



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

Utilizing general-purpose computing on  
graphic processing units for engineering  
algorithm speedup.

**Teodor Ande Elstad**  
**Simen Haugerud Granlund**

Master of Science in Engineering and ICT

Submission date: June 2014

Supervisor: Ole Ivar Sivertsen, IPM

Co-supervisor: Bjørn Haugen, IPM  
Adel Chemaly, Technosoft Inc.  
Alok Mathur, Technosoft Inc.

Norwegian University of Science and Technology  
Department of Engineering Design and Materials



**MASTER THESIS SPRING 2014  
FOR  
STUD.TECHN. TEODOR ANDRE ELSTAD AND  
SIMEN HAUGERUD GRANLUND**

**UTILIZING GENERAL-PURPOSE COMPUTING ON GRAPHIC PROCESSING UNITS (GPGPU) FOR ENGINEERING ALGORITHM SPEED UP**  
**Utnytte beregningskraft i grafiske prosessorer (GPGPU) til å øke beregningshastighet for ingeniøralgoritmer.**

Most software tools for KBE development is today based around special purpose Modeling frameworks like the AML KBE language developed by TechnoSoft Inc. While this approach has been very successful there are continuously requirements for extending the capabilities both regarding new functionality and computational performance.

TechnoSoft would like to have certain algorithms sped up, by utilizing general-purpose computing on graphics processing units (GPGPU). The goal of the master thesis will be to implement a small library of GPGPU-enhanced algorithms that could be used in TechnoSoft applications.

The technology to be investigated is NVIDIA <http://www.nvidia.com/page/technologies.html> including the CUDA architecture and OpenCL.

Towards the end of the work with the master assignment, when algorithms for grid smoothing for CFD analysis are developed, the candidates will study if these algorithms are well suited for speed up by graphic processor implementation.

The assignment includes:

1. A study of advantages offered by using GPGPU. How to best utilize the high capacity for parallel processing? What kind of problems can be sped up?
2. An evaluation of suitable GPGPU tools. Should development be based on OpenCL or CUDA?
3. An evaluation of GPGPU libraries like ViennaCL and cuBLAS.
4. Select engineering algorithms for test applications, in cooperation with the advisers, that are well suited and relevant for speed up by implementation on the graphic processor.

5. Extensive performance testing of different implementations on different problem sizes and hardware. Are we getting good speed up on real life problems and hardware, or is the overhead associated with parallel computing hampering performance?

Three weeks after start of the thesis work, an A3 sheet illustrating the work is to be handed in. A template for this presentation is available on the IPM's web site under the menu "Masteroppgave" (<http://www.ntnu.no/ipm/masteroppgave>). This sheet should be updated one week before the Master's thesis is submitted.

Performing a risk assessment of the planned work is obligatory. Known main activities must be risk assessed before they start, and the form must be handed in within 3 weeks of receiving the problem text. The form must be signed by your supervisor. All projects are to be assessed, even theoretical and virtual. Risk assessment is a running activity, and must be carried out before starting any activity that might lead to injury to humans or damage to materials/equipment or the external environment. Copies of signed risk assessments should also be included as an appendix of the finished project report.

The thesis should include the signed problem text, and be written as a research report with summary both in English and Norwegian, conclusion, literature references, table of contents, etc. During preparation of the text, the candidate should make efforts to create a well arranged and well written report. To ease the evaluation of the thesis, it is important to cross-reference text, tables and figures. For evaluation of the work a thorough discussion of results is appreciated.

The thesis shall be submitted electronically via DAIM, NTNU's system for Digital Archiving and Submission of Master's thesis.

Contact persons:

From the industry:


Alok Mathur, Technosoft Inc., Email: [alok.mathur@technosoft.com](mailto:alok.mathur@technosoft.com)

Adel Chemaly, Technosoft Inc., Email: [adel.chemaly@technosoft.com](mailto:adel.chemaly@technosoft.com)

From NTNU:

Bjørn Haugen

  
for Torgeir Welo  
Head of Division

  
for Ole Ivar Sivertsen  
Professor/Supervisor



NTNU  
Norges teknisk-  
naturvitenskapelige universitet  
Institutt for produktutvikling  
og materialer

---

# Utilizing general-purpose computing on graphic processing units for engineering algorithm speed up

*Master thesis spring 2014*

*Stud.techn. Simen Haugerud Granlund, Stud.techn. Teodor Ande Elstad*

Department of Engineering Design and Materials  
Faculty of Engineering Science and Technology  
Norwegian University of Science and Technology

## Abstract

The goal of this thesis, is to investigate the possible benefits of using general-purpose computing on graphics processing units (GPGPU), in order to speed up the execution of calculations in engineering applications.

The thesis focuses primarily on speeding up the process of analyzing laser scan point clouds, in software developed by TechnoSoft Inc. This is achieved by developing faster algorithms for solving the k-nearest neighbors (kNN), and All-kNN problem, optimized for point cloud data.

A parallel brute-force algorithm is developed, which is capable of solving the kNN problem up to 70 times faster than a similar algorithm developed by Garcia et.al. [13], when working on comparable data.

By utilizing the k-d tree data structure, a parallel tree-build and query algorithm is developed, suitable for solving the All-kNN problem. This algorithm improves the result obtained by the brute-force algorithm by up to 300 times.

---

## Sammendrag

Målet med denne avhandlingen, er å utforske mulighetene knyttet til å anvende GPGPU (general-purpose computing on graphics processing units) for å forbedre ytelsen til tunge beregninger i programvarebaserte ingeniørverktøy.

Avhandlingen tar for seg en problemstilling, knyttet til analyse av punktskyer, i programvare utviklet av TechnoSoft Inc. Ytelsen i denne programvaren blir forsøkt økt, ved å utvikle raskere algoritmer, for løsning av kNN (k nærmeste naboer) og Alle-kNN problemet, optimalisert for punktskybaserte data.

En GPU-parallellisert brute-force algoritme blir utviklet, som er i stand til å løse kNN problemet 70 ganger raskere enn en tilsvarende algoritme, utviklet av Garcia et.al. [13], anvendt på tilsvarende data.

Ved å anvende k-d trær, en GPU-parallellisert algoritme blir utviklet, egnet for å løse Alle-kNN problemet. Denne algoritmen forbedrer resultatet oppnådd gjennom bruk av brute-force algoritmen med 300 ganger.

---

# Preface

This thesis is based on work conducted by Stud.techn. Simen Haugerud Granlund and Teodor Ande Elstad in the spring of 2014, as a part of a collaboration between the department of Engineering Design and Materials, Norwegian University of Science and Technology (NTNU), and TechnoSoft Inc. (TSI).

The authors would like to thank Professor Ole Ivar Sivertsen, NTNU, and Alok Mathur, TSI, for help, guidance, feedback and inspiration throughout the project, as well as Associate Professor Bjrjn Haugen and Adel Chemaly at Technosoft Inc. for their support.

Trondheim, June 10, 2014

---

Simen Haugerud Granlund, Teodor Ande Elstad









# Table of Contents

<b>Preface</b>	<b>5</b>
<b>Table of Contents</b>	<b>11</b>
<b>List of Tables</b>	<b>13</b>
<b>List of Figures</b>	<b>16</b>
<b>List of Algorithms</b>	<b>17</b>
<b>List of Research Questions</b>	<b>19</b>
<b>Abbreviations</b>	<b>20</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 The k-nearest neighbors search problem . . . . .	3
2.2 A short introduction to parallel programming principles . . . . .	4
2.2.1 Shared memory architecture . . . . .	5
2.2.2 Distributed memory architecture . . . . .	6
2.2.3 Parallel speedup . . . . .	6
2.2.4 The APOD design cycle . . . . .	7
2.3 A short introduction to GPU programming and CUDA . . . . .	7
2.3.1 General-purpose computing on graphics processing units . . . . .	8
2.3.2 NVIDIA GPU architecture . . . . .	8
2.3.3 CUDA programming model . . . . .	10
<b>3 The quest for a faster kNN search</b>	<b>11</b>
3.1 A short evaluation of OpenCL and CUDA . . . . .	12
3.1.1 Matrix multiplication benchmark . . . . .	12

---

3.1.2	A quick evaluation of available documentation . . . . .	14
3.2	A brute-force based approach . . . . .	14
3.2.1	Progress made by Garcia et.al. . . . .	15
3.2.2	Optimizing the brute-force algorithm for point cloud data . . . . .	15
3.3	Application of k-d trees to the kNN problem . . . . .	20
3.3.1	Why k-d trees? . . . . .	20
3.3.2	Building k-d trees for point cloud data . . . . .	21
3.3.3	Querying the k-d tree . . . . .	23
3.3.4	Testing a serial k-d tree based kNN solver . . . . .	25
3.4	Development of a parallel k-d tree build algorithm . . . . .	28
3.4.1	From recursive to iterative implementation . . . . .	29
3.4.2	Parallelization strategy . . . . .	29
3.4.3	Parallelization of Subtree-Balance . . . . .	31
3.4.4	Further improvements . . . . .	34
3.5	Development of a parallel k-d search algorithm . . . . .	37
3.5.1	Parallelization strategy . . . . .	37
3.5.2	From recursive to iterative implementation . . . . .	38
3.5.3	CUDA implementation . . . . .	38
3.6	CUDA Optimizations . . . . .	43
<b>4</b>	<b>Final results, discussion and conclusion</b>	<b>45</b>
4.1	Test environment . . . . .	45
4.2	Final results and discussion . . . . .	46
4.2.1	Solving the kNN problem . . . . .	46
4.2.2	Solving the All-kNN problem . . . . .	49
4.2.3	Parallelization performance increase . . . . .	52
4.3	Conclusion . . . . .	55
	<b>Bibliography</b>	<b>56</b>
	<b>Appendix A Data sheets</b>	<b>59</b>
	<b>Appendix B Source code documentation</b>	<b>63</b>
B.1	Api documentation . . . . .	63
B.2	Installation instructions . . . . .	66
	<b>Appendix C Source code</b>	<b>71</b>
C.1	API header . . . . .	71
C.2	Brute Force . . . . .	72
C.2.1	Bitonic sort version . . . . .	72
C.2.2	min-reduce version . . . . .	79
C.3	k-d tree build . . . . .	88
C.3.1	Radix select . . . . .	89
C.3.2	Parallel quick select . . . . .	97
C.3.3	Multiple radix select . . . . .	102
C.3.4	Parallel k-d tree build . . . . .	111

---

C.4	k-d tree search . . . . .	119
C.4.1	CUDA k-d tree search . . . . .	119
C.4.2	OpenMP k-d tree search . . . . .	133
<b>Appendix D</b>	<b>Risk assessment</b>	<b>139</b>



# List of Tables

3.1.1 Query results from Stackoverflow . . . . .	14
3.1.2 Query results from Google . . . . .	14
3.2.1 Selected results from Figure 3.2.2. . . . .	19
3.4.1 Development of a k-d tree build with a million points, showing how the different tree level operations varies throughout the build process. . . . .	35
4.1.1 Tabulated information about the three test environments. . . . .	46

---



# List of Figures

2.3.1 A visualization of the memory hierarchy in CUDA. . . . .	9
2.3.2 The relationship between threads and blocks in a grid [3]. . . . .	10
3.1.1 Matrix multiplication benchmark results . . . . .	13
3.2.1 A visualization of the parallel min-reduce operation. . . . .	18
3.2.2 Three different kNN brute-force implementations. The timing results is based on a $k$ equal to 10. . . . .	19
3.3.1 A set of points on a plane, with a possible k-d tree indicated. . . . .	21
3.3.2 Tree representation of the points in Figure 3.3.1. . . . .	22
3.3.3 Timing results for recursive k-d tree building . . . . .	25
3.3.4 Timing results for mean query time with $k$ equal to one . . . . .	26
3.3.5 Comparison of mean query time with $k$ equal to one with fast brute-force and recursive k-d tree based algorithms . . . . .	27
3.3.6 Comparison of timing of $n$ queries with $k$ equal to one with fast brute-force and recursive k-d tree based algorithms . . . . .	27
3.4.1 Development of subtasks as the kd-tree generation progresses. It shows, at each tree level, how many nodes there is to parallelize and how big each node balancing is. . . . .	30
3.4.2 An illustration of radix selection [9]. . . . .	32
3.4.3 Timing results from a intermediate version of the parallel k-d tree build algorithm. . . . .	35
3.4.4 Visualization of the final improvements on the k-d tree build implementation.	36
3.5.1 The stacks memory usage, compared to the amount of shared memory. Here $k$ was sat to 100. . . . .	41
3.5.2 Search time comparison between different stack memory types. The test are done with $k$ equals 10 and $1 \cdot 10^6$ queries per tree size. . . . .	42
3.5.3 Timing results from two different k-heap implementations, with varying $k$ and $1 \cdot 10^6$ referance points. . . . .	42

---

3.6.1 Three different memory transactions, where A and B result in good coalesced and cached transactions, while C shows a stride access pattern with bad coalescing. . . . .	44
4.2.1 Comparison of brute-force algorithm developed by Garcia et.al. and min-reduce based brute-force algorithm developed in this thesis. $k$ is fixed at 10 and $m$ is increasing. . . . .	47
4.2.2 Comparison of min-reduce brute-force and k-d tree based algorithms for solving the kNN problem for $k = 100$ and increasing $m \leq 1e7$ . . . . .	48
4.2.3 Comparison of min-reduce brute-force and k-d tree based algorithms for solving the kNN problem for $m = 1,0 \cdot 10^6$ and increasing $k \leq 200$ . . . . .	49
4.2.4 Comparison of min-reduce brute-force and k-d tree based algorithms with CPU and GPU parallelized query. The graph compares runtime for solving the All-kNN problem for $k = 100$ and increasing $m$ . . . . .	50
4.2.5 Comparison of min-reduce brute-force and GPU parallelized k-d tree based algorithms for solving the All-kNN problem for $m = 1e6$ and increasing $k \leq 200$ . . . . .	51
4.2.6 Comparison of runtime for GPU (GTX 560) and CPU (OpenMP) parallelized k-d tree based n query. . . . .	52
4.2.7 Comparison between serial and parallel k-d tree build performance. . . . .	53
4.2.8 Parallel speedup for the k-d tree implementation for varying values of $m$ . . . . .	53
4.2.9 Comparison between serial and parallel All-kNN query performance. . . . .	54
4.2.10 Parallel speedup comparison for the All-kNN query between the CUDA and OpenMP implementation. . . . .	55

# List of Algorithms

3.1	General work distribution in CUDA . . . . .	16
3.2	Iterative Bitonic sort . . . . .	17
3.3	Serial $\otimes$ -reduction . . . . .	18
3.4	Recursive k-d tree build . . . . .	22
3.5	Recursive kNN k-d tree search . . . . .	24
3.6	Iterative k-d tree build . . . . .	29
3.7	Parallel subtree balance . . . . .	33
3.8	Iterative kNN k-d tree search . . . . .	39



# List of Research Questions

**RQ 1.** *Can high performance be achieved by a parallel brute-force kNN algorithm on large point clouds?*

**RQ 2.** *Can a parallel brute-force kNN algorithm be fast enough to solve the All-kNN problem within reasonable time?*

**RQ 3.** *It is possible to use a k-d tree to increase the performance of kNN queries, compared to a parallel brute-force solution?*

**RQ 4.** *It is possible to use a k-d tree to increase the performance of All-kNN queries, compared to a parallel brute-force solution?*

**RQ 5.** *It is possible to parallelize the k-d tree build algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm?*

**RQ 6.** *It is possible to parallelize the All-kNN query algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm?*

---

# Abbreviations

ALU	=	Arithmetic logic unit
APOD	=	Assess, Parallelize, Optimize, Deploy
ATLAS	=	Automatically Tuned Linear Algebra Software
BLAS	=	Basic Linear Algebra Subprograms
DMA	=	Distributed memory architecture
CPU	=	Central processing unit
CUDA	=	Compute Unified Device Architecture
GPGPU	=	General-purpose computing on graphics processing units
GPU	=	Graphics processing unit
kNN	=	k-nearest neighbors
MPI	=	Message Passing Interface
OpenCL	=	Open Computing Language
OpenMP	=	Open Multi-Processing
RQ	=	Research question
SIMT	=	Single-Instruction, Multiple-Thread
SM	=	Streaming Multiprocessor
SMA	=	Shared memory architecture
TSI	=	TechnoSoft Inc.

# Chapter 1

## Introduction

In more recent years, the hardware developers have increased the performance of new systems, by utilizing the power of parallel processing. The demand for better computer graphics has been a driving force for creating more powerful parallel computing hardware. Powerful graphics cards, which features a highly parallel architecture, is today available in the consumer market to a relatively low price.

Unfortunately, this parallel power is not automatically utilized by traditional algorithms written with sequential execution in mind. In addition, parallelization is not a magic bullet. It can deliver blisteringly fast execution when the type of problem permits it, but for many classes of problems, benefiting from parallel processing capabilities, is not trivial, or even possible.

Due to this need for different algorithms and specialized knowledge, in order to harness the power of parallel execution, a lot of software is currently not benefiting from the possible performance of modern hardware. This is a point to be concerned with, in relation to engineering software, where complex, time consuming computations is commonplace.

In our thesis, we have studied how to utilize the power of general processing of graphical processing units (GPGPU) in engineering software applications. We have studied this problem, by developing faster, parallel algorithms, for use in point cloud analysis software, made by TechnoSoft Inc (TSI).

During the course of our work, we discovered that some of the assignment topics was not relevant, in the way initially envisioned. At the same time, new possibilities arose, that was not accounted for during the draft of the assignment text. In collaboration with our thesis advisors, we therefore choose to divert some from the original assignment.

An evaluation of GPGPU libraries like ViennaCL and cuBLAS was expected to be relevant for the thesis, but this proved to be entirely irrelevant for the particular problem we studied, since we ended up not being able to rely on any prefabricated GPGPU library. In addition,

the grid smoothing algorithms for CFD analysis was not developed in time to be a part of our work, so this part of the assignment was left out as well.



# Background

The purpose of this chapter, is to cover some background material, giving the reader a foundation for the later chapters. A introduction to the kNN problem is given, and it's relation to All-kNN, and Q-kNN problem is discussed. In addition, relevant parallel programming principles, practices and tools are presented.

## 2.1 The k-nearest neighbors search problem

The k-nearest neighbors (kNN) search problem, is, the problem of locating the  $k$  closest neighbors to a given query point, with  $k$  being some positive natural number. This is intuitively a quite simple problem to grasp, and for most of our purposes, an intuitive understanding of the problem will be sufficient, but let us start by looking a bit closer at the properties of kNN search in general.

If we consider  $p$  to be a point in d-dimensional space so that  $p = [x_1, x_2, \dots, x_d]$ . Given a set of size  $m$ , consisting of such points  $S = [p_1, p_2, \dots, p_m]$ , a additional query point  $q$ , also in d-dimensional space, and some distance metric  $D = f(p, q)$ , the kNN problem is to find  $k$  points in  $S$ , such that the sum of distances in relation to  $q$  is minimized. We introduce the term reference points to be the set  $S$ , to differentiate it from the query points  $Q$ .

It is worth to note that since we are not limited, either in the number of dimensions, or distance metric we choose, the kNN problem is applicable in many different situations. If we e.g. wanted to construct a system for finding beer with similar flavor to one beer we just tasted, we could build a database of different beers, categorized by flavor dimensions like bitterness, maltiness, sweetness, and so on. Then, to find beers similar in flavor to the one we just tasted, we would perform a kNN query on this database, using a suitable distance metric, and the beer we just tasted as our query point.

The general nature of the kNN problem makes it relevant in many research and industrial settings, from 3-d object rendering, content based image retrieval, statistics and biology [14, Introduction].

When wanting to query point cloud data, one can make some simplifications to this general kNN problem. In our research we are only concerned with three spacial dimensions, and their Euclidean relations.

This is not a universal way to simplify the kNN problem to point cloud data, and other dimension might be interesting to add for other applications. Other metrics commonly related to point cloud data, e.g. the color value of each point, could also be interesting to include. But for most applications, TSI application included, three dimensions, and an Euclidean distance metric is all we need. Throughout this text we will use the term kNN to refer to this simplification of the kNN problem.

When studying point clouds, it can be interesting to find the k-closest points to all points in the point cloud. In order to compute this, you would simply perform kNN queries, using all the points in the point cloud as query points. In order to refer to this variant of the kNN search problem, we will use the term All-kNN.

Another similar variant of the kNN problem, is application of the kNN algorithm to a set of query points of size  $q$ . In this version of the problem, you are not limited to the query point set being the points in the point cloud, it can be any set of three dimensional points. We will refer to this problem variant as Q-kNN, and note that All-kNN is a subproblem of Q-kNN.

## 2.2 A short introduction to parallel programming principles

Parallel programming is programming targeting parallel computing, where sections of the program is executed simultaneously on multiple cores, processors, machines, or other suitable environments. We use the term parallelization to mean transformation of computational instructions intended for sequential execution to simultaneous, or parallel, execution.

Parallelization can be introduced on several different levels in a computer. Bit-level parallelization, where bit level operations is parallelized within a processor as is the case for 64-bit, 32-bit, 16-bit, etc. processors, being a common low level form. In order to avoid confusion, this text is not concerned with such levels of parallelization, but will instead focus on higher level parallelization, related to developing and implementing algorithms in a regular programming language. We will also use the term parallel unit as a general term for a single computational unit in a theoretical parallel machine, with a unlimited number of parallel units.

It is easy to grasp that parallelization can speed up the execution of a program. Given a problem where we want to add one to every element in a list of numbers. In a sequential

program we could go through the list of numbers, adding one to each element in each step. This would roughly take the time needed for adding one number to another, times the number of elements in the list. In our theoretical parallel machine, we can simply assign all the elements in the list to a different parallel unit and ask every unit to add one to its assigned number. This would take much less time than the sequential algorithm, just the time needed to add a number, since all numbers in the list is added at the same time.

Adding one to the elements of a list is a trivial problem, and unfortunately not all problems can be parallelized as easy as this. Even simple problems can be hard to parallelize. Consider adding all the numbers together, instead of adding one to each number. This is a very similar problem, we still only has to add a number to all of the elements in the list. The problem is that we does not know what to add to a given element before all the previous elements has been added together. We could calculate the sub-sum of small subsections of the list in parallel, and then add the resulting sub-sums together in a sequential fashion, but then a part of our program does not execute in parallel. This exemplifies, that for many problems, we cannot entirely parallelize the execution, because some data has to be transferred between the threads. We need a way to let the parallel units communicate with each other.

Communication is often a large factor in limiting the performance that can be harnessed from parallelization. Communication can be the source of implied sequential execution, since the parallel units have to wait for data from another unit. It is also often inherently slow, and carries a high overhead due to low data transfer speeds between hardware components. The possibility of minimizing the amount of communication, is therefor a major factor in determining if a sequential algorithm can be successfully parallelized.

### 2.2.1 Shared memory architecture

In a parallel computer using shared memory architecture (SMA), the different parallel units all share a global shared memory. The parallel units usually are different processors, often located within the same physical chip like a multi-core CPU.

This is the architecture used by most modern desktop computers. Different varieties exist, with separate processor cache for each processor core, shared cache or even a combination, with shared L2 cache and distributed L1 cache. From the point of view of this thesis, all these varieties would fall under the SMA classification.

Shared memory is easy to work with and understand, and communication between different parallel units can be facilitated quite easily, and relatively fast, by reading and writing the same memory. The drawback is that the programmer must ensure that the different parallel units does not try to access the same memory at the same time, or in the wrong order. SMA still works well for smaller parallel computer systems.

## 2.2.2 Distributed memory architecture

In a computer system using distributed memory architecture (DMA), each parallel unit contains both processor and memory. The processor can only access data from its own local memory, and if data is needed from another parallel unit, it has to be transferred from that unit's local memory into the memory of the unit in need of the data. DMA computer systems usually scale better when using a high number of parallel units, compared to SMA systems, since one memory does not have to facilitate all parallel units. The drawback is that communication carries a higher overhead, since data has to be transferred between the different local memories of each unit.

Computer systems using a distributed memory architecture, often resemble many individual computers, working in parallel on the same problem, and communicating using specialized networking components.

This is the architecture favored in today's supercomputers. Many varieties exist, especially concerning the network layout between the different machines. Since each individual parallel unit in such systems usually can be considered to be an individual computer, each unit often has an internal shared memory architecture, like a desktop computer. This makes the entire system capable of harnessing the strength of both SMA and DMA, but increases the complexity that the programmer has to handle. As we will discover later in the text, GPUs are organized using an architecture with this kind of hybrid architecture.

## 2.2.3 Parallel speedup

Parallel speedup, or just speedup as it is often called, is a measure of how much faster a parallel algorithm is compared to its sequential counterpart.

Let  $T_s$  be the execution time of the sequential algorithm, and  $T_p$  be the execution time of the parallel algorithm on a system with  $p$  parallel units. The speedup  $S_p$  is then defined as  $S_p = T_s/T_p$ .

In the ideal situation, the relation between the speedup and the number of parallel units will be linear. Due to overhead related to possible communication, and the use of a more complex framework for the parallel code, this is usually not possible to achieve. We therefore have the parallel efficiency metric  $E_p = S_p/p$ , which describes how much is lost to such factors.

Speedup and efficiency can be a good measure of how well the algorithm is parallelized, but it can not necessarily be used to determine if one parallel algorithm is faster than another. Parallelizing many inefficient brute-force algorithms can be done with excellent speedup and efficiency, but the resulting algorithm will often be considerably slower than the parallel version of a better sequential algorithm, or even just a better sequential algorithm. Parallelizing bad code will result in just as bad code, and thorough study of efficient algorithms should always be carried out before attempting any parallelization.

### 2.2.4 The APOD design cycle

In order to work efficiently with parallelization problems, many programmers adopt a work-flow similar to the APOD design cycle[2, Preface]. The cycle consists of four steps:

1. **Assess:** Locate the parts of the code which take up most of the run-time. Re-writing an entire application for parallel execution is usually very hard and time consuming, if it is even possible. The best results for the user might be to just parallelize the one algorithm that is taking a long time to execute. Parallelization might not even be the answer, a faster sequential algorithm could also be a solution or part of the solution.
2. **Parallelize:** Investigate if any pre-written library of parallel functions could be used as part of, or the entire solution. Try to pinpoint which part of the code that can execute in parallel, and which part is dependent on communication. If the code is inherently dependent on communication, sequential alternatives, more suitable for parallelization, might be researched. Just not be tempted to use an inefficient algorithm because it is easy to parallelize.
3. **Optimize:** Dependent on the parallel computer system, programming language, and other tools you are using, a number of conventional optimization strategies might be applied. This should not be forgotten when writing parallel code. On a CUDA GPU you will often want to check that you are using the optimal type of memory for different tasks, minimize the divergence in the code and try to optimize the use of arithmetic operations.
4. **Deploy:** Run your application on real hardware, test thoroughly, and compare the results to the original. Did the parallelization actually increase the performance of the application? Deploy the code to potential users. They will benefit from the increased performance, and you will get feedback if any bugs exist.

## 2.3 A short introduction to GPU programming and CUDA

Moore's law has been a gift to all programmers the past 50 years. The law predicts that performance of integrated circuits would double every two years, and it has become the de facto standard for computer processing capabilities since it was first stated. However, since the so called Power Wall in 2002, the world of computer hardware performance has been changing. In order to keep up with Moore's law, the hardware vendors have been moving from single core processors, where all computation is performed by one fast processor core, to processors with multiple cores, working in parallel. As a result, many programs and algorithms has to be rewritten, in order to benefit from this new hardware architecture.

In recent years, many tools for working with parallel programming, has been developed. Frameworks and libraries like OpenMP, CUDA, MPI and OpenCL being some of the

more notable examples. These tools support various types of parallel hardware architecture.

