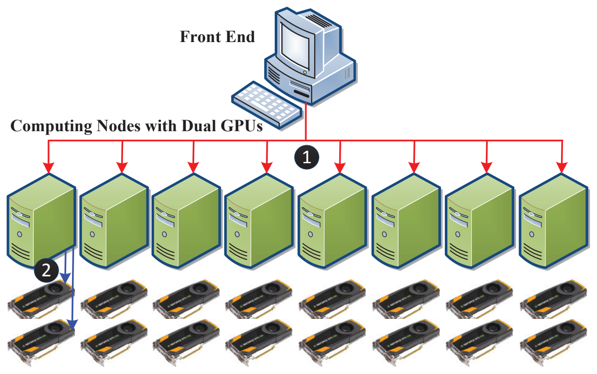


Figure 5.2: MichiganTech GPU computing cluster

KD-tree construction and the NNS algorithms in the PCL [55]

5.4.1 Construction of Forest of Randomized KD-trees

To do a fast construct of the forest of randomized KD-trees, we do a coarse-grain parallel distribution of the workload among the cluster nodes, and then perform a fine-grain parallelism on each node using the MPA fabric of GPUs. Prior to constructing the randomized KD-tree forest, the search target dataset is first randomly sampled and scattered to each node on the cluster first using the MPI, (step 1 in Figure 5.2). On each node, using the multi-core *pthread*, the dataset that has been scattered to a node is further randomly scattered and copied from that node to two corresponding GPUs. Therefore, a single randomized KD-tree is created on each GPU (step 2 in Figure 5.2).

For a single KD-tree construction on each GPU device, we extend the earlier work in [4] by randomly choosing the cut from one of the F dimensions with the highest variance.

In this work, we have set $F = 5$, as it performs well across the dataset we tested in the experiments. With this choice of F we build a forest of 16 randomized KD-trees to the maximum number of the GPU devices in the IVS cluster. We employ the BFS to fully exploit the fine-grained parallelism of the GPU and its streaming multiprocessor architecture in all stages of the KD-tree construction. At each BFS step in the parallel implementation in this work, every KD-tree node with the same tree distance from the root spawns a new CUDA thread, with the number of threads doubling from the preceding step. Following the conventional KD-tree construction, the algorithm in this work can be described in the following major steps.

† index all the K–dimensional subset of S of $N' = N/16$ points.

† sort points in each dimension, and store the results in the index array of the respective dimension.

† compute the AABB of each intermediate node, its split dimension and value based on the AABB.

† split nodes iteratively in each level of the tree.

Algorithm 6 Construction of the Forest of Randomized KD-trees

```
1: Input:   $k$ dimensional points (descriptor)
2: Output: KD-trees on Each GPU device
3: procedure KD-TREE-FOREST CONSTRUCTION
4:     MPI_Init();
5:     MPI_Scatter($k$ dimensional points) to each node;
6:
7:     //on each node two pthreads invoke two randomized KD-Tree
8:     scatter the searchable points on this node into two subsets, invoke pthread_create()
9:     to create two pthreads on two CPU cores; each pthread launches one randomized
10:    KD-tree construction algorithm on one of the two GPU devices attached to a node;
11:
12:    //Single randomized KD-tree construction with single pthread
13:    $N' \leftarrow$ number of points;
14:    $M' \leftarrow N'$/number of points in one leaf;
15:    for all $m$ pre-allocated nodes do
16:        allocate global memory for the children/parent/current nodes array;
17:        allocate global memory for the split array ;
18:        allocate global memory for the AABB array;
19:    end for
20:
21:    for all $N'$ points $\in S'$ do
22:        allocate global memory for all points index array;
23:        allocate global memory for all owners index array;
24:    end for
25:
26:    for all $N'$ point in each of the $K$ dimensions do
27:        allocate global memory for points array;
28:        allocate global memory for points index array;
29:        allocate global memory for owner nodes array;
30:        allocate global memory for left and right marks array;
31:    end for
32:
33:    //point preprocessing
34:    assign indices (0 to $N' - 1$) to $N'$ points;
35:    for each of the $K$ dimensions do
36:        assign indices (0 to $N' - 1$) to $N'$ points in index array;
37:        sort $N$ points along the dimension and update index array;
38:    end for
39:
40:    //prepare for the split at root node
41:    compute AABB for root node according to the minimal and
42:     maximal value at each dimension;
43:
44:    //split nodes of KD-tree
45:    NODE-SPLITS;
46:
47:    //Sync pthreads in single node
48:    two pthreads are synchronized through pthread_join().
49:
50:    MPI_Finalize();
51: end procedure
```

The details of the construction of the forest of KD-trees are shown in Algorithms 6 and 7. For single tree construction, before launching the GPU kernel, we allocate global memory on the GPU for the needed data structures on each of the cluster nodes and each point within a node. For each tree node, we define the node data structure with `struct` and `union`. Prior to splitting a node, we store indices of the leftmost and rightmost points in the sub-array for the current node in the `Split` structure. After the split, we also store the split dimension and its value.

To avoid using the AOS that have inefficient uncoalesced global memory accesses on the GPU [119], we allocate the following arrays on the GPU global memory; the array of points, the arrays of dimensional values (one array per dimension), the array of pre-allocated parent nodes, children nodes, the array of owners and split node indices, the array of bounding boxes for all nodes, left and right binary marks for all points, and so on. To benefit from the coalesced global memory accesses we performed the preprocessing through the SOA that significantly improves the efficiency of accessing arrays of set S, temporary points, children nodes, parent nodes, and left and right binary marks. To compute the AABB for each node's cell, we first sort all points along all dimensions. Sorts are performed by multiple GPU kernel launches (one launch per dimension). After the sorts, the maximum and minimum values in each dimension are stored in the AABB array.

The split operation on the GPU is also implemented with parallel reduction kernels.

One CUDA thread works on one node split. The number of nodes involved in the split doubles with each iteration. There are two major steps in each split iteration. The first step as shown in Algorithm 7 (lines 4 to 67) computes the indices of the parent and children of the split node, as well as the split value and dimension. In the event of a node undergoing a split, its associated thread first checks to see if enough memory space has been allocated for the addition of new nodes. If the number of points in the current node falls below the threshold of the number of points in a leaf node, no further split will be undertaken and the node will be marked as a leaf node. Otherwise, the left and right split nodes indices for the current split node i are computed as $(2i + 1)$ and $(2i + 2)$. The `Split` information of current split node is also updated with the new split value and dimension. After the split, all the CUDA threads are synchronized to ensure completion of all split operations at a given level from the root of the tree. Finally, at the end of this step, a check is made to see if any node remains that requires split in this iteration. If no nodes are left the loop breaks out and the procedure terminates. Otherwise, we will prepare for the next split.

The second major step of single KD-tree construction as shown in Algorithm 7 (lines 69 to 139) focuses on the re-distribution of points to the children nodes once all the split related information for children and parent nodes are computed out. The algorithm launches N CUDA threads to process N' groups of K-dimensional points. Each group contains K points with each point from a sorted list of N values in ascending order in one dimension. Each thread projects the dimensional values in

Algorithm 7 Node split

```
1: Input:   indices of points in all $K$ dimensions
2: Output: parent node, child nodes, split info update
3: procedure NODES-SPLIT
4:     //Initialization for first split
5:     $node\_count \leftarrow 1$;
6:     $M' \leftarrow N'/node\_size$;
7:     $out\_space \leftarrow false$;
8:     $last\_node\_count \leftarrow 1$;
9:
10:    while $true$ do
11:        //launch (last_node_count) threads for this kernel
12:        [GPU Kernel]split_check
13:        $split\_enable \leftarrow false$;
14:        shared $new\_nodes\_to\_add$;
15:        shared $allocated\_enough$;
16:        if ($threadIdx == 0$) then
17:            $new\_nodes\_to\_add \leftarrow 0$;
18:        end if
19:        synchronization;
20:
21:        //check if any node in this round undergoes split, and
22:        //and if so, compute its children node numbers
23:        if ($child[threadIdx] == -1$) and (($splits[threadIdx].right$
24:            $-splits[threadIdx].left) > node\_size$) then
25:            $split\_enable \leftarrow true$;
26:            atomicAdd($new\_nodes\_to\_add$, 2);
27:        end if
28:        synchronization;
29:
30:        //check the total number of nodes split sofar and
31:        //check if enough number of nodes are pre-allocated
32:        if ($threadIdx == 0$) then
33:            atomicAdd($node\_count$, $new\_nodes\_to\_add$);
34:            $allocated\_enough \leftarrow (node\_count < M')$;
35:            if (!$allocated\_enough$) then
36:                atomicAdd($node\_count$,$-new\_nodes\_to\_add$);
37:            end if
38:        end if
39:        synchronization;
40:
41:        //split current node and update split/child/parent info
42:        if ($split\_enable$) and ($allocated\_enough$) then
43:            $left \leftarrow 2 * threadIdx + 1$
44:            $splits[threadIdx].split\_dim \leftarrow$ randomly select one
45:                of the top five dimensions with maximal span;
46:            $splits[threadIdx].split\_value \leftarrow$
47:                compute the split value with mean;
48:            $child[threadIdx] \leftarrow left$;
49:            $parent[left] \leftarrow threadIdx$;
50:            $parent[left + 1] \leftarrow threadIdx$;
51:        end if
```

each group to the split dimension. Each point in each group is placed in one of the children nodes on the left or right through a comparison of the projected value with the split value. The results are stored in the left and right marks, and the owner arrays for each node. Next, the points in the newly created node are sorted in all dimensions through three *Thrust* library functions working on the left and right mark flags; `exclusive scan`, `transform` and `scatter`. These operations are performed on all the K dimensions for all points in the dataset S. The sorted lists in each node are used to compute the AABB of the left and right children for the next iteration of the split.

With reference to Figure 5.1, Figure 5.3 shows the first iteration of node split. As seen, points are sequenced from 0 to $(N' - 1)$, with $N' = 10$. At the start, points in set S are assigned three sequences as shown in table a. The sorting operations in the ascending order in the x and y dimensions, as shown in table b, are performed through two GPU kernels. So, two sequences in x and y dimensions are, respectively, computed as $\{9, 4, 5, 3, 0, 6, 7, 2, 8, 1\}$ and $\{0, 5, 2, 1, 3, 6, 4, 8, 7, 9\}$. This corresponds to preparation for node split in Algorithm 6 (lines 27 to 32). After sorting the points in two-dimensional sequences, the AABB of the root node is formed by recording the minimum and maximum values in the sorted arrays in two dimensions from table b. These extreme values defining the AABB are stored in table c. Through the AABB in table c, the split value in the x dimension is easily computed as $(p[9].x + p[1].x)/2)$. The initial computation for the AABB is shown in lines 34 to 36 of Algorithm **??**.

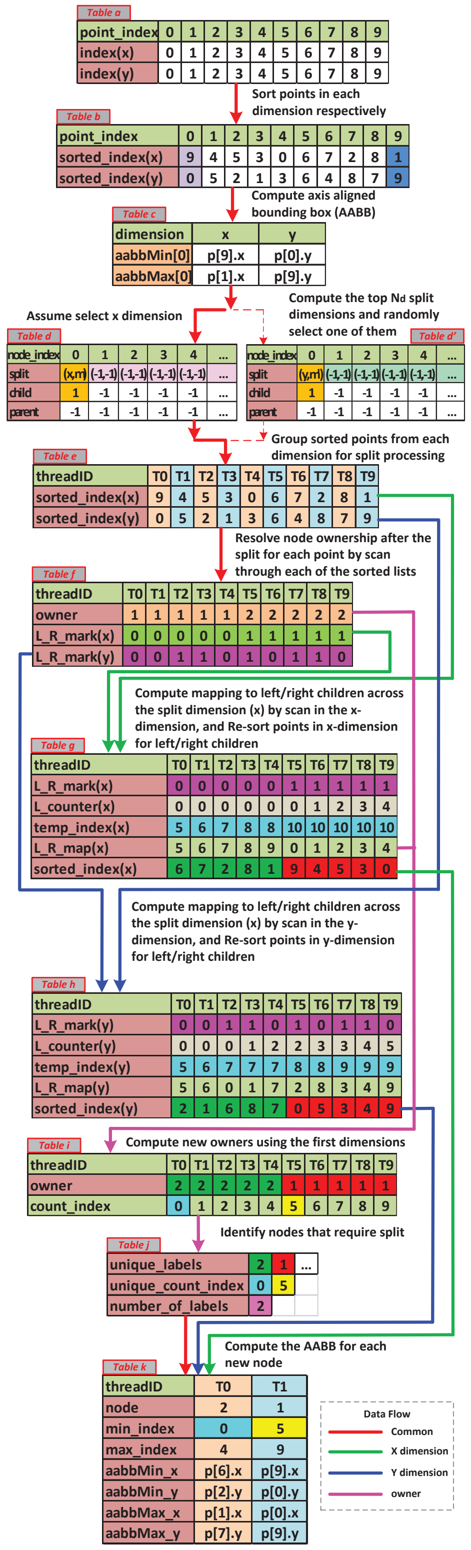

Sort points in each dimension respectively
Compute axis aligned bounding box (AABB)
Compute the top Nd split dimensions and randomly select one of them
Assume select x dimension
Group sorted points from each dimension for split processing
Resolve node ownership after the split for each point by scan through each of the sorted lists
Compute mapping to left/right children across the split dimension (x) by scan in the x-dimension, and Re-sort points in x-dimension for left/right children
Compute mapping to left/right children across the split dimension (x) by scan in the y-dimension, and Re-sort points in y-dimension for left/right children
Compute new owners using the first dimensions
Identify nodes that require split
Compute the AABB for each new node
Data Flow
Common
X dimension
Y dimension
owner

Table a

point_index	0	1	2	3	4	5	6	7	8	9
index(x)	0	1	2	3	4	5	6	7	8	9
index(y)	0	1	2	3	4	5	6	7	8	9

Table b

point_index	0	1	2	3	4	5	6	7	8	9
sorted_index(x)	9	4	5	3	0	6	7	2	8	1
sorted_index(y)	0	5	2	1	3	6	4	8	7	9

Table c

dimension	x	y
aabbMin[0]	p[9].x	p[0].y
aabbMax[0]	p[1].x	p[9].y

Table d

node_index	0	1	2	3	4	...
split	(x,m)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	...
child	1	-1	-1	-1	-1	...
parent	-1	-1	-1	-1	-1	...

Table d'

node_index	0	1	2	3	4	...
split	(y,m)	(-1,-1)	(-1,-1)	(-1,-1)	(-1,-1)	...
child	1	-1	-1	-1	-1	...
parent	-1	-1	-1	-1	-1	...

Table e

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
sorted_index(x)	9	4	5	3	0	6	7	2	8	1
sorted_index(y)	0	5	2	1	3	6	4	8	7	9

Table f

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
owner	1	1	1	1	1	2	2	2	2	2
L_R_mark(x)	0	0	0	0	0	1	1	1	1	1
L_R_mark(y)	0	0	1	1	0	1	0	1	1	0

Table g

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
L_R_mark(x)	0	0	0	0	0	1	1	1	1	1
L_counter(x)	0	0	0	0	0	0	1	2	3	4
temp_index(x)	5	6	7	8	8	10	10	10	10	10
L_R_map(x)	5	6	7	8	9	0	1	2	3	4
sorted_index(x)	6	7	2	8	1	9	4	5	3	0

Table h

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
L_R_mark(y)	0	0	1	1	0	1	0	1	1	0
L_counter(y)	0	0	0	1	2	2	3	3	4	5
temp_index(y)	5	6	7	7	7	8	8	9	9	9
L_R_map(y)	5	6	0	1	7	2	8	3	4	9
sorted_index(y)	2	1	6	8	7	0	5	3	4	9

Table i

threadID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
owner	2	2	2	2	2	1	1	1	1	1
count_index	0	1	2	3	4	5	6	7	8	9

Table j

unique_labels	2	1	...
unique_count_index	0	5	
number_of_labels	2		

Table k

threadID	T0	T1
node	2	1
min_index	0	5
max_index	4	9
aabbMin_x	p[6].x	p[9].x
aabbMin_y	p[2].y	p[0].y
aabbMax_x	p[1].x	p[0].x
aabbMax_y	p[7].y	p[9].y

Figure 5.3: Construction of the array based forest of randomized KD-trees on the GPU

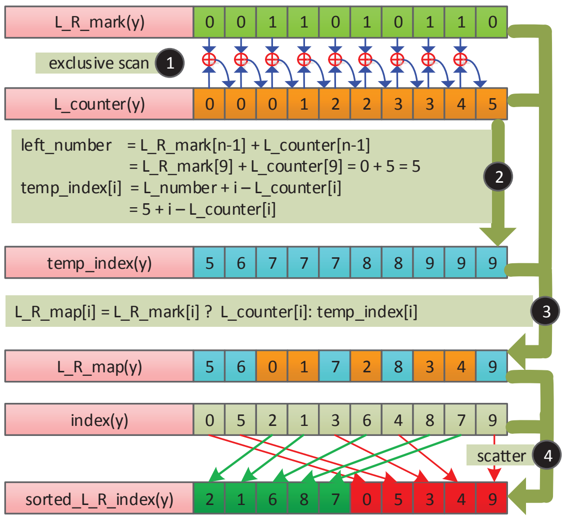


Figure 5.4: Operations in table g and table h in Figure 5.3

Next, the node split is launched through the function `NODE-SPLITS` (line 39 in Algorithm 6). The details of function `NODE-SPLITS` are shown in Algorithm 7. In this algorithm, all node splits are performed through multiple iterations of the `while` loop.

