Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Distributed k-nearest Neighbors using Locality Sensitive Hashing and SYCL

Marcel Breyer

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	M.Sc. Gregor Daiß
Commenced:	July 3, 2020
Completed:	December 3, 2020

Abstract

More and more data is made available nowadays. This has the consequence that automatic or semi-automatic methods must be used to process these large data sets. One example of such a method is the classification. The nearest neighbor classifier, for example, assigns a class to a data point x based on its neighborhood. The naive approach to calculate nearest neighbors compares the data point x to all other data points in the data set. However, calculating the nearest neighbors for each data point using the naive approach has a quadratic complexity. This becomes more infeasible, the larger the data set grows. Therefore, approximate algorithms become more attractive. Such an algorithm is the Locality-Sensitive Hashing (LSH) algorithm, which uses hash tables together with locality-sensitive hash functions to reduce the data points that must be examined to calculate the nearest neighbors.

In the course of this work `sycl_lsh` library has been developed. This library implements the LSH algorithm with two different locality-sensitive hash functions, random projections, and entropy-based hash functions. The implementation uses `C++17` together with SYCL, which is an abstraction layer for OpenCL that allows targeting different hardware with a single source code. To support large data sets, the implementation utilizes multiple GPUs using MPI to enable the usage of both shared and distributed memory systems.

The results include specific tests for all important runtime parameters showing their influence on the runtime, recall, and error ratio. Knowing the behavior of the LSH algorithm concerning the different parameters is essential to be able to tune the algorithm to achieve the desired results while meeting the runtime requirements. These tests have been conducted for both hash function types, which are further compared to each other. Besides, the obtained results show that the used approach can easily scale on multiple GPUs using both locality-sensitive hash function types, achieving a parallel speedup of up to 7.0 when utilizing eight GPUs. Furthermore, it is shown that the `sycl_lsh` library can be used with three different SYCL implementations, ComputeCpp, hipSYCL, and oneAPI, to target different hardware architectures without any significant performance differences.

Kurzfassung

In der heutigen Zeit stehen immer mehr Daten zur Verfügung. Daher müssen immer häufiger automatische oder semi-automatische Verfahren verwendet werden, um eine derartige Menge an Daten verarbeiten zu können. Ein Beispiel für ein solches Verfahren ist die Klassifizierung. Der Nächste-Nachbarn-Klassifikator zum Beispiel weist einem Datenpunkt x eine Klasse basierend auf den Klassen seiner Nachbarn zu. Der naive Ansatz, um die nächsten Nachbarn eines Punktes zu berechnen, vergleicht den Datenpunkt x mit allen anderen Datenpunkten. Bei der Berechnung der nächsten Nachbarn jeden Punktes mit Hilfe des naiven Ansatzes liegt eine quadratische Komplexität vor, was für große Datenmengen zu ineffizient ist. Um dies zu vermeiden, können approximative Verfahren verwendet werden. Ein solches Verfahren ist das Locality-Sensitive Hashing (LSH). Dieses Verfahren verwendet Hash-Tabellen zusammen mit lokalitätserhaltenden Hash-Funktionen, um die Anzahl an Datenpunkten, die bei der Suche der nächsten Nachbarn betrachtet werden müssen, zu reduzieren.

Im Zuge dieser Masterarbeit wurde die `sycl_lsh` Bibliothek entwickelt. Diese Bibliothek implementiert den LSH Algorithmus mittels zweier verschiedener lokalitätserhaltender Hash-Funktionen, den random projections und den entropy-based Hash-Funktionen. Dabei verwendet die Implementierung `C++17` zusammen mit SYCL, einer Abstraktionsschicht für OpenCL, die es erlaubt, mit nur einem Quellcode unterschiedliche Hardware ansprechen zu können. Um große Datenmengen verarbeiten zu können, unterstützt die Implementierung außerdem mehrere Grafikkarten in einem potentiell verteilten System unter Verwendung von MPI.

Die Resultate umfassen, unter anderem, Tests für alle wichtigen Laufzeitparameter, um deren Auswirkungen auf die Laufzeit, Genauigkeit und Fehlerrate aufzuzeigen. Dieses Wissen um die Charakteristiken des LSH Algorithmus in Bezug auf die unterschiedlichen Parameter ist von großer Wichtigkeit, um die gewollten Resultate zu erreichen und gleichzeitig die Laufzeitanforderung aufrecht zu erhalten. Diese Tests wurden für beide Arten der lokalitätserhaltenden Hash-Funktionen durchgeführt, und deren Ergebnisse miteinander verglichen. Außerdem zeigten weitere Tests, dass der verwendete Parallelisierungsansatz auch unter Verwendung von mehreren Grafikkarten noch gut skaliert, unabhängig von dem verwendeten Typ der Hash-Funktionen. Dabei konnte eine Speedup von bis zu 7.0 unter Verwendung von acht GPUs erreicht werden. Des weiteren zeigen die Resultate, dass verschiedene SYCL Implementierungen verwendet werden können, um Code für unterschiedliche Hardware-Architekturen zu erstellen, ohne dabei signifikante Performance-Unterschiede hinnehmen zu müssen.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AoS Array of Structures. 7, 41

ARFF Attribute-Relation File Format. 47

CPU Central Processing Unit. 18

FPGA Field Programmable Gate Array. 37

GPU Graphics Processing Unit. 3, 4, 6

HCC Heterogeneous Compute Compiler. 21

HPC High-Performance Computing. 22

HPX High Performance ParallelX. 21

k-NN *k*-nearest neighbors. 7, 13, 17

LLNL Lawrence Livermore National Laboratory. 21

LSH Locality-Sensitive Hashing. 3, 4, 5, 7

MARS Multivariate Adaptive Regression Splines. 62

MPI Message Passing Interface. 3, 4, 7, 17, 18, 22

RAII Resource Acquisition Is Instantiation. 35

r-NN *r*-nearest neighbors. 23, 24

RTTI Runtime Type Information. 35

SMCP Single-Source Multiple Compiler-Passes. 35

SoA Structure of Arrays. 7, 41

TBB Thread Building Blocks. 21

1 Introduction

More and more data is made available nowadays, e.g., due to the sheer amount of smart devices, social media websites, or space telescopes. This massive amount of data has the consequence that automatic or semi-automatic methods must be used to process these big data sets. The problem of classification is one example for such a semi-automatic algorithm. Given a new data point, it should get assigned one of the possibly many predefined classes, based on already annotated data points.

One simple classifier can be the *k*-nearest neighbors (k-NN) algorithm already described by Cover and Hart [CH67] in 1967. Given a data point x, the k-NN classifier calculates the k points that are the nearest to x, with respect to a given distance or similarity metric. The resulting class can be determined by using a majority voting on the calculated k-NN points. Example applications for the k-NN classifier are text classification ([Tan06; TMD14; YYS09]) or gene selection ([LWDP01]). In the light of the current events surrounding COVID-19, attempts have been made by Shaban et al. [SRSA20] to use k-NN classifiers as a new patient detection algorithm. Da Silva et al. [SRMS20] compared different approaches, including a k-NN classifier, to forecast COVID-19 cases in Brazil or America.

The straightforward way to determine the k-NN of a data point is to calculate the distances to all other data points. With the growing size of data sets, this approach is becoming increasingly inefficient. Therefore, other algorithms must be used to process big data sets, such as approximate algorithms. Those approaches can improve the performance but will not necessarily return the exact nearest neighbors. One algorithm falling into this category is Locality-Sensitive Hashing (LSH) proposed by Indyk and Motwani [IM98] in 1998. The LSH algorithm inserts all data points into hash tables using locality-sensitive hash functions. To calculate the k-NN of a data point x, only those data points in the same hash bucket as x must be investigated, reducing the points to examine and therefore resulting in a possibly better performance.

In addition to using more efficient algorithms, taking advantage of the available hardware is essential, too. Therefore, the LSH algorithm used in this work has been implemented using GPUs to benefit from their highly parallel compute capabilities. Since more and more supercomputers are equipped with multiple GPUs per compute node, it is also important to utilize many GPUs in a distributed system. Looking at the TOP500 list from November 2020[1], the second place Summit[2] has six NVIDIA Volta V100 GPUs equipped per node,

[1] https://www.top500.org/lists/top500/2020/11/
[2] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

resulting in 27 648 GPUs in total. The third place Sierra[3] has four NVIDIA Volta V100 GPUs equipped per node, totaling 17 280 GPUs. Seven supercomputers in the top10 are equipped with GPUs (Summit, Sierra, Selene, JUWELS Booster Module, HPC5, Frontera, and Dammam-7). In order to access the GPUs in such distributed systems, the Message Passing Interface (MPI) is used.

An additional difficulty is that the supercomputers are equipped with many different hardware configurations. For example, the upcoming supercomputers Frontier[4] and El Capitan[5] will consist of AMD Central Processing Units (CPUs) and GPUs. The supercomputer Perlmutter[6] will contain AMD CPUs and NVIDIA GPUs, whereas Intel announced the supercomputer Aurora[7], which will include Intel CPUs and GPUs. The TOP500's eighth place supercomputer HPC5[8] contains Intel CPUs and NVIDIA GPUs. Therefore, it would be great to write a single program, which could run on every supercomputer independently of its hardware configuration. This work uses SYCL ([Ron20]) as an abstraction layer for the underlying hardware.

This work implements the `sycl_lsh` library utilizing the LSH algorithm to speed up the k-NN calculation. The implementation is based on my bachelor thesis "Ein hoch-performanter (approximierter) k-Nächste-Nachbarn Algorithmus für GPUs" ([Bre18]). It implements two different locality-sensitive hash functions for the Euclidean space, random projections and entropy-based hash functions. It uses SYCL to target different hardware, specifically GPUs from different vendors. Additionally, multiple GPUs are supported using MPI to enable the usage of both shared and distributed memory systems.

The next chapter, Chapter 2: "Related Work", covers other approaches for determining the k-NN. It also discusses methods for the distributed calculation of the k-NN and alternatives to SYCL for programming on different hardware architectures. The following Chapter 3: "Theory" deals with the theoretical basics, including the k-nearest neighbors' problem and the LSH algorithm together with the required locality-sensitive hash functions. Chapter 4: "SYCL" discusses the fundamental ideas of SYCL, followed by a small example and concluded by a comparison of currently existing SYCL implementations. Actual implementation details for the `sycl_lsh` library can be found in Chapter 5: "Implementation". Chapter 6: "Results" covers the obtained results in detail, including a comparison of different SYCL implementations, parameter tests for both hash function types, and scaling tests using up to eight GPUs. Finally, Chapter 7: "Conclusion" summarizes this work, while Chapter 8: "Future Work" provides an outlook for possible improvements and extensions.

[3] https://hpc.llnl.gov/hardware/platforms/sierra
[4] https://www.olcf.ornl.gov/frontier/
[5] https://www.amd.com/de/press-releases/2020-03-04-next-generation-amd-epyc-cpus-and-radeon-instinct-gpus-enable-el-capitan
[6] https://www.nersc.gov/systems/perlmutter/
[7] https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html
[8] https://www.eni.com/en-IT/operations/green-data-center-hpc5.html

2 Related Work

This chapter's primary objective is to give an overview of the related work for this master thesis. At first, other nearest neighbor algorithms are mentioned, e.g., other LSH variants or tree-based approaches. Next, algorithms for the distributed k-NN search are discussed. Afterward, alternatives for SYCL to develop applications supporting different hardware configurations are outlined.

2.1 k-Nearest Neighbors Algorithms

This section describes different approaches for calculating the k-NN, including, among other schemes, variations of the LSH algorithm or tree-based approaches.

2.1.1 Locality-Sensitive Hashing

A variation of LSH, called multi-probe LSH, has been proposed by Lv et al. [LJW+07]. The main idea is that not only the data points in the respective hash bucket are examined as potential k-NN candidates, but also data points in near hash buckets of the same hash table that are also likely to contain nearest neighbor points. This reduces the number of needed hash tables drastically while maintaining similar accuracies.

A similar approach is used in the a posterior multi-probe LSH proposed by Joly and Buisson [JB08]. Instead of only using probabilities as it is used in the standard multi-probe LSH algorithm, a posterior multi-probe LSH also takes prior about the queries and the searched data point into account. This results in a more accurate selection of other hash buckets likely to contain k-NN data points.

Panigrahy [Pan06] proposed the entropy-based LSH algorithm. Instead of indexing multiple hash buckets with the same query point x as in the multi-probe approach, it uses slight permutations of the query point x to index multiple hash buckets. This also reduces the number of needed hash tables.

Pan and Manocha [PM12] have proposed the combination of trees together with LSH as bi-level LSH. It divides the data set into subsets using RP-Tress and then creates LSH hash tables for each subset separately. This reduces the number of data points per hash table and therefore increases the performance.

Bawa, Condie, and Ganesan [BCG05] proposed, and later Andoni, Razenshteyn, and Nosatzki [ARN17] analyzed the LSH forest algorithm. The idea is to build a prefix tree based on labels created using locality-sensitive hash functions. Multiple such LSH trees form the proposed LSH forest.

Another type of hash function to form a forest has been proposed by Tao et al. [TYSK10]. Instead of prefix trees, the proposed algorithm uses locality-sensitive B-trees (LSB-tree) constructed using z-curves. The k-NN search is based on the length of the longest common prefix.

Terasawa and Tanaka [TT] proposed the Spherical LSH (SLSH) algorithm, where the data points lie on the surface of a $(d-1)$ unit sphere embedded in an $\mathcal{R}^d$ space. Therefore, another type of hash function has been created: the random rotated regular polytopes.

2.1.2 Tree-Based Approaches

Other approaches for calculating the k-NN besides LSH are tree-based methods.

In lower dimensions, normal kd-trees can be used to speed up the k-NN calculation ([Ben75; BL; FBF77; RS19]). To be able to use kd-trees in higher dimensions, they must be extended to randomized kd-trees ([EKNT12; EN13; ML09; SH08]).

There also exist other tree structures to speed up the k-NN calculations. Those include, for example, RP-Trees proposed by Dasgupta and Freund [DF08], hierarchical k-means trees proposed by Fukunaga and Narendra [FN75], priority search k-means trees proposed by Muja and Lowe [ML14], metric trees or spill trees proposed by Liu et al. [LMYG05], cover trees proposed by Beygelzimer, Kakade, and Langford [BKL06], or BBD trees proposed by Arya et al. [AMN+98].

However, Weber, Schek, and Blott [WSB98] have shown that all clustering or partitioning methods will degenerate to a linear search in high enough dimensions.

2.1.3 Other Approaches

Besides LSH or tree-based algorithms, other approaches can be used to calculate the k-NN, too. These approaches include, among other algorithms, k-NN graphs ([EMK+20; HASZ11; LZ09; PC05]) or product quantization ([GHKS13; HLY19; JDS11; KA14]).

2.2 Distributed k-Nearest Neighbors Algorithms

Work in different directions has already been done to calculate k-NN in distributed systems.

Neeb and Kurrus [NK16] examined the differences between serial and distributed algorithms such as direct approaches or hybrid spill trees using map-reduce.

Haghani, Michel, and Aberer [HMA09] developed an algorithm to minimize the network access in peer-to-peer networks using LSH. In contrast, Haghani, Michel, Aberer, et al. [HMA+08] tried to minimize the network traffic and improve the load balance.

Bahmani, Goel, and Shinde [BGS12] have proposed layered LSH, a distributed implementation of entropy LSH. It distributes the data points based on an LSH scheme and uses a map-reduce approach to calculate the k-NN. Zhang, Li, and Jestes [ZLJ12] developed a novel algorithm to efficiently calculate k-NN joins for large data sets using the map-reduce framework Hadoop.

The Spark-LSH scheme proposed by Zhang et al. [ZLXZ16] combines a shuffle-efficient indexing scheme with a location-aware querying scheme to improve performance by using the Apache Spark framework.

Developed by Sundaram et al. [STS+13], the Parallel LSH (PLSH) algorithm can utilize multiple nodes and cores to support high-throughput streaming of new data. It uses novel ideas like a cache-conscious hash table layout, a two-level merge algorithm for hash table construction, efficient duplicate elimination while querying, an insert-optimized hash table structure, an efficient data expiration algorithm for data streaming, and an accurate performance estimation model.

Patwary et al. [PSS+16] developed a parallel, highly optimized kd-tree algorithm utilizing Xeon Phis to support massive data sets (e.g., 189 billion points in 3 dimensions).

2.3 SYCL Alternatives

Besides SYCL, other abstraction layers exist to support different hardware without the need to use vendor-specific languages or constructs.

The most prominent representative for cross-platform development is OpenCL which is developed by Khronos OpenCL Working Group [Khr20]. It is a standard for general-purpose parallel programming across CPUs, GPUs, and other processors.

A more similar alternative for SYCL is Kokkos. Developed by Edwards, Trott, and Sunderland [ETS14], Kokkos is a model in `C++` for writing performance portable applications. It is an abstraction layer for parallel execution and data management. Supported backends are CUDA, High Performance ParallelX (HPX), OpenMP, and pthreads. Additionally, support for AMD's HIP backend is planned.

RAJA, developed by the Lawrence Livermore National Laboratory (LLNL) ([BHSV19]), is a `C++` library to write single-source applications that can target different hardware (e.g., CPUs, GPUs, or Xeon Phi) using multiple programming model backends such as OpenMP, CUDA, Intel's Thread Building Blocks (TBB), or AMD's Heterogeneous Compute Compiler (HCC).

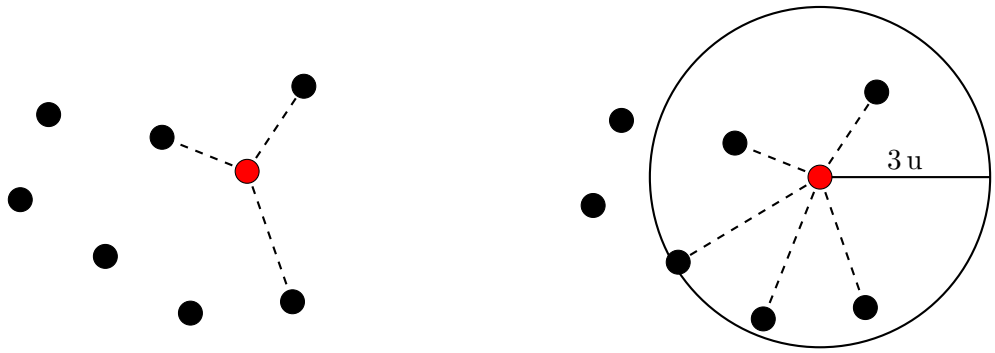
Furthermore, efforts have been made to incorporate the ability to target different hardware (e.g., GPUs) by using `C++` standard executor extensions ([HGK+18; HGK+20; HK19]).

This master thesis focuses on a standard LSH implementation suitable for High-Performance Computing (HPC) in contrast to the already proposed map-reduce approaches. In contrast to the kd-tree based implementation targeting Xeon Phi accelerators, the developed `sycl_lsh` library targets distributed multi-GPU systems using Message Passing Interface (MPI). However, a preliminary, simplified multi-probe LSH has already been implemented but has not been tested or merged into the final `sylc_lsh` library. Additionally, the library uses SYCL to support a wide range of hardware.

3 Theory

In this chapter, all theoretical fundamentals will be discussed. At first, the problem of the nearest neighbors will be explained. After that, the LSH algorithm and the related locality-sensitive hash functions will be described.

3.1 The Nearest Neighbors Problem



(a) Calculate the *k*-nearest neighbors (with $k = 3$) for the highlighted data point. For that, the distances to all other points must be determined. Afterward, the three points with the smallest distances will be returned as *k*-nearest neighbors.

(b) Calculate the *r*-nearest neighbors (i.e., in this example, all points within the radius $r = 3\,\mathrm{u}$). For that, the distance to all other points must be determined. Afterward, all data points within a distance of $3\,\mathrm{u}$ to the highlighted point will be returned as *r*-nearest neighbors.

Figure 3.1: Example of both nearest neighbor types in the two-dimensional Euclidean space: **(a)** the *k*-nearest neighbors and the **(b)** *r*-nearest neighbors (fixed-radius nearest neighbors).

In many cases, the search for the nearest neighbors of a data point is part of another algorithm. One example is the algorithm for data classification, which was already proposed in 1967 by Cover and Hart [CH67]. In the proposed algorithm, a data point must be assigned one of M different classes. To classify the data point x, the algorithm calculates the nearest neighbor of x and determines the class c of this nearest neighbor. Afterward, the class c is assigned to the data point x. The algorithm can be extended using not only the nearest neighbor to determine the class c but the *k*-nearest neighbors. The resulting class c will then be determined by using a majority voting mechanism.

To determine, what the "nearest" neighbors of a point are, a distance or similarity metric is used. Example metrics include the Manhattan distance, Euclidean distance, Hamming distance, cosine similarity, or Jaccard index. For example, the metric used in Figure 3.1 is the Euclidean distance, which is also the only distance metric considered in the remaining of this master thesis.

The nearest neighbors problem can be interpreted in two different ways, both displayed in Figure 3.1. Figure 3.1a shows the *k*-nearest neighbors. For any data point, precisely k neighbors will be calculated. In contrast to this, with the *r*-nearest neighbors (also called fixed-radius nearest neighbors) depicted in Figure 3.1b, all points within the given radius r are returned. The major advantage of the *k*-nearest neighbors variant is that the number of nearest neighbors to calculate is known in advance. In contrast, the resulting number of nearest neighbors in the *r*-nearest neighbors variant cannot be determined ahead of time. Therefore, in the following, only the *k*-nearest neighbors variant will be used.

The exact *k*-nearest neighbors can be calculated as shown in Algorithm 3.1.

Algorithm 3.1 Naive algorithm to calculate the *k*-nearest neighbors of a single data point.

```
1: Given: A set of data points 𝒟.
2: function kNearestNeighbors(k ∈ ℕ₊, x ∈ 𝒟)
3:     for all data points y ∈ 𝒟 \ {x} do
4:         Calculate the distance between x and y.
5:     end for
6:     Determine the k data points y_i, for which the distances to x are the smallest.
7: return The calculated k-nearest neighbors.
8: end function
```

The function described in Algorithm 3.1 calculates the distances to all other points to determine the *k*-nearest neighbors for the given point x. To calculate the k-NN for all data points, this function must be called for each data point, resulting in an overall runtime complexity of $\mathcal{O}(n^2)$ given the number of data points is n. This runtime complexity prevents big data sets from being used even on modern hardware. For example, let the distance between two data points be calculable in $1\,\mathrm{ns}$. Determining the nearest neighbors for all points, given a data set of size one billion, would take approximately 31.7 years.

Therefore, another approach must be used to process data sets with millions or billions of data points. One possibility is to use approximation algorithms to calculate the k-NN. This type of algorithm does not return the exact k-NN, but neighbors, which should be sufficiently close enough to the correct ones. However, for many algorithms using the approximate nearest neighbors can be sufficient. For example, the classification algorithm mentioned earlier in this section can be used with approximate nearest neighbors. Since data points that are sufficiently close to the correct nearest neighbors have the same class with high probability, they can be used for classification.

3.2 Locality-Sensitive Hashing

The following section covers the approximation algorithm Locality-Sensitive Hashing (LSH) proposed by Indyk and Motwani [IM98].

The basic idea of LSH is to sort all data points of a data set into hash tables according to special locality-sensitive hash functions. Using these hash functions, points that are close or similar to each other should have the same hash value with a high probability. In contrast, points that are far away from each other or dissimilar should have the same hash value only with low probability. To calculate the *k*-nearest neighbors of a data point, not all other data points must be examined, but only those inserted into the same hash buckets. This can reduce the number of necessary distance calculations drastically, resulting in better performance.

At first, locality-sensitive hash functions, in general, will be discussed, followed by two concrete examples in Section 3.2.1: “Random Projections Hash Functions” and Section 3.2.1: “Entropy-Based Hash Functions”. Afterward, in Section 3.2.2, the LSH algorithm will be examined closer.

3.2.1 Locality-Sensitive Hash Functions

There are many different types of hash functions: universal hash functions for minimizing the probability of hash collisions, hash functions for calculating checksums, or cryptographic hash functions. This section covers another type of hash functions: the locality-sensitive hash functions. The main idea behind this kind of hash functions is that points which are closer to each other are assigned the same hash value with a high probability, whereas points that are far from each other will get the same hash value with only a small probability.

At first, the formal definition of a locality-sensitive hash function will be given, followed by two hash function types for the Euclidean distance metric, namely random projections and entropy-based hash functions.

Locality-Sensitive Hash Family

Normal hash functions try to minimize the number of collisions, i.e., for two points x and y, the probability of having the same hash value $h(x) = h(y)$ is small. On the other hand, locality-sensitive hash functions want to maximize the probability of hash collisions if the two points x and y are similar to each other with respect to the chosen distance or similarity metric. However, for points that are far from each other, and thus not similar, the probability for hash collision should be small. Indyk and Motwani [IM98] defined such a family of hash functions as described in Definition 3.2.1.

Definition 3.2.1 (locality-sensitive hash family)
Given a data set $\mathcal{D}$ and a metric $dist$ a hash family $\mathcal{H}$ is said to be (r_1, r_2, p_1, p_2)-sensitive, where r_1 and r_2 denote distances and p_1 and p_2 probabilities, if the following holds for any $x, y \in \mathcal{D}$:

- *if $dist(x, y) \leq r_1$, then $P(h(x) = h(y)) \geq p_1$*
- *if $dist(x, y) \geq r_2$, then $P(h(x) = h(y)) \leq p_2$*

for $r_1 < r_2$ and $p_1 > p_2$.

The relationships between r_1, r_2 and p_1, p_2 can be explained by the simple fact that if $r_1 > r_2$ or $p_1 < p_2$ would hold, two points would get the same hash value with a high probability even though they are not similar. That would contradict the intention of locality-sensitive hash functions to generate collisions with a high probability only if two points are similar.

Random Projections Hash Functions

One type of locality-sensitive hash functions proposed by Datar et al. [DIIM04] uses p-stable or α-stable distributions defined by Mainardi [Mai07] and Nolan [Nol18].

Definition 3.2.2 (stable distribution)
A random variable X is said to have a stable distribution if for any $n \geq 2$ holds

$$X_1 + X_2 + \cdots + X_n \stackrel{d}{=} c_n \cdot X + d_n$$

where $X_1, X_2, \ldots, X_n$ are independent copies of X, $c_n > 0$, and $d_n \in \mathbb{R}$. The operator $\stackrel{d}{=}$ denotes the equality in distribution, i.e., both sides of the equation have the same distribution.

A distribution is stable if a linear combination of two or more independent random variables from the distribution preserves its shape, up to scale and shift factor. Examples for p-stable distributions are the 1-stable Cauchy distribution, the 2-stable normal distribution, and the $\frac{1}{2}$-stable Lévy distribution.

Datar et al. [DIIM04] proposed the random projections using p-stable distributions as a locality-sensitive hash function.

Definition 3.2.3 (random projections)
Given a data point $x \in \mathbb{R}^d$, a vector $a \in \mathbb{R}^d$ with entries drawn from a p-stable distribution, a scalar $w \in \mathbb{R}_+$ and a second scalar b drawn from a uniform distribution of the range $[0, w]$, the hash value $h(x)$ is calculated as:

$$h(x) = \left\lfloor \frac{a \cdot x + b}{w} \right\rfloor$$

In case of the Euclidean distance, also called L^2 norm, the entries of the vector a must be drawn from a 2-stable distribution, i.e., the normal distribution.

Generating such hash functions is comparatively efficient since it only involves drawing $d + 1$ random values from different distributions. Calculating the hash value $h(x)$ is also relatively efficient because it only requires one dot product along with one addition and an integer division.

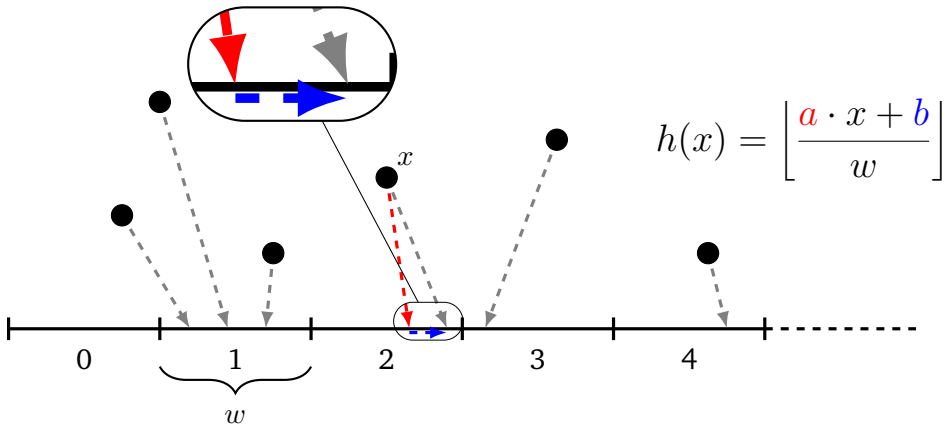
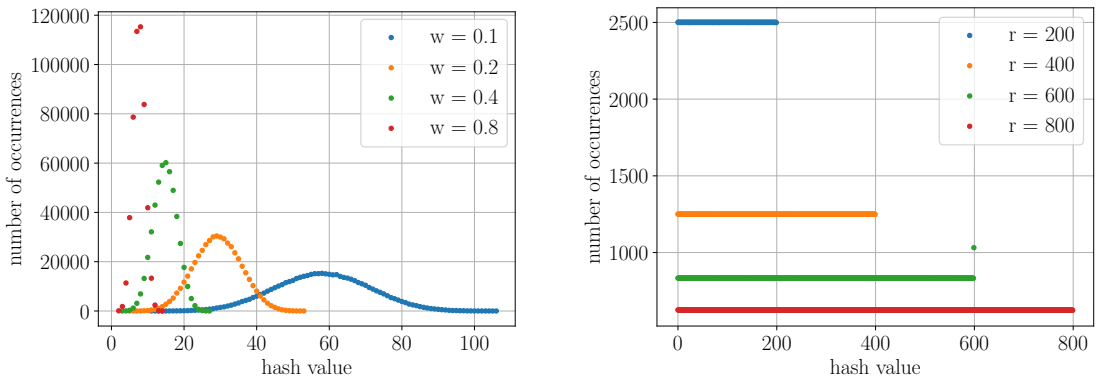


Figure 3.2: Example of the random projection hash functions. The data points are projected onto a number line, which is divided into segments of size w. The resulting hash value corresponds to the projected segment.