OpenMP supports parallelization, targeting shared memory architecture. It is an implementation of multi threading, whereby a master thread divides the workload to different forked slave threads.

The Message Passing Interface (MPI), is a library specification for message passing, often used for parallel programming, targeting distributed memory architecture. It is maintained and promoted by a committee, consisting of a wide selection of academics, hardware and software vendors.

In the consumer market, GPUs represent hardware with a high number of parallel units compared to the price. The NVIDIA GeForce GTX 480 GPU is e.g. able to execute 23,040 threads in parallel [16]. The reason GPUs can have such a massive amount of parallel units at this price point, compared to traditional hardware like a CPU, is that each parallel thread is very lightweight. Individually, each thread has relatively low performance, but together they can achieve an extremely high instruction throughput. This makes targeting the GPU, a good solution for high performance parallel computing on a desktop computer.

### **2.3.1 General-purpose computing on graphics processing units**

GPGPU is the utilization of a GPU in applications to perform heavy computations, normally handled by the CPU. This is most efficiently accomplished by using GPGPU specific programming tools. Two widely used approaches are the Compute Unified Device Architecture (CUDA) and the Open Compute Language (OpenCL).

OpenCL is a low-level framework for heterogeneous computing for both CPU and GPU's. It includes a programming language, based on C99, for writing kernels. Kernels are methods that execute on the GPU. It also includes an API that are used to define and control the platform.

In contrast to the open OpenCL, the dominant proprietary framework, CUDA, is only designed for GPU programming. It is, as OpenCL, based on a programming language and a programming interface. Since CUDA is created by the vendor, it is developed in close proximity with the hardware.

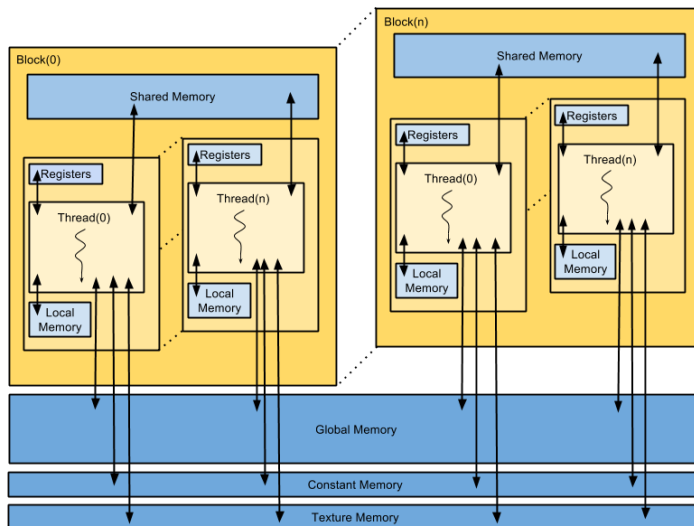
For a deep and thorough survey of GPGPU programming, techniques and applications take a look at John D. Owens article from 2007 [19].

### **2.3.2 NVIDIA GPU architecture**

A CPU is usually optimized for low memory latency. This enables the CPU to quickly fetch data and instructions from memory. For a chip to achieve low latency, it needs to have a large amount of cache available. This makes it hard to physically fit a very many cores on a single chip.

The GPU is optimized for high throughput. Throughput is a metric of how fast a processor is able to process data, and this is desirable when processing graphics, where a data has to be updated fast, in order to redraw the screen. To achieve high throughput, a large number of cores, or ALUs, are needed. GPUs is therefore usually organized with a small control unit and cache, but a large number of cores.

A NVIDIA GPU is build around a scalable array of multi-threaded Streaming Multiprocessors (SMs). A multiprocessor are designed to execute hundreds of threads concurrently. It is organized according to an architecture called Single-Instruction, Multiple-Thread (SIMT), where each SMs creates, manages, schedules and executes parallel threads in groups of 32, called a warp. Threads composing a warp starts at the same program address, but they have their own register state and instruction address, and is therefore free to branch and execute independently [3].



**Figure 2.3.1:** A visualization of the memory hierarchy in CUDA.

Another important part of the GPU architecture is the memory hierarchy, see Figure 2.3.1. Global memory is the bottom-most part of this hierarchy and is analogous to RAM on the CPU. The global memory can be used to transfer memory to and from the host CPU. Each SM contains a fast L1 on-chip memory, which is divided into a shared memory and a cache. The fast shared memory is shared across each thread in a block, and resembles the shared memory found on the CPU. The threads also have their own 32-bit registers. Other types of memory also exist. Constant memory and texture memory are read-only, and therefore highly cacheable. Compared to the CPU, the peak floating-point capability and memory bandwidth of the GPU, is an order of magnitude slower [20].

As better hardware is developed by NVIDIA, some properties change. These changes are clustered into versions, called compute capabilities. All NVIDIA devices are backward

compatible, so a device with compute capability of  $3.x$  also have all properties that is found in compute capability  $1.x$ .

### 2.3.3 CUDA programming model

The CUDA programming model is designed to make the heterogeneous GPU/CPU programming easier. The GPU works as a computation accelerator for the CPU, and the programming model should therefore be a bridge between the two. CUDA have created this bridge based on a runtime library, compiler and C language extensions. The C language extensions, enables the programmer to create and launch methods on the GPU, through the runtime library. These methods are called kernels.

A CUDA program, is based on a data-parallel behavior, where threads are executed in parallel. The execution of a kernel, is organized as a grid of blocks consisting of threads, see Figure 2.3.2. When a kernel grid is launched on the host CPU, blocks of the grid is enumerated and distributed among the SMs. The blocks then executes concurrently on each SM. Only threads in a block can communicate with each other. This is done by creating synchronization walls. Communicate can also be facilitated through the fast shared memory, located on the individual SMs. Each block and thread have their own id, which often is used to determines what portion of data the thread should process.

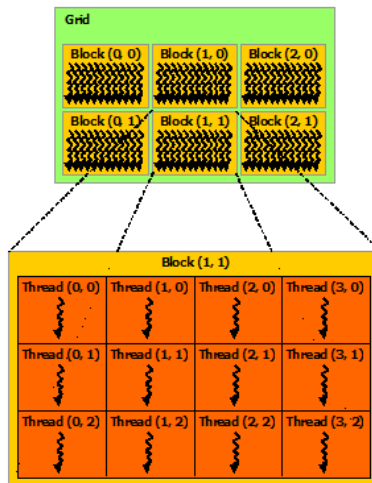


Figure 2.3.2: The relationship between threads and blocks in a grid [3].



# Chapter 3

## The quest for a faster kNN search

TechnoSoft Inc. is currently developing point cloud analysis software library. This software library is developed, with the goal of making comparisons between 3D models of an engineering design, and laser scans of the finished product. Such a comparison, would help engineers to pinpoint production errors faster, and more accurately.

In order to get good precision in the laser scan data, a large amount of points in 3D space has to be recorded. This set of data is commonly called a point cloud, and it can easily consist of  $1 \cdot 10^6$  to  $1 \cdot 10^8$  3D points, or even more, as a larger number of recorded points give better accuracy in the data.

Processing such a large number of point, is a time consuming task, and reducing the time required for individual operations become important, since they will be repeated many times. Working on point cloud data is also a problem seemingly suitable for parallelization, since operations on individual points could be executed concurrently.

TSI has analyzed their current point cloud analysis algorithms, and determined that a lot of time is spent solving the All-kNN problem, for the point clouds. They would therefore try to improve performance by developing GPU parallelized algorithms, capable of solving the kNN and All-kNN problem for a high number of points, at least in the order of  $1 \cdot 10^7$  to  $1 \cdot 10^8$ , and a low value for  $k \leq 100$ .

In this chapter, we will try to develop algorithms capable of solving the kNN and All-kNN problem, with the performance required by TSI.

## 3.1 A short evaluation of OpenCL and CUDA

As mentioned in Section 2.3.1, there are two dominant frameworks for GPGPU programming, CUDA and OpenCL. They both have their strengths and weaknesses, and in order to determine which was the most suitable for our work, we performed a small evaluation of both. This evaluation was based on a short benchmark test, where a matrix multiplication application was developed in both frameworks. The development time for both the CUDA and OpenCL matrix multiplier was limited, in order to highlight any differences in ease of use, between the two frameworks.

In addition, a quick analysis of available documentation for both frameworks was made, using common online search engines.

In all our tests CUDA outperformed OpenCL. Although our tests were very limited in scope, they support the opinion that currently, CUDA is faster and better documented than OpenCL. If the portability offered by OpenCL is not required, we would recommend using CUDA for GPGPU programming.

### 3.1.1 Matrix multiplication benchmark

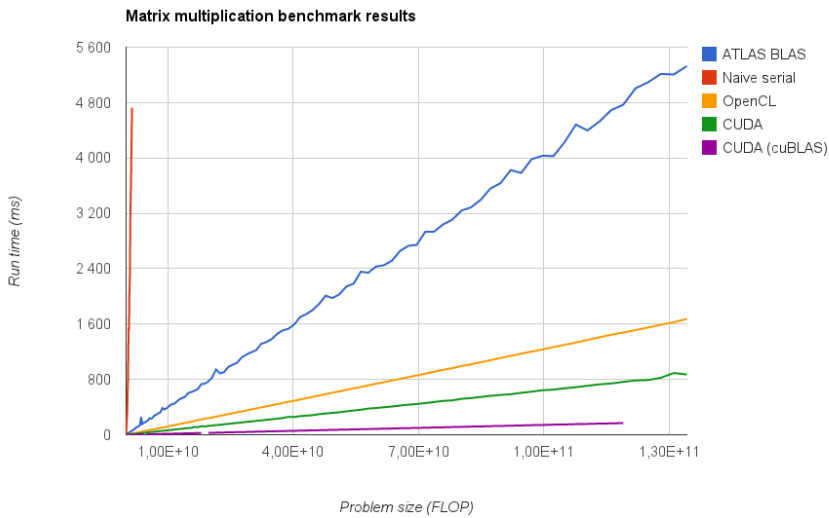
In order to compare the performance differences between CUDA and OpenCL, a simple matrix multiplication algorithm was implemented in both CUDA and OpenCL. These implementations were based on examples provided by NVIDIA and AMD. In order to establish a baseline, to which the CUDA and OpenCL results could be compared, additional implementations of the matrix multiplication algorithm were made, as both a naive serial implementation in C and a highly optimized implementation using the Automatically Tuned Linear Algebra Software (ATLAS[1]) implementation of BLAS. Finally, a highly optimized CUDA implementation was made using the cuBLAS[4] library.

The test algorithm multiplies two square matrices of size  $N \times N$ . This is an interesting problem to use for performance benchmarking for a number of reasons:

- Matrix multiplication is often used as a subroutine in more advanced mathematical algorithms.
- Matrix multiplication can be parallelized over a large number of computational cores, making it suitable for GPGPU programming.
- The mathematics of matrix multiplication is trivial, making it an easy to understand example problem.

The four implementations were tested on test environments described in Table 4.1.1. The results are presented in Figure 3.1.1

We see that the naive serial implementation quickly becomes unusable, due to a rapid increase in run time. The improvement gained by using ATLAS BLAS is very large compared to the naive implementation, although it cannot keep up with the run times achieved by the CUDA and OpenCL implementations.



**Figure 3.1.1:** Matrix multiplication benchmark results

The difference between CUDA and OpenCL is quite small, compared to the naive and BLAS implementations, but the CUDA implementation is on average about twice as fast as the OpenCL implementation. This is quite a big difference, and this could be related to all tests being run on a NVIDIA graphics card. It might also have been caused by different quality between the NVIDIA and AMD examples.

Looking at the results for the cuBLAS implementation, we can also see the impact of using a highly optimized library for GPGPU programming. The cuBLAS implementation is faster than using the basic CUDA example, indicating that proper use of libraries can be very beneficial.

It is also important to note that this is a very small test. In order to be able to conclude if CUDA is indeed faster than OpenCL, one would have needed to implement a wide selection of algorithms and test them on several different hardware configurations. Although this test is non conclusive regarding this question, the results seem to support several older investigations, concluding that CUDA is faster than OpenCL. One notable example being A Comprehensive Performance Comparison of CUDA and OpenCL[11] by Janbin Fang et al.

### 3.1.2 A quick evaluation of available documentation

When we were installing CUDA and OpenCL, and implementing our test algorithms, we relied on the online documentation available for the two GPGPU frameworks. Our subjective experience was that finding good documentation for CUDA was a lot easier than for OpenCL. In order to investigate this, we made a series of queries for pages related to CUDA and OpenCL on Google, Google scholar and Stackoverflow.com (a popular programming QA site). The results are shown in the following tables (all data from 16. Jan 2014).

Query	No of Stackoverflow.com results
Tagged OpenCL	1935
Tagged CUDA	6137
Open search OpenCL	5818
Open search CUDA	16174

**Table 3.1.1:** Query results from Stackoverflow

Query	No of Google results	No of Google Scholar results
opencv paralell programming	322000	7480
cuda paralell programming	558000	17100
opencv gpgpu	558000	5230
cuda gpgpu	816000	13500
opencv programming	875000	8160
cuda programming	2790000	22700

**Table 3.1.2:** Query results from Google

## 3.2 A brute-force based approach

As a starting point in our quest for a fast kNN search, we investigated relevant literature. Consulting our advisors lead us to two papers by Garcia et.al. [13, 14]. In these papers, a parallel brute-force algorithm is developed, capable of solving the kNN problem orders of magnitude faster than comparable fast serial algorithms. Unfortunately, the algorithm developed by Garcia et.al. has some limitations, especially regarding the number of points that the kNN problem can be solved for. This could be due to the general nature of the algorithm developed by Garcia et.al. as it is optimized for solving problems in a large number of dimensions. By optimizing the algorithm for point cloud data, we could be able to get around this limitation.

**RQ 1.** *Can high performance be achieved by a parallel brute-force kNN algorithm on large point clouds.*

Given a fast kNN algorithm for point cloud data, the All-kNN problem can be solved easily, by applying the algorithm sequentially for all points in the cloud. This, however, is a strategy sensitive to minor inefficiencies in the kNN algorithm. The algorithm will have to be very fast for a single problem, in order to solve the All-kNN problem within reasonable time. Whether this is achievable with a fast brute-force based algorithm will have to be investigated.

**RQ 2.** *Can a parallel brute-force kNN algorithm be fast enough to solve the All-kNN problem within reasonable time?*

### 3.2.1 Progress made by Garcia et.al.

The algorithm developed by Garcia et.al. is general in nature, and solves the kNN problem for any given dimension. It does this by performing the following three steps:

1. Compute all distances between the query point  $q$ , and all reference points in  $S$ .
2. Sort the distances.
3. Pick the  $k$  shortest distances.

If this general brute-force algorithm is to be used on  $n$  query points the time complexity will be  $O(n m d)$ . To an experienced programmer, this might seem like a inefficient choice for a kNN algorithm. Usually, brute-force based algorithm is frowned upon, but when taking into account parallelization, it can be a valid choice. Brute-force algorithms tend to perform a lot of isolated computations, which can be easily parallelized. Combined with potential poor performance of the serial brute-force algorithm, this gives great speedup, and in some cases actual good performance. In addition brute-force algorithms tend to be very robust on different data, and behave in a predictable manner.

### 3.2.2 Optimizing the brute-force algorithm for point cloud data

In order to improve performance of the general brute-force algorithm, we want restrict it to 3D space, and optimize it for low values of  $k$ . We also want to iron out any implementation choices made, that limit the number of points,  $m$ , the algorithm can operate on. Let us start by addressing the limit on number of points.

The implementation made by Garcia et.al. only supports problem sizes up to 65535. The limitation of 65535 corresponds to the number of theoretical blocks a CUDA kernel is allowed to spawn. This limitation is therefore only an implementation issue, not an issue with the underlying algorithm, and can be solved by applying a general partitioning algorithm. This algorithm splits the work amongst different CUDA blocks, and is shown in Algorithm 3.1.

Additional improvements can be made to the general base algorithm, if we take advantage of our restrictions to the kNN problem. With the dimension fixed to three, the Euclidean distance can be calculated in a more optimized fashion. CUDA consists of lightweight

**Algorithm 3.1** General work distribution in CUDA

---

Let  $L$  be any dividable work quantity of size  $l$ .

**function** CUDA-KERNEL( $L$ )

$b \leftarrow \text{blockIdx}.x$

▷ Current block id.

$d \leftarrow \text{gridDim}.x$

▷ Numbers of theoretical blocks in current grid.

**while**  $b < l$  **do**

    DO-WORK( $L(b)$ )

$b \leftarrow b + d$

**end while**

**end function**

---

threads, which makes reducing the amount of arithmetic calculations important for achieving good performance. The general base algorithm, who has to take any number of dimensions into account, uses two vectors and cuBlas to calculate the distance. This increases the complexity and data bandwidth required, and can be removed in our case.

The final, and maybe the most important, optimization we want to make, is changing the sorting operation, used to sort the computed distances. For problem instances in a low number of dimensions, the most time consuming operation is the sorting operation. For instance, given a search with 8 dimensions, the sort consumes 62% of the runtime [13].

### Bitonic sort

It is proven that general sorting can be performed with a time complexity of  $O(m \log(m))$  [10]. This is a costly operation, if one only need the smallest  $k$  values in a list, as is the case with the brute-force algorithm. To improve this, a sorting algorithm that sorts the list in a linear fashion, where the smallest elements are sorted first, could be used. This strategy is applied to the general brute-force algorithm.

However, using this strategy often forces us to select a sorting algorithm with bad time complexity. For instance, the insertion sort algorithm, used in the general brute-force algorithm [10], has a time complexity of  $O(m^2)$ , the time complexity of finding the  $k$  smallest points will therefore be  $O(m k)$ . Asymptotically, this would give better timing results in cases where  $k$  is smaller than  $\log(m)$ . Let us analyze a case with  $k = 100$ . For the insertion sort to get any asymptotically advantage over the best sorting algorithms, the problem size has to be larger than  $2^{100}$ , which requires a point cloud of  $1.3e30$ . This is unrealistic for our problem. Choosing another sorting algorithm could therefore be a better strategy.

Graham Nolan discusses the possibility of improving Garcia's algorithm by using bitonic sort, and he states that it gives a significant improvement [18]. Bitonic sort is a known  $O(m \log(m))$  algorithm, and is based around a sorting network. The network is a series of interleaving bitonic sequences. A sequence is bitonic if it monotonically increases and

then monotonically decreases [10]. An iterative version of the bitonic sort is described in Algorithm 3.2.

---

**Algorithm 3.2** Iterative Bitonic sort
 

---

**Input:** A list  $L$  with length  $m$ .

**Output:** A sorted list,  $L$

$P \leftarrow \{2^i \mid i \in \mathcal{N}\}$

**function** BITONIC-SORT( $L$ )

**for all**  $\{p \in P \mid p \leq m\}$  **do**

**for all**  $\{k \in P \mid p \geq k > 0\}$  **do**

**for all**  $0 \leq i < m$  **do**

$pos \leftarrow k \vee p$

▷  $\vee$  is the bitwise *xor* operator

**if**  $pos < i$  **then**

**if**  $\neg(i \& p)$  **then**

▷  $\&$  is the bitwise *and* operator

            COMPARE( $L(i)$ ,  $L(pos)$ )

**end if**

**if**  $(i \& p)$  **then**

            COMPARE( $L(pos)$ ,  $L(i)$ )

**end if**

**end if**

**end for**

**end for**

**end for**

**end function**

**function** COMPARE( $a$ ,  $b$ )

**if**  $a > b$  **then**

    SWAP( $a$ ,  $b$ )

**end if**

**end function**

---

### Min-reduce

Sorting the distances, with a  $O(m \log(m))$  time complexity, still looks like a high price to pay to get the smallest values. Especially if  $k$  is reasonably small. Do we need to sort the list in the first place? An algorithm that is more suitable, and also highly parallelizable, is the reduce operation. Cormen [10] defines  $\otimes$ -reduction of an array  $d$  of size  $m$ , where  $\otimes$  is any associative operator, to be the value  $y$ , given by the following formula:

$$y = d[1] \otimes d[2] \cdots \otimes d[m].$$

In the serial case, this is a typical linear algorithm with time complexity  $O(m)$ , as shown in Algorithm 3.3.

**Algorithm 3.3** Serial  $\otimes$ -reduction

Let  $\otimes$  be any associative operator.

**function** REDUCE( $d, \otimes$ )

$y \leftarrow d[0]$

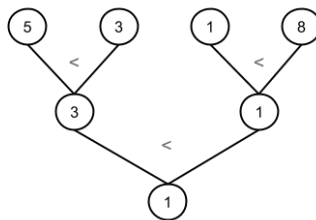
**for**  $i \leftarrow 1, m$  **do**

$y \leftarrow y \otimes d[i]$

**end for**

**end function**

Since the operator  $\otimes$  is associative, there is no difference in which way the values are calculated or if it's done on parallel. A tree based approach, like Figure 3.2.1, could be used. It is a good parallelization strategy, where every possible independent subtask is parallelized. Here each tree level do the associative operations in parallel, the results are combined as the tree level progresses downwards, until only one element remains. The parallel equivalent to Algorithm 3.3 is therefore done in  $O(\log(n))$  time.



**Figure 3.2.1:** A visualization of the parallel min-reduce operation.

To solve our problem the associative operation has to be the minimum operator. Implementing the min-reduce algorithm can easily be done in CUDA, but in order to get the best performance some optimization techniques, like loop unrolling, sequential addressing and warp unrolling, described in Section 3.6 should be applied.

## Results

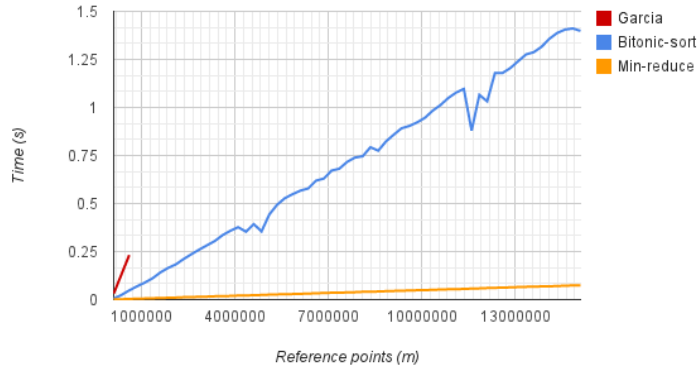
Two variations of our restricted brute-force algorithm were implemented, one using the bitonic-sort strategy, and one using the min-reduce operator. Both were compared to the algorithm developed by Garcia et.al. The complete implementations can be found in Appendix C.2

Figure 3.2.2 shows the timing results with  $k$  equals 10, for the three different brute-force implementations.

We see that the general brute-force algorithm has the worst performance. This is as expected, since it is developed for solving a more general problem, and does not use the optimizations described in our previous sections. The results for the brute-force imple-



mentation using bitonic sort, shows that Graham Nolan’s idea of improving the sorting algorithm give a huge impact. It is almost five times faster then Garcia’s implementation, shown in Table 3.2.1. As a note, bitonic sort has a sorting network that is most suited for lengths that is a power of two. This explains the two drops observed in the graph.



**Figure 3.2.2:** Three different kNN brute-force implementations. The timing results is based on a  $k$  equal to 10.

Reference points (m)	Garcia	Bitonic sort	Min-reduce
$6,0 \cdot 10^5$	231.8ms	48.1ms	3.3ms
$1,1 \cdot 10^7$	-	1077.2ms	54.2ms

**Table 3.2.1:** Selected results from Figure 3.2.2.

The big winner in this comparison is the min-reduce version. It is 70 times faster than the general brute-force algorithm, and almost 15 times faster then the bitonic version.

Although the min-reduce brute-force algorithm is the best choice in this test for low values of  $k$ , this is not always the case. Performing  $k$  min-reduce operations takes  $O(k \log(m))$  time, because one min-reduce has a time complexity of  $O(\log(m))$ . If we increase  $k$  towards  $m$ , the result would be a sorted list, and the time complexity will go towards  $O(m \log(m))$ . This is the same time complexity as our bitonic sort, but a sorting operation with min-reduce have a bigger constant time penalty than the bitonic version. However, for our use case, where  $k$  is kept reasonably small, the min-reduce method is far superior.

## 3.3 Application of k-d trees to the kNN problem

A common strategy when wanting to improve the performance of repeated queries in a large dataset, is to organize the dataset into some data structure suited for fast querying. This strategy trades the additional time required building an data structure, for increased performance on each query. In Section 3.2 we developed an optimized parallel brute-force algorithm for performing kNN queries on a large point cloud. In this section we will investigate the possibility of improving on the brute-force algorithm by using the k-d tree data structure.

**RQ 3.** *It is possible to use a k-d tree to increase the performance of kNN queries, compared to a parallel brute-force solution?*

**RQ 4.** *It is possible to use a k-d tree to increase the performance of All-kNN queries, compared to a parallel brute-force solution?*

A brief argument for why k-d trees is well suited for kNN query operations is given, then we will present the k-d tree data structure, and show how it can be used for operating on three-dimensional point cloud data. Finally a set of tests are performed on implementations of the k-d tree based algorithms, in order to determine the possible benefits of a parallel k-d tree based algorithm.

### 3.3.1 Why k-d trees?

A large part of this thesis is devoted to applying k-d trees to the kNN problem. The reader might ask themselves why this is so. Other possible data structures exist which is optimized for querying in geometrical data. Why choose to investigate k-d trees in particular?

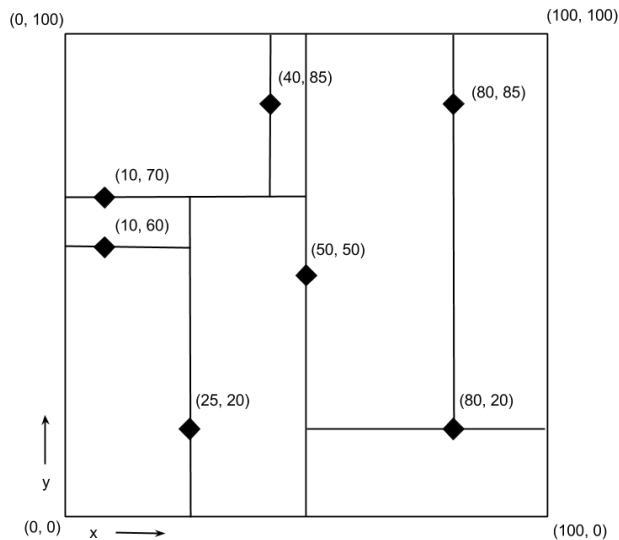
Part of the explanation has to do with the scope and time resources available for the work in this thesis. Performing a full analysis and parallelization of every possible data structure, and their associated query algorithms, would just not be feasible within our time frame. That said, k-d trees is a very attractive data structure for our use case.

- k-d trees are easy to understand and implement, leaving more time to thoroughly investigate parallelization of the algorithms.
- k-d trees are a very minimal data structure, and balanced k-d trees are complete binary trees. This makes reducing the amount of additional memory required in addition to the 3-d points a relative simple task. This is important considering the memory bounds on GPUs, and the time penalty associated with moving data from system memory to GPU memory.
- k-d trees are well adapted to performing associative queries, where the query is for a point that is not equal to, but close to the query point.
- Studies on parallel kNN queries based on k-d trees has been documented in literature with encouraging results[19, 21, 8].

### 3.3.2 Building k-d trees for point cloud data

A k-d tree can be thought of as a binary search tree in  $k$  dimensions. A binary search tree is constructed such that, for a given node, one child-subtree is consisting of elements smaller than the current node, and the other child-subtree is consisting of elements larger than the current node. The same strategy is applied when constructing a k-d tree, but at each level we are sorting the child-subtree elements according to one selected dimension, called the discriminant for this level. This discriminant is cycled through the different dimensions, as we move down each level in the tree. A formal description of k-d trees is given by Jon Louis Bentley in the paper *Multidimensional Binary Search Trees Used for Associative Searching*[7].

