I, Lee A Carraher, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
A Parallel Algorithm for Query Adaptive, Locality Sensitive Hash Search

Student's name: Lee A Carraher

This work and its defense approved by:

Committee chair: Fred Annexstein, PhD
Committee member: Kenneth Berman, PhD
Committee member: Yizong Cheng, PhD
Committee member: Anca Ralescu, PhD
Lee A Carraher, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
A Parallel Algorithm for Query Adaptive, Locality Sensitive Hash Search

This work and its defense approved by:

Committee chair: Fred Annexstein, PhD
Committee member: Kenneth Berman, PhD
Committee member: Yizong Cheng, PhD
Committee member: Anca Ralescu, PhD
A CUDA Based Parallel Decoding Algorithm for the Leech
Lattice Locality Sensitive Hash Family

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the School of Electric and Computing Systems
of the College of Engineering and Applied Sciences

November 03, 2011

by

Lee A Carraher
BSCE, University of Cincinnati, 2008

Thesis Advisor and Committee Chair: Dr. Fred Annexstein
Abstract

Nearest neighbor search is a fundamental requirement of many machine learning algorithms and is essential to fuzzy information retrieval. The utility of efficient database search and construction has broad utility in a variety of computing fields. Applications such as coding theory and compression for electronic communication systems as well as use in artificial intelligence for pattern and object recognition. In this thesis, a particular subset of nearest neighbors is consider, referred to as c-approximate k-nearest neighbors search. This particular variation relaxes the constraints of exact nearest neighbors by introducing a probability of finding the correct nearest neighbor c, which offers considerable advantages to the computational complexity of the search algorithm and the database overhead requirements. Furthermore, it extends the original nearest neighbors algorithm by returning a set of k candidate nearest neighbors, from which expert or exact distance calculations can be considered. Furthermore thesis extends the implementation of c-approximate k-nearest neighbors search so that it is able to utilize the burgeoning GPGPU computing field. The specific form of c-approximate k-nearest neighbors search implemented is based on the locality sensitive hash search from the E2LSH package of Indyk and Andoni [1]. In this paper, the authors utilize the exceptional properties of the Leech Lattice [2], as a subspace quantizer for the locality sensitive hash families. The Leech Lattice is unique in that it provides the closest lattice packing of equal sized spheres in 24 dimensional space. In addition, advances from coding theory provide a very favorable decoding algorithm for finding the nearest lattice center to a query point in euclidean 24 dimensional space [3] [4]. The multilevel construction of the Leech Lattice provides an excellent opportunity for parallelization as it contains the minimization of many independent sub-lattice decodings resulting from the lattices exceptional symmetry among lattices. These decodings are additionally highly floating point computationally intensive, and because of which suggest a favorable implementation on GPGPU architectures such as NVIDIA's CUDA based framework. Further-
more, the overall construction of a locality sensitive hash based, nearest neighbors search algorithm, is able to be parallelized fairly efficiently as the hash decodings are completely independent of one another. The goal of this thesis is to present a CUDA optimized parallel implementation of a bounded distance Leech Lattice decoder [4] for use in query optimized c-approximate k-nearest neighbors using the locality sensitive hash framework of E2LSH. The system will be applied to the approximate image retrieval of SIFT transformed [5] image vectors.
Acknowledgments
# Contents

## 1 Introduction
- 1.0.1 Problem Statement ................................................. 1
- 1.0.2 Current Approaches .............................................. 3
- 1.0.3 Our Solution ..................................................... 4
- 1.1 Hypothesis .......................................................... 5
- 1.2 Thesis Overview .................................................... 5

## 2 Background
- 2.1 Nearest Neighbors .................................................. 7
- 2.2 Curse of Dimensionality ............................................ 9
- 2.3 Error Correcting Codes ........................................... 10
  - 2.3.1 Types of Channels ........................................... 11
  - 2.3.2 The Golay Code ............................................. 11
  - 2.3.3 Lattices .................................................. 12
  - 2.3.4 Codes to Lattices ........................................ 14
  - 2.3.5 ECC to Nearest Neighbors ............................... 16
- 2.4 CUDA GPGPU Framework .......................................... 17
  - 2.4.1 CUDA Framework ........................................ 19
- 2.5 SIFT ............................................................... 22

## 3 Related Work ......................................................... 24
- 3.1 Alternate Approximate KNN Frameworks .......................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>Standard Algorithms for Approximate Nearest Neighbors</td>
<td>24</td>
</tr>
<tr>
<td>3.1.2</td>
<td>GPU based KNN search</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Alternate LSH Functions</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1</td>
<td>LSH a comparison of hash function types and query mechanisms</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Query Adaptive LSH</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Spherical LSH</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Other Uses for LSH</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Adaptive Mean-shift Accelerated by LSH</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Overview of the Approach</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>LSH based c-Approximate r-Near Neighbor</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1</td>
<td>A Locality Sensitive Hash Function</td>
<td>29</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Nearest Neighbor Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Complexity Analysis</td>
<td>31</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Lattice Based LSH Variant</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>The Leech Lattice</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Brief History</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2</td>
<td>$\Lambda_{24}$ as an Error Correcting Code</td>
<td>34</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Decoding and Bounded Distance Decoding</td>
<td>34</td>
</tr>
<tr>
<td>4.2.4</td>
<td>As an LSH Function</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>LSH c-Approximate k-Nearest Neighbors</td>
<td>35</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Sequential Algorithm Improvements</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Parallel Lattice Decoding</td>
<td>37</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Improvements</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>Implementation Details</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Parallel Lattice Decoder Implementation</td>
<td>39</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Overview</td>
<td>39</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Implementation of The Sequential Decoder</td>
<td>45</td>
</tr>
</tbody>
</table>
CONTENTS

5.1.3 Implementation of Lattice Based LSH ........................................ 49
5.1.4 Modifications for Parallel Implementation ............................. 50
5.1.5 SIFT Application Search ..................................................... 57

6 Performance Analysis ......................................................... 61
6.1 Performance of the CUDA Design ......................................... 61
6.2 Performance Analysis of the Leech Lattice ............................ 65
   6.2.1 Why the Leech Lattice .................................................. 66
   6.2.2 KNN-LSH and Exhaustive Search of SIFT Vectors .............. 69

7 Conclusions & Future Research ............................................... 75
7.1 Summary of Findings .......................................................... 75
   7.1.1 Parallel Lattice Decoder .............................................. 75
   7.1.2 LSH-KNN Speedup ..................................................... 76
   7.1.3 E8 vs Leech ............................................................. 76
   7.1.4 Comparison to Exhaustive Search of SIFT Vectors ............ 76
7.2 Detailed Conclusions .......................................................... 76
   7.2.1 Parallel Lattice Decoding .............................................. 77
   7.2.2 LSH-KNN Speedup ..................................................... 77
   7.2.3 Comparison to E8 and other Lattices .............................. 80
7.3 Suggestions for Future Work ................................................. 83
   7.3.1 Other Applications of Parallel LSH ................................. 83
   7.3.2 Where Locality Sensitive Hash Functions Really Shine ....... 83
   7.3.3 E8 for CUDA ........................................................... 84
   7.3.4 Getting a Better Sense of the Data through Variance Testing 84
## List of Figures

2.3.1 Hexagonal packing in $\mathbb{R}^2$ .................................................. 13
2.3.2 Performance of Some Coded and Unencoded Data Transmission Schemes ............................................ 16
2.4.1 Transistor Allocation for CPU vs. GPU Architectures ................................................................. 18
2.4.2 Example of Scalable Block Scheduling ............................................................... 20
2.5.1 Top image is a scene and the bottom contains the SIFT image matches .......................... 23

4.1.1 Random Projection of $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ .................................................. 29

5.1.1 Quadrature Amplitude Modulation ............................................................. 41
5.1.2 Ungerbock .......................................................... 41
5.1.3 Leech Lattice Decoder ............................................................. 44
5.1.4 E2LSH Framework: Query ............................................................. 52
5.1.5 E2LSH Framework: Preprocessing .................................................. 53
5.1.6 Memory Overlap for 4 $Q\Lambda_{24}$ decoders ............................................. 57
5.1.7 Parallel Leech Lattice Decoder ............................................................. 58

6.1.1 CUDA Blocks per Speedup ............................................................. 62
6.1.2 Ratio of Parallel to Total Time for Queries on an LSH Search Database using Real World SIFT Sample Data ................................................................. 63
6.1.3 Ratio of Parallel to Total Time for the Generation of the LSH Search Database using Real World SIFT Sample Data ................................................................. 64
6.2.1 Performance of Some Coded and Unencoded Data Transmission Schemes .......................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2</td>
<td>Intersection Probability Algorithm 1 vs. Algorithm 2 Hexacode Bounded Distance Decoder</td>
<td>66</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Correct Decoding Probability Rate of k-E8 sub projections compared to the Leech Lattices Decoding Rate</td>
<td>68</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Correct Decoding Probability Rate of k-E8 sub projections compared to the Leech Lattices Decoding Rate, (Zoomed)</td>
<td>68</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Elapsed Time for SIFT Vector Image Query</td>
<td>70</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Extrapolated Average Search Query Times</td>
<td>71</td>
</tr>
<tr>
<td>6.2.7</td>
<td>Fit of ( n ) for Linear Search Data</td>
<td>72</td>
</tr>
<tr>
<td>6.2.8</td>
<td>Fit of ( n^\rho ) for LSH Search Data</td>
<td>72</td>
</tr>
<tr>
<td>6.2.9</td>
<td>Recall Accuracy of Random SIFT Vector Image Query</td>
<td>73</td>
</tr>
<tr>
<td>6.2.10</td>
<td>Data Recall Selectivity for Query Adaptive and Unranked LSH</td>
<td>74</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Theoretical Speedup for Serial and Parallel Code as a function of Parallel Speedup for Database Query</td>
<td>78</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Theoretical Speedup for Serial and Parallel Code as a function of Parallel Speedup for Database Generation</td>
<td>79</td>
</tr>
<tr>
<td>7.3.1</td>
<td>KNN High Level System Diagram</td>
<td>85</td>
</tr>
<tr>
<td>7.3.2</td>
<td>KNN High Level System Diagram</td>
<td>86</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Weight Distributions of the $G_{24}$ Code .................................................. 15
2.2 GPU Memory Organization ................................................................. 21
5.1 Projections of $H_6 \rightarrow GF(2)$ ......................................................... 43
5.2 Memory Requirements of Lattice Decoding Data Structures ....................... 55
5.3 Compressed Data Structures Requirements for Lattice Decoding ............... 56
5.4 Quarter Lattice Data Structure Requirements ...................................... 57
6.1 Results of SIFT Search Accuracy ......................................................... 72
7.1 Various Lattices and their ECC Attributes ........................................... 82
Chapter 1

Introduction

1.0.1 Problem Statement

As the deluge of data acquisition continues to escalate, so to does our need to be able to search, organize and interpret this data. New sources of data are created almost daily and often exists in highly heterogeneous formats. The variety of forms that data can exist in often corresponds to the method by which it was gathered. For real world inputs, this presents tremendous diversity as most data acquisition technologies are highly specific to their data-streams and are often further separated by the various methods used for human interpretation. One such example of diversity is in the fundamental differences between data recorded in different mathematical domains. A data-stream that exists in the time domain may have little or no self relation in the frequency domain or vice versa. This example is of course very broad, and yet still a general system of interpretation is not feasible for data of both types. Despite this, within the context of a single domain, one of the best ways to interpret data is to use the data range as its own definition. This self referential interpretation can be generalized by distance between data in the dataset. The metric for which the data is converted to distance and projected into some embedding space remains somewhat of an expert task, but for the majority of data streams, this is already part of the acquisition process and is not a concern. Distance between objects is actually not very useful unless you are comparing it to some defined scale or its relation to other data. This invokes the concept of similarity and dissimilarity, so the extension of distance and focus of this thesis is distance utilized for establishing nearest neighbors to a queried data point. The concept of nearest neighbors seems to be highly applicable to data of a wide variety of sources. This is due
to there not being some preset scale or known data range, instead the distance metrics are directly realized as similarity and dissimilarity between data.

The basic definition and goal of nearest neighbor search is to return a data point in a dataset that minimizes some distance metric to a queried data point. The distance metric can be any form of distance preserving admissible function such as hamming distance or euclidean distance. A naive, and for high dimensional data, optimal method [6], is to simply calculate the distance metric between a query point and all data in the dataset, and return the least distance data point.

The concept of searching nearest neighbors is fundamental to almost all data analysis in which the goal is to make conclusions about the dataset’s structure and the data’s relation to one another. Applications of nearest neighbors search are also not exclusive to data analytics. Some other abstract uses include web search and fuzzy database querying. Due to its far-reaching utility, the speed of search is very important, and for very large datasets the above, brute force method is infeasible. Due to prohibitively large datasets and their often tertiary data storage, various methods of accelerating nearest neighbors searching have been devised. The most popular method for accelerating nearest neighbor search is in the use of KD-trees for space partitioning. This method however is limited by the number of dimensions that it performs optimally in and eventually degenerates to brute force linear search for very high dimensional data [6]. The need for efficient, high dimensional nearest neighbors search gave rise to the creation of inexact, approximate search methods. Accelerating nearest neighbor searching by searching for approximate nearest neighbors removes the constraint of exactness, thereby allowing heuristics to prune the search space. The new definition for approximate nearest neighbors becomes; return the nearest neighbor within some approximation factor bound of the query point. The relaxation may seem like a major drawback from exact analysis, however in the case of many data-streams we can assume some amount of error and redundancy such that an approximate nearest neighbor is as good or nearly as good as the exact one. Furthermore, mathematical support for relaxing exactness in nearest neighbors can be attributed to an interpretation of Curse of Dimensionality. The interpretation of Curse of Dimensionality is that descriptiveness of any distance metric converges to zero as the number of dimensions goes to infinity. Therefore, approximate nearest neighbor search may also have an advantage over exact nearest neighbor search for high dimensions due to the loss of descriptiveness. This is covered more thoroughly in a paper by Piotr Indyk [7].
1.0.2 Current Approaches

As stated above, for exact nearest neighbor search, the most commonly used method is k-d tree partitioning. A short description of k-d trees can be found in the background section of this thesis. Being the most common it has a multitude of adaptations some of which being approximate search methods which is the basis of this thesis’ nearest neighbor algorithm. In general the approximate search methods avoid some of the issues that plague high dimensional k-d trees by providing some method of controlling the tree depth. An important form of approximate K-D Trees is available in the FLANN [8] [9] package which provides a software toolkit for approximate nearest neighbors, utilizing randomized K-d tree partitioning. Randomized K-d trees allow control over tree depth and by extension query complexity, by only selecting a fixed number of dimensions to partition the tree. The selected dimensions are not completely random in the FLANN implementation and offer a heuristic improvement that factors in the data’s variance along a selected dimension to provide a weighted random selection of dimensions. Randomized k-d trees have found success in a number of applications, in particular the application of SIFT transformed image comparison, which is used to validate this thesis’s solution. An issue with Randomized k-d trees is that they offer no exact approximation bound as the algorithm is data dependent. Randomized k-d trees also still exhibits some of the issues of standard k-d trees and for higher dimensional datasets where variance is relatively uniform they tend to again degrade to brute force search complexity. In this way, randomized k-d trees seem to be more of a band-aid solution that simply stretches the effectiveness of k-d trees to slightly higher dimensions. That said, for many applications randomized k-d trees remain to be the fastest solution.

Another method for approximate nearest neighbor search is similar to k-d partitioning of the data to facilitate faster searches, however instead of being random or variance based, it utilizes k-means to perform clustering on the data. A query step then consists of finding the nearest cluster center and then finding the closest data point in that cluster. Again their are many variants, one of the more successful being a hierarchical k-means nearest neighbor search. All of these methods however assume the feasibility of offline k-means clustering of the data.

The next method for approximate k-means is the primary focus of this thesis and it is a lattice based extension of locality sensitive hash based approximate nearest neighbors [7]. LSH-KNN provides an approximate method for nearest neighbors searching that utilizes a hash function with the distance preserving
property in order to partition a dataset and minimize query time. In the original E2LSH framework a real number vector was converted via a rounded hamming projection into a bit vector. The bit vector was created as a concatenation of the rounded integer representations for every value in the vector. The most fundamental distance preserving hash results in a very similar system and performance of k-d and approximate k-d trees, where the hash chooses a random dimension and partitions the data, either in or out of the hash bucket. By doing this over multiple random dimensions, an \( c \) approximation factor and query bound can be guaranteed.

An extension of random dimensional hashes, is to use a 24 dimensional lattice previously utilized heavily in error correcting codes and space quantizing, known as the leech lattice. The leech lattice is an exceptional mathematical object with a myriad of properties relating to its symmetry, density and uniformity in 24 dimensions. Its application for \( c \)-approximate K-nearest neighbors is in its ability to select simultaneously 24 dimensions per vector and apply a locality sensitive hash label \([10]\) with good approximation bounds for near neighbor hash collisions.

1.0.3 Our Solution

In this thesis a solution is presented that is based primarily on Andoni and Indyk locality sensitive hash based nearest neighbor search algorithm with a leech lattice decoder as the principle hash family. Their solution offers favorable theoretical bounds in terms of search complexity and storage overhead with equally well defined bounds on approximation accuracy.