Figure 3.2 shows the graphical interpretation of the random projections as defined in Definition 3.2.3. Multiplying the data point x with the vector a maps the high-dimensional data point to an one-dimensional value on the number line. Afterward, the displacement value b is applied. The resulting value is divided by the segment size w using an integer division. The hash value of x is the result of this integer division. In Figure 3.2, the data point x is mapped to 2.65 using the dot product with the vector a. Next, the mapped value is shifted by b to the right resulting in 2.9. The final hash value $h(x)$ of x is the integer part of 2.9 and hence 2.

As shown in Figure 3.3a, the resulting distribution of the hash values resembles the normal distribution. The parameter w influences the shape of the resulting normal distribution. A small w means that the segments are smaller and, therefore, fewer data points have the same hash value resulting in a flatter normal distribution curve. In contrast, a bigger w means that the segments are wider. This results in more data points with the same hash value and a steeper curve in Figure 3.3a.



(a) Distribution of the hash values using the random projections. The hash value distribution resembles the normal distribution. Reducing the parameter w results in smaller segments and, therefore, on average fewer points per hash bucket.

(b) Distribution of hash values using the entropy-based hash functions. The hash values are uniformly distributed. Reducing the parameter r results in fewer used hash values and, therefore, more points per hash bucket.

Figure 3.3: Example of the distributions of the hash values using random projections and entropy-based hash functions.

Entropy-Based Hash Functions

Another type of locality-sensitive hash function for the Euclidean distance has been proposed by Wang et al. [WGLG12] providing a more uniform distribution of the hash values.

Definition 3.2.4 (entropy-based hash functions)
Given a data point $x \in \mathbb{R}^d$ and a vector $a \in \mathbb{R}^d$ the initial mapping is calculated by using:

$$h'(x) = a \cdot x$$

These initially mapped values will be sorted and then split into $r \in \mathbb{N}_+$ groups of the same size. Afterward, the resulting $r-1$ cut-off-points are recorded as $q_1, \ldots, q_{r-1}$. The final hash value of the data point x will be calculated as:

$$h(x) = \begin{cases} 0, & \text{if} \quad h'(x) \leq q_1 \\ 1, & \text{if} \quad q_1 < h'(x) \leq q_2 \\ \quad \vdots & \\ r-2, & \text{if} \quad q_{r-2} < h'(x) \leq q_{r-1} \\ r-1, & \text{if} \quad h'(x) > q_{r-1} \end{cases}$$

Generating entropy-based hash functions is more expensive compared to the random projections. At first, d values must be drawn from a random distribution to get the vector a.

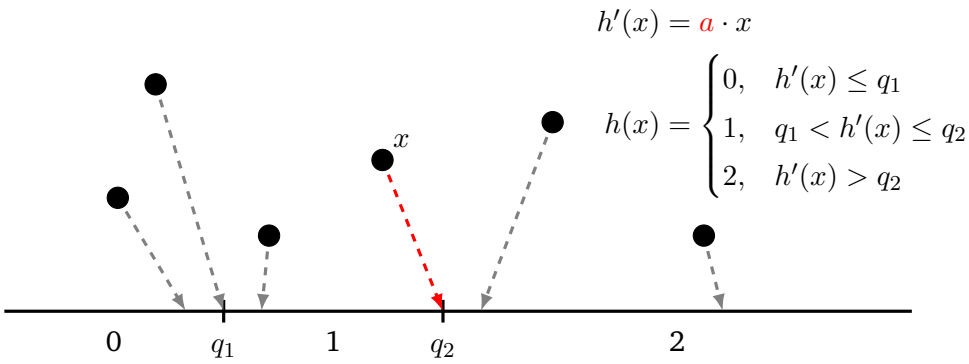


Figure 3.4: Example of the entropy-based hash functions with $r = 3$. The data points are projected onto a number line divided into segments of different sizes according to the cut-off points q_i. The resulting hash value corresponds to the projected segment.

Afterward, the initial mapping for all data points must be calculated, followed by sorting these values. The sorted values are used to retrieve the cut-off points needed to calculate the final hash value $h(x)$. This results in a time complexity of $\mathcal{O}(n + n \cdot \log(n))$. Calculating the hash value $h(x)$, however, is relatively efficient. It requires a single dot product and a lookup involving the cut-off points to calculate the final hash value.

Figure 3.4 shows the graphical interpretation of the entropy-based hash functions as defined in Definition 3.2.4. Multiplying the data point x with the vector a results in the initial mapping from the high-dimensional data point to a one-dimensional value on the number line. In the next step, these mapped values must be sorted. Afterward, the $r - 1$ cut-off points are calculated by splitting the sorted values into r distinct groups of the same size. For example, Figure 3.4 contains six data points with an r value of three. Therefore, the six data points must be divided into three groups, each containing two data points. The resulting cut-off points are the values on the border between those groups, displayed in Figure 3.4 as q_1 and q_2. The hash value of x is the group in which $h'(x)$ mapped x according to the cut-off points. In Figure 3.4, the mapped value of the data point x corresponds to the second cut-off point q_2. Therefore, since $q_1 < h'(x) \leq q_2$ holds, the hash value $h(x)$ of x is one.

As shown in Figure 3.3b, the resulting distribution of the hash values resembles a uniform distribution. All hash buckets have the same number of data points assigned. How many data points are assigned to a hash bucket is determined by the parameter r. A small r means that only a few hash values are used, resulting in more data points per hash bucket. In contrast, a bigger r means that more hash values are used. This results in fewer data points per hash bucket.

The uniform distribution of the data points over the hash buckets means that the same number of data points must be considered when calculating the k-NN. This results in a better load balance compared to the random projections, which is beneficial when using GPUs.

Hash Signature

The previous Section 3.2.1 and Section 3.2.1 described hash functions that form a locality-sensitive hash family $\mathcal{H}$ as defined in Definition 3.2.1. However, to use those hash functions to index LSH hash tables, they must be further refined. To achieve that, a second family of hash functions $\mathcal{H}'$ is created by concatenating m hash functions of a hash family $\mathcal{H}$ ([DIIM04]).

Definition 3.2.5 (hash signature)
Given a locality-sensitive hash family $\mathcal{H}$ and m hash functions $h_i \in \mathcal{H}$, the hash signature g of the data point $x \in \mathcal{D}$ is calculated using concatenation:

$$g(x) \coloneqq h_1(x) \circ h_2(x) \circ \ldots \circ h_m(x)$$

The resulting hash functions g_j form a new locality-sensitive hash family $\mathcal{H}'$.

The hash functions h_i can be concatenated to the hash signature g using two different approaches, as described by Leskovec, Rajaraman, and Ullman [LRU14].

Using the and-concatenation, for two data points $x, y \in \mathcal{D}$ the two hash signatures $g(x) = g(y)$ are equal, if $\forall 1 \leq i \leq m : h_i(x) = h_i(y)$ holds. Since the hash functions h_i are drawn uniformly at random from the hash family $\mathcal{H}$, the and-concatenated hash signature g is (r_1, r_2, p_1^m, p_2^m)-sensitive. For two data points $x, y \in \mathcal{D}$, applying an or-concatenation, the two hash signatures $g(x) = g(y)$ are equal, if $\exists 1 \leq i \leq m : h_i(x) = h_i(y)$ holds. Since the hash functions h_i are drawn uniformly at random from the hash family $\mathcal{H}$, the or-concatenated hash signature g is $(r_1, r_2, 1-(1-p_1)^m, 1-(1-p_2)^m)$-sensitive.

These concatenations can be further combined. For example, given a hash family $\mathcal{H}$ using an and-concatenation with $m_1 = 4$, a new hash family $\mathcal{H}_1$ can be constructed. Afterward, using an or-concatenation with $m_2 = 3$, a third hash family $\mathcal{H}_2$ can be generated. This hash family $\mathcal{H}_2$ would be an $(r_1, r_2, 1-(1-p_1^4)^3, 1-(1-p_2^4)^3)$-sensitive hash family using $m_1 \cdot m_2 = 12$ hash functions from $\mathcal{H}$ to generate one hash signature function g.

3.2.2 Locality-Sensitive Hashing Algorithm

Algorithms, which use partitioning schemes to support a more efficient lookup, like LSH or tree-based approaches, can be separated into two steps. As a first step, the partitioning structures, e.g., hash tables or trees, will be constructed. In a second step, those partitioning structures are used to speed up the respective calculations.

In the following, both steps, creating the hash tables and, afterward, the k-NN calculation will be examined for the LSH algorithm as explained by Indyk and Motwani [IM98].

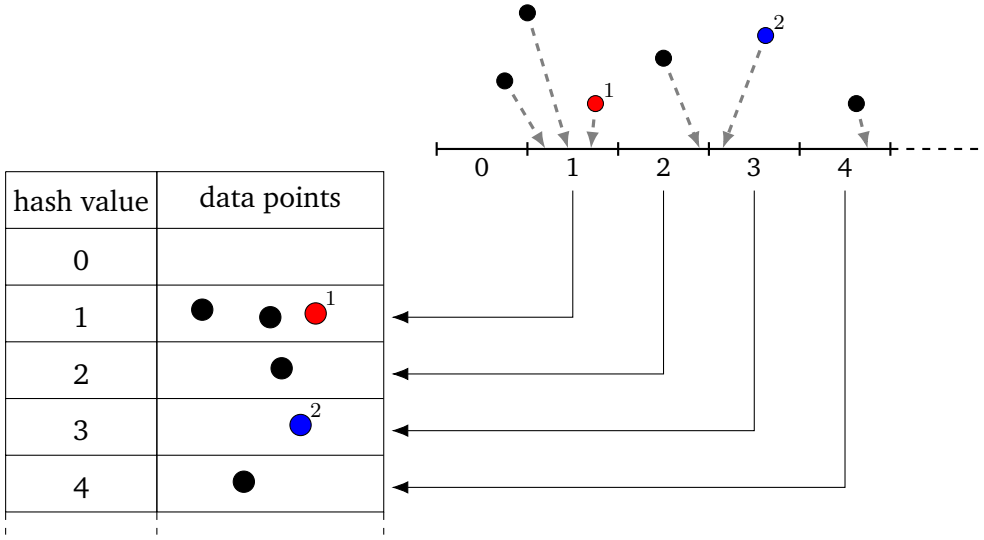


Figure 3.5: Example of the creation of a single LSH hash table together with the calculation of the k-NN of a data point. All data points are inserted into hash buckets according to their hash value. Afterward, the k-NN are calculated by only examining the data points inside the same hash bucket.

Algorithm 3.2 The algorithm to create the LSH hash tables.

1: Given: A set of data points $\mathcal{D}$, the number of hash tables l and the number of hash functions per hash table m.
2: Create a pool of locality-sensitive hash functions $\mathcal{H}$.
3: Generate l different hash signature functions $g_i \in \mathcal{H}'$ each by concatenating m hash functions $h_j \in \mathcal{H}$.
4: **for all** data points $x \in \mathcal{D}$ **do**
5: **for all** hash tables $i \in [1, l]$ **do**
6: Calculate the hash value of x using the hash function $g_i \in \mathcal{H}'$.
7: Add x to the hash bucket $g_i(x)$ in hash table i.
8: **end for**
9: **end for**

Creation of the Hash Tables

The algorithm used to create the LSH hash tables can be seen in Algorithm 3.2.

At first, a hash pool consisting of locality-sensitive hash functions is created. Afterward, the hash signature functions g are generated by concatenating hash functions from the hash pool. These hash signature functions are used to assign each data point to its corresponding hash bucket in each hash table. The data point x is added to the hash bucket $g_i(x)$ of the hash table i. Since the hash values calculated using the g_i hash signature functions can be rather large, standard hashing methods must be used to map the hash values to a smaller number of hash buckets.

Figure 3.5 shows an example of the creation of a single LSH hash table. Data point one has the hash value one and therefore gets assigned to the hash bucket one along with two other data points. It can also happen that only one data point is assigned to a hash bucket, like data point two in Figure 3.5, or that a hash bucket is empty, like hash bucket zero.

Calculation of the k-Nearest Neighbors

Algorithm 3.3 The algorithm to calculate the *k*-nearest neighbors of a single data point using the LSH algorithm.

1: Given: A set of data points $\mathcal{D}$, a locality-sensitive hash family $\mathcal{H}'$ containing hash signature functions and the previously created set of LSH hash tables $\mathcal{L}$.
2: **function** kNEARESTNEIGHBORS($k \in \mathbb{N}_+$, $x \in \mathcal{D}$)
3: **for all** hash tables $l \in \mathcal{L}$ **do**
4: Calculate the hash value of x using the hash signature function $g_l \in \mathcal{H}'$.
5: **for all** data points y in the hash bucket $g_l(x)$ of hash table l **do**
6: Calculate the distance between x and y.
7: **end for**
8: Update the k-NN, if data points with a smaller distance to x have been found.
9: **end for**
10: **return** The calculated *k*-nearest neighbors.
11: **end function**

After all hash tables have been created successfully, the actual k-NN search can be performed, as shown in Algorithm 3.3. For a given query point x, each hash table must be examined. At first, the hash value for the query point x for hash table l must be calculated using the g_l hash signature function. To calculate the k-NN of x, only those data points inside the hash bucket $g_l(x)$ must be inspected. Data points in other hash buckets of the same hash table are not considered during the k-NN search, reducing the number of data points to examine. To calculate the k-NN inside a hash bucket, the naive algorithm described in Algorithm 3.1 can be used.

Figure 3.5 can once again be used as an example. The goal is to calculate the 1-NN of data point one. At first, the hash signature of point one must be calculated, which is one in the presented example. Therefore, to calculate the 1-NN, all other data points in the hash bucket one must be considered, resulting in two comparisons. The point of those two, which is closer to point one, will be returned as the 1-NN. Since only two data points have been considered in the 1-NN search, the number of comparisons has been reduced by three compared to the naive k-NN algorithm. However, the returned point is not the actual 1-NN of the data point one since the correct nearest neighbor has been assigned to hash bucket two. The reason for this to happen is that Locality-Sensitive Hashing is an approximation algorithm. The locality-sensitive hash functions try to maximize the probability of hash collisions if two data points are similar, but achieving $100\,\%$ is quite difficult.

As another example, the 1-NN of the data point two in Figure 3.5 can be calculated. To do that, all other data points in the hash bucket three must be considered. Since no other

point is contained in the hash bucket three, no 1-NN can be found for the data point two using the LSH algorithm with the example hash table in Figure 3.5.

Both problems can be solved if the LSH parameters are selected carefully. Increasing the number of hash tables rises the number of hash signatures to calculate for each data point, increasing the probability that two similar points match in at least one hash signature. Reducing the number of hash functions per hash signature g increases the probability of two points to match in the hash signature since they must conform in fewer hash functions to get the same hash signature value.

The complexity of the Locality-Sensitive Hashing (LSH) algorithm has been described by Indyk and Motwani [IM98]. Suppose an (r_1, r_2, p_1, p_2)-sensitive hash family with a distance or similarity metric $dist$ exists. In that case, there also exists an algorithm with space complexity $\mathcal{O}(dn + n^{1+p})$ and time complexity $\mathcal{O}(n^p)$ with $p = \frac{\ln \frac{1}{p_1}}{\ln \frac{1}{p_2}}$ for calculating the nearest neighbors assuming the hash tables already exist as shown by Indyk and Motwani [IM98]. Since $p_1 > p_2$ holds (see Definition 3.2.1), the time complexity is sub-linear in the number of data points for searching the k-NN for a single data point. To calculate the k-NN for each data point, the algorithm above must be repeated n times, resulting in a sub-quadratic runtime compared to the quadratic runtime of the naive algorithm described in Algorithm 3.1.

4 SYCL

SYCL (pronounced “sickle”) is a cross-platform abstraction layer for OpenCL. The Khronos Group develops its specification with the current version being 1.2.1 ([Ron20]), while SYCL 2020 Provisional is currently work in process. The main idea of SYCL is the combination of OpenCL’s concepts, portability, and efficiency with the ease of use of modern C++.

SYCL uses Single-Source Multiple Compiler-Passes (SMCP) to compile a single source file with multiple compilers. This allows SYCL to compile host code using the host compiler, while the device compiler will compile only the kernel code.

Since SYCL is based on OpenCL, it inherits its notations (such as work-groups, work-items, private memory, or local memory) from OpenCL.

The main advantages of SYCL are simplicity, reuse, and efficiency. SYCL kernels are written using pure C++ and do not need complicated separation of host and device source code. This allows for easier development of SYCL applications compared to pure OpenCL. Since SYCL device code is written using pure C++, host code constructs can be reused. However, a few C++ features cannot be used inside the SYCL kernel code. These features include virtual functions, function pointers, exceptions, Runtime Type Information (RTTI), or compiler-specific features. Specifically, this means that widely used C++ features like inheritance (without virtual functions) or templates can be used inside SYCL kernel code. Because the SYCL host and device code are tied together more closely, it is easier for the compiler to generate more specialized device code based on decisions made in the host code, e.g., better function inlining.

As stated previously, SYCL is based on OpenCL. However, SYCL extends the underlying OpenCL model in two ways. Firstly, it introduces a new syntax for hierarchical parallelism. Furthermore, it separates the data access from the data storage using C++ well known Resource Acquisition Is Instantiation (RAII) idiom. This also removes the need for explicit data movement by the user.

Hereafter, an example of a SYCL code for a simple vector addition will be examined, followed by comparing current SYCL implementations.

Listing 4.1 Example code for a simple parallel vector addition using SYCL.

```
#include <array>
#include <iostream>
#include <numeric>

#include <CL/sycl.hpp>
namespace sycl = cl::sycl;

class kernel_name;

int main() {
  constexpr std::size_t size = 10;
  std::array<float, size> a;
  std::iota(a.begin(), a.end(), 1.0);
  std::array<float, size> b;
  std::iota(b.begin(), b.end(), 1.0);
  std::array<float, size> c;

  {
    sycl::queue queue(sycl::default_selector{});

    sycl::buffer<float, 1> buffer_a(a.data(), a.size());
    sycl::buffer<float, 1> buffer_b(b.data(), b.size());
    sycl::buffer<float, 1> buffer_c(c.data(), c.size());

    queue.submit([&](sycl::handler& cgh) {
      auto acc_a = buffer_a.get_access<sycl::access::mode::read>(cgh);
      auto acc_b = buffer_b.get_access<sycl::access::mode::read>(cgh);
      auto acc_c = buffer_c.get_access<sycl::access::mode::discard_write>(cgh);

      const auto exec_range = sycl::range<>(size);
      cgh.parallel_for<kernel_name>(exec_range, [=](sycl::item<> item) {
        const auto idx = item.get_linear_id();

        acc_c[idx] = acc_a[idx] + acc_b[idx];
      });
    });
  }

  for (const float val : c) {
    std::cout << val << ' ';
  }

  return 0;
}
```

4.1 Example: Vector Addition

The code in Listing 4.1 shows an example of a simple vector addition using SYCL to perform the calculation in parallel using CPUs, GPUs, or Field Programmable Gate Arrays (FPGAs).

All of SYCL's functionality can be included using the `<CL/sycl.hpp>` header (line 5). Since all functions reside in the `cl::sycl` namespace, it can be more convenient to create a short namespace alias as in line 6.

In line 19, a `sycl::queue` is used to schedule kernels on a SYCL device. The target device can be selected using a `sycl::device_selector`. Predefined `sycl::device_selectors` are: `sycl::default_selector` (selecting a device based on an implementation-defined heuristic), `sycl::gpu_selector`, `sycl::accelerator_selector`, `sycl::cpu_selector` or `sycl::host_selector` (must always return a valid `sycl::device`). Furthermore, it is possible to derive from the `sycl::device_selector` class to select a device based on a custom heuristic, e.g., whether a device supports a specific OpenCL extension.

Optionally, a `sycl::async_handler` function can be provided to the `sycl::queue` constructor. On a call to `sycl::queue::wait_and_throw`, in the case of asynchronous exceptions, the registered exception handler will be called.

Next, the data must be made visible to the SYCL runtime. To achieve that, the ownership of the data needs to be transferred to SYCL. This can be done by the means of the `sycl::buffer` class in lines 21-23. Since the data ownership is transferred to the `sycl::buffer`, it is undefined to access the data through the original arrays (lines 12-16) after the transfer. After the destruction of a `sycl::buffer` at the end of its scope at line 37, the ownership will be transferred back to the original array.

The next step is the creation of a command group using the `sycl::queue::submit` call in line 25. This command group object is used to represent the required operations to process data on a device.

In lines 26-28 for each `sycl::buffer` a `sycl::accessor` is created, because those accessors are the only way to access the data inside a SYCL kernel. A `sycl::accessor` has two important properties, a `sycl::access::mode`, and a `sycl::access::target`. The `sycl::access::mode` must be provided to a call to `sycl::buffer::get_access`. Possible values are `sycl::access::mode::read` (only), `sycl::access::mode::write` (only) and `sycl::access::read_write`, `sycl::access::mode::discard_write` and `sycl::access::mode::read_write` (discard current values in buffer), and `sycl::access::mode::atomic`.

The `sycl::access::target` specifies what the accessor provides access to. The default accessor target is the global memory (`sycl::access::target::global_buffer`). Another important value is `sycl::access::target::local` for accessing work-group local memory.

The call to the previously mentioned `sycl::buffer::get_access` function can include an optional `sycl::handler` parameter. This allows the SYCL runtime to automatically detect and resolve dependencies between accessors within a command group. For example, let there be a second kernel in the current example code accessing the `buffer_c` in `sycl::access::mode::read`. Since this read operation depends on the write operation in the first kernel, the second kernel is only called after the first kernel finished execution. On the other hand, if the second kernel only accesses the `buffer_a` in `sycl::access::mode::read` and does not access `buffer_c` at all, both kernel executions can overlap since there is no data dependency between the accessors.

The actual kernel gets enqueued to the command group in line 31 using the `sycl::handler::parallel_for` construct. This allows the kernel to be executed in parallel on the device specified during the `sycl::queue` construction. Another option would be the `sycl::handler::single_task` function, which results in only one invocation of the provided kernel. The first parameter to the `sycl::handler::parallel_for` handler is the number of work-items. The example in line 31 uses the `sycl::range` version, which allows the SYCL runtime to determine the used work-groups size automatically. Instead of `sycl::range`, the `sycl::ndrange` class can also be used as the first parameter. In this mode of execution, the kernel executes in work-groups of the specified size. This is useful since work-items within a work-group can share data in local memory and can be synchronized.

The second parameter is the actual kernel function. This can be a lambda function or a named function object. If the kernel function is not a named function object whose type is globally visible, a kernel name must be explicitly provided. If a lambda function represents the kernel, the lambda must capture by value.

In general, all user-defined types can be used in a SYCL kernel, as long as they satisfy the `C++11` standard layout requirements. This means `C++` constructs like templates are explicitly allowed in SYCL kernels. Pointers or references to host data cannot be used directly but can be passed to a kernel through the `sycl::accessor` class or an explicit copy of the underlying data.

In line 34, the actual vector addition takes place. Accessing the data for the calculation is only possible through the previously created `sycl::accessor`. To index the `sycl::accessor` based on the current work-item, the `sycl::item` class provides a `sycl::item::get_linear_id` function. In the case of a kernel invocation using the `sycl::ndrange` version, the kernel parameter must also be a `sylc::nd_item`. The `sycl::nd_item` class provides the `sycl::nd_item::get_global_linear_id` function to get the per work-item unique id or the `sycl::nd_item::get_local_linear_id` function to get the work-item id inside the parent work-group.

	ComputeCpp	hipSYCL	oneAPI	triSYCL	sycl-gtx
Intel CPUs	OpenCL[1]	OpenMP	OpenCL	OpenMP or TBB	all hardware supporting OpenCL 1.2[4]
AMD CPUs	-	OpenMP	-	OpenMP or TBB	
Intel GPUs	OpenCL	-	OpenCL	-	
AMD GPUs	-	HIP/ROCm	-	-	
NVIDIA GPUs	OpenCL + PTX[2]	CUDA	CUDA + PTX[3]	-	
ARM Mali GPUs	OpenCL	-	-	-	
Intel FGPAs	-	-	OpenCL	-	
Xilinx FGPAs	-	-	-	OpenCL + SPIR	

[1] SSE4.1 required
[2] experimental
[3] contribution from Codeplay to Intel's `DPC++` (experimental)
[4] discontinued

Table 4.1: Overview of SYCL implementations and their supported hardware. As the table shows, no SYCL implementation currently supports all types of hardware since sycl-gtx is discontinued.

4.2 Implementations

Currently, a variety of different SYCL implementations for the specification 1.2.1 exist, each focusing on other types of supported hardware and different backends.

ComputeCpp[1] is a SYCL implementation developed by Codeplay. It supports Intel CPUs and GPUs as well as ARM Mali GPUs using OpenCL as a backend. Additionally, ComputeCpp has experimental PTX support to be able to also work with NVIDIA GPUs. Upon request[2], it has been confirmed that ComputeCpp currently does not support AMD GPUs because of a lack of OpenCL SPIR support from the side of the recent AMD drivers.

Alpay and Heuveline [AH20] are developing hipSYCL[3] at the University of Heidelberg as another SYCL implementation. Supported platforms are CPUs via OpenMP, NVIDIA GPUs directly using CUDA, and AMD GPUs using HIP/ROCm. It currently is the only implementation not relying on OpenCL as a backend as proposed in the SYCL 1.2.1 specification. Therefore, it is the only implementation able to easily target AMD hardware.

[1] https://developer.codeplay.com/products/computecpp/ce/home/
[2] https://support.codeplay.com/t/questions-about-supported-sycl-targets/388/2
[3] https://github.com/illuhad/hipSYCL

OneAPI[4] is another SYCL implementation developed by Intel. It facilitates the custom compiler `DPC++` to target Intel hardware, i.e., Intel CPUs, GPUs, and FPGAs. However, NVIDIA GPUs are also supported using a contribution from Codeplay to `DPC++`, as confirmed upon request[5].

Mainly developed by Xilinx, triSYCL[6] states that it should not be used in production code since it is more like a testing ground for new features for upcoming SYCL specifications. Upon request, a triSYCL developer confirmed that it supports CPUs using OpenMP or Intel's TBB and Xilinx FPGAs.

Sycl-gtx[7] developed by Zuzek [Zuz16] as a Masters project uses OpenCL 1.2 as a backend.

An overview of the supported hardware for each SYCL implementation can be found in Table 4.1. As shown, currently no SYCL implementation can target all major hardware platforms.

In the result Chapter 6 only ComputeCpp, hipSYCL, and oneAPI will be compared.

[4] https://software.intel.com/content/www/us/en/develop/tools/oneapi.html
[5] https://community.intel.com/t5/Intel-oneAPI-Base-Toolkit/Intel-oneAPI-supported-targets/m-p/1217938#M646
[6] https://github.com/triSYCL/triSYCL
[7] https://github.com/ProGTX/sycl-gtx

5 Implementation

This chapter covers the implementation details of the `sycl_lsh` library created for this master thesis. The `sycl_lsh` library is implemented using the `C++17` standard.

The different memory layout types, Array of Structures (AoS) and Structure of Arrays (SoA), are explained first since using the wrong layout type can degenerate the performance. Therefore, the `sycl_lsh` library allows changing the memory layout using a simple non-type template parameter.

Next, an overview of the `sycl_lsh` library is given, including explanations of the most important classes shown in Figure 5.2.

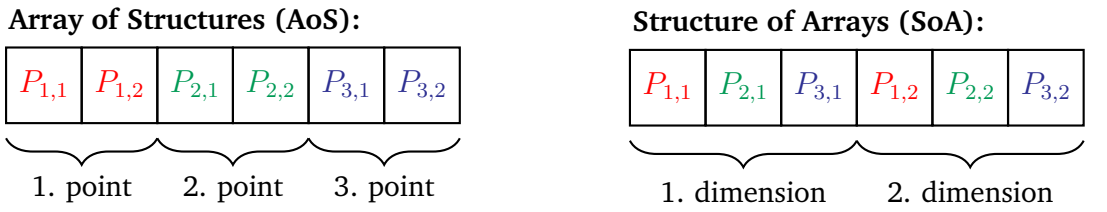
To support multiple GPUs, on possibly multiple compute nodes, the MPI framework is used. The SYCL specification 1.2.1 supports addressing multiple GPUs in a single host process. However, hipSYCL only supports a single GPU per process. Therefore, the `sycl_lsh` library spawns one MPI process per GPU, even on a single, shared memory node. The resulting implementation is shown in Section 5.3: “Multi-GPU Support”.

5.1 Memory Layout Types

Most data sets can be interpreted as a two-dimensional matrix, where the rows represent the data points and the columns the dimensions. However, since it is often more efficient, i.e., cache friendlier, to store large data sets linearly in memory, the question arises how to transform the two-dimensional data matrix to a one-dimensional memory layout.

The `sycl_lsh` library supports the two different memory layout types, Array of Structures (AoS) and Structure of Arrays (SoA). Figure 5.1 shows the differences between these two memory layout types. The AoS layout saves the data point-wise, i.e., all dimensions of the first point, followed by all dimensions of the second point, and so on. In contrast, the SoA layout saves the data dimension-wise, i.e., the first dimension of all points, followed by the second dimension.