Let us have a look at an example using data for two dimensions. Figure 3.3.1 shows us a set of points on a two dimensional plane. The lines through each point indicate the split plane formed by the discriminant associated with the different points.



**Figure 3.3.1:** A set of points on a plane, with a possible k-d tree indicated.

The corresponding k-d tree is shown in Figure 3.3.2. Note that lower values in each level are placed in the left branches, and higher values are placed in the right branches.

By extending this example with three fixed dimensions for the spatial dimensions,  $x$ ,  $y$ , and  $z$ , we get a k-d tree suitable for storing point cloud data.

It is possible to construct several algorithms for building k-d trees from a set of points, and one simple approach is using a recursive function. Algorithm 3.4 shows pseudocode for such a simple tree building algorithm. In the pseudocode, we have chosen to represent the different dimensions as a natural number. This means that  $x$  is represented by 0,  $y$  is

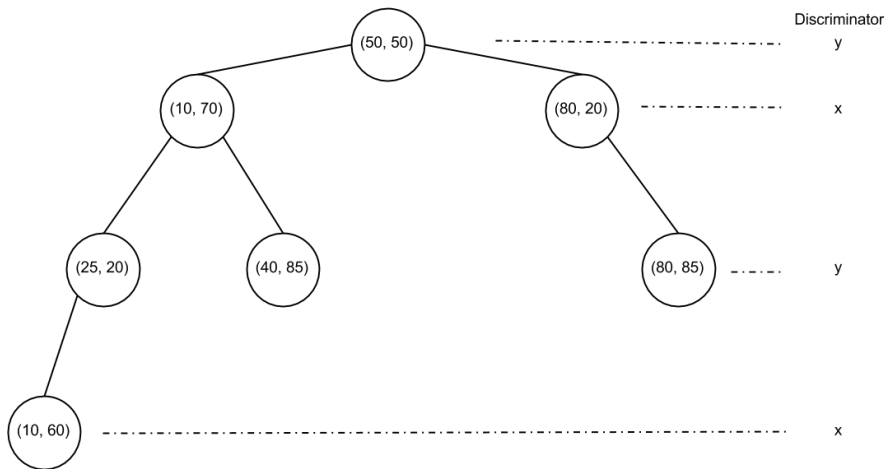


Figure 3.3.2: Tree representation of the points in Figure 3.3.1.

represented by 1, z is represented by 2 and so on. Given a set of point,  $P$ , in  $k$  space, and an initial split dimension  $i$ , it constructs a balanced k-d tree.

---

**Algorithm 3.4** Recursive k-d tree build

---

```

function BUILD-KD-TREE( $P, i$ )
  if  $P.length = 0$  then                                ▷ We have reached the end of a branch
    return NIL
  else
     $m \leftarrow \text{Median}(P)$ 
    Let  $L$  be all elements of  $P < m$  in dimension  $i$ 
    Let  $H$  be all elements of  $P > m$  in dimension  $i$ 
     $i' \leftarrow (i + 1) \bmod k$                             ▷  $k = 3$  for a three dimensional k-d tree
     $m.left \leftarrow \text{Build-KD-Tree}(L, i')$ 
     $m.right \leftarrow \text{Build-KD-Tree}(H, i')$ 
  end if
  return  $m$ 
end function
  
```

---

Algorithm 3.4 starts by checking if there is any more points left in  $P$ . If not, it returns NIL as an end of branch marker. If there still is points left, the algorithm selects the median point,  $m$ , as the root node. Then it sorts all remaining points into a collection of points lower than the median,  $L$ , and higher than the median,  $H$ . The dimension,  $i$ , is incremented, and the Build-KD-Tree function is called recursively on both collections of points. Finally the root node is returned, so it can be assigned as the child of it's parent node, or be used as a global root node.

It is worth to note that the performance of this k-d tree build algorithm is sensitive to the

choice of a median finding algorithm, since we will be querying for the median  $O(m)$  times. Choosing to just sorting the collection  $P$ , and selecting the median from the middle of the sorted collection, will not give optimal results. Fortunately, several  $O(m)$  median selecting algorithms exist[10] (Get chapter citation), quickselect, being the choice for our initial implementations. Given a fixed number of dimensions, this gives a algorithm with a time complexity of  $O(m \log(m))$ [12].

A final note about Algorithm 3.4, is that it does not handles points with duplicate values in one dimension. If the algorithm where to be feed with a point collection where all points had the same value for  $x$ , it would not be able to handle it, since such a point does not explicitly belong in  $L$  or  $H$ . Several modifications can be made to handle this case. We can choose to place all conflicting median points, except one, in either  $L$  or  $H$ . The problem with this solution, is that we are not guaranteed to get a balance tree. If we where to have a set of points, where all points where tha same, we would get a tree at all, but just one long branch of length  $n$ . Another strategy is to try to place the conflicting medians, equally in  $L$  and  $H$ . This way the median we select will be the midmost element in the point collection, retaining the balance in the finished k-d tree. Given that we consider that duplicate median points can be located in both subtrees of a node, this will not affect search operations on the tree, as we will see later.

### 3.3.3 Querying the k-d tree

With a k-d tree we can perform efficient searches for the closest point to a given point in  $O(\log(m))$  average time[12]. By maintaining a collection of the  $k$  closest points during execution of the query, we can even perform kNN searches. An example of a kNN search algorithm is shown in Algorithm 3.5.

The procedure will take the root of a k-d tree,  $r$  and a query point, for which we want to find the  $k$  closest points. In addition, it requires a initial dimension,  $i$ , which should be the same as the initial dimension used when building the tree. It uses this data to manipulate a collection of the  $k$  closest points to  $q$ . This collection is called the k-heap,  $K$ .

The k-heap is a data structure with some special properties. You can query it for the maximum distance value of the  $k$  points stored in it, and it will only store a predetermined number of points. If you try to insert more points than the predetermined number of points, it will discard the highest values, and only keep the  $k$  lowest values. This data structure can be easily implemented as a modified max-heap [10, Chapter 6]. When the size of the heap is lower than  $k$ , it is used in the usual manner, but when the heap is of size  $k$ , a slight modification to the insertion operation is made. Instead of adding the new element to the heap, the new element is swapped with the maximum value of the heap, if it is lower than the current maximum value in the k-heap. Then the heap is re-balanced using the standard max-heap balance algorithm. In our code, we assume the k-heap to be filled at the start with  $k$  points of either a random sample of points from the k-d tree, or with positive infinity. This way we do not need to check if the heap is filled during the recursive execution of the procedure.

**Algorithm 3.5** Recursive kNN k-d tree search

---

```
procedure kNN-KD-TREE( $K, r, q, i$ )  
  if  $r = \text{NIL}$  then                                     ▷ We have reached the end of a branch  
    return  
  end if  
   $d \leftarrow \text{Distance}(r, q)$   
   $dx \leftarrow r.x[i] - q.x[i]$   
  if  $d < K.max$  then                                     ▷ Is  $r$  closer to  $q$  than the current  $k$  best points?  
     $r.distance \leftarrow d$   
     $\text{Insert}(K, r)$   
  end if  
   $i' \leftarrow (i + 1) \bmod k$                                ▷  $k = 3$  for a three dimensional k-d tree  
  if  $dx > 0$  then                                       ▷ Select  $t$  and  $o$  so we traverse towards closest point first  
     $t \leftarrow r.left, o \leftarrow r.right$   
  else  
     $t \leftarrow r.right, o \leftarrow r.left$   
  end if  
   $\text{kNN-KD-Tree}(K, t, q, i')$   
  if  $dx^2 < K.max$  then                                     ▷ Can there be closer points in the other subtree?  
     $\text{kNN-KD-Tree}(K, o, q, i')$   
  end if  
end procedure
```

---

Algorithm 3.5 starts by checking if we have reached the end of a branch. If not, it calculates the Euclidean distance between the query point,  $q$ , and the current root point,  $r$ . Calculating this distance is a costly step, since it usually involves calculating a square root. This can be circumvented when implementing, by relying on using the square of the Euclidean distance as the distance metric, instead of the actual distance. This will not make a difference for the algorithm. The distance,  $dx$ , between the current root and the query point in dimension  $i$  is also calculated.

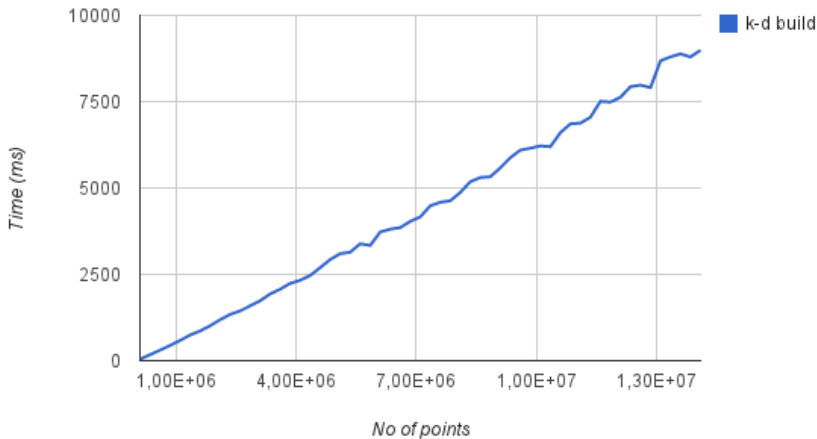
The algorithm then checks if the current root point is closer to the query point than one of the points in the k-heap. If this is the case, it inserts the current root into the k-heap. The next dimension,  $i'$ , is calculated, and then the algorithm determines if it should traverse to the right or left child node first. For efficient querying, we want to traverse down the branch that would contain the query point. In other words, if the query point is lower than the current root point in the current dimension, we want to traverse to the left child, and vice versa. The child node that we want to traverse first, is often called the target, and its corresponding subtree is often called the target subtree. In the algorithm the symbol  $t$  is used to represent target. The child and child-subtree that is not chosen for immediate traversal is called other and other-subtree. In the algorithm the symbol  $o$  is used to represent other. The ability to prune away the other subtree, given our current best estimates stored in the k-heap and the distance  $dx$ , is what makes the k-d tree efficient for kNN searches.

After recursively investigating the target subtree, we ask if our estimates in the k-heap is better than the distance  $dx$ , remembering that the distances stored in the k-heap is squared. If this is the case, we know that there cannot be a closer point in the other subtree, and we can prune it from our search. If not, we have to check the other subtree as well. When the procedure terminated, the k closest points to the query point is stored in the k-heap.

### 3.3.4 Testing a serial k-d tree based kNN solver

In order to gain some real world insight into the performance characteristics of k-d tree building and querying, a serial implementation of the build and query algorithm was made. These implementations is available in Appendix C.3 and Appendix C.4.1. These two implementations where then subjected to several tests, using test setup Y. All tests were performed on a set of randomly generated points 3-d points, with the number of points ranging from  $10^5$  to  $1.41 * 10^7$ . The result of these test are summed up in the following figures.

Figure 3.3.3 shows the timing results for the recursive k-d tree build algorithm.



**Figure 3.3.3:** Timing results for recursive k-d tree building

We observe that constructing a k-d tree for a large number of points is a costly operation. Given a tree of size  $1.41^7$  the algorithm uses nearly 9 seconds to construct the tree. We also note that the timing results seem to scale linearly in relation to the number of points. This relates nicely to calculated time complexity of the algorithm.

Figure 3.3.4 shows the timing results for querying a k-d tree of a given size. The k-d tree is queried for one point with  $k = 1$ . Since we are interested in investigating the average

performance,  $10^5$  consecutive queries was timed, and the average value for one query was calculated.



**Figure 3.3.4:** Timing results for mean query time with  $k$  equal to one

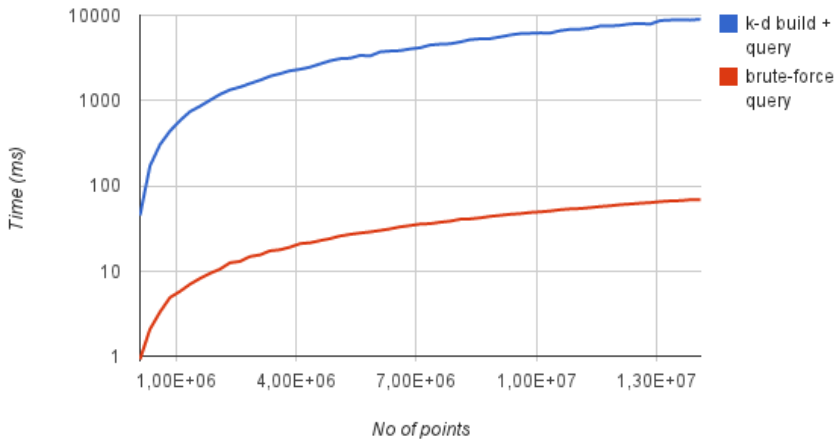
We see that querying the k-d tree is very fast on average. Querying for one point in a tree of size  $1.41^7$  takes about 0.0014 milliseconds. It has to be taken into account, that a query with  $k = 1$  will give the best query time, since the time complexity of the query algorithm scales with  $k$ . Still, for queries with a low  $k$ , we should expect good performance. The graph also seems to scale with the logarithm of the number of points, as expected by the time complexity calculation.

In order to try to answer RQ 3, we compare the timing results gained from the fastest brute-force algorithm developed in Section 3.2. Figure 3.3.5 compare the average time required for building a k-d tree of a given size, and performing a single  $k = 1$  query, to the time required to compute the same result with the fastest brute-force algorithm obtained in Section 3.2.

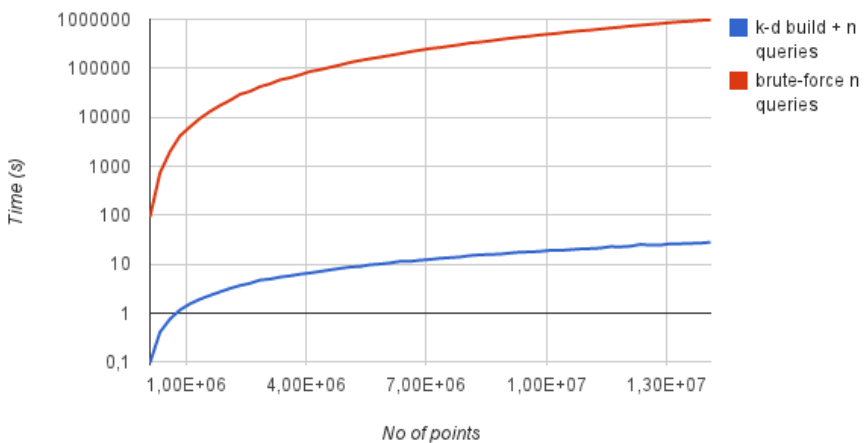
In this comparison, the k-d tree based algorithm does not seem like a good option. When performing just one query, the additional time required to build the k-d tree heavily outweighs the benefit of the improved query time, compared to the brute-force solution. This result is to be expected, since we are not really utilizing the benefit of the k-d tree, but it is still an important point that a brute-force algorithm can be very efficient for certain use-cases.

Let us finally look at some results more closely related to the use-case given by TSI. Figure 3.3.6 does the same comparison as Figure 3.3.5, but instead of comparing the time taken to perform one query,  $n$  repeated queries are performed, with  $n$  being the size of the k-d tree.





**Figure 3.3.5:** Comparison of mean query time with  $k$  equal to one with fast brute-force and recursive k-d tree based algorithms



**Figure 3.3.6:** Comparison of timing of  $n$  queries with  $k$  equal to one with fast brute-force and recursive k-d tree based algorithms

We observe that in this use-case, the k-d tree based approach have much better results than the brute-force based approach. Now the k-d tree only have to be built once, but we benefit

from the decreased query time in all  $n$  queries. Performing  $n$  queries on a point cloud of size  $1.41^7$  with the brute-force based algorithm takes about  $9,7 \cdot 10^5$  seconds, or about 11 days. With the recursive k-d based algorithm, the same operation can be calculated in just over a minute. Considering the needs of TSI, it seem that this approach is worth developing further into an parallel algorithm.

Despite these initial positive results, some problems are apparent from our initial tests.

The k-d tree building algorithm is very slow. Given that we want to perform kNN queries on larger point clouds than  $1.41^7$ , finding an efficient parallelization of this algorithm would be very beneficial. This is not as trivial as it might seem, as tree-based algorithms do not lend themselves very well to trivial parallelization.

When scaling the number of repeated queries from one to  $n$  we observed the huge impact a seemingly small change in the time required for performing one query had on the time needed to compute the entire result. A change from several milliseconds to a fraction of a milliseconds might seem trivial, but given enough repeated queries, this was the difference between minutes and days of computation time. Will we be able to keep the query time down when increasing the value of  $k$ , and moving the computation over to the GPU, which generally has a slower clock cycle than the CPU.

In the next sections we will address these challenges, along with others, and develop a parallel algorithm for performing kNN queries based on k-d trees.

### 3.4 Development of a parallel k-d tree build algorithm

The build process is by far the most expensive operation on a k-d tree, and parallelizing it could reduce the overall runtime significantly, when solving the kNN problem. This is not as straight-forward as it might seem. The serial k-d tree build algorithm is usually implemented as a recursive function, since recursive functions tend to go along well with tree-based data structures. On the other hand, performance in CUDA is based on efficient use of a massive number of lightweight threads, and to get a fast algorithm one have to split the work between as many threads as possible. This is only possible if it is easy to divide the work into independent subtasks, where data communication is kept at a minimum. In a recursive context, execution flow is hidden inside each threads call stack. Information needed by other threads is therefore not easy to obtain. To solve this, a iterative approach should be used, which can be implemented with a global accessible execution flow. This will however, give a more complex k-d tree build algorithm, and it is not certain that this algorithm is easy to parallelize.

**RQ 5.** *It is possible to parallelize the k-d tree build algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm.*

In order to investigate RQ 5, we have to look a bit closer at the different steps of the k-d tree build algorithm and investigate different parallelization strategies.

### 3.4.1 From recursive to iterative implementation

Before we dive into the parallelization strategy and how the parallelization can be done, lets try to make a iterative solution. We can start by enumerating the different steps in the recursive implementation.

1. Find the median of the points along a specified axis. This median point becomes the value of the current node.
2. Sort all points with lower values than the median to the left of the median, and all the points with higher values than the median to the right.
3. Perform this algorithm recursively on the left and right set of nodes.

From the steps one can see that for each node in the k-d tree, one have to partition a list around it's median. We will call this operation Balance-Subtree. If we analyze the Algorithm 3.4, we see that there are two recursive calls. This is logical, because we are building a binary tree where each node have two children. The interesting observation is that a node balance is only dependent on the parent node. This means that each tree level are independent and can be done iteratively.

The k-d tree construction basically boils down to successively balance each node in the tree. This leads to a basic reimplementaion, see Algorithm 3.6. It goes through each level of the tree, starting at the top, and balances each node successively down the tree.

---

**Algorithm 3.6** Iterative k-d tree build

---

**Input:** An array of points,  $T$

**Output:**  $T$ , as a k-d formatted array

**function** BUILD-KD-TREE( $T$ )

**for all**  $L \in \{\text{all levels in } T\}$  **do**

**for all**  $S \in L$  **do**

$d \leftarrow |L| \bmod k$

      ▷  $k = 3$  for a three dimensional k-d tree

      BALANCE-SUBTREE( $S, d$ )

**end for**

**end for**

**end function**

---

### 3.4.2 Parallelization strategy

Now that an iterative solution have been created, lets start looking at how this algorithm might be parallelized. First a good overall parallelization strategy has to be found. A good strategy manage to easily split the main task into small individual subtasks, that can be performed in parallel, while maintaining a minimal need for communication between the subtasks.

When we converted our k-d tree algorithm from a recursive to a iterative solution some interesting observations was made. One observation is that a node balance is only depen-

dent on the parent node. This means that each tree level are independent, which acts as a good start for our parallelization strategy.

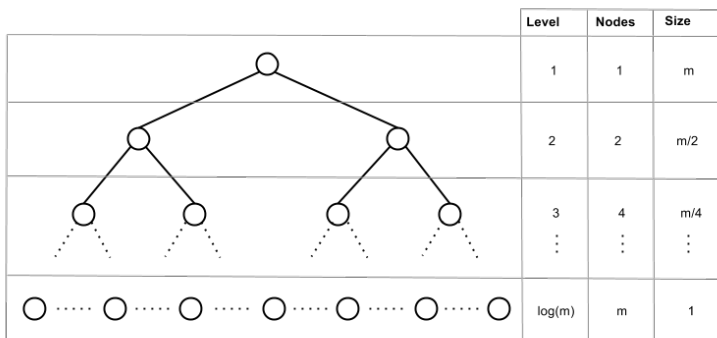
This also implies that all subtrees in the k-d tree generation are independent. Hence, the tree corresponding to the left and right child of a node can be done in parallel without any communication.

The data is also independent, as a result of how we represent the tree as an array. By data independent, it is meant that the data structure easily can be partitioned to each subtask. In our case this will be to successively partition the tree into contentious subtrees.

These observation implies that we can divide the tree levels into dependent tasks, where each node balance in a tree level is a independent subtask. This gives us an power of two increasing number of parallel tasks as we go down a tree level. The subtree size will decrease with a factor of two in each downward step, see Fig 3.4.1

This parallelization strategy gives many concurrent operations at the lower level of the tree, but at the initial levels, will hardly get any parallelization at all. To compensate for this one could seek to parallelize the work done in each Balance-Subtree procedure, which also act as a parallelizations strategy.

Both strategies can be used in conjunction with each other. The parallel balance node task algorithm can be used to speed up the early iterations, where the amount of nodes in a tree level is small. As well as further parallelize the subtasks in later tree level iterations. This strategy also fit well to our choice of tree representation. One parallel operation can now take the tree representation, split it into subtrees, and balance each one.



**Figure 3.4.1:** Development of subtasks as the kd-tree generation progresses. It shows, at each tree level, how many nodes there is to parallelize and how big each node balancing is.

### CUDA parallelization

CUDA have a special architecture that should be taken into account when parallelizing an algorithm. To efficiently use CUDA, the program has to keep thousands of threads

occupied, otherwise the benefit of CUDA disappears. The CUDA programming model is build up by a grid of independent blocks, i.e. execution can not be synchronized across blocks. Execution can only be synchronized between the 1 to 1024 theoretical threads launched inside a block [3]. Thread synchronization is important when multiple threads cooperate on one task, because at some point information has to be exchanged.

Our parallelization strategy states that we have to balance one tree level after another, since they are dependent. This implies that the threads need to communicate between each tree level. One CUDA kernel should therefore balance a complete tree level. The other alternative would be to build the whole tree in one block, which would restrict our kernel to only be executed on one SM.

The next step is to split the tree level balance between the CUDA resources. The number of nodes in a tree level increases with the power of two, as we go down the tree. Fig 3.4.1 shows that our kernel, the tree balance, changes throughout the build process. First only one node needs to be balanced, e.g. only one parallel operation. At the end there are  $m$  different nodes to work on. The problem size also changes, at the top, it is  $m$  per node and goes towards 1, as the tree level increases.

This varying problem sizes and subtasks, makes it hard to create a good work distribution between the CUDA resources. At the top part of the tree it is optimal to use many blocks to balance a node, but at the end, it is desirable to balance many nodes inside a block. We choose a middle ground, to balance one node in one block. This means that there should be an overall good performance with a peak at the middle three levels.

#### 3.4.3 Parallelization of Subtree-Balance

With the overall parallelization strategy planned, we can start investigate the most time-consuming operation, balancing a tree level. We have already determined that the parallelization should be done in one block, which means that one operation is done per SM. In other words, the task can potentially be parallelized between 1024 theoretical threads. Let's start investigating different approaches.

The main operation is to find a median. As we have seen in Section 3.3, many algorithms for finding median exist. Since we now want to implement the algorithm with CUDA, the environment has changed, and quick-select may not be the best alternative anymore. The first problem with quick-select is that it is recursive, which makes it hard to parallelize on CUDA. Therefore it may be profitable to look at other, more parallelization friendly, algorithms.

First reusing the bitonic sort was investigated. Given a sorted list one can find the median directly, by simply looking at the midmost element of the array. The partitioning is also done in the process. Unfortunately this strategy proved unsuccessful, as re-purposing the bitonic algorithm for such a task proved difficult. The reason for this is that a pure bitonic sorting network only manages to sort lists with a length of power of two. The normal solution is to create a longer list than needed, but wasting this much memory on the GPU is not a very good solution.

Another way to make bitonic sort work with a list of any size is described by K.E. Batcher [6]. The description of this solution is very lengthy, and has been omitted, since it introduces a lot of divergence, that would be detrimental to performance on the GPU. We also have the inherent downside of sorting a list in order to find the median, since  $O(m)$  algorithms for finding the median exist, compared to the  $O(m \log(m))$  time required by sorting.

Bucket-sort and radix-sort based algorithms are investigated in the paper Fast K-selection Algorithms for Graphics Processing Units by Alabi et.al. [5]. The big difference between them is the constant time penalty. The radix sort have a more exact time complexity of  $O(bm)$ , where  $b$  is the number of bits in each number. While the penalty for bucket select is  $O(am)$ , where  $a$  denotes the degree of agglomeration in the values. In other words, the algorithm is weak when the points are clustered together. His results shows that bucket select normally is slightly faster, except when  $a$  is high. Although bucket select normally have better results, we expect a high degree of agglomeration in our application, so we choose radix select.

### Radix select

The radix select is based on a bitwise partitioning, much like radix sort [10, Chapter 8.3]. In each step, elements are partitioned in two subgroups based on the current bit. Then the subgroup that contains the median is determined, and the search continue in that subgroup until the median is found.

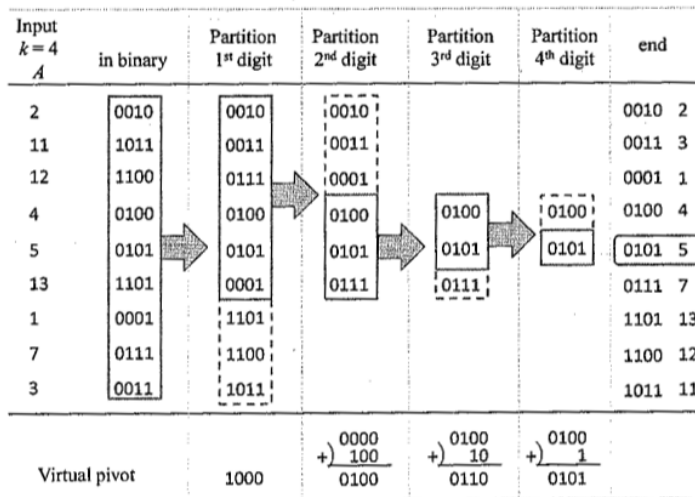


Figure 3.4.2: An illustration of radix selection [9].

When it comes to create a parallelization strategy for radix select it is first advisable to take a look at a highly optimized radix sort, like the variant introduced by Merrill [17]. The radix select can easily be reduced from a radix sort, and many concepts can therefore

be reused. An other interesting implementation is the radix select by Alabi [5]. They both uses a parallelization strategy by splitting each radix partition into parallel operations. The way our and Alabi's solution differs from Merrill's solution, is that we start on the most significant digit, since a least significant digit approach will not reduce the k'th order statistic problem in each step.

### Our implementation

Our implementation is based on Algorithm 3.6. Calling Balance-Subtree on all nodes in a tree-level, is performed on the GPU as a CUDA kernel. The outer for-loop is executed on the CPU, and launches the kernel for increasing tree-levels, until the entire tree is built. The complete implementation can be found in Appendix C.3.4.