The original LSH method consists of a hashing function that preserves distance by way of a hash collision probability that is greater for vectors that are ’close’ and less for vectors that are ’far away’ from one another. The difference in near and far probability constraints forces an approximation bound on the probability of finding near neighbors that is determined by the number of hash functions applied to the data. The hash collisions of data from the database with the query data generates a list of similar data. The list is then sorted by the number of hash buckets shared by each of the intersecting data. This gives a similarity metric that can then be used to sort the data items in order of similarity. An additional constraint is applied to the list’s length to assure a deterministic run time of \( \Theta(2^k) \). The Leech Lattice method acts similarly for data with greater than 24 dimensions, where multiple selections of values of length 24 are taken from the total number of values in the vector. The difference is that within the 24 dimensional samplings, hash
collision probability ratios for near and far data is much greater requiring fewer hashes to be applied and still guaranteeing a similar approximation bound for $c$. This is due to the leech lattices optimal sphere packing density among lattices in 24 dimensions \[1\].

Our contribution to the current work in $c$-approximate K-nearest neighbors is an extension of a somewhat under-utilized of the leech lattice and hash based nearest neighbor methods in general. The generative nature of the leech lattice and its decodings by extension provides a universal hash functions for proximity. This directly implies a parallel implementation. Furthermore the bounded distance hexacode based decoding algorithm of \[12\] is a data intensive algorithm well suited for vector processing.

A further addition to the Leech based LSH algorithm is to use sphere center decoding distance as a metric to rank intersections similar to the method used in \[13\] with the E8 lattice. The bounded distance hexacode decoder uses distances as it final comparison method and it is already being computed, so it adds no additional complexity to add it, while offering a sound heuristic for ranking hashing intersection lists.

### 1.1 Hypothesis

This thesis will describe a system for performing Leech Lattice decoding on a GPGPU system for use in an adaptive locality sensitive hash $c$-approximate k-nearest neighbors search and database construction. In addition, the level of parallelization will be shown to be scalable to an arbitrary number of GPGPUs. Data will also be presented showing that despite the better real world running times for Random K-d trees over serial E2LSH, utilizing the Leech Lattice, the presented parallel lsh algorithm offers better scalability and performance.

### 1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 Background: This section contains pertinent background needed for the description of our LSH-KNN system. This will consist of sections on the general K-nearest neighbors problem, Error Correcting Codes and Information Theory, Lattices as Space Quantizers, the CUDA framework for GPGPU computing, and the Scale Invariant Feature Transform (SIFT).
CHAPTER 1. INTRODUCTION

1.2. THESIS OVERVIEW

Chapter 3 Related Work: The section contains a few pieces of related work in this field that is not necessarily addressed elsewhere in this thesis but is in the same field of research. The subsection in this section contain a reference to the industry standard approximate nearest neighbors search package (FLANN), as well as a section on alternative functions for LSH, and a section on other uses for LSH functions outside of KNN.

Chapter 4 Overview: Here we will give a general overview or our approach to developing a parallel LSH-KNN system in CUDA. The first section will contain the general framework of Andoni [1] for an LSH-KNN search, as well as their analysis on its running time complexity. We will then move to a discussion of the Leech Lattice and its various attributes. Following this, will gave an algorithmic outline for our parallel algorithm.

Chapter 5 Detailed Implementation: In this section we will cover all of the details of our CUDA based lattice decoder and LSH implementation. In addition we will discuss tweaks to the decoding algorithm to make it more adaptable to the GPGPU’s memory and processing architecture. Following this we will detail our method of using SIFT based Image search to evaluate our parallel LSH-KNN algorithm.

Chapter 6 Analysis: The analysis section contains the results from testing our system in various context. We will start out by demonstrating our algorithms equivalence to the standard LSH-KNN algorithm, as well as the equivalence of our bounded-distance Leech Lattice decoder. We will also show results for parallel speedup.

Finally, Chapter 7 Conclusions: Here we will attempt to qualitatively describe our findings in the analysis section. We will follow the structure of the previous section for purposes of continuity, and then attempt to summarize our findings. Following this, we will suggest some directions for future work in this field.
Chapter 2

Background

This section contains the background information required to understand the pieces needed to create the nearest neighbors search system. It will cover the general nearest neighbors search and a few of its implementations, some signal theory and basics of error correcting codes, the CUDA framework and GPU architecture, and the Scale invariant feature transforms used for image description. Due to the somewhat lengthy background section it will be separated into subsections as follows.

1. Nearest Neighbors

2. Error Correcting Codes

3. CUDA GPGPU Framework

4. Scale Invariant Feature Transform

2.1 Nearest Neighbors

We begin the section on nearest neighbor search with a high level view of its utility and applications by presenting a simple description of a naive implementation. Nearest neighbors search has a variety of applications.

- Pattern Recognition

- Image/Object Recognition - Robot Vision systems, object and motion tracking
2.1. NEAREST NEIGHBORS

- Genomics - similar and exact gene sequences in long DNA reads
- Biology/Medical - drug interaction and symptoms
- Recommender Systems - similar items based on user submission

The concept of exact nearest neighbor search is quite simple, find an item in a database that is closest by some metric to the query item.

**Definition 2.1.1** (Exact NN). [14] Given a set of points \( P \) in \( \mathbb{R}^d \) and query point \( q \) return a point \( p \in P \) such that

\[
p = \text{Argmin}\{ \text{dist}(p', q) \},
\]

where \( \text{dist} \) is some metric function.

The above definition suggests a simple search algorithm. First decide upon a metric function to provide a measurement of similarity between two items. Then run the similarity between the query and every item in the database, outputting the item corresponding to the minimum metric. This algorithm has complexity \( \Theta(nk) \). The above algorithm is considered to be a naive brute force search method, as it does nothing to minimize the search space being considered for minimizing the distance metric. An important note though is that as an effect of increasing the number of dimensions and the so called **Curse of Dimensionality**, more advanced algorithms often decompose into being no more efficient than brute force searching.

More advanced algorithms of course followed brute force searching that took a variety of forms and optimization techniques. One of the most successful techniques for small to medium dimensionality took into account the databases dimensional partitioning to generate a tree for tree based searching of a query point. One of the most successful algorithms for exact nearest neighbor search is **Kd-Trees**.

**Definition 2.1.2** (k-d tree). [14] A k-d tree is a space partitioning data structure for k-dimensional search algorithms.

K-d trees work by partitioning the k dimensional space by creating a separating hyperplane for each dimension k of the space. Items that have a greater value in the ith dimension than the separating hyperplane’s value are placed on one side of the tree while items that have a smaller values are placed on the other. Each successive dimension creates a new branch in the tree. Leaf nodes in the tree represent an item. By using this data structure, a query point search is able to avoid calculating distance metrics exhaustively with all other points in the database. A problem arises in real world data, as separating hyperplanes are not often
balanced, and the tree search yields little advantage over the naive implementation (2001 Bohm, Beyer et al 1999). Further advancements in k-d tree based searching involved algorithms to balance the tree partitioning. While offering speed up for many practical problems, no guarantees could be made about the search algorithms complexity. Furthermore, tree balancing is still data dependent and has complexity $\Theta(n\log(n))$.

An extension of the standard nearest neighbors problem is the k-nearest neighbors problem. The k-nearest neighbors search problem has a similar complexity to its exact nearest neighbor equivalent.

**Definition 2.1.3** (k-Nearest Neighbor Search). [14] Given a set of points $P$ in $\mathbb{R}^d$ and query point $q$ return the k nearest points $p \in P$ to $q$.

Another nearest neighbors search, and focus of this thesis, achieves reductions in computational complexity by relaxing the exact nearest neighbors requirement and instead replacing it with a constant probability of finding the exact nearest neighbor. Many algorithms utilizes the $c$-Approximate approach to validate heuristics for search tree traversal and data structure constructions. Such methods include Randomized K-D trees [5] [8], Ring Trees [15], and Ball Partitioning [7].

**Definition 2.1.4** (c-Approximate Nearest Neighbor Search). [14] Given a set of points $P$ in $\mathbb{R}^d$ and query point $q$ return a point $p' \in P$ such that it’s distance to $q$ is less than $c \ast \text{dist}(q, p)$ where $p$ is the exact solution.

A further relaxation of c-approximate NN algorithms is the adaptation to a decision problem with a constant probability of correctness. This extension is important to locality sensitive hashing methods as the $\delta$—approximation is related to $\rho$ for the probability of near and far data point collisions.

**Definition 2.1.5** (Randomized c-Approximate Nearest Neighbor). [14] Given a set of points $P$ in $\mathbb{R}^d$ and query point $q$ return a point $p' \in P$ such that it’s distance to $q$ is less than $c \ast \text{dist}(q, p)$ with probability $(1 - \delta)$, where $p$ is the exact solution.

### 2.2 Curse of Dimensionality

Although the **Curse of Dimensionality** (COD) has a variety of interpretations, its application to nearest neighbors search for high dimensional data, primarily restricts the complexity of the exact search algorithm.
In general, variants of exact nearest neighbor search algorithms tend to be no better than the naive linear search as the number of dimensions increases. Their is further evidence of this even for regular point lattices which will be mentioned later.

A more mathematical interpretation of the **Curse of Dimensionality** demonstrates a connection to the algorithmic interpretation, as many search techniques involve euclidean distance as heuristic search metric. The COD can be regarded as the volume ratios of the euclidean distance inscribed n-sphere, and n-hypercube resulting from an orthogonal metric embedding. The intuitive result is that as the number of dimensions increases, the volume of the unit sphere decreases.

\[
Vol(S) = \frac{\pi^{d/2}}{\Gamma(1 + d/2)}
\]

where \( \Gamma(x) = (x - 1)! \). Using the volume of a sphere in d dimensions, while the unit cube’s volume remains constant, we can directly show that as d increases, the descriptiveness of euclidean distance decreases, or

\[
\lim_{d \to \infty} \frac{d_{\text{max}} - d_{\text{min}}}{d_{\text{min}}} = 0
\]

Below are two examples using the densest packing of lattices known in 8 and 24 dimensions, the Gosset Lattice(E8) and Leech Lattice( \( \Lambda_{24} \)) respectively.

\[
V(E_8) = \frac{\pi^4}{4! \times 16} \approx 0.25367
\]

\[
V(\Lambda_{24}) = \frac{\pi^{12}}{12!} \approx 0.00193
\]

2.3 Error Correcting Codes

To begin the background section on error correcting codes we will start with their original motivation. All data channels are susceptible to some form of unexpected error introduced by the operating environment. In some cases it could be a mechanical or electrical error in the system itself, resulting in an incorrectly flipped bit. One such example would be temperature effecting the switching frequency of a transistor or relay resulting in a error bit. The first person to tackle such errors was Richard Hamming, who’s research lead to the development of the Hamming Codes. Another way in which errors are introduced is through analogue
channels in which ever present electromagnetic background radiation imparts some amount of additional noise on a circuit. Both of which systems require that some redundancy be added to the signals to avoid having to retransmit the entire message. The system used to apply the redundancy is the error correcting code. To be more specific this section will only cover forward error correcting codes, and all reference to error correcting codes herein will be assumed to refer to forward error correcting codes. Forward error correcting codes are one-way codes in which the sender adds systematic redundancy to a message, so that it can be decoded by a receiver using a corresponding decoding. A further extension of the forward error correcting codes in ECC are linear error correcting codes. Linear error correcting codes make use of linear equations over a finite fields to make encoding and decoding systematic algebraic tasks.

2.3.1 Types of Channels

There are two primary channels and subsequently two types of interrelated schemes. The first chronologically to be tackled through error correcting codes was the binary erasure channel. This channel is fairly easy to analyze and are can be implementable in binary logic. The first error correcting codes of hamming are generally considered to occur of this channel.

For the EM spectrum there is the additive gaussian white noise channel (AWGN) model. This channel effects analogue real number valued signals by way of the addition of some random error following a gaussian distribution of a known power spectral density. These codes tend to require more sophisticated decoding methods as the distance calculations and decoding must occur over the real numbers.

2.3.2 The Golay Code

Although the focus of this thesis is on data over the real numbers, many connections between binary and Gaussian channel codes exist. In fact most Gaussian error correcting schemes utilize binary error correcting codes in there constructions (ex: E8/Gosset’s Lattice, the Barnes Wall Lattices).

**Definition 2.3.1** (C is an (n,k,d) Binary Code [16]). Properties:

- i) if $d \leq 4$ the minimum norm of $\mu$ of $\Lambda(C)$ is $\mu = \frac{d}{2}$ if $d > 4 \mu = 2$
- ii) $\det \Lambda(C) = 2^{n-2k}$
CHAPTER 2. BACKGROUND

2.3. ERROR CORRECTING CODES

- iii) $\lambda(C^\perp) = \Lambda(C)^*$

- iv) $\Lambda(C)$ is integral iff $C$ is self orthogonal

- v) $\Lambda(C)$ is type I if $C$ is type I

- vi) $\Lambda(C)$ is type II iff $C$ is type II

The Golay code is a binary code with deep connections to group theory and the Leech lattice. Using the nomenclature from above, the Golay code is a (24,12,8) linear error correcting code. It has a variety of constructions, but the simplest and most intuitive is the lexicographic construction.

Definition 2.3.2 (Lexicographical Constructions). Begin with an empty set $S$. Lexicographically add 24 bit words to the $S$ that differ in at least 8 bits from every other word in the $S$. The Linear span $S$ is the generator matrix for the Binary Golay code.

This construction is useful in visualizing the golay code as a space partitioning object. Each codeword is exactly 8 bits away from every other codeword and it completely covers all possible words in $GF(2)^{24}$, and forms a nearly perfect code.

2.3.3 Lattices

Lattices can be visually defined as a regular tiling of some $n$ dimensional space. They are represented as an infinite subset of points in $\mathbb{R}^n$ that spans the $n$ dimensional space.

Definition 2.3.3 (Lattice in $\mathbb{R}^n$). \cite{16} let $v_1, \ldots, v_n$ be $n$ linear independent vectors where $v_i = v_{i,1}, v_{i,2}, \ldots, v_{i,n}$

The lattice $\Lambda$ with basis $\{v_1, \ldots, v_n\}$ is the set of all integer combinations of $v_1, \ldots, v_n$ the integer combinations of the basis vectors are the points of the lattice.

$$\Lambda = \{z_1v_1 + z_2v_2 + \ldots + z_nv_n | z_i \in \mathbb{Z}, 1 \leq i \leq n\}$$

The Figure 2.3.1 shows the hexagonal packing of circles in $\mathbb{R}^2$ for a finite partition of $\mathbb{R}^2$. The finite analogue of a lattice is called a constellation and is generated by apply a finite bounding region to the lattice.
The basis for the hexagonal lattice is \([-1, 1]\)|\([0, -1]\). By applying the integer combinations up to \(n=2\) as in the above definition we get the lattice:

\[
\Lambda = \begin{bmatrix}
-2 & -1 \\
1 & -1
\end{bmatrix}
\]

A variety of important attributes can be obtained for lattice packings. Of importance to this thesis is the sphere packing density of a lattice, as it will act as a space quantizer for high dimensional space partitioning. With maximum density being the main goal of the space partitioning. The sphere packing density is given by:

\[
\text{volume of one sphere} \quad \text{volume of fundamental region} = \text{volume of one sphere} \quad (\det \Lambda)^{1/2}
\]

By setting the radius equal to \(r = \frac{1}{\sqrt{2}}\) in the example above, we get sphere volume \(\frac{\pi}{2}\) and \(\det(\Lambda)^{1/2} = \sqrt{3}\), for a density of \(\frac{\pi}{2\sqrt{3}} \approx 0.9069\).

From the definition of a lattice with a basis \(v\) we can also define a **generator matrix** for the lattice given by:

\[
M = \begin{bmatrix}
v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\
v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
v_{n,1} & v_{n,2} & \cdots & v_{n,n}
\end{bmatrix}
\]

The generator matrix can be used to generate the lattice points for integer combinations, and its connection
to error correcting codes will be discussed in the following section as a method to map integer vectors to \( \mathbb{R}^n \) centers of an n-dimensional lattice.

### 2.3.4 Codes to Lattices

In Conway and Sloane’s: Sphere Packings Lattices and Groups, the authors suggest a construction of lattices based on binary codes by converting the sphere centers of the lattice to their equivalent coordinate array representation.

**Definition 2.3.4 (Coordinate Array).** Let \( x \) be a point in \( \mathbb{R}^n \) where \( x = (x_1, ..., x_n) \). The coordinate array representation consists of the binary expansions of the \( n \)-integers along the columns where each row is a successive \( 2^n \) coefficient. For negative numbers the complementary form is used.

**Example:** \( x = (4, 3, 2, 1, 0, -1, -2, -3) \)

\[
x = \begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
... & ... & ... & ... & ... 
\end{bmatrix}
\]

They also note that because lattices are infinite, the coordinate array representation may also have an infinite number of rows, but the latter rows will be identical after the first few \( \mathbb{I} \)

**Definition 2.3.5 (Construction A).** Let \( C \) be an \( (n, M, d) \) binary code. \( x = (x_n, ..., x_n) \) is a lattice center if and only if \( x \) is congruent (modulo 2) to a codeword of \( C \).

This definition is useful for some codes such as hamming codes and their associated lattices, however we must extend this in order to use it for the Golay Code construction of the Leech Half Lattice. Construction B follows similarly with the decomposition of points into coordinate arrays and the required congruence, but it also adds another restriction.