`NODE-SPLITS` in Algorithm 7 begins with some parameter initialization (lines 4 to 8). Notable is the pre-allocation of tree nodes (line 6). Prior to the split, a check is made to see if there are nodes at the current level of tree that deserve split, and if enough pre-allocated nodes are available for the pending splits (lines 12 to 39 in Algorithm 7). The results are updated in table d. Note that the function `atomicAdd` returns value of its first argument before the update.

Next step (lines 42 to 51 in Algorithm 7) determines the split dimensions for the nodes at the current level of the tree that undergo split (x dimension for node 0; the only node in the first iteration) and their corresponding mean values in their respective dimensions. In the first iteration, the left child of current node (root) is recorded as 1, with the right child having an implied value of 2. The parent node is fixed as -1. Further, the index for the parent of the current node's children is updated to 0. The entries for other nodes are initialized to -1. Table d only shows the columns for the first five nodes that have been pre-allocated.

After the split, information for nodes at the current level of the tree is updated, and some bookkeeping checks are performed on the CPU to prepare for the point distribution. First, a check is made to see if there are any nodes left to be split. If no node needs splitting, the algorithm will break out of the `while` loop and the KD-tree construction ends (lines 55 to 58). Next, another check is made to see if enough nodes have been pre-allocated, and the size will be doubled if more allocation is required(lines 62 to 67).

After computing the split dimension and value, assignment of points in a node to the descendant nodes is carried out (lines 70 to 95). Table e presents the assignment of workload to each thread in the first iteration. Each thread processes a group of two points (K points in general), from each of the sorted lists in x and y dimensions in table b. For each point in both lists, a check is made to resolve the left or right descendant in the split dimension. Assignments of points to descendant nodes are shown in table f, where each node is identified by its owner and a marker. As an example, the first element in owner array in dimension x is 1, indicating point 9 encountered in the x-sorted list belongs to node 1, along with the split dimension x. The L_R_mark for the first element is similarly set to 0 indicating ownership of the left partition node (node 1).

Next step is the distribution of points to the descendant nodes and sorting of points along all dimensions for each pair of descendant nodes (lines 100 to 108 in Algorithm 7). Three `Thrust` library functions; `exclusive_scan`, `transform` and `scatters` perform these tasks. Table g presents the partitioning and sequencing subsequent to the split in dimension x. The first and second halves of $sorted\_index(x)$ are the sorted sub-sequences in the x dimension for the left and right children after the split. Note that the left and right children and owners have been switched; a fallout from the use of `Thrust` library functions. The variable $count\_index$ assigns a count index to each point in the sorted sub-sequences.

Steps in table h are identical to those in table g, and demonstrate the process of re-sorting the sub-sequences in the left and right split nodes along the non-split dimension y without two explicit sorts. This is achieved through an exclusive scan of row $L\_R\_mark(y)$ in table h and recording the results in $L\_counter(y)$. The last element in $L\_counter$ corresponds to the number of points in the left child ($5 = 4 + 1$). Rows $temp\_index(y)$ and $L\_R\_map(y)$ in table h provide a mapping mechanism for the sort in the y dimension into the two sub-sequences. The details of the mapping through the `scan` and `scatter` operations in `Thrust` is shown in Figure 5.4. The last row in table h lists the indices of sorted lists in the left and right nodes in the y dimension subsequent to the split. Steps associating with tables g or h are repeated for each non-split dimension. Next, using the $L\_R\_mark(y)$ and $sorted\_index(y)$ saved in the previous split iteration, new $sorted\_index(y)$ are computed as shown in table i (lines 109 to 110).

The computations of the AABB for the left and right children of current node are shown in tables j to k. The first row in table j keeps record of unique nodes that have been split so far in the current round. The second row maintains the record of the starting count index in each sequence after the split. The last row keeps the record of the number of nodes generated in this iteration. This part of algorithm is performed through the `Thrust` function `unique_by_key_copy` (lines 114 to 117). Information in table j, the updated $stored\_index(x)$ in table g, and updated $stored\_index(y)$ in table i are used to compute the AABBs of left and right children. Since the nodes have already been sorted in each dimension, we can calculate the bounding box through the first and last elements in each dimension, in each of the left and right children nodes, through a launch of a kernel (lines 121 to 137).

The procedure detailed in tables a to k corresponds to the first split iteration. If any node requires a further split, additional iterations are performed through steps in tables d to k. In this example, the node splits terminate at the sixth iteration.

The constructions of the forest of KD-trees on the GPU devices are brought to a completion through the synchronization of pthreads.

5.4.2 BANNS on the Forest of Randomized KD-trees

In this section, we explore a massively parallel P-BANNS on the GPU cluster.

5.4.2.1 Nodes Distribution

Compared with the parallel ANNS on a single KD-tree [4], in this work, before launching the nearest neighbor querying on each GPU device, the query dataset is broadcast to each cluster nodes and then transferred to each GPU device. Next, the fine-grain parallel search is launched on the GPU device for the assigned query points. After $L = 5$ iterations of search (*i.e.* 5 backtracks), the local search results on each single tree is transferred to host node. On the host, the intermediate global best candidates for each query point is computed and broadcast to each node and their corresponding GPU devices. The global best candidates are used on each GPU device to update the local search priority queue, and then perform a further search on the local KD-tree. Once all the queries on each node are completed, the MPI is employed to collect the query results for the points from each node to the frontend node to perform a reduction and output the final results.

5.4.2.2 BANNS Queries

The BANNS for each query point is performed on the randomized KD-tree in L iterations of search independently. After each L iterations, we perform a synchronization among the nodes and the GPU devices to prune the local priority queues. Even though synchronization incurs a significant penalty, it results in pruning a large number of unproductive searches that are enqueued on the priority queue.

On single randomized KD-tree, since the ANNS for the query points are independent of each other, we perform parallel searches by launching one GPU thread for each query. Additionally, for the ANNS on each GPU device, we design BANNS using a novel sliding window to improve the speed performance of the query. Each query point uses the DFS to traverse down the tree until it reaches a leaf node. Once there, the P nearest neighbor point candidates will be selected in the leaf node. Some or all of the P nearest points, however, may reside outside of this leaf node. We, therefore, need to backtrack the tree to search for the other possible candidate points in the neighboring nodes. One possible approach is to invoke multiple ANNS queries in parallel in an uncoordinated manner [4]. However, uncoalesced memory accesses and uneven numbers and path divergences in the backtracks among the parallel execution threads severely compromise the performance. We partially alleviate the uneven workloads by letting the threads working on the ANNS queries to follow divergent paths to reach the leaf nodes. We partially alleviate the uneven workloads by letting the threads working on the ANNS queries to follow divergent paths to reach the leaf nodes while progressing through the same sequence of program instructions.

The major procedure for this algorithm is shown in Figure 5.5. In Step 1, the query dataset is broadcast from the cluster frontend to each node through MPI broadcast interface. In Step 2, we launch two *pthread* to copy the query dataset from the CPU to the two GPU devices (GPU-0 and GPU-1) on each cluster node. In Step 3, each *pthread* creates a CUDA kernel to perform the BANNS for query points transferred to its corresponding GPU device. One CUDA thread performs the BANNS for a single query point. In Step 4, one backtrack is performed to do a further search on another leaf node. In Step 5, once L iterations of backtracking are performed on each single tree. Next, the query results including the indices and distances of query points to neighbor candidates are copied back to CPU. In step 6, on the root node, we launch the MPI reduction interface to collect the best candidates with minimum distance to query points. Next, global reduction results are broadcast to each node which is similar step 1 to step 3. On each node and device, the local priority queues are updated with the global results, and the further L iterations search will be performed until the best P nearest neighbors for query points are obtained.

For a single query on a single randomized KD-tree in L backtrack iterations of search, Figure 5.6 depicts the major steps of the BANNS as described below.

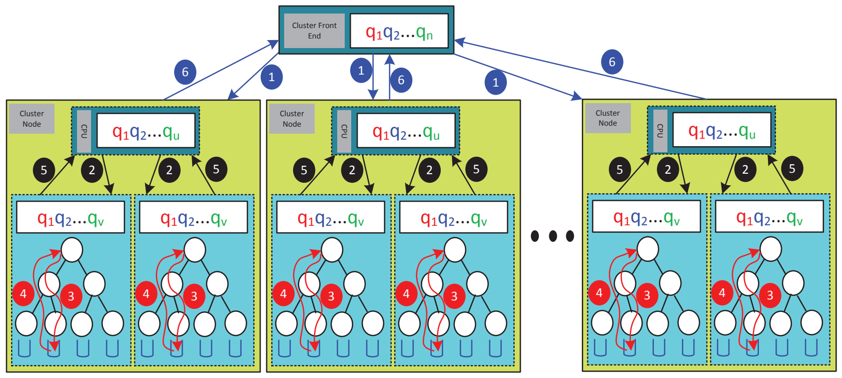


Figure 5.5: BANNS procedure on the GPU cluster

† In Step 1 in Figure 5.6 the BANNS query points are placed in the buffer of leaf nodes. As soon the buffers are full or all the query points have been scattered into buffers, the distance computation for finding the P nearest neighbors begins. The scattering of query points to the buffers is handled through a CUDA kernel, with each thread responsible for the traversal of a single query point from set Q to a leaf node. As a thread descends towards a leaf node, at each intermediate node, the priority queue is updated with the node index and the distance between the query point and the AABB of the child node which it is not branched to.

† Once a buffer in a leaf node is full or all the involved query points have been inserted into buffer, a second CUDA kernel is launched to compute the node-local P nearest neighbors for these query points (Step 2 in Figure 5.6). To compute the P shortest distances, we employ a *sliding window*. In the same step, the entries in the priority queue whose distance is greater than the current

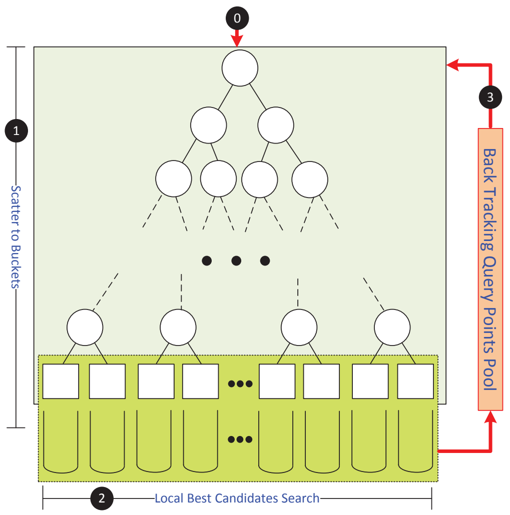


Figure 5.6: Major steps for the BANNS on single KD-tree

P^{th} shortest distance will be removed.

† In Step 3 in Figure 5.6, threads perform the status check. If the priority queue for a query point has become empty, or the number of backtracks has reached its upper bound before reaching the root node, we will terminate the search procedure for this query point. Otherwise, it will be inserted into the backtrack pool for further search in the next iteration.

† As the number of the backtrack of the query points from the leaf nodes drops below a threshold we terminate the process. The residual BANNS queries are performed through a parallel brute force search CUDA kernel [54]. Since each backtrack iteration requires launches of two time-consuming CUDA kernels, it is more compute efficient to do the search for these remaining query points employing the parallel brute force.

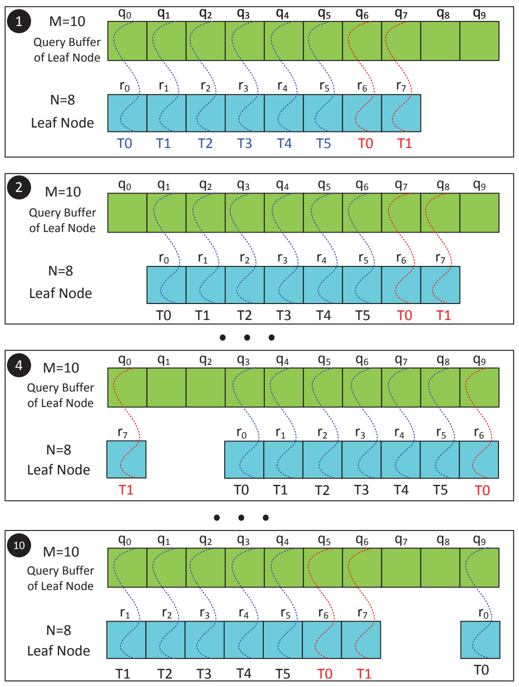


Figure 5.7: Sliding window for distance computing

Unlike the work in [117], the use of sliding window in Step 2 of Figure 5.6 for the computation of the P nearest neighbor distances ensures that all global memory accesses are coalesced. Figure 5.7 depicts the working of the threads in the sliding window where each thread access one pair of query point and leaf node to compute the distance between them. After a sequence of slides the node-local P shortest distances for all the query points stored in the node buffer are computed. In the depiction of Figure 5.7 with a leaf node with eight points, ten query points in its associated buffer, and assignment of eight threads for the work, the process takes 10 slides of the window.

Algorithm 8 presents the implementation details, where the recursion in the traditional DFS is converted to an iterative `while` loop that works better on the GPU. The backtracks are implemented through iterative exchange of three pointers ($leftchild$, *bestchild* and *otherchild*), and the priority queue. The work is spread across three kernels. In the first kernel, the concurrent threads following divergent paths steer the query points into the leaf node buffers (one query point per thread). The second kernel associates a thread block to one leaf node and uses the sliding window to compute the distances for the P nearest neighbors local to the node. The update of priority queue is also performed by this kernel. When the number of residual query points, left over from the backtrack iterations, falls below a threshold, the third kernel performs a brute force parallel BANNS.

Algorithm 7 Node split(Continued)

```
52:     [CPU coordinate]while loop termination and node
53:       reallocation.
54:     //no node deserving split, break out while loop
55:     if (($last\_node\_count$ == $node\_count$) and
56:        ($allocated\_enough$)) then
57:        break;
58:     end if
59:     $last\_node\_count \leftarrow node\_count$;
60:
61:     //resize pre-allocated node size
62:     if (!$allocated\_enough$) then
63:        double pre-allocated node size;
64:        update new nodes' split/child/parent info;
65:        $M \leftarrow 2 * M$;
66:        continue;
67:     end if
68:     //launch $N'$ threads for this kernel, one thread
69:       //works for one column of dimensional values;
70:     [GPU Kernel]L_R_mark
71:     $owner \leftarrow owner[threadIdx]$;
72:     $leftchild \leftarrow child[owner]$;
73:
74:     //leaf node does not deserve split
75:     if ($leftchild$ == -1) then
76:        return;
77:     end if
78:     //compute split dimension and split value
79:     $split\_dim \leftarrow splits[owner].dim$;
80:     $split\_value \leftarrow splits[owner].value$;
81:
82:     //projection of points at each dimension to split dimension
83:     for $j = 0$ to $K - 1$ do
84:        $project[j] \leftarrow$ [projection]
85:           ($split\_dim$, $point\_array[j, threadIdx]$);
86:     end for
87:     //update owner and left/right marks at each dimension.
88:     for $i = 0$ to $K - 1$ do
89:        $L\_R\_mark[i, threadIdx]$
90:           $\leftarrow (project[i] > split\_value)$;
91:     end for
92:     $owner[threadIdx]$
93:           $\leftarrow leftchild + L\_R\_mark[0, threadIdx]$;
94:     synchronization;
95:
```

Algorithm 7 Node split(Continued)

```
96:         //launch n threads in the following three Thrust kernels,
97:           //to sort points in each leaf node.
98:         [GPU Kernels]distribute_points_to_children_nodes
99:         for $i = 0$ to $K - 1$ do
100:            $L\_R\_temp[i, threadIdx] \leftarrow$ [exclusive_scan]
101:                                    $(L\_R\_mark[i, threadIdx];)$
102:            $L\_R\_map[i, threadIdx] \leftarrow$ [transform]
103:                                    $(L\_R\_temp[i, threadIdx]);$
104:            $sorted\_index[i, threadIdx] \leftarrow$ [scatters]
105:                $(L\_R\_map[i, threadIdx], sorted\_index[i, threadIdx]);$
106:         end for
107:         $owner[threadIdx] \leftarrow$ [scatters]
108:                      $(L\_R\_mark[0, threadIdx], owner[threadIdx]);$
109:
110:         //launch $N'$ threads in the following Thrust kernels, to
111:           //compute unique labels and count.
112:         [GPU Kernel]point_unique_ownership_label
113:         $number\_of\_labels \leftarrow$ [unique_by_key_copy]
114:                    $(owner, count\_index, unique\_labels,$
115:                    $unique\_count\_index);$
116:
117:         //launch number_of_labels number of threads to compute
118:           //AABB for children nodes after the split.
119:         [GPU Kernel]compute_AABB_for_children_nodes
120:         //gather split info for current children nodes
121:         $index \leftarrow unique\_lables[threadIdx];$
122:         $left \leftarrow unique\_count\_index[threadIdx];$
123:         $splits[index].left \leftarrow left;$
124:         if $(threadIdx < (number\_of\_labels - 1))$ then
125:             $right \leftarrow unique\_count\_index[threadIdx + 1];$
126:         else
127:             $right \leftarrow N';$
128:         end if
129:         $splits[index].right \leftarrow right$
130:
131:         //compute AABB info for current children nodes
132:         for $k = 0$ to $K - 1$ do
133:             update $AABB[k]$ of current node according to left and
134:                 right indices;
135:         end for
136:     end while
137: end procedure
```