Algorithm 3.7 is parallelized only within a single CUDA block. This means that the parallel threads are able to communicate. The algorithm is based around a repeat-until loop, which basically do all the work. The loop keeps track of a partition array. This array is then sliced in to, by giving each thread a portion of the array, each thread then counts how many zeros it finds in the current bit position. The cut, as the arrows in Figure 3.4.2 shows, is calculated by doing a reduce sum operation [15]. This is repeated until the partition contains the median, which is when every bit is used or when the partition size is one. After the loop, the array is transformed. Such that the median is in the center, with lesser elements on the left and bigger on the right.

---

#### Algorithm 3.7 Parallel subtree balance

---

**Input:** A subtree  $S$  of length  $m$ , and dimension  $d$

**Output:** A balanced subtree,  $S$

**function** BALANCE-SUBTREE( $T, d$ )

Let  $l$  and  $u$  be the upper and lower bond of current partition.

Let  $P$  be all nodes in  $S$

**repeat**

**for all**  $\{p \in P \mid l < p < u\}$  **do**

$Z(t) \leftarrow$  Occurrences of zeros in current bit,  $b$ , found by thread,  $t$ .

**end for**

$c \leftarrow$  SUM-REDUCE( $Z$ )

$\triangleright c$  is the cut of the current partition  $P$ .

**if**  $u - c \geq m/2$  **then**

$u \leftarrow u - c$

**else**

$l \leftarrow u - c$

**end if**

$b \leftarrow b + 1$

$\triangleright$  Move to the next bit

  SYNCHRONIZE-THREADS()

**until**  $u - l < 1 \vee b > \text{RADIX}(p \in P)$

  PARTITION( $S, P$ )

**end function**

---

In CUDA thread instruction run sequentially in warps of 32. Control flow divergence within a warp can therefore significantly destroy the instruction throughput. This is because the different execution paths must be serialized, as all the thread in a warp share the same program counter [2]. The total number of instruction in a warp will therefore increase. Any conditional operator, e.g. `if` and `switch`, should be used with care, since it may branch the control flow. Optimizing the use of conditionals, in order to reduce the amount of branching in the program control flow, will give better performance.

In our implementation, all threads perform the same thing in every iteration. This will give a low thread branching. The loop is also almost done equal times by all thread, one time for each bit used to represent the points. The one `if` statement in the iteration, can be reduced to only contain one statement, which make the divergent thread branch small. The code is therefore good in regard of divergence.

An other aspect to consider, is the CUDA memory hierarchy. It is beneficial to use the fastest suitable memory. In our case, this include the shared memory, which is a fast memory shared between all threads in a block. The downside is the memory size, it may not be enough space to store our subtree. Although we may not be able to store the hole subtree, there are some data we can put in the shared memory. The zero counter array, shown as  $Z(t)$  in Algorithm 3.7, is a perfect candidate. Every thread only use one integer and it is shared between all threads throughout the execution. It will therefore cause a big impoverishment.

The complete implementation can be found in Appendix C.3.3.

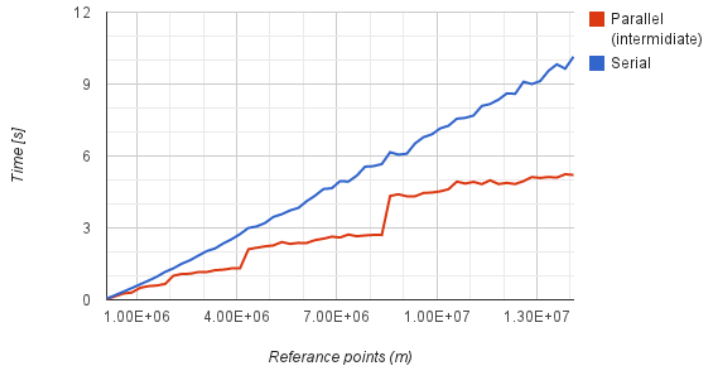
### 3.4.4 Further improvements

Let us take a step back and look at RQ 5. The possibility of parallelizing the build algorithm is achieved. Initial optimization has been performed, so according to the APOD design cycle, we should test our implementation.

With the test setup as described in Section 4.1, the current version of the algorithm gave results as shown in Figure 3.4.3. The most interesting observation are the big jumps in the graph. If one look closely these jumps happens every time the problem size exceeds a power of two, as for example when the size passes 8388608 the timing increases from 2703ms to 4335ms. The results of further investigation is shown in Table 3.4.1. It shows how long time each tree level takes, and how the different tree level operations varies throughout the build process.

The table reveal some weaknesses of our algorithm, that is based around how the CUDA resources was divided in Section 3.4.2. It performs badly when the problem size or the number of subtrees is relatively large. The potential for parallelizing the workload for the first and last iterations is not being fully utilized. This is due to the implementation forcing one version of the radix select algorithm to work on all problem types. This is not optimal for dividing CUDA resources, and as a result, we get high penalties when the problem reaches unsuitable values.





**Figure 3.4.3:** Timing results from a intermediate version of the parallel k-d tree build algorithm.

Tree level	Time [ms]	Subtrees	Size
1	52	1	1000000
2	26	2	500000
3	13	4	250000
4	8	8	125000
5	7	16	62500
6	6	32	31250
7	6	64	15625
8	6	128	7812
9	7	256	3906
10	7	512	1953
11	8	1024	976
12	10	2048	488
13	16	4096	244
14	26	8192	122
15	52	16384	61
16	105	32768	30
17	202	65536	15
18	389	131072	7
19	768	262144	3

**Table 3.4.1:** Development of a k-d tree build with a million points, showing how the different tree level operations varies throughout the build process.

This hypothesis can also explain the big jumps in runtime. The observations above correlates perfectly with the tree hight, since the hight of a binary tree is the binary logarithm of the tree size. This implies that the jumps happens when an additional tree-level is required

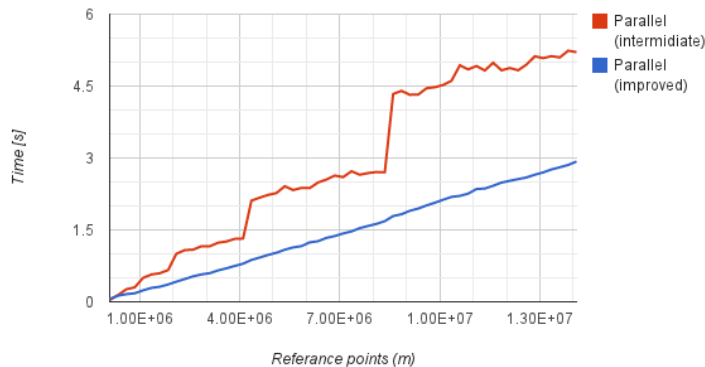
in order to fit the tree.

Tuning the algorithm to alternate between different algorithms to balance a subtree, eliminates this problem. This removes the penalty for calculating the median at unsuitable problem sizes.

Our current implementation only use one block per subtree, which means we are only able to use one SM to balance the subtree. By utilizing several blocks at the same time, big subtrees could be processed faster. However, since all blocks are used to balance one subtree, only one subtree can be balanced at a time.

The implementation details are omitted here, but can be found in Appendix C.3.1. In short, the implementation follows the outline of the radix-select implementation, but the thread synchronization is performed on the CPU, between kernel launches. This enables us to communicate between the different blocks.

We also would like to improve performance, when Balance-Subtree is applied to many small subtrees. For instance, at level 18 there are 131072 different subtask with only an average size of 7. The previous implementation divided all these subtask between a small number of SM, typically 8 – 32 on current NVIDIA GPUs. The algorithm the uses to many cores to balance a subtree of only 7 elements, which is not a efficient way to divide resources.



**Figure 3.4.4:** Visualization of the final improvements on the k-d tree build implementation.

Letting just one thread handle the Balance-Subtree operation for small subtrees, would let us process more subtrees in parallel, improving performance. With this parallelization, each thread is responsible for it's own subtree, and communication with other threads is no longer needed. With the need for communication eliminated, we can utilize Algorithm 3.4.

Now that a lot of improvements have been made, lets take a look at the results. Figure 3.4.4 compare our intermediate result with our new and improved version, and the changes made

a huge impact on the performance. The jumps disappeared, and was replaced by a faster and smoother curve, indicating that the CUDA resources is perfectly balanced. The final implementation can be found in Appendix C.3.2.

## 3.5 Development of a parallel k-d search algorithm

In this section, we will investigate parallelization of the k-d tree based query algorithm, applied to the All-kNN problem. The following research question is stated:

**RQ 6.** *It is possible to parallelize the All-kNN query algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm.*

To investigate RQ 6, we will devise a parallelization strategy, rewrite the k-d search algorithm as a iterative algorithm and optimize our implementation for CUDA execution. Finally results obtained from testing this implementation is presented.

### 3.5.1 Parallelization strategy

Solving the All-kNN problem, can be done by repeated application of the kNN query algorithm. This is an algorithm that is easily parallelized, by distributing individual kNN queries across the available parallel units. However, there are still some possible pitfalls to address. Should a query be done in one block, maybe each query should be done single handedly by one thread, or maybe we should use one thread per  $k$  in a query.

We have previously determined that a single query on a k-d tree size  $m$ , will in average visit  $\log(m)$  nodes. This indicates that not a lot of GPU resources is needed to perform a individual query. Assigning an entire CUDA block to one query therefore seems excessive. Combined with the communication heavy nature of the query algorithm, the best parallelization strategy is therefore to use one thread per query and equally distribute the queries amongst the GPU's SM.

Let us now consider if we can use Algorithm 3.5 directly with this parallelization strategy. As discussed in previous sections, GPUs and recursion don't get along well, the main drawback being the inherent need for communication between the recursive calls. Unfortunately, even with our current parallelization strategy, there are still reasons not to use a recursive algorithm.

The GPU threads are lightweight, with restricted available memory and cache. This means that the call stack, where the all program instructions are managed, is relatively small. Given a non tail-recursive algorithm, the program context and instructions are appended to the call stack at each recursive call. This will eventually fill up the limited call stack available for an individual CUDA thread. It might be possible to still use a recursive algorithm, given that the call stack never gets to large.

To determine if the recursive k-d tree query algorithm can fit within a CUDA thread, call stack tests was performed. On a individual block, 64 theoretical threads was spawned, each querying a k-d tree of increasing size. The results showed that when the k-d tree size passed  $1 \cdot 10^5$  points, unknown errors started to appear. This indicated a stack overflow.

Divergence also needs to be considered. In a recursive algorithm the decision of whether a recursive call should be made or not, is entirely up to a single thread. Once two threads have made different decisions, there is no guaranty that they will stay synchronized.

Both problems would be solved by rewriting Algorithm 3.5 into an iterative algorithm.

### 3.5.2 From recursive to iterative implementation

To rewrite Algorithm 3.5 into a iterative algorithm by explicitly managing the recursion stack, some properties about how the search traverse the k-d tree is needed. From Algorithm 3.5 one can see that this is a variant of the depth-first traversal, since the work of the current node is done before and between the recursive calls. This traversal in also the best strategy in a binary tree search, because the pruning of subtrees is maximized. How to make a standard binary search tree in an iterative fashion is described in Cormen [10, Chapter 12], but since this is a k-d tree search the implementation is slightly different, as shown in Algorithm 3.8,

The algorithm works in the same way as the recursive algorithm, but adds a stack,  $S$ , called the s-stack, and a while loop in order to handle the tree traversal iteratively. While there is a element assigned to the root variable,  $r$ , the algorithm will traverse down the target branch, updating the dimension,  $i$ , calculating the distance,  $dx$ , determining the target,  $t$ , and other,  $o$ , child node. Then it will collect  $r$ ,  $o$ ,  $i$  and  $dx$  into one element, and push it on the s-stack. Finally the root variable is assigned to the target child, or NIL if we have reached the end of a branch.

While there still is elements in the s-stack, but  $r$  is assigned to NIL, we are traversing back up a branch. While this is happening, the algorithm pops elements from the s-stack, determines if they should be added to the k-heap, before it determines if it need to investigate the other branch of this node. If that is the case, the other node is assigned to  $r$ , and the algorithm will traverse down this subtree using the previously stated rules.

### 3.5.3 CUDA implementation

Our simple parallelization strategy, combined with an iterative implementation of the k-d tree search algorithm, resulted in a trivial CUDA implementation, as we did not need to parallelize the iterative search algorithm itself. The implementation can be found in it's entirety in Appendix C.4.1. In addition, we will highlight some implementation details, and look at the results obtained from this code.

**Algorithm 3.8** Iterative kNN k-d tree search

---

```

procedure ITERATIVE-KNN-KD-TREE( $K, r, q$ )
  Let  $S$  be a stack for collecting tree nodes
   $i \leftarrow 2$ 
  while ! $S.empty$  or  $r \neq \text{NIL}$  do
    if  $r = \text{NIL}$  then
       $r \leftarrow \text{POP}(S)$ 
       $i \leftarrow r.dimension$ 
      if  $r.dx^2 < K.max$  then  $\triangleright$  Can there be closer points in the other subtree?
         $r \leftarrow r.other$ 
      else
         $r \leftarrow \text{NIL}$ 
      end if
    else
       $d \leftarrow \text{DISTANCE}(r, q)$ 
      if  $d < K.max$  then  $\triangleright$  Is  $r$  closer to  $q$  than the current  $k$  best points?
         $r.distance \leftarrow d$ 
         $\text{INSERT}(K, r)$ 
      end if
       $i \leftarrow (i + 1) \bmod k$   $\triangleright k = 3$  for a three dimensional k-d tree
       $r.dimension \leftarrow i$ 
       $r.dx \leftarrow r.x(i) - q.x(i)$ 
      if  $r.dx > 0$  then  $\triangleright$  Select  $t$  and  $o$  so we traverse towards closest point first
         $t \leftarrow r.left, r.other \leftarrow r.right$ 
      else
         $t \leftarrow r.right, r.other \leftarrow r.left$ 
      end if
       $\text{PUSH}(S, r)$ 
       $r \leftarrow t$ 
    end if
  end while
end procedure

```

---

Algorithm 3.8 does not have a lot of divergence, and the remaining branching can be further reduced. If threads in a warp is traversing completely different parts of the tree, they will access different nodes. This is called data divergence. The solution is to let each warp search for points that are located closely in the k-d tree. This will cause all the threads to traverse down the tree in roughly along roughly the same branch, reducing the data divergence. Due to the nature of our k-d tree implementation, this can be achieved by feeding the points to the search algorithm as they are placed in the k-d tree.

The explicit stack also makes an interesting question about where to store the new stack. This is data that are modifiable and thread independent, which means that the possible options memory options are shared memory, local memory and global memory. Local memory is the memory each thread can allocate dynamically from the heap. Global memory is a possible candidate. It has enough space, it is modifiable and accessible to all threads. The drawback is the access time, it takes around 400 – 600 clock cycles[2], and it would therefore be beneficial to use some other kind of memory. Shared memory would be a perfect candidate, because the memory is fast and the need to communicate between blocks is nonexistent. The only drawback is the amount of data available in shared memory, which is around 49kb on current NVIDIA GPU's.

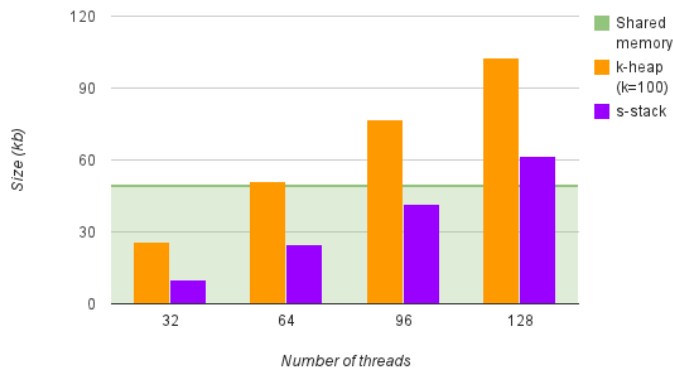
The iterative search algorithm uses one stack and one heap, both stored as arrays in memory. Th number of arrays are dependent on the number of threads used in each block. The s-stack array size is dependent on how many elements the depth-first tree traversal needs to store. If one looks on how the algorithm handles the stack, one can see that elements are pushed on the way down, and popped on the way up. This means that the stack never will be longer then the tree hight. One stack element uses 16 bytes of space, which means that the stack memory is s subset of  $\Theta(16 \log_2(n)T)$ . Here  $T$  represent the number of threads and  $n$  is the k-d tree size. The k-heap array size, depends on the number of closest neighbors,  $k$ , and one element uses 8 bytes. This implies that it's memory usage will be,  $\Theta(8kT)$ .

In Figure 3.5.1 the memory usage of each stack is compared to the available shared memory. Some basic assumptions and approximations have been done in regard to the data. Treads are only compared in multiples of 32, since this is the warp size and is therefore the most optimal thread numbers. The value of  $k$  is dependent on the problem in hand, and as our application only needs a value of 100, so that value is used.

Figure 3.5.1 shows that the k-heap will not fit in shared memory. Already at a thread count of 64 the memory is filled up. The size of the k-heap is also highly dependent on the size of  $k$  which is hard to predict. However, locating the s-stack on shared memory looks promising. The memory size has also a relatively low asymptotic growth,  $O(\log(n))$ , in regard to the tree size.

To decide what kind of memory is optimal location for the s-stack, Figure 3.5.2 has been created. Surprisingly shared memory looks like the slowest alternative. One likely reason is that elements in shared memory is synced between all threads in a block. This property is not needed in the s-stack, since the s-stack is only used by one thread.

Although global and local memory presumably is stored at the same place, the are some

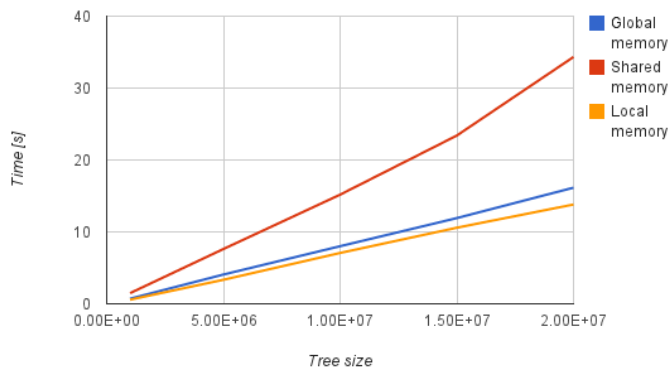


**Figure 3.5.1:** The stacks memory usage, compared to the amount of shared memory. Here  $k$  was sat to 100.

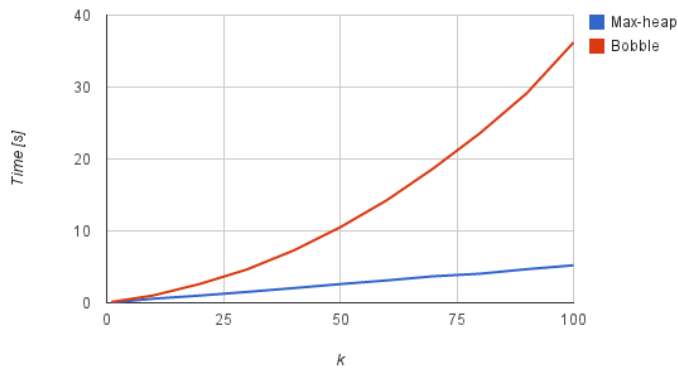
noticeable differences that can explain the time gap between them. The cache may be a factor. The cache is placed on the same on-chip memory as the shared memory, and should therefore be equally fast. The difference is that caching is not programmable and therefore not controlled by the programmer. However some properties in the local memory may suggest that it is a more likely candidate to be cached. The local memory is thread dependent and is not accessible to other threads or blocks as the global memory are. The compiler can therefore logically imply that the data is not going to be modified by other threads and caching becomes much more likely. Figure 3.5.1, also shows us that the cache can fit the whole s-stack in a block, which correlates with the timing results. To enforce cache use even further, CUDA gives a runtime option to enforce more of the on-chip memory to caching.

Testing different memory locations for the s-stack, showed that the memory location is important for performance. Even small improvements in the performance of the s-stack, gives a significant improvement in overall runtime. This implies that the k-heap should be highly optimized.

Figure 3.5.3 shows the runtime difference between two k-heap variants. One uses a bobble sort [10] like implementation. It works by always keeping a sorted list. Elements are inserted by placing it at the end of the list, and swapping it to the adjustment element, until it is in the right place. The other method is based on a heap sort implementation, that is explained in Section 3.3.3. The performance difference is the insertion time, where the bobble variant is a  $O(n)$  time complexity, while heap sort variant has a  $O(\log_2(n))$ . Resulting in a almost 7 times faster k-heap, with only a  $k$  value of 100.



**Figure 3.5.2:** Search time comparison between different stack memory types. The test are done with  $k$  equals 10 and  $1 \cdot 10^6$  queries per tree size.



**Figure 3.5.3:** Timing results from two different k-heap implementations, with varying  $k$  and  $1 \cdot 10^6$  reference points.

### Open-MP

The high impact the stack had on performance make an interesting question in regard to RQ 6. Could a parallel implementation on the CPU outperform the CPU version? When the latency effect, as the stacks showed, had such a huge impact on the performance. The CPU has a lot more cache then the GPU and would therefor not be affected that much by memory overhead. The question if this is enough to offset the lower number of parallel threads on the CPU.



For this to be investigated properly, an OpenMP version of the k-d tree search has to be created. The parallelization strategy is the same as for the CUDA implementation, only differing in implementation details. Since the CPU has vastly more cache and the system memory latency is not very high, we do not need to consider memory usage in the same manner as with the CUDA implementation. The implementation can be found in Appendix C.4.2.

## 3.6 CUDA Optimizations

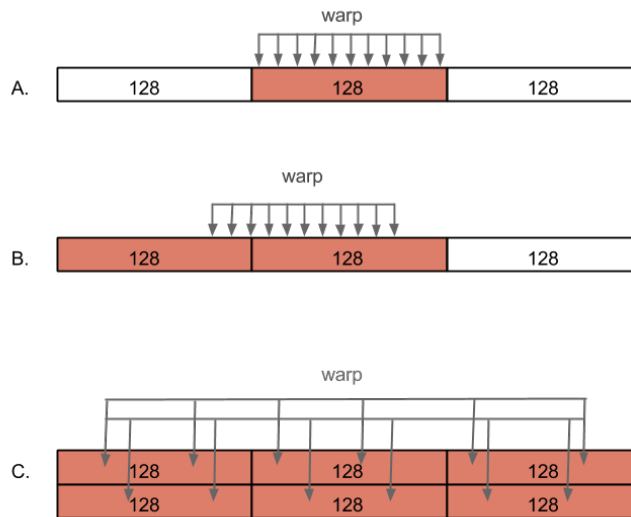
Throughout this quest many optimizations have been done, some focusing on the algorithmic aspect, others more on implementation. This section is focusing on different CUDA optimizations and why it is necessary in regard of performance. Some performance considerations, like divergence, have already been mentioned, due to it's direct relation to the different implementations. Other important factors, occupancy, coalescing, loop-unrolling, block and thread load balancing.

Occupancy is a metric, which relates to how many active warps there are on a SM. Earlier we have talked about how thread instructions are executed sequentially, resulting in alternating warps, one warp is paused while the other is executing. The time a stalled warp will use to retrieve data, increases with the number of warps per SM. One should note that high occupancy does not always result in high performance, but low occupancy will always result in an inability to hide latency, which result in bad performance.

The struggle to always have the right amount of occupancy, also relates to dividing CUDA resources. As well as keeping a right amount of warps in a SM, one must also keep every SM in activity. Forcing the algorithm to work over unsynchronizable blocks. The number of blocks, that are optimal to keep in activity, changes with different GPUs. It is therefore important to think of how many blocks and threads that are launched with each kernel. We have solved this issue with methods that, based on different algorithmic parameters, calculates how many threads and blocks are needed for a particular launch.

To coalesce memory access to global memory, is probably one of the most performance increasing optimizations in CUDA, especially in our memory intense application. Global memory that is loaded and stored by threads in a warp, can be coalesced into only one transaction, if the right conditions are met. How a device coalesce memory depends on the compute capability, but some basic properties are common. A warps access will coalesce into onto a number of transactions that equals the number of cache lines needed to service all the threads in the warp. Devices with compute capability  $2.x$  will by default cache directly to L1, which has 128-byte lines. Higher capabilities will always cache to L2 cache, that have 32-byte segments [2].

If we focus on compute capability  $2.x$ , Figures 3.6.1, shows how memory are coalesced. Green indicate memory lines that are retrieved, while blue indicate non retrieved lines. The first figure illustrate perfect coalescing, a warp performs a sequential 128-byte transaction that fit perfectly in a 128-byte lines. The second shows a misaligned sequential retrieval,



**Figure 3.6.1:** Three different memory transactions, where A and B result in good coalesced and cached transactions, while C shows a stride access pattern with bad coalescing.

resulting in two transactions. The third uses stride access pattern with a offset of 128 resulting in bad coalescing.

We have tried to maximize coalescing by always using sequential addressing. This kind of addressing can be achieved in many ways. One way, that we have use throughout the code, is based on how data are partitioned and iterated. The generic partitioning algorithm, Algorithm 3.1, that we used to expand Garcia’s algorithm, shows how this could be done.

The last optimization keyword we would like to introduce is unrolling. This is a technique we have used on many of our algorithms, like min-reduce, and also in some of our utility functions, like for example to accumulate an array. Unrolling is a standard technique in ordinary high performance serial programming, optimizing pipelining, and is given an extra dimensions on CUDA.

Loop unrolling, is the procedure of rewriting a loop, containing conditional operators, into hard-coded sequential steps. This way, the result of conditional operators may be determined at compile-time, eliminating branching of the control flow. On a CUDA context this is of course the case, but in addition it will minimize divergence.

The idea of loop unrolling can also be applied to warps. This is called warp unrolling, and it can be used if we know we are in a single warp. The results being, that no expensive thread synchronization is needed, since every warp is accessing a unique memory location.

# Final results, discussion and conclusion

## 4.1 Test environment

To test and investigate our results three test environments have been used, as shown in Table 4.1.1. They all have different properties that are used to test different aspects of our algorithms. Test environment 3 has a normal Windows setup, with a graphics card that is used for both display rendering and CUDA computation. On the other hand, test environment 1, uses the same hardware, but is setup with a dedicated GPU. CUDA computations done on a environment without a dedicated GPU, is forced to split the resources with the running OS, which implies some restrictive properties. For instance, it is normal for an OS to kill long running GPU processes. On windows the normal timeout is around 30 seconds. To get around this restriction some editing in regedit is necessary.

The last environment, number 2, is build on an Amazon web service (AWS) cloud instance. Using AWS, enables us to test implementations on a more powerful GPU, than what we currently have in our personal proccession. The biggest advantage of the AWS GPU for our applications, is the increased on-board GPU memory. This enables us to store larger point clouds directly on the GPU. The increased number of CUDA cores, also makes it possible to test performance ratios, and different variants of resource distributions.

All tests was run multiple times and averaged, to get more accurate results. It should also be noted that it is common practice to warm up the GPU before any test, because it may take as long time for the CUDA runtime to create a CUDA context, as launching the kernel itself.

All tests used synthetic data, generated as uniformly distributed random points in a unit cube. Where needed, random points was generated and stored to disk. This data could

<b>Test environment</b>	<b>1</b>	<b>2</b>	<b>3</b>
<b>OS</b>	Ubuntu 14.04	Ubuntu 12.04	Windows 7
<b>OS type</b>	x64	x64	x64
<b>Kernel</b>	3.13.0-24-generic	3.2.0-58-virtual	Windows 7
<b>CPU</b>	i7-2600K	E5-2670	i7-2600K
<b>CPU memory</b>	7.8 Gb	16 Gb	7.8 Gb
<b>GPU</b>	GeForce GTX 560 Ti	NVIDIA GRID K520	GeForce GTX 560 Ti
<b>GPU memory</b>	1024 Mb	4095 MB	1024 Mb
<b>Dedicated GPU</b>	Yes	Yes	No
<b>CUDA cores</b>	384	1536	384
<b>CUDA capability</b>	2.1	3.0	2.1
<b>CUDA driver</b>	5.5	5.5	5.5
<b>CUDA runtime</b>	5.5	5.5	5.5

**Table 4.1.1:** Tabulated information about the three test environments.

then be used as the source for several different tests, eliminating deviation in tests used for comparison of between different implementations.