Another difference shown in Figure 5.1 is the required indexing scheme for the data access. Given the requested point `i` and dimension `j` to retrieve the correct value for the AoS layout, the first `i` points must be skipped. Since AoS stores the data point-wise, skipping the first `i` points can be done using $\texttt{i} \cdot \texttt{DIMS}$, where `DIMS` is the total number of dimensions per data point. To get the same value in the SoA layout, `j` dimensions of each data point must be skipped. That can be achieved by using $\texttt{j} \cdot \texttt{SIZE}$, since the SoA format stores all data points dimension-wise.




```
// points saved in AoS layout
std::vector<float> points(SIZE * DIMS);

float get_aos(size_t i, size_t j) {
  return points[i * DIMS + j];
}
```

```
// points saved in SoA layout
std::vector<float> points(SIZE * DIMS);

float get_soa(size_t i, size_t j) {
  return points[j * SIZE + i];
}
```

Figure 5.1: Differences between the two memory layout types Array of Structures (AoS) and Structure of Arrays (SoA). The AoS layout saves the data point-wise, whereas the SoA format saves the data dimension-wise. This difference can also be seen in the different index calculations.

The `sycl_lsh` library enables these layout types using the enum class `sycl_lsh::memory_layout` with the possible values `sycl_lsh::memory_layout::aos` and `sycl_lsh::memory_layout::soa`. These values can be passed to the factory functions of the `sycl_lsh::data`, hash functions, and `sycl_lsh::hash_tables` classes as non-type template parameter.

As already stated in Figure 5.1, the indexing scheme depends on the used memory layout type. Since in the `sycl_lsh` library the layout type can be easily changed using a non-type template parameter, a way to index the data structures independently of the used memory layout must be provided. This is managed by using the templated functor `sycl_lsh::get_linear_id` together with partial template specialization. An example skeleton for the implementation of the `sycl_lsh::get_linear_id` functor can be seen in Listing 5.1. The idea is to specialize the `sycl_lsh::get_linear_id` functor in lines 6 and 7 based on the type for which the index should be calculated. The actual index calculation takes place in the overloaded function call operator (line 11). Inside this function, the index can be calculated based on the provided `sycl_lsh::memory_layout` type.

This functor can be used to index, for example, the a `sycl_lsh::data` class object given a `sycl_lsh::memory_layout` as shown in Listing 5.2. Here, the parameter `attrs` is of type `sycl_lsh::data_attributes`, which encapsulates information about the used data set, such as the total data set size or number of dimensions. For more information, see Section 5.2.2.

Listing 5.1 Templated functor to calculate the one-dimensional index, given a multi-dimensional one, for a class type based on the given `sycl_lsh::memory_layout`.

```
namespace sycl_lsh {
  enum class memory_layout { aos, soa };

  // specialize the templated functor based on the class type for which the index
  // given the memory layout type should be calculated
  template <memory_layout layout, ...>
  struct get_linear_id<...> {

    // overload the function-call operator to calculate the one-dimensional index
    [[nodiscard]]
    auto operator()(...) const noexcept {
      if constexpr (layout == memory_layout::aos) {
        // memory layout type is Array of Structures
        return ...;
      } else {
        // memory layout type is Structure of Arrays
        return ...;
      }
    }

  };
}
```

Listing 5.2 Usage example of the `sycl_lsh::get_linear_id` functor for a `sycl_lsh::data` class object. Only the used `sycl_lsh::data` type must be specified.

```
namespace sycl_lsh {
  // data class type
  template <sycl_lsh::memory_layout, typename Options>
  class data;
}

// example usage
using options_type = sycl_lsh::options<...>;
using data_type = sycl_lsh::data<sycl_lsh::memory_layout::soa, options_type>;
const sycl_lsh::get_linear_id<data_type> get_linear_id_soa;
const auto idx = get_linear_id_soa(point, dim, attrs);
// use idx to access elements of buffers contained in the data class
```

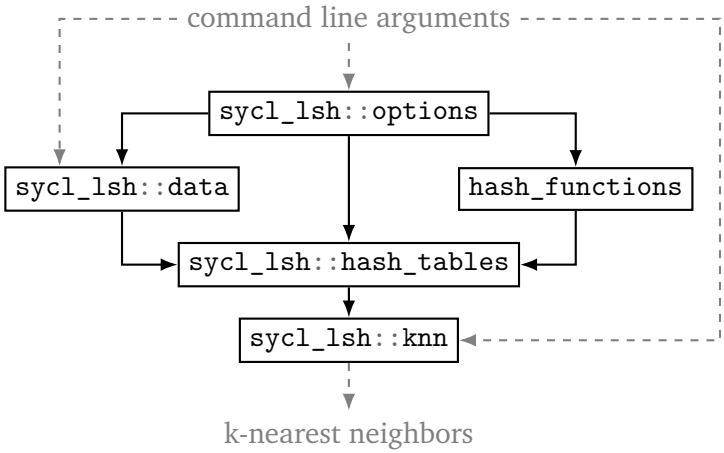


Figure 5.2: Simplified code architecture of the `sycl_lsh` library.

5.2 Code Architecture

This section explains the key classes of `sycl_lsh` library implemented for this master thesis. The simplified code architecture can be seen in Figure 5.2. The `sycl_lsh::options` class controls the used algorithm's behavior utilizing, for example, provided command line arguments. It encapsulates options like the used floating-point type or the number of hash tables or hash functions used in the LSH algorithm. The `sycl_lsh::data` class represents the used data set. It reads the data from a file provided through command line arguments. In addition, it uses the `sycl_lsh::options` class to determine the used data types. In subsection 5.2.3 the implementation of the used locality-sensitive hash functions, `sycl_lsh::random_projections` and `sycl_lsh::entropy_based` hash functions, are discussed. Again, the `sycl_lsh::options` class determines the used data types and the used hash function type through template parameters. The `sycl_lsh::hash_tables` class represents the actual LSH hash tables. It receives a previously created `sycl_lsh::data` object, and constructs the hash functions based on the provided `sycl_lsh::options`. Afterward, the `sycl_lsh::hash_tables` object can be used to calculate the k-NN of all data points. The computed k-NN are returned using a `sycl_lsh::knn` object. This object can be used to calculate the achieved recall and error ratio. Additionally, the calculated k-NN and their distances can be saved to files provided through command line arguments. Other components of the `sycl_lsh` library are mainly helper functions or RAII wrapper classes around MPI objects.

The following sections introduce each of the `sycl_lsh` components, their respective purpose, and their note-worthy attributes in detail.

Listing 5.3 Example for the creation of a `sycl_lsh::options` object with explicitly specified compile-time parameters.

```
using options_type = sycl_lsh::options<
    float,                                                  // real_type
    std::uint32_t,                                          // index_type
    std::uint32_t,                                          // hash_value_type
    10,                                                     // blocking_size
    sycl_lsh::hash_functions_type::random_projections>; // used_hash_functions_type

const options_type opt;
```

5.2.1 The `sycl_lsh::options` Class

The `sycl_lsh::options` class is the central point that controls the behavior of the used LSH algorithm. It is divided into compile-time and runtime options.

Compile-time options, which must be specified as template parameters during the creation of a `sycl_lsh::options` object, are:

real_type
: The used floating point type for the data and hash functions. The type must fulfill the `C++11` type trait `std::is_floating_point`.

index_type
: The used integral type for indexing and index calculations. The type must fulfill the `C++11` type trait `std::is_integral`.

hash_value_type
: The used unsigned integer type for the calculation of the hash values. The type must fulfill the `C++11` type trait `std::is_unsigned` since, before `C++20`, for a negative `x` the behavior of `x << y` is undefined and the value of `x >> y` is implementation defined.

blocking_size
: The blocking size of type `index_type` used in the SYCL kernel for calculating the k-NN. The value must be greater than zero.

used_hash_functions_type
: The type of the used hash functions. The value must be either `sycl_lsh::hash_functions_type::random_projections` or `sycl_lsh::hash_functions_type::entropy_based`.

For example, creating a `sycl_lsh::options` object could look like in Listing 5.3. In this example the used data values are interpreted as `float`, the type for index and hash value calculations is `std::uint32_t`, the blocking size is ten, and the used locality-sensitive hash functions are random projections.

The other type of options are the runtime options. If the command line argument `--options_file path-to-file` is present, the `sycl_lsh` library tries to read the options given by the provided file. After that, the options are overwritten by the present command line arguments. All options that are not initialized using the file or command line will be initialized to their default value. If further adjustments must be made, the options can easily be changed using a simple member assignment later on. The runtime options are:

hash_pool_size
: The number of hash functions in the hash pool. They are used to construct the hash signature functions, as described in Section 3.2.2. To generate a single hash signature function `num_hash_functions` many hash functions will be drawn uniformly at random from the hash pool. The value must be greater than zero, and the default is 32.

num_hash_functions
: The number of hash functions per hash signature used to index the LSH hash tables. This means that `num_hash_functions * num_hash_tables` hash functions, will be drawn from the hash pool. More hash functions mean that it is more likely that fewer data points have the same hash signature. This results in a lower recall, but possibly better performance since fewer points must be considered per hash bucket. The value must be greater than zero, and the default is twelve.

num_hash_tables
: The number of LSH hash tables. More hash tables mean that more data points are considered since more hash buckets will be examined. This results in a higher recall but possibly worse performance. The value must be greater than zero, and the default is eight.

hash_table_size
: The size of each hash table. The value determines the maximum possible hash value for a hash table using `hash_value % hash_table_size`. A smaller hash table size means that more data points have the same hash value due to the modulus operator's nature. This results in a higher recall but possibly worse performance. The value must be greater than zero and should be a prime number. The default value is $105\,613$.

w
: The segment size for the random projection hash functions: $h(x) = \left\lfloor \frac{a \cdot x + b}{w} \right\rfloor$. This value is only used if the `used_hash_functions_type` is `sycl_lsh::hash_functions_type::random_projections`. Small values of w mean that each segment is wider. Since the hash value of a single hash function is determined by the projected segment, it is more likely that more data points have the same hash value resulting in a better recall but possibly worse performance. The value must be greater than zero, and the default value is 1.0.

num_cut_off_points
: The number of cut-off points for the entropy-based hash functions. This value is only used if the `used_hash_functions_type` is `sycl_lsh::hash_functions_type::entropy_based`. A higher number of cut-off

points mean more segments, resulting in fewer data points per segment. This results in a lower recall but possibly better performance. The value must be greater than zero, and the default value is six.

The current options can be saved to the file specified by the command line argument `--options_save_file path-to-file` using the `sycl_lsh::options::save` function.

5.2.2 The `sycl_lsh::data` Class

The `sycl_lsh::data` class represents the used data set.

It reads the provided data file (via command line arguments; see Appendix B for more information) and parses it using the specified file parser utilizing MPI IO. The default file parser, the `sycl_lsh::mpi::binary_parser`, expects the file in binary form, whereby the first line contains the total number of data points, the second line the number of dimensions, and the following lines the actual data points. Thereby, both sizes must be saved using a type compatible with the `sycl_lsh::options::index_type` and the data points compatible with the `sycl_lsh::options::real_type`. The `sycl_lsh::mpi::arff_parser` expects the file saved in the Attribute-Relation File Format (ARFF)[1]. Since the ARFF format is a textual format and MPI IO expects the files to be stored in binary form, this parse is currently not implemented. More file parser can be implemented by inheriting from the `sycl_lsh::mpi::file_parser` base class.

The file parser expects the data to be given in AoS format. However, if the requested memory layout type is `sycl_lsh::memory_layout::soa`, the data will be converted accordingly. Since this only requires a single pass over all data points, resulting in a complexity of $\mathcal{O}(n)$ where n is the size of the data set, it is insignificant for the total runtime.

The `sycl_lsh::data` class holds an instance of the `sycl_lsh::data_attributes` class, which represents the attributes of the data set. A `sycl_lsh::data_attributes` object holds three members. The `sycl_lsh::data_attributes::total_size` corresponds to the total number of data points in the data set. In contrast, the `sycl_lsh::data_attributes::rank_size` represents the number of data points per MPI process. If the total data set size is not divisible by the number of MPI processes, the last MPI rank gets the necessary dummy points such that all ranks are responsible for the same number of data points. Additionally, the `sycl_lsh::data_attributes::dims` corresponds to the number of dimensions of the data set.

Internally the `sycl_lsh::data` class holds two one-dimensional arrays, each of size `sycl_lsh::data_attributes::rank_size * sycl_lsh::data_attributes::dims`. The arrays are one-dimensional since a `std::vector<std::vector<T>>` is not guaranteed to be laid out consecutively in memory, which would result in additional unnecessary indirections. One array represents the data residing on the device and is of type `sycl_lsh::buffer`. The other array resides in the host buffer. This array is used to overlap

[1] https://waikato.github.io/weka-wiki/formats_and_processing/arff_stable/

MPI communications and device calculations. Therefore, in total, a `sycl_lsh::data` object uses `sycl_lsh::data_attributes::rank_size * sycl_lsh::data_attributes::dims * sizeof(sycl_lsh::options::real_type)` bytes both on the device and the host buffer.

The `sycl_lsh::data` class specializes the `sycl_lsh::get_linear_id` functor and, therefore, can easily be used with different memory layout types.

5.2.3 Locality-Sensitive Hash Functions

The `sycl_lsh` library currently implements two types of hash functions, `sycl_lsh::random_projections` and `sycl_lsh::entropy_based`. The hash function type can be set using the non-type template parameter `sycl_lsh::options::used_hash_functions_type` in the `sycl_lsh::options` class.

Both hash function classes specialize the `sycl_lsh::get_linear_id` functor to be able to use the different memory layout types.

In the following, the distributed generation of each hash function type is described, followed by explaining how the hash signature for a single LSH hash table is calculated.

The `sycl_lsh::random_projections` Hash Functions Class

Generating the random projection hash functions is simple compared to the entropy-based hash functions. Recall the random projection hash functions defined in Definition 3.2.3:

$$h(x) = \left\lfloor \frac{a \cdot x + b}{w} \right\rfloor \qquad x, a \in \mathbb{R}^d, w \in \mathbb{R}_+, b \in [0, w]. \tag{5.1}$$

At first, the hash pool must be created by generating `sycl_lsh::options::hash_pool_size` many hash functions on the host side. The vector x, a data point for which the hash value should be calculated, and the scalar w, a hyperparameter provided, for example, on the command line, are given. Therefore, to generate a random projection hash function, only the vector a and the scalar b must be determined. To create the vector a, `sycl_lsh::data_attributes::dims` many random values are drawn from a `std::normal_distribution<sycl_lsh::options::real_type>`. Additionally, the scalar b is drawn from a `std::uniform_real_distribution<real_type>` restricted to values in the range `[0, sycl_lsh::options::w]`.

In a second step, `sycl_lsh::options::num_hash_tables * sycl_lsh::options::num_hash_functions` hash functions are selected uniformly at random from the previously generated pool, to form the hash signatures used for filling the LSH hash tables.

All hash functions are calculated on the MPI master rank, i.e., the hash function pool only exists on the MPI master rank. After selecting the actual hash functions, those functions are

broadcasted to all other MPI ranks. Thereafter, each MPI rank copies the hash functions to its respective device using a `sycl_lsh::buffer`.

In total, a `sycl_lsh::random_projections` object uses `sycl_lsh::options::num_hash_tables * sycl_lsh::options::num_hash_functions * (sycl_lsh::data_attributes::dims + 1) * sizeof(sycl_lsh::options::real_type)` bytes on the device buffer.

The `sycl_lsh::entropy_based` Hash Functions Class

Calculating the entropy-based hash functions is more expensive compared to the random projection hash functions. Recalling the entropy-based hash functions defined in Definition 3.2.1, their generation can be split into two steps, creating the initial mapping values and calculating the cut-off points. The initial mapping is calculated using:

$$h'(x) = x \cdot a \qquad x, a \in \mathbb{R}^d. \tag{5.2}$$

The vector x, the data point for which the hash value should be calculated, is given. Therefore, only the vector a must be determined to calculate the initial mapping values. To create the vector a, `sycl_lsh::data_attributes::dims` many random values are drawn from a `std::normal_distribution<sycl_lsh::options::real_type>`. Since the `sycl_lsh::options::hash_pool_size` many mapping functions are only generated on the MPI master rank, they are broadcasted to all other MPI ranks. On every MPI rank, these hash functions $h'(x)$ are used to calculate the initial mappings on the device using a SYCL kernel.

The cut-off points for the final hash function

$$h(x) = \begin{cases} 0, & \text{if} \quad h'(x) \leq q_1 \\ 1, & \text{if} \quad q_1 < h'(x) \leq q_2 \\ \quad \vdots & \\ r-2, & \text{if} \quad q_{r-2} < h'(x) \leq q_{r-1} \\ r-1, & \text{if} \quad h'(x) > q_{r-1} \end{cases} \tag{5.3}$$

must be calculated in the second step. First, the initial mapped values must be sorted across all MPI ranks. Since this is currently done by using a modified distributed bubble-sort algorithm, implementing a more efficient distributed sorting algorithm can improve the hash pool generation's performance.

The cut-off points are determined by first calculating the indices of the cut-off points, which are `((sycl_lsh::data_attributes::rank_size * MPI_WORLD_SIZE) / sycl_lsh::options::num_cut_off_points) * (i + 1)` with `i = 1, ..., sycl_lsh::options::num_cut_off_points`. Afterward, the indices are used to determine the cut-off point values on the respective MPI rank. Each MPI rank

Listing 5.4 Hash combine function in the case of `std::uint32_t` as `sycl_lsh::options::hash_value_type`. This function is used to combine a single hash value with hash signature.

```
1 namespace sycl_lsh::detail {
2   inline
3   std::uint32_t hash_combine(const std::uint32_t seed, const std::uint32_t val) noexcept
4   {
5         return seed ^ (val + static_cast<std::uint32_t>(0x9e3779b9U)
6                            + (seed << static_cast<std::uint32_t>(6))
7                            + (seed >> static_cast<std::uint32_t>(2)));
8   }
9 }
```

collects the cut-off point values at the indices, for which it is responsible. The resulting values are combined and broadcasted to each MPI rank. For example, given 25 mapped values per MPI rank, four MPI ranks, and three cut-off points, the indices are 33 and 66. Therefore, only the MPI ranks two and three can report the cut-off point values. However, these values are then combined and broadcasted such that all four MPI ranks have both cut-off point values.

As the last step, `sycl_lsh::options::num_hash_tables * sycl_lsh::options::num_hash_functions` hash functions are selected uniformly at random on the MPI master rank from the previously generated pool and are broadcasted to all other MPI ranks. After that, each MPI rank copies the hash function to its respective device using a `sycl_lsh::buffer`.

In total, a `sycl_lsh::entropy_based` object uses `sycl_lsh::options::num_hash_tables * sycl_lsh::options::num_hash_functions * (sycl_lsh::data_attributes::dims + sycl_lsh::options::num_cut_off_points - 1) * sizeof(sycl_lsh::options::real_type)` bytes on the device buffer.

Hash Signature Calculation

To calculate the hash signature depending on the hash function type, a similar approach to the `sycl_lsh::get_linear_id` functor is used. In this case, the templated functor is named `sycl_lsh::lsh_hash` and uses the `operator()` overload to perform the actual hash signature calculation. This templated struct is specialized for both hash function types, `sycl_lsh::random_projections` and `sycl_lsh::entropy_based`.

To calculate the hash value of a data point for a single LSH hash table, `sycl_lsh::options::num_hash_functions` hash functions must be evaluated based on the hash function type. These hash functions must be combined to form the resulting hash signature used to index the hash table. It is inefficient to calculate all hash values and then combine them to the hash signature in one step. Therefore, the

hash signature is calculated on the fly and gets updated after each hash value calculation. This combination of a hash value with the hash signature is shown in Listing 5.4. The code shows the `sycl_lsh::detail::hash_combine` function in case of a `sycl_lsh::options::hash_value_type` of type `std::uint32_t`. Other overloads exist for `std::uint16_t` and `std::uint64_t` with different magic numbers. These functions are based on the hash combination method proposed by Josuttis [Jos18]. The magic number `0x9e3779b9` is derived from the golden ratio as described by Jenkins [Jen96] and depends on the number of bits in the used `sycl_lsh::options::hash_value_type`.

$$\Phi = \frac{1+\sqrt{5}}{2} \tag{5.4}$$

$$trunc(\frac{2^{32}}{\Phi}) = 2654435769 \tag{5.5}$$

$$2654435769_{10} = 0x9e3779b9_{16} \tag{5.6}$$

The final hash value is derived from the hash signature using the modulo operator to reduce the number of hash buckets to `sycl_lsh::options::hash_table_size`.

5.2.4 The `sycl_lsh::hash_tables` Class

This section describes the creation of a `sycl_lsh::hash_tables` object. Two important aspects must be considered to ensure reasonably good performance.

Normally, perfect hash functions try to achieve as few hash collisions as possible. A hash collision occurs if two different data points have the same hash value. Too many collisions can result in a degeneration of performance since dealing with them can be rather expensive. In LSH, however, if two data points are close to each other, they will have the same hash value with a high probability. Therefore in LSH, many hash collisions can occur, and the hash tables' implementation must take that into account.

Another problem can be race conditions. Because the hash tables are filled in parallel, multiple threads may try to write to the same memory location. To prevent race conditions, such memory accesses must be synchronized in one way or another. If there are no efficient ways to synchronize memory accesses, performance will degenerate with an increasing number of collisions.

To guarantee that the previously stated problems have no negative impact on the performance of the creation of a `sycl_lsh::hash_tables` object on the specified device, the hash tables are created in three steps. These steps can also be seen in Figure 5.3 and are further described in the next subsections.

The resulting `sycl_lsh::hash_tables` object consist of two `sycl::buffer`. The `hash_tables_buffer` represents all hash tables. Each hash table contains each data point exactly once. Since it is too expensive to store the data points directly, only their ids are stored. All hash tables are laid out consecutive in memory, whereby all ids are sorted by their respective hash value. This means that all data points

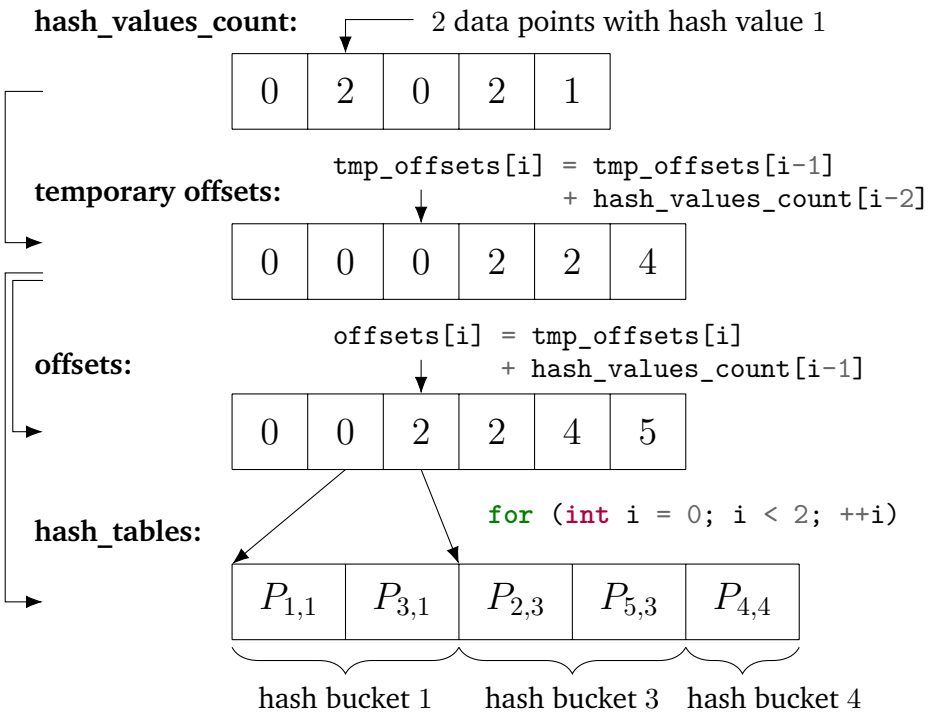


Figure 5.3: The steps for creating the necessary data structures for the `sycl_lsh::hash_tables` class. All three steps are performed on the current device using different SYCL kernels. The points P_i should be inserted into their respective hash buckets. To do that, the occurrence of each hash value must be determined. After that, a temporary offsets array is created using the `hash_values_count` array. This temporary array is then used to insert the data points into their respective hash buckets and simultaneously updated to the final `offsets` array. In the resulting `hash_tables` $P_{i,j}$ denotes that the i-th data point has been inserted into the j-th hash bucket.

with corresponding hash value zero are stored first, followed by all with hash value one, and so on. This guarantees efficient access for each data point id inside a hash bucket. The `hash_tables_buffer` uses `(sycl_lsh::options::num_hash_tables * sycl_lsh::data_attributes::rank_size + sycl_lsh::options::blocking_size) * sizeof(sycl_lsh::options::index_type)` bytes on the device. The additional `sycl_lsh::options::blocking_size` bytes are needed to enable blocking in the SYCL kernel to calculate the k-NN.

The `offsets_buffer` is used to create the hash tables and to index a specific hash bucket of a hash table efficiently. It consumes `sycl_lsh::options::num_hash_tables * (sycl_lsh::options::hash_table_size + 1) * sizeof(sycl_lsh::options::index_type)` bytes on the device buffer.

The next sections will describe how the hash tables are filled in parallel, followed by an explanation on how the *k*-nearest neighbors are calculated.

Creating the Hash Tables

The steps performed to create the LSH hash tables can be seen in Figure 5.3. The data points should be laid out consecutively in memory, sorted by hash buckets. Therefore, to insert a data point in hash bucket i, the number of data points in all hash buckets $j < i$ must be known. In the first step seen in Figure 5.3, the number of data points per hash bucket per hash table is calculated. The corresponding hash values are calculated in parallel on the device using a SYCL kernel for each data point. These parallel updates of the hash value counts can result in race conditions if no synchronization takes place. Synchronizing a `sycl::buffer` access using atomic operations, such as `fetch_add(1)`, is done using a `sycl::accessor` with `sycl::access::mode::atomic` as access mode. Afterward, if accessing the `hash_values_count` buffer at position one results in value two, two data points will be sorted into the hash bucket one. This `hash_values_count` buffer consumes `sycl_lsh::options::num_hash_tables * sycl_lsh::options::hash_table_size * sizeof(index_type)` bytes on the device. Since this buffer is only needed during the first two steps of the hash table creation, the memory will be freed afterward.

In the second step, the `hash_value_count` buffer is used to calculate the temporary `offsets_buffer`. This buffer is later used to fill the `hash_tables_buffer` and, in this step, gets transformed into the final `offsets_buffer`. The temporary `offsets_buffer` is filled using a modified prefix-sum algorithm on the device using a SYCL kernel. As shown in Figure 5.3, the i-th value in the temporary `offsets_buffer` is calculated using `offsets_buffer[i] = offsets_buffer[i - 1] + hash_values_count[i - 2]`. This modified algorithm shifts the values by two to the right side. The first shift is needed to correctly index the first hash bucket elements, which start at index zero. The second shift is needed for the third step of the hash table creation to determine a data point's actual position inside its respective hash bucket.

In the third step, the `hash_tables_buffer` is filled, and the temporary `offsets_buffer` is updated to the final `offsets_buffer`. For each data point, the hash values are calculated in parallel on the device using a SYCL kernel. The id of the current data point is inserted at the position `acc_offsets_buffer[hash_value + 1].fetch_add(1)` into the `hash_tables_buffer`. Again, an atomic operation is used to prevent race conditions if two data points are inserted into the same hash bucket simultaneously. In the same step, the temporary `offsets_buffer` is updated to support efficient per hash bucket access. Figure 5.3 may clarify this. For example, the value of the final `offsets_buffer` at position two can be calculated by `hash_values_count[1] + offsets_buffer[2]`.

To access all elements in the i-th hash bucket of the j-th hash table using the final `offsets` buffer, the code snippet in Listing 5.5 can be used. The start of the hash bucket is directly determined by indexing the `offsets` buffer with the calculated hash signature i. The end of the inspected hash bucket corresponds the the value at the next position in the `offsets` buffer. An example is shown in Figure 5.3. Iterating over all elements in the hash bucket one of the first hash table can be done by using the displayed for-loop. This for-loop indeed iterates over the necessary two data points in the hash bucket one.

Listing 5.5 Example code to access all elements of the i-th hash bucket of the j-th hash table using the `offsets` buffer.

```
const options_type opt = ...;
using index_type = typename options_type::index_type;

const index_type bucket_begin = acc_offsets[j * (opt.hash_table_size + 1) + i];
const index_type bucket_end   = acc_offsets[j * (opt.hash_table_size + 1) + i + 1];
for (index_type elem = bucket_begin; elem < bucket_end; ++elem) {
  // perform calculations on the points in the hash bucket
}
```

Calculating the k-Nearest Neighbors

After all hash tables have been created, the k-NN can be calculated, which is the same on all MPI ranks.

At first, the current k-NN and their respective distances are moved from the host side to the device memory using two `sycl::buffer`.

The k-NN search is performed in parallel for all data points on the SYCL device. Every data point is treated independently from all other points. For each data point, all hash tables are iterated. Firstly, the hash signature of the current data point x in the current hash table is calculated. All data points in the hash bucket corresponding to the hash signature are considered for the k-NN search. Therefore, the distances between these data points and x are calculated. Currently, only the Euclidean distance metric is implemented, whereby the square root is omitted, since it would not change the resulting nearest neighbor order but is computational expensive.

Afterward, the current k-NN are updated. It is necessary to ensure that a data point only occurs once as nearest neighbor, and the current point x is not marked as a nearest point to itself. Without special guarantees, a data point could occur multiple times as a nearest neighbor if it assigned to the same hash bucket as x in multiple hash tables.

To update the k-NN, both buffers, the nearest neighbor ids and distances, must be changed. Since both buffers are sorted by decreasing distance, only the first element of the distance buffer must be examined to determine if the new data point is a better nearest neighbor. If this is the case, the first entries of both buffers are updated. Afterward, a single bubble-sort pass over the distance buffer is performed while simultaneously updating the nearest neighbor id buffer, too. This bubble-sort pass ensures that the updated nearest neighbor is moved to the correct place and the next largest distance is at the first position of the buffers, guaranteeing that the buffers' invariant still holds.

After the destruction of the `sycl::buffer` at the end of their lifetimes, the calculated k-NN are automatically transferred back to the host memory.