**Definition 2.3.6 (Construction B).** Let \( C \) be an \( (n, M, d) \) binary code. \( x = (x_n, ..., x_n) \) is a lattice center if and only if \( x \) is congruent (modulo 2) to a codeword of \( C \) and \( \sum_{i=1}^{n} x_i \) is divisible by 4.
In Conway and Sloane’s: Sphere Packings Lattices and Groups, Chapter 10 [11] the Barnes-Wall Lattices are constructed from the Reed-Muller codes (m,n), where n is the number of dimensions \( n \leq 64 \) and is of the form \( 2^m \) by using construction C for generating lattices from binary codes. Naturally there are two other constructions, a and b for which the symmetries of binary codes can be used as generators for \( \mathbb{R}^n \) space lattices.

By using lattice constructions based on binary codes, advancements in the decoding of the binary code, inherently offer an advancement in the decoding of the lattice center points. Of particular importance to this work is the decoding of the binary (24,12,8) Golay Code, and its use in the construction of the half leech lattice: \( H\Lambda_{24} \). This half lattice, when combined with another half lattice construction shifted by \( < -\frac{1}{2}, \left(\frac{1}{2}\right)^{23} > \) forms the vectors of the full leech lattice.

The binary golay code has long been used as an error correcting code [17] for binary signals. It has the ability to correct up to 3 bit errors, and detect up to 4, while requiring only a doubling of the information bits. A variety of methods are known for the generation of the codewords of the golay code with the first being Golay’s original one page construction as a cyclic code with generator polynomial

\[
g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}
\]

This generator forms the perfect (23,11,7) code, which is then augmented with a parity to form the nearly perfect (24,12,8) binary golay code. The weight distributions of the (24,12) code are as follows

From the above distribution, it is clear there is a great deal of symmetry in the codewords. Using these symmetries, many other construction can be developed. For instance, the generation of the Golay code from codewords the hexacode can be obtained using the Miracle Octad Generator of Curtis [18]. The MOG arranges the binary symbols of a codeword in \( G_{24} \) into a 6 x 4 array. This decoding scheme will be described in greater detail in the implementation section as it is a principle part of the Bounded Distance Decoder [4].
2.3.5 ECC to Nearest Neighbors

In the evaluation of codes the ability to decode accurately and quickly became the main focal point. It was not enough to simply correct a large number of errors, but given the limited bandwidth and computational abilities of systems, it was also important to be able to decode and encode efficiently. A common tool for evaluating a codes abilities is the $dbN_0/errno$ Figure 2.3.5.

The Figure 2.3.5 show a few common error correction techniques for sending binary signals over a channel subject to Additive Gaussian White Noise. The independent variable is the inverse log average of the noise floor’s power spectral density. The dependent variable is the ratio of correctly sent bits and the number of redundant bits. The goal is to send a signal subject to a variant amount of random noise, while minimizing its number of needed redundancy.

The connections between ECCs and NN can now be described in the context of error correcting a data vector. All ECC have an associated number of errors in the case of binary codes, or correcting radius that

Figure 2.3.2: Performance of Some Coded and Unencoded Data Transmission Schemes
they are able to tolerate. This correction distance determines the needed similarity between vectors in order to classify them as being close to one another. So by taking two vectors that are near (say within the error correcting bound of spheres around points in the leech lattice) both data vectors will be decoded to the same point.

The similarity radius is however bounded by Shannon’s information rate limit and actually has connections in the previously discussed **Curse of Dimensionality**. Due to this no longer existent limitation, certain trade-offs were consider for codes that made even capacity reaching codes less attractive than there slightly less effective counterparts. The Leech Lattice as an error correcting code, however does not approach this limit, so why not choose a code that does. Capacity approaching codes do exist. One such code is the low density parity check code [maybe add ldpc decoder diagram]. Although these codes dominate much of current signal technology, their use is only the result of excessive computational power. In relation to NN problems, this code as a probability code would likely result in performance similar to concatenated vector selection such as in Indyk’s original work [10]. In addition for linear codes specifically, the decoding times increase exponentially with respect to coding gain [19]. For maximum likelihood decoding, this method quickly decomposes into nearest centroid, where all points in the dataset are codes in the error correcting code.

Due to the exponentially increasing complexity bound, it is important to focus on codes with acceptable decoding complexities, otherwise hashing will result in functions just as difficult as brute force Voronoi partitioning, as n increases becomes no better than exhaustive search.

### 2.4 CUDA GPGPU Framework

GPU based computing has grown tremendously in the last few years due in part to the release of GPGPU toolkits being released by manufacturers and open source organizations. Prior to the toolkit’s release, GPGPU computing was growing in popularity as video graphics consumer’s demands constantly pushed the computational power of GPUs, however it still remained a somewhat untapped resource as the learning curve of what amounted to driver development, precluded man non-OS oriented programmers.

With NVIDIA’s release of CUDA as a framework for GPGPU computing as a simple extension to the C language, many new and prospective developers were able to start leveraging their newfound power from
the GPU. Because in its inception, the GPU’s purpose is for video and 3d graphics processing, special care must be taken by both the programming and framework to avoid various pitfalls resulting from the GPU’s non-standard architecture.

The basic framework of CUDA exists as an extension to the C language, that added syntax to expose various features of the GPU’s parallel processing capabilities. The extensions came in the form of threads, shared memory, and access routines, along with synchronization capabilities among blocks of threads. These extensions allow the programmer tremendous flexibility while, to some extent, enforcing appropriate use of the hardware.

The figure above shows the basic transistor allocation of the GPU processing unit in comparison to a multicore CPU. The first and most important attribute is the GPU’s allocation of transistors to ALU processing. This of course comes at a trade-off as the lack of a large data cache results in forcing the programmer to enforce proper memory management to keep the ALU’s busy. This is further exacerbated by the far higher number of ALU’s than the number of ALU’s available to a more standard processor. The alteration to the architecture and subsequently the program flow, gives rise to the term compute intensive computing. Compute intensive computing focuses on data throughput and processing and limits the range of computing problems GPU computing is adaptable to. Certain aspects of the CUDA framework (especially caching in CUDA 2.1+) along with some clever transformation of problems (Subset sum GPU example) extend the range of problems, but in the end, data access patterns will determine the overall speedup and success of a GPGPU implementation.
2.4.1 CUDA Framework

The CUDA software framework for NVIDIA’s GPU architecture is organized by a hierarchy of blocks and threads intended to partition a general programming problem into a problem that will run on a compute intensive architecture. Blocks can be organized into 1, 2, and 3 dimensional grids to allow further control of execution. All threads in a block are able to communicate with one another through a fast (approximate 1-5 cycle latency) shared memory. Threads that are not in the same block must use global memory for communication at a far greater latency cost (100-200 cycles). The reason for restricting communication to only among only the threads within a block is twofold. First, the block scheduling, done by the GPU, has no predetermined sequence. This is mainly an architectural manifestation as the GPU’s Sequential Multiprocessing cores are SIMD, and execute a subset of threads from a block in parallel. Although block-thread data partitioning seems like a disadvantage compared to direct GPU execution, it provides a transparent mechanism for scaling to GPU’s with an arbitrary number of Sequential Multiprocessing cores. The figure below shows a scaling of a system with 2 sequential multiprocessors, to one with 4 sequential multiprocessors. Decoupling threads into blocks of concurrent data access provides a transparent mechanism for scaling.

The connections between hardware through software primitives ends here however, as the physical execution of threads on the hardware is controlled entirely by the GPU’s task scheduler. Actual instruction execution of threads on physical processing units is organized into warps. A block’s threads are organized into these warps consisting of 32 threads each regardless of the thread block size. Intuitively it helps to imagine the warps being processed in SIMD parallel and certain memory access advantages apply, such as fast access to registers. Warps are actually further subdivided into half-warps of 16 threads. The division allows an apparently parallel, full-warp to hide memory access latencies during the swapping between the half-warps of a full-warp. Warps and half warps are not specified by the programmer, but they can be controlled to some extent through the sequential partitioning of threads, such as into sub-blocks of 16 sequential threads in a block. It is important to be aware of the scheduling actions of warps in order to optimize memory access, however it is much more important to be aware of their actions to avoid pitfalls, such as non-sequential access and memory bank conflicts. One such optimization is that threads within a warp can access the same global memory through a broadcasts operation, as opposed to 32 separate global memory requests. Another issue is coalesced and sequential access of shared memory. Because the half-
warps hide memory latencies to shared memory by swapping back-and-forth, it helps to avoid bank conflicts and non-sequential access by separating conflicting threads into separate half warps.

Another important aspect of the CUDA architecture resides in its function execution. The scalar processors are pipelined in an unintuitive way, that is optimized for graphics processing, but not general purpose computing. The pipelining of operations executions cause pitfalls for GPGPU programmers because otherwise single cycle operations often become multi-cycle operations, that are not hidden by pipeline parallelism, such as the case with CPUs. In short, most floating point operations take a full cycle regardless of their pipelining, meanwhile integer calculations, and shortened floating point operations on 24-bit datatypes enjoy full pipeline advantages. For this reason, if 24-bit instead of 32-bit error is tolerable, datatypes should be specified as CUDA 24 bit types. Another set of execution optimizations can be leveraged from the various special functions available on CUDA processing cores. As stated before, these operations mainly cater to

![Program Block Organization](image)

Figure 2.4.2: Example of Scalable Block Scheduling
graphics processing, however their use as processing directives can take advantage of various sub-processing units that are available on the cores. These functions consist of special integer operations for counting bits and math functions, as well as modified basic math operations for single-precision floating point datatypes such as 24-bit floats.

The optimizations to the execution operations as stated above are fairly easy to implement, as they are drop-in replacements for existing functions. However they will only offer a fixed amount of speedup depending on the amount of such operations and the particular operations being replaced. It is far more important to optimize memory. Unlike execution optimization, memory optimizations often require a complete restructuring of code, and sometimes even a restructuring of the problem solution. Below is a brief summary of the available memory locations on CUDA and their access characteristics. In general, not cached off-chip DRAM access requires 600 cycles of latency. Meanwhile on-chip and cached memories, with proper warp organization can be as fast as register access of 1-2 cycles. In general there are 4 levels of memory hierarchy. The first level is global memory. Global memory resides in off-chip DRAM, and is the only memory location that is accessible to the CPU. All data and program instructions start at this level. Global memory is rather large, but its off-chip DRAM characteristics come with high latency costs of around 600 cycles per read/write. The next level of the hierarchy is texture and static memory. This memory also resides off-chip, and is only writable by the CPU. The difference between constant and texture memories and global memory, is that these memories are cached on chip. The trade-off in size and speed is at the cost of being unable to edit the contents of this memory from within a running CUDA core. The next level of the memory hierarchy is shared memory. Shared memory is on chip and as its name implies, it has concurrent access privileges by the threads within a shared block. Its concurrent access and speed are further augmented by the warp and half-warsps abilities to overlap execution with memory access latencies. With a good memory
access sequence, shared memory can operate on par with single cycle access, register memory. The fastest access for any GPU core memory is register access. Although it provides the fastest access times for any of the memories, it can also severely limit the number of threads possible per block. A low thread per block size often limits the GPU’s scheduler’s abilities in keeping the occupancy high, resulting in inefficient SIMD execution. This non-sequential SIMD execution can cause bottlenecks that exceed memory latencies. Unfortunately the memory architecture of GPU’s make it difficult to organize and optimize programs, and for this reason it is often not very useful to hypothesize why various access methods are better than others, and instead suggests that the programmer employ a more experimental approach.

2.5 SIFT

Scale invariant feature transform is an image transform patented by David Lowe of Carnegie Mellon University, for identifying local features in an image that are invariant to scale, rotational, affine and to some extent lighting transformations. The SIFT method produces a set of equal length feature vectors that correspond to the number of distinctive features in a given image. These features act as a fingerprint for the image and the main algorithm for image comparison once the SIFT vectors of an image are computed is to find the nearest neighbor image based on the distance in regards to the SIFT vectors. For this reason part of the original SIFT algorithm contains a reference to a specific form of KNN searching. The method used for searching is highly domain specific and is tuned to a static SIFT vector length. The method used is best bin first (BFF) KNN search [5]. They show a speedup of 2 orders of magnitude over standard k-d trees, with a loss of only 5% accuracy. This improvement however requires very domain specific tuning such as distance and depth search pruning, which for a arbitrary datasets is often not available. In this regard, our searching algorithm tries to improve upon the scalability and heterogeneity issues of nearest neighbor search, while simply using SIFT as an example application. The primary focus being its ability to scale to arbitrarily large datasets with arbitrary length SIFT vectors. Below is an examples of a set of images on which the SIFT transform was performed. The images were then compared with a simulated composite image, where the contents were arbitrarily rotated and overlapped. The gray lines connect equivalent SIFT vectors.
Figure 2.5.1: Top image is a scene and the bottom contains the SIFT image matches
Chapter 3

Related Work

3.1 Alternate Approximate KNN Frameworks

3.1.1 Standard Algorithms for Approximate Nearest Neighbors

The current ‘industry standard optimal’ algorithms for approximate K nearest Neighbors search can be found in the FLANN package [8], and offers implementations for Hierarchical K-Means tree algorithm and Randomized KD-Tree search. In addition to the two methods described below, FLANN also offers a blended approach using approximate KD-Trees and Hierarchical Kmeans Trees. The blended algorithm automatically selects the optimal of the two previous algorithms based on various computed parameters from the dataset such as branching and iterations of Kmeans [8].

Randomized KD-Tree

Randomize KD-Trees arise naturally from the standard KD-Tree format. In the case of randomized KD-Trees a subset of splitting vectors is chosen as opposed to generating the complete tree partitioning. In addition random kd-trees split the regions per dimension based on the random selection of the splitting plane. A heuristic extension of this algorithm is to allow the data to steer the vector selection by choosing randomly from a subset of the top $l$ highest variance vectors. Randomized KD-trees offer the ability to restrict the search depth of the KD-tree and thus bound the search algorithm complexity in exchange for returning possibly sub-optimal solution to the KNN problem.
Hierarchical Kmeans Tree

The next algorithm implemented in FLANN for approximate KNN is the Hierarchical Kmeans Tree. This algorithm works similarly to the above randomized KD-tree search in that it partitions the dataset into a user specified depth tree. However the difference between the two is that instead of using the variance as the variable of partitioning, HKT uses a hierarchical partitioning of clusters based on kmeans. Intuitively the HKT algorithm will partition the dataset into clusters and sub-clusters of data based on the means of the data. This clustering offers various advantages over the previous method as the means tend to offer a better decision regions for the data, but at the cost of higher preprocessing and data overhead.

3.1.2 GPU based KNN search

“The Fast k Nearest Neighbors Search using GPU” \cite{20} suggests a natural parallel extension of the linear search K-nearest neighbors problem. The suggested extension consists of utilize the parallel compute cores of the GPU to compute the distance and sort calculations of the linear search problem in parallel. The sorting and searches are then agglomerated into a fully merged sorted list of distances. This method is similar to ours in that it uses the efficiency and speed of GPU’s, however its use of linear searching means that it is bounded by the linear search complexity.

3.2 Alternate LSH Functions

In this section we will discuss some alternatives for LSH functions to be used in the standard implementation of LSH-KNN \cite{10}.

3.2.1 LSH a comparison of hash function types and query mechanisms

It is apparent that Jegou, Pauleve et. al have taken up the torch of pursuing practical implementations of LSH-KNN. Much of the work prior to theirs focused on theoretical bounds of LSH-KNN implementations and showed very little application based tuning. In particular, ”Locality sensitive hashing: A comparison of hash function types and querying mechanisms” \cite{21} suggests a variety of variants of lattice LSH functions as well as some more data defined LSH functions such as Kmeans cluster based partitioning. These methods
are then empirically compared with variants of LSH functions and those of other non-LSH KNN search methods using the selectivity metric in the context of SIFT image retrieval. Another comparison within the context of LSH is offered as a relation to Voronoi partitioning of 2d spaces.

3.2.2 Query Adaptive LSH

As previously stated Jegou offers a variety of alternatives to the Leech Lattice for LSH. One such alternative suggested in greater detail a particular alternative Lattice. The E8 or Gosset’s Lattice in “Query-adaptive locality sensitive hashing” [13] is an 8-dimensional lattice packing with a decreased decoding complexity constant than the leech lattice. In addition, a further heuristic to matching is implemented that utilizes the query point’s lattice center distance metric to order the length 2l exact nearest neighbor search list. This replaces the random exact NN list truncation for bounded LSH-KNN complexity found in Andoni [10].

3.2.3 Spherical LSH

“Spherical LSH for Approximate Nearest Neighbor Search on Unit Hypersphere” [22] utilizes the premise that a unit polytope embedded in a unit hypersphere in $\mathbb{R}^d$, when randomly rotated, will always project a point on the $(d - 1)$-sphere’s surface. The decoding of a point hash then consists of locating the nearest simplex of the polytope. This can be done in $\Theta(d^2)$ time [22].

By covering the sphere’s 2d surface similar to Voronoi diagrams of 2d-surfaces, and applying an arbitrary labeling, an LSH function can be created for any d-dimensional space. This has an advantage over lattice packing because dense lattices with efficient decoding algorithms are only know for small finite set of unique dimensions. This results in avoiding the random projection step of Lattice based approximate KNN, which causes slightly less optimal values for $\rho$.