Using uniformly distributed data is not necessarily a good representation for all real world point cloud data, but this should not affect our results for the brute-force and k-d tree build algorithm, given that these algorithms have a non-stochastic runtime, not affected by the location of our test points. Given that we always balance the k-d tree, the k-d query algorithm should not be significantly affected by changing the point distribution, although slight deviations might be observed.

Due to the reasons above, and the available time for this thesis, we have chosen not to include other point distributions in our tests, but this could be an interesting study for further work.

## 4.2 Final results and discussion

During the development of our algorithms, we have presented many intermediate results, in order to argument for the design and implementation choices made. In this section, we will present our final results for the GPU parallelized brute-force, GPU parallelized and CPU parallelized k-d tree based kNN algorithm.

The different research questions stated during development of our algorithms are revisited, and answered are given, based on the presented results.

### 4.2.1 Solving the kNN problem

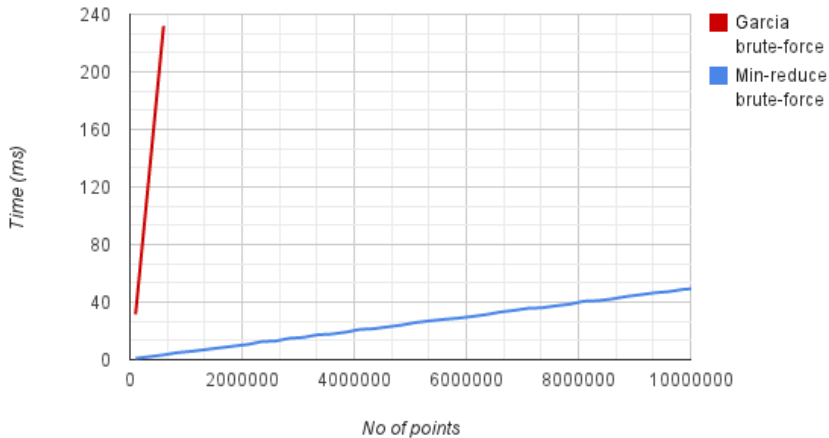
In Section 3.2 we started on our quest for a faster kNN search, by investigating a possible brute-force algorithm, pioneered by Garcia et.al. The following research question was

asked.

**RQ 1.** *Can high performance be achieved by a parallel brute-force kNN algorithm on large point clouds.*

In order to discuss this question, we have to clarify what we consider to be high performance in this context. The work of Garcia et.al. contains the fastest brute-force algorithm, for solving the kNN problem, we have found in current literature. We would therefore consider a kNN algorithm to have high performance, if it is able to solve the kNN problem for point cloud data, in the 3D format specified by TSI, at comparable speeds to the algorithm developed by Garcia et.al.

Although the algorithm developed by Garcia et.al. is the fastest brute-force algorithm we have managed to find, this is not as a high benchmark as it might initially seem. The algorithm developed by Garcia et.al. is optimized for solving the kNN problem for a more general version of the kNN problem than required by TSI. Where we are only concerned with solving the kNN problem for three dimensions, Garcia's algorithm will solve problems stated in any dimension. Given that we solve a more restricted version of the kNN problem, any less than comparable speeds to the implementation made by Garcia et.al. could not be considered to be of high performance in our eyes.



**Figure 4.2.1:** Comparison of brute-force algorithm developed by Garcia et.al. and min-reduce based brute-force algorithm developed in this thesis.  $k$  is fixed at 10 and  $m$  is increasing.

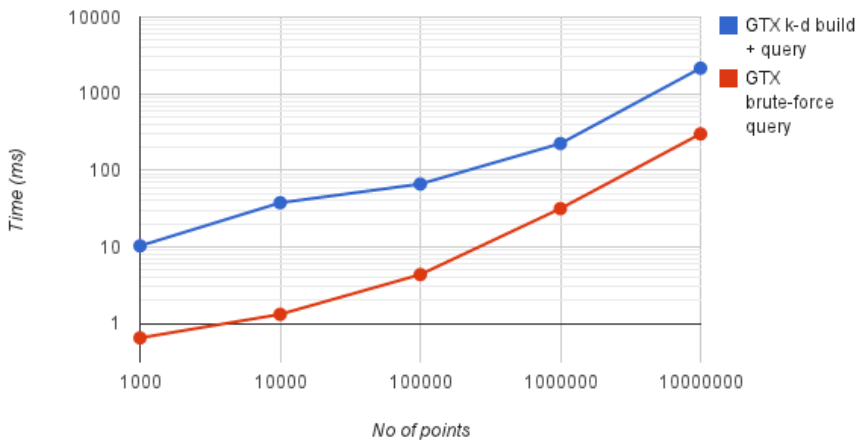
In Section 3.2.2, we discussed Figure 3.2.2. As a reminder, part of this figure is presented as Figure 4.2.1, which shows the result of a comparison between the brute-force algorithm developed by Garcia et.al. and our min-reduce brute-force algorithm developed in Section 3.2.2. The test is performed with a low value for  $k = 10$ , and focuses on performance for large values of number of points,  $m$ . In this test, our min-reduce brute-algorithm is

shown to be almost 70 times faster than the algorithm developed by Garcia et.al. In addition, it is capable of solving the kNN problem for much larger values of  $m$ .

We therefore conclude that RQ 1 can be answered with yes. High performance can be achieved with a brute-force based algorithm.

In Section 3.3 we introduced the k-d tree, a data-structure with a known  $O(\log(m))$  nearest neighbor query algorithm. We therefore presented a new research question, which we try to answer in Figure 4.2.2.

**RQ 3.** *It is possible to use a k-d tree to increase the performance of kNN queries, compared to a parallel brute-force solution?*

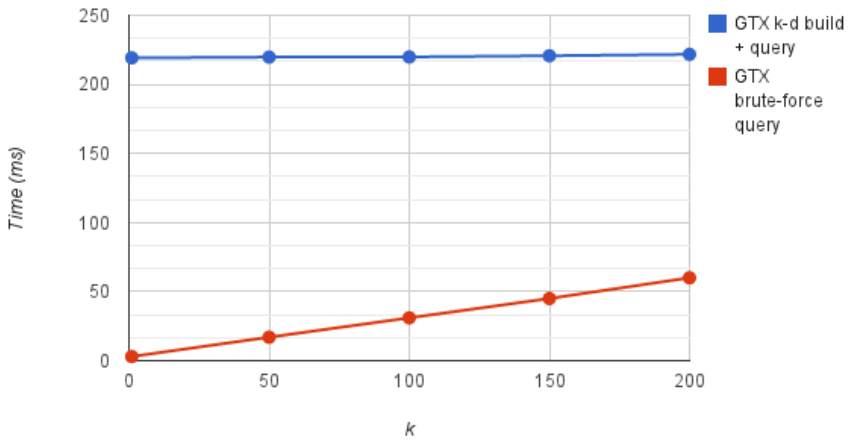


**Figure 4.2.2:** Comparison of min-reduce brute-force and k-d tree based algorithms for solving the kNN problem for  $k = 100$  and increasing  $m \leq 1e7$ .

The graph compares the runtime of the k-d tree build and query algorithms developed in Section 3, with the min-reduce brute-force algorithm. All test data is generated using test environment 1.

Figure 4.2.3 compares the same two algorithms, but for a fixed value of  $m$ , and increasing value of  $k$ . This test is also performed using test environment 1.

Both Figure 4.2.2 and 4.2.3 shows that this algorithm is slower than a brute-force approach. This answers RQ 3. In order to solve the kNN problem, the k-d tree based solution first has to construct the k-d tree, then query for the closest points. Given that our previous calculation shows that the time complexity of the k-d tree build algorithm is larger than the time-complexity for the brute-force algorithm, this result should come as no surprise. Constructing the k-d data-structure for a single query is simply a waste of resources, if just one kNN query is to be performed.



**Figure 4.2.3:** Comparison of min-reduce brute-force and k-d tree based algorithms for solving the kNN problem for  $m = 1,0 \cdot 10^6$  and increasing  $k \leq 200$ .

## 4.2.2 Solving the All-kNN problem

The All-kNN problem was studied in both relation to the brute-force algorithm and the k-d tree based algorithm. In Section 3.2.2 we proposed RQ 2. We also wanted to compare our parallel k-d tree based algorithm to the brute-force algorithm, postulated in RQ 4.

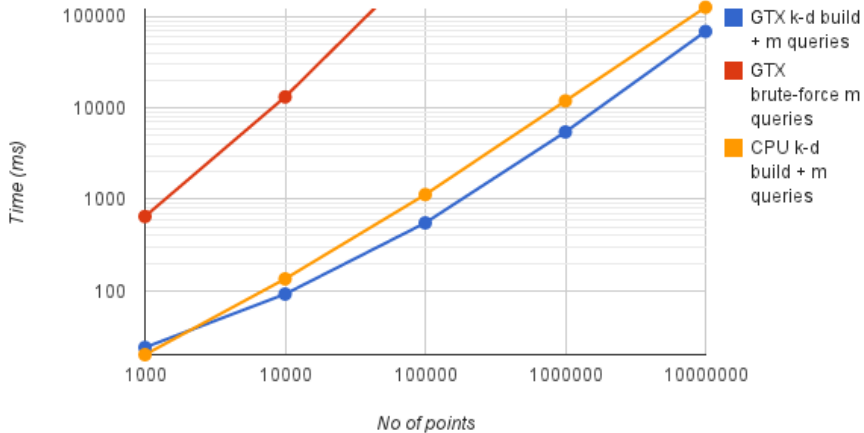
**RQ 2.** *Can a parallel brute-force kNN algorithm be fast enough to solve the All-kNN problem within reasonable time?*

**RQ 4.** *It is possible to use a k-d tree to increase the performance of All-kNN queries, compared to a parallel brute-force solution?*

Figure 4.2.4 compares the runtime of the min-reduce brute-force algorithm to the k-d tree based algorithm, for increasing values of  $m \leq 1e7$ , and a fixed value of  $k = 100$ . The data series for the min-reduce algorithm is estimated from the data obtained in Figure 4.2.2. This estimation is valid, since all GPU resources are used to perform one kNN query, when using the brute-force algorithm. Solving the All-kNN problem with the brute-force algorithm, can therefore only be performed as repeated application of the brute-force kNN algorithm, given a reasonable hardware setup, with one CUDA enabled GPU.

The data series for the k-d tree based algorithms are, on the other hand, generated from actual runtime results. This is due to the k-d query algorithm being developed as a parallelized Q-kNN query, where individual kNN queries are parallelized, instead of one single kNN query, as is the case with the brute-force algorithm. We have developed two different implementations of this k-d tree based Q-kNN query. One parallelized on the GPU, anno-

tated with GTX k-d build +  $m$  queries, and one parallelized on the CPU, annotated with CPU k-d build +  $m$  queries. All tests was performed using test environment 1.



**Figure 4.2.4:** Comparison of min-reduce brute-force and k-d tree based algorithms with CPU and GPU parallelized query. The graph compares runtime for solving the All-kNN problem for  $k = 100$  and increasing  $m$ .

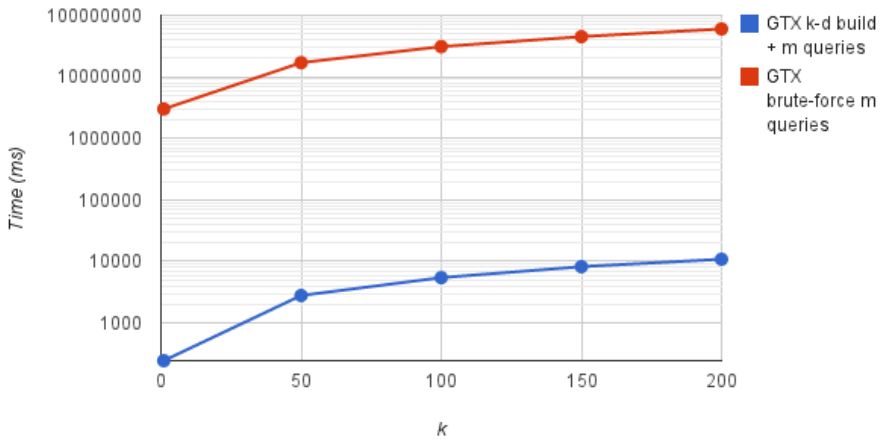
Figure 4.2.5 compares the GPU parallelized k-d tree algorithm, with the parallel brute-force algorithm, for a fixed value of  $m$ , and increasing values for  $k \leq 200$ . Test environment 1 is also used in this comparison.

Both graphs clearly shows the benefit of using a k-d tree based algorithm for solving the All-kNN problem. As discussed in Section 3.3.4, part of this increased performance over a brute-force based solver, is due to the k-d query algorithm being able to execute in  $O(\log(m))$  time. Since we do not have to rebuild the tree between the individual kNN queries when performing a All-kNN query, the reduction in query runtime has larger impact on the overall execution of the algorithm.

Another important factor is that each k-d kNN query, is parallelized. This can be done for a k-d based query algorithm, since the resource requirements for each individual kNN query is lower than for the brute-force algorithm. These two improvements combined, result in an algorithm that is capable of solving the All-kNN problem for values of  $m$  and  $k$ , that would not be feasible with a brute-force algorithm. This answers RQ 4.

Figure 4.2.4 shows that the brute-force algorithm is capable of solving the All-kNN for small point clouds, although significantly slower than the k-d tree based algorithm. With a point cloud containing just 10000, the brute-force algorithm will take at least 10 s to execute, compared to the 100 ms required by the k-d tree based algorithm. Although slow, for low values of  $m$ , the brute-force algorithm computes the answer within arguably





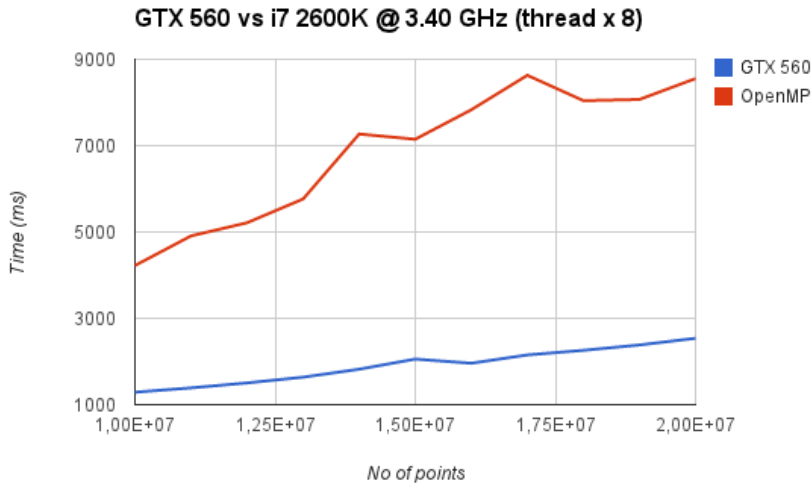
**Figure 4.2.5:** Comparison of min-reduce brute-force and GPU parallelized k-d tree based algorithms for solving the All-kNN problem for  $m = 1e6$  and increasing  $k \leq 200$ .

reasonable time. When  $m$  is increased, this changes. For a point cloud of size  $1e7$ , the brute-force algorithm would require about  $3e9$  ms to compute the answer, or almost 34 days. This would most certainly not be considered to be within reasonable time. The answer to RQ 2 is therefore dependent on the size of  $m$ . It can be argued that the brute-force algorithm is capable, but not the best alternative, for solving the All-kNN problem for small values of  $m$ .

In Figure 4.2.4 the impression is that the GPU and CPU parallelized k-d algorithms performs similarly. We will therefore investigate additional results, to get a better understanding of how they compare.

Figure 4.2.6 compares the difference between the CPU and the GPU parallelized k-d query algorithm. Test environment 1 is again used.

In Figure 4.2.6 the CPU based parallelization is slower than the GPU based parallelization. Although smaller than the difference between the brute-force algorithm and the k-d tree based algorithm, it is still a significant difference. This indicates, that for this algorithm, the benefit of having faster individual cores, and less overhead related to memory transfer on the CPU, is not enough to offset the drawback of the CPU has a lot fewer parallel cores than the GPU. Using the GPU parallelized version, where possible, is therefore recommended.



**Figure 4.2.6:** Comparison of runtime for GPU (GTX 560) and CPU (OpenMP) parallelized k-d tree based n query.

### 4.2.3 Parallelization performance increase

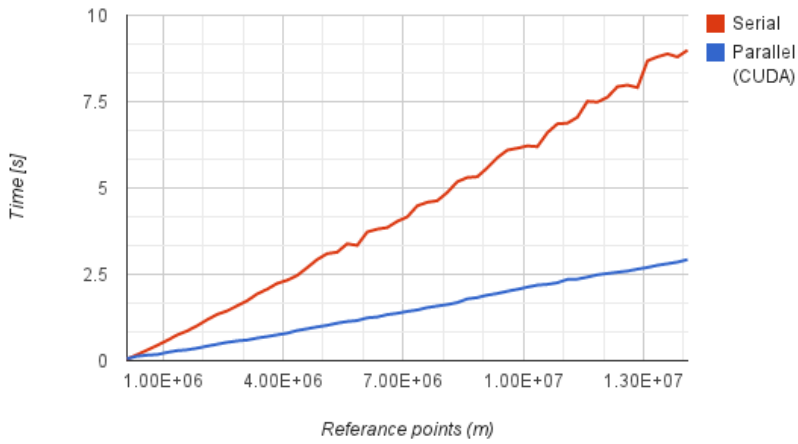
Parallelization of the k-d tree build was introduced with RQ 5, in Section 3.4.

**RQ 5.** *It is possible to parallelize the k-d tree build algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm.*

This research question is based around the complex nature of the k-d tree build, and the uncertainty of it achieving a acceptable parallel speedup. This question was investigated though implementation prototypes, together with a thorough discussion about the parallelization strategy and the intermediate results.

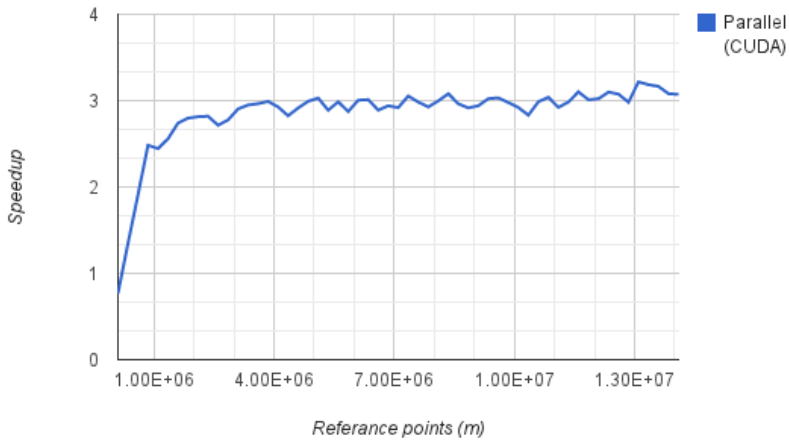
Figure 4.2.7 tries to answer RQ 5, by comparing the serial and parallel k-d tree build implementation. Both graphs follows the same trend, which correlates with the shared time complexity of  $O(m \log(m))$ . We see that the impact of the parallel overhead is decreasing as the problem size increase, and the profit of multiple cores is getting more and more be dominant. Resulting in a faster parallel implementation.

To get a better picture of the parallel improvement, it is natural to talk about parallel speedup. Figure 4.2.8 shows how the parallel speedup develops, as the problem size increase. Here we see that the speedup starts below 1, indicating that the serial version is faster then the parallel version, but from Figure 4.2.7 one can see that the time to build such small k-d trees is almost negligible. As the problem size increase, the trend quickly changes, until the speedup flattens out. The speedup increases as the problem size allows utilization of more and more threads, until the limit is reached, and the curve flattens out



**Figure 4.2.7:** Comparison between serial and parallel k-d tree build performance.

into a lower gradient.



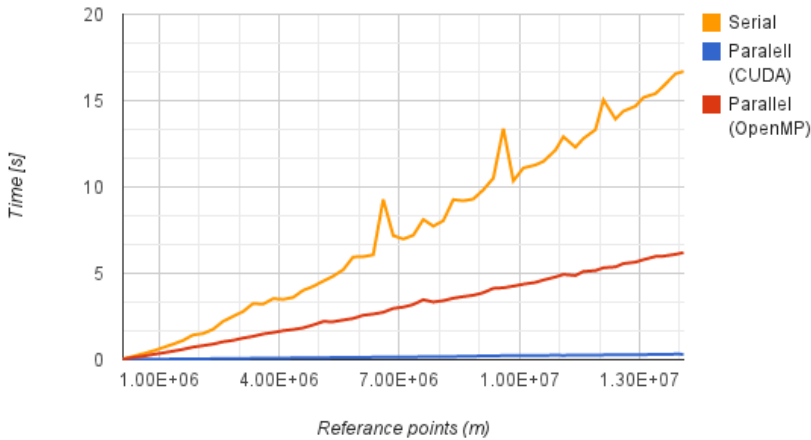
**Figure 4.2.8:** Parallel speedup for the k-d tree implementation for varying values of  $m$ .

With the complex nature of the k-d tree build process, a speedup of three is acceptable, and we consider overall performance increase to be significant compared to the serial

algorithm, answering RQ 5.

Parallelization of the All-kNN query was introduced with RQ 6, in Section 3.5.

**RQ 6.** *It is possible to parallelize the All-kNN query algorithm, in such a way that it gives a significant speed improvement compared to the serial algorithm.*

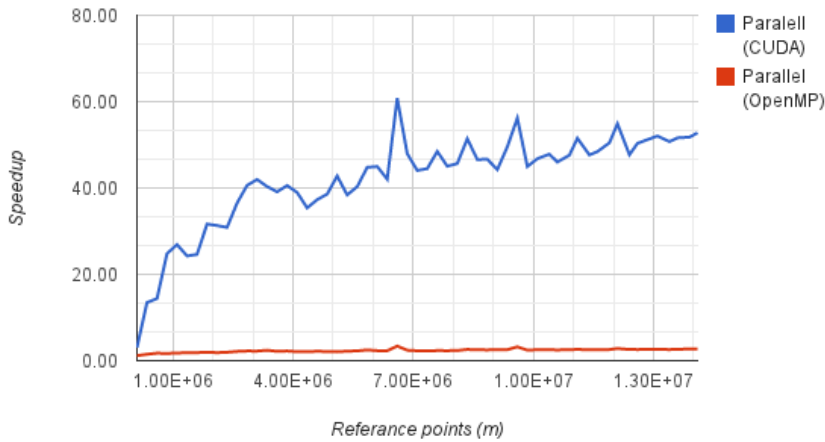


**Figure 4.2.9:** Comparison between serial and parallel All-kNN query performance.

Figure 4.2.9 display the results from the two different parallel All-kNN query implementations, CUDA and OpenMP, compared to the serial version. The linear trend, also found in the k-d tree build algorithm, is not surprising, as the time complexity for all algorithms are  $O(m \log(m))$ . The parallel improvement is only shown in the gradient these slopes have, which is reasonable, because the work is only divided amongst more cores. In both OpenMp and CUDA the parallel improvement is significant.

If we look at the parallel speedup, shown in Figure 4.2.10, we can again conclude that the OpenMP version is outperformed by the CUDA implementation. The trend resembles what we saw in the k-d tree build parallelization, only this time the speedup goes towards 50 in the CUDA version. This correlations well with the discussion in Section 3.5.1, and we can answer RQ 3. Our All-kNN query has a significant parallel improvement.

An final note, is that the speedup for the k-d tree based All-kNN algorithms are lower than the speedup for both our and Garcia's[13] brute-force implementations, which shows that speedup don't equal a fast implementations for this problem.



**Figure 4.2.10:** Parallel speedup comparison for the All-kNN query between the CUDA and OpenMP implementation.

## 4.3 Conclusion

In this thesis, we have investigated the possible benefits of using general-purpose computing on graphics processing units, in order to speed up the execution of calculations in engineering applications. We have investigated this topic, by improving the performance of point cloud analysis in engineering software developed by TechnoSoft Inc.

By utilizing the parallelization possibilities offered by CUDA enabled GPUs, and optimizing our algorithms for 3D point cloud data, we have been able to develop fast algorithms for solving the kNN and All-kNN problem.

The parallel brute-force algorithm developed in this thesis, is 70 times faster than the brute-force algorithm developed by Garcia et.al. [13] on comparable problem sizes. Considering the algorithm developed by Garcia et.al. is significantly faster than conventional libraries, being up to 407 times faster than Matlab, and up to 148 times faster than ANN [13, Table 1], this is a notable result.

A parallel k-d tree based Q-kNN algorithm has also been developed in this thesis, and optimized for solving the All-kNN problem. The parallel k-d tree based algorithm is able to solve the All-kNN problem 300 times faster than the parallel brute-force implementation, and this could enable All-kNN analysis of much larger point clouds than was previously feasible.

In addition, all algorithms has been implemented with memory scalability in mind, resulting in a finished library of algorithms, which solves kNN and All-kNN problems faster,

---

and for larger point clouds, than other alternatives, known from literature. This library is currently being integrated into point cloud analysis software at TechnoSoft Inc.

In conclusion, our results indicate that large runtime improvements can be achieved in engineering software, by utilizing the parallel performance of GPUs to speed up time-consuming algorithms.

# Bibliography

- [1] Automatically tuned linear algebra software (atlas) homepage. (available online at <http://math-atlas.sourceforge.net/>), 2014.
- [2] Cuda c best practices guide. (available online at <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>), 2014.
- [3] Cuda c programming guide. (available online at <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz32C74scPM>), 2014.
- [4] Homepage cublas. (available online at <https://developer.nvidia.com/cuBLAS>), 2014.
- [5] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *J. Exp. Algorithmics*, 17:4.2:4.1–4.2:4.29, Oct. 2012.
- [6] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, pages 307–314, New York, NY, USA, 1968. ACM.
- [7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
- [8] S. Brown and J. Snoeyink. Gpu nearest neighbor searches using a minimal kd-tree. In *Proc. MASSIVE*, June 2010. (available online at <http://cs.unc.edu/~shawndb/>).
- [9] J. S. K. T. S.-H. C. Cayman Mitchell, Nelson Schoenbrot. Radix selection algorithm for the k order statistic. may 2012.
- [10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

- 
- [11] J. Fang, A. L. Varbanescu, and H. J. Sips. A comprehensive performance comparison of CUDA and openCL. In *Proc. International Conference on Parallel Processing (40th ICPP'11) USB*, pages 216–225, Taipei, Taiwan, Sept. 2011. IEEE Computer Society.
- [12] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, Sept. 1977.
- [13] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. *CoRR*, abs/0804.1448, 2008.
- [14] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3757–3760, Sept 2010.
- [15] M. Harris. Optimizing parallel reduction in cuda. (available online at <http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>), 2014.
- [16] T. Karras. Collision detection on the gpu. 2012.
- [17] D. Merrill and A. S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(2):245–272, 2011.
- [18] G. Nolan. Improving the k-nearest neighbour algorithm with cuda. 2009.
- [19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [20] Y. L. L. J. Shenshen Liang, Cheng Wang.
- [21] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, Dec. 2008.