Without special care, both k-NN buffers reside in the global memory. In the update step, possibly multiple accesses to both buffers are made. However, accessing global memory is expensive. Therefore, both buffers are loaded into local memory to speed

Listing 5.6 Comparison between the loops for accessing all data points in a specific hash bucket with and without blocking. When using blocking, the outer loop increments each iteration by `sycl_lsh::options::blocking_size`, while the inner loop increments by one up to the blocking size.

```
1 // Normal iteration over all elements of a hash bucket
2 for (index_type elem = bucket_begin; elem < bucket_end; ++elem) {
3   ...
4 }
5
6 // Iterate over all elements with blocking
7 constexpr auto blocking_size = sycl_lsh::options::blocking_size;
8 for (index_type elem = bucket_begin; elem < bucket_end; elem += blocking_size) {
9   ...
10  for (index_type i = 0; i < blocking_size; ++i) {
11    ...
12  }
13  ...
14 }
```

up that accesses. The local memory is a memory area specific to a single work-group. It can be thought of as a user-controlled cache and can be accessed more efficiently than the global memory. SYCL enables accessing the local memory using a special access target, `sycl::accessor<type, 1, sycl::access::mode::read_write, sycl::access::target::local> acc(sycl::range<>(local_memory_size), cgh)`. Since the `local_memory_size` must be specified per work-group, they must be created manually using a `sycl::nd_range` as execution range. This `sycl::nd_range` consists of a `global_size` and `local_size`. The `local_size` represents the number of threads per work-group and is determined as the largest power of two smaller than the maximum work-group size that does not exceed the available local memory size on the device. The `global_size` is the smallest multiple of the `local_size` that is greater or equal than the `sycl_lsh::data_attributes::rank_size`. For calculating the k-NN, the `local_memory_size` equals `local_size * k`. Since both buffers must be loaded into local memory, the occupied local memory per work-group is `local_size * k * (sizeof(sycl_lsh::options::index_type) + sizeof(sycl_lsh::options::real_type))`. The data must be loaded manually from the global memory into the local memory at the beginning of the kernel. At the end of the kernel, the data must be loaded back from local memory to global memory.

Another optimization is the utilization of blocking. For this purpose, the compile-time non-type template parameter `sycl_lsh::options::blocking_size` for the `sycl_lsh::options` class has been introduced. Using blocking, the distance calculation and nearest neighbors updates are not performed one by one, but in batches of `sycl_lsh::options::blocking_size` by using a private, constant sized array inside the SYCL kernel.

An example of a for-loop using blocking is displayed in Listing 5.6. The loop counter of the outer loop in line 8 is incremented by `sycl_lsh::options::blocking_size` instead one

as for the non-blocking for-loop in line 2. The inner loop's counter is then incremented by one in line 10 up to the blocking size.

Since the number of data points per hash bucket is not necessarily dividable by the blocking size, some type of padding must be applied. However, this padding is not needed for each hash bucket since reading beyond one hash bucket accesses the next hash bucket's first element. Only reading past the last hash bucket of the last hash table would result in an out-of-bounce memory access. Therefore, it is sufficient to add the padding of size `sycl_lsh::options::blocking_size` once at the end of the `hash_tables_buffer`.

5.2.5 The `sycl_lsh::knn` Class

The `sycl_lsh::knn` class represents the result of the *k*-nearest neighbors search. It consists of two buffers residing in the host memory. One buffer holds the k-NN point ids using a `std::vector<index_type>`. The other buffer represents the k-NN distances using a `std::vector<real_type>`. In total, a `sycl_lsh::knn` object uses `k * sycl_lsh::data_attributes::rank_size * (sizeof(sycl_lsh::options::index_type) + sizeof(sycl_lsh::options::real_type))` bytes on the host memory.

The calculated k-NN and the corresponding distances can be saved using the `sycl_lsh::knn::save_knns`, respectively `sycl_lsh::knn::save_distances` functions.

The computed k-NN can be evaluated using two functions. The `sycl_lsh::knn::recall` function calculates the recall using the ratio

$$\frac{\text{true positives}}{\text{relevant elements}}. \tag{5.7}$$

Verbalized, it calculates the amount of the exact k-NN that were correctly found. The function uses the correct nearest neighbors provided in the file given by the `--evaluate_knn_file path-to-file` command line argument. This evaluation metric is a hard metric. The calculated k-NN could be near or close to the correct neighbors but not identical, resulting in a low recall. However, sufficiently close neighbors could still be enough for an approximate algorithm. Therefore, a second evaluation metric has been implemented.

The `sycl_lsh::knn::error_ratio` function calculates the relative distance error between the correct k-NN distances and the calculated k-NN distances using

$$\frac{1}{N} \cdot \sum_{i=1}^{N} \left(\frac{1}{k} \cdot \sum_{j=1}^{k} \frac{dist_{LSH_j}}{dist_{correct_j}} \right). \tag{5.8}$$

This metric is more suited for an approximate algorithm since it takes into account that small differences in the resulting distances can still be good enough for approximate algorithms. The function uses the correct nearest neighbor distances provided in the file given by the `--evaluate_knn_dist_file path-to-file` command line argument. If for

any given data point no k k-NN could be found, this data point is excluded from the error ratio calculation. Instead, the function returns two additional values besides the error ratio, one being the number of data points for which less than k nearest neighbors were found, and the other being the total number of nearest neighbors that could not be found.

5.3 Multi-GPU Support

Because of the large size of current data sets, a single GPU device can easily run out of memory. Therefore, it is essential to use multiple devices to fulfill the growing demand of necessary memory. In addition, even if the data sets do not grow in size, more devices allow for faster executions since more compute power and memory bandwidth is available. Because of that, the `sycl_lsh` library can utilize multiple devices to speed up the computation, or to allow the usage of larger data sets.

Everything discussed in the last chapter applies to a single MPI process if not stated differently. Therefore, only two things remain to be clarified, namely how to select only a single device per MPI rank, and the communication scheme between the MPI ranks.

5.3.1 Selecting a SYCL Device

At some point, the SYCL runtime must select the devices used for execution. The SYCL standard natively supports handling multiple devices per host thread using multiple `sycl::queues`. However, hipSYCL currently does not support multiple devices due to its runtime limitations[2]. Therefore, the `sycl_lsh` library needs another way to support multiple devices.

Per GPU device, a single MPI process is spawned. The environment variable `CUDA_VISIBLE_DEVICES` for NVIDIA GPUs, respectively `HIP_VISIBLE_DEVICES` for AMD GPUs, is used to determine the device based on the provided CMake configuration variable `SYCL_LSH_TARGET` (see Appendix A for more information). Setting `CUDA_VISIBLE_DEVICES`, for example, to 0 means that only the CUDA device with device id zero is visible for the remainder of the application. This is used together with the MPI rank to enable only a single device per MPI process. At first, the `MPI_COMM_WORLD` communicator, containing all spawned processes, is split using `MPI_COMM_TYPE_SHARED`. The resulting node communicators contain only the MPI ranks located on the respective shared memory node. The environment variable `CUDA_VISIBLE_DEVICES` is then set according to the MPI rank in the node communicators.

SYCL itself selects the device using the `sycl::device_selector` class's `select_device` member function. This function, in turn, calls the `sycl::device::operator()(sycl::device)` for each available SYCL device and returns

[2] https://github.com/illuhad/hipSYCL/issues/272#issuecomment-654243168

the device with the highest score. A device with a negative score will never be used. Because of the environment variable `CUDA_VISIBLE_DEVICES`, only a single GPU device will be listed. Therefore, it is sufficient inside the `sycl::device::operator()(sycl::device)` member function to check whether the given `sycl::device` is a GPU device from the provided vendor. In all other cases, a negative score will be returned.

Currently, only a single Intel GPU device can be select since no environmental variable like `CUDA_VISIBLE_DEVICES` exists for Intel GPUs. A similar mechanism could be implemented using the DPCT namespace[3]. However, due to encountered problems[4] regarding the DPCT namespace on Intel's devcloud currently no such implementation exists for the `sycl_lsh::library`.

5.3.2 MPI Communication Scheme

As already mentioned in the last section, the `sycl_lsh` library implements the multi-GPU support in a way that each MPI rank uses a single distinct GPU. To calculate the k-NN, the MPI ranks must communicate with each other.

Each MPI rank reads its part of the data from the provided file using MPI IO and stores it in a `sycl_lsh::data` object. In the same step, the data gets loaded onto the device. Afterward, the hash functions are created. For more information about possible MPI communication while creating the hash functions, see Section 5.2.3. The hash tables are created on each MPI rank separately using only its part of the data read from the data file.

Therefore, before the k-NN calculation starts, the situation is as follows: the read data resides once on the device and once in the host memory. Additionally, the hash functions and hash tables are located on the device, whereas the current k-NN are still located in the host memory.

The k-NN calculation is split into as many rounds as MPI ranks exist. In the first round, the preliminary k-NN are calculated using the data already residing on the GPU. Simultaneously each MPI rank sends its data to the next MPI rank and receives new data from the previously MPI rank using `MPI_Sendrecv_replace`. This simultaneously sending is executed in another `std::thread`. Therefore, the MPI environment must be initialized with the `MPI_Init_thread` function with a level of thread support of `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`. This overlapping of computation and communication reduces the total runtime compared to purely sequential execution. After the preliminary k-NN have been calculated, they are also sent to the next MPI rank, and other preliminary k-NN are received from the previous one.

In the second round, the received data and k-NN are loaded onto the GPU, and the preliminary k-NN are updated. This ring-like send and receive pattern is repeat `MPI_COMM_WORLD`

[3] https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top/dpct-namespace-usage-guide.html
[4] https://community.intel.com/t5/Intel-DevCloud/Multi-GPU-run/m-p/1226560

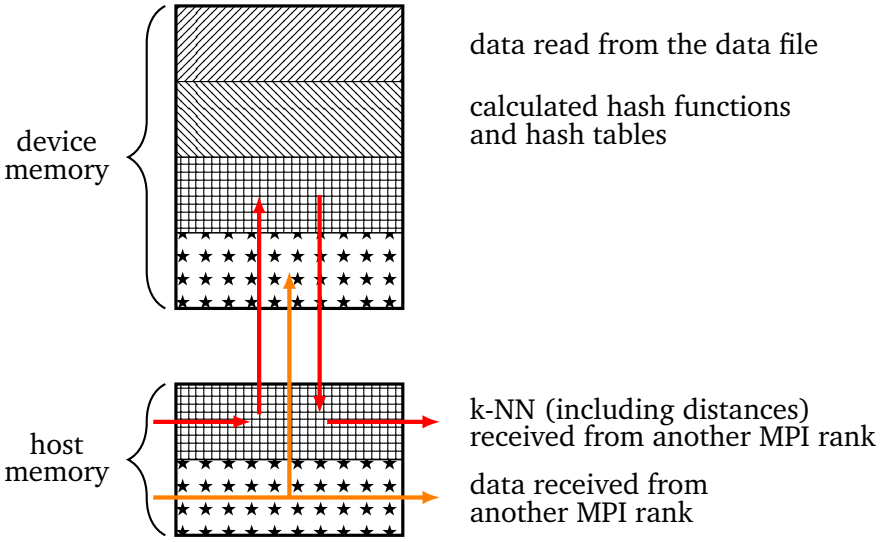


Figure 5.4: The held and communicated data per MPI rank. To calculate the final k-NN multiple communication and calculation rounds are performed. Once loaded onto the GPU, the data read from the data file, and the calculated hash functions and hash tables stay in the GPU memory. In each round, a MPI rank receives the next portion of the full data set from the previous MPI rank, loads it onto the GPU, and simultaneously sends it to the next MPI rank using a ring-like communication pattern. The same happens for the already calculated k-NN, however, they are sent to the next rank only after the calculations on the GPU for the current round have been finished.

size times. Afterward, the k-NN calculation is finished, and the recall or error ratio can be calculated.

An example of a single round, except the first round, is shown in Figure 5.4. The data read from the data file, and the calculated hash functions and hash tables reside on the device during the whole k-NN calculation. The data and preliminary k-NN for the current round are received in the host memory and then loaded onto the device. While the k-NN can only be sent to the next MPI rank after the current round is finished, the data is sent directly to the next MPI rank after being copied to the device.

6 Results

This chapter presents and explains the results obtained by the implementation discussed in the last chapter.

At first, the utilized hardware and data sets are described.

The next section describes the runtime characteristics of the `sycl_lsh` library on a single GPU. For this purpose, the three SYCL implementations, ComputeCpp, hipSYCL, and oneAPI, are compared to each other also utilizing different hardware showing the portability of the `sycl_lsh` library across SYCL implementations and hardware. Afterward, the effects of the various runtime parameters described in Section 5.2.1 are presented. As metrics for the achieved accuracy of the calculated k-NN, the recall and error ratio as described in Section 5.2.5 are used. Since these parameters heavily influence the resulting runtime, recall, and error ratio, it is important to know how to tune these parameter to achieve the required recall or error ratio while maintaining acceptable performance. While the first part of the section investigates the random projections as locality-sensitive hash functions, the second part focuses on the entropy-based hash functions. The same tests were conducted for both hash function types comparing their results.

In the last section, the scaling behavior for both hash function types and ComputeCpp and hipSYCL as SYCL implementations on multiple GPUs is investigated. Furthermore, the achieved speedup is shown to prove the scalability of the `sycl_lsh` library implementation.

The compile time parameters mentioned in Section 5.2.1 were the same for all conducted tests. The `sycl_lsh::options::real_type` is set to `float`, the `sycl_lsh::options::index_type` and `sycl_lsh::options::hash_value_type` are set to `std::uint32_t` and the `sycl_lsh::options::blocking_size` is set to ten.

6.1 Setup

The following subsections describe the used hardware, data sets, compiler settings, as well as the used library versions.

	argon-gtx	Intel's devcloud (1)	Intel's devcloud (2)
number of nodes	1	1	1
processors	Intel Xeon Gold 5120	Intel Xeon E-2176G	Intel i9-10920X
number of sockets	2	1	1
processor frequency	2.2 GHz	3.7 GHz	3.5 GHz
total number of cores	28 (56 threads)	6 (12 threads)	12 (24 threads)
main memory	754 GB	64 GB	32 GB
accelerators	8x NVIDIA GeForce 1080 Ti	Intel UHD Graphics P630 Gen9	Intel Iris X^e MAX

Table 6.1: The hardware used to generate the different results.

6.1.1 Hardware

The `sycl_lsh` library has been tested on different clusters, the `argon-gtx` cluster located at the University of Stuttgart and the `devcloud`[1] powered by Intel. A detailed list of the used hardware is shown in Table 6.1.

6.1.2 Data Sets

Two data sets were used. The synthetic `friedman` data set proposed by Friedman [Fri91] in 1991 to illustrate Multivariate Adaptive Regression Splines (MARS). This data set variant consists of 500 000 points in 10 dimensions. Another real-world data set is based on a reduced `HIGGS` data set provided by Baldi, Sadowski, and Whiteson [BSW14] containing 1 000 000 data points in 27 dimensions. It is used for a classification problem that tries to distinguish a signal processes that produce Higgs bosons from other background processes that do not produce Higgs bosons.

6.1.3 Compiler and Libraries

The used compilers and toolkits or libraries vary based on the cluster. On the `argon-gtx` cluster `GNU GCC 9.2.0`, `OpenMPI 4.0.1`, `CUDA 10.2`, `ComputeCpp 2.1.0`, `hipSYCL master branch`, `CMake 3.18.4`, and `{fmt} 7.1.0` were used. On Intel's `devcloud`, `GNU GCC 9.3.0`, `Intel MPI 2021.1-beta10`, `oneAPI DPC++ Compiler Pro 2021.1`, `CMake 3.18.2`, and `{fmt} 7.1.0` were used.

[1] https://software.intel.com/content/www/us/en/develop/tools/devcloud.html

No additional performance tuning compiler flags are enabled, besides the flags automatically set when using `Release` as `CMAKE_BUILD_TYPE`. One exception is the specific compiler flag `-no-serial-memop` which is only set if the used SYCL implementation is ComputeCpp.

The compiler flags `-Wall`, `-Wextra`, and `-Wpedantic` are enabled to faster detect potential issues or bad habits.

6.2 Performance Evaluation on a single GPU

The following sections discuss the performance characteristics on a single GPU.

At first, the overall runtime behavior for the SYCL implementations based on the achieved recall and error ratio using the random projection hash functions, is considered. All three SYCL implementations, ComputeCpp, hipSYCL, and oneAPI, were used together with the `friedman` data set, whereas only ComputeCpp and hipSYCL were used for tests on the larger `HIGGS` data set. Because the resulting runtimes, recalls, and error ratios are heavily affected by the used parameters, the next subsections will give an overview of each parameter's influence. To test a single parameter's influence, all other parameters are set to a default value, while the currently investigated parameter's value varies in a defined range. The knowledge obtained by these tests can be used to efficiently tune the algorithms behavior the achieve the desired recall or error ratio while maintaining acceptable performance.

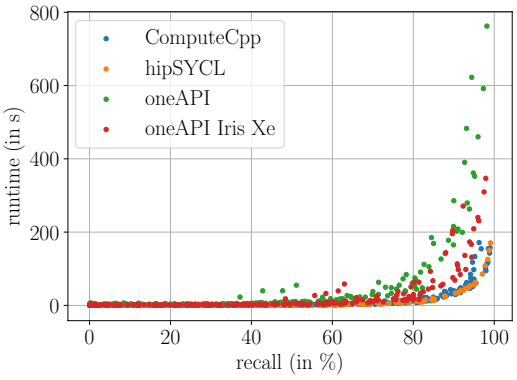
After that, the same tests are repeated for the entropy-based hash functions. Besides, the results for both hash function types are compared to each other.

6.2.1 Evaluating the Random Projection Hash Functions

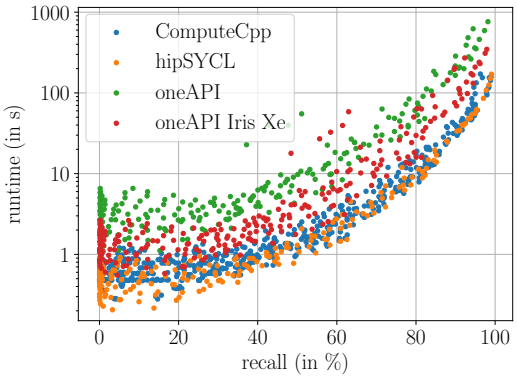
This section focuses on the random projection hash functions defined in Definition 3.2.3. A description of the implementation in the `sycl_lsh` library can be found in Section 5.2.3.

Figure 6.1 shows the runtime characteristics of the LSH algorithm for the three SYCL implementations, ComputeCpp, hipSYCL, and oneAPI, depending on the resulting recall and error ratio using the `friedman` data set. The runtimes include the generation of the hash functions, the construction of the hash tables and the calculation of the k-NN. Each dot represents the average of five samples using a specific set of parameter values. Intel's oneAPI implementation has been tested using two different GPUs, an Intel UHD Graphics P630 Gen9 GPU and an Intel Iris X^e MAX GPU. The SYCL implementations ComputeCpp and hipSYCL have been tested on a NVIDIA GEFORCE GTX 1080 Ti.

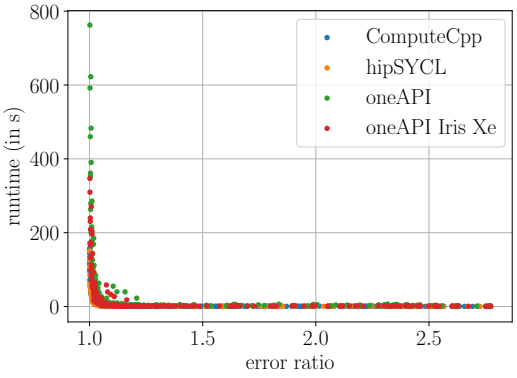
Figure 6.1a shows the runtime based on the achieved recall. For recalls lower than $20\,\%$, the runtimes of all three implementations do not increase significantly. Afterward, while the recall improves, the runtime increases exponentially. As can be seen, it is possible to achieve nearly $100\,\%$ recall for all implementations but the runtime worsens significantly.



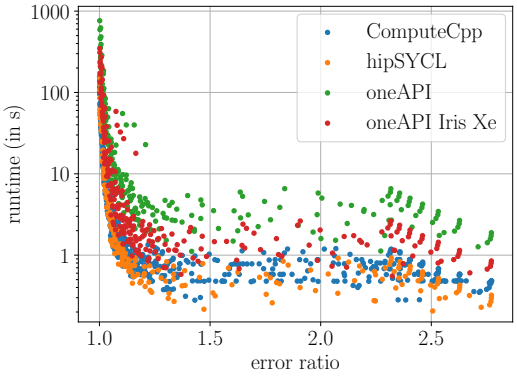
(a) The runtime of the three SYCL implementations depending on the recall.



(b) The runtime of the three SYCL implementations depending on the recall using a logarithmic plot.



(c) The runtime of the three SYCL implementations depending on the error ratio.



(d) The runtime of the three SYCL implementations depending on the error ratio using a logarithmic plot.

Figure 6.1: The runtime for the creation of the hash functions and hash tables, and the k-NN search, using the three SYCL implementations ComputeCpp, hipSYCL, and oneAPI, depending on the recall and error ratio using the `friedman` data set as well as random projections. Each point represents the average of five samples. The runtimes for ComputeCpp and hipSYCL were collected on a NVIDIA GEFORCE GTX 1080 Ti, the oneAPI runtimes were gathered on an Intel UHD Graphics P630 Gen9 and Intel Iris X^{e} MAX GPU.

Using an Intel UHD Graphics P360 Gen9 GPU, reaching a recall of $98.1\,\%$ results in a runtime of $12.7\,\mathrm{min}$. In contrast, by using an Intel Iris X^{e} MAX GPU instead, a recall of $97.9\,\%$ can be reached in $5.7\,\mathrm{min}$. Therefore, the new Intel GPU reduces the runtime by a factor of two while achieving a comparable recall. ComputeCpp using a NVIDIA GEFORCE

GTX 1080 Ti needs $2.8\,\mathrm{min}$ for a recall of $96.2\,\%$, while hipSYCL takes $2.8\,\mathrm{min}$ for a recall of $99.0\,\%$.

Using a logarithmic runtime scale in Figure 6.1b allows for a better insight of the actual runtime behavior. All SYCL implementations are subject to the same overall behavior. The shift of all implementations on the y-axis and, therefore, the different total runtimes can be explained by the fact that the tests were performed on different hardware with varying performance capabilities. While the Intel UHD Graphics P360 Gen9 GPU has the worst overall runtime, the Intel Iris X^{e} MAX GPU performs slightly better. Even better performance has the NVIDIA GEFORCE GTX 1080 Ti using the hipSYCL or ComputeCpp implementation.

Figure 6.1c displays the runtimes based on the achieved error ratio. Reaching an error ratio close to 1.0 corresponds to achieving a recall of nearly $100\,\%$. Therefore, just like the recall considerations, reaching an error ratio close to 1.0 results in a drastically increasing runtime. However, as shown in Figure 6.1d, if the target error ratio should roughly be 1.3, the runtime can already be reduced drastically. After that, worsening the error ratio does not result in a significantly lower runtime. All four implementations' overall runtime characteristics are similar, while the different hardware configurations can explain the differences between the total runtimes.

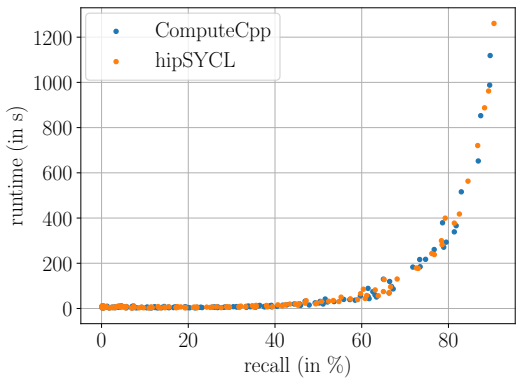
All in all, it can be said that the runtime characteristics do not depend on the used SYCL implementation for this example code.

Figure 6.2 displays the same situation as Figure 6.1, however, instead of the `friedman` data set, the `HIGGS` data set has been used to show the differences for a larger data set compared to the `friedman` data set. Furthermore, the only investigated SYCL implementations are ComputeCpp and hipSYCL. As shown in Figure 6.2a, both SYCL implementations behave the same. After a recall of $20\,\%$, the runtimes start to increase drastically for both implementations. Reaching a recall of $89.6\,\%$ using the ComputeCpp implementation takes $18.6\,\mathrm{min}$. Achieving a similar recall of $90.5\,\%$ for the hipSYCL implementation results in a runtime of $21.0\,\mathrm{min}$.

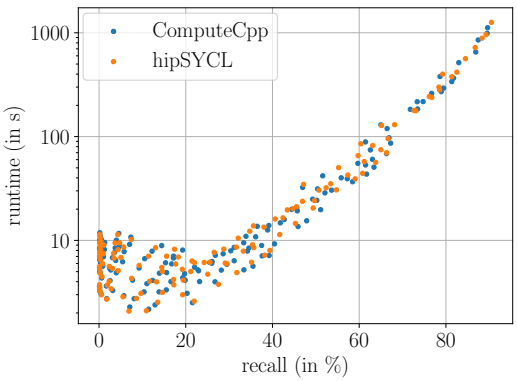
A similar behavior can also be seen in Figure 6.2b. For recalls lower than $20\,\%$, the runtimes of both implementations decrease by a few seconds. After that, the runtimes increase again.

Figure 6.2c displays a similar picture for the error ratio. As with the `friedman` data set, the runtime drastically decreases even for relatively small error ratios. However, for an error ratio close to 1.0, the runtimes increase significantly. The logarithmically scaled Figure 6.2d shows that both versions behave similarly for small and large error ratios. As for the `friedman` data set, after an error ratio of roughly 1.25, the runtime does not significantly increase anymore.

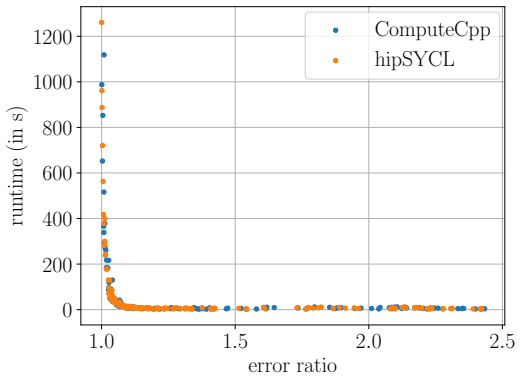
Comparing the `friedman` data set with the `HIGGS` data set shows that the overall runtime characteristics do not change. Only the total runtimes increase using the larger `HIGGS` data set. Comparing the achieved recalls between the two data sets shows that, although the same parameter combinations were used, a recall of nearly $100\,\%$ was reached for the



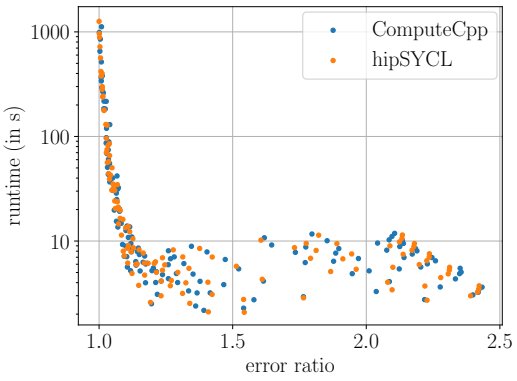
(a) The runtime of ComputeCpp and hipSYCL depending on the recall.



(b) The runtime of ComputeCpp and hipSYCL depending on the recall using a logarithmic plot.



(c) The runtime of ComputeCpp and hipSYCL depending on the error ratio.



(d) The runtime of ComputeCpp and hipSYCL depending on the error ratio using a logarithmic plot.

Figure 6.2: The runtime for the creation of the hash functions and hash tables and the k-NN search, using ComputeCpp and hipSYCL depending on the recall and error ratio using the `HIGGS` data set as well as random projections. Each point represents the average of five samples. All runtimes were collected on a NVIDIA GEFORCE GTX 1080 Ti.

`friedman` data set, whereas using the `HIGGS` data set, only a recall of around 90 % could be achieved.

In the following, the influence of the various `sycl_lsh::options` runtime parameters on the resulting runtime of the LSH algorithm, recall, and error ratio is investigated. The runtimes include the generation of the hash functions, the creation of the hash tables and the calculation of the k-NN. For this purpose, all parameters are set to a default value

`hash_pool_size`	300
`num_hash_functions`	20
`num_hash_tables`	40
`hash_table_size`	105 613
`w`	1.5
`k`	5

Table 6.2: Default parameter applied during the LSH parameter tests using random projections and the `friedman` data set.

according to Table 6.2, resulting in an overall runtime of $3.8\,\mathrm{s}$ and a k-NN search runtime of $3.4\,\mathrm{s}$, a recall of $70.5\,\%$, and an error ratio of 1.03.

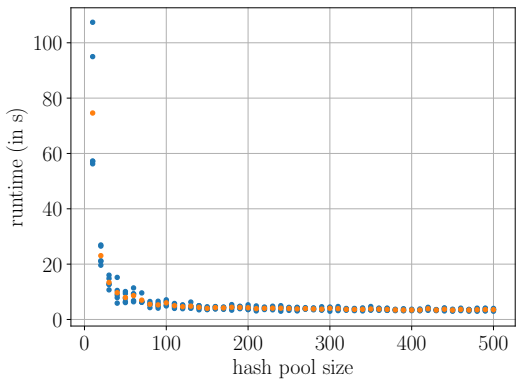
Only the ComputeCpp SYCL implementation and the `friedman` data set were used to obtain the following results. All parameter combinations were repeated five times. The blue dots represent these test runs, whereas the orange dots represent the resulting average times.