3.3 Other Uses for LSH

Here we will consider some other utilizations of LSH outside of the the standard nearest neighbors search problem.
3.3.1 Adaptive Mean-shift Accelerated by LSH

In "A variant of adaptive mean shift-based clustering" [23] the LSH function is considered as an alternative to K-NN search in adaptive Mean-Shift Algorithm. The general maximization step of Mean Shift Clustering consists of a hill climbing iteration. The algorithm converges when the density mean has a gradient of zero, or in other words, the means are stable. The mean-shift algorithm is computationally intensive, and considers points that are often very 'far' from the current density mode when computing each subsequent step of the iteration. For this reason, an obvious improvement is to limit the scope of each iterations computation of the kernel gradient function. However the width, often referred to as bandwidth, of the neighborhood is fixed. For a given distance kernel function \( K(\cdot) \), and a bandwidth estimate \( h \) we get the following update iteration step for the gradient iteration \( \vec{x} \).

\[
\vec{x} = \sum_{i=1}^{n} K'(\frac{x-x_i}{h}) \vec{x} \sum_{i=1}^{n} K'(\frac{x-x_i}{h})
\]

An adjustment to this algorithm is called adaptive-mean shift, and consists of limiting the considered near points by defining its k nearest neighbors as its neighborhood of points, and allowing \( h \) to vary with the data’s natural density.

\[
\vec{x} = \sum_{x_i \in N(x)} K'(x-x_i) \vec{x} \sum_{x_i \in N(x)} K'(x-x_i)
\]

The set \( N(x) \) consists of the predefined set of nearest neighbors to a point, at the current iteration of the algorithm. A simple extension would be to use LSH-KNN at this step.

In general the cost of an adaptive method comes at the need for repeat calls to the nearest neighbors search. However in the case of LSH functions this can be done much faster in the approximate case. The drawbacks of approximation are not very detrimental as the densities are take iteratively given opportunity for all nearest neighbors to be considered in subsequent iterations.
Chapter 4

Overview of the Approach

The basic approach followed in this thesis is that of E2LSH’s Lattice based nearest neighbor algorithm, with the addition of compute intensive parallelism and query adaptive list truncation. We will first discuss the E2LSH algorithm as a random vector selections as in Andoni’s original paper, and build the theoretical framework of locality sensitive hashing for c-approximate K nearest neighbors. This method uses the concatenation of random projections and partitioning as the hashing algorithm. The simplicity of using random projections allows for a rigorous probabilistic analysis of the algorithm. The probabilistic framework was then adapted to lattices of t-dimensions, which was later realized by random projections of 24 dimensional vectors to be decoded by the leech lattice. Following this we will discuss some of the attributes of the leech lattice, and why it was chosen as the hashing function for this algorithms. In addition we will discuss the specific bounded distance hexacode decoding algorithms. Due to the complex nature of the lattice decoding, monte carlo simulations were needed to calculate various values of the randomized partitioning framework, which were presented by in Andoni and will be shown here but not calculated. Once the algorithmic framework is established, we can begin to discuss the CUDA based parallelization of lattice decoding, and varous features of the algorithm in regards to its implementation of GPU hardware. We will also show the slight extension as in for query adaptive searching, using the distance metrics as returned by the decoder for pruning the hash lists. The section will conclude with some theoretical analysis of speedup, as well as a comparison to a standard distributed computing approach in terms of energy efficiency for the practical application of SIFT based image search.
4.1 LSH based c-Approximate r-Near Neighbor

4.1.1 A Locality Sensitive Hash Function

LSH solves the c-approximate kNN problem for a radius R as discussed in the background section [25].

**Definition 4.1.1** (Locality Sensitive Hash Function). Let $\mathbb{H} = \{ h : S \rightarrow U \}$ be $(r_1, r_2, p_1, p_2)$-sensitive if for any $u, v \in S$

1. if $d(u, v) \leq r_1$ then $Pr_{H}[h(u) = h(v)] \geq p_1$
2. if $d(u, v) > r_2$ then $Pr_{H}[h(u) = h(v)] \leq p_2$

For $\mathbb{H}$ to be a locality sensitive hash function the parameters $r_1 < r_2$, and $p_1 > p_2$. In [15] the authors suggest a simple random projection followed by a space partitioning along the projected line, of partitions of width $w$. An example of this is given below where points in $\mathbb{R}^2$ are randomly projected onto a line in $\mathbb{R}^1$, that is partition into sections of length $w$.

In the Figure 4.1.1 of $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ the probability of intersections for hashing functions is acceptable, however when the dimensionality is much greater, the probability $p_1$ of intersection for vectors that are ‘near’ one another by randomly selecting a vector decreases $\propto \frac{1}{n}$ and the chosen partition width $w$. Furthermore the probability $p_2$ for items ‘far’ from each other increases. To avoid the convergence of $p_1$ and $p_2$ as $n$
CHAPTER 4. OVERVIEW OF THE APPROACH

LSH BASED C-APPROXIMATE R-NEAR NEIGHBOR

increases, an additional outer hash function is used to augment the deviation between the two probabilities. This hash function is of a simple form and consists of simply concatenating the hash function results of random projection and partitioning. A formal definition of the hash function $G$ follows.

**Definition 4.1.2** (Locality Sensitive Hash Function 2). $G : g(x) \rightarrow U$

$g(x) = \{h_1(x), ..., h_n(x)\}$

The subsequent concatenation of hash labels, because it may be a potentially large string of bits, can undergoes a secondary hash digest of the resulting bit vector. For this hash, a universal hashing family can be used. Furthermore, the size of the set of only the non-empty concatenated hash buckets is used thus reducing the size of the hash universe significantly. According to “Universal Hashing in Constant Time and Linear Space” [26] this process can be done for a known size set in linear space requiring a constant time lookup.

4.1.2 Nearest Neighbor Algorithm

With the locality sensitive hash function defined, we can begin to consider the framework of a c-approx r-near neighbor search algorithm and its reduction to a c-approx near neighbor algorithm.

**Algorithm 1** Preprocessing

Require: $X = \{x_1, ..., x_m\}, x_k \in \mathbb{R}^n$

$U(x)$ is a universal hash function

$h_k(x) \in \mathbb{H}$ is $(r, cr, p_1, p_2)$-sensitive

choose $l$ s.t.

$G \leftarrow i \in \mathbb{Z}^l$ from $[0, n)$

$g(x) = \{h_1(x), ...h_j(x)\}$

$D \leftarrow []$

for all $x_k \in X$ do

$D \leftarrow U(g(x_k))$

end for

return $G, D$

This algorithm solves the c-approx Nearest Neighbors problem with log overhead [25] using $O(dn + n^{1+\rho})$-space and $O(n^\rho + \log(n))$-evaluations for the query.
Algorithm 2 Query: c-approx k-nearest neighbor

Require: $\hat{x} \in \mathbb{R}^n$, $G$, $H$, $U()$, $D$

$L = []$

for all $g_j(x) \in G$ do

$L \leftarrow U(g_j(\hat{x}))$

end for

$K = []$

for all $l \in L$ do

$d \leftarrow \text{dist}(\hat{x}, D[l])$

$K \leftarrow \{d, D[l]\}$

end for

sort($K$)

return $K[0 : k]$

4.1.3 Complexity Analysis

The proof of the analysis for w-partitions for l random projections is deferred to [15] on s-stable distributions for LSH based Nearest Neighbors. The results will be given here for reference with some information regarding their origin.

Definition 4.1.3 (Theorem: $r_1, r_2$-Point Location Equal Balls). Suppose $H$ is an $(r_1, r_2, p_1, p_2)$-sensitive hash family then there exists an algorithm $(r_1, r_2)$-PLEB which uses $O(dn + n^{1+\rho})$-space and $O(n^\rho)$ evaluations for each hash where

$$\rho = \frac{\ln(p_1)}{\ln(p_1/p_2)}$$

Because the above definition for a PLEB would require a linear search to find the nearest ball location, we turn to a similar but far more efficient method of encoding and decoding LSH labels based on known generative dense lattice packings, from signal theory. Below is a definition of this variant based on lattices with redundant shifts and rotations of the point lattice to cover the $d$-dimensional space.

4.1.4 Lattice Based LSH Variant

Let’s start by recalling the definition from the background section of a generalized lattice, $\Lambda_t \in (R)^t$. We will assume we can shift and rotate the lattice in its native space such that the combinations of shifts and rotations will cover the $t$-dimensional space. Using the set of lattice and lattice transforms we can define a ball partitioning of $t$, and subsequently a new LSH function.
Definition 4.1.4 (Lattice LSH). Given a lattice $\Lambda_d$, generate all shifts $v_i \in \mathbb{R}^d$ of $\Lambda_d$, $\Lambda_d + v_i$ such that the entire space $\mathbb{R}^d$ is covered by a ball of radius $r$ around the lattice points. Label the shifts by the vector $v_i$.

A hash function based on the uniform lattice $\lambda_n$ maps all $x_n \rightarrow U$ by storing all shifts $v_i$ with the integer referenced uniform lattice center in $\Lambda_n$. ie. $x_1 = \{\{v_0, \Lambda_d(T)\}, \ldots, \{v_i, \Lambda_d(T)\}\}$

The hash family function now consists of finding the nearest ball center to a given point, for a given rotation and projection. For a uniform grid of balls in $d$-dimensional space the search for the nearest ball becomes a nearest neighbor search, among the candidate balls in the constellation. Furthermore, a uniform grid may require many shifts and rotations to cover $d$-dimensional space, as its density is not likely optimal. In fact, the densities of all lattices tend to degrade rapidly with increasing dimension, and this can be seen as a direct result of $\text{COD}$.

A practical variant of the ideal lattice LSH is considered in Andoni [10]. The ideal hash family will instead use well known dense lattices such as E8, and the Leech Lattice as their representatives. One drawback to using fixed lattices is that it ties the algorithm to a specific dimension, and the previous randomized LSH algorithm must be used for dimensions greater than those fixed dimensions. However the drawback is very much mitigated by the presence of some well studied families of lattices having nearest neighbor search methods that are orders of magnitude less than direct nearest neighbor search. These will from here on be referred to herein as decoding complexities. Although the densities of lattices are far better than a uniform grid, the lattice still does not cover all of the $d$-dimensional space and will require a similar shift and rotation technique as the ideal lattices above.

Due to the low densities of high dimensional, lattices, we will require multiple hash tables of the points corresponding to the random shifts and rotations of the lattice constellation. The shifts will server to cover the space, but have complexity that is related to the $\rho$ intersection probability metric, $O(dn + n^{1+\rho})$.

However, a further reduction in the space complexity can be achieved by reducing the number hash families, corresponding to random shifts and rotations, needed to cover the subspace $d$. This result is found in Panigrahy [27], and reduces the required space to have linear complexity. Panigrahy’s suggestion is to generate the shifts and rotations to cover the subspace of $n$, dynamically in the query stage. The intuition is that by choosing a projection matrix that projects around the mean of the point in higher dimensional space, at radius greater than a lattice covering’s covering radius, various nearby candidate lattice centers can
also be investigated by the query. The shifts and rotations are performed by generating projections into the d-dimensional lattice space. The random projection matrices are chosen from a scaled normally distributed probability distribution. According to Panigrahy, this distribution when used as a projection matrix, will have symmetry about the mean of the original point [?].

The result requires us to search multiple points during the query stage, and has additional complexity \( n^p \) overhead. However, in regard to the previously stated query complexity\[4.1.3\] this addition attributes only a constant factor increase, and thus maintains the original query complexity big-Oh bound. The space reduction also results in a practical speedup to the algorithm as it is now feasible to maintain the entire hash database in orders of magnitude faster, RAM memory.

### 4.2 The Leech Lattice

The details of the leech lattice directly imply its success as an error correcting code. In addition the lattice decoder can be adapted to be a valuable family of locality sensitive hash functions for use in our nearest neighbor algorithm as will be presented here.

- **Dimension:** 24
- **Minimum Square Distance:** \( d_{\text{min}}^2(\Lambda) = \sqrt{2} \)
- **Number of Nearest Neighbors:** \( K_{\text{min}}(\Lambda) = 196560 \)
- **Volume:** \( V(\Lambda) = 1 \)
- **Sphere Packing Density:** \( \frac{\pi^{12}}{12!} \approx 0.00192957 \)
- **Hermite Parameter (nominal coding gain):** \( \gamma_c = \frac{d_{\text{min}}^2(\Lambda)}{V(\Lambda)^{3/2}} = 4(6.02dB) \)

The leech lattice gives the highest density sphere packing of any lattice, and is likely optimal for all sphere packings in 24 dimensions. Cohn and Kumar in “Optimality and Uniqueness of the Leech Lattice Among Lattices” suggest that the sphere packing density could only be exceeded by no more than \( 1.65 \times 10^{30} \) [28]. These attributes lead to its initial use as an error correcting code for use in Additive Gaussian White Noise channels.
4.2.1 Brief History

The Leech lattice was introduced in 1963 by John Leech and was created from the gluing of two Leech Half lattices. The Leech half lattice results from the lattice packing that can be constructed via construction A, as stated in the background section on lattice constructions, by applying the integers $\mathbb{Z}$ over the binary vectors of $G_{24}$.

The two half lattice are ‘glued’ together such that the centers of the two lattices line up with the deep-holes of the other $[11]$. This gluing results in a density doubling of the lattice and is an affine shift $< -1 \frac{1}{2}, (1^2)_{23} >$. Using the Ungerbock’s binary partitioning of euclidean $\mathbb{R}^2$ space $[3]$, binary signals can be encoded for transmission over the Gaussian Error Channel, and encoded as centers of the Leech Lattice, resulting in an error tolerant signal, with nominal gain 4. The next step in the utilization of the Leech Lattice as an error correcting code was to minimize the steps needed for decoding.

4.2.2 $\Lambda_{24}$ as an Error Correcting Code

The general form of the Leech Lattice used as an error correcting code is as follows. A 12 bit binary signal is encoded as a 24 bit binary vector via the Binary Golay encoding. The Binary Golay encoding, along with the parity of the 24 bit vector, is used to determine points in the QAM 64 lattice in accordance with its Ungerbock’s point encodings. The resulting vector, $v \in \mathbb{R}^{24}$ space is then sent over the noisy AWGN channel. After subject to noise the signal is then decoded by finding the nearest lattice points in the Leech Lattice to the received signal. Naturally finding the nearest lattice center becomes the most computationally expensive part in this error correcting system, and was a key subject of ECC during the early to mid-90’s.

4.2.3 Decoding and Bounded Distance Decoding

The first optimizations to decoding a point to its nearest lattice point in $\Lambda_{24}$ over brute force searching of the 196,556 vectors of $\Lambda_{24}$ is often attributed to Conway & Sloane’s $[11]$. They presented a decoder based on the E8 Lattice using the Miracle Octet Generator form of the vectors of the Leech Lattice. Using this construction they were about to decrease the operation count to only 55968 operations. Later constructions of the Leech Lattice used the Wagner Decoding Rule and Trellis Decoding to further reduce the operations count to around 3600 operations. For maximum likelihood decoding, this operations count has remained
fixed. But a new form of bounded distance decoding which allowed for a slight loss in error correcting gain of 0.1dB was created based on the Golay Code. This allowed for decoding complexities of around 1000 operations. The next and final addition was the construction of a bounded distance Golay Decoder based on a hexacode construction to reduce the decoding complexity to only 571 operations at the cost of another 0.1dB of error gain. In this thesis we use the final decoder and trade the 0.2dB losses for the improved decoding complexity, but most importantly, in terms of a CUDA implementation, is the benefit of a reasonable memory overhead.

### 4.2.4 As an LSH Function

The leech lattice as a successful error correcting code also merits worth as a good space quantizer. And it is this attribute that is a focus of our LSH algorithm. The Leech Lattice will act as our ball partitioning, similar to the Voronoi region creation of two dimensional spaces, for fast KNN searching in 24 dimensions. The higher the density of the lattice, as defined by its volume to principle region ratio, suggests a reasonable metric of quality for a lattice’s utilization as an LSH function. The leech lattice is of course the densest regular lattice in 24 dimensions. The regularity of the lattice is of importance as it allows decoding by a much more efficient error correcting code decoder based on the Hexacode construction of the Golay Code as stated above in the Decoding subsection.

### 4.3 LSH c-Approximate k-Nearest Neighbors

The next step in the construction in our nearest neighbor search algorithm is to combine the Leech Lattice and its bounded distance decoder, with our algorithm for nearest neighbor search utilizing an n dimensional lattice. The first problem is that the dimensionality of the lattice is fixed in 24 dimensions. The result of this is that we must combine the random projection concept from the original LSH algorithm, with the ball partitioning of the latter algorithm. The combination of the two algorithms results in a somewhat difficult to analyze complexity, and in Poir and Indyk’s 2006 paper they opt to simulate the results and calculate the required parameters using monte-carlo methods. We further include the addition of Panigrahy in the query portion making the storage requirement linear in n. This addition consists of storing only a single hash table for the database hash of points as opposed to multiple hash tables under random translations and
rotations as is the case in Andoni’s LSH construction [10].

Algorithm 3 Preprocessing: \( \Lambda_{24} \) LSH \( c \)-Approx \( k \)-Nearest Neighbors

Require: \( X = \{x_1, \ldots, x_m\}, \ x_k \in \mathbb{R}^n \)

\( U(x, n) \) is a universal hash function for a size \( n \) set

\( \lambda_v(x) \in \Lambda_{24} \) is a Leech Lattice Decoder

choose \( k \) s.t. \( k > n/24 \)

\( G \leftarrow c \in \mathbb{Z}^{24} \) from \([0, n)\) for \( k \)

\( g(x) = \{\lambda_1(x), \ldots, \lambda_k(x)\} \)

\( D \leftarrow \emptyset \)

for all \( x_k \in X \) do

\( D \leftarrow U(g(x_k)) \)

end for

tune(\( U \)) choose parameters

return \( \ G, D, U \)

Algorithm 4 Query: \( \Lambda_{24} \) LSH \( c \)-Approx \( k \)-Nearest Neighbors

Require: \( \hat{x} \in \mathbb{R}^n, \ G, \ H, \ (\ ), \ D \)

\( L = \emptyset \)

for all \( g_j(x) \in G \) do

\( L \leftarrow U(\lambda_j(\hat{x})) \)

for all \( r \in R \) random permutations do

\( \hat{x} = \hat{x} + \text{RandNormalVariate()}^{24} \)

\( L \leftarrow U(\lambda_j(\hat{x})) \)

end for

end for

\( K = \emptyset \)

for all \( l \in L \) do

\( d \leftarrow \text{dist}(\hat{x}, D[l]) \)

\( K \leftarrow \{d, D[l]\} \)

end for

sort(\( K \))

return \( K[0 : k] \)

4.3.1 Sequential Algorithm Improvements

An improvement for the exact query variant of the above algorithm is suggested in [13] for the E8 lattice.