Appendix **A**

## Data sheets

Variable reference points	K=100									
(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
GTX k-d build	GTX k-d query	GTX k-d m query	GTX k-d build + query	GTX k-d build + m queries	GPU k-d m query	GPU k-d build + m queries	GTX brute-force query	GTX brute-force m queries		
4	6.33	20	10.30717	24	16	20	0.65	650		
31	6.73	61	37.733423	92	104	135	1.32	13200		
59	7.27	494	66.267632	553	1069	1128	4.36	436000		
213	10.56	5245	223.556185	5458	11728	11941	31.66	3166000		
2109	44.85	66624	2153.850982	68733	124203	126312	299	2990000000		
<b>1.00E+04</b>	<b>1.00E+05</b>	<b>1.00E+06</b>	<b>1.00E+07</b>							
5.86922	6.26602	8.71837	44.47341							
3.79834	7.0743	13.06278	42.81011							
6.18493	7.39702	8.82883	48.44749							
8.53898	8.45648	11.42278	48.46307							
8.36416	8.00269	12.06602	43.21453							
6.95187	8.07635	9.91392	42.84016							
6.19469	4.98074	9.98352	46.37597							
7.76026	9.74704	10.94211	43.52374							
6.17837	6.90768	9.73318	44.89286							
7.49341	5.768	10.89034	43.46848							
6.733423	7.267632	10.556185	44.850982							





# Appendix **B**

## Source code documentation

### **B.1** Api documentation

# KNN GPGPU Documentation

## Includes

**point.h** Contains definitions of the different point struct data-types used by the knn gpgpu algorithms.

## Members

**void buildKdTree(struct Point *points*, int *n*, struct Node *tree*)** Accepts a list of *n* PointS. Builds a balanced kd-tree from these points on the GPU, and writes this tree to the Point list tree.

**void queryAll(struct Point *query\_points*, struct Node *tree*, int *n\_qp*, int *n\_tree*, int *k*, int *\*result*)** Queries a previously built kd-tree of size *n\_tree* for the *k* closest neighbors to the points specified in the *query\_points* list of size *n\_qp*. The index location of the *k* closest points are written to the result array. Uses a wrapper to partition the problem, in order to handle memory overflow.

**void cuQueryAll(struct Point *query\_points*, struct Node *tree*, int *n\_qp*, int *n\_tree*, int *k*, int *\*result*)** Queries a previously built kd-tree of size *n\_tree* for the *k* closest neighbors to the points specified in the *query\_points* list of size *n\_qp*. The index location of the *k* closest points are written to the result array.

**void mpQueryAll(struct Point *query\_points*, struct Node *tree*, int *n\_qp*, int *n\_tree*, int *k*, int *\*result*)** Performes same operations as cu-QueryAll, but is parallelized on the CPU using OpenMP instead of CUDA.

**void knn\_brute\_force\_garcia(float *ref\_host*, int *ref\_width*, float *query\_host*, int *query\_width*, int *height*, int *k*, float *dist\_host*, int *ind\_host*)** Performs a brute force knn-search based on the code written by Garcia.

**void knn\_brute\_force(float *ref\_host*, int *ref\_nb*, float *query\_host*, int *dim*, int *k*, float *dist\_host*, int *ind\_host*)** Performs a improved brute force knn-search.

## Utils

**size\_t getFreeBytesOnGpu()** Return the current amount of free memory on the GPU in bytes.

**void cuSetDevice(int device)** Sets device as the current device for the calling host thread.

**int cuGetDevice()** Returns the device on which the active host thread executes the device code.

**int cuGetDeviceCount()** Returns the number of devices accessible.

**size\_t getNeededBytesForBuildingKdTree(int n\_tree)** Returns needed bytes on GPU to build a tree of size n\_tree.

**size\_t getTreeSize(int n\_tree)** Returns the size in bytes of a tree with length n\_tree.

**size\_t getNeededBytesForQueryAll(int n\_qp, int k, int n\_tree)** Returns needed bytes on GPU to perform a queryAll operation on CUDA.

**size\_t getNeededBytesInSearch(int n\_qp, int k, int n\_tree, int thread\_num, int block\_num)** Returns needed bytes on GPU to perform a queryAll operation on CUDA without taking the tree size into account.

## **B.2 Installation instructions**



## Installation notes for Ubuntu 13.04

### Installing CUDA

```
sudo apt-get install nvidia-cuda-toolkit
```

### Installing Git

```
apt-get install git
```

### Installing CMake

```
sudo apt-get install cmake
```

### Build with

```
...\tsi-gpu> mkdir build $$ cd build  
...\tsi-gpu/build> cmake ../  
...\tsi-gpu/build> make
```

All executables will be in /build/bin and all libraries will be in /build/lib/.

## Installation notes on an Amazon instance

1. Create an amazon instance by following amazon's instructions (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html>). Select ubuntu-precise-12.04-amd64-server, and select the GPU instance type g2.2xlarge.
2. After the instance is setup, SSH into it.
3. Setup dependencies needed to install CUDA (gcc):

```
sudo apt-get update
sudo apt-get install gcc
```

4. Download and install CUDA. For our choice in os, grab the following .deb file.:

```
wget http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1204/x86_
```

5. Then run:

```
sudo dpkg -i cuda-repo-ubuntu1204_5.5-0_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

6. Setup environment; Run the following lines. Add them to ~/.bashrc to make it permanent.

```
export PATH=/usr/local/cuda-5.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64:$LD_LIBRARY_PATH
```

7. Install CUDA samples (optional) to some directory:

```
cuda-install-samples-5.5.sh .
```

8. Verify an example works:

```
cd <NVIDIA_CUDA-5.5_Samples>/1_Uutilities/deviceQuery
make
./deviceQuery
```

9. Now that CUDA is installed, lets start building the project. First, clone the project from Github.

```
sudo apt-get install git
git clone https://github.com/hgranlund/tsi-gpgpu.git
```

10. Install cmake and build the project.

```
cd tsi-gpgpu
sudo apt-get install cmake
mkdir build && cd build
cmake ..
make
```

11. Now the project is created, test the solution by:

```
make test
```



## Source code

### C.1 API header

```
#ifndef _KNN_GPGPU_
#define _KNN_GPGPU_

#include "point.h"

void buildKdTree(struct Point *points, int n_tree, struct
    Node *tree);

void queryAll(struct Point *h_query_points, struct Node *
    h_tree, int n_qp, int n_tree, int k, int *h_result);
void cuQueryAll(struct Point *query_points, struct Node *
    tree, int n_qp, int n_tree, int k, int *result);
void mpQueryAll(struct Point *query_points, struct Node *
    tree, int n_qp, int n_tree, int k, int *result);

void knn_brute_force_garcia(float *ref_host, int ref_width,
    float *query_host, int query_width, int height, int k,
    float *dist_host, int *ind_host);
void knn_brute_force(float *ref_host, int ref_nb, float *
    query_host, int dim, int k, float *dist_host, int *
    ind_host);

// #### Utils
size_t getFreeBytesOnGpu();
```

```

void cuSetDevice(int device);
int cuGetDevice();
int cuGetDeviceCount();

// Tree build
size_t getNeededBytesForBuildingKdTree(int n_tree);
size_t getTreeSize(int n_tree);

// Search
size_t getNeededBytesForQueryAll(int n_qp, int k, int
    n_tree);
size_t getNeededBytesInSearch(int n_qp, int k, int n_tree,
    int thread_num, int block_num);

#endif // _KNN_GPGPU_

```

## C.2 Brute Force

```

#ifndef _DATA_TYPES_
#define _DATA_TYPES_

struct Distance
{
    int index;
    float value;

    __device__ __host__ volatile Distance &operator=(
        volatile Distance &a) volatile
    {
        index = a.index;
        value = a.value;
        return *this;
    }
};

#endif // _DATA_TYPES_

```

### C.2.1 Bitonic sort version

```

#ifndef _KNN_BRUTE_FORCE_
#define _KNN_BRUTE_FORCE_

__global__ void cuComputeDistanceGlobal( float *ref, int
    ref_nb, float *query, int dim, float *dist);

```

```

__global__ void cuBitonicSort(float *dist, int *ind, int n,
    int dir);
__global__ void cuParallelSqrt(float *dist, int k);

void bitonic_sort(float *dist_dev, int *ind_dev, int n, int
    dir);

void knn_brute_force_bitonic_sort(float *ref_host, int
    ref_nb, float *query_host, int dim, int k, float *
    dist_host, int *ind_host);

#endif

// Includes
#include <kNN-brute-force.cuh>
#include <stdio.h>
#include <math.h>
#include <cuda.h>
#include <time.h>
#include <assert.h>

#include "helper_cuda.h"

#define SHARED_SIZE_LIMIT 1024U
#define checkCudaErrors(val)          check ( (val), #val,
    __FILE__, __LINE__ )

__device__ void cuCompare(float &distA, int &indA, float &
    distB, int &indB, int dir)
{
    float f;
    int i;
    if ((distA >= distB) == dir)
    {
        f = distA;
        distA = distB;
        distB = f;
        i = indA;
        indA = indB;
        indB = i;
    }
}

__constant__ float query_dev[3];

```

```

--global-- void cuComputeDistanceGlobal( float *ref, int
    ref_nb, int dim, float *dist, int *ind)
{
    float dx, dy, dz;

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    while (index < ref_nb)
    {
        dx = ref[index * dim] - query_dev[0];
        dy = ref[index * dim + 1] - query_dev[1];
        dz = ref[index * dim + 2] - query_dev[2];
        dist[index] = (dx * dx) + (dy * dy) + (dz * dz);
        ind[index] = index;
        index += blockDim.x;
    }
}

--global-- void cuBitonicSortOneBlock(float *dist, int *ind
    , int n, int dir)
{
    int blockoffset = blockIdx.x * blockDim.x * 2;
    dist += blockoffset;
    ind += blockoffset;

    for (int size = 2; size <= blockDim.x * 2; size <<= 1)
    {
        int ddd = dir ^ ((threadIdx.x & (size / 2)) != 0);
        for (int stride = size / 2; stride > 0; stride >>=
            1)
        {
            __syncthreads();
            int pos = 2 * threadIdx.x - (threadIdx.x & (
                stride - 1));
            cuCompare(dist[pos], ind[pos], dist[pos +
                stride], ind[pos + stride], ddd);
        }
    }
}

--global-- void cuBitonicSort(float *dist, int *ind, int n,
    int dir)
{

```



```

int blockoffset = blockIdx.x * blockDim.x * 2;
dist += blockoffset;
ind += blockoffset;

for (int size = 2; size <= blockDim.x * 2; size <<= 1)
{
    int ddd = dir ^ ((threadIdx.x & (size / 2)) != 0);
    for (int stride = size / 2; stride > 0; stride >>=
        1)
    {
        __syncthreads();
        int pos = 2 * threadIdx.x - (threadIdx.x & (
            stride - 1));
        cuCompare(dist[pos], ind[pos], dist[pos +
            stride], ind[pos + stride], ddd);
    }
}

int ddd = blockIdx.x & 1;
{
    for (int stride = blockDim.x; stride > 0; stride
        >>= 1)
    {
        __syncthreads();
        int pos = 2 * threadIdx.x - (threadIdx.x & (
            stride - 1));
        cuCompare(dist[pos], ind[pos], dist[pos +
            stride], ind[pos + stride], ddd);
    }
}
}

__global__ void cuBitonicMergeGlobal(float *dist, int *ind,
    int n, int size, int stride, int dir)
{
    int global_comparatorI = blockIdx.x * blockDim.x +
        threadIdx.x;
    int comparatorI = global_comparatorI & (n / 2 -
        1);

    int ddd = dir ^ ((comparatorI & (size / 2)) != 0);
    int pos = 2 * global_comparatorI - (global_comparatorI
        & (stride - 1));
    cuCompare(dist[pos], ind[pos], dist[pos + stride], ind[
        pos + stride], ddd);
}

```

```

}

--global-- void cuParallelSqrt(float *dist, int k)
{
    unsigned int xIndex = blockIdx.x;
    if (xIndex < k)
    {
        dist[xIndex] = sqrt(dist[xIndex]);
    }
}

--global-- void cuBitonicMergeShared(float *dist, int *ind,
    int n, int size, int dir)
{
    int blockoffset = blockIdx.x * blockDim.x * 2;
    dist += blockoffset;
    ind += blockoffset;
    int comparatorI = (blockIdx.x * blockDim.x + threadIdx.
        x) & ((n / 2) - 1);
    int ddd = dir ^ ((comparatorI & (size / 2)) != 0);
    for (int stride = blockDim.x; stride > 0; stride >>= 1)
    {
        __syncthreads();
        int pos = 2 * threadIdx.x - (threadIdx.x & (stride
            - 1));
        cuCompare(dist[pos], ind[pos], dist[pos + stride],
            ind[pos + stride], ddd);
    }
}

void bitonic_sort(float *dist_dev, int *ind_dev, int n, int
    dir)
{
    int max_threads_per_block = min(SHARED_SIZE_LIMIT, n);
    int blockCount = n / max_threads_per_block;
    int threadCount = max_threads_per_block / 2;
    blockCount = max(1, blockCount);
    threadCount = min(max_threads_per_block, threadCount);
    if (blockCount == 1)
    {
        cuBitonicSortOneBlock <<< blockCount, threadCount
            >>>(dist_dev, ind_dev, n, dir);
    }
}

```

```

    }
    else
    {
        cuBitonicSort <<< blockCount, threadCount>>>(
            dist_dev, ind_dev, n, dir);
        for (int size = 2 * max_threads_per_block; size <=
            n; size <<= 1)
        {
            for (int stride = size / 2; stride > 0; stride
                >>= 1)
            {
                if (stride >= max_threads_per_block)
                {
                    cuBitonicMergeGlobal <<< blockCount,
                        threadCount>>>(dist_dev, ind_dev, n,
                            size, stride, dir);
                }
                else
                {
                    cuBitonicMergeShared <<< blockCount,
                        threadCount>>>(dist_dev, ind_dev, n,
                            size, dir);
                    break;
                }
            }
        }
    }
}

int factorRadix2(int *log2L, int L)
{
    if (!L)
    {
        *log2L = 0;
        return 0;
    }
    else
    {
        for (*log2L = 0; (L & 1) == 0; L >>= 1, *log2L++);
        return L;
    }
}

void knn_brute_force_bitonic_sort(float *ref_host, int
    ref_nb, float *query_host, int dim, int k, float *

```

```

dist_host , int *ind_host)
{
    unsigned int size_of_float = sizeof(float);
    unsigned int size_of_int    = sizeof(int);

    float      *ref_dev;
    float      *dist_dev;
    int        *ind_dev;

    int log2L;
    int factorizationRemainder = factorRadix2(&log2L,
        ref_nb);
    // assert(factorizationRemainder == 1);

    checkCudaErrors(cudaMalloc( (void **) &dist_dev , ref_nb
        * size_of_float));
    checkCudaErrors(cudaMalloc( (void **) &ind_dev , ref_nb
        * size_of_int));
    checkCudaErrors(cudaMalloc( (void **) &ref_dev , ref_nb
        * size_of_float * dim));

    checkCudaErrors(cudaMemcpy(ref_dev , ref_host , ref_nb *
        dim * size_of_float , cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMemcpyToSymbol(query_dev ,
        query_host , dim * size_of_float));
    int threadCount = min(ref_nb , SHARED_SIZE_LIMIT);
    int blockCount = ref_nb / threadCount;
    blockCount = min(blockCount , 65000);
    cuComputeDistanceGlobal <<< blockCount , threadCount>>>(
        ref_dev , ref_nb , dim , dist_dev , ind_dev);
    bitonic_sort(dist_dev , ind_dev , ref_nb , 1);
    cuParallelSqrt <<< k , 1>>>(dist_dev , k);

    checkCudaErrors(cudaMemcpy(dist_host , dist_dev , k *
        size_of_float , cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(ind_host , ind_dev , k *
        size_of_int , cudaMemcpyDeviceToHost));

    checkCudaErrors(cudaFree(ref_dev));
    checkCudaErrors(cudaFree(ind_dev));
}

```

## C.2.2 min-reduce version

```
#ifndef _KNN_BRUTE_FORCE_REDUCE_
#define _KNN_BRUTE_FORCE_REDUCE_
#include <data_types.h>

__global__ void cuComputeDistance( float *ref, unsigned int
    ref_nb, float *query, unsigned int dim, Distance *
    dist);
__global__ void cuParallelSqrt(Distance *dist, unsigned int
    k);
void min_reduce(Distance *d_dist, unsigned int n, unsigned
    int k, unsigned int dir);

void knn_brute_force(float *ref_host, int ref_nb, float *
    query_host, int dim, int k, float *dist_host, int *
    ind_host);

#endif

#include "knn-brute-force-bitonic.cuh"
#include "knn-brute-force-reduce.cuh"
#include "knn-gpgpu.h"
#include "reduction-mod.cuh"

#include <stdio.h>
#include <math.h>

#include "helper_cuda.h"

#define SHARED_SIZE_LIMIT 512U
#define checkCudaErrors(val) check ( (val), #val,
    __FILE__, __LINE__ )

__constant__ float d_query[3];

__global__ void cuComputeDistance( float *ref, int ref_nb,
    int dim, Distance *dist)
{
    float dx, dy, dz;

    int index = blockIdx.x * blockDim.x + threadIdx.x;
```

```

while (index < ref_nb)
{
    dx = ref[index * dim] - d_query[0];
    dy = ref[index * dim + 1] - d_query[1];
    dz = ref[index * dim + 2] - d_query[2];
    dist[index].value = pow(dx, 2) + pow(dy, 2) + pow(
        dz, 2);
    dist[index].index = index;
    index += gridDim.x * blockDim.x;
}
}
__global__ void cuParallelSqrt(Distance *dist, int k)
{
    int xIndex = blockIdx.x;
    if (xIndex < k)
    {
        dist[xIndex].value = rsqrt(dist[xIndex].value);
    }
}

void knn_brute_force(float *h_ref, int ref_nb, float *
    h_query, int dim, int k, float *dist, int *ind)
{
    float          *d_ref;
    Distance       *d_dist, *h_dist;
    int i;
    h_dist = (Distance *) malloc(k * sizeof(Distance));
    checkCudaErrors(cudaMalloc((void **) &d_dist, ref_nb *
        sizeof(Distance)));
    checkCudaErrors(cudaMalloc((void **) &d_ref, ref_nb *
        sizeof(float) * dim));

    checkCudaErrors(cudaMemcpy(d_ref, h_ref, ref_nb * dim *
        sizeof(float), cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpyToSymbol(d_query, h_query,
        dim * sizeof(float)));

    int threadCount = min(ref_nb, SHARED_SIZE_LIMIT);
    int blockCount = ref_nb / threadCount;

    blockCount = min(blockCount, 65536);
    cuComputeDistance <<< blockCount, threadCount>>>(d_ref,
        ref_nb, dim, d_dist);
}

```

```

for (i = 0; i < k; ++i)
{
    dist_min_reduce(d_dist + i, ref_nb - i);
}

cuParallelSqrt <<< k, 1>>>(d_dist, k);
checkCudaErrors(cudaMemcpy(h_dist, d_dist, k * sizeof(
    Distance), cudaMemcpyDeviceToHost));

for (i = 0; i < k; ++i)
{
    dist[i] = h_dist[i].value;
    ind[i] = h_dist[i].index;
}

checkCudaErrors(cudaFree(d_ref));
checkCudaErrors(cudaFree(d_dist));
}

#ifndef REDUCTION_MOD
#define REDUCTION_MOD
#include <data_types.h>

void dist_min_reduce(Distance *dist_dev, int n);

#endif

#include "reduction-mod.cuh"
#include "cuda.h"
#include "stdio.h"
#include "helper_cuda.h"

#define CUDART_INF_F __int_as_float(0x7f800000)
#define THREADS_PER_BLOCK 512U
#define MAX_BLOCK_DIM_SIZE 65535U

bool isPow2(int x)
{
    return ((x & (x - 1)) == 0);
}

__device__ void cuMinR(Distance &distA, Distance &distB,
    int &min_index, int index, int dir)
{
    if ((distA.value >= distB.value) == dir)

```

```

    {
        distA = distB;
        min_index = index;
    }
}

int nextPow2(int x)
{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

void getNumBlocksAndThreads(int n, int maxBlocks, int
    maxThreads, int &blocks, int &threads)
{
    threads = (n < maxThreads * 2) ? nextPow2((n + 1) / 2)
        : maxThreads;
    blocks = (n + (threads * 2 - 1)) / (threads * 2);
    blocks = min(maxBlocks, blocks);
}

template <int blockSize, bool nIsPow2>
__global__ void cuReduce(Distance *g_dist, int n)
{
    __shared__ Distance s_dist[blockSize];
    __shared__ int s_ind[blockSize];
    int dir = 1;

    Distance min_dist = {1, CUDART_INF_F};
    int min_index = 0;

    int tid = threadIdx.x;
    int i = blockIdx.x * blockSize * 2 + threadIdx.x;
    int gridSize = blockSize * 2 * gridDim.x;

    while (i < n)
    {
        cuMinR(min_dist, g_dist[i], min_index, i, dir);
        if (nIsPow2 || i + blockSize < n)

```



```

    {
        cuMinR(min_dist, g_dist[i + blockSize],
              min_index, i + blockSize, dir);
    }
    i += gridSize;
}

s_dist[tid] = min_dist;
s_ind[tid] = min_index;

__syncthreads();

if (blockSize >= 512)
{
    if (tid < 256)
    {
        cuMinR(min_dist, s_dist[tid + 256], min_index,
              s_ind[tid + 256], dir);
        s_dist[tid] = min_dist;
        s_ind[tid] = min_index;
    }
    __syncthreads();
}

if (blockSize >= 256)
{
    if (tid < 128)
    {
        cuMinR(min_dist, s_dist[tid + 128], min_index,
              s_ind[tid + 128], dir);
        s_ind[tid] = min_index;
        s_dist[tid] = min_dist;
    }
    __syncthreads();
}

if (blockSize >= 128)
{
    if (tid < 64)
    {
        cuMinR(min_dist, s_dist[tid + 64], min_index,
              s_ind[tid + 64], dir);
        s_ind[tid] = min_index;
        s_dist[tid] = min_dist;
    }
}

```

```

    --syncthreads();
}

if (tid < 32)
{

    volatile int *v_ind = s_ind;
    volatile Distance *v_dist = s_dist;

    if (blockSize >= 64)
    {
        if ((min_dist.value >= v_dist[tid + 32].value)
            == dir)
        {
            min_dist = v_dist[tid] = v_dist[tid + 32];
            min_index = v_ind[tid] = v_ind[tid + 32];
        }
    }

    if (blockSize >= 32)
    {
        if ((min_dist.value >= v_dist[tid + 16].value)
            == dir)
        {
            min_dist = v_dist[tid] = v_dist[tid + 16];
            min_index = v_ind[tid] = v_ind[tid + 16];
        }
    }

    if (blockSize >= 16)
    {
        if ((min_dist.value >= v_dist[tid + 8].value)
            == dir)
        {
            min_dist = v_dist[tid] = v_dist[tid + 8];
            min_index = v_ind[tid] = v_ind[tid + 8];
        }
    }

    if (blockSize >= 8)
    {
        if ((min_dist.value >= v_dist[tid + 4].value)
            == dir)
        {
            min_dist = v_dist[tid] = v_dist[tid + 4];

```

```

        min_index = v_ind[tid] = v_ind[tid + 4];
    }
}

if (blockSize >= 4)
{
    if ((min_dist.value >= v_dist[tid + 2].value)
        == dir)
    {
        min_dist = v_dist[tid] = v_dist[tid + 2];
        min_index = v_ind[tid] = v_ind[tid + 2];
    }
}

if (blockSize >= 2)
{
    if ((min_dist.value >= v_dist[tid + 1].value)
        == dir)
    {
        min_dist = v_dist[tid] = v_dist[tid + 1];
        min_index = v_ind[tid] = v_ind[tid + 1];
    }
}
}

if (tid == 0)
{
    i = blockIdx.x;
    min_dist = g_dist[i];
    g_dist[i] = g_dist[s_ind[tid]];
    g_dist[s_ind[tid]] = min_dist;
}
}

```

```

void reduce(int size, int threads, int blocks, Distance *
g_dist)
{
    dim3 dimBlock(threads, 1, 1);
    dim3 dimGrid(blocks, 1, 1);

    int smemSize = (threads <= 32) ? 2 * threads * (sizeof(
        Distance) + sizeof(int)) : threads * (sizeof(
        Distance) + sizeof(int));
    if (isPow2(size))

```

```

{
    switch (threads)
    {
    case 512:
        cuReduce< 512, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 256:
        cuReduce< 256, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 128:
        cuReduce< 128, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 64:
        cuReduce< 64, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 32:
        cuReduce< 32, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 16:
        cuReduce< 16, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 8:
        cuReduce< 8, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 4:
        cuReduce< 4, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 2:
        cuReduce< 2, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 1:
        cuReduce< 1, true> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    }
}
else
{
    switch (threads)
    {
    case 512:
        cuReduce< 512, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 256:
        cuReduce< 256, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;

```

```

    case 128:
        cuReduce< 128, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 64:
        cuReduce< 64, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 32:
        cuReduce< 32, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 16:
        cuReduce< 16, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 8:
        cuReduce< 8, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 4:
        cuReduce< 4, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 2:
        cuReduce< 2, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
    case 1:
        cuReduce< 1, false> <<< dimGrid, dimBlock,
            smemSize >>>(g_dist, size); break;
}
}
}

```

```

void dist_min_reduce(Distance *g_dist, int n)
{
    int numBlocks = 0;
    int numThreads = 0;
    getNumBlocksAndThreads(n, MAX_BLOCK_DIM_SIZE,
        THREADS_PER_BLOCK, numBlocks, numThreads);

    reduce(n, numThreads, numBlocks, g_dist);
    n = numBlocks;
    while (n > 1)
    {
        getNumBlocksAndThreads(n, MAX_BLOCK_DIM_SIZE,
            THREADS_PER_BLOCK, numBlocks, numThreads);
        reduce(n, numThreads, numBlocks, g_dist);
        n = numBlocks;
    }
}

```

```
}
```

### C.3 k-d tree build

```
#ifndef _POINT_  
#define _POINT_  
  
// #define ADD_POINT_ID  
  
// Main point struct used in finished kd-tree  
struct Node  
{  
    float p[3];  
    int left;  
    int right;  
# ifdef ADD_POINT_ID  
    int id;  
#endif  
};  
  
// Small point used as in-data to the kd-tree building  
algorithm.  
struct Point  
{  
    float p[3];  
# ifdef ADD_POINT_ID  
    int id;  
#endif  
};  
#endif // _DATA_TYPES_  
  
#ifndef _STACK_  
#define _STACK_  
  
// Collection of structs for internal stack use  
  
// Used to handle the points when in the kd search stack.  
struct SPoint  
{  
    int index;  
    int dim;  
    float dx;  
    int other;  
};
```

```

// Used to handle the points when they are potential k
// nearest points.
struct KPoint
{
    int index;
    float dist;
};

#endif // _DATA_TYPES_

```

### C.3.1 Radix select

```

#ifndef _RADIX_SELECT_
#define _RADIX_SELECT_
#include <point.h>
#include <stack.h>

#define THREADS_PER_BLOCK_RADIX 512U
#define MAX_BLOCK_DIM_SIZE_RADIX 65535U
#define MAX_SHARED_MEM 49152U

void radixSelectAndPartition(struct Point *points, struct
    Point *swap, int *partition, int n, int dir);

#endif

#include "sum-reduction.cuh"
#include "radix-select.cuh"
#include <stdio.h>

#include <helper_cuda.h>

#define checkCudaErrors(val)          check ( (val), #val,
    __FILE__, __LINE__ )