Algorithm 8 BANNS on the Forest of Randomized KD-trees

```
Input:  cell data structure (parent, child, splits), aabbMin, aabbMax,
        query, $p$, $bt\_sh$, $L$
Output: results data structure
procedure BANNS
    MPI_Init();
    MPI_Broadcast(query points dataset);
    while (!result_flag) do
        //each pthread perform search on one randomized KD-Tree
        create two pthreads on two cores through pthread_create(), each
         pthread thread invokes BANNS on one randomized KD-tree on
         one of the two GPU devices attached to a node, with half of the
         query points assigned to this computing node.

        // Tasks for each pthread
        allocate memory and copy $V$ query points to GPU;
        $backtrack \leftarrow false$;
        $currentnode \leftarrow 0$;
        $bk\_num \leftarrow 0$;
        create a priority queue $Q_p$;
        $query\_pool\_empty \leftarrow false$;

        while ((!$query\_pool\_empty$) || ($bk\_num < L$)) do
            [GPU Kernel:Scatter Query Points into Leaf Buffers]
            $leftchild \leftarrow child[currentnode]$;
            if $leftchild == -1$ then
                $bestchild \leftarrow leftchild$;
                $otherchild \leftarrow leftchild$;
                //check candidate node in left or right child node
                $split \leftarrow splits[currentnode]$;
                $delta \leftarrow query[i].[split.dim\_val] - split.split\_val$;
                if $delta < 0$ then
                    $otherchild{+}{+}$;
                else
                    $bestchild{+}{+}$;
                end if

                //compute distance from query point to AABB
                $d_{AABB} \leftarrow$ compute the distance from query point to
                   bounding box AABB of $otherchild$;
                $enqueue(Q_p) \leftarrow (otherchild, d_{AABB})$;
                //prepare for the next search iteration
                $currentnode \leftarrow bestchild$;
            else
                insert the query point index in the buffer associated with
                   this leaf node;
            end if
```

Algorithm 8 BANNS on the Forest of Randomized KD-trees (Continued)

```
47:            [CPU Coordinate:Boundary Computing]
48:            Check the status of each query point in the query pool. If
49:              the priority queue with it is empty or the next search target
50:              is root, remove this points in the pool; then, compute the
51:              index boundary of rest query points in the buffer and train
52:              points in leaf nodes; finally, copy these information to GPU.
53:
54:            [GPU Kernel:Local Candidates Search by Sliding Window]
55:            //Apply Sliding Window Technology to compute local NN.
56:            if $train\_point > query\_point$ then
57:                   Query points slide on train points, update the
58:                   NN for each query point;
59:            else
60:                   Train points slide on query points, update the
61:                   NN for each query point;
62:            end if
63:
64:            //extract node with mindist from $Q_p$
65:            $(currentnode, mindist) \leftarrow dequeue(Q_p)$;
66:            //examine candidate sub-tree in next iteration
67:            if $(mindist \leq restults[i].index[P-1])$ then
68:                $backtrack \leftarrow false$;
69:            end if
70:
71:            [GPU Kernel:Residual Query Points Brute Force Search]
72:            if $num\_res\_query\_point > res\_threshold$ then
73:                all the residual query points will scan all the train points
74:                     in brute force method;
75:                $query\_pool\_empty \leftarrow true$;
76:            end if
77:        end while
78:        MPI_Reduction(query_results);
79:    end while
80:    MPI_Finalize();
81: end procedure
```

5.5 Performance Optimization

Performance of construction of forest of KD-trees and the BANNS algorithms on the GPU cluster are influenced by several factors, including the global memory access coalescing, shared memory bank conflicts, branch divergences, local and global synchronization overhead and the organization of thread blocks [119].

First, as mentioned before, the single KD-tree is constructed using linear SOA. The GPU SPMT architecture can process vectors more efficiently than the nonlinear data structure such as tree. Second, to ensure that global memory accesses on the GPU are coalesced, *i.e.* threads accesses to memory are combined into a single transaction, into an aligned and contiguous block of global memory, we perform a preprocessing, prior to copying the input data from the CPU host to the GPU device. We restructure data employing the SOA instead of the AOS in the representation of data structures. Third, breaking of Algorithm 8 into three kernels allows us to implement the most time-consuming part of the algorithm as linear vectors, resulting in very high performance. Forth, in the design of this work, the read-only data are placed in the texture and surface memories to boost the performance of reading accesses. The texture and surface memories reside on the device and are cached in the texture cache, and therefore, presents a better alternative to accessing the global memory. Fifth, to improve the performance, we try to reduce the branch conditions in loops algorithms. We achieve this by loop unrolling through the use of `#pragma unroll` directive in CUDA. Loop unrolling, of course, results in register pressure, which is alleviated through increase in the size of L1 cache. There are multiple large `for` loops in the parallel algorithms that are candidates for loop unrolling.

The performance benefit of randomized KD-tree forest comes in two way. First dividing the N searchable data points across 16 GPU devices, results in single KD-trees whose heights are smaller, and therefore, lower number of backtracks. Further, the global synchronization among the individual KD-trees in the forest reduces the number of backtracks number. This due to the fact that the local priority queues are updated with the global results where unproductive target cells will are pruned.

5.6 Experiments and Results

In this section, we provide experimental performance validations of the GPU cluster accelerated parallel forest of KD-trees construction and the BANNS algorithms[1]. We adopted real-world image descriptor datasets with a wide range of sizes and dimensions from Winder and Brown dataset [120] [121], as well as datasets from high-dimensional SHOT feature descriptors, extracted from typical point clouds [55].

[1]Michigan Tech IVS Computing Cluster consists of eight nodes, with each node equipped with a 4-core CPU and two GPUs. Each CPU is a 4-core, 3.2 GHz Intel $i7-970$ processor, with Ubuntu 12.04 OS, with 1.14 GHz, and the GPU device is GeForce GTX 680 GPU with 4 GB RAM with 7 Streaming multiprocessors

We performed multiple sets of experiments to evaluate the performance of the parallel construction of the forest of KD-trees and the BANNS. We also explored the major performance impact factors of the BANNS, such as the size of the reference (S) and query (Q) datasets, dimensionality, tree height, the number of backtracks, and the number of nearest neighbor of single query point (P).

Table 5.1
Comparison with the related works

Work	**KD-tree**	**NNS**	**Dim** (d)	**Query Size** (Q)	**P-NNS** (P)	**Target Applications**	S_{kdtree}	S_{NNS_kdtree}	S_{NNS_BF}	A_{NNS}
[115]	–	linear, GPU	96	38400	20	entropy, KL divergence	–	137	35	exact
[118]	–	linear, GPU	78	100k	1	machine learning	–	–	21	$\sim$ 90%
[117]	CPU	nonlinear, GPU	12	10M	10	machine learning	–	89	–	exact
[64]	CPU	nonlinear, CPU cluster	128	100k	1	image descriptors	–	–	10	>90%
Work on Single GPU	GPU	hybrid, GPU	96	38400	20	image descriptors	8	84	93	>90%
			96	100900	4		12	133	136	
			128	100900	4		16	138	86	
			256	100900	4		17	163	95	
			512	100900	4		14	128	118	
This Work	GPU	hybrid, GPU cluster	96	38400	20	image descriptors	35	97	104	>90%
			96	100900	4		28	629	643	
			128	100900	4		28	1117	716	
			256	100900	4		36	1344	781	
			512	100900	4		30	953	876	

5.6.1 Evaluation on Real-world Image Descriptors

In these experiments, we have used the library of real-world images to sample image descriptors of different dimensionality from Trevi Fountain image patches [120][2]. We

[2]The data is taken from Photo Tourism reconstructions from Trevi Fountain (Rome). Each dataset consists of a series of corresponding patches, obtained by projecting 3D points from Photo Tourism

have employed the approach in [120] [121] to generate real-world image descriptors with varying dimensions. Moreover, the query sets are the same as the reference sets that were used for the construction of the KD-tree forest ($S = Q$). This is justified as in the image descriptor matching, every point descriptor has to be matched. To make sure the search accuracy is beyond 90%, we have adjusted the number of backtracking steps according to the size and dimensionality of the dataset and the number of query points involved.

5.6.1.1 Performance Comparison with Related Works

Table 5.1 provides a brief comparison of this work with the related works in higher dimensions. The speedup factors (S_{kdtree}) is with respect to the construction runtime using the PCL library [55]. S_{NNS_kdtree} and S_{NNS_BF} are the speedups of parallel algorithm on the GPU cluster over the sequential counter part on CPU, and over the serial brute force linear search on the CPU, respectively. We adopted the sequential linear brute force NNS algorithm in the PCL as the common benchmark reference for a fair speedup comparison between various schemes. A_{ANNS} represents the accuracy of the ANNS.

As shown, the algorithms presented in this work provide the highest performance for

reconstructions back into the original images. The Trevi Fountain image patch suites includes up to 100900 key point descriptors. `http://phototour.cs.washington.edu/patches/`

the large datasets having high dimensionality of up to 512, for both the construction of the forest of KD-trees and the BANNS. We also achieve an accuracy of more than 90% for the BANNS similar to works in [64] and [118]. Additionally, Table 5.1 presents the results for the proposed technique when only one KD-Tree is employed and the workload is assigned to a single node with a single GPU. Implementation in a single GPU results in a factor of 4 to 8 slower speedups for S_{NNS_kdtree} and S_{NNS_BF}, respectively.

5.6.1.2 Comparison Between the Parallel Algorithms on the GPU and CPU Clusters

Next, we compare the performance of parallel construction of the forest of KD-trees and the BANNS on the GPU cluster with respect to their counterparts on the CPU cluster. We generated seven target N-point sets S, with $N = 2560$, 5120, 10240, 20480, 40960, 81920, and 100900 to cover a wide range of KD-tree sizes. The maximum dataset size of the real-world image is up to 100900 descriptors. We chose a high-dimensional case of $K = 512$. Table 5.2 presents the results. Parameters T_{crkdt}/T_{grkdt} represent runtimes for parallel randomized KD-tree forest construction on the CPU/GPU cluster. Parameters T_{cbanns}/T_{gbanns} represent runtimes for parallel BANNS on the CPU/GPU cluster. Parameter S_{rkdt}/S_{banns} denotes speedup factor of parallel algorithm on the GPU cluster over counterparts on the CPU cluster for the construction of the randomized forest of KD-trees/the BANNS. As seen the speedup of parallel construction of the forest of KD-trees and the BANNS on the GPU cluster can reach up to 2.5 and 61.7, respectively.

Table 5.2
Construction of the forest of KD-trees and the BANNS runtimes (in ms) and speedups in high-dimensional ($d = 512$) space

size of	**CPU**		**GPU**		**Speedup**	
dataset	T_{crkdt}	T_{cbanns}	T_{grkdt}	T_{gbanns}	S_{rkdt}	S_{banns}
2560	14	1773	65	3146	0.2	0.6
5120	23	5696	69	4434	0.3	1.3
10240	38	17431	75	5279	0.5	3.3
20480	82	48437	89	5911	0.9	8.2
40960	166	122174	101	7423	1.6	16.5
81920	347	381206	145	9964	2.4	38.3
100900	383	711219	154	11532	2.5	61.7

5.6.2 Speedup and Accuracy Impact Factors

In this subsection, we study the speedup and accuracy impact factors to the construction of forest of KD-trees and the BANNS.

5.6.2.1 Effect of the Dataset Size and the Dimensionality

Figure 5.8 plots the runtimes of the forest of KD-trees construction versus the number of points for several dimensions K for the parallel construction algorithm. The plots demonstrate the power of the MPA where the massive parallelism works best for

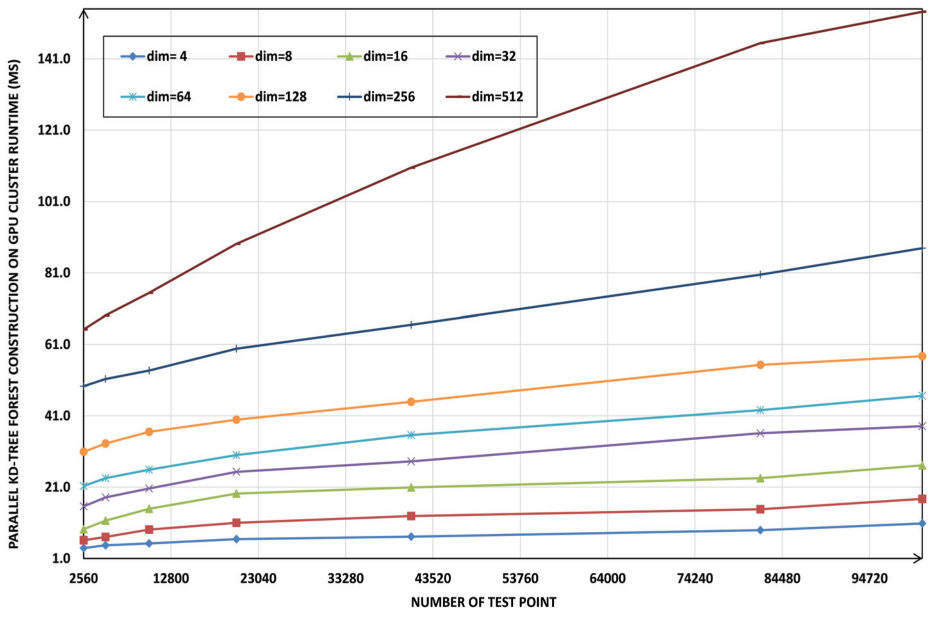


Figure 5.8: Runtime of parallel forest of KD-trees construction versus the number of points in set S for various dimensionality

large datasets when thread operations are coordinated (*i.e.* `sort, scatter`, *etc*); a 40-fold increase in the data size results in only 2-fold increase in the runtime. Further, a closer observation of Figure 5.8 reveals that for a given number of reference points the runtime slowly increases with the dimension K (specially for smaller values of K). This is due to optimization steps in Section 5.5; the loop unrolling, the use of SOA, and efficient use of L1 cache.

Figure 5.9 plots the runtimes of the BANNS versus the number of points for several dimensions K using the parallel algorithm. The rate of increase in the runtime in the case of the BANNS is much higher than that of the construction of the forest of KD-trees in Figure 5.8. The reason for this is the GPU architecture is ill-suited for nonlinear tree operations.

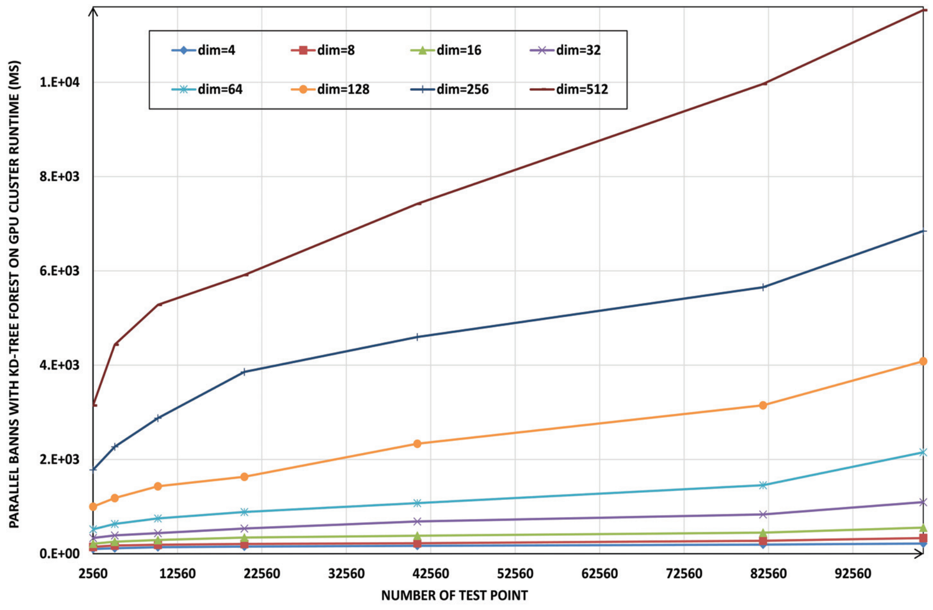


Figure 5.9: Runtime of parallel P-BANNS ($P = 4$) versus the number of points in set S for various dimensionality

5.6.2.2 Effect of the KD-tree Height

Plots in Figure 5.10 present the effect of the KD-tree height on the runtime of the parallel BANNS for the 256-dimensional KD-tree forest. As seen, for the number of query points less than $20,000$, the optimum KD-tree height lies between 3 to 5. The optimum height ranges from 6 to 8 for the number of query points between $40,960$ to $100,900$. With a smaller tree height, the BANNS is more a linear-like search, resulting in an increase in the runtime. On the other hand, a large tree height yields higher number of backtracks, degrading the runtime performance. So, in the experiments of this work, the tree heights were chosen in accordance with the image descriptor datasets dimensionality and sizes. Due to the effectiveness of global synchronization

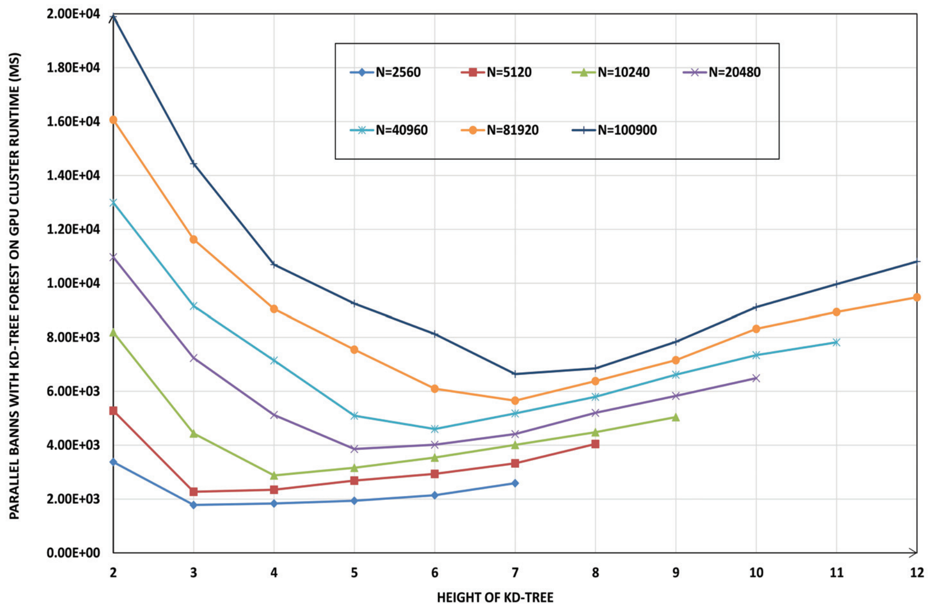


Figure 5.10: Runtime of parallel P-BANNS ($P = 4$) versus the tree height for a 256-dimensional forest of KD-tree for various dataset sizes S.

on the pruning of the priority queue of the BANNS, the height of the KD-tree has a lesser impact on the performance than on single KD-tree.