Because the original LSH algorithm using the Leech Lattice as a locality sensitive hash family did not use the distance parameter that results from the Bounded Distance Decoding Algorithm, it was assumed there was no distance available. However this is not the case and the direct comparison of total distance is required.
within the algorithm, and is thus available for use as an adaptive query parameter. The improvement therefore takes the same form of that in [13] and sorts the exact search list and truncates the $>2L$ lattice hashings. $L$ is chosen such that it bounds the query complexity by an increase in complexity that is less than $\log(n)$ in accordance with [7] original analysis of $L$.

4.4 Parallel Lattice Decoding

The parallelization of the Lattice decoder can be done in a variety of ways due in part by the fact that the lattice decoding labels are generative and only dependent upon the preprocessed random projection matrix $M$ that maps the vector from $\mathbb{R}^d$. The random selection vectors, by design to $\mathbb{R}^{24}$. Because of this lack of restriction, the algorithm can essentially be embarrassingly parallelized by simply giving each processing node a set of data vectors to decode. The vectors are arranged in sets of 12 values from $\mathbb{R}^2$ and projected onto the QAM16 Constellation partitioned following Ungerbock’s maximum distance binary encoding. Another option for parallel execution is to distribute full vectors to each processing node, and distribute sets of random vectors to be processed. This method allows for dynamic random selections of attribute vectors by the query algorithm, however it is not very space efficient for GPGPU based processing. This is due to the static data vectors being underutilized per query for a given random attribute vector. The latter method would be better accustomed to distributed data systems such as MapReduce, Hadoop, or MPI/Beowulf Clusters Computing.

4.4.1 Improvements

Although the naive parallelization offers expected speedup over the sequential algorithm, it is far from linear speedup. The GPGPU’s restricted memory access architecture is responsible for this non-linear speedup result. Because of this, it is important to restructure the current algorithms memory footprint and access patterns in order to better utilize the available fast shared memory. In order to decrease the memory footprint we can exploit the SIMD parallel architecture of the GPGPU by altering subdecoder’s organization from the sequential bounded distance decoder algorithm. One improvement employed in our final decoder involves parallelizing the bounded hexacode based subdecoders. This result will be presented in the Detailed Implementation Section 5 as its implementation requires that we first show the sequential decoding algorithm in
rigorous detail.
Chapter 5

Implementation Details

In this chapter we will discuss the implementation of our preprocessing, query and SIFT comparison system. In addition we will discuss the way in which metrics for speedup and quality of search were measured and used for comparison. The first section will detail the parallel lattice decoder by first describing the sequential algorithm, and then getting into the specific details of the parallel algorithm and its implementation on the GPGPU computing device. The conclusion will contain the SIFT vector generation system, and the basic application framework for comparing images based on SIFT vectors and our implementation of KNN on a GPU.

5.1 Parallel Lattice Decoder Implementation

5.1.1 Overview

The lattice decoding for the nearest center label for points in the leech lattice is used as the locality sensitive hash function in this c-approximate k-nearest neighbors search. The main article used as reference for the Leech Lattice Decoder is Be’ery and Vardy’s, “Even Faster Bounded Distance Hexacode Decoder” [4]. This publication unfortunately does not cover the full construction of the decoder and instead only offers updates to specific parts of previous decoders [4] [12] [24] [3]. For this reason a variety of papers where written as incremental improvements to the bounded distance decoding of the Leech Lattice, mainly through improvements in partitioning and decoding of the lattice with the $H_6$ sub-decoder. The final partitioning and
decoding algorithms of this line of research result in our parallel decoder algorithm optimized for GPGPU processing.

The general outline from Forney for a minimal trellis based E8 decoder is directly applied to Conway and Sloane’s constructions for the Leech lattice as a construction from three E8 lattice sub-group packings. Forney’s decoder used Viterbi decoding to walk through the 256 state trellis for the E8 decoders, instead of Conway and Sloane’s method of directly searching the minimum distance centers of the E8 lattice to the query point. Although this improvement is important for maximum likelihood decoders, Viterbi decoding is in general performed by tree traversal and is not very adaptable to GPGPU implementations. Therefore the method used in this thesis is for a bounded distance decoder which decodes a point only if it is within a bounded distance of a lattice center. In this decoder the lattice is obtained by the concatenation of two leech lattice half decoders. The leech half decoders are constructed from Construction B of binary codes to lattices. The two half decoders result in the full decoder by translating one of the decoders by $\langle -1 \frac{1}{2}, (\frac{1}{2})^{23} \rangle$. The result of decoding corresponds to the nearest decoding from the two half decoders. The half decoders decoding is further reduced to the decoding of the binary Golay code, by way of partitioning the Euclidean space by the $\mathbb{Z}_2$ lattice, and minimizing the distance between the Euclidean points and their nearest representative in the QAM16$^{[5.1.1]}$ constellation with binary encodings corresponding to the Ungerbock partitioning $^{[3]}$. The Ungerbock labeling of points in the QAM constellation provides the maximum separation distance for the corresponding codewords in the Binary Golay Code. The A and B labellings along with the requirement that a lattice is either of A-type or B-type, provides our second distance requirement between the two ‘glued’ and translated Leech half lattices.

Forney’s maximum likelihood trellis decoder is capable of maximum likelihood decoding of the binary Golay Code requiring only 1351 operations. In general using the Golay Code decomposition, the leech lattice can be decoded in approximately 2 times the number of operations need to decode the Golay Code. For this reason advancements in either decoding method generally result in advancements for the other. Because our method is high probabilistic in nature, the optimal decoding with its higher operations count is exchanged for a suboptimal bounded distance decoder. Therefore our subsequent concatenation of the binary encoding is performed by the bounded distance hexacode based Golay decoder for each leech half lattice. The decoder in terms of error rate is .2dB less effective over the memoryless noise channel subject to AWGN.
This result, although not directly correlated to sphere packing, yields an approximate lessening in packing density. The suboptimal decoder trade-off of optimal decoding for a sub-optimal decoding algorithm that is more amicable to parallelization, is well worth the substitution.

The Hexacode as a Partition of The Golay Code

The hexacode is (6,3,4) code with 64 codewords over $GF(4)^6$. The generator matrix of the code is
and is applied to all permutations of characters in $GF(4)^3$, therefore giving the $4^3$ codewords of the hexacode. In the following implementation we will use the Bounded Distance Hexacode decoding from Amrani’s article [12]. The method reduces the complexity of decoding the hexacode while maintaining the same guaranteed error corrections radius. The method consists of first locating the least reliable characters in the first and second half of the code separately. $x = x_1, x_2, x_3, x_4, x_5, x_6$ and $y_1, y_2$ where $y_1 = x_1, x_2, x_3$ and $y_2 = x_4, x_5, x_6$. All candidate codewords in $H_6$ are generated for all possible values of the least reliable characters. The overall distance for the nearest codeword is computed and the least distance is returned. The $H_6$ decoding only requires 20 operations.

The hexacode based decoder operates on the Golay code by projecting the codeword onto a 6x4 matrix as follows.

\[
G_{(6,3,4)} = \begin{bmatrix}
1 & 0 & 0 & 1 & \bar{\omega} & \omega \\
0 & 1 & 0 & 1 & \omega & \bar{\omega} \\
0 & 0 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\]

For a given vector to be a codeword in the Binary Golay Code, the above matrix decomposition must follow the following constraints. The columns must have equal parity to the parity of the first row. The columns must also project to one of the 4 possible $GF(2)^4$ projections of the hexacode characters in $GF(4)$. And the projections of the 6 columns into $GF(4)$, must be a codeword of the Hexacode. We will define these characters as $\{0, 1, \omega, \bar{\omega}\}$. As stated, each hexacode character has 4 projections into $GF(2)^4$. They have an even projection, and odd projection, as well as the two binary complements of those projections.
Multi-Level Definition of the Leech Lattice

Using the construction of the Binary Golay Code from the Hexacode, and the Leech Lattice as the partitioning of two Leech Half Lattices as generated by Construction B of Binary Codes to lattices based on the Binary Golay Code, we are now able define the Leech Lattice as follows from this construction.

**Definition 5.1.1** (Leech Lattice, Hexacode Construction (Amrani ’94)). The Leech Lattice is the set of all $2 \times 6$ arrays whose entries are points of the $D_2$, such that each array satisfies the following conditions:

a) It is either $A$-type or $B$-type.

b) It consists of either only even columns or only odd columns.

c) The overall $k$-parity is even if the array is $A$-type, and odd otherwise.

d) The overall $h$-parity is even if the array columns are even, and odd otherwise.

e) The projection of the array is a codeword of $H_6$.

*The definition of the Binary Golay Code may be obtained simply by omitting the first constraint to either A or B type.

The new definition for the Leech Lattice provides a fairly straightforward decoding algorithm, consisting of 4 parallel decoders of the $Q_{24}$ or Quarter Leech Lattice. The $Q_{24}$ is the sub-lattice decoding for a fixed combination of $h$-parity and $k$-parity constraints. This partitioning allows for a 4-way parallel decomposition of the Leech Lattice, in which only the minimal distance of the 4 decoders need be compared at the end.

Below is a diagram of the sequential decoder’s block diagram\[5.1.3\] as well as a parallelized decoder\[5.1.7\] for 4 simultaneous threads.

<table>
<thead>
<tr>
<th>Char</th>
<th>even projections</th>
<th>odd projections</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 1111</td>
<td>1000 0111</td>
</tr>
<tr>
<td>1</td>
<td>0011 1100</td>
<td>0100 1011</td>
</tr>
<tr>
<td>$\omega$</td>
<td>1010 0101</td>
<td>0010 1101</td>
</tr>
<tr>
<td>$\bar{\omega}$</td>
<td>0110 1001</td>
<td>0001 1110</td>
</tr>
</tbody>
</table>

Table 5.1: Projections of $H_6 \rightarrow GF(2)$
Figure 5.1.3: Leech Lattice Decoder
5.1.2 Implementation of The Sequential Decoder

The lattice allows for the input of a 24 dimensional vector of real numbers uniformly distributed between \([0.0, 8.0]\). The algorithm first reorganizes the vector from \(\mathbb{R}^{24}\) and projects it onto \(\mathbb{R}^{2}\). The order in which the projection occurs is inconsequential, however for ease of partitioning we will use the following decomposition.

\[
\begin{bmatrix}
    x_1 & x_2 & x_3 & x_4 & \ldots & x_{22} & x_{23} & x_{24}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
    x_1 & x_2 & x_3 & x_4 & \ldots & x_{21} & x_{23} \\
    x_2 & x_4 & x_6 & \ldots & x_{22} & x_{24}
\end{bmatrix}
\]

QAM Metrics

The 12 points in \(\mathbb{R}^{2}\) are projected onto the QAM16 constellation. Each point in the QAM constellation is represented by 3 binary labels, \(i, j, k\). Furthermore there are two types of points, Type-A and Type-B, for a total of \(2^4 = 16\) points. For the remainder of the algorithm description we will omit the B-type half lattice as it is symmetric to the Type-A half lattice. The first set of metrics recorded are the \(d_{i,j}\) metrics. The \(d_{i,j}\) metrics consist of 4 nearest distances of the 4 \(i,j\) labelings (00,01,10,11). These are recorded for each of the 12 points for a total of 48 distances. In addition to the nearest labels, the parity of the \(k\)th bit in the point representation is recorded as the \(k\)-parity vector. Another set of distances is also maintained to resolve \(k\)-parity violations, but recording the 48 distances between the 12 points, and their complement \(k\)-parities. This vector will be called \(d_{i,j,k}\). The total number of metrics so far is 48*2 (32bit) floating point distances and 48 \(k\)-parity bits. The size of the metrics will become important as we move to the CUDA hardware due to its SIMD core memory constraints.

Block Confidences

The next step in the algorithm is to generate the block confidences for each of the 6, 4-bit projections of the characters in \(GF(4)\). These confidences serve as a distance metric between the input vector in \(\mathbb{R}^{2}\) and the hexacode character representation in \(GF(4)\). Only the minimum odd and even representations are returned, for a total of \(6 \times 4 + 6 \times 4\) µ’s

The block confidences will be used to in the subsequent Hexacode decoder.
Algorithm 5 $d_{ijk}$’s A-type Constellation Points

Require: $v \in \mathbb{R}^{24}$, $\text{dist}(\cdot)$ is a distance function, $\text{Pts}[\cdot]$ is a QAM point constellation

for all $i=0$ to $12$ do
    $d_{000} = \text{dist}(v(i), \text{Pts}(0))$
    $d_{110} = \text{dist}(v(i), \text{Pts}(1))$
    $d_{001} = \text{dist}(v(i), \text{Pts}(2))$
    $d_{111} = \text{dist}(v(i), \text{Pts}(3))$

    if $d_{000} < d_{001}$ then
        $d_{000}(i) = d_{000}$
        $d_{00k}(i) = d_{001}$
        $P(i) = 0$
    else
        $d_{000}(i) = d_{001}$
        $d_{00k}(i) = d_{000}$
        $P(i) = 1$
    end if

end for

Algorithm 6 Block Confidences

for all $i=0$ to $6$ do
    $\mu_{\text{even}}(i)_0 = \min\{d_{00}(4i) + d_{00}(4i + 1), d_{11}(4i) + d_{11}(4i + 1)\}$
    $\mu_{\text{even}}(i)_1 = \min\{d_{11}(4i) + d_{00}(4i + 1), d_{00}(4i) + d_{11}(4i + 1)\}$
    $\mu_{\text{even}}(i)_\omega = \min\{d_{10}(4i) + d_{10}(4i + 1), d_{01}(4i) + d_{01}(4i + 1)\}$
    $\mu_{\text{even}}(i)_{\bar{\omega}} = \min\{d_{01}(4i) + d_{10}(4i + 1), d_{10}(4i) + d_{01}(4i + 1)\}$
end for

Hexacode Decoding

The next part of decoding is finding the minimum distance codeword in the hexacode corresponding to the lattice point. This method is called for each of the 4 quarter leech lattice decoders $Q\lambda_{24}$. As mentioned before we will use the bounded distance decoder of [4].

The decoder of Amrani and Be’ery uses the bounded distance property of the half projections of the each $H_6$ codeword and only considers the least reliable or greatest sum of $\mu$’s of the corresponding half codeword’s block confidences to be the error in the codeword. Each possible alternate value, including the original value along with the other 2 characters of the half Hexacode word, are used to generate the full $H_6$ codeword from the encoding matrix above. The $\mu$ distances are then summed for the candidate codeword.
The least distance candidate codeword is then used as the quarter leech lattice \((Q\Lambda_{24})\) decoding. The partitioning of the \(H_6\) codeword is as follows.

\[
x_i \in GF(4)
\]

\[
x = [x_1, x_2, x_3, x_4, x_5, x_6]
\]

\[
x \in H_6
\]

\[
x = [y_1, y_2]
\]

\[
y_1 = [x_1, x_2, x_3]
\]

\[
y_2 = [x_4, x_5, x_6]
\]

\[
x = y_1 \times G_{(6,3,4)} = y_2 \times G_{(6,3,4)}^{-1}
\]

**Algorithm 7 \(H_6\) Decoding (First Half)**

**Require:** \(\mu_{even}, \mu_{odd}\)

\[
\text{min} = [0, 0, 0]
\]

for all \(x_i \in y_1\) do

  if \(\mu(x_i) < \text{min}[0]\) then

    \[
    \text{min} = [\mu(x_i), i]
    \]

  end if

end for

candidate=[\(y_1 \times G_{(6,3,4)}\)]

minword=[\(y_1 \times G_{(6,3,4)}, 0.0\)]

for all \(c \in GF(4)\) do

  \[
y_1(\text{min}) = c
\]

  \[
x = y_1 \times G_{(6,3,4)}
\]

  \[
  \text{sum} = 0
\]

  for all \(x_i \in x\) do

    \[
    \text{sum} = \text{sum} + \mu(x_i)
    \]

  end for

  if \(\text{sum} < \text{minword}[1]\) then

    \[
    \text{minword} = [\text{candidate}, \text{sum}]
    \]

  end if

end for

:

return \(\text{minword}\)
Chapter 5. Implementation Details

Parallel Lattice Decoder Implementation

Resolve H-Parity

The H-parity maintains the properties of the definition for the Hexacode based Lattice Decoding by assuring that the parity of each projection of the hexacode characters into $GF(2)^4$ matches the overall parity of the the top row of the first bit of each Hexacode characters projection. The H-parity is a result of part d of the Leech Lattice construction. If the resulting code is found to violate this parity restriction, the complementary projection will be used. The complementary project encodes the same character, and has the same overall parity (odd/even), however all the bits are their complements, therefor changing the overall first row parity. The method in which this is computed is straightforward, and consists of checking each of the 6 characters, and complementing the character that results in the least distance deviation from the h-parity violating codeword. The overall distance is then updated by adding the penalty of changing the characters projection, and is computed by summing the $d_{ij}$ bits of the complementary form.