#define debug 0
#define FILE (strchr(__FILE__, '/') ? strchr(__FILE__, '/')
    + 1 : __FILE__)
#define debugf(fmt, ...) if(debug)printf("%s:%d:_ " fmt,
    FILE, __LINE__, __VA_ARGS__);

int nextPowerOf2(int x)
{
    --x;
    x |= x >> 1;

```

```

x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
return ++x;
}

__device__ void cuCalculateBlockOffsetAndLocalN(int n, int
&local_n, int &block_offset)
{
    int rest = n % gridDim.x;
    local_n = n / gridDim.x;
    block_offset = local_n * blockDim.x;

    if (rest >= blockDim.x - blockDim.x)
    {
        block_offset += rest - (blockDim.x - blockDim.x);
        local_n++;
    }
}

__device__ void cuAccumulateIndex_(int *list, int n)
{
    int i, j, temp,
        tid = threadIdx.x;

    if (tid == blockDim.x - 1)
    {
        list[-1] = 0;
    }

    for ( i = 1; i <= n; i <<= 1)
    {
        __syncthreads();
        int temp_index = tid * i * 2 + i - 1;
        if (temp_index + i < n)
        {
            temp = list[temp_index];
            for (j = 1; j <= i; ++j)
            {
                list[temp_index + j] += temp;
            }
        }
    }
}

```



```

__device__ void cuSumReduce_(int *list, int n)
{
    int tid = threadIdx.x;

    if (n >= 1024)
    {
        if (tid < 512)
        {
            list[tid] += list[tid + 512];
        }
        __syncthreads();
    }

    if (n >= 512)
    {
        if (tid < 256)
        {
            list[tid] += list[tid + 256];
        }
        __syncthreads();
    }

    if (n >= 256)
    {
        if (tid < 128)
        {
            list[tid] += list[tid + 128];
        }
        __syncthreads();
    }

    if (n >= 128)
    {
        if (tid < 64)
        {
            list[tid] += list[tid + 64];
        }
        __syncthreads();
    }

    if (tid < 32)
    {
        volatile int *smem = list;
    }
}

```

```

    if (n >= 64)
    {
        smem[tid] += smem[tid + 32];
    }

    if (n >= 32)
    {
        smem[tid] += smem[tid + 16];
    }

    if (n >= 16)
    {
        smem[tid] += smem[tid + 8];
    }

    if (n >= 8)
    {
        smem[tid] += smem[tid + 4];
    }

    if (n >= 4)
    {
        smem[tid] += smem[tid + 2];
    }

    if (n >= 2)
    {
        smem[tid] += smem[tid + 1];
    }
}
__syncthreads();
}

__global__ void cuPartitionSwap(struct Point *points,
    struct Point *swap, int n, int *partition, int last, int
    dir)
{
    __shared__ int ones[1025];
    __shared__ int zeros[1025];
    __shared__ int medians[1025];
    __shared__ float median_value;

    int big,
        less,
        mid,

```

```

    par ,
    *zero_count = ones ,
    *one_count = zeros ,
    *median_count = medians ,
    tid = threadIdx.x;

float point_difference;
struct Point point;

zero_count++;
one_count++;
median_count++;
zero_count[threadIdx.x] = 0;
one_count[threadIdx.x] = 0;
median_count[threadIdx.x] = 0;

while (tid < n)
{
    if (partition[tid] == last)
    {
        median_value = points[tid].p[dir];
    }
    tid += blockDim.x;
}

tid = threadIdx.x;
__syncthreads();

while (tid < n)
{
    swap[tid] = point = points[tid];
    point_difference = (point.p[dir] - median_value);
    par = partition[tid] ;
    if (point_difference < 0)
    {
        par = -1;
        zero_count[threadIdx.x]++;
    }
    else if (point_difference > 0)
    {
        par = 1;
        one_count[threadIdx.x]++;
    }
    else
    {

```

```

        par = 0;
        median_count[threadIdx.x]++;
    }
    partition[tid] = par;
    tid += blockDim.x;
}

__syncthreads();
cuAccumulateIndex_(zero_count, blockDim.x);
cuAccumulateIndex_(one_count, blockDim.x);
cuAccumulateIndex_(median_count, blockDim.x);
__syncthreads();

tid = threadIdx.x;
one_count--;
zero_count--;
median_count--;
mid = zero_count[blockDim.x] + median_count[threadIdx.x];
less = zero_count[threadIdx.x];
big = one_count[threadIdx.x];

while (tid < n)
{
    if (partition[tid] < 0)
    {
        points[less] = swap[tid];
        less++;
    }
    else if (partition[tid] > 0)
    {
        points[n - big - 1] = swap[tid];
        big++;
    }
    else
    {
        points[mid] = swap[tid];
        mid++;
    }
    tid += blockDim.x;
}
}

__global__ void cuPartitionStep(struct Point *data, int n,
    int *partition, int *zeros_count_block, int last, int

```

```

bit, int dir)
{
    __shared__ int zero_count[THREADS_PER_BLOCK_RADIX];

    int tid = threadIdx.x,
        is_one,
        block_offset,
        local_n,
        radix = (1 << 31 - bit);

    cuCalculateBlockOffsetAndLocalN(n, local_n,
        block_offset);

    zero_count[threadIdx.x] = 0;

    data += block_offset;
    partition += block_offset;

    while (tid < local_n)
    {
        if (partition[tid] == last)
        {
            is_one = partition[tid] = (bool)((*(int *) & (
                data[tid].p[dir]))&radix);
            zero_count[threadIdx.x] += !is_one;
        }
        else
        {
            partition[tid] = 2;
        }
        tid += blockDim.x;
    }
    __syncthreads();

    cuSumReduce_(zero_count, blockDim.x);

    if (threadIdx.x == 0)
    {
        zeros_count_block[blockIdx.x] = zero_count[0];
    }
}

__global__ void fillArray(int *array, int value, int n)
{

```

```

int local_n ,
      block_offset ,
      tid = threadIdx.x;

cuCalculateBlockOffsetAndLocalN(n, local_n ,
    block_offset);

array += block_offset;
while (tid < local_n)
{
    array[tid] = value;
    tid += blockDim.x;
}
}

void getThreadAndBlockCountPartition(int n, int &blocks ,
int &threads)
{
    threads = min(nextPowerOf2(n), THREADS.PER_BLOCK_RADIX)
        ;
    blocks = n / threads / 2;
    blocks = max(1, nextPowerOf2(blocks));
    blocks = min(MAX_BLOCK_DIM_SIZE_RADIX, blocks);
    debugf("blocks = %d, threads = %d, n = %d\n", blocks ,
        threads , n);
}

void radixSelectAndPartition(struct Point *points , struct
    Point *swap, int *partition , int n, int dir)
{
    int numBlocks ,
        numThreads ,
        cut ,
        l = 0 ,
        u = n ,
        m_u = (int)ceil((float)n / 2) ,
        bit = 0 ,
        last = 2 ,
        *h_zeros_count_block ,
        *d_zeros_count_block;

    getThreadAndBlockCountPartition(n, numBlocks ,
        numThreads);

    fillArray <<< numBlocks , numThreads>>>(partition , 2, n)

```

```

;

h_zeros_count_block = (int *) malloc(numBlocks * sizeof
(int));
checkCudaErrors(
    cudaMalloc((void **) &d_zeros_count_block ,
        numBlocks * sizeof(int)));

do
{
    cuPartitionStep <<< numBlocks, numThreads>>>(points
        , n, partition , d_zeros_count_block , last , bit
        ++, dir);

    sum_reduce(d_zeros_count_block , numBlocks);
    cudaMemcpy(h_zeros_count_block , d_zeros_count_block
        , sizeof(int) , cudaMemcpyDeviceToHost);

    cut = h_zeros_count_block[0];

    if ((u - cut) >= (m.u))
    {
        u = u - cut;
        last = 1;
    }
    else
    {
        l = u - cut;
        last = 0;
    }
}
while (((u - l) > 1) && (bit <= 32));

cuPartitionSwap <<< 1, min(nextPowerOf2(n) ,
    THREADS_PER_BLOCK_RADIX) >>> (points , swap , n ,
    partition , last , dir);

checkCudaErrors(
    cudaFree(d_zeros_count_block));
}

```

### C.3.2 Parallel quick select

```

#ifndef _QUICK_SELECT_
#define _QUICK_SELECT_
#include <point.h>

```

```

#include <stack.h>

#define MAX_SHARED_MEM 49152U
#define THREADS_PER_BLOCK_QUICK 64U
#define MAX_BLOCK_DIM_SIZE 65535U

void quickSelectAndPartition(struct Point *d_points , int *
    d_steps , int n , int p , int dir);
void quickSelectShared(struct Point *points , int *steps ,
    int p , int dir , int size , int numBlocks , int numThreads)
    ;
void getThreadAndBlockCountForQuickSelect(int n , int p , int
    &blocks , int &threads);
void cpuQuickSelect(struct Point *points , int n , int dir);

#endif

#include "quick-select.cuh"
#include <stdio.h>

__device__ void cuPointSwap(struct Point *p , int a , int b)
{
    struct Point temp = p[a];
    p[a] = p[b] , p[b] = temp;
}

__device__ void cuCalculateBlockOffsetAndNoOfLists(int n ,
int &n_per_block , int &block_offset)
{
    int rest = n % gridDim.x;
    n_per_block = n / gridDim.x;
    block_offset = n_per_block * blockIdx.x;

    if (rest >= gridDim.x - blockIdx.x)
    {
        block_offset += rest - (gridDim.x - blockIdx.x);
        n_per_block++;
    }
}

__device__ void cuCopyPoints(struct Point *s_points , struct
    Point *l_points , int n)
{
    int i;
    for (i = 0; i < n; ++i)
    {

```



```

        s_points[i] = l_points[i];
    }
}

template <int max_step, bool in_shared> __global__
void cuQuickSelect(struct Point *points, int *steps, int p,
    int dir)
{
    __shared__ struct Point ss_points[max_step *
        THREADS_PER_BLOCK_QUICK];

    struct Point *s_points = ss_points, *l_points;

    float pivot;

    int pos,
        i,
        left,
        right,
        m,
        step_num,
        n,
        list_in_block,
        tid = threadIdx.x,
        block_offset;

    cuCalculateBlockOffsetAndNoOfLists(p, list_in_block,
        block_offset);

    steps += block_offset * 2;
    s_points += (tid * max_step);

    while (tid < list_in_block)
    {
        step_num = tid * 2;
        l_points = points + steps[step_num];
        n = steps[step_num + 1] - steps[step_num];
        m = n >> 1; // same as n/2;
        left = 0;
        right = n - 1;

        if (in_shared)
        {
            cuCopyPoints(s_points, l_points, n);
        }
    }
}

```

```

    else
    {
        s_points = l_points;
    }
    while (left < right)
    {
        pivot = s_points[m].p[dir];
        cuPointSwap(s_points, m, right);
        for (i = pos = left; i < right; i++)
        {
            if (s_points[i].p[dir] < pivot)
            {
                cuPointSwap(s_points, i, pos);
                pos++;
            }
        }
        cuPointSwap(s_points, right, pos);
        if (pos == m) break;
        if (pos < m) left = pos + 1;
        else right = pos - 1;
    }
    if (in_shared)
    {
        cuCopyPoints(l_points, s_points, n);
    }
    tid += blockDim.x;
}
}

void quickSelectAndPartition(struct Point *d_points, int *
d_steps, int step, int p, int dir)
{
    int numBlocks, numThreads;
    getThreadAndBlockCountForQuickSelect(step, p, numBlocks
, numThreads);
    if (step > 16)
    {
        cuQuickSelect<1, false><<< numBlocks, numThreads
>>>(d_points, d_steps, p, dir);
    }
    else if (step > 8 && step * sizeof(Point) * numThreads
< MAX_SHARED_MEM)
    {
        cuQuickSelect<16, true><<< numBlocks, numThreads
>>>(d_points, d_steps, p, dir);
    }
}

```

```

    }
    else if (step > 4 && step * sizeof(Point) * numThreads
             < MAX_SHARED_MEM)
    {
        cuQuickSelect<8, true><<< numBlocks, numThreads
            >>>(d_points, d_steps, p, dir);
    }
    else if (step * sizeof(Point) * numThreads <
             MAX_SHARED_MEM)
    {
        cuQuickSelect<4, true><<< numBlocks, numThreads
            >>>(d_points, d_steps, p, dir);
    }
    else
    {
        cuQuickSelect<1, false><<< numBlocks, numThreads
            >>>(d_points, d_steps, p, dir);
    }
}

void getThreadAndBlockCountForQuickSelect(int n, int p, int
&blocks, int &threads)
{
    threads = min(THREADS_PER_BLOCK_QUICK, p);
    blocks = p / threads;
    blocks = min(MAX_BLOCK_DIM_SIZE, blocks);
    blocks = max(1, blocks);
    // printf("block = %d, threads = %d, n = %d, p = %d\n",
        blocks, threads, n, p );
}

void cpuQuickSelect(struct Point *points, int n, int dir)
{
#define SWAP(a, b) { tmp = points[a]; points[a] = points[b
]; points[b] = tmp; }

    struct Point tmp;

    float pivot;

    int pos,
        left,
        right,
        i,

```

```

        m;

m = n >> 1;    // same as n/2;
left = 0;
right = n - 1;

while (left < right)
{
    pivot = points[m].p[dir];
    SWAP(m, right);
    for (i = pos = left; i < right; i++)
    {
        if (points[i].p[dir] < pivot)
        {
            SWAP(i, pos);
            pos++;
        }
    }
    SWAP(right, pos);
    if (pos == m) break;
    if (pos < m) left = pos + 1;
    else right = pos - 1;
}
}

```

### C.3.3 Multiple radix select

```

#ifndef MULTI_RADIX_SELECT_
#define MULTI_RADIX_SELECT_
#include <point.h>
#include <stack.h>

#define THREADS_PER_BLOCK_MULTI_RADIX 512U
#define MAX_BLOCK_DIM_SIZE_MULTI_RADIX 65535U

void getThreadAndBlockCountMulRadix(int n, int p, int &
    blocks, int &threads);
void multiRadixSelectAndPartition(struct Point *data,
    struct Point *data_copy, int *partition, int *steps, int
    n, int p, int dir);

__global__
void cuBalanceBranch(struct Point *points, struct Point *
    swap, int *partition, int *steps, int p, int dir);

```

```

#endif

#include "multiple-radix-select.cuh"
#include <stdio.h>

#include <helper_cuda.h>

#define checkCudaErrors(val)          check ( (val), #val,
    __FILE__, __LINE__ )

#define debug 0
#define FILE (strchr(__FILE__, '/') ? strchr(__FILE__, '/')
    + 1 : __FILE__)
#define debugf(fmt, ...) if(debug) printf("%s:%d:_" fmt,
    FILE, __LINE__, __VA_ARGS__);

int nextPowTwo(int x)
{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

bool isPowTwo(int x)
{
    return ((x & (x - 1)) == 0);
}

int prevPowTwo(int n)
{
    if (isPowTwo(n))
    {
        return n;
    }
    n = nextPowTwo(n);
    return n >>= 1;
}

__device__ void cuAccumulateIndex(int *list, int n)
{
    int i, j, temp,

```

```

        tid = threadIdx.x;
if (tid == blockDim.x - 1)
    {
        list[-1] = 0;
    }
for ( i = 1; i <= n; i <<= 1)
    {
        __syncthreads();
        int temp_index = tid * i * 2 + i - 1;
        if (temp_index + i < n)
        {
            temp = list[temp_index];
            for (j = 1; j <= i; ++j)
            {
                list[temp_index + j] += temp;
            }
        }
    }
}

__device__ void fillArray_(int *partition , int last , int n
)
{
    int tid = threadIdx.x;
    while (tid < n)
    {
        partition[tid] = last;
        tid += blockDim.x;
    }
}

__device__ int cuSumReduce(int *list , int n)
{
    int tid = threadIdx.x;

    if (n >= 1024)
    {
        if (tid < 512)
        {
            list[tid] += list[tid + 512];
        }
        __syncthreads();
    }
    if (n >= 512)
    {

```

```

    if (tid < 256)
    {
        list[tid] += list[tid + 256];
    }
    --syncthread();
}

if (n >= 256)
{
    if (tid < 128)
    {
        list[tid] += list[tid + 128];
    }
    --syncthread();
}

if (n >= 128)
{
    if (tid < 64)
    {
        list[tid] += list[tid + 64];
    }
    --syncthread();
}

if (tid < 32)
{
    volatile int *smem = list;

    if (n >= 64)
    {
        smem[tid] += smem[tid + 32];
    }

    if (n >= 32)
    {
        smem[tid] += smem[tid + 16];
    }

    if (n >= 16)
    {
        smem[tid] += smem[tid + 8];
    }

    if (n >= 8)

```

```

    {
        smem[tid] += smem[tid + 4];
    }

    if (n >= 4)
    {
        smem[tid] += smem[tid + 2];
    }

    if (n >= 2)
    {
        smem[tid] += smem[tid + 1];
    }
}
__syncthreads();
return list[0];
}

__device__ void cuPartitionSwap(struct Point *data, struct
    Point *swap, int n, int *partition, int *zero_count, int
    *one_count, int *median_count, float median_value, int
    dir)
{
    int tid = threadIdx.x,
        big,
        mid,
        par,
        less;

    float point_difference;
    struct Point point;

    zero_count++;
    one_count++;
    median_count++;
    zero_count[threadIdx.x] = 0;
    one_count[threadIdx.x] = 0;
    median_count[threadIdx.x] = 0;

    while (tid < n)
    {
        point = data[tid];
        swap[tid] = point;
        point_difference = (point.p[dir] - median_value);
        if (point_difference < 0)

```



```

    {
        par = -1;
        zero_count[threadIdx.x]++;
    }
    else if (point_difference > 0)
    {
        par = 1;
        one_count[threadIdx.x]++;
    }
    else
    {
        par = 0;
        median_count[threadIdx.x]++;
    }
    partition[tid] = par;
    tid += blockDim.x;
}

__syncthreads();
cuAccumulateIndex(zero_count, blockDim.x);
cuAccumulateIndex(one_count, blockDim.x);
cuAccumulateIndex(median_count, blockDim.x);
__syncthreads();

tid = threadIdx.x;
one_count--;
zero_count--;
median_count--;
less = zero_count[threadIdx.x];
big = one_count[threadIdx.x];
mid = zero_count[blockDim.x] + median_count[threadIdx.x];

while (tid < n)
{
    if (partition[tid] < 0)
    {
        data[less] = swap[tid];
        less++;
    }
    else if (partition[tid] > 0)
    {
        data[n - big - 1] = swap[tid];
        big++;
    }
}

```

```

        else
        {
            data[mid] = swap[tid];
            mid++;
        }
        tid += blockDim.x;
    }
}

__device__ int cuPartition(struct Point *data, int n, int *
partition, int *zero_count, int last, int bit, int dir)
{
    int is_one,
        tid = threadIdx.x,
        radix = (1 << 31 - bit);

    zero_count[threadIdx.x] = 0;

    while (tid < n)
    {
        if (partition[tid] == last)
        {
            is_one = partition[tid] = (bool)((*(int *) & (
                data[tid].p[dir]))&radix);
            zero_count[threadIdx.x] += !is_one;
        }
        else
        {
            partition[tid] = 2;
        }
        tid += blockDim.x;
    }

    __syncthreads();
    return cuSumReduce(zero_count, blockDim.x);
}

__device__ void cuRadixSelect(struct Point *data, struct
Point *data_copy, int n, int *partition, int dir)
{
    __shared__ int one_count[1025];
    __shared__ int zeros_count[1025];
    __shared__ int medians_count[1025];
    __shared__ float median_value;

```

```

int l = 0,
      bit = 0,
      last = 2,
      u = n,
      m_u = ceil((float)n / 2),
      tid = threadIdx.x;

fillArray_(partition, last, n);

do
{
    __syncthreads();
    int cut = cuPartition(data, n, partition,
        zeros_count, last, bit++, dir);

    if ((u - cut) >= (m_u))
    {
        u = u - cut;
        last = 1;
    }
    else
    {
        l = u - cut;
        last = 0;
    }
}
while (((u - l) > 1) && (bit < 32));

tid = threadIdx.x;

while (tid < n)
{
    if (partition[tid] == last)
    {
        median_value = data[tid].p[dir];
    }
    tid += blockDim.x;
}
__syncthreads();

cuPartitionSwap(data, data_copy, n, partition,
    one_count, zeros_count, medians_count, median_value,
    dir);
}

```

```

__global__
void cuBalanceBranch(struct Point *points, struct Point *
    swap, int *partition, int *steps, int p, int dir)
{
    int blockoffset,
        n,
        bid = blockIdx.x;

    while (bid < p)
    {
        blockoffset = steps[bid * 2];
        n = steps[bid * 2 + 1] - blockoffset;
        cuRadixSelect(points + blockoffset, swap +
            blockoffset, n, partition + blockoffset, dir);
        bid += gridDim.x;
    }
}

void getThreadAndBlockCountMulRadix(int n, int p, int &
    blocks, int &threads)
{
    threads = n - 1;
    threads = prevPowTwo(threads / 4);
    blocks = min(MAX_BLOCK_DIM_SIZE_MULTIRADIX, p);
    blocks = max(1, blocks);
    threads = min(THREADS_PER_BLOCK_MULTIRADIX, threads);
    threads = max(128, threads);
    debugf("N=%d, p=%d, blocks=%d, threads=%d\n", n,
        p, blocks, threads);
}

void multiRadixSelectAndPartition(struct Point *d_data,
    struct Point *d_data_copy, int *d_partition, int *
    d_steps, int n, int p, int dir)
{
    int numBlocks, numThreads;
    getThreadAndBlockCountMulRadix(n, p, numBlocks,
        numThreads);
    cuBalanceBranch <<< numBlocks, numThreads>>>(d_data,
        d_data_copy, d_partition, d_steps, p, dir);
}

//For testing - One cannot import a __device__ kernel
__global__ void cuRadixSelectGlobal(struct Point *data,

```

```

    struct Point *data_copy , int n, int *partition , int dir)
{
    cuRadixSelect(data , data_copy , n, partition , dir);
}

```

### C.3.4 Parallel k-d tree build

```

#ifndef _KD_TREE_NAIVE_
#define _KD_TREE_NAIVE_
#include <point.h>
#include <stack.h>

void buildKdTree(struct Point *points , int n, struct Node *
    points_out);
void getThreadAndBlockCountMulRadix(int n, int p, int &
    blocks , int &threads);
void getThreadAndBlockCountForQuickSelect(int n, int p, int
    &blocks , int &threads);
int store_locations(struct Node *tree , int lower , int upper
    , int n);

__global__
void cuBalanceBranch(struct Point *points , struct Point *
    swap, int *partition , int n, int p, int dir);

__global__
void cuQuickSelectGlobal(struct Point *points , int n, int p
    , int dir);

template <int maxStep> __global__
void cuQuickSelectShared(struct Point *points , int n, int p
    , int dir);

#endif

#include "kd-tree-build.cuh"
#include "multiple-radix-select.cuh"
#include "quick-select.cuh"
#include "radix-select.cuh"

#include "stdio.h"
#include "point.h"

#include "helper_cuda.h"

int nextPowerOf2_(int x)

```

```

{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

void UpDim(int &dim)
{
    dim = (dim + 1) % 3;
}

void getThreadAndBlockCountForBuild(int n, int &blocks, int
&threads)
{
    threads = min(nextPowerOf2_(n), 512);
    blocks = n / threads;
    blocks = max(1, blocks);
    blocks = min(MAX_BLOCK_DIM_SIZE, blocks);
    // printf("block = %d, threads = %d, n = %d\n", blocks,
        threads, n);
}

__device__ void cuCalculateBlockOffsetAndNoOfLists_(int n,
int &n_per_block, int &block_offset)
{
    int rest = n % gridDim.x;

    n_per_block = n / gridDim.x;
    block_offset = n_per_block * blockIdx.x;

    if (rest >= gridDim.x - blockIdx.x)
    {
        block_offset += rest - (gridDim.x - blockIdx.x);
        n_per_block++;
    }
}

__device__ void cuPointSwapCondition(struct Point *p, int a
, int b, int dim)
{
    struct Point temp_a = p[a], temp_b = p[b];

```

```

    if (temp_a.p[dim] > temp_b.p[dim] )
    {
        p[a] = temp_b, p[b] = temp_a;
    }
}

--global-- void balanceLeafs(struct Point *points , int *
steps , int p, int dim)
{
    struct Point    *l_points;

    int list_in_block ,
        block_offset ,
        tid = threadIdx.x,
        step_num ,
        n;

    cuCalculateBlockOffsetAndNoOfLists_(p, list_in_block ,
        block_offset);

    steps += block_offset * 2;

    while ( tid < list_in_block)
    {
        step_num = tid * 2;
        l_points = points + steps[step_num];
        n = steps[step_num + 1] - steps[step_num];
        if (n == 2)
        {
            cuPointSwapCondition(l_points , 0, 1, dim);
        }
        else if (n == 3)
        {
            cuPointSwapCondition(l_points , 0, 1, dim);
            cuPointSwapCondition(l_points , 1, 2, dim);
            cuPointSwapCondition(l_points , 0, 1, dim);
        }
        tid += blockDim.x;
    }
}

int store_locations(struct Node *tree , int lower , int upper
, int n)
{
    int r;

```

```

    if (lower >= upper)
    {
        return -1;
    }

    r = (int) ((upper - lower) / 2) + lower;

    tree[r].left = store_locations(tree, lower, r, n);
    tree[r].right = store_locations(tree, r + 1, upper, n);

    return r;
}

__device__ __host__
void pointConvert(struct Node &p1, struct Point &p2)
{
    p1.p[0] = p2.p[0], p1.p[1] = p2.p[1], p1.p[2] = p2.p
        [2];
#ifdef ADD_POINT_ID
    p1.id = p2.id;
#endif
}

__global__
void convertPoints(struct Point *points_small, int n,
    struct Node *points)
{
    int local_n,
        block_offset,
        tid = threadIdx.x;

    cuCalculateBlockOffsetAndNoOfLists_(n, local_n,
        block_offset);

    points += block_offset;
    points_small += block_offset;

    while (tid < local_n)
    {
        pointConvert(points[tid], points_small[tid]);
        tid += blockDim.x;
    }
}

```



```

void nextStep(int *steps_new , int *steps_old , int n)
{
    int i, midpoint, from, to;
    for (i = 0; i < n / 2; ++i)
    {
        from = steps_old[i * 2];
        to = steps_old[i * 2 + 1];
        midpoint = (to - from) / 2 + from;

        steps_new[i * 4] = from;
        steps_new[i * 4 + 1] = midpoint;
        steps_new[i * 4 + 2] = midpoint + 1;
        steps_new[i * 4 + 3] = to;
    }
}

void swap_pointer(int **a, int **b)
{
    int *swap;
    swap = *a, *a = *b, *b = swap;
}

void singleRadixSelectAndPartition(struct Point *d_points ,
    struct Point *d_swap, int *d_partition , int *h_steps ,
    int p, int dir)
{
    int nn, offset , j;
    for (j = 0; j < p; j ++ )
    {
        offset = h_steps[j * 2];
        nn = h_steps[j * 2 + 1] - offset;
        if (nn > 1)
        {
            radixSelectAndPartition(d_points + offset ,
                d_swap + offset , d_partition + offset , nn,
                dir);
        }
    }
}

size_t getFreeBytesOnGpu_()
{
    size_t free_byte , total_byte ;
    cudaError_t cuda_status = cudaMemGetInfo( &free_byte , &
        total_byte ) ;
}

```