5.6.2.3 Effect of the Number of Backtracks

As discussed, an increase in the number backtracks yields a higher search accuracy, as more nodes are inspected. Plots in Figure 5.11 present the effect of the number of backtracks on the runtime of the parallel BANNS for the dataset size of $81,920$ with maximum leaf size of 64, for dimension choices of 4, 16, 64 and 256. The runtimes saturate after a certain number of the backtracks, depending on the dimension of the descriptor in the dataset. For 4, 16, 64 and 256-dimensional KD-tree forest, the saturation points are around 50, 160, 280, and 400, respectively. In the experiments, with the optimum tree height (Figure 5.10), to achieve an accuracy of more than 90% for the BANNS, the number of backtracks was chosen based on the dimensionality of the descriptor. The number of backtracks in this design is much lower than that on a single KD-tree, since a large number of backtracks are pruned during the synchronization among multiple KD-trees.

Next, we study the relation between the required search accuracy and the corresponding minimum number of backtracks. Plots in Figure 5.12 present the minimum number of required backtracks for search accuracy levels of 50%, 60%, 70%, 80% and 90%, for dataset size of 81920, for dimensions 4, 16, 64, 256 and 512. As expected for the higher dimensions the rate of increase in number of backtracks with the accuracy is significantly higher.

5.6.2.4 Effect of the Number of Nearest Neighbors of Single Query Point (P)

In all experiments so far, a value of $P = 4$ has been assumed. To study the effect of number of nearest neighbor points, we evaluated the runtime performance of the P-BANNS in a range of P values for several 256-dimensional datasets. From the plots in Figure 5.13 the runtime of the algorithm first increases linearly with P, and tends to saturate beyond a certain point. This is due to the effectiveness of sliding

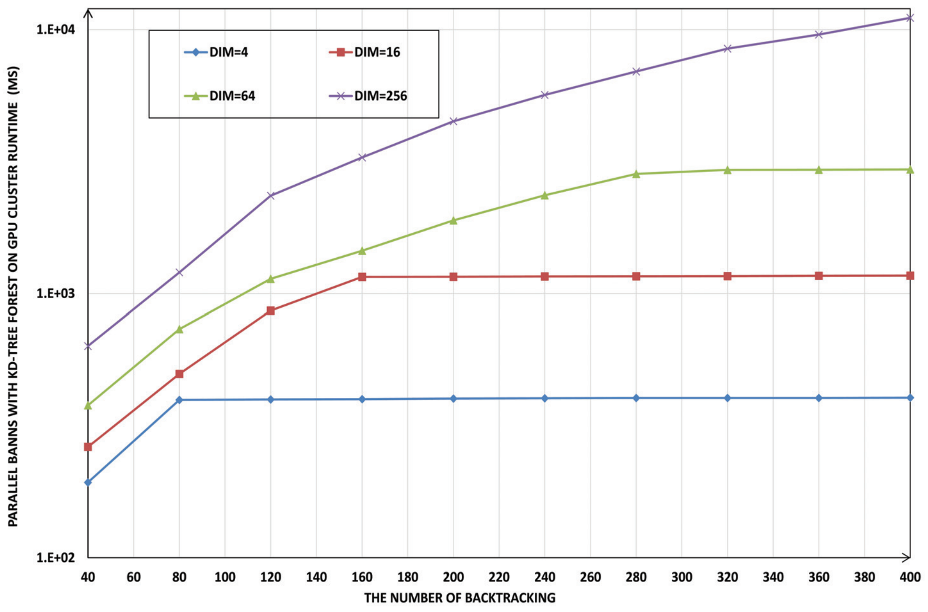


Figure 5.11: Runtime of parallel P-BANNS ($P = 4$) versus the number of backtracks for different dimensionality with dataset size $S = 81920$

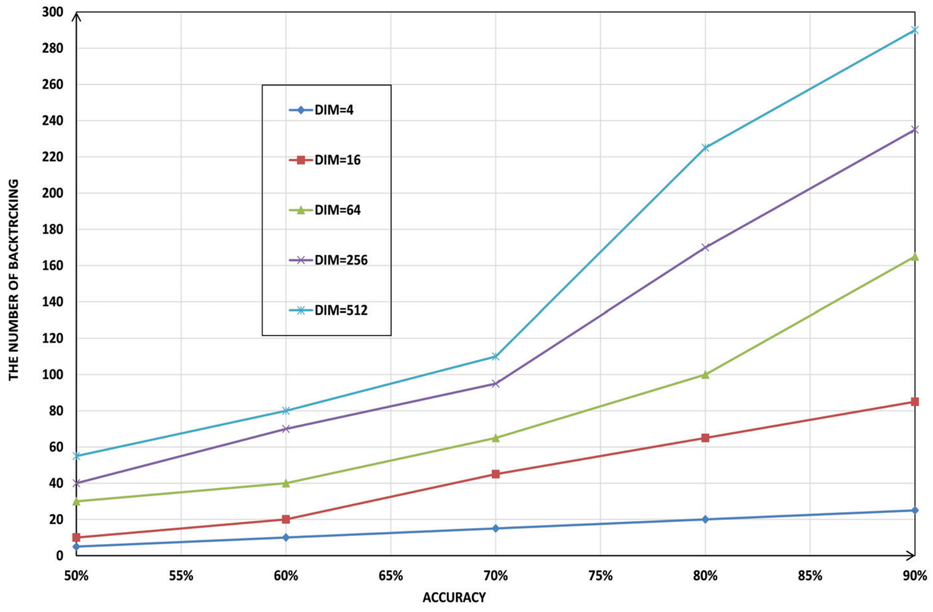


Figure 5.12: Minimum number of backtracks versus the accuracy for five different dimensionality for dataset size of $S = 81920$ and $P = 4$

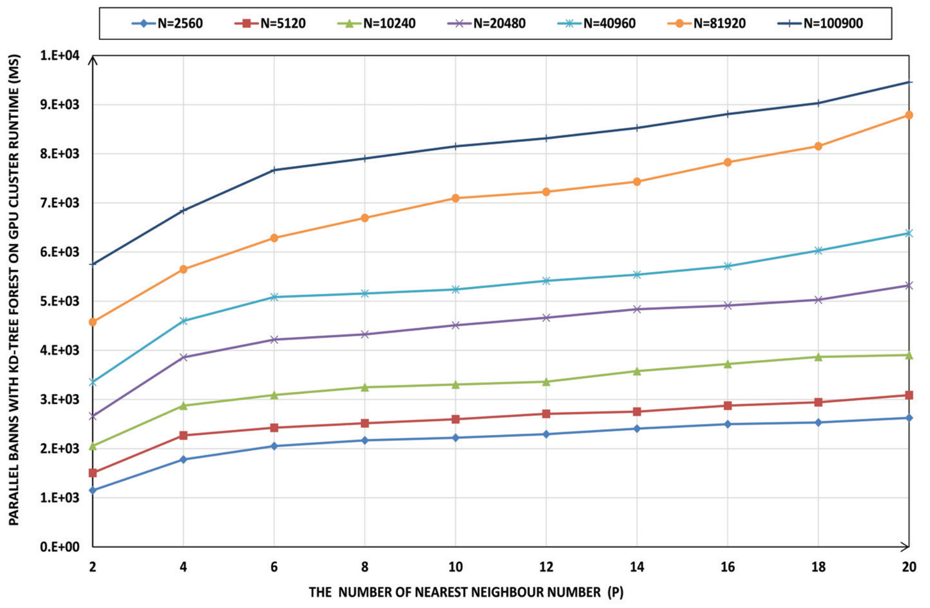


Figure 5.13: Runtime of the parallel P-BANNS versus the value of P for a 256-dimensional KD-tree with several datasets S.

window for P-BANNS on the forest of KD-tree for larger values of P.

5.6.2.5 Effect of Global Synchronization to the Number of Backtracks

To study the impact of global synchronization to the number of backtracks in the forest of KD-trees, we design two sets of experiments. In the first experiment set each single KD-tree is constructed with the whole searchable dataset containing N points. We perform $M = N$ queries on each single KD-tree. In the second experiment set, each single KD-tree is constructed with the $N/16$ searchable points. We again perform $M = N$ queries on each single KD-tree. In each experiment, we execute two tests, one with and one without the global synchronization enabled. In each test, we set N to 81920, with two different irrationalities ($d = 256$ and $d = 512$). Table 5.3 present the results of the experiments. When the size of each KD-tree is set to $N/16$, the number of backtracks is reduced by 46% and 51%, respectively, for 256 and 512-dimensional datasets when the synchronization across the KD-trees in the forest is enabled. For KD-tree size of N the corresponding reductions in the number of backtracks are 74% and 76%, respectively, for 256 and 512-dimensional datasets.

Table 5.3
Effect of global synchronization ($N = 81920$) on the forest of KD-tree

Test	**Test Scenario**			**Backtrack Num**	
Case	Tree Size	Query Size	Global Sync	d=256	d=512
I	N/16	N	without	435	595
II	N/16	N	with	235	290
III	N	N	without	1400	1700
IV	N	N	with	365	410

5.6.3 Evaluation of the Sliding Windows

Next, we evaluate the speed performance improvement of the sliding windows for BANNS on buffered KD-tree forest, with priority queue on GPU. For evaluation we used the datasets of dimensions 256 and 512, with the number of the points ranging from $2,560$ to $100,900$. The runtimes of the parallel BANNS ($P = 4$) with and without sliding window are presented in Table 5.4. Compared with the parallel BANNS on the GPU cluster without the sliding window, the novel parallel BANNS on the GPU cluster can improve the runtime performance by up to 38% while maintaining the same accuracy.

Table 5.4
Sliding window runtime performance ($P = 4$) evaluation

dataset	**W/O SW (ms)**		**W/ SW (ms)**		**Improvement**	
size	d=256	d=512	d=256	d=512	d=256	d=512
2560	1841	3317	1779	3146	3.5%	5.4%
5120	2463	4899	2268	4434	8.6%	10.5%
10240	3234	6035	2875	5279	12.5%	14.3%
20480	4412	6989	3856	5911	14.4%	18.2%
40960	5409	8995	4597	7423	17.7%	21.2%
81920	6682	12982	5651	9964	18.2%	30.3%
100900	8354	15993	6846	11532	22.0%	38.7%

5.6.4 Experiments on the SHOT Matching

To verify the performance of the massively parallel high-dimensional construction of the forest of the KD-trees and the BANNS, on another real world application, we conducted a series of matching experiments on nine real point cloud datasets [55]. The sets chosen include six 3D point cloud models and three scenes, commonly used in computer graphics and computer vision. The sets are shown in Table 5.5. All models and scenes are available online for download [55]. For each point cloud dataset, we first sampled out the key points and then computed SHOT local descriptors for each key point. The SHOT as a novel 3D object local descriptor can achieve a good balance between descriptiveness and robustness. The dimensionality of regular SHOT descriptor is 320 [59] [60]. Next, we constructed a forest of KD-trees with those 320-dimensional descriptors and then search for $P = 4$ nearest neighbors for each key point on the KD-tree. In other words, the query descriptor sets are the same as the descriptor set for the KD-tree construction ($Q = S$). The results are presented in Table 5.5, where the parameter N_{key} denotes the number of key points. Further, parameters T_{ccnst} and T_{gcnst} denote runtimes of the parallel construction of the forest of KD-trees on the CPU cluster and its parallel counterpart on the GPU cluster. Similarly, T_{csrch} and T_{gsrch} denote the runtimes of the parallel BANNS on the CPU cluster and parallel equivalent on GPU cluster. Parameters S_{cnst} depicts the speedup of the parallel construction of the forest of KD-trees on the GPU cluster over the parallel counterpart on the CPU cluster. Also, S_{srch} demonstrates the speedup of the parallel BANNS on the GPU cluster over the parallel equivalent on the CPU cluster. As seen, the maximum speedup of the parallel construction of the forest of KD-trees on the GPU cluster reaches to 2.4. The speedup of the BANNS reaches to 62.

5.7 Conclusion

This chapter presented the design of high performance parallel construction of the randomized forest of KD-trees and the BANNS on the GPU cluster for high-dimensional

Table 5.5
Matching runtime (in ms) and speedup of the parallel algorithm over the serial algorithm

Model/Scene Dataset	N_{key}	T_{ccnst}	T_{csrch}	T_{gcnst}	T_{gsrch}	S_{cnst}	S_{srch}
Milk Box Model	13704	53	22146	74	5133	0.72	4.31
Office Chair Model	18715	81	41793	83	6042	0.98	6.92
Stanford Bunny Model	20446	89	43428	98	5927	0.91	7.33
Chicken Model	85693	368	523147	153	10132	2.41	51.63
Stanford Dragon Model	80047	312	440045	136	9863	2.29	44.62
Happy Buddha Model	99614	413	703953	174	12345	2.37	57.02
Office Scene	89031	387	597681	158	10571	2.45	56.54
Table Scene	66053	247	180179	125	8019	1.98	22.47
Five people Scene	91143	394	682391	165	10984	2.39	62.13

image descriptor matching. The proposed algorithms are of comparable quality to the traditional sequential counterparts on CPU, while achieve high speedup performance in a wide range of dimensions. The massively parallel algorithms presented in this chapter were tested on real-world image descriptors, with varying dimensionality, as well as classical point cloud descriptors in real applications. The speedups of the construction of the forest of KD-trees and the BANNS reach up to 2.5 and 61.7 respectively with real-world image descriptors with varying dimensionality. For the real application with SHOT descriptor dataset, the corresponding speedups raise up to 2.4 and 62. The implementations in this work will benefit real-time 3D image registration in low-dimensional spaces, and image descriptor matching employing high-dimensional forest of KD-trees.

Chapter 6

Conclusion

In this chapter, we summarize the major contributions of the designed algorithms, discuss the constrains and limitations of this work, and present some suggestions for the future work.

6.1 Completed Work

This dissertation presented a set of highly efficient algorithms for 3D object recognition on heterogeneous parallel computing platforms. To reduce descriptor computing, indexing and matching runtime while maintaining comparable quality, the problem is tackled in a novel mix of parallel and distributed computing arrangement. A set of heterogeneous parallel and distributed algorithms on the MPA and cluster are designed and developed in this work.

1. We investigated the development of suitable massively parallel algorithms on the GPU for computation of high density and large-scale 3D object local descriptors. We designed two alternative parallel algorithms (G-SHOT); one exact, and one approximate, on the GPU to speed up the original serial SHOT. Experimental results show both algorithms exhibit outstanding speed performance.

2. We presented a massively parallel ANNS on the KD-tree on the modern MPA. The proposed algorithm is of comparable quality to the traditional sequential counterpart on the CPU. Moreover, it achieves a high speedup factor when applied to high-dimensional real-world image descriptor datasets. The algorithm is also studied for factors that impact its performance to obtain the optimal runtime configurations for various datasets. The implementation in this work will potentially benefit real-time image descriptor matching in high dimensions.

3. We presented a parallel KD-tree construction for image descriptor indexing and a parallel BANNS on the MPA. To improve the runtime performance of the BANNS, we propose an efficient sliding window for a parallel BANNS on the KD-tree to mitigate the high cost of global memory access. When applied to high-dimensional real-world image descriptor datasets, the proposed KD-tree construction is of comparable quality to the traditional sequential counterpart

on the CPU while outperforming its serial counterpart on the CPU by a significant speedup factor.