Algorithm 8 Resolve H-Parity

Require: C (24-bit codeword), weight, $d_{ij}$’s, rep (even/odd)

\[ p = \text{parity}([c_1, c_5, c_9, c_{13}, c_{17}, c_{21}]) \]

if $p = \text{rep}$ then

return C, weight

end if

$\Delta_{c_{\text{min}}} = [0, 0]$

for all $i = 1$ to $6$: 4 do

$[\tilde{c}_i, c_{i+1}, c_{i+2}, c_{i+3}] = \text{comp}([c_i, c_{i+1}, c_{i+2}, c_{i+3}])$

$\Delta_c = (d_{c_i, c_{i+1}} + d_{c_{i+2}, c_{i+3}}) - (d_{\tilde{c}_i, c_{i+1}} + d_{\tilde{c}_{i+2}, \tilde{c}_{i+3}})$

if $\Delta_c < \Delta_{c_{\text{min}}}$ then

$\Delta_{c_{\text{min}}} = [\Delta_c, i]$

end if

end for

$i = \Delta_{c_{\text{min}}}[1]$

C = $[c_1, c_2, ..., \tilde{c}_i, c_{i+1}, c_{i+2}, c_{i+3}...c_{23}, c_{24}]$

return C, $\Delta_{c_{\text{min}}} + \text{weight}$

Resolve K-Parity

Similar to H-parity, K-parity is also required by the Hexacode construction of the Leech Lattice. In the case of K-parity the requirement is that the overall parity of the 24bit codeword be even if its an A-type lattice point, and odd if it is a B-type Lattice point. The K-parity corresponds to the density doubling by
combining both $H\Lambda_{24}$ half lattices resulting from the Binary Golay Code Lattice under construction B. For this reason, Resolving the k-parity is not needed for decoding just the Binary Golay Code. Also similar to $H$-parity, the method for dealing with violations of $K$-parity type is to find the least distance penalty from the current code, that satisfies the parity constraint. In this case, we only need to complement one bit of the code, which can be done by choosing the least distance $d_{ijk}$ QAM constellation point, in the codeword. This complement will satisfy K-parity without changing the codeword or violating $H$-parity. It will not change the resulting codeword, but it will change the overall distance, which is used by the 4 quarter lattice $Q\Lambda_{24}$ decoders to find the minimal candidate decoding.

Algorithm 9 Resolve $K$-Parity

Require: C (24-bit codeword), weight, $d_{ij}$’s, A/B

p = parity(C)
if p = A/B then
    return C, weight
end if

$\Delta_{cmin} = 0$
for all i=1 to 24: 2 do
    $\Delta_c = d_{c_i,c_{i+1}} - d_{c_i,c_{i+1}}\bar{k}$
    if $\Delta_c < \Delta_{cmin}$ then
        $\Delta_{cmin} = \Delta_c$
    end if
end for
return weight+$\Delta_{cmin}$

Final Decoding

The final decoding results in the minimum distance decoding of the 4 $Q\Lambda_{24}$ decoders. The above decoder description focuses on the $A$-type Even codeword decoding, however the other decoders: $A$-type Odd, $B$-type Even $B$-type Odd are also needed to perform this final step. The least weight codeword is the final decoding for the Leech Lattice ($\Lambda_{24}$).

5.1.3 Implementation of Lattice Based LSH

The implementation of the LSH portion of the c-approximate r-nearest neighbor search is not essential to the performance comparison of our method versus previous methods (10). For this reason, a performance
CHAPTER 5. IMPLEMENTATION DETAILS  PARALLEL LATTICE DECODER IMPLEMENTATION

deficient LSH implementation was built in python as a proof of concept that the parallel decoder would act
as a drop in replacement for a more optimized LSH implementation. The lookup algorithm used is a slightly
modified version of [10], in lines of the improvement suggested in [13] in which the decodings weight is
used to limit the size of the direct comparison vector to 2L, thus achieving the complexity bound of [10] for
c-approximate r-nearest neighbor search, while also providing a practical improvement to the approximation
factor.

5.1.4 Modifications for Parallel Implementation

The original algorithm, and modifications of [29] where taken into the final parallel LSH algorithm as well
as the improvement previously mentioned by [13]. These modifications improve upon the original search,
but are not specific to parallel processing. A great deal of effort was put into adapting the leech decoding
algorithm of [4] perform well on the CUDA framework for the Nvidia, GPGPU architecture. The next
portion of adaptation was for the LSH algorithm itself which was somewhat simply adapted to allow for
multiple simultaneous decoding of length 24 vectors, as a naive parallelization. The next improvement for
parallelism is the simultaneous decoding and comparison of the 4 quarter Leech Decoders.

CUDA Improvements

We will begin with the CUDA improvements to decoding the Leech Lattice vectors. We will not cover
every improvement, as many are trivial, and follow a similar pattern to how other improvements were im-
plemented.

The first important aspect of the CUDA architecture as mentioned in the background section, is that
the SMP cores running in a half-warp, are executed functionally in lock-step. This is a direct consequence
the cores being SIMD partitioned per half-warp. Due to this limitation, multiple branch path code will
always operate at the slowest branch. Although our improvement does not directly remove this limitation, it
does mitigate its greater than slowest branch path penalties by allowing better memory access patterns to be
exploited by CUDA scheduler. The CUDA scheduler is able to perform concurrent memory transfers in this
case, as opposed to the otherwise conditional access that would result from branching execution. The way
in which conditionals are dealt with throughout the the code, is by pre-computing the results of conditional
operations, and allowing the result to determine the memory locations.

example: The below conditional code:

```c
if(dist000<dist001)
{
    dijs[i*4]=dist000;
    dijks[i*4]=dist001;
    kparities[i*4] = 0;
}
else{
    dijs[i*4]=dist001;
    dijks[i*4]=dist000;
    kparities[i*4] = 1;
}
```

is replaced by the following code with sequential access:

```c
d = dist000<dist001;
    dijs[i*4]=dist000*d+dist001*(!d);
    dijks[i*4]=dist001*d+dist000*(!d);
    kparities[i*4] = (!d);
```

In the event of the conditional code, both conditions of the if statement must be traversed on average for the 8 sequential cores. This results in 2 accesses for the 3 distance arrays, which in CUDA architecture results in a major slowdown. In the concurrent code, the conditional array accesses occur simultaneously, which allows the scheduler to swap in an non-blocking IO execution block, while the current block awaits data. Furthermore, the additional computation is negligible, as the cores allow for some pipelining of arithmetic operations, that would otherwise not be utilized. Unfortunately the overall execution paths are difficult to measure or visualize as the CUDA framework is somewhat closed from view. Instead results can only be obtained from full system simulation with random point data. This data is then used to measure basic speedup. In the case of this alteration for most conditional branches in the code, the results were significant (3:5 speed increase).
Naive LSH Improvements

As mentioned above, most of the improvement to the LSH KNN algorithm, is contained in the improvement of the Lattice Decoding. In fact the only parallel specific changes to the original decoding, consist of issuing simultaneous decoding tasks to the GPU. For this reason, the parallel decoder is best suited to running on systems that have very heavy concurrent query loads, or ones with considerably long query vectors. In addition, it is also adept at dealing with highly transient datasets in which the lattice query database must be rebuilt often. The parallel aspects are presented below, as well the previously mentioned addition pertaining to the improvement of [13] for using the distance metric on query adaptive search of the E8 lattice, for the Leech Lattice. The algorithms below state the final algorithm used for our parallel LSH KNN algorithm.

Some notation details:

\( \bar{x} \) is a string or list

\( \hat{x} \) is a permutation of \( x \)

\( \otimes \) is the concatenation operation

\( \oplus \) is the list append operation

\( H(x) \) returns the standard string object hash in python

Figure 5.1.4: E2LSH Framework: Query
**CHAPTER 5. IMPLEMENTATION DETAILS**

**PARALLEL LATTICE DECODER IMPLEMENTATION**

E2LSH Framework

Pre-processing:

- 1. Separate vector into random samplings of length 24 vectors
- 2. Concatenate lattice centers of the sampled vectors to create one ConcatID
- 3. Store the ConcatID as the entry to the hashtable linked to a list of Data vector IDs with the same ConcatID.

![Figure 5.1.5: E2LSH Framework: Preprocessing](image)

\[ N(\mu, \sigma) \] returns a random variable normally distributed with mean \( \mu \) and variance \( \sigma \)

\( \Lambda_{24}(x) \) is the leech lattice decoder

\( map(f(x), GPUs) \) maps \( f(x) \) to the available GPU Cores

**Using Available Shared Memory**

Due to the Nvidia GPGPU architecture having a highly partitioned, pertaining to memory access, processing structure, memory sharing for data residing in the Shared Data location, is fast for small groups of threads (≤16 threads). In particular, shared memory can be accessed at speeds on par with register memory. This is in comparison to global memory, which comes at a penalty of around 100-200 cycles according to Nvidia’s Specifications [30].
Algorithm 10 Preprocessing: Parallel Lattice LSH KNN

Require: $X = \{x_1, ..., x_d\}, x_i \in \mathbb{R}^d$, GPUs
choose $k$ s.t. $k = \frac{\log n}{\log 1/P_2}$
for all $i \in \text{Range}(k)$ do
  $t = [N(0,1)_{\sqrt{2^1}}, N(0,1)_{\sqrt{2^1}}, \ldots, N(0,1)_{\sqrt{2^d}}]$
  $g(x,t) = \Lambda_{24}(\{x_1, ..., x_{24}\} \times t)$
  $T \leftarrow t$
for all $i \in \text{Range}(m)$ do
  $D \leftarrow D \oplus \text{map}(\bar{x} = x \otimes g(x, t), \text{GPUs})$
  $i \leftarrow i + \text{len}(\text{GPUs})$
end for
for all $\bar{d} \in \bar{D}$ do
  $D = D \oplus H(\bar{d})$
end for
return $D,T$

Algorithm 11 Query: $c$-approx k-nearest neighbor

Require: $q \in \mathbb{R}^n, D,T$, GPUs
choose $\rho = \frac{\log P_1}{\log P_2} = .3674$ for leech lattice \cite{1}
for all $t \in T$ do
  $g(x,t) = \Lambda_{24}(\{x_1, ..., x_{24}\} \times t)$
for all $i \in \text{Range}(n^\rho)$ do
  $\tilde{q} = q + \{N(0,1)_{\sqrt{2^1}}, N(0,1)_{\sqrt{2^1}}, \ldots, N(0,1)_{\sqrt{2^d}}\}$
  $\tilde{Q} \leftarrow \tilde{Q} \oplus \text{map}(q = \tilde{q} \otimes g(\tilde{q}, t), \text{GPUs})$
end for
choose $l_{\text{max}}$ to be the maximum list length
for all $\bar{q} \in \bar{Q}$ do
  if $|L| < l_{\text{max}}$ then
    $L \leftarrow D[H(\bar{q})]$
  else
    $L[l_{\text{max}}] = D[H(\bar{q})]$ if distance is less than list’s least distance
  end if
end for
return $L$
CHAPTER 5. IMPLEMENTATION DETAILS PARALLEL LATTICE DECODER IMPLEMENTATION

<table>
<thead>
<tr>
<th>Step</th>
<th>data field</th>
<th>data type</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>QAM</td>
<td>dijs</td>
<td>float</td>
<td>$12 \times 4$</td>
</tr>
<tr>
<td></td>
<td>dijks</td>
<td>float</td>
<td>$12 \times 4$</td>
</tr>
<tr>
<td></td>
<td>kparities</td>
<td>byte</td>
<td>12</td>
</tr>
<tr>
<td>Block Confidence</td>
<td>muE</td>
<td>float</td>
<td>$4 \times 6$</td>
</tr>
<tr>
<td></td>
<td>muO</td>
<td>float</td>
<td>$4 \times 6$</td>
</tr>
<tr>
<td></td>
<td>prefRepE</td>
<td>byte</td>
<td>$4 \times 6$</td>
</tr>
<tr>
<td></td>
<td>prefRepO</td>
<td>byte</td>
<td>$4 \times 6$</td>
</tr>
<tr>
<td>minH6</td>
<td>codeword</td>
<td>byte</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>charwts</td>
<td>float</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.2: Memory Requirements of Lattice Decoding Data Structures

Overcoming Shared Memory Limitations

The first attempt at a CUDA optimized implementation of the Leech Lattice Decoder simply tries to use shared memory instead of global memory. Intuitively the more we rely on faster memory, the faster the overall algorithm should operate. In this vain all global memory allocations were replaced with shared memory allocations. Immediately an issue arises as the maximum shared memory size is only 16kB per block. Given our data requirements, a single decoder needs about around 778 bytes. The data footprint of an $H_{A_{24}}$ decoder is shown below (the other half uses the same memory and follows sequentially).

Given the shared memory limit and the required data footprint for the decoder, a block can contain no more than 20 simultaneous lattice decoder threads. However according to NVIDIA's documentation on CUDA Occupancy and memory optimization [30] threads per block should exceed 32, allowing for two fully occupied halfwarps to be run.

To remedy we decided to trade computation for memory occupancy. In this case it comes in the form of recomputing the dijk metrics from the QAM portion of the lattice decoder. By doing this along with reusing the k-parity array as the prefreps array, and the dijks array as the muE and muO arrays, we are able to compress the memory footprint down to 444 bytes. The memory requirements are shown in the table below.
CHAPTER 5. IMPLEMENTATION DETAILS

5.1. PARALLEL LATTICE DECODER IMPLEMENTATION

<table>
<thead>
<tr>
<th>New Datafield</th>
<th>Datatype</th>
<th>Size</th>
<th>Variable Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>dij</td>
<td>float</td>
<td>12 × 4</td>
<td>always needed</td>
</tr>
<tr>
<td>fblock</td>
<td>float</td>
<td>48</td>
<td>dijk’s→[QAM,kparity] (μE,μO) →[BlockConf, H6]</td>
</tr>
<tr>
<td>bblock</td>
<td>byte</td>
<td>48</td>
<td>kparities→[QAM,kparity] (prefRepE,prefRepO)→[BlockConf, h-parity]</td>
</tr>
<tr>
<td>yE</td>
<td>float</td>
<td>6</td>
<td>always needed</td>
</tr>
<tr>
<td>yO</td>
<td>float</td>
<td>6</td>
<td>always needed</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td></td>
<td>444 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Compressed Data Structures Requirements for Lattice Decoding

Using this reformatting with our new compressed memory footprint, we are able to theoretically run 36 threads per block. In general blocks should be multiples of 32, so we set our thread per block count to 32.

**Partitioning the Leech Decoder over the QΛ_{24} Subdecoder**

Although the above reorganization allows us to meet NVIDIA’s minimum requirements, they suggest having far more threads per block in order to allow the CUDA thread scheduler to efficiently schedule threads to optimally occupy the GPGPU’s processing cores.

At this point further optimization will not be achieved by compressing memory footprints any further, and instead a reorganization of our algorithm is needed. Fortunately the Bounded Distance Hexacode Based Decoder of Amrani et al. \[12\] has a fairly straightforward method of parallelization. The partition simply breaks the decoder along the lines of the 4 QΛ_{24} subdecoders. In addition to parallel computation, it may also be useful to utilize the shared nature of CUDA’s shared memory to our advantage. We do this by looking at the memory overlaps of the various subdecoders. The Venn Diagram\[5.1.6\] shows where the 4 quad decoders overlap in their memory requirements, where each circle represents a QΛ_{24} decoder.

Using the QΛ_{24} partitioning along with the conservative memory organization in the Figure\[5.1.6\] we get a new outline for our decoder.

The below reorganization of the Lattice Decoder\[5.1.7\] results in the parallel execution of the 4 QΛ_{24} lattice decoders of the full Leech Lattice decoder. This decoder requires 1344 bytes of shared memory, but exists as 4 separate threads. Using the same calculation as before to see how many simultaneous threads we can run, we get \(16kB/1344B = 12\) decoders. But since each decoder consists of 4 threads, we can actually
CHAPTER 5. IMPLEMENTATION DETAILS  PARALLEL LATTICE DECODER IMPLEMENTATION

get 48 simultaneous threads per block.

5.1.5 SIFT Application Search

In the case of SIFT vectors, the length of the vectors is fixed at a 128 dimensional based image search was created to demonstrate the speed and validity of our parallel KNN search.