Influence of the `sycl_lsh::options::hash_pool_size` Parameter using Random Projections

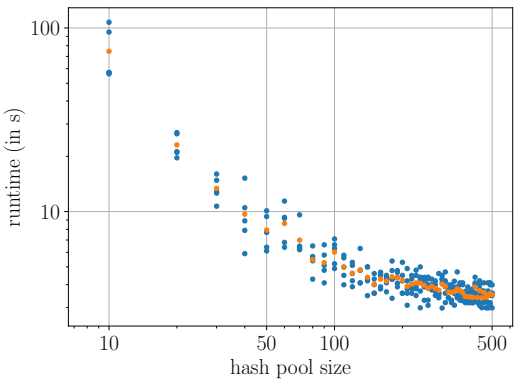
Figure 6.3 shows the `sycl_lsh` library implementation's behavior with a varying size of the used hash pool. All parameters besides the hash pool size have been set according to Table 6.2. The hash pool size has been varied between ten and 500 in steps of ten.

As shown in Figure 6.3a, the runtime drastically increases up to $74.6\,\mathrm{s}$ for a small number of hash functions in the hash pool. Increasing the hash pool size reduces the runtime from $74.6\,\mathrm{s}$ down to $3.5\,\mathrm{s}$ when using 500 hash functions in the hash pool. Figure 6.3b shows that the runtime continuously decreases with a growing hash pool size, although the runtime gain becomes smaller with larger hash pools. Therefore, increasing the variation of hash functions used for calculating the hash signatures improves the resulting runtime.

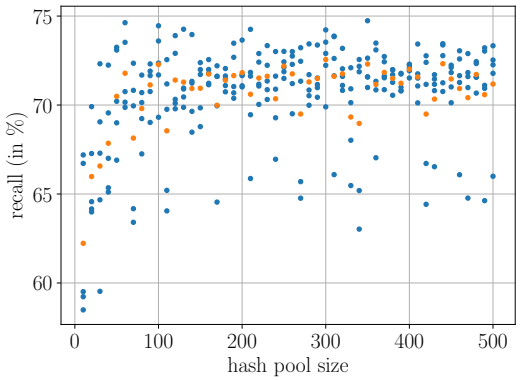
A different behavior compared to the runtime improvements can be seen for the recall in Figure 6.3c and the error ratio in Figure 6.3d. For small hash pool sizes, the recall only reaches $62.5\,\%$ on average. Few different hash functions in the hash pool result in a higher probability of different hash signatures consisting of the same hash functions. If the hash signatures used for different hash tables are identical, the inspected hash buckets result in the same data points investigated for the k-NN search. Therefore, the recall is not as high as if the hash signatures would vary more. Increasing the hash pool size improves the recall until 100 hash functions are reached. After that, increasing the hash pool size does not improve the recall any further, although the runtime still improved after 100 hash functions. At this point, the hash signatures are on average different enough that the data



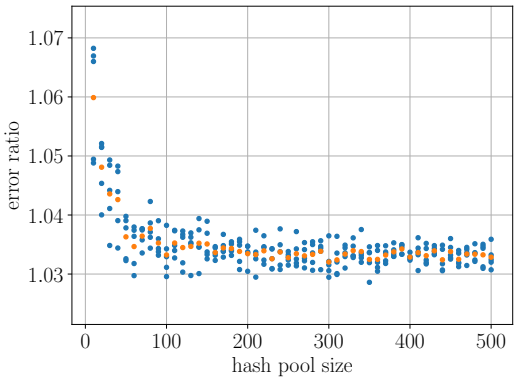
(a) The runtime depending on the hash pool size.



(b) The runtime depending on the hash pool size using a logarithmic plot.



(c) The recall depending on the hash pool size.



(d) The error ratio depending on the hash pool size.

Figure 6.3: The LSH algorithm's behavior depending on the hash pool size utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same hash pool size.

points inside specific hash buckets are not identical between multiple hash tables. Also worth mentioning is that the recall given a hash pool size varies rather heavily. Given a hash pool size of 20, the recall varies by 12.8 % between 59.5 % and 72.3 %. Variations in the recall are expected since the hash signatures depend on the used hash functions drawn from random distributions. However, such significant variations were surprising.

Similar behavior can be recognized for the error ratio. Increasing the hash pool size improves the error ratio until 100 hash functions in the hash pool are reached. Afterward, the error ratio improvements stagnate. However, even for a small hash pool size, an average

error ratio of 1.05 can be achieved which is good considering the average error ratio using 500 hash functions is still only 1.032.

Although, the number of hash function in the hash pool grew, the runtime to generate the hash pool has not increased. Drawing a few hundred to thousand random numbers from a random distribution does not incur a significant runtime overhead.

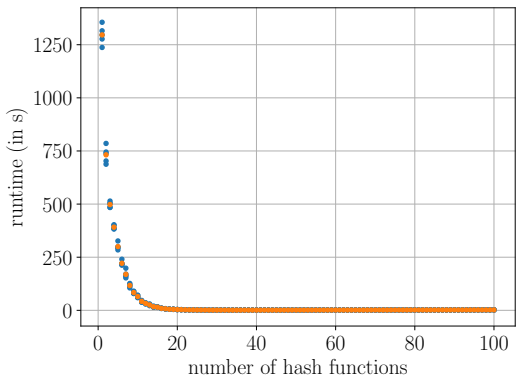
Influence of the `sycl_lsh::options::num_hash_functions` Parameter using Random Projections

The behavior of the `sycl_lsh` library implementation depending on the number of hash functions per hash table is shown in Figure 6.4. The number of hash functions has been varied between one and 100 while all other parameters have been set according to Table 6.2

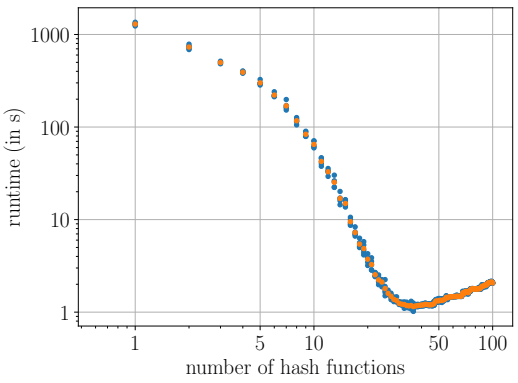
Figure 6.4a displays the runtime based on the number of used hash functions. A small number of hash functions means that fewer hash functions are used to compute a hash signature. Therefore, it is more likely that two data points have the same hash signature value since they must correspond in fewer hash values. If more data points have the same hash signature, more possible nearest neighbors must be considered during the k-NN search, and, therefore, the runtime increases. For example, using only one hash function per hash signature results in a runtime of more than $21\,\mathrm{min}$. Increasing the number of hash functions to calculate a hash signature reduces the probability of two points being assigned to the same hash bucket. Thus, fewer data points are considered during the k-NN search reducing the runtime to $2\,\mathrm{s}$ when using 100 hash functions per hash signature. This fact can better be seen in Figure 6.4b. The runtime decreases while increasing the number of hash functions until 35 hash functions. After that, the runtime starts to increase again. That can be explained by the fact that a single hash value calculation needs `sycl_lsh::data_attributes::dims` many multiplications to calculate the dot product. To determine the hash signature, `sycl_lsh::options::num_hash_functions` many dot products must be calculated. At first, the increasing cost of the hash signature calculation is dominated by the decreasing number of data points considered for the k-NN search. However, at some point, the reduction of data points per hash bucket is no longer enough to outweigh the increasing hash signature calculation's cost, and, therefore, the runtime increases.

As already mentioned, increasing the number of hash functions reduces the probability of two data points having the same hash signature. Therefore, fewer data points are assigned to the same hash buckets resulting in fewer data points considered during the k-NN search. This can directly be seen in Figure 6.4c. While it is possible to reach a recall of $100\,\%$ using only one hash function, the recall drops to $0.6\,\%$ when using 100 hash functions.

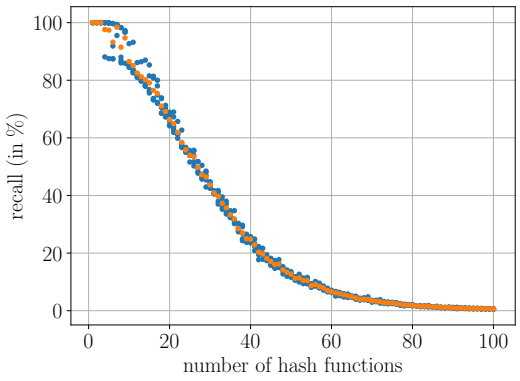
The same holds for the error ratio in Figure 6.4d. Reducing the number of hash functions improves the error ratio from 2.25 using 100 hash functions to 1.0, using only one hash function for calculating a hash signature.



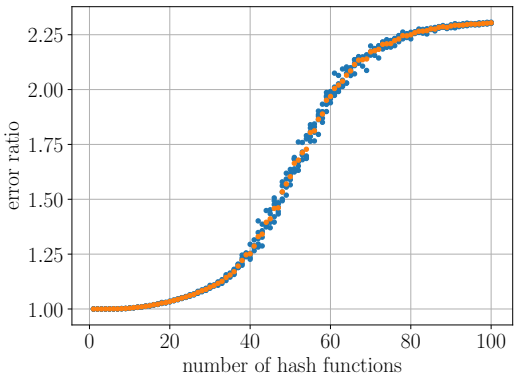
(a) The runtime depending on the number of hash functions per hash table.



(b) The runtime depending on the number of hash functions per hash table using a logarithmic plot.



(c) The recall depending on the number of hash functions per hash table.

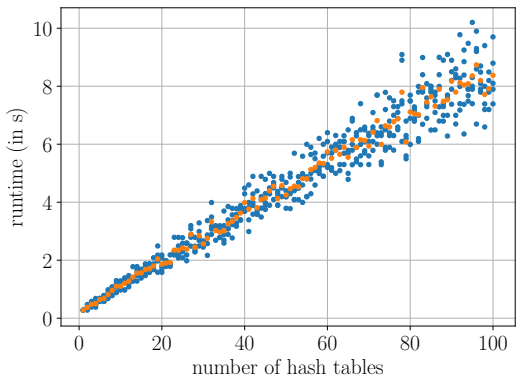


(d) The error ratio depending on the number of hash functions per hash table.

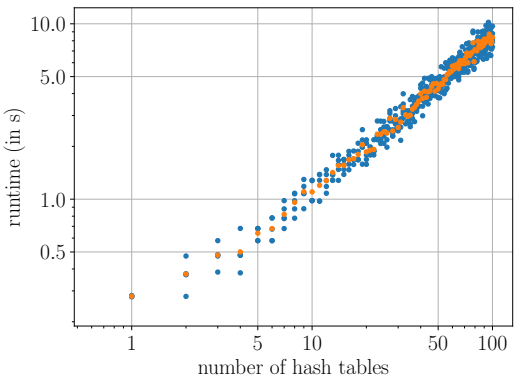
Figure 6.4: The LSH algorithm's behavior depending on the number of hash functions per hash table utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same number of hash functions.

Influence of the `sycl_lsh::options::num_hash_tables` Parameter using Random Projections

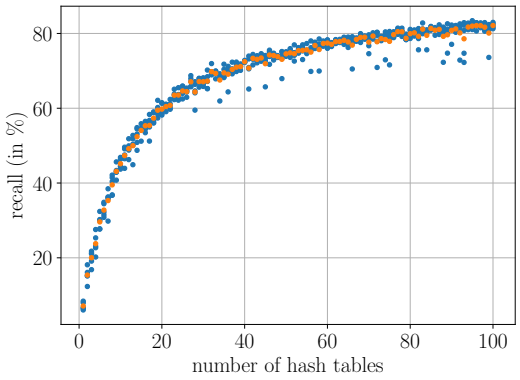
This subsection focuses on the effects of the number of hash tables on the resulting runtime, recall, and error ratio. The behavior of the `sycl_lsh` library implementation for a varying number of hash tables between one and 100 is displayed in Figure 6.5. All other parameters were set according to Table 6.2.



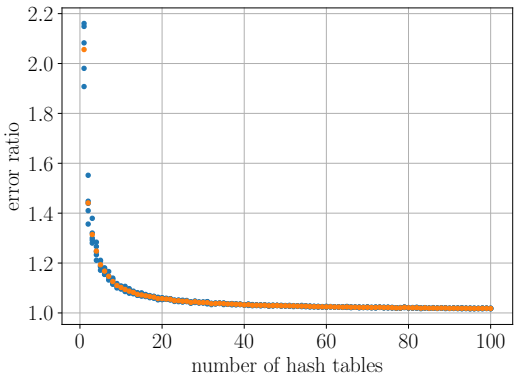
(a) The runtime depending on the number of hash tables.



(b) The runtime depending on the number of hash tables using a logarithmic plot.



(c) The recall depending on the number of hash tables.



(d) The error ratio depending on the number of hash tables.

Figure 6.5: The LSH algorithm's behavior depending on the number of hash tables utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same number of hash tables.

Figure 6.5a shows the runtime based on the number of hash tables. Increasing the number of hash tables also increases the runtime for the k-NN search. Using one hash table results in a runtime of $280\,\text{ms}$, while using 100 hash tables increases the average runtime to $8.3\,\text{s}$. In addition, doubling the number of hash tables, e.g., from 40 to 80, also roughly doubles the runtime from $3.7\,\text{s}$ to $7.0\,\text{s}$. Although the runtime increases linearly with a growing number of hash tables, the total runtimes do not get as worse as with a small number of hash functions. This behavior can be explained by the fact that increasing the number of hash tables results in more data points considered for the k-NN search since more hash buckets must be examined. Doubling the number of hash tables means that, to calculate

the k-NN, twice as many hash buckets must be considered. Therefore, on average, twice as many data points must be examined for calculating the k-NN of a data point resulting in a linear increase of the runtime. Figure 6.5b confirms the linear behavior. Another interesting observation is the variance of the test samples. With a growing number of hash tables, the runtime variance also increases. For example, the samples gathered using one hash table have a runtime variance of 2.5×10^{-6}, while the variance increases to 0.6 when using 100 hash tables.

The explanation of the runtime behavior can also explain the recall behavior shown in Figure 6.5c. If more hash tables are considered in the k-NN search, more data points are examined. Therefore, it is more likely that one of the correct nearest neighbors gets assigned the same hash bucket in at least one hash table, resulting in a higher recall. However, although the runtime only increases linearly, the recall behaves asymptotically. Changing from one hash table to 50 hash tables increases the recall by $67.7\,\%$ from $7.1\,\%$ to $74.8\,\%$. However, increasing the number of hash tables again to 100 improves the recall only by $7.3\,\%$ up to $82.1\,\%$. Therefore, based on other parameters' choices, not all correct nearest neighbors of a data point x may be assigned to at least one hash bucket also containing x, even if the number of hash tables increases.

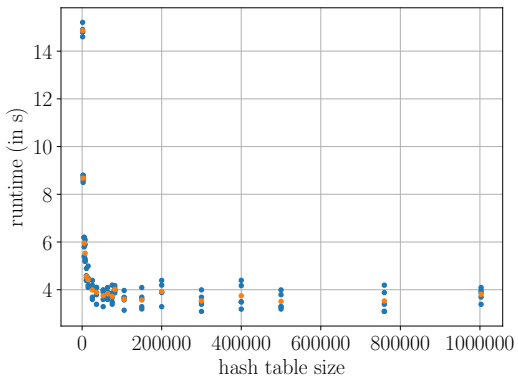
The same can be seen in Figure 6.5d. Increasing the number of hash tables reduces the error ratio. Although the maximum achieved recall is only $82.1\,\%$, the lowest error ratio is rather good, being approximately 1.03.

Influence of the `sycl_lsh::options::hash_table_size` Parameter using Random Projections

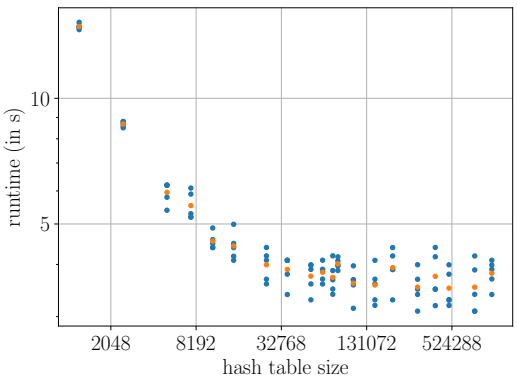
Figure 6.6 shows the influence of the hash table size on the behavior of the `sycl_lsh` implementation. All parameters have been set according to Table 6.2 besides the hash table size, which has been set to different values between 1223 and $1\,002\,893$.

Figure 6.6a shows the runtime based on the hash table size. Increasing the hash table size results in a reduced runtime. However, only small hash table sizes result in a significant higher runtime compared to larger hash table sizes. Additionally, the runtime increase by more than $10\,\mathrm{s}$ for a hash table size of 1223 is not much compared to the influence of other parameters, such as the number of hash functions previously discussed. Figure 6.6b shows that the runtime asymptotically decreases with an increasing hash table size. The hash table size reduces the potentially large hash signature values to moderate hash values needed to index the hash buckets. This is done by calculating `i % hash_table_size` with `i` being the hash signature of the inspected data point. Therefore, in the hash bucket `x` all data points with `x = i % hash_table_size` are inserted, resulting in more data points per hash bucket if the hash table size is small.

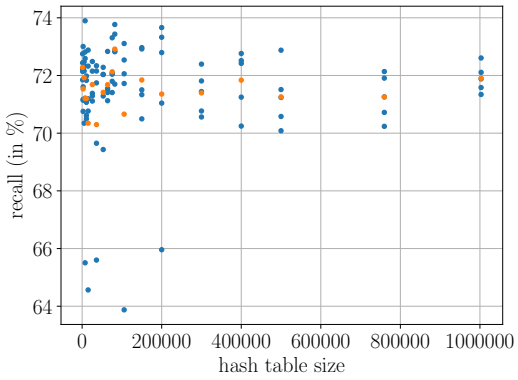
Although more data points reside in the same hash bucket when using small hash table sizes, Figure 6.6c shows that the hash table size does not significantly influence the resulting recall. The average recall varies between $70\,\%$ and $73\,\%$, independently of the hash table size. The recall does not change since the additional data points assigned to a hash bucket



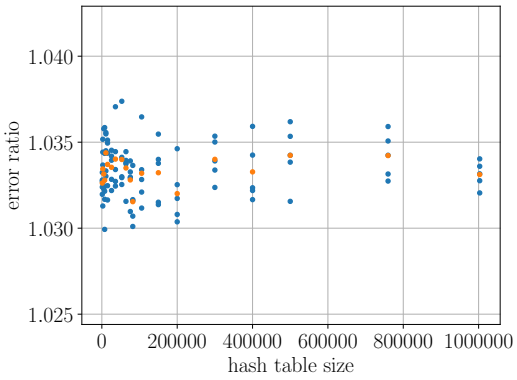
(a) The runtime depending on the hash table size.



(b) The runtime depending on the hash table size using a logarithmic plot.



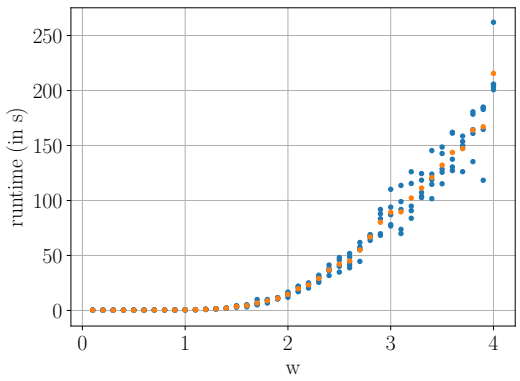
(c) The recall depending on the hash table size.



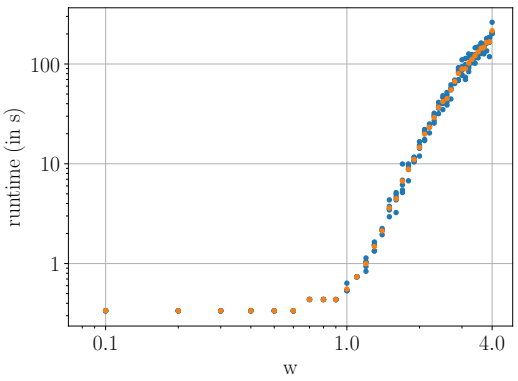
(d) The error ratio depending on the hash table size.

Figure 6.6: The LSH algorithm's behavior depending on the hash table size utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same hash table size.

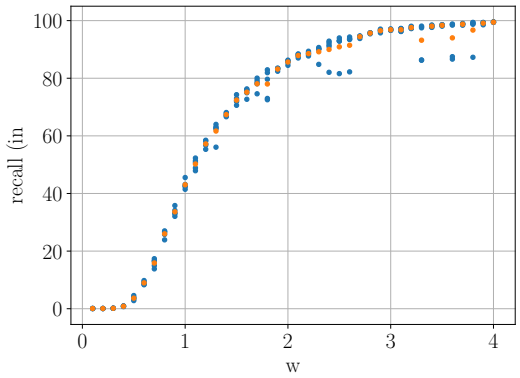
have considerably different hash signatures. Therefore, those data points were not similar to each other and will not improve the recall. The same holds for the error ratio in Figure 6.6d.



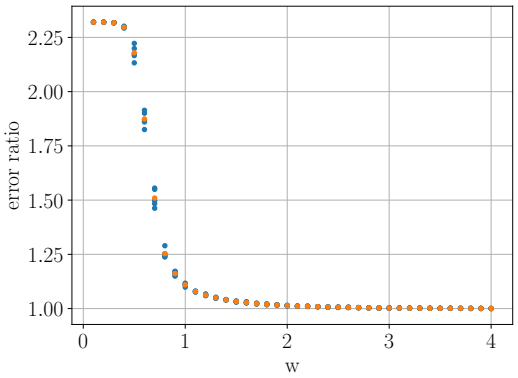
(a) The runtime depending on the segment size `w`.



(b) The runtime depending on the segment size `w` using a logarithmic plot.



(c) The recall depending on the segment size `w`.



(d) The error ratio depending on the segment size `w`.

Figure 6.7: The LSH algorithm's behavior depending on the parameter `w`, controlling the segment size, utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same segment size `w`.

Influence of the `sycl_lsh::options::w` Parameter using Random Projections

Figure 6.7 shows the behavior of the `sycl_lsh` library implementation with a varying segment size `w`. All parameters besides the `w` parameter have been set according to Table 6.2, while the parameter `w` has been varied between 0.1 and 4.0 in steps of 0.1.

As described in Definition 3.2.3, the parameter `w` is directly used in the random projections hash functions. It denotes the size of the segments in which the number line is divided to calculate the hash value. A larger `w` results in wider segments and, therefore, more data

points with the same hash value. The resulting runtime behavior can be seen in Figure 6.7a. Increasing the segment size `w` increases the runtime drastically. Using a segment size of 0.1 results in a runtime of $780\,\mathrm{ms}$. Increasing the segment size to 1.0 worsens the runtime only a bit, resulting in $1\,\mathrm{s}$. However, picking a higher value of `w`, e.g., 4.0, results in a significantly worse runtime of $3.6\,\mathrm{min}$.

The runtime behavior can better be seen in Figure 6.7b. Until a segment size of 1.0, the runtime does not increase significantly. This means that until `w = 1.0` expanding the segment size does not result in significantly more data points per hash bucket. After that, however, increasing the segment size increases the number of data points per hash bucket drastically, resulting in a significantly worse runtime.

Figure 6.7c shows the recall depending on the segment size `w`. Since more data points get the same hash value for wider segments and, therefore, more data points are assigned to a single hash bucket, it is more likely that more of the correct k-NN are found in a hash bucket. This results in an improvement of the recall with increasing segment size. A segment size of 0.1 results in a recall of $0.07\,\%$. Increasing `w` up to 0.3 does not increase the recall significantly. As already discussed, while looking at Figure 6.7b, increasing the segment size to 1.0 does not significantly increase the runtime. However, the recall improves to up to $43\,\%$. This means that the few data points assigned to the same hash bucket when widening the segments are part of the correct k-NN to a high percentage. Using 4.0 as segment size means that many data points are assigned to the same hash bucket, and, therefore, it becomes more likely that the correct k-NN are considered during the nearest neighbors search, resulting in a recall of $99.5\,\%$.

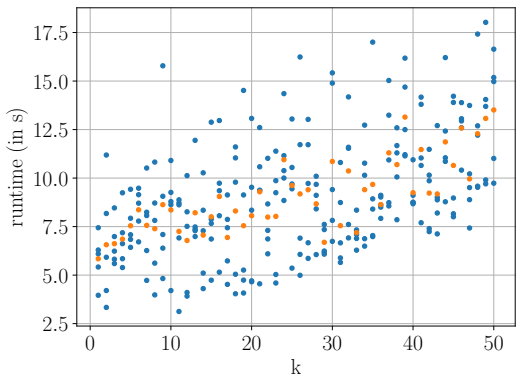
Figure 6.7d shows similar behavior for the error ratio. However, although a segment size of `w = 1.0` resulted in a recall of only $43\,\%$, the error ratio already dropped to a relatively good value of approximately 1.1.

Influence of the `k` Parameter using Random Projections

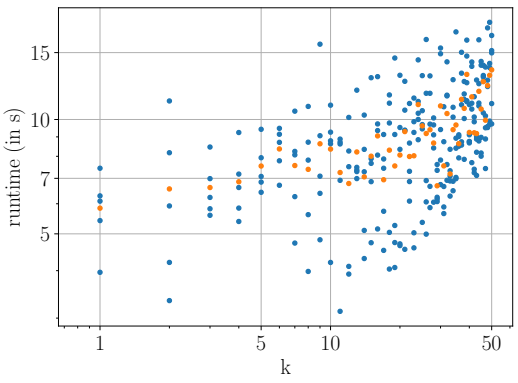
The behavior of the `sycl_lsh` library implementation depending on number of k-NN to search is shown in Figure 6.8. All parameters have been set according to Table 6.2. However, the parameter`k` has been varied between one and 50.

Figure 6.8a shows the runtime depending on the number of k-NN to search for. Increasing the parameter `k` increases the k-NN search's runtime because the nearest neighbors are updated in the k-NN search kernel using a single bubble-sort pass. Therefore, increasing the parameter `k` increases the cost for a single nearest neighbor update, resulting in a worse runtime. Comparing the runtimes for `k = 1` and `k = 50` shows that they increase by $7.7\,\mathrm{s}$. The same behavior can be seen in Figure 6.8b. However, a relatively large variance can be seen between samples for a specific value of `k`.

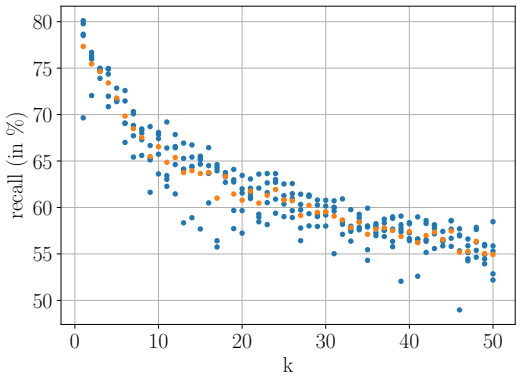
As shown in Figure 6.8c, the recall decreases with increasing `k`. The parameters were determined in a way that the recall for `k = 5` is approximately $70\,\%$. If `k` is less than five, fewer nearest neighbors must be found to achieve the same recall. Therefore, it is likely



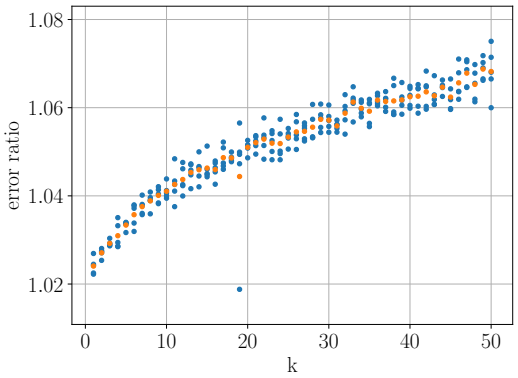
(a) The runtime depending on the parameter `k`.



(b) The runtime depending on the parameter `k` using a logarithmic plot.



(c) The recall depending on the parameter `k`.



(d) The error ratio depending on the parameter `k`.

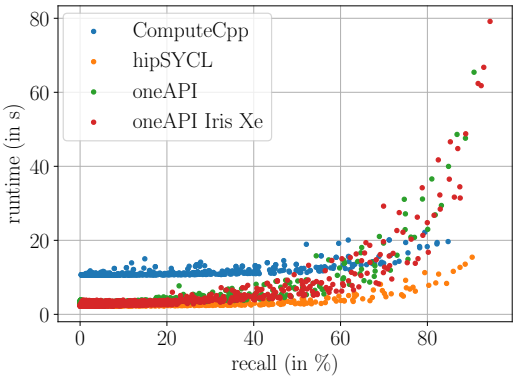
Figure 6.8: The LSH algorithm's behavior depending on the parameter `k`, representing the number of nearest neighbor to search, utilizing random projections. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same value for the parameter `k`.

that the recall is higher than for `k = 5`. In contrast, if `k` is greater than five, the recall worsens. To increase the recall for values of `k` greater than five, the already discussed parameters must be adjusted accordingly.

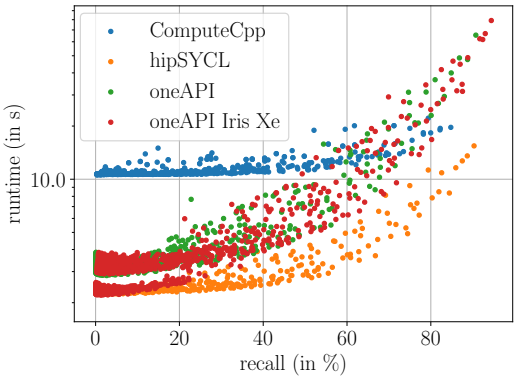
A similar behavior can be seen in Figure 6.8d. Increasing the parameter `k` worsens the resulting error ratio. However, even with `k = 50`, the error ratio is still considerably good with a value of 1.068.