4. We presented parallel and distributed algorithms for the construction of the forest of randomized KD-trees and the BANNS on a cluster equipped with the MPA devices of the GPU. In order to utilize the GPU cluster platform more fully, we designed a distributed randomized KD-tree forest for the BANNS to alleviate the backtracking cost on a single KD-tree. Additionally, the algorithms are studied for the performance impact factors to obtain the optimal runtime configurations for various datasets.

6.2 Constraints and Limitations

Even though a set of high efficient parallel and distributed algorithms are explored in this works, there are some constraints and limitations in the algorithms and the heterogeneous computing platforms. In this section, we will discuss the SM occupancy of the GPU, memory utilization on the GPU and the communication cost of parallel machines on the cluster.

6.2.1 SM Occupancy

SM Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Low occupancy results in poor instruction issue efficiency, because there are not enough eligible warps to hide latency between dependent instructions. Ideally, we like the number of active SM warps to be equal to the maximum number defined by CUDA compute capability. However, due to hardware resource limitation (shared memory, and register) and the organization of thread blocks (number of threads per block), it cannot reach its peak value. When occupancy is at a sufficient level to hide latency, increasing it further may degrade performance due to the reduction in per thread available resource.

6.2.1.1 SM Occupancy of Parallel Descriptor Computing

The SM occupancies of the two parallel G-SHOT descriptors on GPU are 33% and 42% respectively. These relative lower occupancies are due to intrinsic complexity of the workload for each single thread on SM and SM hardware limitation. The maximum register per thread is 48 on GTX 470 with compute capability 2.0. The the LRF and descriptor computing kernels involve a large number of register which triggers the register spilling observed in our profiling results. The major complexity of these two kernels come from the EVD computing and SHOT computing complexity for each thread on a wrap. To reduce the register spilling, we can enlarge the cache at the expense of reducing the size of share memory. In this case, only increasing the number of single processor (SP) in SM cannot increase the speedup. Enlarging the maximum number of concurrent blocks per SM and the maximum number of register per thread can, however, help with the speedup.

6.2.1.2 SM Occupancy of Parallel KD-Tree Construction

There are two major categories of CUDA kernels for the parallel KD-tree construction on the GPU. The kernels in the first category perform expansion or reduction. The number of threads doubles or halvers with each iteration. In this category, the SM occupancy is higher (above 60%) when the number of the invoked thread is more than hundreds. In the second category the number of thread remains unchanged with the iterations. In this category, the SM occupancy is higher (above 55%). On the whole, the occupancies of kernels in these two categories are high. However, from our observation, the speedup of the parallel KD-tree construction is not remarkable. There are a few major limitations leading to the lower speedup will be discussed in 6.2.4.

6.2.1.3 SM Occupancy of the Parallel ANNS

The SM occupancy of the ANNS is 44% in the experiment with 100900 256-dimensional descriptors. The maximum register per thread is 64 on the GTX 680 GPU, the platform used for matching test and data profiling. The higher occupancy for the ANNS with respect to parallel SHOT descriptor is due to two factors. First, the available register per thread on GTX 680 is higher than that of the GTX 570. Second, the workload and its complexity per thread of the ANNS is lower than that of the parallel SHOT descriptor.

6.2.1.4 SM Occupancy of the Parallel BANNS

As for the SM occupancy of the BANNS, with lower tree height and less branches in the BANNS, the SM occupancy of forward search kernel is 49%. The SM occupancy of the local NNS search kernel and the brute force search kernel are 58% and 64% respectively. The SM occupancies of the BANNS kernels are higher than that of the regular ANNS because the BANNS parallel algorithms break the word load of single thread in the ANNS into thread stages with three different kernels. Even thought the occupancies are improved, more synchronization and kernel launch cost are introduced.

6.2.2 Memory Utilization

The memory utilization is high in the design of descriptor computing, indexing and matching. The high memory consumption stems from three factors. First, the original input datasets are with large data size and high dimensionality. Also, due to big data size, the shared memory on the single SM is not large enough to hold the intermediate and temporal variables used in the algorithm. So, we need to use the global memory for these variables. Moreover, to improve the speed performance of the parallel algorithms, we sacrifice the space complexity for time complexity in the design. However, the standard memory configuration range from 2 GB to 4 GB. The GTX 570 and GTX 680 are with 2 GB and 4 GB DDR5 memory respectively in our experiment. In the descriptor computing and matching stages studied in this work, the parallel KD-tree construction on the GPU consumes the maximum global memory, which can exceed the maximum global memory size. For this reason, the dimensionality upper bound is set to 512 in our experiment for all the test cases.

6.2.3 Communication Cost

The parallel and distributed BANNS algorithms on the GPU cluster involves the inter-GPUs communications. There are two general cases of the communications.

One is the communication among GPUs within single cluster node via peer-to-peer (P2P) or shared host memory. The other is the communication among GPUs across the cluster nodes via host-side message passing such as the MPI. The MPI related communications in this algorithm including the launching of MPI, broadcasting of data, scattering of data, and synchronization which result in large communication overhead. There are 16 GPUs on the cluster we used. Ideally, the speedup benefitting from the cluster can be 16 if there is no communication cost. However, we achieve speedup of about 8 with respect to the parallel BANNS on single GPU, in the experiment with 1009000 256-dimensional descriptors. We only utilized about 50% of the cluster computing capability due to communication cost and memory transfer cost.

6.2.4 Other Constraints

There are still other factors restricting the speed performance of the parallel algorithms we studied in the work, such as the branch conditions, kernel launch cost and synchronization cost on the GPU.

From our observation, even though the occupancy of the KD-tree construction kernels are hight, but the speedup of the parallel KD-tree construction is not remarkable.

There are a few major limitations leading to the lower speedup. First, the KD-tree construction works in the BFS fashion. In this scenario, when the number of involved threads shrinks to a lower number, the kernels work with low efficiency. Second, a lot of coordinations are need on the host (the CPU). Thirds, the number of kernel launches is extremely high. The cost to launch kernel is expensive, since it involve global synchronization among all the threads. The frequent kernel launches will limit the speedup improvement irrespective of the number of the SPs in the SM architecture.

Moreover, the parallel NNS and it variants in this work involve branching during the forward traversal. Even though we design quite a few strategies to mitigate the impact of branch divergency on the GPU, it still can not be avoided completely. Branch divergence reduce the parallel performance due to lower utilization of the execution units, since the branches will sequentialized during the execution. The branch conditions the algorithm threshold the further performance improvement.

6.3 Future Work

Heterogeneous computing architecture with higher number of cores and more friendly developing ecosystem are likely to remain as the dominant computing platform in the future. However, there are still some challenges and constraints to develop high efficient and scalable parallel algorithms for the point cloud base 3D object recognition algorithm. A few possible future research directions regarding high efficient 3D object recognition are proposed in this subsection.

6.3.1 Data Compression

As we see in the preceding session, memory consumption is one of the key constraints to the scalability of the parallel algorithms. Recently, there have been a few attempts at compacting 2D image descriptors to allow for faster matching while retaining outstanding recognition precision. These proposed techniques relies on quantization [130], [131] and dimensionality reduction [132], [133]. Inspired by these proposals, the 3D image descriptor data set can be quantized or compressed to reduce the memory consumption in the future work. For the SHOT descriptor, we use double data type with 64 bits to store the values for each of the 320 dimensions. There are ten bins in one of the 32 divisions in the sphere of each key point. The value of each bin, the cosine of key point normal and the LRF, ranges from -1 to 1. So, we can only use 4 bits to quantize each bin value. In this way, the memory size can be reduce by 93.75%. For the synthetic descriptor with varying dimensionality, we use float data type with 32 bits for each dimension. Since the value of each dimension data ranges from 0 to 255. We only need 8 bits to quantize each bin value. In this way, the memory size can be reduce by 75%.

6.3.2 Binarization and Hashing

Data compression can mitigate the memory problem, but it will introduce the extra padding and extracting operations during the computing. The works in [134] [135] [136] [137] [138] took advantage of the binary code and hashing for the high dimensional descriptors construction and matching. The binarization is performed by multiplying the regular high dimensional descriptors by a projection matrix, mapping the original data into binary string. Each bit of the binary descriptor is used to represent the feature of descriptor, and the distance of two descriptors are measured through Hamming metric. The index construction and matching are performed through hash table construction and hash searching respectively. Both of them can be performed efficiently since the time complexity is reduced greatly with hash table. The challenges lie on the regular descriptor projection approaches and hashing functions selection. Improper projection and hashing leads to low matching precision. Enlightened by these proposals the challenges in this work, in the future, we can further explore effective binary representation of the SHOT descriptor on the GPU without descriptiveness loss. Also, instead of using with the Euclidean metric, Hamming distance can be used to compute the similarity of two descriptors. Moreover, for the descriptor indexing and matching, we can use hash table in place of the space partition like KD-tree. Since the single NNS on the hash table can also be processed independently in the batch query, we can further explore the high efficient parallel hash table based NNS on the GPU.

References

[1] L. Hu and S. Nooshabadi, "Parallel 3d local descriptor on nvidia gpu," in *GPU Technology Conference 2014 (GTC 2014)*, (San Jose, CA), January May.

[2] L. Hu and S. Nooshabadi, "A 3D local descriptor SHOT on massively parallel processors," in *IEEE International Conference on Consumer Electronics (ICCE)*, (Las Vegas, NV), pp. 186–187, January 2015.

[3] L. Hu and S. Nooshabadi, "G-SHOT: GPU accelerated 3D local descriptor for surface matching," *Journal of Visual Communication and Image Representation (JVCI)*, vol. 30, pp. 343–349, July 2015.

[4] L. Hu and S. Nooshabadi, "Massively parallel KD-tree construction and nearest neighbor search algorithms," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, (Lisbon, Portugal), pp. 2752–2755, May 2015.

[5] L. Hu, S. Nooshabadi, and M. Ahmadi, "Parallel randomized kd-tree forest on

gpu cluster for image descriptor matching," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pp. 582–585, IEEE, 2016.

[6] L. Hu and S. Nooshabadi, "Massive parallelization of approximate nearest neighbor search on kd-tree for high-dimensional image descriptor matching," *Journal of Visual Communication and Image Representation (JVCI)*, vol. 44, p. 106115, November 2017.

[7] C. Astua, R. Barber, J. Crespo, and A. Jardon, "Object detection techniques applied on mobile robot semantic navigation," *Sensors*, vol. 14, pp. 6734–6757, April 2014.

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and F.-F. Li, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[9] M. Lu, Y. Guo, J. Zhang, Y. Ma, and Y. Lei, "Recognizing objects in 3d point clouds with multi-scale local features," *Sensors*, vol. 14, no. 12, pp. 24156–24173, 2014.

[10] R. B. Rusu, Z. C. Marton, N. Blodow, M. Dolha, and M. Beetz, "Towards 3d point cloud based object maps for household environments," *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 927–941, 2008.

[11] A. Golovinskiy, V. G. Kim, and T. Funkhouser, "Shape-based recognition of 3d point clouds in urban environments," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2154–2161, IEEE, 2009.

[12] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[13] A. Aldoma, Z.-C. Marton, F. Tombari, W. Wohlkinger, C. Potthast, B. Zeisl, R. B. Rusu, S. Gedikli, and M. Vincze, "Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation," *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 80–91, 2012.

[14] Y. Li and M. Hielsberg, "A tutorial for 3d point cloud editor," in *http://people.cse.tamu.edu/yli/software/pceditor.html*, 2012.

[15] Y. Guo, M. Bennamoun, F. Sohel, M. Lu, and J. Wan, "3d object recognition in cluttered scenes with local surface features: a survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2270–2287, 2014.

[16] S. S. Bucak, R. Jin, and A. K. Jain, "Multiple kernel learning for visual object recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 7, pp. 1354–1369, 2014.

[17] T. Dongping *et al.*, "A review on image feature extraction and representation techniques," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 8, no. 4, pp. 385–396, 2013.

[18] A. Gordo, J. Almazán, J. Revaud, and D. Larlus, "Deep image retrieval: Learning global representations for image search," in *European Conference on Computer Vision*, pp. 241–257, Springer, 2016.

[19] L. Feng, J. Wu, S. Liu, and H. Zhang, "Global correlation descriptor: a novel image representation for image retrieval," *Journal of Visual Communication and Image Representation*, vol. 33, pp. 104–114, 2015.

[20] H.-Y. Wu, H. Zha, T. Luo, X.-L. Wang, and S. Ma, "Global and local isometry-invariant descriptor for 3d shape comparison and partial matching," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 438–445, IEEE, 2010.

[21] R. Mohamed, M. Mohamed, *et al.*, "A hybrid feature extraction for satellite image segmentation using statistical global and local feature," in *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*, pp. 247–255, Springer, 2016.

[22] D. Zhang, J. He, Y. Zhao, Z. Luo, and M. Du, "Global plus local: a complete framework for feature extraction and recognition," *Pattern Recognition*, vol. 47, no. 3, pp. 1433–1442, 2014.

[23] I. S. Kwak, A. C. Murillo, P. N. Belhumeur, D. J. Kriegman, and S. J. Belongie, "From bikers to surfers: Visual recognition of urban tribes.," in *BMVC*, vol. 1, p. 2, 2013.

[24] F. Rothganger, S. Lazebnik, C. Schmid, and J. Ponce, "3d object modeling and recognition using local affine-invariant image descriptors and multi-view spatial constraints," *International Journal of Computer Vision*, vol. 66, no. 3, pp. 231–259, 2006.

[25] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[26] C. Tomasi and T. Kanade, "Shape and motion from image streams: a factorization method," *Proceedings of the National Academy of Sciences*, vol. 90, no. 21, pp. 9795–9802, 1993.

[27] J. Tang, S. Miller, A. Singh, and P. Abbeel, "A textured object recognition pipeline for color and depth image data," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 3467–3474, IEEE, 2012.

[28] T. Schlömer, B. Poppinga, N. Henze, and S. Boll, "Gesture recognition with a wii controller," in *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pp. 11–14, ACM, 2008.

[29] L. A. Alexandre, "3d descriptors for object and category recognition: a comparative evaluation," in *Workshop on Color-Depth Camera Fusion in Robotics*

at the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vilamoura, Portugal, vol. 1, p. 7, 2012.

[30] S. Salti, F. Tombari, R. Spezialetti, and L. Di Stefano, "Learning a descriptor-specific 3d keypoint detector," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2318–2326, 2015.

[31] S. Filipe and L. A. Alexandre, "A comparative evaluation of 3d keypoint detectors in a rgb-d object dataset," in *Computer Vision Theory and Applications (VISAPP), 2014 International Conference on*, vol. 1, pp. 476–483, IEEE, 2014.

[32] S. Filipe and L. A. Alexandre, "A comparative evaluation of 3d keypoint detectors," in *9th Conference on Telecommunications, Conftele*, pp. 145–148, 2013.

[33] F. Tombari, S. Salti, and L. Di Stefano, "Performance evaluation of 3d keypoint detectors," *International Journal of Computer Vision*, vol. 102, no. 1-3, pp. 198–220, 2013.

[34] Y. Guo, M. Bennamoun, F. Sohel, M. Lu, J. Wan, and J. Zhang, "Performance evaluation of 3d local feature descriptors," in *Asian Conference on Computer Vision*, pp. 178–194, Springer, 2014.

[35] A. Zeng, S. Song, M. Niessner, M. Fisher, J. Xiao, and T. Funkhouser, "3dmatch: Learning local geometric descriptors from rgb-d reconstructions," *arXiv preprint arXiv:1603.08182*, 2016.

[36] D. Gragnaniello, G. Poggi, C. Sansone, and L. Verdoliva, "An investigation of local descriptors for biometric spoofing detection," *IEEE transactions on information forensics and security*, vol. 10, no. 4, pp. 849–863, 2015.

[37] S. Berretti, N. Werghi, A. Del Bimbo, and P. Pala, "Matching 3d face scans using interest points and local histogram descriptors," *Computers & Graphics*, vol. 37, no. 5, pp. 509–525, 2013.

[38] F. Tombari, S. Salti, and L. D. Stefano, "Unique signatures of histograms for local surface description," in *European Conference on Computer Vision (ECCV)*, (Hersonissos, Greece), September 5-11 2010.

[39] Y. Ke and R. Sukthankar, "Pca-sift: A more distinctive representation for local image descriptors," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 2, p. II, IEEE, 2004.

[40] Y. Guo, F. A. Sohel, M. Bennamoun, M. Lu, and J. Wan, "Trisi: A distinctive local surface descriptor for 3d modeling and object recognition.," in *GRAPP/I-VAPP*, pp. 86–93, 2013.

[41] F. Stein and G. MEDIONI, "Structual indexing: Efficient 3d object recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence,*, vol. 5, pp. 1645 –1650, July 1992.

[42] C. Chua and R. Jarvis, “Point signatures: A new representation for 3d object recognition,” *International Journal of Computer Vision*, vol. 25, pp. 63–85, October 1997.

[43] Y. Sun and M. Abidi, “Surface matching by 3d points fingerprint.,” in *IEEE International Conference on Computer Vision (ICCV)*, vol. 2, pp. 263–268, 2001.