<table>
<thead>
<tr>
<th>New Datafield</th>
<th>Datatype</th>
<th>Size</th>
<th>Need By</th>
</tr>
</thead>
<tbody>
<tr>
<td>dijA</td>
<td>float</td>
<td>12 × 4</td>
<td>AE, AO</td>
</tr>
<tr>
<td>dijB</td>
<td>float</td>
<td>12 × 4</td>
<td>BE, BO</td>
</tr>
<tr>
<td>dijkA</td>
<td>float</td>
<td>12 × 4</td>
<td>AE, AO</td>
</tr>
<tr>
<td>dijkB</td>
<td>float</td>
<td>12 × 4</td>
<td>BE, BO</td>
</tr>
<tr>
<td>musEA</td>
<td>float</td>
<td>6 × 4</td>
<td>AE</td>
</tr>
<tr>
<td>musOA</td>
<td>float</td>
<td>6 × 4</td>
<td>AO</td>
</tr>
<tr>
<td>musEB</td>
<td>float</td>
<td>6 × 4</td>
<td>BE</td>
</tr>
<tr>
<td>musOB</td>
<td>float</td>
<td>6 × 4</td>
<td>BO</td>
</tr>
<tr>
<td>PrefrepsEA</td>
<td>byte</td>
<td>6 × 4</td>
<td>AE, BO</td>
</tr>
<tr>
<td>PrefrepsOA</td>
<td>byte</td>
<td>6 × 4</td>
<td>AO</td>
</tr>
<tr>
<td>PrefrepsEB</td>
<td>byte</td>
<td>6 × 4</td>
<td>BE</td>
</tr>
<tr>
<td>PrefrepsOB</td>
<td>byte</td>
<td>6 × 4</td>
<td>BO</td>
</tr>
<tr>
<td>kparitiesA</td>
<td>byte</td>
<td>12 × 4</td>
<td>AE, AO</td>
</tr>
<tr>
<td>kparitiesB</td>
<td>byte</td>
<td>12 × 4</td>
<td>BE, BO</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td></td>
<td>1344 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Quarter Lattice Data Structure Requirements
CHAPTER 5. IMPLEMENTATION DETAILS PARALLEL LATTICE DECODER IMPLEMENTATION

Figure 5.1.7: Parallel Leech Lattice Decoder
CHAPTER 5. IMPLEMENTATION DETAILS

5.1. PARALLEL LATTICE DECODER IMPLEMENTATION

Generating SIFT Vectors

The generation of SIFT vectors for images is performed by the program from SIFT [9], which generates a set of $n \times 128$ of integers corresponding to the scale invariant features of the image. A database is generated for a set of images, and stored in a database file using python’s ‘shelve’ data structure. The images are then copied to a test location, where each images is rotated randomly by [0,90,180,270] degrees. The image then undergoes a Gaussian blur and a 70% compression via jpeg encoding using tools from the Imagemagick [31] package for linux. The deteriorated image set is then used as the query set for the KNN algorithm. The goal of using this algorithm is to achieve similar results to the exact matching of SIFT vectors. One issue arises for large image databases, in that the latter exact search becomes computationally infeasible. For this reason, accuracy tests were generated on a much smaller dataset, and a larger dataset was then used for testing the speed of the algorithm.

Test Plan for SIFT Searched LSH-KNN vs Linear Search

Using a random subset sampling of the above database, we compared our search application in terms of speed and time to that of pseudo-linear search. We first must define the linear search technique as it is somewhat different than a standard search technique, and is built around the SIFT application’s match demo program. Each image is treated as a bag-of-words set of SIFT vectors. A true exhaustive search would need to find a minimal pairing of all vectors in the query image to the that of every image in the image database. Matching each vector in a set of vectors with its best match in another set of vectors results in solving the Stable Marriage Problem(SMP). This problem is known to have an optimal solution in $O(n^4)$ time [32]. For SIFT vector Image search $n$ is the average number of SIFT vectors per image. The number of SIFT vectors for an image varies with the images content and can be in the 1000’s for some images. In that case the resulting $O(\cdot) \rightarrow 10^{12}$ is infeasible for any test simulation. Instead, Lowe suggests a different, but for our purposes, reasonable algorithm which still suffices to provide an example of linear search. This algorithm consists of finding the two nearest matches between a SIFT vector and a candidate image’s SIFT vectors. If the closes vectors distance to a vector is less than the $0.6 \times$ the next closest vector, then those vectors are recorded as a match. For our comparison we will simply run both the pseudo-linear search and our approximate LSH-KNN search on the images and the image set, and record the amount of time to query
the database, along with the accuracy of matches.

**Standard Test of LSH-KNN**

We will also present a standard LSH-KNN and in fact standard data mining search metric as utilized by Pauleve and Jegou [21] [13]. Pauleve and Jegou present a method of demonstrating an LSH algorithms quality by showing its selectivity as a metric of the amount of the database to be linear searched and the recall of that search. In the context of LSH this would be the length of the intersection of all the retrieved hash buckets for a given query compared to the size of the database, and the recall of that search. The metric is similar to precision recall or standard PR curves, and fairly accurately shows the quality of an LSH function in terms of its ability to differentiate near and far neighbors.
Chapter 6

Performance Analysis

The performance analysis section will consist primarily of performance in terms of speedup for the lattice decoding process due to the parallel CUDA implementation. In addition, the current algorithm is compared to an exact match algorithm in general data mining metrics of precision recall and ROC. These metrics will be used to establish the approximation loss of our approximate search method versus the exact search, and not the quality of the SIFT generation method, in terms of recall. Again we are not stating anything new here, and are simply showing an equivalence to similar approximate search methods using the LSH scheme of [1]. Following the qualitative metrics of the LSH algorithm and parallelization, we will discuss the performance of the leech decoder itself, again to establish equivalence to the current algorithm. In addition, following suggestions of [13] we consider the benefits and trade offs of using a different lattice, namely the E8 lattices, as a space quantizer for a our hashing family.

6.1 Performance of the CUDA Design

The primary result of this analysis will be to show the decoding speed up for the leech lattice from our new parallel implementation. In terms of the LSH algorithm using lattice decoding, we will show the approximate amount of available speedup by comparing the amount of time spent decoding lattice vectors, to that of performing the database lookup and concatenation. Because our LSH algorithm is written in python, the amount of available speed up is somewhat of a lower bound, as the LSH algorithm could be further optimized in a non-interpreted language such as C. Figure 6.1.1 shows the speedup ratio of a single
CHAPTER 6. PERFORMANCE ANALYSIS

6.1. PERFORMANCE OF THE CUDA DESIGN

gpu, with 240 processing cores, versus a similarly clocked standard CPU. The independent variable is the CUDA, user specified block size. The block size defines the restriction on which the CUDA GPGPU’s scheduler is required to adhere to. The primary result of speedup can be taken as the optimal block size selection for the GPU, as well as the overall speedup resulting from a single GPU.

The next two Figures 6.1.2, 6.1.3 show the total time for actual SIFT transformed image search data, versus the amount of time spent in a sequential decoder. Using the sequential decoder on a similar processing architecture as the rest of our algorithm, we can get a better ratio of sequential to parallel code, which we can apply to our parallel CUDA speedup. As stated in the Performance Analysis overview, the sequential decoder is an optimize C implementation, while the LSH algorithm is an unoptimized python implementation. We will first look at the KNN search or query speed up as a result of our parallel leech decoder. The average decoding time ratio to total LSH computation time is 0.9340. Due to the mixed python and C code this is likely a lower bound for the ratio of parallel to sequential code, and thus parallel to total time, as python implementations are generally less efficient than C implementations. From a standpoint of parallelism we

![Figure 6.1.1: CUDA Blocks per Speedup](image)

Figure 6.1.1: CUDA Blocks per Speedup
would hope for this number to be as high as possible, as greater ratio of parallel to sequential code will result in better overall performance for our parallel implementation as the amount of parallel speedup increases. Figure 6.1.2 shows the ratio of time spent in parallel portions for the code of an LSH query, versus the total LSH query time. The sample data is a set of 294 random images from the Caltech 256 image dataset’s \[33\] generated SIFT vectors. E2LSH queries

The next portion of the algorithm is the generation of the database for querying. This portion is slightly less important for optimization as its generation is often performed offline and is often not performance imperative. Similar to our analysis above, Figure 6.1.3 shows the ratio of time spent in parallel portions LSH database generation, versus the total database generation time. Additionally this was performed over the same random set of images and SIFT vectors as above. Here the average ratio was smaller, and in the case of speedup, will have an adverse effect on performance increase from parallelism.

Figure 6.1.2: Ratio of Parallel to Total Time for Queries on an LSH Search Database using Real World SIFT Sample Data
6.1. PERFORMANCE OF THE CUDA DESIGN

Figure 6.1.3: Ratio of Parallel to Total Time for the Generation of the LSH Search Database using Real World SIFT Sample Data
6.2 Performance Analysis of the Leech Lattice

This section demonstrates the performance of our Leech Lattice decoder in terms of a standard error correcting code. We demonstrate this comparison as a confirmation of our parallel Leech Lattice decoder’s performance by way of comparison with other error correcting codes of known differences in gain. We also show a few different unencoded modulation schemes. The unencoded modulations are very similar to the results of the hash function described in earlier papers on LSH using random projection [7]. This is due to the QAM lattice being essentially $\mathbb{Z}_2$ partitioned as a 16 subset constellation. As stated in the background section, all codes coding gain is fundamentally limited by the Shannon Information Limit. Therefore we hope to establish a sense of diminishing returns as the complexity of our codes and modulation schemes increases. Naturally our error correcting codes below demonstrate this information limit as shown in Figure 6.2.1.

The probability of intersection versus distance plot is another way to verify the merit of a an error

![Error Correcting Performance (3072000 Samples)](image)

Figure 6.2.1: Performance of Some Coded and Unencoded Data Transmission Schemes
The Leech Lattice is of great mathematical importance, however the question as to why its useful for approximate nearest neighbor search has not been experimentally shown. In particular it needs to be shown that the Leech Lattice has some characteristics that separate it from other options for space quantizing our dataset’s embedding. One such option is to use a space quantizer of lower decoding complexity at the cost of performance. The advantage however in the second decoding algorithm is that it has a considerably smaller decoding complexity, making the loss an acceptable trade-off.

6.2.1 Why the Leech Lattice

The Leech Lattice is of great mathematical importance, however the question as to why its useful for approximate nearest neighbor search has not been experimentally shown. In particular it needs to be shown that the Leech Lattice has some characteristics that separate it from other options for space quantizing our dataset’s embedding. One such option is to use a space quantizer of lower decoding complexity at the cost of performance. The advantage however in the second decoding algorithm is that it has a considerably smaller decoding complexity, making the loss an acceptable trade-off.
of residing in a lower dimensional space. On the other hand it may be more advantageous to lower the base factor of complexity by increasing the dimensionality of the projected space, at the cost of a higher decoding complexity. Both of which will be considered below. The Figure 6.2.1 shows the error correcting performance of the E8 and Leech Lattices. The Leech Lattice is modulated by the 16-QAM lattice while E8 is a binary-PSK scheme. Two unencoded 64 and 16 QAM schemes are also presented, to show the effects of the different modulation schemes on data.

E8

One such consideration is that of Jegou’s article on using E8 instead of the Leech lattice as a space quantizer [13]. In their work, they suggest that although the Leech Lattice is of academic interest, for LSH Based KNN, it may be inappropriate. They show this as a trade-off between decoding complexity and intersection probability in lieu of other codes with far simpler decoding complexities, and only slightly less attractive intersection probabilities.

In this thesis we would like to refute this conclusion at least for sequential lattice based LSH, a parallel consideration is left to future work and discussed in the Future Development section. To evaluate an LSH function it is essential to develop a metric for quality. Here we will use the \( \rho = \log(1/p_1)/\log(1/p_2) \) metric as means for lattice LSH algorithms. Because we are only comparing \( \Lambda_{24} \) and E8, it is reasonable to set the decoding region of \( \Lambda_{24} \) as our standard, and try to find the needed replication and permutations of E8’s sub projections in order to achieve the same correction radius. The Figures 6.2.1 and 6.2.1 show a comparison between our lattice decoders distance to decoding probability (same method as 6.2.2) and the E8 lattices decoding probability for various values of k as the random projection replication factor. Because the points were taken from normally distributed points in \( \mathbb{R}^{24} \) (the Leech Lattices native space) from n centroids, we will assume the goal of the E8 lattice is to replicate its \( \rho \) value by employing the concatenation of decodings along with the multiple independent rounds of projection and hashing. These options are familiar as they are the same methods used to apply the leech lattice in dimensions that are greater than 24. A further analysis would demonstrate a variety of greater than native embedding dimensionality for the candidate lattices, and compare their \( \rho \) values as a function of increasing d.
CHAPTER 6. PERFORMANCE ANALYSIS

6.2. PERFORMANCE ANALYSIS OF THE LEECH LATTICE

Figure 6.2.3: Correct Decoding Probability Rate of k-E8 sub projections compared to the Leech Lattices Decoding Rate

Figure 6.2.4: Correct Decoding Probability Rate of k-E8 sub projections compared to the Leech Lattices Decoding Rate, (Zoomed)
6.2.2 KNN-LSH and Exhaustive Search of SIFT Vectors

In this section we will use our image search application utilizing KNN-LSH to search for near neighbor vectors of images generated by the scale invariant feature transform [5]. Our goal in this paper is not to show any gain or superiority over other search methods, but instead to simply establish an equivalence to other similar methods. Our main result was generated from a random selection of 294 images from the Caltech 256 Image Database [33]. The reason for selecting such a small subset of images was the nature of the linear search technique employed for comparison to our algorithm. We will present 3 metrics here in regards to the performance of our LSH algorithm. As stated before, the LSH portion resides mainly in GPGPU processing while the LSH-KNN framework is implemented in python. For this reason the times are considerably slower than an optimized LSH-KNN would yield, however the overall complexity as a function of \( n \) should be consistent with our expectations of the KNN-LSH algorithm of Andoni [10]. After establishing equivalence between our algorithm and previous similar LSH algorithms, we will show a comparison by way of Selectivity Recall comparing our new query-adaptive Leech Lattice search algorithm, and that of the previous unsorted algorithm. Due to the heuristic nature of the nearest lattice point ranking, the overall complexity bound will remain unchanged. However the adaptive algorithm shows significant advancement in real world applications for exact c-approximate nearest neighbor search.

Comparison of Time for Searching

Figure 6.2.2 shows a random subset of times to perform exact linear searches as well as times to perform LSH searches for a given size database A few issues prevent us from showing the true advantages of LSH searching, due to linear searching quickly becoming intractable for large databases. Initially a subset was sampled from the database and used to generate an approximate average time. However, even for relatively small image databases (\( n > 1000 \) images, \( \approx 300 \) SIFT vectors each) results from linear searching began to be effected by memory thrashing, which is not a basis for our comparison. Furthermore, the variance, as depicted by the error bars for linear search, increases as the database size increases. This is due to images with \( \gg 300 \) SIFT vectors, requiring search complexity increasing with \( n \) thus generating an increasing variance.

To avoid these issues, and still provide a reasonable comparison for more realistically sized datasets,
we will simply refer to the complexity bounds as referenced in [4.1.3] and further derived and discussed in Andoni’s Dissertation [1]. We will extend the exact times from data in Figure 6.2.2 and use this formula along with the linear search complexity, to extrapolate speedup. Figure 6.2.2 shows the previous chart extrapolated using the corresponding big-oh functions for LSH and Linear search, in a semi-logarithmic chart, for an n size search database. The functions used to fit the previous data are $\Theta(n^\rho)$ and $\Theta(n)$, where $\rho = .3671$ and least squares constant values are used to establish a curve fit via the somewhat overkill, but standard Levenberg Marquardt variable optimization algorithm. Figure 6.2.2 and Figure 6.2.2 show the fit quality of the corresponding equations. An important note, is that the LSH algorithm tends to jump, instead of grow gradually with n. This is due to the random projection matrix generation being a discrete variable, with direct implications on the search time. The variable is still bounded above by $\Theta(n^\rho)$ according to Panigrahy [27] as n grows large.
It is clear from the figure that sub-linear time approximate methods are the only viable methods for searching large databases of SIFT vectors.

**Accuracy of Searching**

Figure 6.2.2 shows a chart comparing linear exhaustive search with lattice LSH search accuracy at various values for the maximum exhaustive LSH search bucket size. The metric $l$ refers to the number of exhaustive search vectors to be queried as selected by the LSH hash function and pruned to length $l$. Prior to pruning, a distance ranking is employed, based on the vectors distance to a lattice center or hash bucket. In this modification to the original Lattice LSH [10], closer lattice centers have precedence over more distant ones as provided by the decoding algorithm. The intuition is that closer lattice representations provide a better representation for the given vector. This heuristic is in line with that of “Query Adaptive LSH” [13].

![Extrapolated Comparison of Time for LSH and Linear Search (Semilog)](image-url)

Figure 6.2.6: Extrapolated Average Search Query Times
CHAPTER 6. PERFORMANCE ANALYSIS

6.2. PERFORMANCE ANALYSIS OF THE LEECH LATTICE

Figure 6.2.7: Fit of $n$ for Linear Search Data

Figure 6.2.8: Fit of $n^p$ for LSH Search Data

<table>
<thead>
<tr>
<th>L</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.7848605577689243</td>
</tr>
<tr>
<td>10</td>
<td>0.8725099601593626</td>
</tr>
<tr>
<td>10</td>
<td>0.900398406374502</td>
</tr>
<tr>
<td>Linear Search</td>
<td>0.9881889763779528</td>
</tr>
</tbody>
</table>

Table 6.1: Results of SIFT Search Accuracy
Selectivity Recall

Figure 6.2.2 comes from a similar analysis as in “LSH: A comparison of hash function types and querying mechanisms” [21] which compares the selectivity of various hash functions, and implies a natural heuristic for comparing hash functions in terms of their KNN-LSH search utility. In the figure we are comparing our new distance weighted selectivity for the leech lattice, against the previous unranked method. The results are fairly dramatic, in regards to the much better selectivity of the weighted version.
Figure 6.2.10: Data Recall Selectivity for Query Adaptive and Unranked LSH
Chapter 7

Conclusions and Suggestions for Future Research

This chapter contains a summary of the findings of this thesis. The section nearly mirrors the previous section and offers a digest of the graphs and data presented in those sections. In addition to our empirical results, we present the theoretical speed up of the LSH algorithm in regards to a parallel decoder. This result is of considerable importance as it sets optimal bounds on the advantages of parallel decoding in regards to sequential verse parallel code. Also of interest are some lessons learned while trying to optimize code for the CUDA Framework for GPGPUs.