6.2.2 Evaluating the Entropy-Based Hash Functions

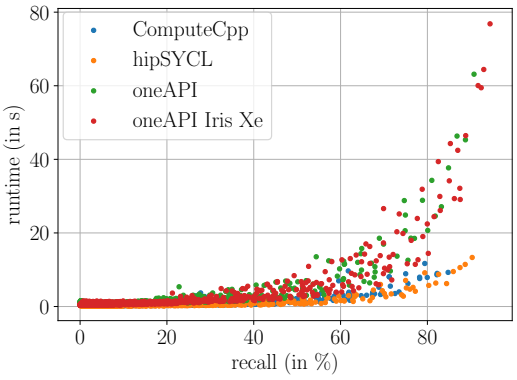
This section focuses on the entropy-based hash functions as another type of locality-sensitive hash functions. Again, at first, an overview over the runtime characteristics of all three investigated SYCL implementations is given. Afterward, the parameter tests for the random projections are repeated for the entropy-based hash functions.



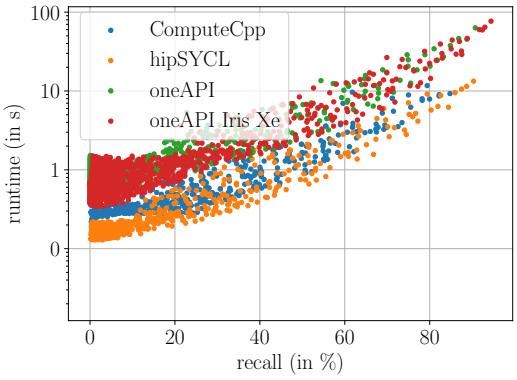
(a) The runtime of the three SYCL implementations depending on the recall.



(b) The runtime of the three SYCL implementations depending on the recall using a logarithmic plot.

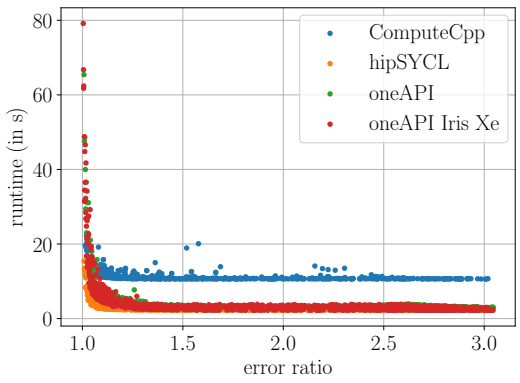


(c) The runtime of the three SYCL implementations depending on the recall without the runtime for the creation of the hash functions.

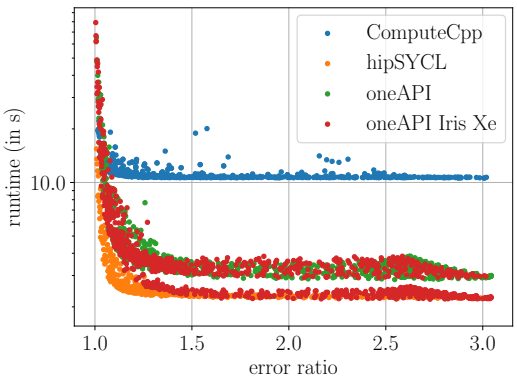


(d) The runtime of the three SYCL implementations depending on the recall using a logarithmic plot without the runtime for the creation of the hash functions.

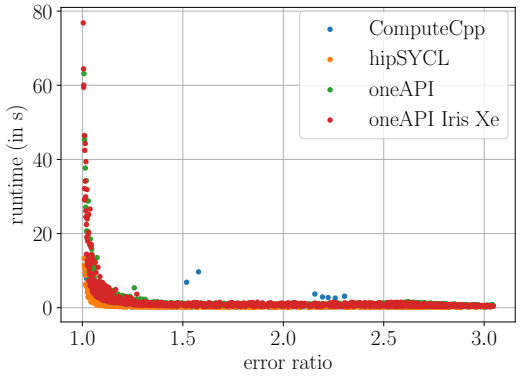
Figure 6.9 displays the same runtime characteristic tests as Figure 6.1, however, instead of the random projections the entropy-based hash functions are used. Again, all three SYCL implementations were compared regarding to the runtime depending on the achieved recall and error ratio. Each parameter combination was repeated five times for the oneAPI implementation and three times for the ComputeCpp and hipSYCL implementation. At first



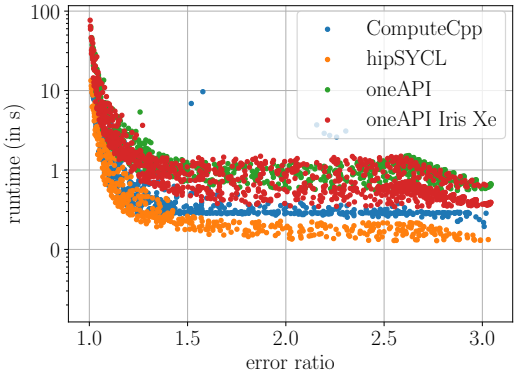
(e) The runtime of the three SYCL implementations depending on the error ratio.



(f) The runtime of the three SYCL implementations depending on the error ratio using a logarithmic plot.



(g) The runtime of the three SYCL implementations depending on the error ratio without the runtime for the creation of the hash functions.



(h) The runtime of the three SYCL implementations depending on the error ratio using a logarithmic plot without the runtime for the creation of the hash functions.

Figure 6.9: The runtimes using the three SYCL implementations ComputeCpp, hipSYCL, and oneAPI, depending on the recall and error ratio using the `friedman` data set as well as entropy-based hash functions. Each point represents the average of five samples. The runtimes for ComputeCpp and hipSYCL were collected on a NVIDIA GEFORCE GTX 1080 Ti, the oneAPI runtimes were gathered on an Intel UHD Graphics P360 Gen9 and Intel Iris X^e MAX GPU.

the results using the `friedman` data sets are discussed followed by the results for the `HIGGS` data set.

Figure 6.9a and Figure 6.9b show the runtime for the `friedman` data set depending on the achieved recall. In general, all three implementations behave the same: improving the recall results in longer runtimes. However, this time, both oneAPI tests have the same

runtime, although they were conducted on different hardware. A reason could be that the overall runtimes are relatively low for most of the test runs, and therefore the different hardware does not have such a significant influence on the runtime. However, the NVIDIA GEFORCE GTX 1080 Ti is more powerful than the Intel GPUs, and hence the runtimes are better, whereby the overall runtime characteristic is the same. The only outlier is the ComputeCpp implementation, which has a static overhead compared to the other implementations.

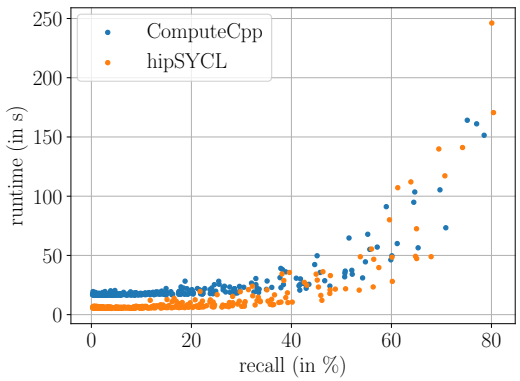
Nevertheless, as Figure 6.9c and Figure 6.9d, where the runtimes for the creation of the hash functions have been omitted, show, the ComputeCpp implementation's overhead is caused by the creation of the entropy-based hash functions. The ComputeCpp version behaves like the hipSYCL version without the runtime overhead for the creation of the hash functions, although slightly more overhead for small recalls can be seen. Inspections have shown, that the overhead for the creation of the hash functions comes from the call to `sycl::queue queue(device_selector{comm}, sycl::async_handler(&sycl_exception_handler));` in the constructor of the `sycl_lsh::entropy_based` class. However, further investigations must be conducted to find the actual reason behind this behavior.

Comparing both hash function types shows that the runtime increases more uniformly for the entropy-based hash functions. In contrast, the runtime for the random projections increases slower for smaller recalls and faster for higher recalls. Another difference are the overall smaller runtimes for the entropy-based hash functions in the case of high recall values.

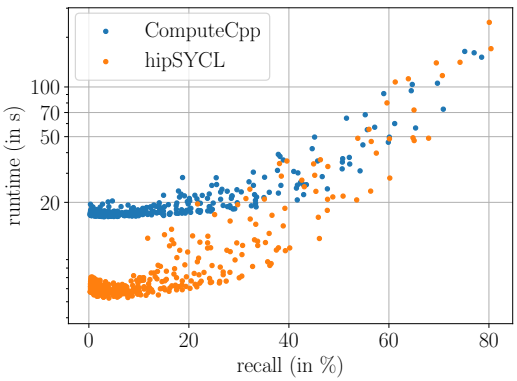
Figure 6.9e and Figure 6.9f show the same behavior but for the error ratio. Even for small error ratios, the runtime already decreases. After an error ratio of approximately 1.4, worsening the error ratio does not improve the runtime anymore. Here again, the ComputeCpp implementation has a static overhead compared to the other implementations for the same reason as stated above. Figure 6.9g and Figure 6.9h show that both oneAPI tests on different hardware again have the same runtimes. The ComputeCpp runtimes are slightly worse than the hipSYCL runtimes, although both tests were performed on the same hardware.

Comparing the runtimes to the ones of the random projections shows that the runtime is better using the entropy-based hash functions for small error ratios. However, for increasing error ratios, the runtimes of ComputeCpp and hipSYCL are roughly the same for both hash function types.

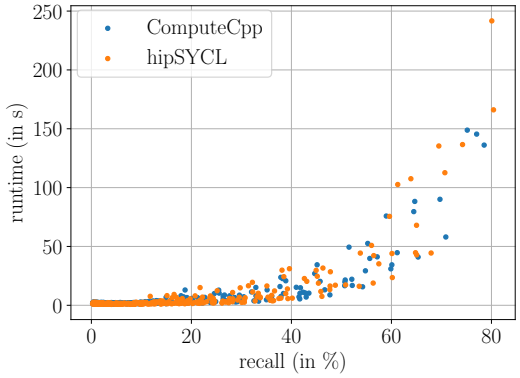
Figure 6.10 displays the same situation as Figure 6.9, however, instead of the `friedman` data set, the `HIGGS` data set was used. Furthermore, only ComputeCpp and hipSYCL were used as SYCL implementation. Figure 6.10c shows that both SYCL implementations behave the same, but the ComputeCpp implementation again has a static overhead. After a recall of $20\,\%$, the runtimes start to increase for both implementations. Compared to the random projections hash functions, the entropy-based hash functions only achieved a maximum recall of $80\,\%$, resulting in a runtime of approximately $2.5\,\mathrm{min}$ for the ComputeCpp implementation. To achieve a similar recall using the random projections, a runtime of



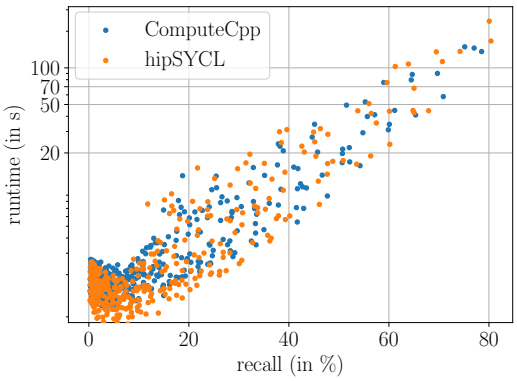
(a) The runtime of the three SYCL implementations depending on the recall.



(b) The runtime of the three SYCL implementations depending on the recall using a logarithmic plot.



(c) The runtime of the three SYCL implementations depending on the recall without the runtime for the creation of the hash functions.

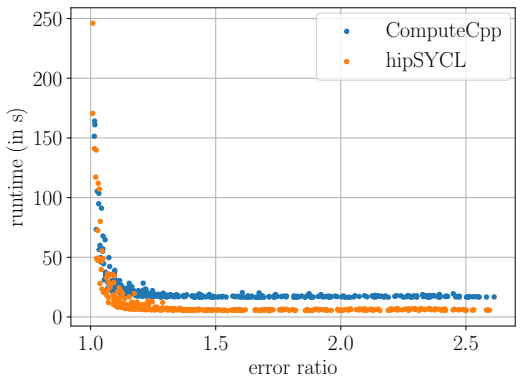


(d) The runtime of the three SYCL implementations depending on the recall using a logarithmic plot without the runtime for the creation of the hash functions.

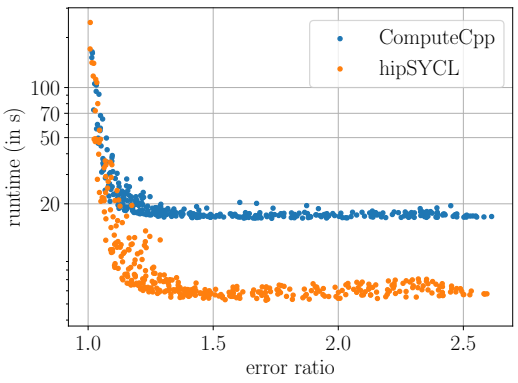
approximately $5\,\mathrm{min}$ is needed. Using other parameters could further increase the recall up to the same as for the random projections. Another difference is that most of the investigated parameter combinations yield a recall below $30\,\%$, whereas the recall values using the random projections were more evenly distributed.

Plotting only the runtimes for the construction of the hash tables and the k-NN search as in Figure 6.10c and Figure 6.10d shows that both implementations behave the same.

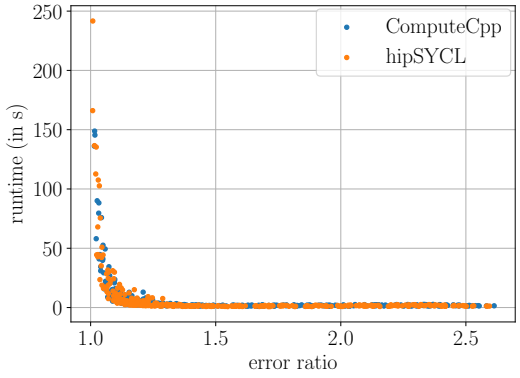
A difference between both hash function types can be seen in Figure 6.10d. Whereas the runtime decreases until a recall of $20\,\%$ using random projections, utilizing entropy-based hash functions the runtime only decreases until $10\,\%$.



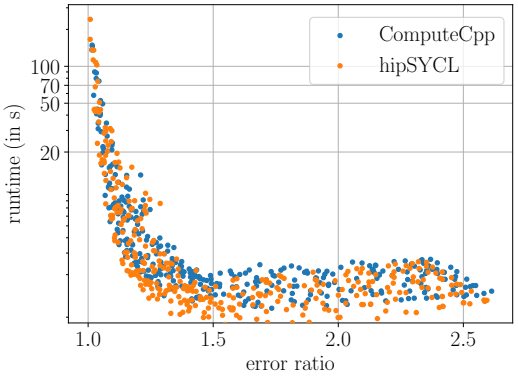
(e) The runtime of the three SYCL implementations depending on the error ratio.



(f) The runtime of the three SYCL implementations depending on the error ratio using a logarithmic plot.



(g) The runtime of the three SYCL implementations depending on the error ratio without the runtime for the creation of the hash functions.



(h) The runtime of the three SYCL implementations depending on the error ratio using a logarithmic plot without the runtime for the creation of the hash functions.

Figure 6.10: The runtime using the ComputeCpp and hipSYCL SYCL implementations depending on the recall and error ratio using the `HIGGS` data set as well as entropy-based hash functions. Each point represents the average of five samples. The runtimes were collected on a NVIDIA GEFORCE GTX 1080 Ti.

Figure 6.10g and Figure 6.10h draw a similar picture compared to the random projections. The runtime already drops down to under $20\,\mathrm{s}$ for small error ratios. Increasing the error ratio above 1.4 does not further decrease nor increase the runtime.

In the following, as for the random projections, the effects of the various parameters on the results are investigated. All tests were performed as in Section 6.2.2 but instead of the random projections the entropy-based hash functions were used. This has the consequence that other default parameters were used as shown in Table 6.3. These parameters result in

hash_pool_size	50
num_hash_functions	6
num_hash_tables	8
hash_table_size	105 613
num_cut_off_points	4
k	5

Table 6.3: Default parameter applied during the LSH parameter tests using entropy-based hash functions.

an overall runtime of 13.8 s and a k-NN search runtime of 3.3 s, a recall of 72.8 % and an error ratio of 1.03.

Again, only the ComputeCpp implementation combined with the `friedman` data set was used. The orange dots are the averages over five runs represented by the blue dots.

Influence of the `sycl_lsh::options::hash_pool_size` Parameter using Entropy-Based Hash Functions

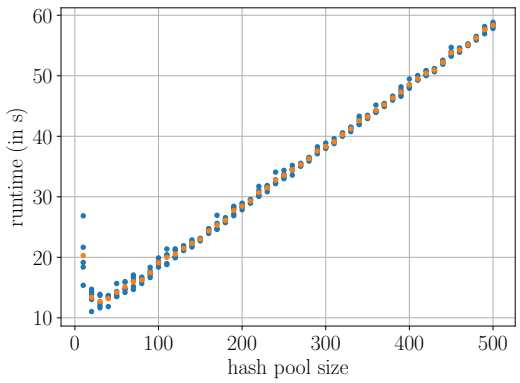
The same tests as in Section 6.2.1 regarding the hash pool size were conducted with the only exception being the used hash function type now corresponds to the entropy-based hash functions.

Figure 6.11a and Figure 6.11b show that the overall runtime behavior using entropy-based hash functions instead of random projections is entirely different. In the case of the random projections, the overall runtime decreased with a growing hash pool size. However, after a hash pool size of 30, the entropy-based hash functions runtime drastically increases.

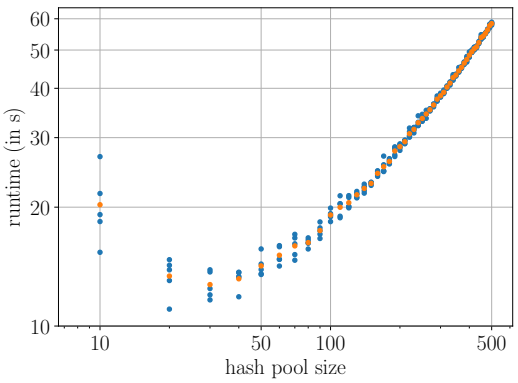
In contrast, the recall and error ratio characteristics depending on the hash pool size are the same for both random projections and entropy-based hash functions, as shown in Figure 6.11c and Figure 6.11d.

The creation of the hash function causes the difference in the overall runtime behavior. Figure 6.11e and Figure 6.11f show the runtime for the construction of the hash tables and the k-NN search. If only the sum of those two runtimes are considered, the runtime behavior is the same as for the random projections: increasing the hash pool size reduces the runtime. However, one difference is that using a small hash pool size increases the runtime only up to 13.9 s compared to the 74 s using random projections.

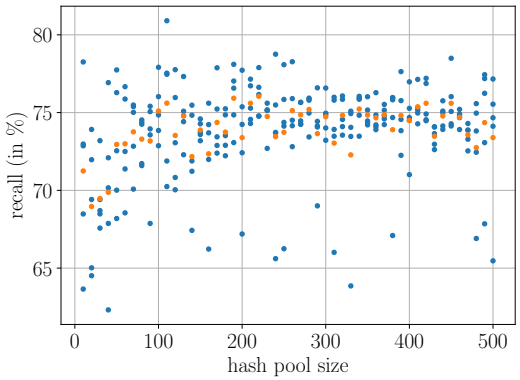
While random projection hash functions can be generated efficiently, creating entropy-based hash functions is considerably more inefficient, as stated in Section 5.2.3. This is shown in Figure 6.11g and Figure 6.11h. The runtime cost of creating the entropy-based hash



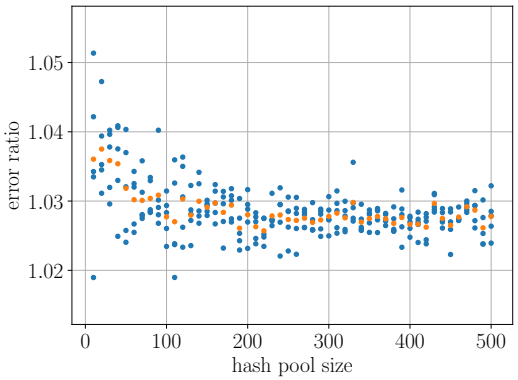
(a) The runtime depending on the hash pool size.



(b) The runtime depending on the hash pool size using a logarithmic plot.



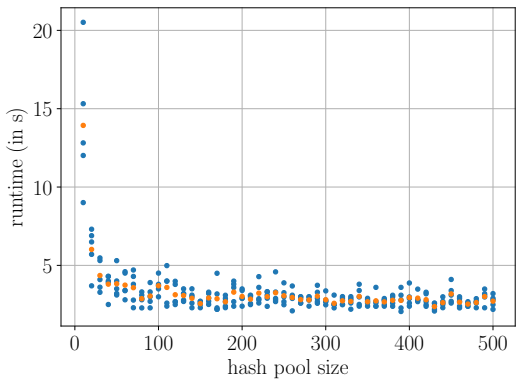
(c) The recall depending on the hash pool size.



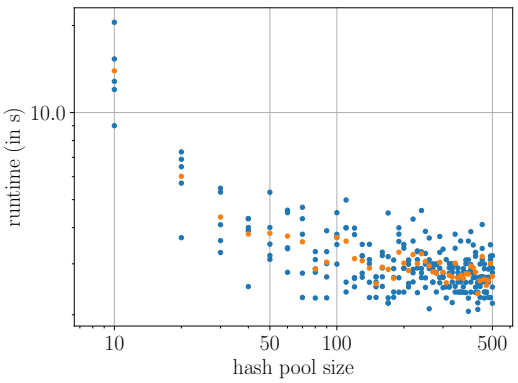
(d) The error ratio depending on the hash pool size.

functions grows linearly with the hash pool size. Generating a hash pool with ten hash functions takes 6.3 s, while creating a hash pool with 500 hash functions already takes 55.6 s. This is even amplified by the static overhead for the ComputeCpp implementation observed at the beginning of Section 6.2.2. For each hash function in the hash pool, the initial mapping values must be calculated for all data points, followed by sorting all these mapped values to determine the cut-off points. These calculations are not trivial, and, therefore, an increase in the runtime with more hash functions in the hash pool can be observed.

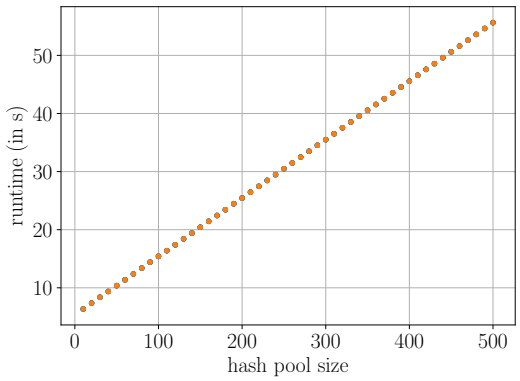
Figure 6.11e and Figure 6.11g also explain why the runtime in Figure 6.11a decreases until a hash pool size of 30. Increasing the hash pool from ten to 20 reduces the runtime for the construction of the hash tables and the k-NN search by 7.9 s, while the runtime for the creation of the hash functions only increases by 1.0 s. Therefor, the total runtime decreases by 6.9 s as shown in Figure 6.11a.



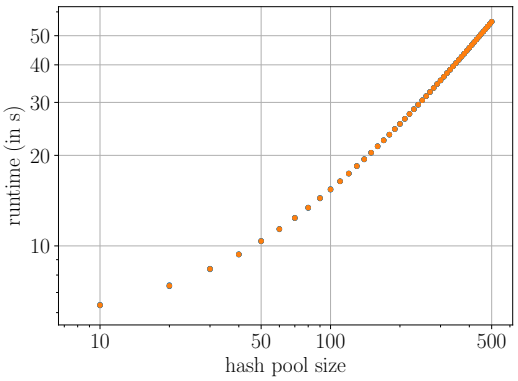
(e) The runtime depending on the hash pool size without the runtime for the creation of the hash functions.



(f) The runtime depending on the hash pool size using a logarithmic plot without the runtime for the creation of the hash functions.



(g) The runtime for creating the hash functions depending on the hash pool size.

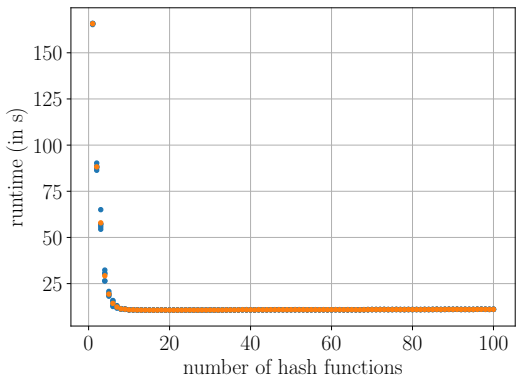


(h) The runtime for creating the hash functions depending on the hash pool size using a logarithmic plot.

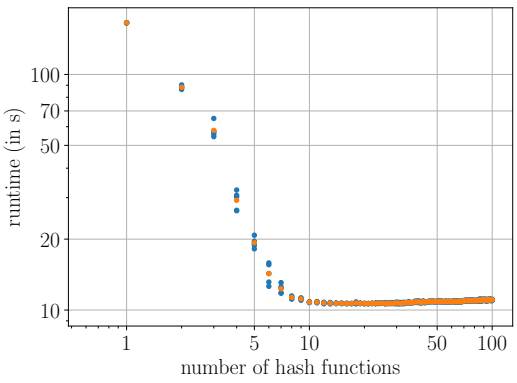
Figure 6.11: The LSH algorithm's behavior depending on the hash pool size utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same hash pool size.

Influence of the `sycl_lsh::options::num_hash_functions` Parameter using Entropy-based Hash Functions

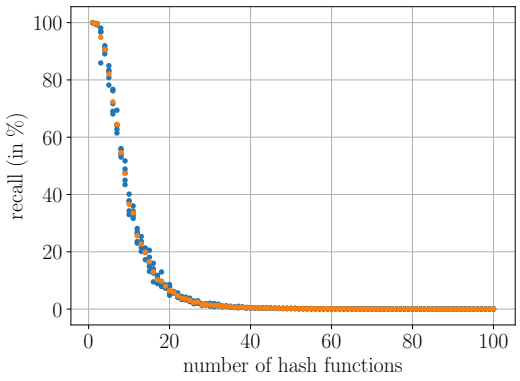
The same tests as in Section 6.2.1 regarding the number of hash functions per hash table were conducted. Again, the only difference is the usage of the entropy-based hash functions to generate the hash signatures.



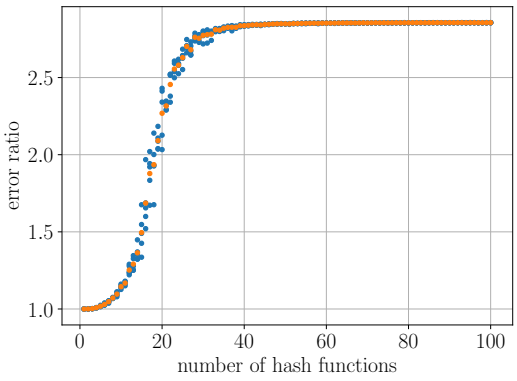
(a) The runtime depending on the number of hash functions per hash table.



(b) The runtime depending on the number of hash functions per hash table using a logarithmic plot.



(c) The recall depending on the number of hash functions per hash table.



(d) The error ratio depending on the number of hash functions per hash table.

Figure 6.12: The LSH algorithm's behavior depending on the number of hash functions per hash table utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same number of hash functions per hash table.

The overall runtime, recall, and error ratio characteristics depending on the number of hash functions are the same between entropy-based hash functions and random projections, as shown in Figure 6.12. However, looking closer, a few differences can be seen.

The entropy-based hash function runtime is significantly better, $2.7\,\mathrm{min}$ vs. $21\,\mathrm{min}$, for fewer hash functions. However, for more hash functions used to calculate a hash signature,

the runtime does not drop as low as for the random projections because of the runtime costs for generating the entropy-based hash functions and ComputeCpp's static overhead.

Figure 6.12b also shows that the runtime slightly increases again if the number of hash functions becomes large enough. When using the random projection hash functions, this happened after using 35 hash functions. However, using the entropy-based hash functions, the runtime increase already happens for 20 hash functions.

Also, the recall behaves slightly different in Figure 6.12c. When using the entropy-based hash functions, the recall worsens earlier than when using the random projections. Both hash function types start with a recall of $100\,\%$ using only one hash function. However, the entropy-based hash function's recall is nearly zero using 40 hash functions. In contrast, using the random projections hash functions, more than 80 hash functions are needed for the recall to drop to nearly zero percent.

The differences in the error ratio behaviors seen in Figure 6.12d are similar to the recalls' differences.

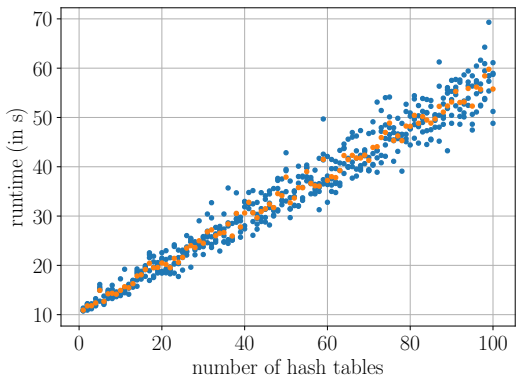
Influence of the `sycl_lsh::options::num_hash_tables` Parameter using Entropy-Based Hash Functions

Next, the influence of the number of hash tables using entropy-based hash functions will be examined. For this purpose, the same tests as in Section 6.2.1 regarding the number of hash tables were conducted.