[44] J. Novatnack and K. Nishino, “Scale-dependent/invariant local 3d shape descriptors for fully automatic registration of multiple sets of range images,” in *European Conference on Computer Vision (ECCV)*, p. Part III, 2008.

[45] A. Johnson and M. Hebert, “Using spin images for effcient object recognition in cluttered 3d scenes,” in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 433–449, 1999.

[46] H. Chen and B. Bhanu, “3d free-form object recognition in range images using local surface patches,” in *Pattern Recognition Letters 28*, pp. 1252–1262, 2007.

[47] J. Koenderink and A. Doorn, “Surface shape and curvature scales.,” in *Image Vision Computing 8*, pp. 557–565, 1992.

[48] A. Frome, D. Huber, R. Kolluri, T. Bulow, and J. Malik, “Recognizing objects in range data using regional point descriptors,” in *European Conference on Computer Vision (EEVC)*, 2004.

[49] Y. Zhong, "Intrinsic shape signatures: A shape descriptor for 3d object recognition.," in *IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)*, 2009.

[50] F. Tombari, S. Salti, and L. D. Stefano, "A combined texture-shape descriptor for enhanced 3d feature matching," in *IEEE International Conference on Image Processing (ICIP)*, (Brussels, Belgium), September 11-14 2011.

[51] T. Brox and J. Malik, "Large displacement optical flow: descriptor matching in variational motion estimation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 3, pp. 500–513, 2011.

[52] F. Bellavia, D. Tegolo, and C. Valenti, "Keypoint descriptor matching with context-based orientation estimation," *Image and Vision Computing*, vol. 32, no. 9, pp. 559–567, 2014.

[53] E. Shechtman and M. Irani, "Matching local self-similarities across images and videos," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pp. 1–8, IEEE, 2007.

[54] N. S. Altman, "An introduction to kernel and nearest neighbors nonparametric regression," *The American Statistician*, vol. 46, pp. 175–185, August 1992.

[55] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[56] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics*, pp. 126–137, 2008.

[57] J. Kim, W. Jeong, and B. Nam, "Exploiting massive parallelism for indexing multi-dimensional datasets on the GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 2258–2271, August 2015.

[58] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[59] F. Tombari, S. Salti, and L. D. Stefano, "Unique signatures of histograms for local surface description," in *11th European Conference on Computer Vision (ECCV)*, (Hersonissos, Greece), September 5-11 2010.

[60] F. Tombari, S. Salti, and L. D. Stefano, "A combined texture-shape descriptor for enhanced 3D feature matching," in *IEEE International Conference on Image Processing (ICIP)*, (Brussels, Belgium), pp. 809–812, September 2011.

[61] J. Sivic and A. Zisserman, "Video google: a text retrieval approach to object matching in videos," in *The Ninth IEEE International Conference on Computer Vision*, (Nice, France), pp. 1470–1477, October 2003.

[62] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE Computer Vision and Pattern Recognition (CVPR)*, (Miami, FL), pp. 248 – 255, June 2009.

[63] L.-J. Li, H. Su, Y. Lim, and L. Fei-Fei, "Object bank: An object-level image representation for high-level visual recognition," *International Journal of Computer Vision (IJCV)*, vol. 107, pp. 20–39, September 2014.

[64] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, pp. 2227–2240, November 2014.

[65] R. Horaud and T. Skordas, "Stereo correspondence through feature grouping and maximal cliques," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1168–1180, 1989.

[66] F. Schaffalitzky and A. Zisserman, "Geometric grouping of repeated elements within images," *Shape, contour and grouping in computer vision*, pp. 81–81, 1999.

[67] V. M. Govindu, "A tensor decomposition for geometric grouping and segmentation," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 1150–1157, IEEE, 2005.

[68] J.-C. Bazin, H. Li, I. S. Kweon, C. Demonceaux, P. Vasseur, and K. Ikeuchi, "A branch-and-bound approach to correspondence and grouping problems," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 7, pp. 1565–1576, 2013.

[69] A. Aldoma, Z.-C. Marton, F. Tombari, W. Wohlkinger, C. Potthast, B. Zeisl, R. B. Rusu, S. Gedikli, and M. Vincze, “Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation,” *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 80–91, 2012.

[70] K.-J. Yoon and I. S. Kweon, “Adaptive support-weight approach for correspondence search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 650–656, 2006.

[71] F. Schaffalitzky and A. Zisserman, “Geometric grouping of repeated elements within images,” *Shape, contour and grouping in computer vision*, pp. 81–81, 1999.

[72] R. M. Haralick, “Computer vision theory: The lack thereof,” *Computer Vision, Graphics, and Image Processing*, vol. 36, no. 2-3, pp. 372–386, 1986.

[73] A. Aldoma, F. Tombari, L. Di Stefano, and M. Vincze, “A global hypotheses verification method for 3d object recognition,” *Computer Vision–ECCV 2012*, pp. 511–524, 2012.

[74] A. Aldoma, F. Tombari, J. Prankl, A. Richtsfeld, L. Di Stefano, and M. Vincze, “Multimodal cue integration through hypotheses verification for rgb-d object recognition and 6dof pose estimation,” in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 2104–2111, IEEE, 2013.

[75] S. Choi, T. Kim, and W. Yu, "Performance evaluation of ransac family," *Journal of Computer Vision*, vol. 24, no. 3, pp. 271–300, 1997.

[76] G. D. Sullivan, K. D. Baker, A. D. Worrall, C. Attwood, and P. Remagnino, "Model-based vehicle detection and classification using orthographic approximations," *Image and vision computing*, vol. 15, no. 8, pp. 649–54, 1997.

[77] O. Chum and J. Matas, "Matching with prosac-progressive sample consensus," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 220–226, IEEE, 2005.

[78] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *Computer*, vol. 26, no. 6, pp. 18–27, 1993.

[79] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed, "A heterogeneous platform with gpu and fpga for power efficient high performance computing," in *Integrated Circuits (ISIC), 2014 14th International Symposium on*, pp. 220–223, IEEE, 2014.

[80] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 69, 2015.

[81] L. Hu, S. Nooshabadi, and T. Mladenov, "Implementation and evaluation of raptor code on gpu," in *Consumer Electronics (ISCE), 2012 IEEE 16th International Symposium on*, pp. 1–6, IEEE, 2012.

[82] L. Hu, S. Nooshabadi, and T. Mladenov, "Forward error correction with raptorq code on gpu," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 281–284, IEEE, 2013.

[83] L. Hu, S. Nooshabadi, and T. Mladenov, "Forward error correction with raptor gf (2) and gf (256) codes on gpu," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 1, pp. 273–280, 2013.

[84] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, 2015.

[85] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0.* Morgan Kaufmann, 2015.

[86] J. Waltz, "Performance of a three-dimensional unstructured mesh compressible flow solver on nvidia fermi-class graphics processing unit hardware," *International Journal for Numerical Methods in Fluids*, vol. 72, no. 2, pp. 259–268, 2013.

[87] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 487–498, ACM, 2013.

[88] J. Lai and A. Seznec, “Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus,” in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–10, IEEE, 2013.

[89] J. S. Beis and D. G. Lowe, “Shape indexing using approximate nearest-neighbour search in high-dimensional spaces,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Puerto Rico), pp. 1000–1006, June 1997.

[90] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, pp. 509–517, Sept. 1975.

[91] C. Silpa-Anan and R. Hartley, “Optimised kd-trees for fast image descriptor matching,” in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, IEEE, 2008.

[92] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, “Physically-based visual simulation on graphics hardware,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 109–118, Eurographics Association, 2002.

[93] J. Fung and S. Mann, “Openvidia: parallel gpu computer vision,” in *Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 849–852, ACM, 2005.

[94] Y. Kitaaki, H. Okuda, H. Kage, and K. Sumi, “High speed 3-d registration using gpu,” in *SICE Annual Conference, 2008*, pp. 3055–3059, IEEE, 2008.

[95] D. Qiu, S. May, and A. Nüchter, “Gpu-accelerated nearest neighbor search for 3d registration.,” in *ICVS*, pp. 194–203, Springer, 2009.

[96] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, “Gpu-based video feature tracking and matching,” in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278, p. 4321, 2006.

[97] NVIDIA, “Nvidia cuda compute unified device architecture programming guide 4.2,” in *http://www.nvidia.com/*, 2011.

[98] NVIDIA, “Fermi compute architecture white paper v1.1,” in *http://www.nvidia.com/*, 2009.

[99] NVIDIA, “Nvidia fermin gtx570 companion processor,” in *http://www.nvidia.com/*, 2009.

[100] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[101] V. Volkov, “Better performance at lower occupancy,” in *GPU Technology Conference (GTC)*, 2010.

[102] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures,," in *International Symposium on High Performance Computer Architecture (HPCA)*, (New Orleans, Louisiana), pp. 1–12, February 2012.

[103] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE MICRO*, vol. 31, pp. 50–59, March 2011.

[104] Y. Kitaaki, H. Okuda, H. Kage, and K. Sumi, "High speed 3d registration using gpu," in *International conference on Instrumentation, Control, Information Technology and System Integration Annual Conference*, pp. 3055–3059, 2008.

[105] D. Qiu, S. May, and A. Nuchter, "Gpu-accelerated nearest neighbor search for 3d registration," in *7th International Conference on Computer Vision Systems*, pp. 194–203, Oct 2009.

[106] S. N. Sinha, J. Frahm, M. Pollefeyes, and Y. Genc, "Gpu-based video feature tracking and matching," in *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[107] S. Arya and D. M. Mount, "Algorithms for fast vector quantization," in *Data Compression Conference (DCC)*, (Snowbird, Utah), pp. 381–390, March-April 1993.

[108] P. Indyk, "Nearest neighbors in high-dimensional spaces," in *Handbook of Discrete and Computational Geometry, 2nd Ed.* (C. D. Toth, J. E. Goodman, and J. O'Rourke, eds.), ch. 39, Boca Raton, FL: CRC Press, 2004.

[109] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, (Switzerland), pp. 41–50, 2003.

[110] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Proceedings of the 6th International Conference on Computational Science*, (U.K.), pp. 196–199, May 2006.

[111] T. Rozen, K. Boryczko, and W. Alda, "Gpu bucket sort algorithm with applications to nearest-neighbor search," *Journal of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, Feb. 2008.

[112] S. Li, L. Simons, J. B. Pakaravoor, F. Abbasinejad, J. D. Owens, and N. Amenta, "kANN on the GPU with shifted sorting," in *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, (Paris, France), pp. 39–47, June 2012.

[113] T. M. Chan, "Approximate nearest neighbor queries revisited," in *Proceedings of*

the Thirteenth Annual Symposium on Computational Geometry, (Nice, France), pp. 352–358, June 1997.

[114] S. Liao, M. Lopez, and S. Leutenegger, "High dimensional similarity search with space filling curves," in *Proceedings of International Conference on Data Engineering*, (Heidelberg, Germany), pp. 615–622, April 2001.

[115] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "Fast k nearest neighbor search using gpu," in *Proceedings of Computer Vision and Pattern Recognition Workshops*, (Anchorage, AK), pp. 1–6, June 2008.

[116] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, (New York, NY), pp. 15–22, July 2005.

[117] G. Fabian, H. Justin, O. Cosmin, and I. Christian, "Buffer k-d trees: Processing massive nearest neighbor queries on GPUs," in *Proceedings of the 31st International Conference on Machine Learning (ICML)*, (Bejing, China), pp. 172–180, June 2014.

[118] L. Cayton, "Accelerating nearest neighbor search on manycore systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Shanghai, China), pp. 402–413, May 2012.

[119] NVIDIA, "Nvidia cuda compute unified device architecture programming guide 4.2," in *http://www.nvidia.com/*, 2011.

[120] S. Winder and M. Brown, “Learning local image descriptors,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Minneapolis, MN), pp. 1–8, June 2007.

[121] M. Brown, G. Hua, and S. Winder, “Discriminative learning of local image descriptors,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, pp. 43–57, January 2011.

[122] B. Gaster and L. Howes, “Can GPGPU programming be liberated from the data-parallel bottleneck?,” *IEEE Computer*, vol. 45, pp. 42–52, August 2012.

[123] M. Muja and D. G. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration,” in *International Conference on Computer Vision Theory and Application (VISSAPP)*, (Lisoba, Portugal), pp. 331–340, Febuary 2009.

[124] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, “K-means for parallel architectures using all-prefix-sum sorting and updating steps,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 1602–612, August 2013.

[125] J. Pan and D. Manocha, “Fast GPU-based locality sensitive hashing for k-nearest neighbor computation,” in *19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems ACM-GIS*, (Chicago, IL), pp. 211–220, November 2011.

[126] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, “FAST: fast architecture sensitive tree search on modern cpus and GPUs,” in *Proceedings of the 2010 International conference on Management of data (SIGMOD)*, (Indianapolis, Indiana), pp. 339–350, June 2010.

[127] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA: Morgan Kaufmann, 2006.

[128] F. Tombari, S. Salti, and L. D. Stefano, “A combined texture-shape descriptor for enhanced 3d feature matching,” in *IEEE International Conference on Image Processing (ICIP)*, (Brussels, Belgium), September 11-14 2011.

[129] J. Hoberock and N. Bell, “An introduction to thrust,” in *https://developer.nvidia.com/thrust*, (online), 2015.

[130] T. Tuytelaars and C. Schmid, “Vector quantizing feature space with a regular lattice,” in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pp. 1–8, IEEE, 2007.

[131] S. Winder, G. Hua, and M. Brown, “Picking the best daisy,” in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 178–185, IEEE, 2009.

[132] D. Gabor, “A performance evaluation of local descriptors,” *J. Inst. Electr. Eng*, vol. 93, pp. 429–459, 1946.

[133] G. Hua, M. Brown, and S. Winder, “Discriminant embedding for local image descriptors,” in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pp. 1–8, IEEE, 2007.

[134] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, “Brief: Binary robust independent elementary features,” *Computer Vision–ECCV 2010*, pp. 778–792, 2010.

[135] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua, “Ldahash: Improved matching with smaller descriptors,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 1, pp. 66–78, 2012.

[136] A. Gionis, P. Indyk, R. Motwani, *et al.*, “Similarity search in high dimensions via hashing,” in *VLDB*, vol. 99, pp. 518–529, 1999.

[137] B. Kulis and T. Darrell, “Learning to hash with binary reconstructive embeddings,” in *Advances in neural information processing systems*, pp. 1042–1050, 2009.

[138] M. Bawa, T. Condie, and P. Ganesan, “Lsh forest: self-tuning indexes for similarity search,” in *Proceedings of the 14th international conference on World Wide Web*, pp. 651–660, ACM, 2005.

[139] S. Klupsch, M. Ernst, S.A. Huss, M. Rumpf, and R. Strzodka, “Real time image processing based on reconfigurable hardware acceleration,” in *Proceedings of IEEE Workshop Heterogeneous Reconfigurable Systems on Chip*, 2002.

[140] J. Fung and S. Mann, “OpenVIDIA: Parallel GPU Computer Vision,” in *ACM MULTIMEDIA*, pp. 849–852, 2005.

[141] J. Fung, S. Mann, “Computer Vision Signal Processing on Graphics Processing Units,” in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 93–96, 2004.

[142] T. Jansen and B. von Rymon-Lipinski and N. Hanssen and E. Keeve, “Fourier Volume Rendering on the GPU Using a Split-stream-FFT,” in *Vision, Modelling and Visualization*, pp. 395–403, Nov. 2004.

[143] N.K. Govindaraju, S. Larsen, J. Gray and D. Manocha, “A Memory Model for Scientific Algorithms on Graphics Processors,” in *Super Computing*, 2006.

[144] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, “A Comparison and Evaluation of Multi-view Stereo Reconstruction Algorithms,” in *IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR)*, vol. vol.1, pp. 519–526, Nov.2006.

[145] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann, 2014.

[146] A.-K. C. Ahamed, A. Desmaison, and F. Magoulès, “Fast and green computing with graphics processing units for solving sparse linear systems,” in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security*, pp. 129–136, IEEE, 2014.

[147] M. Ali and T. Ozkul, "Review of memory/cache management technologies used on heterogeneous computing systems," *International Journal of Computer and Information Technology*, vol. 3, no. 3, 2014.

[148] P. K. Das and G. C. Deka, "History and evolution of gpu architecture," in *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*, pp. 109–135, IGI Global, 2016.

[149] I. Z. Emiris and D. Nicolopoulos, "Randomized kd-trees for approximate nearest neighbor search," *CGL-TR-78, NKUA, Tech. Rep.*, 2013.

[150] H. Murase and S. K. Nayar, "Visual Learning and Recognition of 3-D Objects from Appearance," *International Journal of Computer Vision*, vol. 14, pp. 5–24, 1995.

[151] A. Selinger and R. C. Nelson, "A perceptual grouping hierarchy for appearance-based 3d object recognition," *Computer Vision and Image Understanding*, vol. 76, no. 1, pp. 83–92, 1999.

[152] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.

[153] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[154] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbor search in high dimensional spaces," in *CVPR*, pp. 1000–1006, 1997.

[155] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *CVPR*, 2008.

[156] K. Fukunaga and P. Narendra, "A branch and bound algorithm for computing k-nearest neighbors.," in *IEEE Transaction on Copmuter*, pp. 750–753, 1975.

[157] S. Brin, "Near neighbor search in large metric space.," in *VLDB*, pp. 574–784, 1995.