7.1 Summary of Findings

7.1.1 Parallel Lattice Decoder

Naively Parallel vs. Shared Memory Decoder

The first and most apparent discovery was the naively parallelized sequential algorithm out performing the seemingly more CUDA optimized shared memory algorithm. Despite efforts to better utilize the GPGPU’s faster shared memory, the naively parallel algorithm, using 100x-200x slower global memory [30] still performed better then our shared memory decoder. Furthermore, the performance was not a direct result of block size limitations resulting from the shared memory decoder, and even for the same smaller block sizes,
outperformed the shared memory decoder.

**Lattice Decoder Speedup**

The lattice decoder, at optimal block sizing, offered a speedup of nearly 41x for a single gpu. In our system we did not test all 4 simultaneous gpu’s, but given the naively implemented parallelism, it would stand to reason a speedup of $4 \times 41x = 164x$ could be achieved with proper data transfer overlapping to the CPU’s main memory.

**7.1.2 LSH-KNN Speedup**

The analysis of the overall LSH-KNN algorithm will be further explored in the Detailed Conclusions section by combining the results of the parallel implementation of the Lattice Decoder with the sequential implementation’s ratio of parallel code to sequential code. The results for the query and database construction/preprocessing algorithms ratios of sequential to parallel code on average, suggests a ratio of 0.9450 and 0.8453.

**7.1.3 E8 vs Leech**

The minimum squared error distance value for $k$ the replication factor for random projections of vectors from $\mathbb{R}^{24} \rightarrow \mathbb{R}^{8}$ was 1. Or simply resulted from dividing the 24 dimensional space arbitrarily into 3, 8 dimensional spaces. E8 lattice decoding was performed then performed for each of the 3 sub-projections.

The majority of our findings based on the figures in this section are qualitative, and it is difficult to state any other concrete observations. For this reason the results of this section are reserved for the detailed discussion section below 7.2.3.

**7.1.4 Comparison to Exhaustive Search of SIFT Vectors**

**7.2 Detailed Conclusions**

Following suit with the previous section, here we will present the conclusions of our findings in the same order as above. This section will act as the discussion of the results of the previous two sections and will
attempt to give some explanation of why some things worked and others didn’t. In addition we will consider some other options for LSH functions based on lattices in the E8 section.

### 7.2.1 Parallel Lattice Decoding

As a single component, optimizing the lattice decoder turned out to be a success. The system exhibited considerable speedup while operating on similar power constraint and cost of a single processor system. We find that the optimal block size for the algorithm lies below the CUDA suggested values, and only slightly above their lower limits for block sizes. This is likely due to the high memory overhead in relation to the usually tiny memory overhead of GPGPU programs. The limit caused us to either choose a lower block size if we wanted to reside all in fast shared memory, or endure global memory access hits, if we wanted to option for a larger block size. The results here are somewhat confusing as increasing the block size for the global memory version did not increase the speedup as expected, while the global version at the same block sizes, still performed better than the more latency conscious shared memory version. Unfortunately it is difficult, due to the hidden scheduling and memory allocation actions of the NVIDIA architecture, to know exactly why this is the case. One hypothesis is that the scheduler was able swap blocks more efficiently. This is due to the complete disuse of shared memory, which may have resulted in faster swapping of thread blocks due to the lifted memory shared memory swapping requirement.

### 7.2.2 LSH-KNN Speedup

In this section we will draw conclusions on the theoretical speedup of our parallel LSH-KNN algorithm using Amdahl’s Law for parallel speedup. Although we get good speedup from the GPU LSH function, the algorithm will ultimately be constrained by Amdahl’s law. This constraint is rather bleak, as even with very efficient implementations of the KNN Hashing algorithm, we will always be bounded by some portion of sequential code. The often referenced sequential to parallel bound is Amhdal’s Law. The equation for Ahmdal’s law regarding theoretical speedup as a function of the achievable parallel speedup ($S$), with a constant ratio of sequential to parallel code $P$, is stated below.

\[
\frac{1}{(1 - P) + \frac{P}{S}}
\]
Figure 7.2.1: Theoretical Speedup for Serial and Parallel Code as a function of Parallel Speedup for Database Query

Our calculated ratio of sequential to parallel time as a function of time ratio for sequential lattice decoding versus sequential KNN-LSH code can be seen in Figure 6.1.2 and is on average .9340. We will show this ratio in addition to other standard ratios in Figure 7.2.2. The blue rectangular area depicts the feasible speedup region of our parallel leech decoder on a single GPU. An ideal goal of parallelizing LSH-KNN would be to have a sequential to parallel code ratio that generates a curve which has a constant slope ratio of 1 or slightly less in this region. From the theoretical curves, 97% Parallel(3% Serial) generates a fairly decent speedup in our region of parallel optimization but is far below the ideal goal. A solid black horizontal line shows the predicted parallel speedup using our parallel to serial ratio and the parallel lattice decoding speedup (41x). Using this information we conclude that increased speedup of the lattice decoding is still a somewhat fruitful option as the curve suggests a reasonable speedup increase. However it soon will show signs of diminished returns as the parallelism is increases, and the sequential to parallel code remains constant.

The next speedup analysis is consider for parallel database generation. For this analysis we use a similar construction for speedup as above. Again we begin by considering the ratio of sequential to parallel code which can be seen in Figure 6.1.3. In this case however, the ratio is lower and for Amdahl’s Law curves this
7.2. DETAILED CONCLUSIONS

Figure 7.2.2: Theoretical Speedup for Serial and Parallel Code as a function of Parallel Speedup for Database Generation

has a big impact on overall achievable speedup. The average sequential to parallel ratio is \( 0.8453 \). When we plug this into the equations for Amdahl’s law we get a considerably less optimizable parallel implementation.

With the above speedup up ratio, we consider the case of multi-gpu decoding. In the results section multi-gpu decoding was predicted to achieve a nearly 4x increase for the corresponding 4 GPUs in our system. However, this range would require speedup ratios much closer to 100% Parallelism. For this reason, further optimization of the Leech Decoding algorithm should be forgone and more focus placed on speeding up or parallelizing the LSH-KNN code. Examples of ways to speedup the algorithm further, can be found in the Suggestions for Future Work section.

However there is one issue not considered in our analysis of this portion of code. This issue is in general suggested by the overall use of python for the LSH-KNN code. It is likely that the processing inefficiencies of python extend the processing durations needed for the sequential portions of code. However there may be another culprit in regards to the poor running time ratio of our LSH-KNN implementation. The difference lies within the general database where concatenated LSH hashes are stored. Instead of using a universal hash function for a known size set as defined in most papers pertaining to LSH, we simply use the python...
shelve database hash. This hash is meant for any python object, and hashes items to an undoubtedly larger hashing universe than is need for our LSH vectors. In addition, we are writing to disk for each submission instead of using some more advanced buffer aware database.

### 7.2.3 Comparison to E8 and other Lattices

In the above Figures 6.2.1 and 6.2.1 it is difficult to quantify exactly what makes up a good LSH decoder. One intuitive suggestion would be something that can discretely differentiate between near and far points. The problem with this definition is that being able define near and far may require knowledge of the application domain, and in the case of a general KNN search, would not be available. So instead we were forced to assume that the Leech Lattice, and other densest lattices for that matter, operate optimally in their native dimension. Using this intuition we attempted to vary \( k \), the random projection replication factor to achieve a result similar to the leech lattice in terms of our ‘collision at a distance’ probabilities. The results are fairly messy, with the closes curve being at \( k = 1n \) in which the vectors \( x \in \mathbb{R}^{24} \) were decoded as the concatenation of 3 E8 lattice decodings. Although empirically this resulted in the least squared error distance from the Leech Lattice curve, it must be somehow incorrect. The intuition behind this comes from the densities of lattices and the octocode/E8 construction of the leech lattice [24]. If it were nearly sufficient in terms of density to simply concatenate parts of the E8 lattice, then far more efficient decoders would be possible. This however is not the case, and likely will not be. Therefor the conclusion of this result is that we are unable to prove or disprove the suggestion of Jegou [13] to use the E8 lattice instead. At best we can use our decoding complexity and their decoding complexity for three independent E8 decoders as having only a slightly higher count.

Of interest outside of this thesis would be to find a probabilistic relation based on \( k \), the number of random permutations needed for a concatenation of sub-projections to achieve the same decoding probability rate of the more computationally complex decoding algorithm. An upper bound on this comparison would be the value \( k = \binom{p}{s} \) where \( s < p \) is the ratio of dimensions between the candidate LSH functions and \( k \) is the maximum number of different permutations of vectors. Further insight may be drawn from E8 based constructions of the Leech Lattice [11]. This however is beyond the scope of this thesis and is left for future research.
## 7.2. DETAILED CONCLUSIONS

### BW-32 and larger

In this thesis we do not directly compare larger lattices with our chosen lattice $\Lambda_{24}$. Instead we offer a heuristic for what makes an error correcting code useful in regards to LSH functions for KNN.

Without rigorous proof we will consider the error correcting gain over the AWGN channel of an error correcting code, as a metric of LSH quality. The intuition behind this claim is that the goal of EC gain comparison is to minimize energy per transmitted symbol. First we consider a correlation between symbols and vectors of a dataspace sub-projection. In symbolic decoding, a symbol is completely unrelated to all other symbols, from a metric space context, this implies orthogonality. The next attribute is energy minimization in regard to noise over the channel. This can be seen as, very loosely speaking, the discerning factor between discrete characters. However, despite our informal connection between dB/No and LSH quality, because LSH quality itself is also a not rigorously defined term, this metric offers a reasonable heuristic for judging LSH quality of coding schemes against other similar coding schemes.

Another consideration for LSH quality as addressed in the previous section, is decoding complexity. An LSH function is only useful, if it has a reasonable decoding complexity in regards to defining Vorinio regions for that space. It is known that decoding complexity for known lattices larger than $\mathbb{R}^{24}$, such as the Barnes-Wall lattice of dimensionality 32 increases exponentially with respect to $n$ the dimensionality of the lattice [19]. Using these correlations, we will suggest that larger lattices offer diminishing returns at the cost of exponential increases in decoding complexity.

Providing a mathematical correlation between what defines a ‘good’ LSH function in terms of $\rho$ value and $\gamma_{eff}$ would be of great interest, however here we will settle for similarity between goals. The relation we will suggest is that of selectivity as a good characteristic of an LSH function and a well known limit for information transmission over noisy channels. We will begin with two fundamental definitions from communication theory.

**Definition 7.2.1** (Signal to Noise Ratio (SNR)). SNR is the ratio of expected signal to background noise.

**Definition 7.2.2** (Coding Gain). Coding gain is the difference between the SNR ratio of an unencoded signal to that of the SNR of a coded signal.

Using Coding Gain as our metric, and SNR for our factor of selectivity of an LSH function, we can
begin to formulate a relation between coding gain and what makes an LSH function ‘good’. In this case it is sufficient to evaluate ‘good’ as being directly correlated with the factor $\rho$ from our query complexity analysis for LSH-KNN. It is well known that the limit to coding gain is bounded by the Shannon Limit [34]. This limit is approximately 9dB and no lattice codes in practice approach this limit. This is due to the differences, or gaps in coding gain known as the shaping gap and effective verse nominal coding gain gap. The shaping gap is not of importance to our analysis as it mainly pertains to physical systems and power efficiency. We will simply state that it accounts for approximately 1.53dB [3] of coding gain, and can be obtained as a limit for $d$ dimensions as $d$ approaches infinity, or

$$
\lim_{n \to \infty} \gamma_s(n - sphere) = \frac{\pi e}{6} = 1.53 dB
$$

The more important gap to consider is that between nominal and effective coding gain, because in practice it is effective coding gain that defines the achievable coding gain for coding schemes residing in more than 2 dimensions (recall the leech lattice is in 24). In regards to sphere packing codes, the effectiveness of coding gain is decreases as a function of the kissing number, or number of near neighbors($K_{min}(\Lambda_n)$). As $K_{min}(\Lambda_n)$ increases, the effective coding gain with regard to nominal coding gain decreases. Below is a The table below shows some lattice codes and their coding gains along with their near neighbors counts and nominal coding gains. Most of this data was acquired from David Forney’s “Modulation and Coding for Linear Gaussian Channels” [35].

Using the above table, and the Shannon Limit, it is clear that increasing the dimensionality of a lattice will not likely result in better LSH functions. Furthermore, again citing Tarokh [36], the decoding complexity of
large lattices will quickly overcome the increase in coding gain, and the decrease in $\rho$. Or more explicitly as the value of $\rho$ may decrease, the constant decoding complexity will no longer be trivial.

7.3 Suggestions for Future Work

7.3.1 Other Applications of Parallel LSH

As suggested above, $c$-approximate KNN is not bounded by lattice decoding complexity, and for this reason it would be important to consider alternative applications. One such application considered by not implemented, is for Adaptive Mean-Shift Clustering algorithms. The main iteration of the adaptive mean shift’s convergence to its stable means is a requirement to constantly find nearby points to recalculate the gradient of the stable mean. In a paper [37] on using basic LSH functions for estimating the point’s local neighborhood, the LSH function is used to estimate the local neighborhood of points. This iterated process seems like it would keep the LSH calls busy, while the rest of the system is free to iterate the gradient.

Another option for avoiding the poor parallel speedup would be to simply put more of the LSH function in the GPGPU. One option would be to allow the gpu to compute the exact nearest neighbors from the approximate LSH nearest neighbors in parallel. Again, aside from data transfer, this process is open to naive parallelization. Another option more closely related to the work here, would be to compute more of the hash family for that query vector in the GPU. In this thesis we only compute lattice decodings, however the actual hash used for a vector consists of multiple lattice decodings concatenated and then hashed again in a general hash. This portion of the algorithm is somewhat computationally expensive, and could be tasked to the GPU. This would free up the GPU to spend most of its time transferring data to other GPUs, and decodings to persistent storage.

7.3.2 Where Locality Sensitive Hash Functions Really Shine

Although the suggested system for parallel construction and query of a database for a CUDA architecture show significant speed up and scalability, the real benefit of locality sensitive hash functions has yet to be discussed. Unfortunately this project was unable to demonstrate this utility of LSH, an important next step would be the development of a distributed system for nearest neighbors search among a set of heterogeneous
CHAPTER 7. CONCLUSIONS & FUTURE RESEARCH  7.3. SUGGESTIONS FOR FUTURE WORK

systems and data streams. The commonality being the universal LSH function. No matter what the data input, as long as the sub-projections of the embedding vector space can be agreed upon, a system could be built to allow heavily distributed querying.

7.3.3 E8 for CUDA

Although we were able to suggest that LSH algorithms were not bound by the decoding complexity of their LSH functions, we can still consider parallelizing the E8 lattice for use in other applications as suggested above. The $\Lambda_{24}$ provided a better trade-off in terms of decoding complexity versus vector distance efficiency for a locality sensitive hash function in our tests for sequential efficiency, however we did not consider its advantages for parallelization on the CUDA architecture. Our hypothesis is that the E8 lattice with its much more simplistic and lower memory requirement decoding algorithm may lend itself to better core occupancy of the CUDA sequential multiprocessors. Furthermore the algorithm is very applicable to an SIMD style sequence of operations, which had to be countered in our design by precomputing both branches of decision points in our decoding algorithm. Our maximum speedup per GPU was approximately 53x that of a single standard GPU at 3.2 GHZ, the CUDA occupancy was quite low, and might suggest a much greater occupancy and speed up, which when compared to our decoding losses and vector distance efficiency, may prove to be a better parallel algorithm.

7.3.4 Getting a Better Sense of the Data through Variance Testing

Another place that one could improve the current algorithm, is by either performing a rank reduction of the pre-processed dataset to compress the otherwise sparse linear vectors, or to build the random vector projection matrix based on the parameters, in such a way that higher variance vectors or vectors with greater discernment for the data points in the dataset, receive more weight than others. A brief high level overview of this technique is below. In general, data specific techniques such as these tend to offer considerable practical advantage, but no theoretical guarantees the performance an algorithm. For this reason, such a subject would be better suited for knn search for a specific data mining domain.
7.3. SUGGESTIONS FOR FUTURE WORK

New Data
Vector Selection Order
Compute
Lattice Hash
Hashtable of
Datapoint
Labels
select
24
vectors

1. A new data point is read in.
2. The stored descriptor vector selection order is used to select a set of vectors for generating an integer hash
3. A sequence of hashes is created using the vector selection order
4. The hashes are used as keys to the stored hashtable of datapoint labels. The sequence of labels from each round are concatonate with one another.
5. Datapoints in the concatonated set of labels are compared with the new datapoint by comparing their sequences of hash labels.
6. datapoints from the concatonated sequence are sorted by similarity and the top k are returned as nearest neighbors

Figure 7.3.1: KNN High Level System Diagram
Precomputation

1. Data is read from a storage resource
2. Variance for the descriptors of the data is computed
3. Descriptors are randomly selected with preference to high variance descriptors. The selected vector sequence is stored for use in the search phase.
4. Hashes for all datapoints are computed using the selected descriptor vectors
5. A hash table is created for each round of 24 selected vectors. The keys to the table are the unique hashes which correspond to a sequence of datapoint labels with the key as their hash

Figure 7.3.2: KNN High Level System Diagram
Bibliography