```

    return free_byte;
}

size_t getNeededBytesForBuildingKdTree(int n)
{
    int number_of_leafs = (n + 1) / 2;
    return (number_of_leafs * 2 * sizeof(int)) + (2 * n *
        sizeof(int)) + (2 * n * sizeof(Point));
}

void cuBuildKdTree(struct Point *h_points, int n, int dim,
struct Node *tree)
{
    struct Point *d_points, *d_swap;
    struct Node *d_tree;
    int *d_partition,
        block_num, thread_num,
        *d_steps, *h_steps_old, *h_steps_new,
        step,
        i = 0,
        p = 1,
        number_of_leafs = (n + 1) / 2,
        h = (int)ceil(log2((float)n + 1));

    h_steps_new = (int *)malloc(number_of_leafs * 2 *
        sizeof(int));
    h_steps_old = (int *)malloc(number_of_leafs * 2 *
        sizeof(int));

    h_steps_new[0] = 0;
    h_steps_old[0] = 0;
    h_steps_old[1] = n;
    h_steps_new[1] = n;

    checkCudaErrors(
        cudaMalloc(&d_steps, number_of_leafs * 2 * sizeof(
            int)));
    checkCudaErrors(
        cudaMalloc(&d_partition, n * sizeof(int)));
    checkCudaErrors(
        cudaMalloc(&d_points, n * sizeof(Point)));
    checkCudaErrors(
        cudaMalloc(&d_swap, n * sizeof(Point)));

    checkCudaErrors(

```

```

        cudaMemcpy(d_points , h_points , n * sizeof(Point) ,
                  cudaMemcpyHostToDevice);

radixSelectAndPartition(d_points , d_swap , d_partition ,
                        n , dim);

UpDim(dim);
i++;
while (i < (h - 1) )
{
    nextStep(h_steps_new , h_steps_old , p <<= 1);
    step = h_steps_new[1] - h_steps_new[0];
    checkCudaErrors(
        cudaMemcpy(d_steps , h_steps_new , p * 2 * sizeof
                  (int) , cudaMemcpyHostToDevice));

    if (step >= 9000000)
    {
        singleRadixSelectAndPartition(d_points , d_swap ,
                                       d_partition , h_steps_new , p , dim);
    }
    else if (step > 3000)
    {
        multiRadixSelectAndPartition(d_points , d_swap ,
                                      d_partition , d_steps , step , p , dim);
    }
    else if (step > 3)
    {
        quickSelectAndPartition(d_points , d_steps , step
                                , p , dim);
    }
    else
    {
        getThreadAndBlockCountForBuild(n , block_num ,
                                       thread_num);
        balanceLeafs <<< block_num , thread_num >>> (
            d_points , d_steps , p , dim);
    }
    swap_pointer(&h_steps_new , &h_steps_old);
    i++;
    UpDim(dim);
}

checkCudaErrors(cudaFree(d_swap));
checkCudaErrors(cudaFree(d_partition));

```

```

    checkCudaErrors(cudaFree(d_steps));
    free(h_steps_new);
    free(h_steps_old);

    checkCudaErrors(cudaMalloc(&d_tree, n * sizeof(Node)));

    getThreadAndBlockCountForBuild(n, block_num, thread_num
    );
    convertPoints <<< block_num, thread_num >>> (d_points,
    n, d_tree);

    checkCudaErrors(cudaMemcpy(tree, d_tree, n * sizeof(
    Node), cudaMemcpyDeviceToHost));

    checkCudaErrors(cudaFree(d_points));
    checkCudaErrors(cudaFree(d_tree));
}

void buildKdTreeStep(struct Point *h_points, int n, int dim
, struct Node *tree)
{
    if (n <= 0) return;

    size_t free_bytes, needed_bytes;
    int m = n >> 1;

    free_bytes = getFreeBytesOnGpu();
    needed_bytes = getNeededBytesForBuildingKdTree(n);

    if (free_bytes > needed_bytes)
    {
        cuBuildKdTree(h_points, n, dim, tree);
    }
    else
    {
        cpuQuickSelect(h_points, n, dim);
        pointConvert(tree[m], h_points[m]);

        UpDim(dim);

        buildKdTreeStep(h_points, m, dim, tree);
        buildKdTreeStep(h_points + m + 1, n - m - 1, dim,
            tree + m + 1);
    }
}
}

```

```

void buildKdTree(struct Point *h_points , int n, struct Node
    *tree)
{
    int dim = 0;
    buildKdTreeStep(h_points , n, dim, tree);
    store_locations(tree , 0, n, n);
}

```

## C.4 k-d tree search

### C.4.1 CUDA k-d tree search

```

#ifndef _CU_KD_SEARCH_
#define _CU_KD_SEARCH_
#include <point.h>
#include <stack.h>

#define THREADS_PER_BLOCK_SEARCH 64U
#define MAX_BLOCK_DIM_SIZE 8192U
#define MAX_SHARED_MEM 49152U
#define MIN_NUM_QUERY_POINTS 100U

__device__ __host__ void cuInitStack(struct SPoint **stack)
    ;
__device__ __host__ bool cuIsEmpty(struct SPoint *stack);
__device__ __host__ void cuPush(struct SPoint **stack ,
    struct SPoint value);
__device__ __host__ struct SPoint cuPop(struct SPoint **
    stack);
__device__ __host__ struct SPoint cuPeek(struct SPoint *
    stack);
__device__ __host__ void cuInitKStack(KPoint **k_stack , int
    n);
__device__ __host__ void cuInsert(struct KPoint *k_stack ,
    struct KPoint k_point , int n);
__device__ __host__ struct KPoint cuLook(struct KPoint *
    k_stack);

__device__ __host__ int fastIntegerLog2(int x);

__device__ __host__ float cuDist(struct Point qp, struct
    Node point);

```

```

__device__ __host__ void cuUpDim(int &dim);

__device__ __host__ void cuKNN(struct Point qp, struct Node
    *tree, int n, int k,
                                struct SPoint *stack_ptr,
                                struct KPoint *
                                k_stack_ptr);

size_t getFreeBytesOnGpu();
int getQueriesInStep(int n_qp, int k, int n);
void getThreadAndBlockCountForQueryAll(int n, int &blocks,
    int &threads);
size_t getNeededBytesInSearch(int n_qp, int k, int n, int
    thread_num, int block_num);

void cuQueryAll(struct Point *h_query_points, struct Node *
    tree, int qp_n, int tree_n, int k, int *result);

#endif

#include <stdlib.h>
#include <math.h>
#include <float.h>

#include <helper_cuda.h>
#include <cu-kd-search.cuh>
#include "kd-tree-build.cuh"

__device__ __host__
float cuDist(struct Point qp, struct Node point)
{
    float dx = qp.p[0] - point.p[0],
          dy = qp.p[1] - point.p[1],
          dz = qp.p[2] - point.p[2];

    return (dx * dx) + (dy * dy) + (dz * dz);
}

__device__ __host__
void cuInitStack(struct SPoint **stack)
{
    (*stack)[0].index = -1;
    (*stack)++;
}

__device__ __host__

```

```

bool cuIsEmpty(struct SPoint *stack)
{
    return cuPeek(stack).index == -1;
}

__device__ __host__
void cuPush(struct SPoint **stack, struct SPoint value)
{
    *((*stack)++) = value;
}

__device__ __host__
struct SPoint cuPop(struct SPoint **stack)
{
    return *--(*stack);
}

__device__ __host__
struct SPoint cuPeek(struct SPoint *stack)
{
    return *(stack - 1);
}

__device__ __host__
void cuInitKStack(struct KPoint **k_stack, int n)
{
    (*k_stack)--;
    for (int i = 1; i <= n; ++i)
    {
        (*k_stack)[i].dist = FLT_MAX;
        (*k_stack)[i].index = -1;
    }
}

__device__ __host__
void cuInsert(struct KPoint *k_stack, struct KPoint k_point
, int n)
{
    int i_child, now;
    struct KPoint child, child_tmp_2;
    for (now = 1; now * 2 <= n ; now = i_child)
    {
        i_child = now * 2;
        child = k_stack[i_child];
    }
}

```

```

        child_tmp_2 = k_stack[i_child + 1];
        if (i_child <= n && child_tmp_2.dist > child.dist )
        {
            i_child++;
            child = child_tmp_2;
        }

        if (i_child <= n && k_point.dist < child.dist)
        {
            k_stack[now] = child;
        }
        else
        {
            break;
        }
    }
    k_stack[now] = k_point;
}

```

```

__device__ __host__
struct KPoint cuLook(struct KPoint *k_stack)
{
    return k_stack[1];
}

```

```

__device__ __host__
void cuUpDim(int &dim)
{
    dim = (dim + 1) % 3;
}

```

```

__device__ __host__
void cuChildren(struct Point qp, struct Node current, float
    dx, int &target, int &other)
{
    if (dx > 0)
    {
        other = current.right;
        target = current.left;
    }
    else
    {
        other = current.left;
        target = current.right;
    }
}

```



```

    }
}

__device__ __host__
void cuKNN(struct Point qp, struct Node *tree, int n, int k
,
    struct SPoint *stack, struct KPoint *k_stack)
{
    int dim = 2, target;
    float current_dist;

    struct Node current_point;
    struct SPoint current;
    struct KPoint worst_best;

    current.index = n / 2;

    cuInitStack(&stack);
    cuInitKStack(&k_stack, k);
    worst_best = cuLook(k_stack);

    while (!cuIsEmpty(stack) || current.index != -1)
    {
        if (current.index == -1 && !cuIsEmpty(stack))
        {
            current = cuPop(&stack);
            dim = current.dim;

            current.index = (current.dx * current.dx <
                worst_best.dist) ? current.other : -1;
        }
        else
        {
            current_point = tree[current.index];

            current_dist = cuDist(qp, current_point);
            if (worst_best.dist > current_dist)
            {
                worst_best.dist = current_dist;
                worst_best.index = current.index;
                cuInsert(k_stack, worst_best, k);
                worst_best = cuLook(k_stack);
            }

            cuUpDim(dim);
        }
    }
}

```

```

        current.dim = dim;
        current.dx = current_point.p[dim] - qp.p[dim];
        cuChildren(qp, current_point, current.dx,
            target, current.other);
        cuPush(&stack, current);

        current.index = target;
    }
}

__device__ __host__
int fastIntegerLog2(int x)
{
    int y = 0;
    while (x >>= 1)
    {
        y++;
    }
    return y;
}

__device__ void cuCalculateBlockOffsetAndNoOfQueries(int n,
    int &n_per_block, int &block_offset)
{
    int rest = n % gridDim.x;
    n_per_block = n / gridDim.x;
    block_offset = n_per_block * blockIdx.x;

    if (rest >= gridDim.x - blockIdx.x)
    {
        block_offset += rest - (gridDim.x - blockIdx.x);
        n_per_block++;
    }
}

__device__ __host__ int getSStackSize(int n)
{
    return fastIntegerLog2(n) + 2;
}

size_t getSStackSizeInBytes(int n, int thread_num, int
    block_num)
{
    return block_num * thread_num * ((getSStackSize(n) *

```

```

        sizeof(SPoint));
    }

size_t getNeededBytesInSearch(int n_qp, int k, int n, int
    thread_num, int block_num)
{
    return n_qp * (k * sizeof(int) + sizeof(Point)) +
        (n_qp * k * sizeof(KPoint)) +
        (getSStackSizeInBytes(n, thread_num, block_num))
        ;
}

void populateTrivialResult(int n_qp, int k, int n_tree, int
    *result)
{
    #pragma omp parallel for
    for (int i = 0; i < n_qp; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            result[i * k + j] = j % n_tree;
        }
    }
}

template<int stack_size> __global__
void dQueryAll(struct Point *query_points, struct Node *
    tree, int n_qp, int n_tree, int k, struct KPoint *
    k_stack_ptr)
{
    SPoint stack[stack_size];

    int tid = threadIdx.x,
        block_step,
        block_offset;

    cuCalculateBlockOffsetAndNoOfQueries(n_qp, block_step,
        block_offset);

    query_points += block_offset;
    k_stack_ptr += block_offset * k;

    while (tid < block_step)
    {

```

```

        cuKNN(query_points[tid], tree, n_tree, k, stack,
              k_stack_ptr + (tid * k));
        tid += blockDim.x;
    }
}

void getThreadAndBlockCountForQueryAll(int n, int &blocks,
int &threads)
{
    threads = THREADS_PER_BLOCK_SEARCH;
    blocks = n / threads;
    blocks = min(MAX_BLOCK_DIM_SIZE, blocks);
    blocks = max(1, blocks);
    // printf("blocks = %d, threads = %d, n = %d\n", blocks,
        threads, n);
}

int getQueriesInStep(int n_qp, int k, int n)
{
    int numBlocks, numThreads;
    size_t needed_bytes_total, free_bytes;

    free_bytes = getFreeBytesOnGpu();

    getThreadAndBlockCountForQueryAll(n_qp, numThreads,
        numBlocks);

    needed_bytes_total = getNeededBytesInSearch(n_qp, k, n,
        numThreads, numBlocks);

    if (free_bytes > needed_bytes_total) return n_qp;
    if (n_qp < 50) return -1;

    return getQueriesInStep((n_qp / 2), k, n);
}

int nextPowerOf2_(int x)
{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

```

```

}

void templateQueryAll(struct Point *d_query_points , struct
Node *d_tree , int queries_in_step , int n_tree , int k ,
int stack_size , int numBlocks , int numThreads , struct
KPoint *d_k_stack)
{
    if (stack_size <= 20 )
    {
        dQueryAll<20> <<< numBlocks , numThreads>>>(
            d_query_points , d_tree , queries_in_step , n_tree ,
            k , d_k_stack);
    }
    else if (stack_size <= 25)
    {
        dQueryAll<25> <<< numBlocks , numThreads>>>(
            d_query_points , d_tree , queries_in_step , n_tree ,
            k , d_k_stack);
    }
    else if (stack_size <= 30)
    {
        dQueryAll<30> <<< numBlocks , numThreads>>>(
            d_query_points , d_tree , queries_in_step , n_tree ,
            k , d_k_stack);
    }
    else
    {
        dQueryAll<35> <<< numBlocks , numThreads>>>(
            d_query_points , d_tree , queries_in_step , n_tree ,
            k , d_k_stack);
    }
}

```

```

void cuQueryAll(struct Point *h_query_points , struct Node *
h_tree , int n_qp , int n_tree , int k , int *h_result)
{
    int numBlocks , numThreads , queries_in_step ,
    queries_done , stack_size ;
    struct Node *d_tree ;
    struct KPoint *d_k_stack , *h_k_stack ;
    struct SPoint *d_stack ;
    struct Point *d_query_points ;

    if (k >= n_tree)

```

```

{
    populateTrivialResult(n_qp, k, n_tree, h_result);
    return;
}

checkCudaErrors(cudaDeviceSetCacheConfig(
    cudaFuncCachePreferL1));

checkCudaErrors(cudaMalloc(&d_tree, n_tree * sizeof(
    Node)));
checkCudaErrors(cudaMemcpy(d_tree, h_tree, n_tree *
    sizeof(Node), cudaMemcpyHostToDevice));

queries_in_step = getQueriesInStep(n_qp, k, n_tree);

if (queries_in_step <= 0)
{
    printf("There is not enough memory to perform this
    queries on cuda.\n");
    return;
}

queries_done = 0;
stack_size = getSStackSize(n_tree);
queries_in_step = nextPowerOf2_...(++queries_in_step) >>
    1;
getThreadAndBlockCountForQueryAll(queries_in_step,
    numBlocks, numThreads);

h_k_stack = (KPoint *) malloc(queries_in_step * k *
    sizeof(KPoint));
checkCudaErrors(cudaMalloc(&d_k_stack, queries_in_step
    * k * sizeof(KPoint)));
checkCudaErrors(cudaMalloc(&d_stack, numThreads *
    numBlocks * stack_size * sizeof(SPoint)));
checkCudaErrors(cudaMalloc(&d_query_points,
    queries_in_step * sizeof(Point)));

while (queries_done < n_qp)
{
    if (queries_done + queries_in_step > n_qp)
    {
        queries_in_step = n_qp - queries_done;
    }
    checkCudaErrors(cudaMemcpy(d_query_points,

```

```

        h_query_points , queries_in_step * sizeof(Point),
        cudaMemcpyHostToDevice));

templateQueryAll(d_query_points , d_tree ,
    queries_in_step , n_tree , k, stack_size ,
    numBlocks , numThreads , d_k_stack);

checkCudaErrors(cudaMemcpy(h_k_stack , d_k_stack ,
    queries_in_step * k * sizeof(KPoint),
    cudaMemcpyDeviceToHost));

# pragma omp parallel for
for (int i = 0; i < queries_in_step ; ++i)
{
    for (int j = 0; j < k; ++j)
    {
        h_result[i * k + j] = h_k_stack[i * k + j].
            index;
    }
}

h_query_points += queries_in_step;
h_result += (queries_in_step * k);
queries_done += queries_in_step;
}

free(h_k_stack);
checkCudaErrors(cudaFree(d_query_points));
checkCudaErrors(cudaFree(d_k_stack));
checkCudaErrors(cudaFree(d_tree));
}

```

### **k-d tree split wrapper**

```

#ifndef _CU_UTILS_
#define _CU_UTILS_

void cuSetDevice(int device);
int cuGetDevice();
int cuGetDeviceCount();
size_t getFreeBytesOnGpu();

#endif

#include "helper_cuda.h"

```

```

#include "utils.cuh"
#include "knn-gpgpu.h"

void cuSetDevice(int device)
{
    checkCudaErrors(cudaSetDevice(device));
}

int cuGetDevice()
{
    int device;
    checkCudaErrors(cudaGetDevice(&device));
    return device;
}

int cuGetDeviceCount()
{
    int device_count;
    checkCudaErrors(cudaGetDeviceCount(&device_count));
    return device_count;
}

size_t getFreeBytesOnGpu()
{
    size_t free_byte , total_byte ;
    cudaError_t cuda_status = cudaMemGetInfo( &free_byte , &
        total_byte ) ;
    return free_byte - 1024;
}

#ifndef _KD_SEARCH_
#define _KD_SEARCH_
#include "cu-kd-search.cuh"

#define MIN_NUM_QUERY_POINTS 100U

void queryAll(struct Point *h_query_points , struct Node *
    tree , int qp_n , int tree_n , int k , int *result);

#endif

#include <stdlib.h>
#include <math.h>
#include <float.h>

#include <helper_cuda.h>

```



```

#include <kd-search.cuh>
#include <cu-kd-search.cuh>
#include "kd-tree-build.cuh"
#include "utils.cuh"

size_t getTreeSize(int n)
{
    return n * sizeof(struct Node);
}

size_t getNeededBytesForQueryAll(int n_qp, int k, int n)
{
    int numBlocks, numThreads;
    getThreadAndBlockCountForQueryAll(n, numBlocks,
        numThreads);
    return getTreeSize(n) + getNeededBytesInSearch(
        MIN_NUM_QUERY_POINTS, k, n, numThreads, numBlocks);
}

void maxHeapResultInsert(int *result, struct Node *tree,
    int point_index, struct Point qp, int k)
{
    int child, now;
    struct Node insert_node = tree[point_index];

    if (cuDist(qp, insert_node) > cuDist(qp, tree[result
        [0]]) ) return;

    for (now = 1; now * 2 <= k ; now = child)
    {
        child = now * 2;
        if (child <= k && cuDist(qp, tree[result[child +
            1]]) > cuDist(qp, tree[result[child]]) ) child
            ++;

        if (child <= k && cuDist(qp, insert_node) < cuDist(
            qp, tree[result[child]]))
        {
            result[now] = result[child];
        }
        else
        {
            break;
        }
    }
}

```

```

    result[now] = point_index;
}

void mergeResult(struct Node *tree, struct Point *
    query_points, int k, int n_qp, int root, int *
    result_right, int *result)
{
    #pragma omp parallel for
    for (int i_qp = 0; i_qp < n_qp; ++i_qp)
    {
        int i_k;
        int *result_max_heap = result + i_qp * k - 1;
        for (i_k = 0; i_k < k; ++i_k)
        {
            maxHeapResultInsert(result_max_heap, tree,
                result_right[i_qp * k + i_k], query_points[
                    i_qp], k);
        }
    }
}

void queryAll(struct Point *h_query_points, struct Node *
    h_tree, int n_qp, int n_tree, int k, int *h_result)
{
    if (getFreeBytesOnGpu() > getNeededBytesForQueryAll(
        n_qp, k, n_tree) || k > n_tree)
    {
        cuQueryAll(h_query_points, h_tree, n_qp, n_tree, k,
            h_result);
    }
    else
    {
        struct Node *tree_righth;
        int *h_result_right, root = n_tree / 2,
            n_tree_righth;
        h_result_right = (int *) malloc(k * n_qp * sizeof(
            int));

        queryAll(h_query_points, h_tree, n_qp, root, k,
            h_result);
        cudaDeviceReset();

        n_tree_righth = n_tree - root - 1;
        tree_righth = h_tree + (root + 1);
        store_locations(tree_righth, 0, n_tree_righth,

```

```

        n_tree_rigth);

    queryAll(h_query_points, tree_rigth, n_qp,
            n_tree_rigth, k, h_result_rigth);

    mergeResult(h_tree, h_query_points, k, n_qp, root,
                h_result_rigth, h_result);
    }
}

```

## C.4.2 OpenMP k-d tree search

```

#ifndef _KD.SEARCH.OPENMP
#define _KD.SEARCH.OPENMP
#include <point.h>
#include <stack.h>

void push(struct SPoint **stack, struct SPoint value);
struct SPoint pop(struct SPoint **stack);
struct SPoint peek(struct SPoint *stack);
int isEmpty(struct SPoint *stack);
void initStack(struct SPoint **stack);

void upDim(int &dim);

void initKStack(struct KPoint **k_stack, int n);
void insert(struct KPoint *k_stack, struct KPoint value,
            int n);
struct KPoint look(struct KPoint *k_stack);

void mpQueryAll(struct Point *query_points, struct Node *
                tree, int n_qp, int n_tree, int k, int *result);
void kNN(struct Point qp, struct Node *tree, int n, int k,
          int *result, struct SPoint *stack_ptr, struct KPoint *
          k_stack_ptr);

#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

// #ifdef HAVE_OPENMP
#include <omp.h>

```

```

// #endif

#include "kd-search-openmp.cuh"

float dist(struct Point qp, struct Node point)
{
    float dx = qp.p[0] - point.p[0],
          dy = qp.p[1] - point.p[1],
          dz = qp.p[2] - point.p[2];

    return (dx * dx) + (dy * dy) + (dz * dz);
}

void initStack(struct SPoint **stack)
{
    (*stack)[0].index = -1;
    (*stack)++;
}

int isEmpty(struct SPoint *stack)
{
    return peek(stack).index == -1;
}

void push(struct SPoint **stack, struct SPoint value)
{
    *((*stack)++) = value;
}

struct SPoint pop(struct SPoint **stack)
{
    return *--(*stack);
}

struct SPoint peek(struct SPoint *stack)
{
    return *(stack - 1);
}

void initKStack(struct KPoint **k_stack, int n)
{
    (*k_stack)--;
    for (int i = 1; i <= n; ++i)

```

```

    {
        (*k_stack)[i].dist = FLT_MAX;
        (*k_stack)[i].index = -1;
    }
}

void insert(struct KPoint *k_stack, struct KPoint k_point,
           int n)
{
    int i_child, now;
    struct KPoint child, child_tmp_2;
    for (now = 1; now * 2 <= n ; now = i_child)
    {
        i_child = now * 2;
        child = k_stack[i_child];
        child_tmp_2 = k_stack[i_child + 1];
        if (i_child <= n && child_tmp_2.dist > child.dist )
        {
            i_child++;
            child = child_tmp_2;
        }

        if (i_child <= n && k_point.dist < child.dist)
        {
            k_stack[now] = child;
        }
        else
        {
            break;
        }
    }
    k_stack[now] = k_point;
}

struct KPoint look(struct KPoint *k_stack)
{
    return k_stack[1];
}

void upDim(int &dim)
{
    dim = (dim + 1) % 3;
}

int target(struct Point qp, struct Node current, float dx)

```

```

{
    if (dx > 0)
    {
        return current.left;
    }
    return current.right;
}

int other(struct Point qp, struct Node current, float dx)
{
    if (dx > 0)
    {
        return current.right;
    }
    return current.left;
}

void kNN(struct Point qp, struct Node *tree, int n, int k,
int *result,
        struct SPoint *stack_ptr, struct KPoint *
        k_stack_ptr)
{
    int i, dim = 2;
    float current_dist, dx, dx2;

    struct Node current_point;
    struct SPoint *stack = stack_ptr,
                    current;
    struct KPoint *k_stack = k_stack_ptr,
                    worst_best;

    initStack(&stack);
    initKStack(&k_stack, k);

    worst_best = look(k_stack);
    current.index = n / 2;

    while (!isEmpty(stack) || current.index != -1)
    {
        if (current.index == -1 && !isEmpty(stack))
        {
            current = pop(&stack);
            dim = current.dim;

            dx = current.dx;

```

```

        dx2 = dx * dx;

        current.index = (dx2 < worst_best.dist) ?
            current.other : -1;
    }
    else
    {
        current_point = tree[current.index];

        current_dist = dist(qp, current_point);
        if (worst_best.dist > current_dist)
        {
            worst_best.dist = current_dist;
            worst_best.index = current.index;
            insert(k_stack, worst_best, k);
            worst_best = look(k_stack);
        }

        upDim(dim);
        current.dim = dim;
        current.dx = current_point.p[dim] - qp.p[dim];
        current.other = other(qp, current_point,
            current.dx);
        push(&stack, current);

        current.index = target(qp, current_point,
            current.dx);
    }
}

k_stack++;
for (i = 0; i < k; ++i)
{
    result[i] = k_stack[i].index;
}
}

void mpQueryAll(struct Point *query_points, struct Node *
tree, int n_qp, int n_tree, int k, int *result)
{
    int stack_size = log2((float)n_tree) + 5;
    #pragma omp parallel
    {
        int th_id = omp_get_thread_num();
        struct SPoint *stack_ptr = (struct SPoint *) malloc

```

```

        (stack_size * sizeof(struct SPoint));
struct KPoint *k_stack_ptr = (struct KPoint *)
        malloc(k * sizeof(struct KPoint));

while (th_id < n_qp)
{
    kNN(query_points[th_id], tree, n_tree, k,
        result + (th_id * k), stack_ptr, k_stack_ptr
    );
    th_id += omp_get_num_threads();
}

free(stack_ptr);
free(k_stack_ptr);
}
}

```



# Appendix **D**

## Risk assessment

NTNU	Utarbeidet av	Nummer	Dato
	HMS-avd.	HMSRV2601	22.03.2011
HMS	Godkjent av		Erstatter
	Rektor		01.12.2006

## Kartlegging av risikofylt aktivitet

**Enhet: IPM** **Dato: 14.01.14**

**Linjeleder: Torgeir Welo**

**Deftakere ved kartleggingen (m/ funksjon): Teodor Ande Elstad (Student), Simen Haugrud Granlund (Student)**

*(Ansv. veileder, student, evt. medveiledere, evt. andre m. kompetanse)*

**Kort beskrivelse av hovedaktivitet/hovedprosess:** Masteroppgave for Simen H. Granlund og Teodor A. Elstad. Utilizing general-purpose computing on graphic processing units (GPGPU) for engineering algorithm speedup.

**Er oppgaven rent teoretisk? (JA):** «JA» betyr at veileder inneslår for at oppgaven ikke inneholder noen aktiviteter som krever risikovurdering. Dersom «JA»: Beskriv kort aktivitetsten i kartleggingskemaet under. Risikovurdering trenger ikke å fylles ut.

**Signaturer:** Ansvartlig veileder: *Björn Haugen* Studenter: *Teodor A. Elstad, Simen H. Granlund*

ID nr.	Aktivitet/prosess	Ansvartlig	Eksisterende dokumentasjon	Eksisterende sikringsiltak	Lov, forskrift o.l.	Kommentar
1	Kontorarbeid	Simen	www.ntnu.no/adm/hm s/helse		Arbeidsmiljøloven	