[158] T. Liu, A. W. Moore, K. Yang, and A. G. Gray, "An investigation of practical approximate nearest neighbor algorithms," in *Advances in neural information processing systems*, pp. 825–832, 2005.

[159] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *CVPR*, pp. 2161–2168, 2006.

[160] M. K. Leibe, B. and B. Schiele, "Efcient clustering and matching for object class recognition," in *BMVC*, 2006.

[161] K. Mikolajczyk and J. Matas, "Improving descriptors for fast tree matching by optimal linear projection," in *ICCV*, pp. 1–8, 2007.

[162] A. E. Johnson and M. Hebert, "Surface matching for object recognition in complex 3-d scenes," in *Image and Vision Computing*, 1998.

[163] H. Chen and B. Bhanu, "3d free-form object recognition in range images using local surface patches," in *International Conference on Pattern Recognition (ICPR)*, 2004.

[164] B. Horn, "Closed-form solution of absolute orientation using unit quaternions.," in *Journal of Optical Society of America.*, pp. 629–642, 1987.

[165] P. Besl and N. McKay, "A method for registration of 3d shapes," in *IEEE Transaction of Pattern Analysis and Machine Intelligence*, pp. 239–256, 1992.

[166] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," in *International Journal of Computer Vision (IJCV)*, pp. 119–152, 1994.

[167] P. M. Vaidya, "An o(n log n) algorithm for the all-nearest neighbors problem," *Discrete and Computational Geometry*, vol. 4, pp. 101–115, 1989.

[168] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Proceedings of the 6th International Conference on Computational Science*, pp. 196–199, 2006.

[169] H. W. Jensen, ed., *Realistic image synthesis using photon mapping.* A K Peters/CRC Press, 2001.

[170] NVIDIA, "Fermi compute architecture white paper v1.1," (http://www.nvidia.com/), 2009.

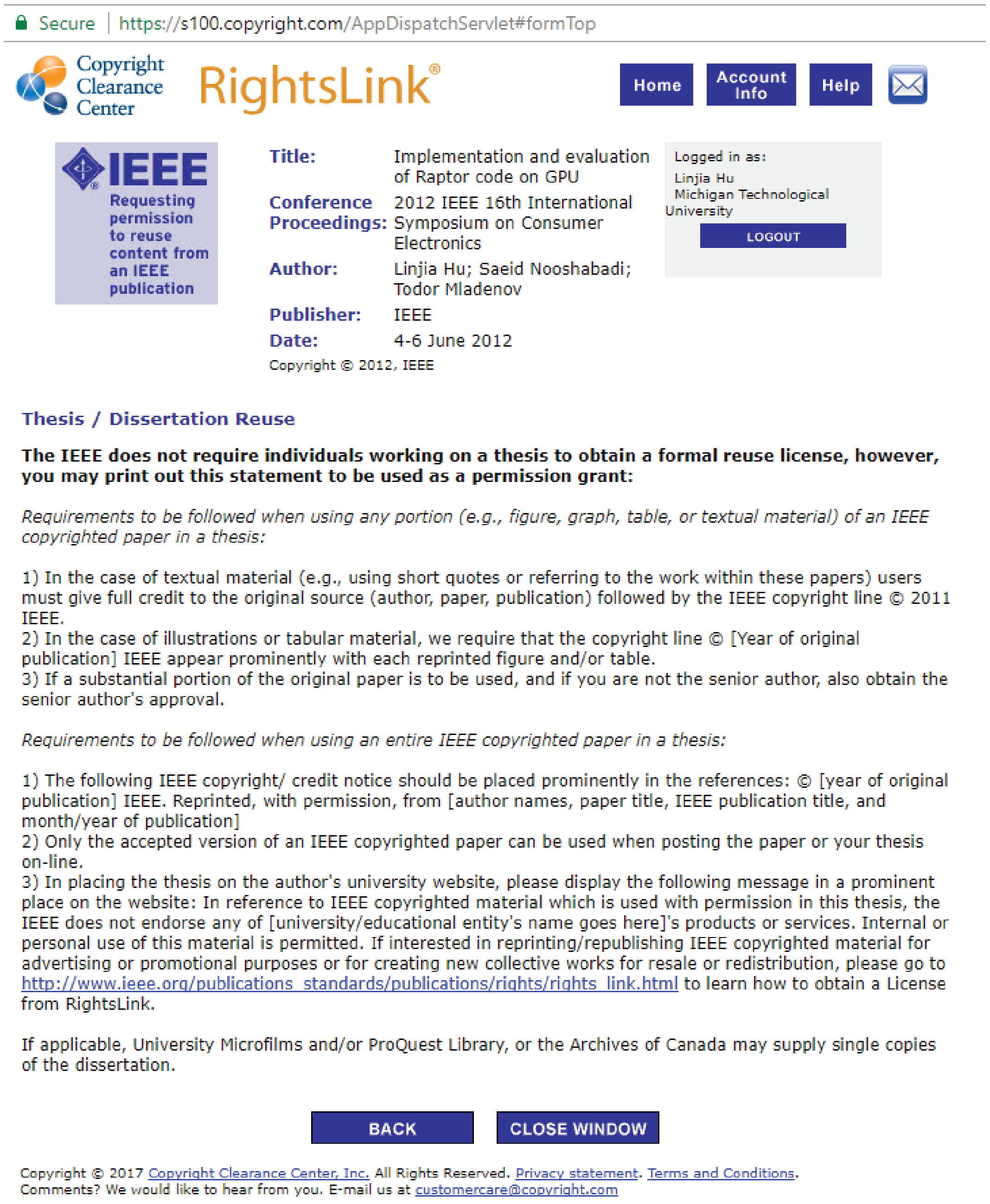
[171] J. Kim, W. Jeong, and B. Nam, "Parallel multi-dimensional range query processing with R-trees on GPU," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 73, pp. 1195–1207, August 2013.

[172] H. W. Jensen, ed., *Realistic image synthesis using photon mapping.* Natick, MA: A K Peters/CRC Press, 2001.

[173] Y. Manolopoulos, A. Nanopoulos, and Y. Theodoridis, eds., *R-Trees: Theory and Applications.* London, UK.: Springer, 2006.

Appendix A

Letters of Permission


Secure | https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication

Title:	Implementation and evaluation of Raptor code on GPU
Conference Proceedings:	2012 IEEE 16th International Symposium on Consumer Electronics
Author:	Linjia Hu; Saeid Nooshabadi; Todor Mladenov
Publisher:	IEEE
Date:	4-6 June 2012

Copyright © 2012, IEEE

Logged in as:
Linjia Hu
Michigan Technological University
LOGOUT

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

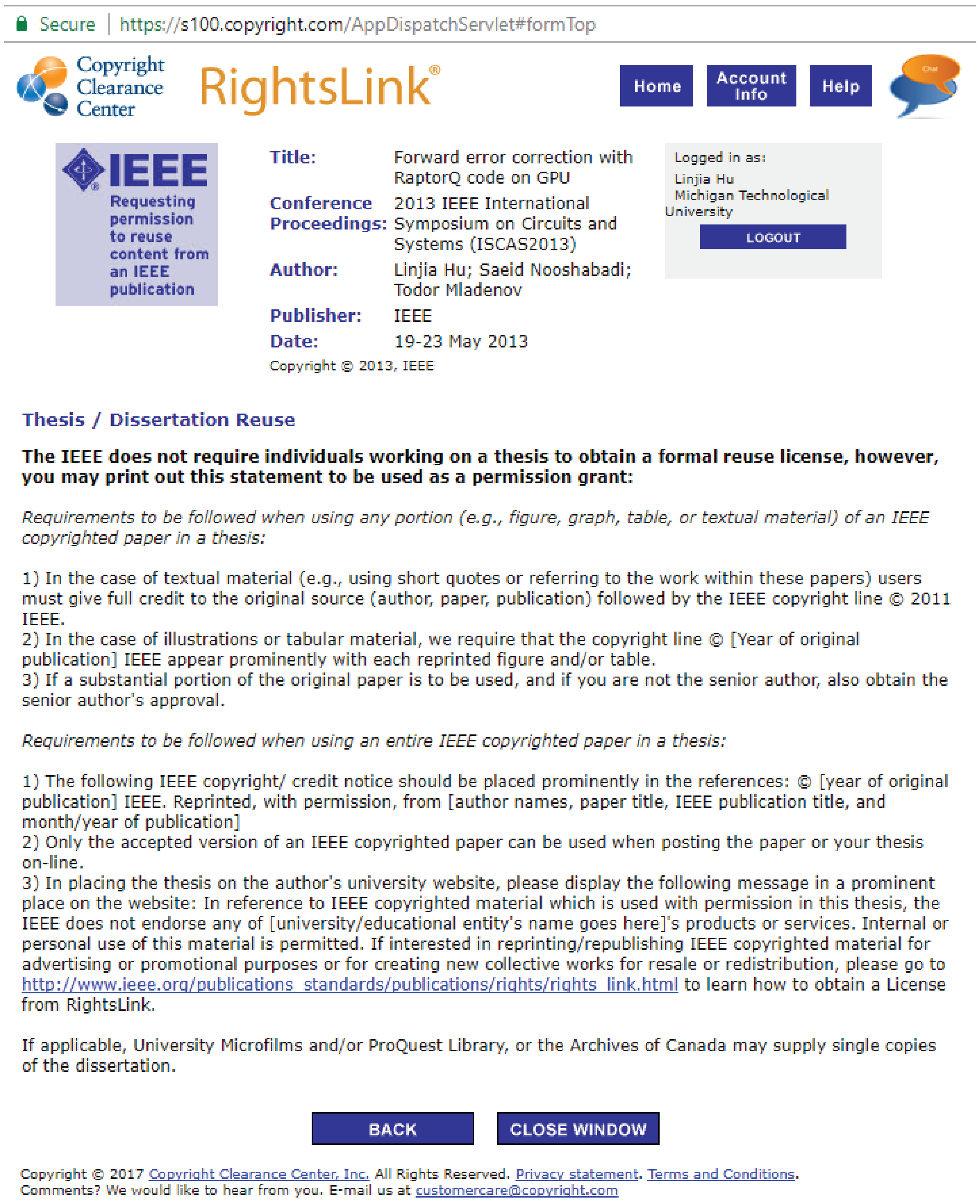
1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Figure A.1: Permission letter form ISCE 2012


Secure | https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication
LOGOUT
BACK
CLOSE WINDOW

Title:	Forward error correction with RaptorQ code on GPU
Conference Proceedings:	2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)
Author:	Linjia Hu; Saeid Nooshabadi; Todor Mladenov
Publisher:	IEEE
Date:	19-23 May 2013

Copyright © 2013, IEEE

Logged in as:
Linjia Hu
Michigan Technological University

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

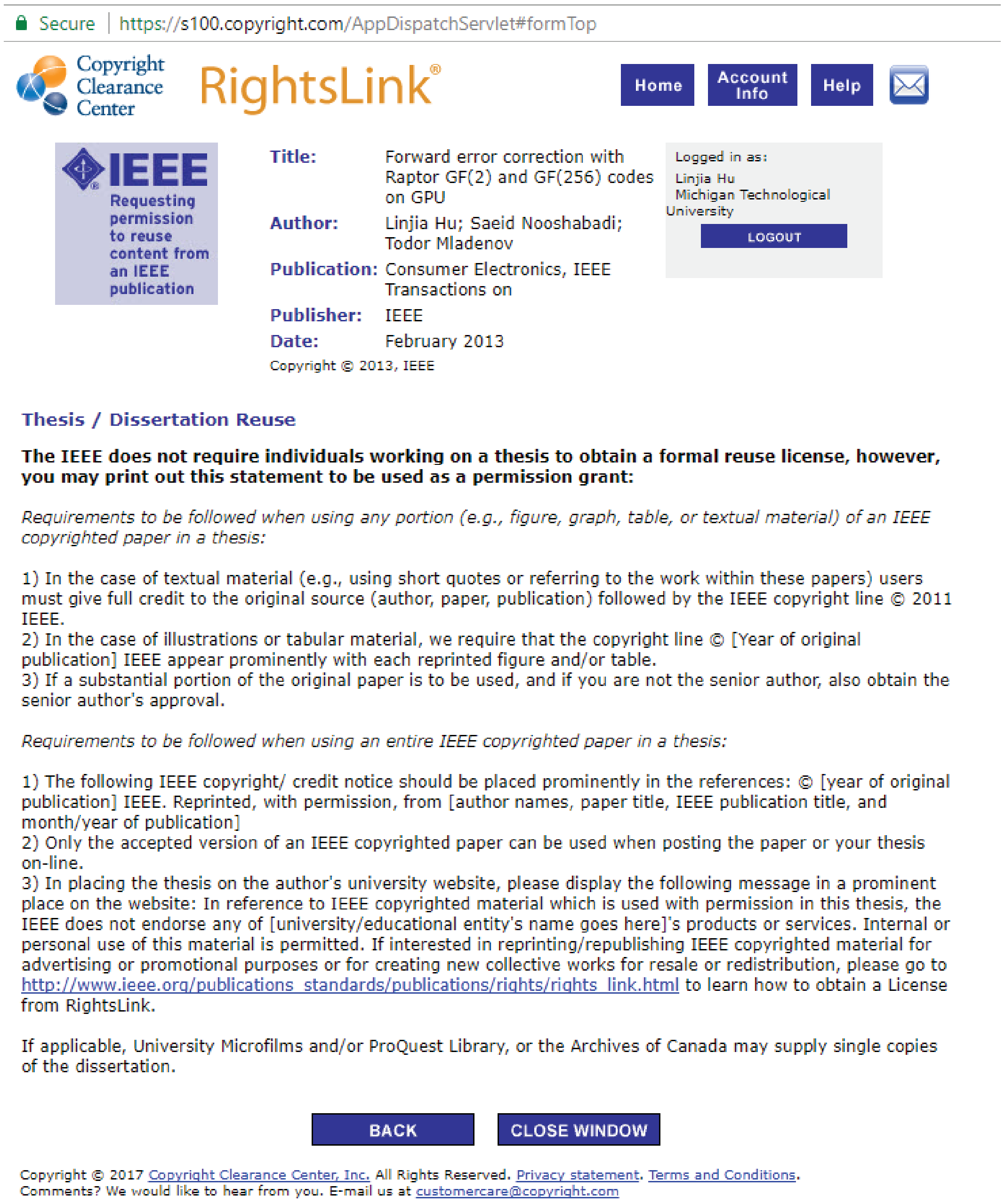
1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Figure A.2: Permission letter form ISCAS 2013


Secure
https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication
LOGOUT
BACK
CLOSE WINDOW

Title:	Forward error correction with Raptor GF(2) and GF(256) codes on GPU
Author:	Linjia Hu; Saeid Nooshabadi; Todor Mladenov
Publication:	Consumer Electronics, IEEE Transactions on
Publisher:	IEEE
Date:	February 2013

Copyright © 2013, IEEE

Logged in as:
Linjia Hu
Michigan Technological University

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

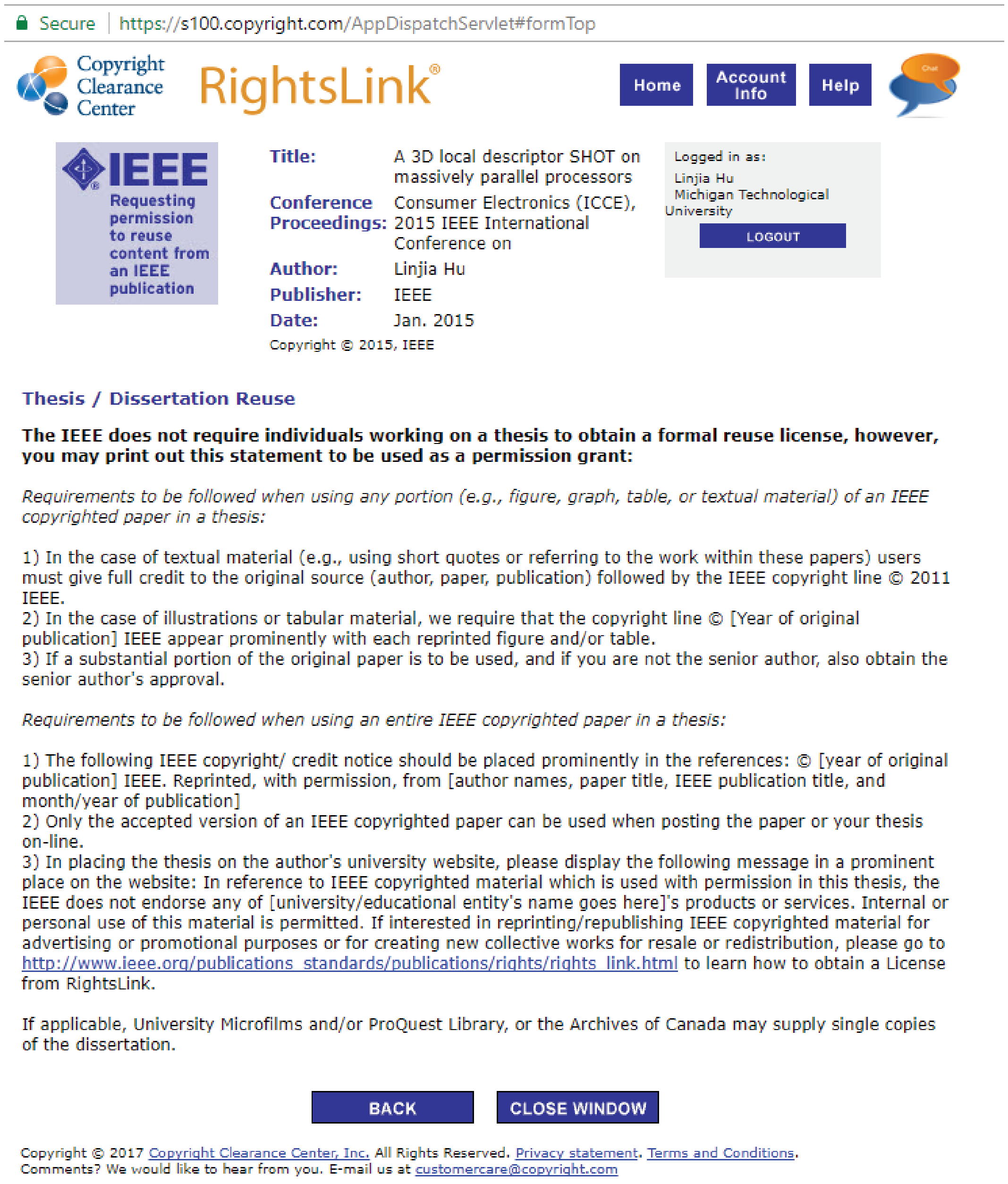
1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Figure A.3: Permission letter form TCE 2013