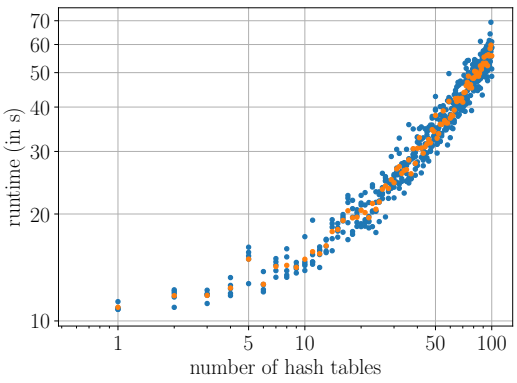
Figure 6.13a shows the runtime depending on the number of hash tables. At first glance, it behaves similarly to the random projections, with the difference being higher runtimes in total. While 100 hash tables took $8.3\,\mathrm{s}$ in case of random projections, the entropy-based hash functions needed nearly $1\,\mathrm{min}$. However, looking at Figure 6.13b shows that the runtime behaves slightly different. At first, the runtime increases more slowly, but the runtime starts to increase faster when using more hash tables. Figure 6.13f shows the same situation. However, this time, only the runtimes for the hash table creation and the k-NN search are plotted. The runtime for creating the hash functions is omitted. In this case, the runtimes behave similarly again except for the total runtimes. Therefore, the different logarithmic runtime characteristics were caused by the overhead of the entropy-based hash function generation.

The behavior of the recall and error ratio using the entropy-based hash functions is similar to the random projections. Two small differences compared to the random projections are shown in Figure 6.13c. The recall improves faster, and asymptotically reaches nearly $100\,\%$. To achieve a recall of more than $80\,\%$, roughly 20 hash tables are needed. In contrast, in the case of the random projection hash functions, more than 80 hash tables were needed to reach a recall of $80\,\%$.

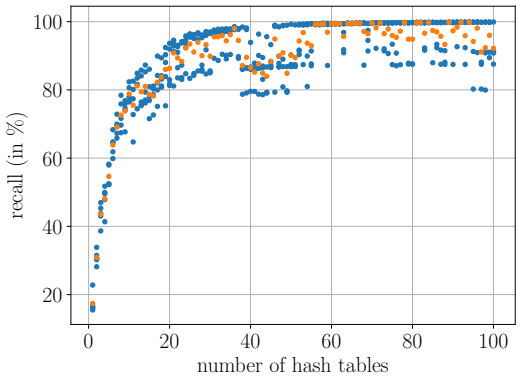
The same holds for the error ratio in Figure 6.13d. The error ratio decreases faster in the case of the entropy-based hash functions compared to the random projections and reaches a value close to 1.0 already when using 17 hash tables.



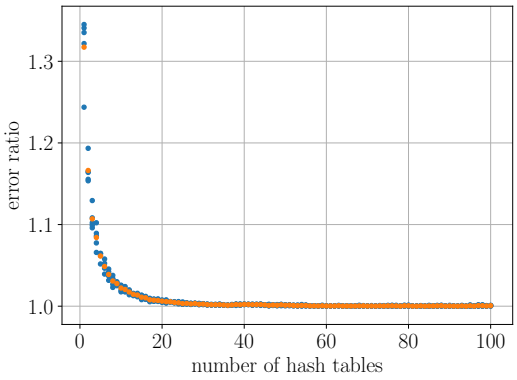
(a) The runtime depending on the number of hash tables.



(b) The runtime depending on the number of hash tables using a logarithmic plot.



(c) The recall depending on the number of hash tables.



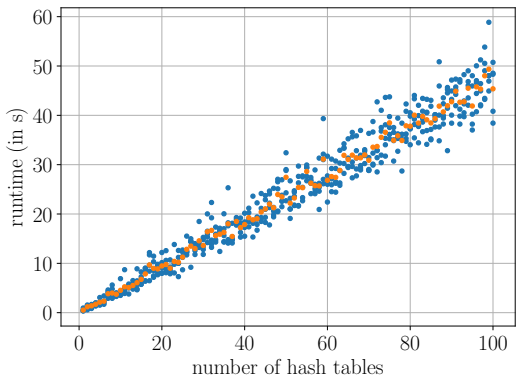
(d) The error ratio depending on the number of hash tables.

Influence of the `sycl_lsh::options::hash_table_size` Parameter using Entropy-Based Hash Functions

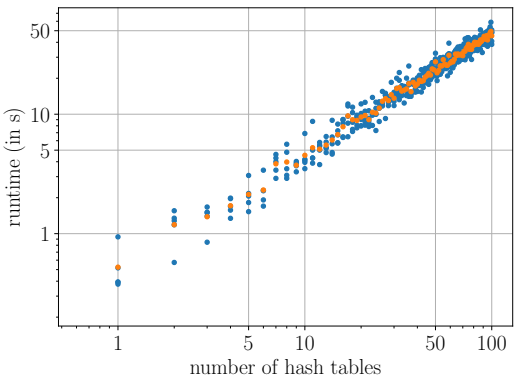
The same tests as in Section 6.2.1 regarding the hash table size were conducted with the only exception being the used hash function type now corresponds to the entropy-based hash functions.

Increasing the hash table size using random projections improved the overall runtime, as discussed in Section 6.2.1. However, increasing the hash table size using entropy-based hash functions does not significantly change the runtime, as shown in Figure 6.14a and Figure 6.14b. The runtime varies between 15.9 s and 13.6 s without any specific behavior.

The difference between both hash functions is that in the case of random projections, the hash value size can vary greatly based on the randomly created hash functions and the data points. In contrast, the hash values using entropy-based hash functions are always in



(e) The runtime depending on the number of hash tables without the runtime for the creation of the hash functions.



(f) The runtime depending on the number of hash tables using a logarithmic plot without the runtime for the creation of the hash functions.

Figure 6.13: The LSH algorithm's behavior depending on the number of hash tables utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same number of hash tables.

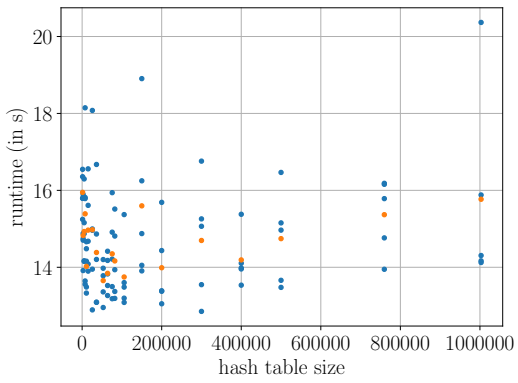
the range between 0 and `sycl_lsh::options::num_cut_off_points - 1`. Therefore, it is more unlikely that different hash signature values are mapped to the same hash bucket, even for small hash table sizes.

In comparison, the recall and error ratio behave similarly for both hash function types. The hash table size does not influence the recall in Figure 6.14c and the error ratio in Figure 6.14d.

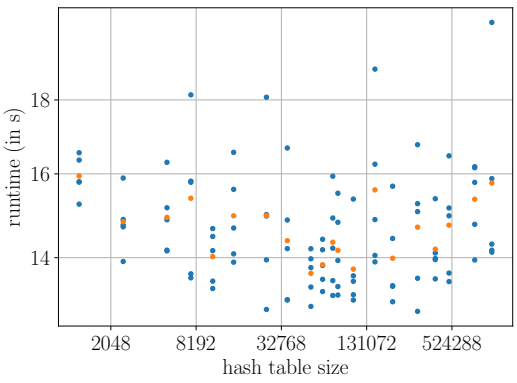
Influence of the `sycl_lsh::options::num_cut_off_points` Parameter using Entropy-Based Hash Functions

Figure 6.15 shows the `sycl_lsh` library implementation's behavior with a varying number of cut-off points. All parameters besides the number of cut-off points have been set according to Table 6.3. The number of cut-off points has been set to different values between one and 50 in steps of one and between 100 and 500 in steps of 50.

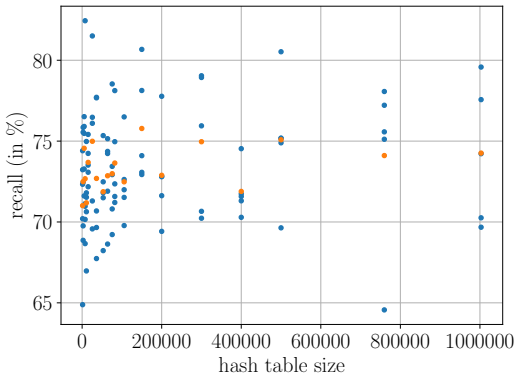
Figure 6.6a shows the runtime based on the number of cut-off points. Using only one cut-off point results in by far the most extended runtime of nearly 10 min. This can be explained by the fact that when using only one cut-off point all data points have the same hash value across all hash tables. Therefore, all data points are inserted into the same hash bucket, which means that the algorithm degenerates to a simple brute force search. More precisely, the LSH algorithm degenerates to `sycl_lsh::options::num_hash_tables`



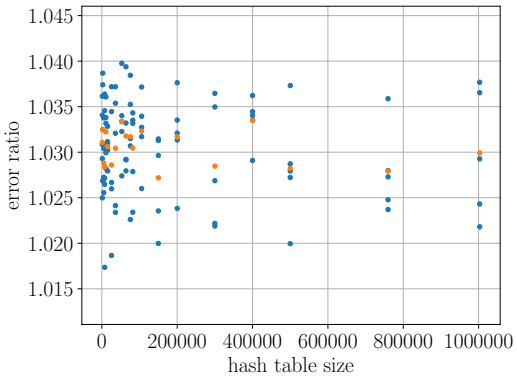
(a) The runtime depending on the hash table sizes.



(b) The runtime depending on the hash table size using a logarithmic plot.



(c) The recall depending on the hash table size.



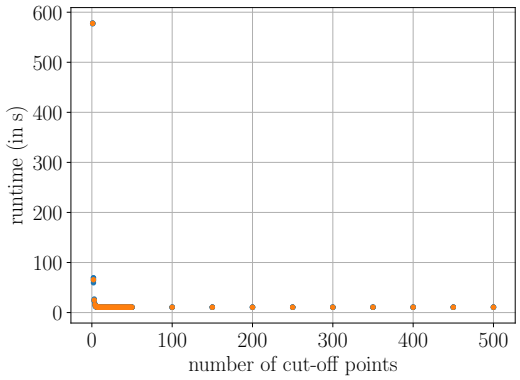
(d) The error ratio depending on the hash table size.

Figure 6.14: The LSH algorithm's behavior depending on the hash table size utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same hash table size.

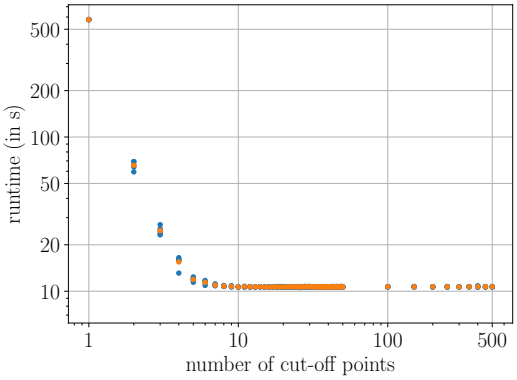
many brute force searches. Adding a second cut-off point, however, already reduces the runtime down to approximately $1\,\mathrm{min}$.

The behavior can better be seen in Figure 6.15b. Increasing the number of cut-off points reduces the runtime until ten cut-off points. Afterward, adding more cut-off points does not result in a runtime improvement. The asymptotic boundary of $10\,\mathrm{s}$ corresponds to the overhead of the hash function creation.

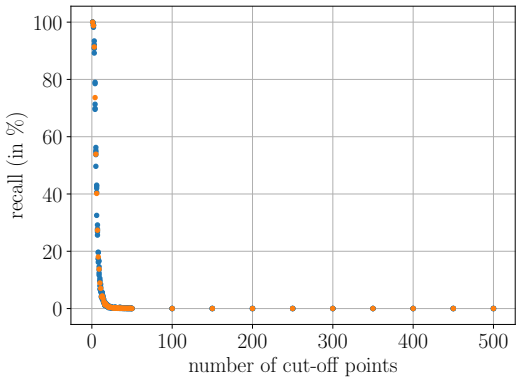
Figure 6.15c shows the behavior of the recall with a varying number of cut-off points. A recall of $100\,\%$ can be reached using one cut-off point since this corresponds to a brute



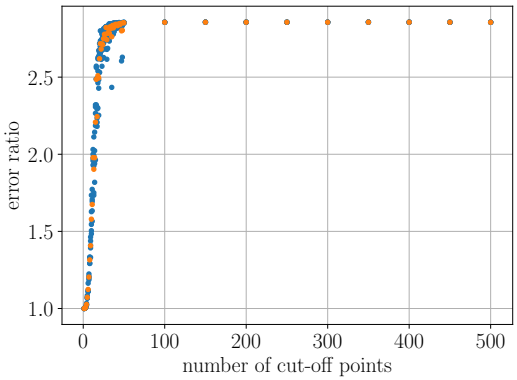
(a) The runtime depending on the number of cut-off points.



(b) The runtime depending on the number of cut-off points using a logarithmic plot.



(c) The recall depending on the number of cut-off points.

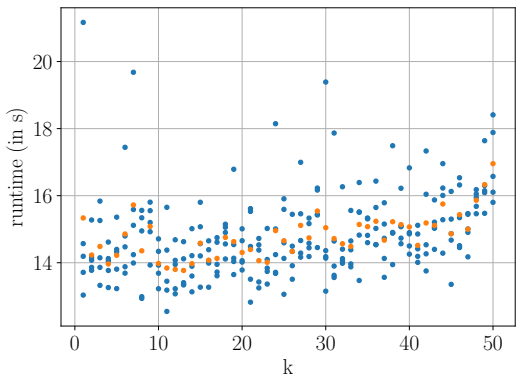


(d) The error ratio depending on the number of cut-off points.

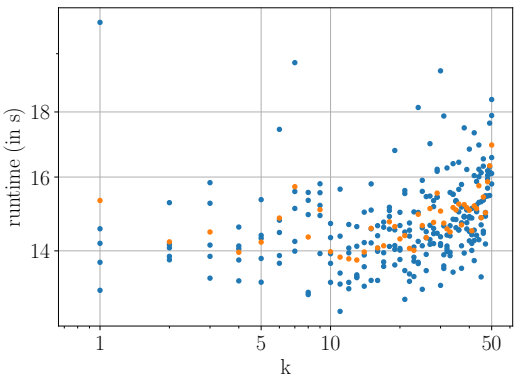
Figure 6.15: The LSH algorithm's behavior depending on the number of cut-off points utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same number of cut-off points.

force search. If 20 cut-off points are used, the recall drops to 0.7 % and reaches 0.01 % if 500 are used.

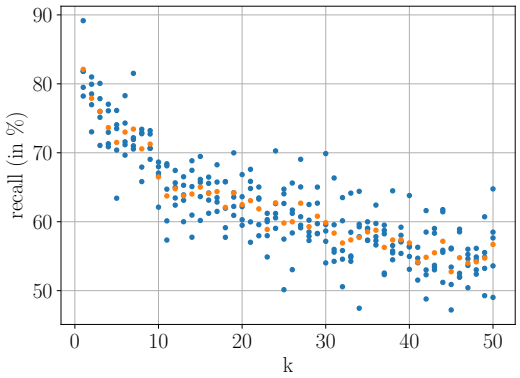
A similar behavior is shown in Figure 6.15d. One cut-off point results in an error ratio of 1.0, whereas 15 cut-off points already result in an error ratio of 2.2.



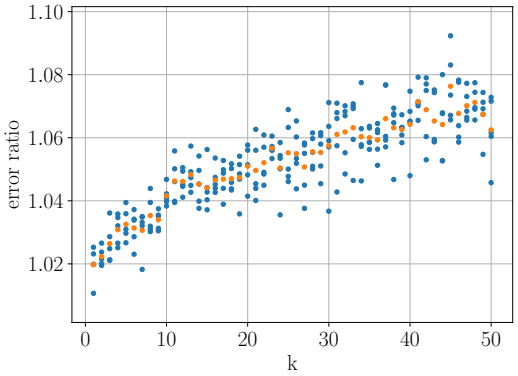
(a) The runtime depending on the parameter `k`.



(b) The runtime depending on the parameter `k` using a logarithmic plot.



(c) The recall depending on the parameter `k`.



(d) The error ratio depending on the parameter `k`.

Figure 6.16: The LSH algorithm's behavior depending on the parameter `k`, representing the number of nearest neighbor to search, utilizing entropy-based hash functions. The presented values are obtained on a single NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp as SYCL implementation and the `friedman` data set. The orange dots are the averages over all blue dots using the same value for the parameter `k`.

Influence of the `k` Parameter using Entropy-Based Hash Functions

Next, the influence of the number of k-NN to search using entropy-based hash functions will be examined. For this purpose, the same tests as in Section 6.2.1 regarding the parameter choice `k` were conducted.

Figure 6.16a and Figure 6.16b display a similar overall runtime behavior as for the random projections. Using a higher number of k-NN to search for increases the runtime. However,

the runtime does not increase as much as for the random projection hash functions. The runtimes using random projections vary between $5.8\,\mathrm{s}$ and $13.5\,\mathrm{s}$, whereas the runtimes for the entropy-based hash functions vary between $13.7\,\mathrm{s}$ and $16.9\,\mathrm{s}$.

The recall shown in Figure 6.16c and the error ratio shown in Figure 6.16d behave the same for the entropy-based hash functions as for the random projections. In both cases the recall decreases and the error ratio increases with a growing value of `k`. Not only the characteristics are the same, but also the resulting recalls and error ratios are identical.

6.2.3 Comparison between Random Projections and Entropy-Based Hash Functions

The results show that both hash function types can be used to achieve similar results with comparable runtimes for both data sets. Only the step for creating the entropy-based hash functions should further be investigated to eliminate the static overhead when using ComputeCpp as SYCL implementation, and generally improve the construction step.

The entropy-based hash functions have the advantage that to reach the same results fewer hash tables were needed. The tests using random projections used between 10 and 40 hash tables. However, the entropy-based hash function tests only used up to 15 hash functions, resulting in less memory consumption on the GPUs.

The various runtime parameter tests showed that overall, both hash function types behave the same. The only exception being the `sycl_lsh::options::hash_pool_size` parameter caused by the expensive generation of the entropy-based hash functions. It is possible for both hash function types to sometimes drastically increase the runtime or worsen the recall and error ratio. However, it is also possible to adjust the LSH algorithm to yield only a recall of, for example, $60\,\%$. This could be enough for an approximate algorithm and has the advantage of a significantly lower runtime compared to higher recalls. Furthermore, even if the recall is relatively low, the error ratio is often considerably good, which again could be enough in specific scenarios.

6.3 Scaling Behavior on multiple GPUs

The following section covers the scaling characteristics of the `sycl_lsh` library implementation to verify whether the implemented distributed multi-GPU support works as intended. For both hash function types the runtimes for one up to eight NVIDIA GEFORCE GTX 1080 Ti are collected using ComputeCpp and hipSYCL as SYCL implementations and for the `friedman` and `HIGGS` data sets. The presented runtimes are the sum of the runtimes for the creation of the hash functions, the construction of the hash tables and nearest neighbor search. Additionally, the parallel speedups S_p for all above combinations are reported using the formula

$$S_p = \frac{T_1}{T_p} \tag{6.1}$$

where T_1 denotes the runtime using only one GPU and T_p represents the runtime when using p GPUs. Each dot represents the average over ten different runs.

6.3.1 Scaling Behavior using Random Projections

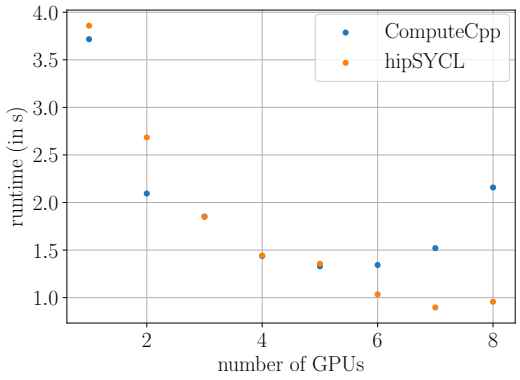
Figure 6.17 shows the runtime in the case of random projections on up to eight GPUs using the `friedman` and `HIGGS` data set.

The parameters used for the scaling tests in the case of the `friedman` data set are the same as in Table 6.2. Figure 6.17a shows that the runtime improves with a small number of data points. However, the runtime using the ComputeCpp implementation increases again when utilizing more than five GPUs. The runtime in the case of the hipSYCL implementation only increases when using eight GPUs.

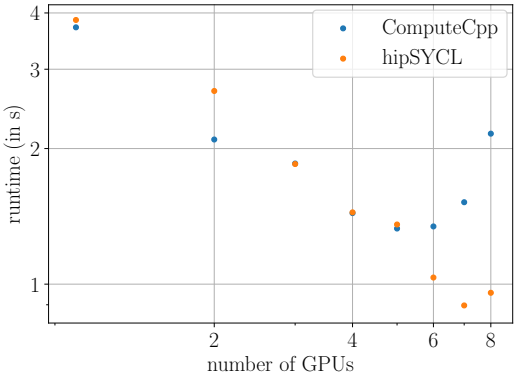
In the logarithmic plot in Figure 6.17b, can be seen that the scaling behaves nearly optimal for the hipSYCL implementation using up to seven GPUs. However, the ComputeCpp implementation behaves suboptimal when using more than five GPUs. The runtime then increases from $1.3\,\mathrm{s}$ back to $2.1\,\mathrm{s}$. This is also shown in Figure 6.17e. The parallel speedup using ComputeCpp and the `friedman` data set when utilizing eight GPUs is only 1.7 compared to the optimal speedup of 8. When using hipSYCL, the parallel speedup of 4.0 is better in contrast to the ComputeCpp implementation. However, even a parallel speedup of 4.0 considerably bad compared to the theoretically achievable speedup of 8.

Both implementations indicate that the `friedman` data set is too small to fully utilize eight GPUs.

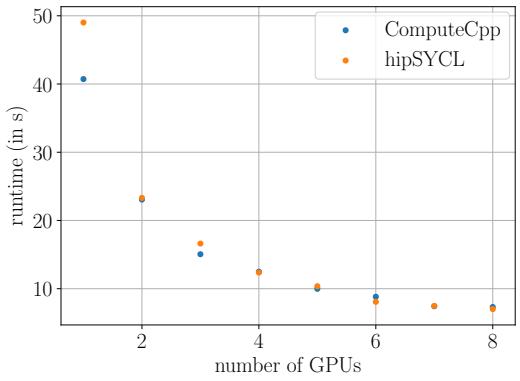
Figure 6.17c and Figure 6.17d represent the same scaling tests, however, this time, the `HIGGS` data set is used instead of the `friedman` data set. Therefore, other runtime parameters are used: 40 hash tables with 20 hash functions per hash table, a segment size `w` of 1.5, the default hash table size, and a hash pool size of 300. In addition, five nearest neighbors were searched. This time, both SYCL implementations show the same behavior. Increasing the number of used GPUs improves the runtime. As shown in Figure 6.17d, the



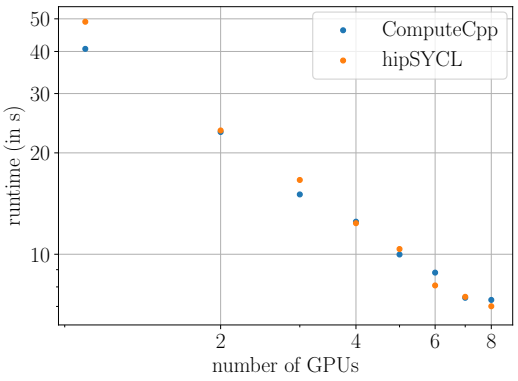
(a) The runtime depending on the number of GPUs using the `friedman` data set.



(b) The runtime depending on the number of GPUs using the `friedman` data set and a logarithmic plot.



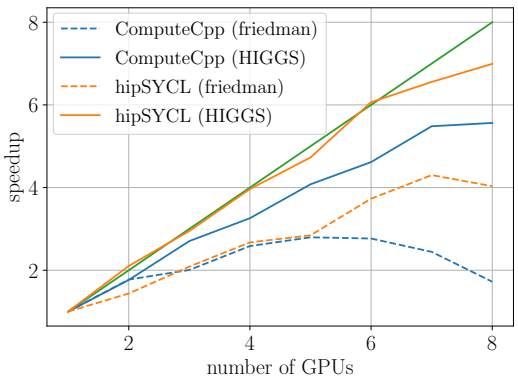
(c) The runtime depending on the number of GPUs using the `higgs` data set.



(d) The runtime depending on the number of GPUs using the `higgs` data set and a logarithmic plot.

runtime decreases consistently until eight GPUs are used. Using one GPU together with the ComputeCpp SYCL implementation results in a runtime of $40\,\mathrm{s}$. Increasing the number of GPUs to eight reduces the runtime to $7.3\,\mathrm{s}$, whereas the optimal runtime would be $5\,\mathrm{s}$.

Compared to the `friedman` data set, the `HIGGS` data set is large enough to better utilize more GPUs. This is also shown in Figure 6.17e. The parallel speedup using the `HIGGS` data set is better for all number of GPUs compared to the parallel speedup of the `friedman` data set. This time the parallel speedup using hipSYCL with a value of 7.0 is significantly better compared to the speedup using the `friedman` data set. Again, the speedup for the ComputeCpp implementation with a value of 5.5 is worse compared to hipSYCL.



(e) The parallel speedup using random projections for both data sets and SYCL implementations.

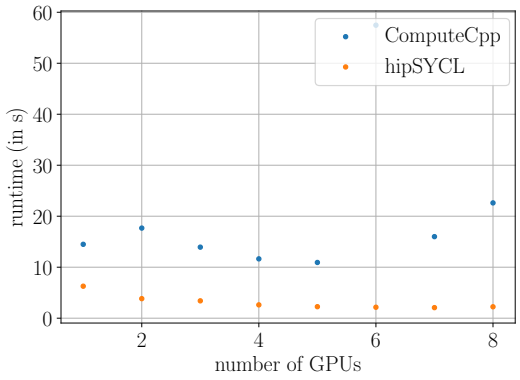
Figure 6.17: The LSH algorithm's behavior depending on the number of GPUs utilizing random projections. The presented values are obtained on up to eight NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp and hipSYCL as SYCL implementations, and the `friedman` and `HIGGS` data sets. All dots are averages over ten test samples.

6.3.2 Scaling Behavior using Entropy-Based Hash Functions

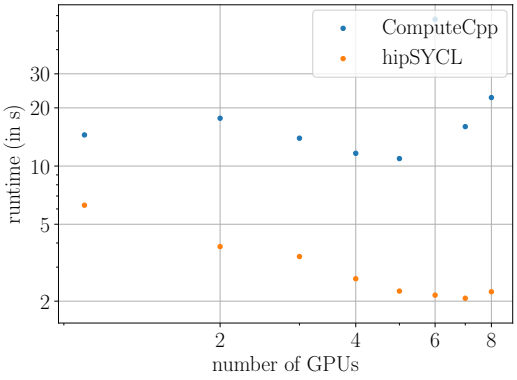
Figure 6.18 shows the runtime using the entropy-based hash functions on up to eight GPUs using the `friedman` and `HIGGS` data set.

The parameters used for the scaling test of the `friedman` data set are the same as in Table 6.3. Figure 6.18a shows that the runtime does not scale well when using multiple GPUs. Using the hipSYCL SYCL implementation, the runtime only drops from $6.2\,\mathrm{s}$ utilizing a single GPU to $2.2\,\mathrm{s}$ when using eight GPUs. The logarithmic plot in Figure 6.18b, shows that the runtimes do scale when using up to seven GPUs, although badly. The ComputeCpp version has an even worse runtime when using eight GPUs ($22.6\,\mathrm{s}$) compared to using only one GPU ($14.4\,\mathrm{s}$). In general, ComputeCpp has worse overall runtimes than the hipSYCL implementation because of the already observed static overhead during the creation of the hash functions. This static overhead also results in a parallel speedup of only 0.64 indicating that the runtime increases when using more GPUs. Although the parallel speedup for the hipSYCL implementation is greater than 1.0, it is worse compared to the speedup using the random projections and the `friedman` data set with a value of 2.8.

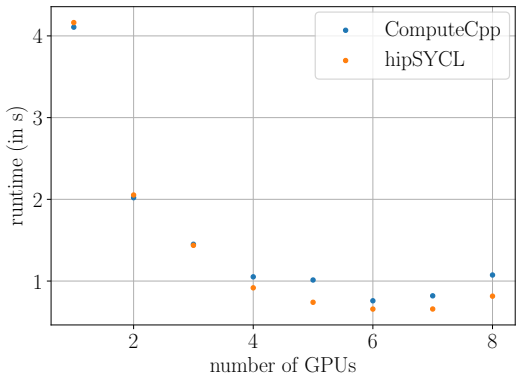
Plotting only the runtime for the construction of the hash tables and the k-NN search in Figure 6.18c and Figure 6.18d shows that both SYCL implementations behave roughly the same and have the same total runtimes. In both cases, using more than six GPUs does not result in a runtime improvement, it actually increases the runtimes again. Compared to random projection hash functions, the behavior and runtimes are similar. It is interesting



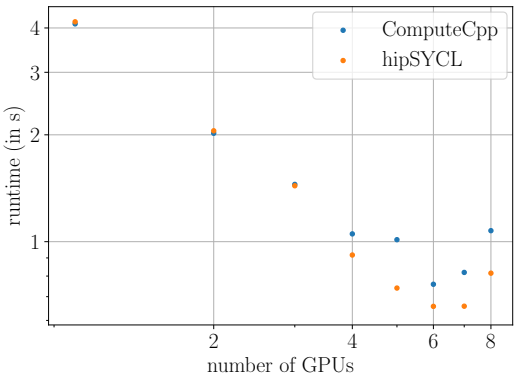
(a) The runtime depending on the number of GPUs using the `friedman` data set.



(b) The runtime depending on the number of GPUs using the `friedman` data set and a logarithmic plot.



(c) The runtime depending on the number of GPUs using the `friedman` data set without the runtime for the creation of the hash functions.

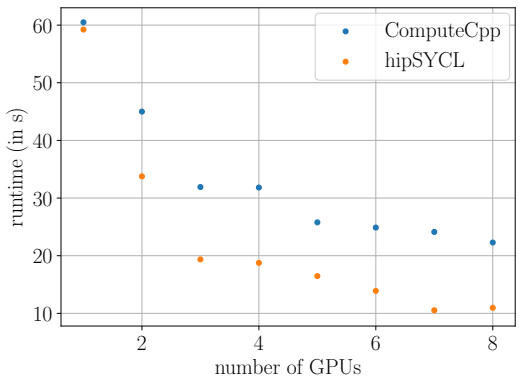


(d) The runtime depending on the number of GPUs using the `friedman` data set and a logarithmic plot without the runtime for the creation of the hash functions.