Secure https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication
LOGOUT
BACK
CLOSE WINDOW

Title:	A 3D local descriptor SHOT on massively parallel processors
Conference Proceedings:	Consumer Electronics (ICCE), 2015 IEEE International Conference on
Author:	Linjia Hu
Publisher:	IEEE
Date:	Jan. 2015

Copyright © 2015, IEEE

Logged in as:
Linjia Hu
Michigan Technological University

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

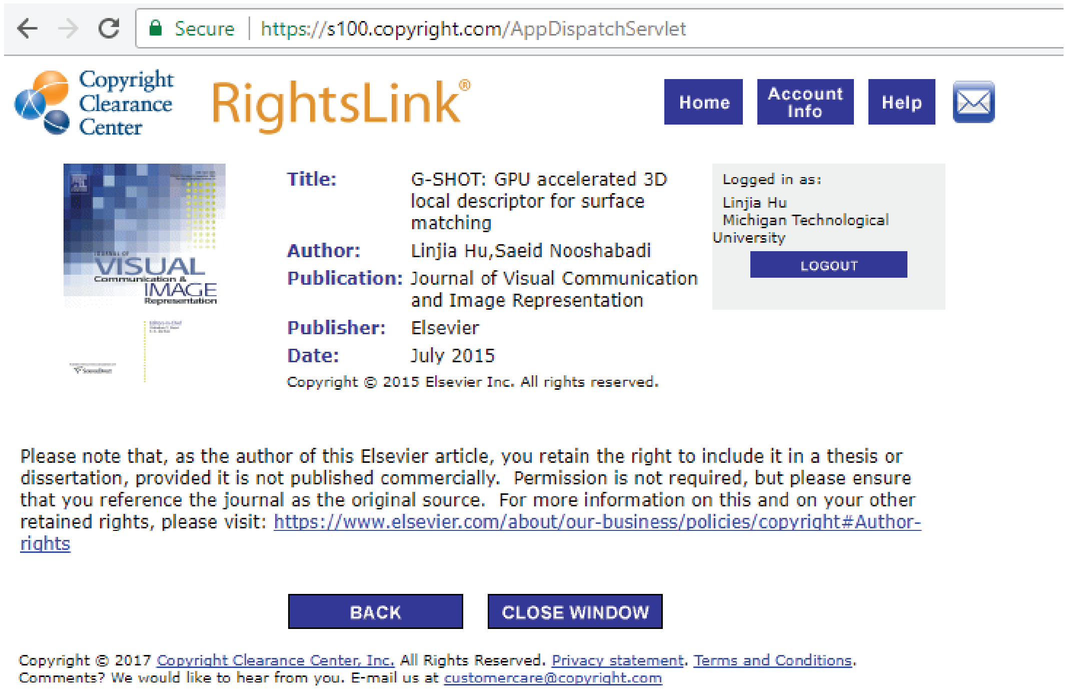
1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Figure A.4: Permission letter form ICCE 2015


Secure
https://s100.copyright.com/AppDispatchServlet
Copyright Clearance Center
RightsLink®
Home
Account Info
Help

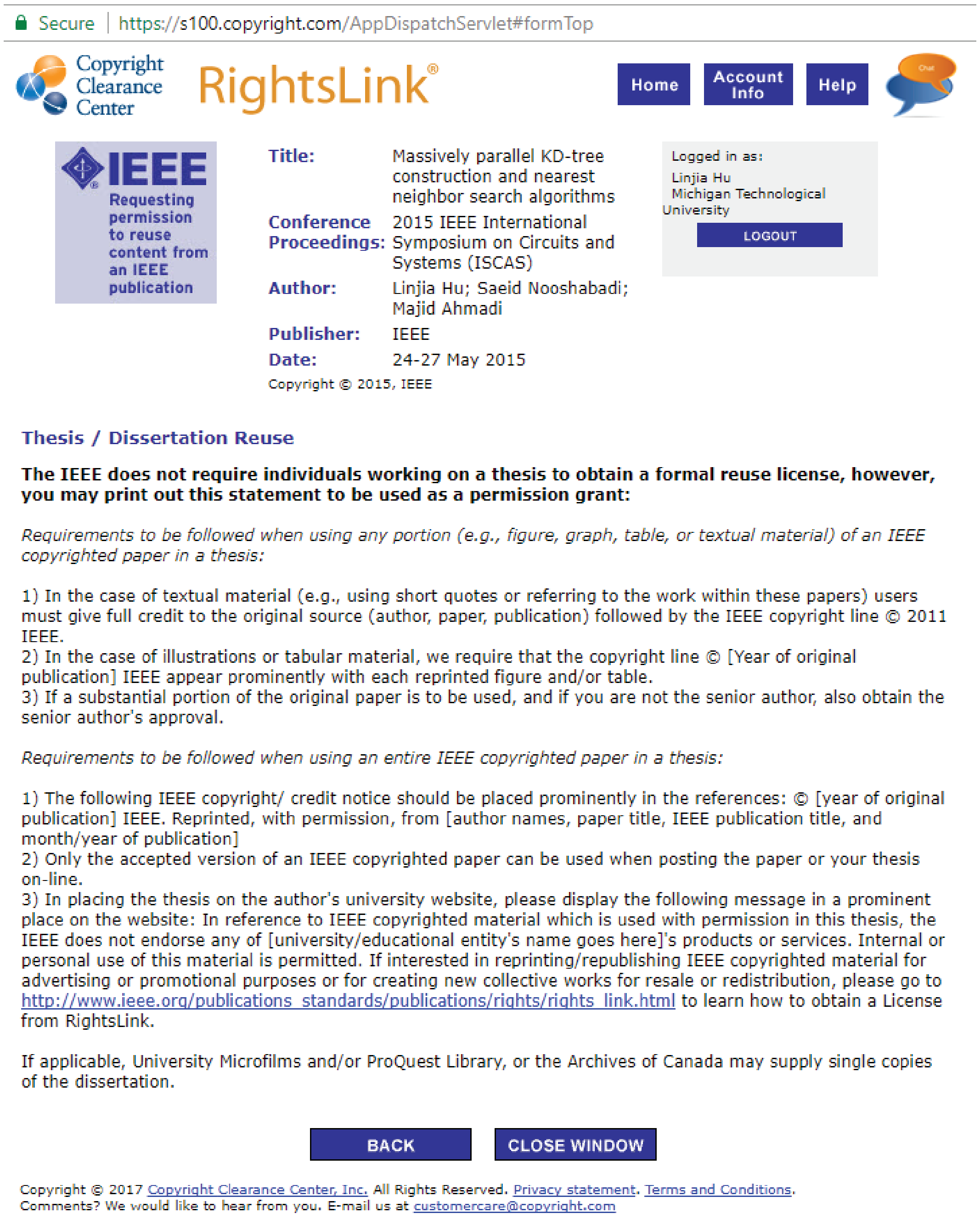
Title: G-SHOT: GPU accelerated 3D local descriptor for surface matching
Author: Linjia Hu,Saeid Nooshabadi
Publication: Journal of Visual Communication and Image Representation
Publisher: Elsevier
Date: July 2015
Copyright © 2015 Elsevier Inc. All rights reserved.

Logged in as:
Linjia Hu
Michigan Technological University

Please note that, as the author of this Elsevier article, you retain the right to include it in a thesis or dissertation, provided it is not published commercially. Permission is not required, but please ensure that you reference the journal as the original source. For more information on this and on your other retained rights, please visit: https://www.elsevier.com/about/our-business/policies/copyright#Author-rights

Copyright © 2017 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

Figure A.5: Permission letter form JVCI 2015


Secure | https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication

Title:	Massively parallel KD-tree construction and nearest neighbor search algorithms
Conference Proceedings:	2015 IEEE International Symposium on Circuits and Systems (ISCAS)
Author:	Linjia Hu; Saeid Nooshabadi; Majid Ahmadi
Publisher:	IEEE
Date:	24-27 May 2015

Copyright © 2015, IEEE

Logged in as:
Linjia Hu
Michigan Technological University

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

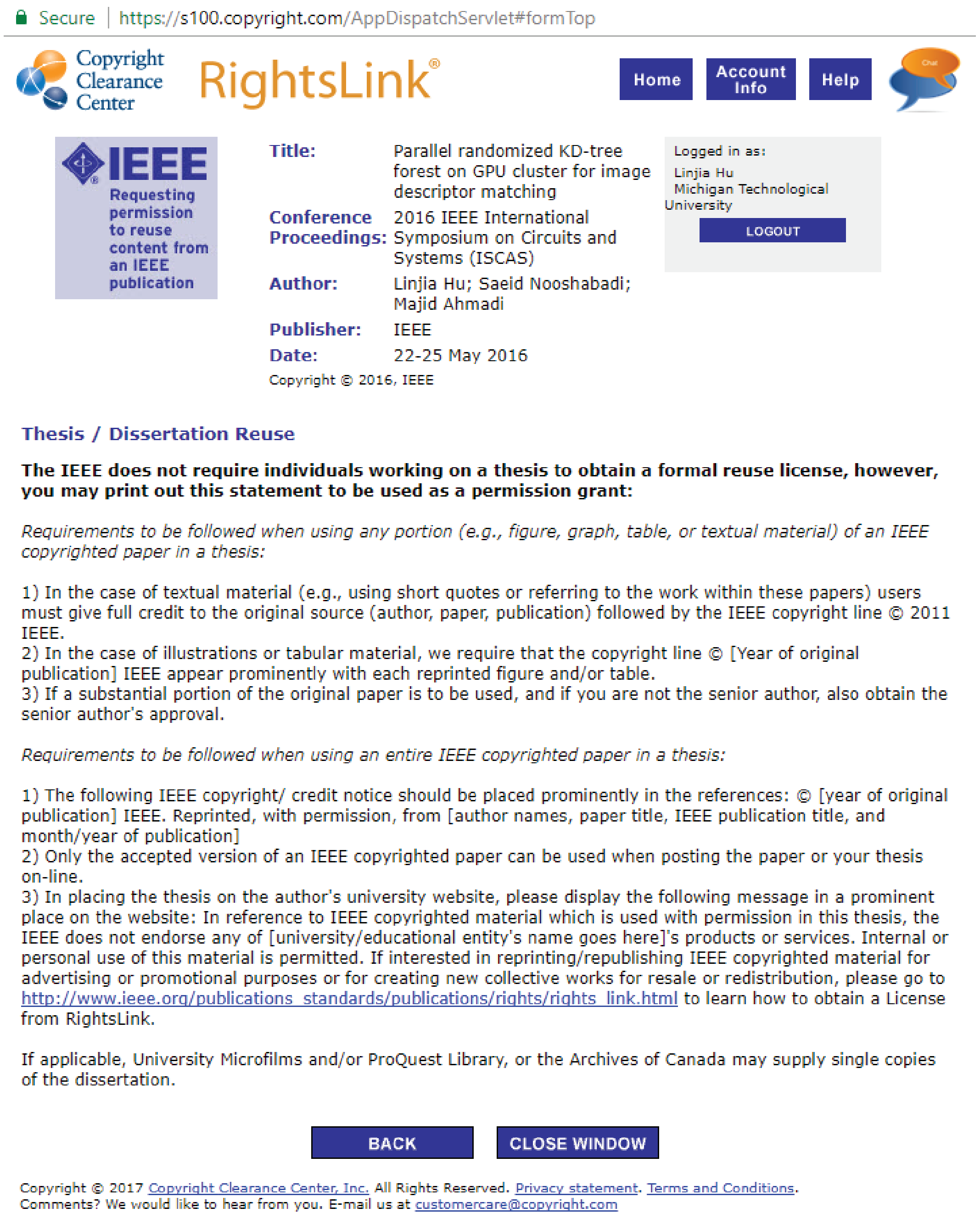
Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2017 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

Figure A.6: Permission letter form ISCAS 2015


Secure | https://s100.copyright.com/AppDispatchServlet#formTop
Copyright Clearance Center
RightsLink®
Home
Account Info
Help
IEEE
Requesting permission to reuse content from an IEEE publication

Title: Parallel randomized KD-tree forest on GPU cluster for image descriptor matching

Conference Proceedings: 2016 IEEE International Symposium on Circuits and Systems (ISCAS)

Author: Linjia Hu; Saeid Nooshabadi; Majid Ahmadi

Publisher: IEEE

Date: 22-25 May 2016

Copyright © 2016, IEEE

Logged in as:
Linjia Hu
Michigan Technological University

LOGOUT

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

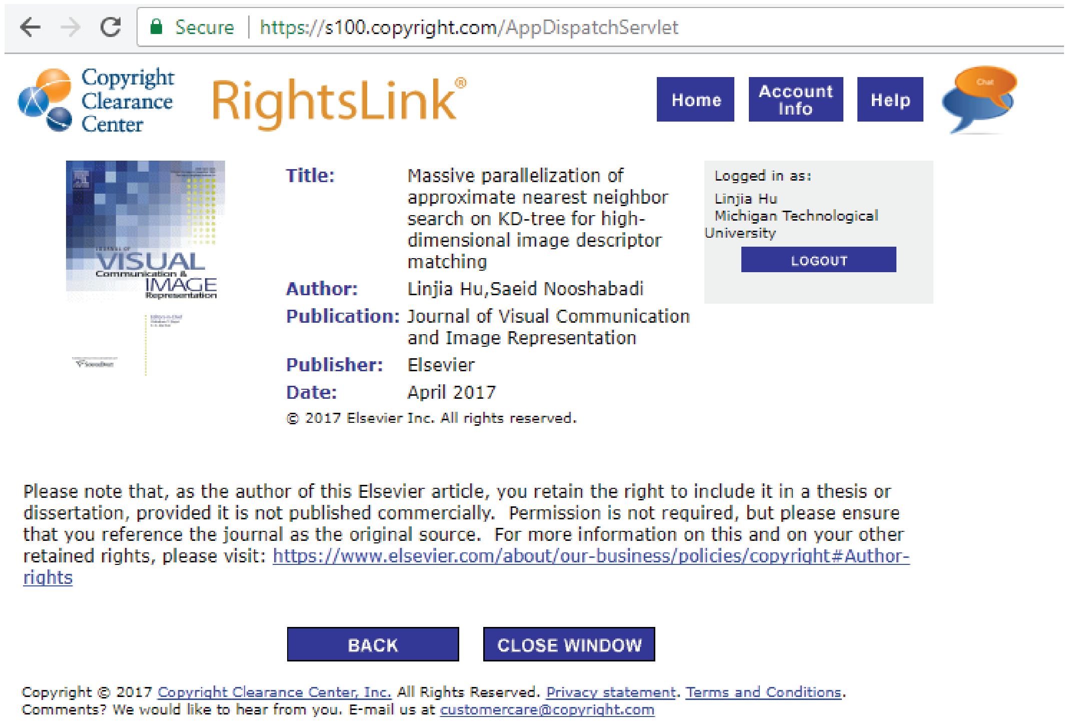
1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Figure A.7: Permission letter form ISCAS 2016


Secure | https://s100.copyright.com/AppDispatchServlet
Copyright Clearance Center
RightsLink
Home
Account Info
Help
Chat

Title: Massive parallelization of approximate nearest neighbor search on KD-tree for high-dimensional image descriptor matching

Author: Linjia Hu,Saeid Nooshabadi

Publication: Journal of Visual Communication and Image Representation

Publisher: Elsevier

Date: April 2017

© 2017 Elsevier Inc. All rights reserved.

Logged in as:
Linjia Hu
Michigan Technological University

Please note that, as the author of this Elsevier article, you retain the right to include it in a thesis or dissertation, provided it is not published commercially. Permission is not required, but please ensure that you reference the journal as the original source. For more information on this and on your other retained rights, please visit: https://www.elsevier.com/about/our-business/policies/copyright#Author-rights

Figure A.8: Permission letter form JVCI 2017