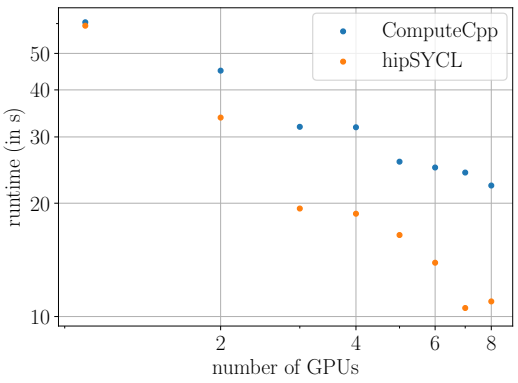
to see that when comparing both hash functions, the ComputeCpp version's runtime starts to increase earlier compared to hipSYCL and finishes on a higher runtime using eight GPUs. Disregarding the runtimes for the creation of the hash functions also improves the parallel speedup. Using up to six GPUs results in a parallel speedup of 5.4, when using ComputeCpp, and a speedup of 6.3, when using hipSYCL, which is even above the theoretical speedup of 6. Adding more GPUs reduces the parallel speedup.

As for the random projections, these results indicate that the `friedman` data set is too small to fully utilize all eight GPUs.

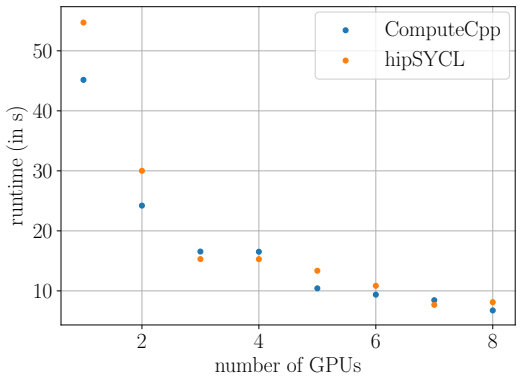
The remaining figures show the same scaling tests using the `HIGGS` data set instead of the `friedman` data set. For this purpose, the runtime parameters were adjusted accordingly:



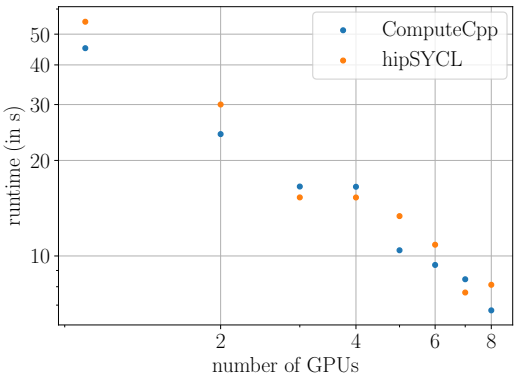
(e) The runtime depending on the number of GPUs using the `higgs` data set.



(f) The runtime depending on the number of GPUs using the `higgs` data set and a logarithmic plot.



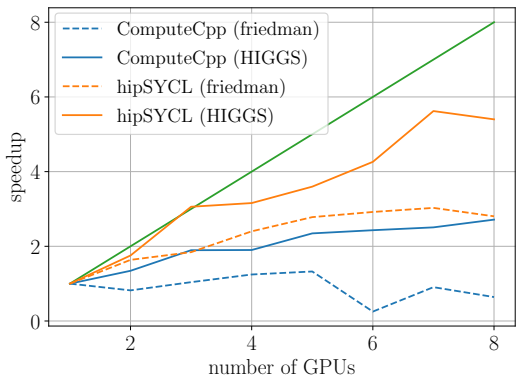
(g) The runtime depending on the number of GPUs using the `higgs` data set without the runtime for the creation of the hash functions.



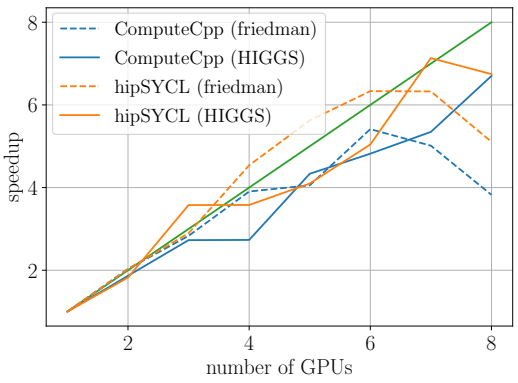
(h) The runtime depending on the number of GPUs using the `higgs` data set and a logarithmic plot without the runtime for the creation of the hash functions.

eleven hash tables with six hash functions each, four cut-off points, the default hash table size, a hash pool size of 50, and `k = 5`.

Again, using the ComputeCpp implementation, an overhead across all tests can be seen in Figure 6.18e. However, using the `HIGGS` data set, the runtime does not increase even when using eight GPUs. Figure 6.18f shows a better scaling behavior using the `HIGGS` data set than the `friedman` data set even if the runtime for the hash function creation is considered. Figure 6.18i shows that the speedup for both SYCL implementations is better when using the `HIGGS` data set compared to the `friedman` data set even if the construction of the hash functions is considered. However, for example, the parallel speedup using the ComputeCpp implementation is still only 2.7.



(i) The parallel speedup using entropy-based hash functions for both data sets and SYCL implementations.



(j) The parallel speedup using entropy-based hash functions for both data sets and SYCL implementations without the runtime for the creation of the hash functions.

Figure 6.18: The LSH algorithm's behavior depending on the number of GPUs utilizing entropy-based hash functions. The presented values are obtained on NVIDIA GEFORCE GTX 1080 Ti using ComputeCpp and hipSYCL as SYCL implementations, and the `friedman` and `HIGGS` data sets. All dots are averages over ten samples.

Disregarding the runtime required for the creation of the hash functions results in the scaling behavior in Figure 6.18g and Figure 6.18h. As can be seen, both SYCL implementations behave the same with comparable runtimes. Increasing the number of GPUs from one to eight reduces the runtime using the ComputeCpp implementation from $45.1\,\mathrm{s}$ down to $6.7\,\mathrm{s}$ and using the hipSYCL implementation from $54.7\,\mathrm{s}$ down to $8.1\,\mathrm{s}$. Interestingly, both implementations have no runtime improvement when increasing the number of GPUs from three to four. Figure 6.18j shows the parallel speedup when disregarding the runtimes for the creation of the hash functions. The speedup is comparable to the speedup using the `friedman` data set, however, even for more than six GPUs the parallel speedup increases when using the `HIGGS` data set. When using eight GPUs both SYCL implementations have a parallel speedup of 6.7.

As for the random projections, the `HIGGS` data set is large enough to utilize all eight GPUs.

Summarizing, it can be said that the `sycl_lsh` library implementation does scale well on multiple GPUs when using a sufficiently large data set.

7 Conclusion

In the course of this master thesis, a new library called `sycl_lsh` has been developed and implemented to solve the problem of the *k*-nearest neighbors (k-NN) using the Locality-Sensitive Hashing (LSH) algorithm for the Euclidean distance utilizing the random projections and entropy-based locality-sensitive hash functions. To enable the usage of large data sets or to speed up the computation of small ones, the `sycl_lsh` library supports multiple GPUs using MPI to enable the usage of both shared and distributed memory systems. Furthermore, SYCL has been used to support different hardware architectures using a single source code.

The LSH algorithm can be used to efficiently calculate the k-NN of a data point. However, the parameters must be selected carefully since the wrong parameter choice can drastically increase the runtime. Nevertheless, the parameters can also be seen as an advantage of the LSH algorithm since they can be used to adjust the resulting recall and adopt to the runtime requirements. For that the knowledge of the influence of the various runtime parameters is crucial. If a recall of $60\,\%$ is sufficient, the parameters can be selected accordingly, resulting in a reduced runtime compared to, for example, $90\,\%$. Because the hash functions are created randomly, and the resulting recall and error ratio depend on these hash functions, the runtime and recall can fluctuate even for the same parameters.

Both locality-sensitive hash function types can be used to achieve similar results. However, the runtimes of the entropy-based hash functions can be further improved by a more efficient hash function generation.

The results indicate that the distributed Multi-GPU implementation of the `sycl_lsh` library scales well with the number of GPUs used. It was possible to decrease the runtime of the LSH algorithm the more GPUs were used given a sufficiently large data set. When using the `HIGGS` data set, consisting of $1\,000\,000$ data points in 27 dimensions, a parallel speedup using random projections of 5.5 (ComputeCpp) and 7.0 (hipSYCL) could be achieved utilizing eight GPUs. Switching to the entropy-based hash functions resulted in a parallel speedup of 6.7 for both ComputeCpp and hipSYCl.

The usage of SYCL as an abstraction layer enabled the `sycl_lsh` library to target different hardware, NVIDIA GEFORCE GTX 1080 Ti, Intel UHD Graphics P630 Gen9, and Intel Iris X^e MAX GPUs, only by using standard conform `C++`. Thereby, it has also been shown that the three different SYCL implementations, ComputeCpp, hipSYCL, and oneAPI, do not have significant performance differences considering the `sycl_lsh` library implementation.

8 Future Work

Currently, the Euclidean distance with the two locality-sensitive hash functions random projections and entropy-based hash functions is the only available distance metric. Other distance metrics could be implemented in the next step, such as the Manhattan distance using random projections proposed by Datar et al. [DIIM04], the Cosine Similarity using SimHash proposed by Charikar [Cha02], the Jaccard distance using MinHash proposed by Broder [Bro], or the Hamming distance using BitSampling proposed by Indyk and Motwani [IM98].

The `sycl_lsh` library uses the standard LSH algorithm. Other LSH variants have been proposed as described in Section 2.1.1. One of these approaches is the multi-probe LSH algorithm proposed by Lv et al. [LJW+07], which tries to reduce the number of needed hash tables. A preliminary implementation has been developed for the `sycl_lsh` library. While the results looked promising, this proof of concept implementation has not been merged into the `sycl_lsh` library yet since further refactoring and tests are necessary.

Furthermore, few tests with other hash combine functions have been performed. The results were that changing the hash combine function drastically influences the expected performance, recall, and error ratio. Therefore, further investigations should be performed. One possible custom hash function type has already been implemented called `sycl_lsh::hash_functions_type::mixed_hash_functions`. It combines the random projections with the entropy-based hash functions. The random projections are used as normal hash functions, and an entropy-based hash function combines these hash functions to the resulting hash signature. However, no further tests have been conducted yet.

A more efficient distributed sort algorithm could be implemented to improve the current performance of the entropy-based hash function creation.

Additionally, more runtime and scaling tests using different hardware, e.g., AMD GPUs, can be performed, together with tests using larger data sets on bigger clusters. A large-scale test run using a bigger data set on the bwUniCluster[1] could not be achieved in time because the long queue times made it impossible to debug a problem with the MPI installation. However, the presented results demonstrated that the distributed implementation works.

[1] https://wiki.bwhpc.de/e/Category:BwUniCluster_2.0

A CMake Configuration Options

The following list contains all custom CMake configuration options.

SYCL_LSH_IMPLEMENTATION
: Specify the used SYCL implementation. Must be one of: `hipSYCL` (default), `ComputeCpp` or `oneAPI`.

SYCL_LSH_TARGET
: Specify the SYCL target to compile for. Must be one of: `CPU`, `NVIDIA` (default), `AMD` or `INTEL`. In case of `NVIDIA`, `AMD` or `INTEL` the device is of type GPU.

SYCL_LSH_TIMER
: Specify which timer functionality should be used. Must be one of: `NONE` (disables timing altogether), `NON_BLOCKING` (enables timing, but without barriers after kernel invocations or MPI communications) or `BLOCKING` (enables timing with barriers; default).

SYCL_LSH_BENCHMARK
: If defined, enables benchmarking by logging the elapsed times in a machine readable way (.csv format) to the provided file. The value must be a valid file name.

SYCL_LSH_ENABLE_DEBUG
: Enables better debugging support. Defines debugging macros and removes the seeding from the random number generators to produces more deterministic results.

SYCL_LSH_ENABLE_DOCUMENTATION
: Enables the documentation target `make doc`. Requires doxygen as dependency.

SYCL_LSH_FMT_HEADER_ONLY
: Enables the header only mode for the required `{fmt}` formatting library. Otherwise tries to link against it.

SYCL_LSH_USE_EXPERIMENTAL_FILESYSTEM
: Enables the `<experimental/filesystem>` header instead of the `C++17 <filesystem>` header.

B Command Line Options

The following list contains all recognized command line options. Note that all options have the form `--option value`.

help
Prints the help screen, listing all recognized command line options.

data_file
Path to the file containing the data points. This option is required.

file_parser
The type of the file parser to parse the data file. Must be one of `arff_parser` or `binary_parser` (default).

k The number of nearest neighbors to search for. This option is required.

options_file
Path to the file containing options for the `syl_lsh::options` class.

options_save_file
Save the currently used options to the specified file.

knn_save_file
Save the calculated *k*-nearest neighbors to the specified file.

knn_dist_save_file
Save the calculated *k*-nearest neighbors distances to the specified file.

evaluate_knn_file
Path to the file containing the correct *k*-nearest neighbors for the current data set. If this option is present, the recall of the calculated *k*-nearest neighbors will be determined.

evaluate_knn_dist_file
Path to the file containing the correct *k*-nearest neighbors distances for the current data set. If this option is present, the error ratio of the calculated *k*-nearest neighbors will be determined.

hash_pool_size
The number of hash functions in the hash pool.

num_hash_functions
The number of hash functions used for the hash signature for each LSH hash table.

num_hash_tables
The number of used LSH hash tables.

hash_table_size
The size of each LSH hash table.

w The segment size for the random projections hash functions $h(x) = \left\lfloor \frac{a \cdot x + b}{w} \right\rfloor$. Only used if the `used_hash_functions_type` is `sycl_lsh::hash_functions_type::random_projections`.

num_cut_off_points
The number of cut-off points for the entropy-based hash functions. Only used if the `used_hash_functions_type` is `sycl_lsh::hash_functions_type::entropy_based`.

Bibliography

[AH20] A. Alpay, V. Heuveline. “SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL.” In: *Proceedings of the International Workshop on OpenCL*. ACM, Apr. 2020. DOI: 10.1145/3388333.3388658 (cit. on p. 39).

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Y. Wu. “An optimal algorithm for approximate nearest neighbor searching fixed dimensions.” In: *Journal of the ACM* 45.6 (Nov. 1998), pp. 891–923. DOI: 10.1145/293347.293348 (cit. on p. 20).

[ARN17] A. Andoni, I. Razenshteyn, N. S. Nosatzki. “LSH Forest: Practical Algorithms Made Theoretical.” In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2017. DOI: 10.1137/1.9781611974782.5 (cit. on p. 20).

[BCG05] M. Bawa, T. Condie, P. Ganesan. “LSH forest.” In: *Proceedings of the 14th international conference on World Wide Web - WWW ’05*. ACM Press, 2005. DOI: 10.1145/1060745.1060840 (cit. on p. 20).

[Ben75] J. L. Bentley. “Multidimensional binary search trees used for associative searching.” In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. DOI: 10.1145/361002.361007 (cit. on p. 20).

[BGS12] B. Bahmani, A. Goel, R. Shinde. “Efficient distributed locality sensitive hashing.” In: *Proceedings of the 21st ACM international conference on Information and knowledge management - CIKM ’12*. ACM Press, 2012. DOI: 10.1145/2396761.2398596 (cit. on p. 21).

[BHSV19] D. Beckingsale, R. Hornung, T. Scogland, A. Vargas. “Performance portable C++ programming with RAJA.” In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 455–456 (cit. on p. 22).

[BKL06] A. Beygelzimer, S. Kakade, J. Langford. “Cover trees for nearest neighbor.” In: *Proceedings of the 23rd international conference on Machine learning - ICML ’06*. ACM Press, 2006. DOI: 10.1145/1143844.1143857 (cit. on p. 20).

[BL] J. Beis, D. Lowe. “Shape indexing using approximate nearest-neighbour search in high-dimensional spaces.” In: *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Comput. Soc. DOI: 10.1109/cvpr.1997.609451 (cit. on p. 20).

[Bre18] M. Breyer. “Ein hoch-performanter (approximierter) k-Nächste-Nachbarn Algorithmus für GPUs.” Apr. 2018 (cit. on p. 18).

[Bro] A. Broder. “On the resemblance and containment of documents.” In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. IEEE Comput. Soc. DOI: 10.1109/sequen.1997.666900 (cit. on p. 101).

[BSW14] P. Baldi, P. Sadowski, D. Whiteson. “Searching for exotic particles in high-energy physics with deep learning.” In: *Nature Communications* 5.1 (July 2014). DOI: 10.1038/ncomms5308 (cit. on p. 62).

[CH67] T. Cover, P. Hart. “Nearest neighbor pattern classification.” In: *IEEE Transactions on Information Theory* 13.1 (Jan. 1967), pp. 21–27. DOI: 10.1109/tit.1967.1053964 (cit. on pp. 17, 23).

[Cha02] M. S. Charikar. “Similarity estimation techniques from rounding algorithms.” In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC ’02*. ACM Press, 2002. DOI: 10.1145/509907.509965 (cit. on p. 101).

[DF08] S. Dasgupta, Y. Freund. “Random projection trees and low dimensional manifolds.” In: *Proceedings of the fourtieth annual ACM symposium on Theory of computing - STOC 08*. ACM Press, 2008. DOI: 10.1145/1374376.1374452 (cit. on p. 20).

[DIIM04] M. Datar, N. Immorlica, P. Indyk, V. S. Mirrokni. “Locality-sensitive hashing scheme based on p-stable distributions.” In: *Proceedings of the twentieth annual symposium on Computational geometry - SCG ’04*. ACM Press, 2004. DOI: 10.1145/997817.997857 (cit. on pp. 26, 30, 101).

[EKNT12] I. Z. Emiris, A. Konstantinakis-Karmis, D. Nicolopoulos, A. Thanos-Filis. “Data structures for approximate nearest neighbor search.” In: *Techn. Ber. Technical Report CGLTR-29, NKUA* (2012) (cit. on p. 20).

[EMK+20] C. Eiras-Franco, D. Martinez-Rego, L. Kanthan, C. Pineiro, A. Bahamonde, B. Guijarro-Berdinas, A. Alonso-Betanzos. “Fast Distributed kNN Graph Construction Using Auto-tuned Locality-sensitive Hashing.” In: *ACM Transactions on Intelligent Systems and Technology* 11.6 (Nov. 2020), pp. 1–18. DOI: 10.1145/3408889 (cit. on p. 20).

[EN13] I. Z. Emiris, D. Nicolopoulos. “Randomized kd-trees for approximate nearest neighbor search.” In: *CGL-TR-78, NKUA, Tech. Rep.* (2013) (cit. on p. 20).

[ETS14] H. C. Edwards, C. R. Trott, D. Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns.” In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257 (cit. on p. 21).

[FBF77] J. H. Friedman, J. L. Bentley, R. A. Finkel. “An Algorithm for Finding Best Matches in Logarithmic Expected Time.” In: *ACM Transactions on Mathematical Software* 3.3 (Sept. 1977), pp. 209–226. DOI: 10.1145/355744.355745 (cit. on p. 20).

[FN75] K. Fukunaga, P. Narendra. “A Branch and Bound Algorithm for Computing k-Nearest Neighbors.” In: *IEEE Transactions on Computers* C-24.7 (July 1975), pp. 750–753. DOI: 10.1109/t-c.1975.224297 (cit. on p. 20).

[Fri91] J. H. Friedman. “Multivariate Adaptive Regression Splines.” In: *The Annals of Statistics* 19.1 (Mar. 1991), pp. 1–67. DOI: 10.1214/aos/1176347963 (cit. on p. 62).

[GHKS13] T. Ge, K. He, Q. Ke, J. Sun. “Optimized Product Quantization for Approximate Nearest Neighbor Search.” In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2013. DOI: 10.1109/cvpr.2013.379 (cit. on p. 20).

[HASZ11] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, H. Zhang. “Fast approximate nearest-neighbor search with k-nearest neighbor graph.” In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011 (cit. on p. 20).

[HGK+18] J. Hoberock, M. Garland, C. Kohlhoff, C. Mysen, C. Edwards, G. Brown, M. Wong. *Executors Design Document (P0761R2)*. Tech. rep. 2018 (cit. on p. 22).

[HGK+20] J. Hoberock, M. Garland, C. Kohlhoff, C. Mysen, C. Edwards, G. Brown, D. Hollman, L. Howes, K. Shoop, L. Baker, E. Niebler. *A Unified Executors Proposal for C++ (P0443R14)*. Proposal P0443R12. 2020 (cit. on p. 22).

[HK19] J. Hoberock, C. Kohlhoff. *C++20 Executors are Resilient to ABI Breakage (P1405R0)*. Tech. rep. 2019 (cit. on p. 22).

[HLY19] J.-P. Heo, Z. Lin, S.-E. Yoon. “Distance Encoded Product Quantization for Approximate K-Nearest Neighbor Search in High-Dimensional Space.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.9 (Sept. 2019), pp. 2084–2097. DOI: 10.1109/tpami.2018.2853161 (cit. on p. 20).

[HMA+08] P. Haghani, S. Michel, K. Aberer, et al. “Lsh at large-distributed knn search in high dimensions.” In: *11th International Workshop on the Web and Databases, WebDB*. CONF. 2008 (cit. on p. 21).

[HMA09] P. Haghani, S. Michel, K. Aberer. “Distributed similarity search in high dimensions using locality sensitive hashing.” In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 2009, pp. 744–755 (cit. on p. 21).

[IM98] P. Indyk, R. Motwani. “Approximate nearest neighbors.” In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC ’98*. ACM Press, 1998. DOI: 10.1145/276698.276876 (cit. on pp. 17, 25, 30, 33, 101).

[JB08] A. Joly, O. Buisson. “A posteriori multi-probe locality sensitive hashing.” In: *Proceeding of the 16th ACM international conference on Multimedia - MM ’08*. ACM Press, 2008. DOI: 10.1145/1459359.1459388 (cit. on p. 19).

[JDS11] H. Jégou, M. Douze, C. Schmid. “Product Quantization for Nearest Neighbor Search.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (Jan. 2011), pp. 117–128. DOI: 10.1109/tpami.2010.57 (cit. on p. 20).

[Jen96] B. Jenkins. *Hash Functions*. Tech. rep. Dr Dobbs article, 1996 (cit. on p. 51).

[Jos18] N. Josuttis. *hash_combine() Again (P0814R2)*. Tech. rep. 2018 (cit. on p. 51).

[KA14] Y. Kalantidis, Y. Avrithis. "Locally Optimized Product Quantization for Approximate Nearest Neighbor Search." In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2014. DOI: 10.1109/cvpr.2014.298 (cit. on p. 20).

[Khr20] Khronos OpenCL Working Group. "The OpenCL Specification." In: 2020 (cit. on p. 21).

[LJW+07] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li. "Multi-probe LSH: efficient indexing for high-dimensional similarity search." In: *Proceedings of the 33rd international conference on Very large data bases*. 2007, pp. 950–961 (cit. on pp. 19, 101).

[LMYG05] T. Liu, A. W. Moore, K. Yang, A. G. Gray. "An investigation of practical approximate nearest neighbor algorithms." In: *Advances in neural information processing systems*. 2005, pp. 825–832 (cit. on p. 20).

[LRU14] J. Leskovec, A. Rajaraman, J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014. DOI: 10.1017/cbo9781139924801 (cit. on p. 30).

[LWDP01] L. Li, C. R. Weinberg, T. A. Darden, L. G. Pedersen. "Gene selection for sample classification based on gene expression data: study of sensitivity to choice of parameters of the GA/KNN method." In: *Bioinformatics Vol. 17 no. 12* (2001) (cit. on p. 17).

[LZ09] Y. Lifshits, S. Zhang. "Combinatorial Algorithms for Nearest Neighbors, Near-Duplicates and Small-World Design." In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2009. DOI: 10.1137/1.9781611973068.36 (cit. on p. 20).

[Mai07] F. Mainardi. "Lévy Stable Distributions in the Theory of Probabilty." In: (2007) (cit. on p. 26).

[ML09] M. Muja, D. G. Lowe. "Fast approximate nearest neighbors with automatic algorithm configuration." In: *VISAPP (1)* 2.331-340 (2009), p. 2 (cit. on p. 20).

[ML14] M. Muja, D. G. Lowe. "Scalable Nearest Neighbor Algorithms for High Dimensional Data." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (Nov. 2014), pp. 2227–2240. DOI: 10.1109/tpami.2014.2321376 (cit. on p. 20).

[NK16] H. Neeb, C. Kurrus. *Distributed k-nearest neighbors*. 2016 (cit. on p. 21).

[Nol18] J. P. Nolan. *Stable Distributions: Models for Heavy Tailed Data*. Math/Stat Department American University, 2018 (cit. on p. 26).

[Pan06] R. Panigrahy. “Entropy based nearest neighbor search in high dimensions.” In: *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm - SODA '06*. ACM Press, 2006. DOI: 10.1145/1109557.1109688 (cit. on p. 19).

[PC05] R. Paredes, E. Chávez. “Using the k-Nearest Neighbor Graph for Proximity Searching in Metric Spaces.” In: *String Processing and Information Retrieval*. Springer Berlin Heidelberg, 2005, pp. 127–138. DOI: 10.1007/11575832_14 (cit. on p. 20).

[PM12] J. Pan, D. Manocha. “Bi-level Locality Sensitive Hashing for k-Nearest Neighbor Computation.” In: *2012 IEEE 28th International Conference on Data Engineering*. IEEE, Apr. 2012. DOI: 10.1109/icde.2012.40 (cit. on p. 19).

[PSS+16] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, P. Dubey, et al. “Panda: Extreme scale parallel k-nearest neighbor on distributed architectures.” In: *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE. 2016, pp. 494–503 (cit. on p. 21).

[Ron20] L. H. Ronan Keryell Maria Rovatsou. “SYCL Specification 1.2.1: SYCL integrates OpenCL devices with modern C++.” In: Revision: 7. Khronos SYCL Working Group. Apr. 2020 (cit. on pp. 18, 35).

[RS19] P. Ram, K. Sinha. “Revisiting kd-tree for Nearest Neighbor Search.” In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, July 2019. DOI: 10.1145/3292500.3330875 (cit. on p. 20).

[SH08] C. Silpa-Anan, R. Hartley. “Optimised KD-trees for fast image descriptor matching.” In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2008. DOI: 10.1109/cvpr.2008.4587638 (cit. on p. 20).

[SRMS20] R. G. da Silva, M. H. D. M. Ribeiro, V. C. Mariani, L. dos Santos Coelho. “Forecasting Brazilian and American COVID-19 cases based on artificial intelligence coupled with climatic exogenous variables.” In: *Chaos, Solitons & Fractals* 139 (Oct. 2020), p. 110027. DOI: 10.1016/j.chaos.2020.110027 (cit. on p. 17).

[SRSA20] W. M. Shaban, A. H. Rabie, A. I. Saleh, M. Abo-Elsoud. “A new COVID-19 Patients Detection Strategy (CPDS) based on hybrid feature selection and enhanced KNN classifier.” In: *Knowledge-Based Systems* 205 (Oct. 2020), p. 106270. DOI: 10.1016/j.knosys.2020.106270 (cit. on p. 17).

[STS+13] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, P. Dubey. “Streaming similarity search over one billion tweets using parallel locality-sensitive hashing.” In: *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013), pp. 1930–1941. DOI: 10.14778/2556549.2556574 (cit. on p. 21).

[Tan06] S. Tan. “An effective refinement strategy for KNN text classifier.” In: *Expert Systems with Applications 30* (2006) (cit. on p. 17).

[TMD14] B. Trstenjak, S. Mikac, D. Donko. “KNN with TF-IDF based Framework for Text Categorization.” In: *Procedia Engineering* 69 (2014), pp. 1356–1364. DOI: 10.1016/j.proeng.2014.03.129 (cit. on p. 17).

[TT] K. Terasawa, Y. Tanaka. “Spherical LSH for Approximate Nearest Neighbor Search on Unit Hypersphere.” In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 27–38. DOI: 10.1007/978-3-540-73951-7_4 (cit. on p. 20).

[TYSK10] Y. Tao, K. Yi, C. Sheng, P. Kalnis. “Efficient and accurate nearest neighbor and closest pair search in high-dimensional space.” In: *ACM Transactions on Database Systems* 35.3 (July 2010), pp. 1–46. DOI: 10.1145/1806907.1806912 (cit. on p. 20).

[WGLG12] Q. Wang, Z. Guo, G. Liu, J. Guo. “Entropy based locality sensitive hashing.” In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Mar. 2012. DOI: 10.1109/icassp.2012.6288065 (cit. on p. 28).

[WSB98] R. Weber, H.-J. Schek, S. Blott. “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces.” In: *VLDB*. Vol. 98. 1998, pp. 194–205 (cit. on p. 20).

[YYS09] Z. Yong, L. Youwen, X. Shixiong. “An Improved KNN Test Classification Algoeithm Based on Clustering.” In: *JOURNAL OF COMPUTERS, VOL. 4, NO. 3.* (2009) (cit. on p. 17).

[ZLJ12] C. Zhang, F. Li, J. Jestes. “Efficient parallel kNN joins for large data in MapReduce.” In: *Proceedings of the 15th international conference on extending database technology*. 2012, pp. 38–49 (cit. on p. 21).

[ZLXZ16] W. Zhang, D. Li, Y. Xu, Y. Zhang. “Shuffle-efficient distributed Locality Sensitive Hashing on spark.” In: *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, Apr. 2016. DOI: 10.1109/infcomw.2016.7562179 (cit. on p. 21).

[Zuz16] P. Zuzek. “Implementation of the SYCL Heterogeneous Computing Library: Masters Thesis: the 2nd Cycle Masters Study Programme Computer and Information Science.” In: (2016) (cit. on p. 40).

All links were last followed on December 1, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature