SCALABLE SEARCHES IN HIGH-DIMENSIONAL SPACES: LEVERAGING MULTI- AND MANY-CORE ARCHITECTURES

By: Brian Donnelly

A Dissertation
Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in Informatics and Computing

Northern Arizona University
May 2025

Approved:
Michael Gowanlock, Ph.D., Chair
Wolf-Dieter Otte, Ph.D.
Igor Steinmacher, Ph.D.
Satish Puri, Ph.D.

ABSTRACT

SCALABLE SEARCHES IN HIGH-DIMENSIONAL SPACES: LEVERAGING MULTI- AND MANY-CORE ARCHITECTURES

BRIAN DONNELLY

High-dimensional search problems are fundamental to many domains, including data analysis, cryptography and computer security. As data complexity and volume increase, traditional search methods become inefficient, necessitating novel approaches to optimize performance. This dissertation presents three primary search strategies across two distinct high-dimensional spaces: Euclidean space and Hamming space.

For Euclidean spaces, we introduce Coordinate Oblivious Similarity Search (COSS) and Multi-Space Tree with Incremental Construction (MiSTIC), two indexing techniques designed to mitigate the curse of dimensionality. COSS employs metric-based indexing to accelerate range queries, while MiSTIC integrates coordinate- and metric-based strategies to improve performance across various dataset characteristics. Experimental results demonstrate that these approaches outperform existing state-of-the-art methods in efficiency and scalability.

In the domain of cryptographic key retrieval, we explore Noisy Probabilistic Response-Based Cryptography (npRBC), a method for authenticating devices in high-noise environments using Physical Unclonable Functions (PUFs). We further develop npRBC-GPU, a GPU-accelerated variant that significantly enhances search throughput compared to its CPU counterpart. Additionally, we investigate optimization techniques for rapid seed generation in cryptographic searches, addressing computational bottlenecks in permutation-based key matching.

By leveraging parallel processing on both CPUs and GPUs, this dissertation provides novel methodologies for efficiently navigating high-dimensional search spaces. These contributions have broad implications for fields such as high-performance computing, cybersecurity, and data science, offering scalable approaches to computationally intensive search problems.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Mike Gowanlock, for his guidance and support throughout my time at NAU. His expertise, patience, and encouragement have been instrumental in shaping this work, and I am truly grateful for the opportunity to be part of his lab.

I would also like to sincerely thank my dissertation committee members, Dr. Satish Puri, Dr. Igor Steinmacher and Dr. Wolf-Dieter Otte, for their insightful feedback, support, and contributions to this dissertation. Their guidance has helped refine and strengthen this work in meaningful ways.

Finally, I want to extend my heartfelt appreciation to my fiancée, Felicity, whose unwavering love, patience, and support have been my greatest source of strength. Her belief in me, even during the most challenging moments, has meant everything, and I could not have done this without her.

To all of you, I am forever grateful.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables ix

List of Figures xii

List of Abbreviations xviii

Preface xix

1 Introduction 1

1.1 Motivation . . . 2

1.1.1 The Curse of Dimensionality . . . 2

1.1.2 Leveraging Modern Hardware Platforms . . . 3

1.2 Parallel and GPU Computing . . . 3

1.3 Dissertation Statement . . . 4

1.4 Dissertation Outline . . . 4

2 Background 7

2.1 Euclidean (metric-space) Searches . . . 7

2.1.1 Formally Defining a Metric-Space . . . 8

2.2 Defining Datasets and Points . . . 8

2.3 Similarity Searches . . . 8

2.4 Indexing . . . 9

2.4.1 Grid Indexing . . . 11

2.4.2 Tree Indexing . . . 12

2.4.3 Metric-Based Indexing and Coordinate-Based Indexing . . . 13

2.5 Metric-Based Indexing . . . 14
2.6 Summary of Challenges for Distance Similarity Searches . . . 16
2.7 Searching Hamming Space . . . 16
2.8 Response Based Cryptography (RBC) . . . 18
2.8.1 Challenges of RBC . . . 19
2.9 Complexity and Evaluation . . . 20
2.10 Parallel Computing and GPUs . . . 21

3 Coordinate Oblivious Similarity Search (COSS) 25
3.1 Abstract . . . 25
3.2 Introduction . . . 26
3.3 Problem Statement . . . 29
3.4 Background . . . 30
3.4.1 Motivation: Selectivity and the Curse of Dimensionality . . . 30
3.4.2 Related Work . . . 31
3.4.3 Reference Implementations . . . 33
3.5 Indexing on Distance Spaces for High-Dimensional Data . . . 34
3.5.1 Overview: Indexing by Distance to Points . . . 34
3.5.2 Bin and Index Construction . . . 35
3.5.3 Searching the Index . . . 37
3.5.4 Selecting the Location of Reference Points . . . 40
3.6 GPU Algorithm and Optimizations . . . 42
3.6.1 Algorithm Overview . . . 42
3.6.2 GPU Thread Allocation . . . 43
3.6.3 Batching Scheme . . . 44
3.6.4 Concurrent Execution of Batches . . . 45
3.6.5 Short Circuiting the Distance Calculations . . . 46
3.6.6 Dimensional Ordering . . . 46
3.7 Experimental Evaluation . . . 46
3.7.1 Experimental Methodology . . . 46
3.7.2 Datasets . . . 47

3.7.3 Implementations . . . 48
3.7.4 Impact of COSS Parameters on Performance . . . 50
3.7.5 Comparison to Reference Implementations . . . 51
3.8 Discussion and Conclusions . . . 55

4 Multi-Space Tree with Incremental Construction (MiSTIC) 57
4.1 Abstract . . . 58
4.2 Introduction . . . 58
4.3 Background . . . 62
4.3.1 Problem Statement . . . 62
4.3.2 Index Supported Range Queries . . . 62
4.3.3 State-of-the-art & Reference Implementations . . . 64
4.3.4 Limitations of the State-of-the-Art . . . 65
4.4 MiSTIC . . . 66
4.4.1 Tree Structure . . . 68
4.4.2 Incremental Tree Construction . . . 71
4.4.3 Searching the Tree . . . 74
4.4.4 Other Optimizations . . . 75
4.5 Experimental Evaluation . . . 77
4.5.1 Evaluation Platform . . . 77
4.5.2 Implementation Configurations . . . 77
4.5.3 Selection of the Search Distance ϵ . . . 78
4.5.4 MiSTIC Performance Analysis . . . 79
4.5.5 Comparison to the Reference Implementations . . . 83
4.6 Conclusion . . . 86

5 Parallel Combination Generation on the CPU and GPU for Secure Key Retrieval 88
5.1 Abstract . . . 89
5.2 Introduction . . . 90
5.3 Definitions and Properties . . . 92

5.3.1 Calculating the Binomial Coefficient . . . 92
5.3.2 Combination Generating Algorithm Properties . . . 92
5.4 Background and Related Work . . . 94
5.5 Summary of Evaluated Algorithms . . . 96
5.5.1 Parallelizing the Algorithms . . . 99
5.6 Experimental Evaluation . . . 100
5.6.1 Experimental Methodology . . . 100
5.6.2 Combination Sets Evaluated . . . 101
5.6.3 CPU Results: Raw Throughput Workload (W_0) . . . 102
5.6.4 GPU Preprocessing Overhead . . . 105
5.6.5 GPU Results: Raw Throughput Workload (W_0) . . . 105
5.6.6 Comparison of CPU and GPU: Raw Throughput Workload (W_0) . . . 106
5.6.7 Examination of Individual Parameters in CS-2: Raw Throughput Workload (W_0) . . . 107
5.6.8 Case Study using Key Retrieval on the GPU: Workloads W_1 and W_2 108
5.6.9 GPU Profiling Results . . . 109
5.6.10 Discussion: Optimized Version of Algorithm 515 . . . 111
5.7 Conclusions . . . 112

6 Authentication in High Noise Environments using PUF-Based Parallel Probabilistic Searches 114

6.1 Abstract . . . 114
6.2 Introduction . . . 115
6.3 npRBC . . . 118
6.3.1 Probabilistic Searches of the PUF Seed Space . . . 122
6.3.2 Noisy Transmission . . . 123
6.3.3 Match Criteria . . . 124
6.3.4 Termination Criteria . . . 124
6.3.5 Candidate Seed Refinement . . . 124
6.4 Experimental Evaluation . . . 125
6.4.1 Experimental Methodology . . . 125

6.4.2 Experimental Results 126
6.5 Discussion & Conclusion 129

7 GPU-Accelerated Authentication in High Noise Environments 130
7.1 Abstract 130
7.2 Introduction & Background 131
7.2.1 Response-Based Cryptography (RBC) 131
7.2.2 Drawbacks and Opportunities 133
7.3 npRBC-GPU: Accelerating Probabilistic Response Based Cryptography . . 134
7.3.1 Hashing with SHA3-512 135
7.3.2 Encryption with AES256 135
7.3.3 Fast Seed Permutation 136
7.3.4 Monolithic to Fine-grained Workloads & Kernels 136
7.3.5 Concurrent Streams for Imbalanced Workloads 137
7.4 Experimental Evaluation 137
7.4.1 Experimental Methodology 137
7.4.2 Implementation Configurations 138
7.4.3 Determining PUF & Transmission Noise Limits 139
7.4.4 GPU Search Success Rate as a Function of PUF and Transmission Error 141
7.4.5 GPU vs. CPU Seed Search Throughput 142
7.4.6 Comparison of GPU and CPU Success Rates 144
7.5 Conclusion 145

8 Conclusion 147

Bibliography 149

A Publications 166

List of Tables

3.1 Datasets used in the evaluation. 47

3.2 Average speedup of COSS over GPU-JOIN and SUPER-EGO across all values of ϵ in Section 3.7.5. 55

4.1 The five real-world datasets (normalized to the range $[0, 1]$) that we use in our evaluation where the data dimensionality n, intrinsic dimensionality i (rounded) and dataset size $|D|$ are shown along with the values of ϵ that correspond to the three target selectivity values. 78

4.2 The partitioning strategy (1-3) dynamically selected for each layer of MISTIC as described in Section 4.4.1.1 for the five real-world datasets and each selectivity level. 79

4.3 Speedup of using the STDDEV heuristic over the SUMSQRS heuristic, where the speedup is the total response time of SUMSQRS over STDDEV. 80

4.4 The average speedup across selectivity levels of performing tree traversals searches over binary searches for each dataset, where speedup is the ratio of the response time for binary searches over tree traversals. The average number of non-empty partitions is included for analysis. 82

4.5 The average speedup across selectivity levels for MISTIC over the reference implementations, where speedup is the ratio of response time of the reference implementation over MISTIC. 83

5.1 Properties of the eight combination generating algorithms that we evaluate. . 94

5.2 The AMD and Intel platforms used in our experimental evaluation, where the number of physical CPU cores are reported. PlatformA contains the GPU used in the evaluation. 100
5.3 Overview of combination sets (CS) where the maximum combinations column corresponds to the n and k values that yield the greatest number of total combinations. |CS| refers to the number of combination spaces for a set. . . 102
5.4 The mean response time (s) for workload W_0 (raw throughput workload) across all combinations in CS-1 and CS-2 for each algorithm as executed on the two multi-core CPU and GPU platforms. The lowest response times are highlighted in bold face. 102
5.5 Scalability of multi-core CPU and GPU algorithms on PlatformA and PlatformB. The parallel speedup (s) and parallel efficiency (e) of the CPU algorithms is shown using $p = 64$ threads on PlatformA and PlatformB; we omit showing T_1 due to space constraints, but it can be calculated using $T_1 = s \cdot T_p$. The response time of the GPU algorithms (T_{GPU}) and speedup over the CPUs in PlatformA and PlatformB are shown. 103
5.6 The mean response time (s) across all combinations in CS-2 for each algorithm as executed on the GPU with varying workloads. The lowest mean times in each column are highlighted in bold face. Note that there are several identical mean times, or times that are very close to each other and have insignificant differences in performance. 110
5.7 Profiler results using Nvidia Nsight Compute for $n = 512, k = 4$ on the A100 GPU. The computational throughput, memory throughput, L1 and L2 Cache hit rates are reported as a percentage where the maximum for each is 100%. 110

6.1 Parameter values used in the evaluation. Varied refers to whether the parameter is varied in the evaluation. 125

6.2 Average execution times and authentication rates for PUF noise levels $n = 20 - 45$, TBER levels 10 – 40% and time limit of 5s. 125

7.1 Parameter values and notation (Not.), where we outline if the parameter is varied or static. 137

List of Figures

1.1 An overview of the search strategies and how they relate. COSS and MiSTIC address range queries in high-dimensional Euclidean space. npRBC is our proposed solution to searching for a correct key in a noisy environment using a probabilistic search through a high-dimensional hamming space. Rapid Seed Generation is a survey of combination generating algorithms evaluated for the GPU and CPU with workloads similar to those in npRBC-CPU and npRBC-GPU (a GPU based version of npRBC-CPU). 5

2.1 (a) An example of a metric-based index (similar to COSS [35]) partitioning a two-dimensional space with two reference points R_1, and R_2. (b) An example of partitioning with a grid (similar to GDS-Join [48]) in two dimensions where each of the dimensions are used for indexing. Both methods use the triangle inequality to exclude points in non-adjacent ϵ-width partitions from the search. 13

2.2 This figure from Hjaltson and Samet [55] demonstrates two types of distance partitioning: (a) a ball partitioning method and (b) a hyperplane partitioning method. 14

2.3 This figure is an example cube (a Hamming space with 3 dimensions) with a path between vertices $(0, 0, 0)$ and $(1, 1, 1)$. The distance between the two vertices is 3, which corresponds to the shortest number of edges between the two vertices. 17

3.1 ϵ vs. n showing the minimum value of ϵ required to find a single neighbor (on average) within a unit n-dimensional hypercube, $\beta = [0, 1]^n$, where the $|D| = 10^6$ data points are uniformly distributed. When $\epsilon \geq 0.5$ (red dashed line), a grid-based index degrades to a brute-force search. 28
3.2 This figure shows a graphical example of an index with a single reference point, RP. The number line in the bottom of the image shows where points fall into ϵ-width bins based on their distance to RP. 36
3.3 Example showing the construction of B and Q for two reference points. Every additional reference point adds an additional construction step. Note that the distance to a reference point within a bin does not impact the sorting, but the stable sort maintains the ordering from previous steps within a bin. . . . 38
3.4 Grid indexing example, where Q is the point array, A is the lookup array, B' is the unique bin array, C is the range array, and D' is the sorted data array. For clarity, only one reference point is shown. When there are multiple reference points, B' will be a multidimensional array, constructed as described in Section 3.5.2. 39
3.5 (a) shows the pattern generated with the RP-Inner placement strategy. (b) shows the pattern generated by the RP-Outer placement strategy. 41
3.6 We compare the runtimes for RP-Inner and RP-Outer reference point placement strategies with *MSD* ($n = 90, \epsilon = 0.007$), *SuSy* ($n = 18, \epsilon = 0.015$), *Uniform* ($n = 10, \epsilon = 0.35$), *Expo* ($n = 16, \epsilon = 0.04$). 41
3.7 Runtime vs. the number of threads (t) on the *MSD* dataset ($n = 90, S = 4 - 1892$) shows that the while a small number of threads per a point has significantly longer runtime, $t = 5 - 10$ achieves good performance. 45
3.8 Index binary search time (s) vs. number of reference points. 49
3.9 Fraction of distance calculations vs. number of reference points. 51
3.10 Runtime (s) vs. number of reference points. 52

3.11 Runtime (s) vs. ϵ on real-world datasets. Comparing COSS, GPU-JOIN, and SUPER-EGO. 53
3.12 Runtime (s) vs. ϵ on synthetic datasets. Comparing COSS, GPU-JOIN, and SUPER-EGO. 55

4.1 (a) An example of a metric-based index (similar to COSS [35]) partitioning a two-dimensional space with two reference points R_1, and R_2. (b) An example of partitioning with a grid (similar to GDS-JOIN [48]) in two dimensions where each of the dimensions are used for indexing. Both methods use the triangle inequality to exclude points in non-adjacent ϵ-width partitions from the search. 63
4.2 Tree indexing example, where L is the tree structure with r layers/reference points. b_x is the maximum partition range in that layer, b'_x is the number of non-empty partitions where x is the layer number. P is the point array and $|D|$ is the number of points in the dataset. 68
4.3 The fraction of the total response time spent constructing the tree for each of the selectivity values S_s, S_m, and S_l across the five real-world datasets for $r = 6$. 79
4.4 The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$. 80
4.5 Response time as a function of the three selectivity values across the five real-world datasets for each reference implementation. 82
4.6 The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$. 82
4.7 Standard deviation of the number of points in each partition vs. selectivity values for the five real-world datasets. 83

5.1 The response time (T_{GPU}) in milliseconds for W_0 showing all n and k combinations in CS-2 for PNXCB on the GPU. 107

5.2 The response time (T_{GPU}) in milliseconds for W_0 showing all n and k combinations in CS-2 for ALGORITHM 515 (TABLE) on the GPU. 108

6.1 The response-based cryptography protocol that is robust to noise (npRBC). The RBC search engine is shown in Figure 6.2. This inspiration for this figure is from similar figures in the literature [73, 74]. 118

6.2 The probabilistic RBC search engine. Colors represent comparisons between client and server information. 119

6.3 This histogram shows the probability of a bit flipping for the PUF used in our experimental evaluation, which is an ISSI 61-64WV6416BLL SRAM chip. There are a total of 768 enrolled cells that are stored in the PUF image on the secure server. While 573 of those cells are stable and will not change between authentication sessions, the other 185 cells have up to a 0.5 (or 50%) probability of their state changing when challenged during authentication. . 121

6.4 This heatmap plots the number of samples at each drift value for each PUF noise (n) using the PUF outlined in Figure 6.3. There are 1,000 trials for each n value with each trial selecting unstable cells at random from the PUF. The intensity of the heat map is the number of trials for each selected n value that had the corresponding amount of drift. 121

6.5 The heatmap plots the success rate of authentication for each PUF bit error rate (n) and drift with a TBER of 30%. Blank values in the heatmap indicate that there were no trials that had that combination of n and drift (see Figure 6.4). A success rate of 1.00 indicates that it always succeeded to authenticate with the given parameters, while a success rate of 0.00 indicates that it never succeeded to authenticate. 127

6.6 The fraction of successful authentications are plotted vs. the TBER (the chance of each transmitted bit flipping in the message digest or ciphertext) for a PUF noise level (n) of 30%. 128

7.1 Probabilistic RBC search engine executed on the GPU used to authenticate Client 1. The major task for the search engine is hashing a seed using SHA3 to create a message digest (M) and comparing it to the client's message digest (M_1') and if these are equivalent, a second check is performed using AES256. Client 1's encrypted payload that was received by the server will vary due to drift in the client's PUF and/or due to transmission error. Tasks computed on the GPU are outlined by the green lines. Terminating the search using a $T = 5$ s time limit occurs on both the host and GPU. 132

7.2 Limitations of mean search success rates between prior work (npRBC-CPU) and our work, npRBC-GPU. There are a total of 36k trials shown, see text for details. (a) Mean success rate as a function of n. (b) Mean success rate as a function of m (%), where the theoretical limit is $m = 50\%$. 140

7.3 npRBC-GPU mean search success rate where we select $n \in \{20, 25, \ldots, 40\}$ and $m \in \{10, 20, \ldots, 40\}$. There are a total of 20 parameter combinations of n and m where each has 1,000 trials yielding a total of 20k trials. Each trial randomly samples PUF cells where there are n bits with a non-zero probability of flipping and $m\%$ of the bits are randomly selected to flip based on transmission error. (a) Search success rate as a function of PUF noise level (n) and the resulting PUF drift (d). (b) Search success rate as a function of TBER (m) and PUF drift (d). 141

7.4 Seed search throughput (average number of seeds searched per second) plotted on a log scale for the experiment conducted in Section 7.4.4. The GPU curve shows search throughput vs. the number of CUDA streams. npRBC-CPU with 64 CPU threads/cores is shown for comparison. 143

7.5 The same as that shown Figure 7.3, except that we show the difference in successful search rates between npRBC-CPU and npRBC-GPU. Positive values indicate that npRBC-GPU has a higher success rate than npRBC-CPU and negative values indicate the opposite. 145

List of Abbreviations

Acronym	Name
EGO	Epsilon Grid Order
BKT	Burkhard-Keller Tree
vp-tree	Vantage Point Tree
mvp-tree	Multi-Vantage Point Tree
m-tree	Metric Tree
gh-tree	Generalized Hyperplane Tree
GNAT	Geometric Near-Neighbor Access Tree
COSS	Coordinate Oblivious Similarity Searches
AESA	Approximating and Eliminating Search Algorithm
FQT	Fixed Queries Tree
FHFQT	Fixed Height Fixed Query Tree
FQA	Fixed Queries Array
FSFQA	Fixed Slices Fixed Query Array
FQFQA	Fixed Quantile Fixed Queries Array
LC	List of Clusters
LSC	List of Super Clusters
PCA	Principle Component Analysis
FFT	Farthest First Traveled
k-nn	k-Nearest Neighbors
MiSTIC	Multi-Space Tree with Incremental Construction
RBC	Response-Based Cryptography
AES	Advanced Encryption System
npRBC	Noisy-Probabilistic Response-Based Crytpography
SHA	Secure Hashing Algorithm
BER	Bit-Error Rate
PQC	Post-Quantum Cryptography

PREFACE

In this dissertation we include several peer reviewed papers which are summarized in Chapters 3,4, 5 and 6. Additionally, the contents of Chapter 7 summarize a paper which is currently under review for publication.

Chapter 1

Introduction

Searching is a canonical problem in computer science with searches varying in complexity from a scan through a dataset, to matching a user name with a password, to fuzzy matching DNA sequences that represent genes shared across species. Most researchers use well established methods for searching but as data grows in both volume and complexity those methods are becoming intractable [14]. One of the major hurdles for these pioneering methods to overcome is the increase in the number of dimensions of modern data [14]. In this work we discuss methods for addressing this increase in dimensionality and the subsequent increase in the search space. As the search space increases we observe an exponential increase in the amount of work needed to search the spaces using naïve approaches. To address this we examine strategies which partition the spaces and search in a logical manner to reduce unnecessary work.

In this dissertation, we introduce three main search strategies across two distinct types of search spaces. The first space, Euclidean distance space, is commonly used in the context of data analysis. For example, particle physics data gathered from the Large Hadron Collider which can be used to detect specific collision events [6]. The second space, permutation space, examines searches that can have near infinite work (we will show an example with a search space larger than 2^{256} items) which occurs often in cryptography and communications [22, 24, 74].

The first search strategy that we introduce in this dissertation is the Coordinate Obliv-

ious Similarity Search (COSS) which is an index designed for searching for data points in high-dimensional Euclidean spaces. The second method we introduce is Multi-Space Tree with Incremental Construction (MiSTIC) which addresses the same scenarios as COSS but utilizes a hybrid metric- and coordinate-based indexing strategy. The third search strategy that we introduce is Noisy Probabilistic Response based Cryptography (npRBC-CPU) which searches a Hamming space to match keys for authentication in the field of computer security. All of these strategies partition the search space to allow for an index (a type of data structure used to organize data for searching) to be constructed and then searched.

1.1 Motivation

As data continues to grow in both volume and complexity, existing search methods are becoming increasingly ineffective, particularly in high-dimensional spaces which contain ≥ 16 dimensions. These search problems arise in numerous domains, including data analysis, cybersecurity and cryptography, where the ability to quickly perfrom a search is critical. However, as the number of dimensions increases, search spaces expand exponentially, making naive or brute-force approaches computationally intractable. This phenomenon, often referred to as the "curse of dimensionality," [14] requires the development of specialized algorithms that can effectively navigate and process such high-dimensional data spaces.

1.1.1 The Curse of Dimensionality

As the number of dimensions increases, computational and analytical tasks—such as searching, clustering, and classification—become exponentially more difficult. This challenge has been coined as "the curse of dimensionality" and can be broken down into 3 main issues as follows:

Exponential Growth of Search Space: In low-dimensional spaces, data points are relatively close to one another. However, as dimensionality increases, the volume of the search space grows exponentially, causing data points to become more sparse. This makes it

difficult to efficiently partition and search the space. For example, if a dataset has 10 points uniformly distributed in a 2-dimensional space, they might be relatively close together. But in a 100-dimensional space, those same 10 points could be extremely far apart, making search operations computationally expensive.

Loss of Meaningful Distance Metrics: Many search algorithms rely on distance metrics (such as Euclidean distance) to compare data points. However, in high dimensions, distances between points tend to become nearly uniform, reducing the effectiveness of these metrics.

Increased Computational and Memory Costs: Storing and processing high-dimensional data requires significantly more memory and computational power. Algorithms that work well in low dimensions (e.g., k-d trees [11, 15]) often become inefficient or break down entirely in high dimensions because they rely on pruning strategies that lose effectiveness as dimensions increase.

1.1.2 Leveraging Modern Hardware Platforms

High-dimensional searches demand increasing computational capabilities with the dimensionality of the data. To address this, parallel processing techniques and hardware accelerators such as multi-core CPUs and GPUs are utulized. In particular, GPUs, which are now integral to modern high-performance computing, provide significant advantages in accelerating search algorithms. However, leveraging these resources efficiently requires tailored algorithms that exploit parallelism while minimizing overhead associated with the increased complexity of the hardware platform.

1.2 Parallel and GPU Computing

High-dimensional searches require so much work that sequential algorithms are intractable for most searches. Therefore, in this dissertation, we examine parallel multi-core CPU and GPU search algorithms. While parallel search algorithms on the CPU are similar to sequential ones, the GPU requires a different approach. As of 2025, 9 of the 10 top supercomputers

use GPUs, highlighting their importance in high performance computing.

1.3 Dissertation Statement

Searching in high-dimensional spaces presents significant challenges due to the complexity and scale of the search space. Traditional search methods often become inefficient as the dimensionality of the search space increases, making it necessary to develop specialized algorithms. Given the immensity of high-dimensional search spaces, additional computational resources, such as multi-core CPUs or GPUs, are required to process and analyze data effectively. To maximize efficiency and performance, it is crucial to design algorithms that fully leverage these parallel architectures, ensuring that searching algorithms remain feasible and scalable even as the size and dimensionality of the data increases.

1.4 Dissertation Outline

The outline for my dissertation is given in Figure 1.1. My dissertation focuses on searching in high-dimensional spaces. This is broken into two main categories, Euclidean Space and Hamming Space. There are two common ways of indexing in Euclidean space, the first is metric-based indexing where the data in the Euclidean space is mapped to a distance-space by replacing the coordinate values of each point in the dataset with the distance to another point in the original Euclidean space. The points are then partitioned in the distance-space that was created. The second method for indexing is to partition the dataset based on the original coordinate values in the Euclidean space. We propose a Coordinate-Oblivious Similarity Search (COSS), a metric-based indexing method for range queries and is designed for the GPU. We show that COSS out-performs other state-of-the-art methods on high-dimensional range queries. We improve on COSS by introducing the Multi-Space Tree with Incremental Construction (MiSTIC). MiSTIC combines both coordinate- and metric-based indexing strategies to partition the Euclidean space. MiSTIC outperforms

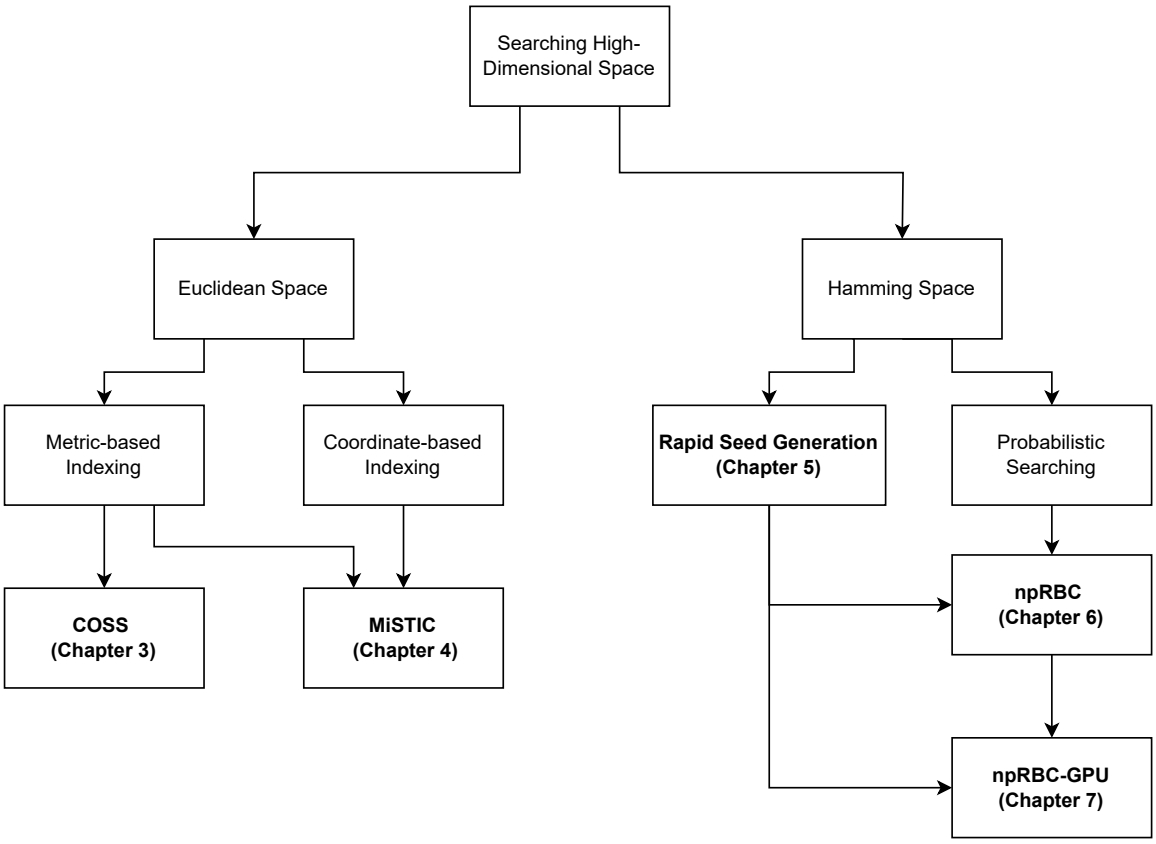


Figure 1.1: An overview of the search strategies and how they relate. COSS and MiSTIC address range queries in high-dimensional Euclidean space. npRBC is our proposed solution to searching for a correct key in a noisy environment using a probabilistic search through a high-dimensional hamming space. Rapid Seed Generation is a survey of combination generating algorithms evaluated for the GPU and CPU with workloads similar to those in npRBC-CPU and npRBC-GPU (a GPU based version of npRBC-CPU).

other methods across a wide range of dataset characteristics, highlighting its robustness.

This dissertation also addresses the problem of searching in Hamming space. Hamming space introduces unique problems compared to Euclidean distance space because of the need for combination generating algorithms to examine the search space. These combination generating methods are used for our proposed protocol Noisy-Probabilistic Response-Based Cryptography (npRBC-CPU).

This dissertation is organized as follows: Chapter 2 gives a general background to search-

ing in high-dimensional spaces, Chapter 3 introduces COSS for similarity searches, Chapter 4 builds on the work in the previous chapter and introduces MiSTIC, Chapter 5 gives a survey of combination generating methods, Chapter 6 introduces npRBC-CPU for authentication in high noise environments, Chapter 7 extends the work of the previous chapter with npRBC-GPU which uses GPGPUs to accelerate authentication and finally this dissertation is concluded in Chapter 8.

Chapter 2

Background

This section presents a brief background of searching in Euclidean Spaces as well as searching for cryptographic keys using Response Based Cryptography (RBC). Additionally, we discuss the indexing and search algorithms as well as the metrics we use to evaluate these algorithms.

2.1 Euclidean (metric-space) Searches

In recent years datasets have been growing both in size and in the scope of the data they contain. It is common for data records to store hundreds or thousands of attributes or dimensions. This rise in both dimensionality and scale has motivated a number of works that reduce computation times on such datasets [9, 14]. One such branch of research is metric-space indexes which reduce the time needed for data analysis. These indexes are designed for use on high-dimensional data where more commonly used indexes perform poorly. Metric-based indexes use a spatial projection technique, usually by reorganizing the data based on a distance to a point. This allows for the data to be indexed in an analogous lower dimensional space as compared to the original data [55, 79, 80]. Metric-based indexes only work in a metric-space (Euclidean is the most common metric-space). The formal definition of a metric-space where metric-based indexing works is given in the following section.

2.1.1 Formally Defining a Metric-Space

The distance in a metric space between two points x and y, $d(x, y)$, is defined by: $d(x, y) = (\sum_{i=1}^{n} |x_i - y_i|^k)^{1/k}$, where x and y are two points and n is the number of dimensions of the two points. k changes depending on the Minkowski distance that is being used [14]. For Euclidean distances, $k = 2$.

A metric space is defined by four characteristics [14, 55, 79]:

- Symmetry: $d(x, y) = d(y, x)$
- Positivity: $d(x, y) \geq 0$
- Indiscernibility: $d(x, y) = 0$ if $x = y$
- Triangle Inequality: $d(x, y) + d(y, z) \geq d(x, z)$

2.2 Defining Datasets and Points

Let us define, D, as a dataset containing $|D|$ points (or feature vectors), in n dimensions. Each point is defined as $p_j \in D$, where $j = 1, 2, \ldots, |D|$. We denote the coordinates of each point, $p_j \in D$, as $p_j = (x_1, x_2, \ldots, x_n)$. The Euclidean distance between points $r \in D$ and $s \in D$ is defined as $dist(r, s) = \sqrt{\sum_{i=1}^{n} (r_i - s_i)^2}$.

2.3 Similarity Searches

Similarity searches are common dataset operations that are computationally expensive [3, 12, 25, 77, 78, 92, 96, 113, 115]. A similarity search finds all data points D within a distance threshold, ϵ, of a data point x. In a self-join operation all of the data points in D are compared to all of the other data points to see if the distance between the points is less than ϵ, $d(x, y) \leq \epsilon$, the operation is denoted as $D \bowtie_{\epsilon} D$. In a semi-join operation there are two datasets P and Q, the points in P are compared to every point in Q to find the pairs of

points that are closer together than ϵ, this operation is denoted as $P \ltimes_\epsilon Q$. The selectivity of a similarity search is the average number of neighbors within ϵ for each point and is defined as $S = (|R| - |D|)/|D|$ where R is the result set and $|R|$ is the total size of the result set, which contains result set pairs of points that are within ϵ of each other.

A popular method for computing a similarity search is to generate a set of candidate points for each query point and then compute the distances from the query point to each candidate point; a search and refine approach. The candidate set of points is found during the search step and contains points that tend to be near to the query point but are not guaranteed to be within ϵ. The refine step is where the candidate points are compared directly to the query point using distance calculations and this is often where the majority of the computation of the similarity search is performed. Methods that reduce the size of the candidate set will significantly reduce the overall computation.

There are a number of ways to measure distance and some indexes are able to work with different types of distance metrics. Most indexes are restricted to a space that upholds the metric-space definitions as defined in Section 2.1.1. One of the most common distance metrics to use is the Euclidean distance which works in Euclidean spaces. In lower dimensions, the distance calculation takes a relatively little amount of time to compute, whereas in higher dimensions (≥ 16) [14, 17, 87] the calculation can require a significant portion of the computation time.

2.4 Indexing

A basic implementation of a similarity search is simple but has a runtime complexity for a semi-join operation of $O(n \cdot |P| \cdot |Q|)$ where n is the dimensionality of the data and $|P|$ is the number of points in a dataset P which is querying a $|Q|$ points form a dataset $|Q|$. A self-join operation has a runtime complexity of $O(n \cdot |D|^2)$ where D is the dataset and n is the dimensionality of the data. For larger datasets this becomes an intractable problem regardless of the hardware used and even datasets of moderate size can become intractable to

compute with if they are high-dimensional [14]. To make similarity searches faster, indexing methods have been developed to reduce the number of distance calculations needed at the cost of preprocessing overhead [19]. These methods prune the search space so that points only need to calculate the distance to a subset of other points that are nearby in the data space. This has allowed similarity searches to be performed on larger datasets and reduced the computational requirements in effectively all cases [79, 80].

There are a number of ways to index a dataset to allow for pruning. There are two main categories that most indexes fall into; trees and grids [14, 35, 55]. Tree indexes create a hierarchical structure where nodes on the tree are broken down into more subsequent nodes. The nodes are divided in such a way as to partition either the data space or the set of data points.

A grid index is created by sectioning the data in geometric patterns, usually linearly, that separates the data into regions [48, 63, 64]. A grid search will add the points in regions adjacent to the query point's region to the candidate set.

Most algorithms directly index on the data point values, but a large subset of methods, metric-based indexes, use one or more points in the data space to index the data [15, 55, 79, 80]. These methods tend to have better performance in higher dimensional spaces because they are able to encapsulate more information in less space. This allows the algorithms to more effectively prune the space where other tree or grid methods may prove ineffective. Metric-based indexes can use other indexing structures on what is essentially a transformed space [14]. The pruning happens in this transformed space but the actual distance calculations still need to occur in the original data space. While metric-indexes have proven to be effective in certain spaces their additional complexity can lead to higher overheads that may make them less attractive in many use cases.

Metric-based algorithms are also commonly used to combat "The Curse of Dimensionality" [9]. This is because most indexes have reduced performance when the search distance ϵ is high. In high dimensions the distance threshold needed for a given selectivity is usually

(for a given data distribution) higher than in lower dimensionality. As the distance threshold begins to approach approximately half of the range in a given dimension, pruning in that dimension is not possible [35]. Metric-based methods can avoid this issue by indexing on every dimension of the data together. This allows them to continue to prune the candidate sets for each point as the distance threshold increases.

We describe some of the advantages and challenges of partitioning the space with both grids and trees in what follows.

2.4.1 Grid Indexing

Grid-based indexes overlay a grid on the data that partitions the space evenly. One example is the Epsilon Grid Order (EGO) that creates grid cells of length ϵ (the distance threshold in a similarity search) [15, 63, 64]. To efficiently partition the space, the algorithm selects some number of dimensions to index on. The dimensions are chosen at runtime based on their variance. This allows the EGO methods to maximize pruning while having low overhead. Each cell generated by the index contains a set of points P_c. All points in P_c will have to check against all other points in the adjacent cells, but not to points in cells that are not adjacent. This is because the cell widths are ϵ, so any points in non-adjacent cells will have a minimum distance of ϵ.

Each cell in a grid index will have 3^k adjacent cells where k is the number of indexed dimensions. Only the non-empty cells are saved in the grid index to reduce the memory storage requirements which would become intractable in high-dimensional spaces. Because only the non-empty cells are retained in the index for a cell to find another adjacent cell, a search will need to be performed. This is often achieved with a binary search that has a runtime complexity of $O(\log_2(|g|))$ where g is the number of non-empty cells. Each cell will have to perform 3^k searches to find all adjacent cells.

There is a trade-off between increasing the number of indexed dimensions, k, and the number of searches that must be performed. When k is low the time spent searching is

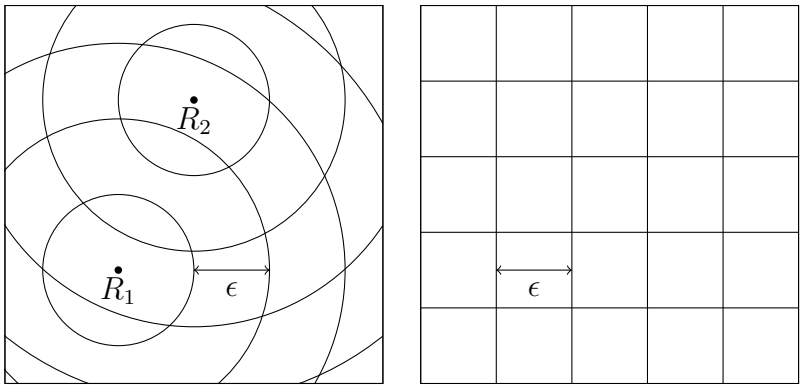
negligible but the index will not prune as many distance calculations resulting in more overall computation. If k is too high the time spent searching the index will reduce the overall performance of the index. Generally, the best value for k has been experimentally found to be around $k = 6$ for real world datasets [35, 48].

2.4.2 Tree Indexing

Trees are similar to grids in that they partition the space to allow for pruning searches of points nearby a query point. One of the main differences between a tree and a grid is how they identify adjacent cells. Instead of the searches that most grids employ, trees have tree traversals that will locate adjacent nodes which will contain the candidate points [12, 57, 109, 115]. As an example, consider a depth first search which starts at the root and, based on the value of the point being evaluated, traverses through the tree. Each layer of the tree is somewhat synonymous with each indexed dimension of the grid described above. The farther down the tree, the more refined the candidate set becomes.

In many cases, trees are able to traverse faster and more efficiently than a search could be performed on a grid index. Trees can have a breadth first traversal in which the tree traversal searches each node of the layer that the point may have candidates in, this approach reduces the amount of checking because a single full traversal will generate the entire candidate set, but increases the amount of memory that will be needed to hold all of the nodes of the tree simultaneously [65]. A depth first traversal starts at the root node and traverses to the leaf node. A depth first search will be more computationally expensive but require less memory resources. Hybrid approaches will combine a partial breadth first search that then spawns depth first traversal from nodes partway down the tree. This hybrid approach makes the most of what memory resources are available to reduce the amount of work of the depth first searches.

Regardless of the partitioning method of the tree-based index, too large of a tree will cause a degradation in performance, while too small a tree will not have sufficient pruning.



(a) Metric-Based (b) Coordinate-Based

Figure 2.1: (a) An example of a metric-based index (similar to COSS [35]) partitioning a two-dimensional space with two reference points R_1, and R_2. (b) An example of partitioning with a grid (similar to GDS-Join [48]) in two dimensions where each of the dimensions are used for indexing. Both methods use the triangle inequality to exclude points in non-adjacent ϵ-width partitions from the search.

Once again, there is a selection process that needs to be evaluated to determine how much partitioning there should be based on the distribution and dimensionality of the data. For a tree, it is important to reduce the depth and the number of leaves that need to be queried. Like the grid-based indexes, there is a trade-off between the search overhead and the number of distance calculations that need to be performed.

2.4.3 Metric-Based Indexing and Coordinate-Based Indexing

The grid- and tree-based indexes discussed in the previous two sections describe how to search partitions of the data space. Here we compare the two methods for representing the points that are indexed. The most common way to construct an index is to partition the data based on its coordinate values, as in the case of a kd-tree [11] or R-tree [8]. The other way is to project the data into a new set of dimensions. The most common type of projection is to use the distances from a set of points as the new dimensions of the data. The projection uses a mapping function and for most indexes to work the mapping needs to be contractive such that the distances between data points only decrease from the coordinate space to the distance space [28]. The methods that use this mapping are called metric-based

indexes, while the methods that do not map into a new space are coordinate-based indexes. A comparison of metric-based indexing and coordinate based indexing is shown in Figure 2.1. This figure shows an example of a 2-dimensional space that has been partitioned with either a metric- or coordinate-based approach. The two reference points shown in Figure 2.1 (a) have multiple concentric rings with a distance between each ring equal to the search radius ϵ. The concentric rings partition the space based on the distance to the reference points. The coordinate-based approach shown in Figure 2.1 (b) partitions the space evenly on the vertical and horizontal axis with ϵ spaced grid lines.

Coordinate-based indexing is straight forward and a commonly known approach, however, metric-based indexing is less commonly known and more complex to implement, therefore in the next section we discuss metric-based indexing in depth.

2.5 Metric-Based Indexing

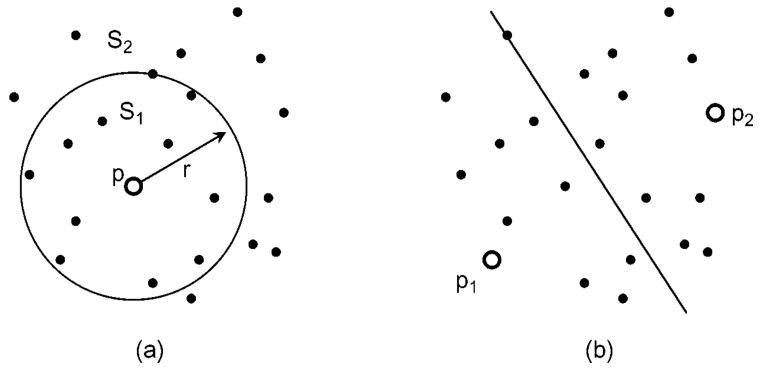


Figure 2.2: This figure from Hjaltson and Samet [55] demonstrates two types of distance partitioning: (a) a ball partitioning method and (b) a hyperplane partitioning method.

There exist two main methods of partitioning a space for a metric-based index. The first is generally referred to as ball or pivot-based indexing, in which the space is divided into spheres based on the distance to some pivot point. A pivot point can be either an arbitrary point in space or a point in the dataset and is used as a point of reference for index construction. Figure 2.2(a) shows how the space is divided into two partitions S_1 and S_2 by

pivot point p. The second is hyperplane partitioning where the space is divided by one or more hyperplanes to create sub groups as in Figure 2.2(b). Hjaltason and Samet [55] classify most metric-based indexes into these two categories. These two types of metric-based indexes can be combined with a number of other methods such as contractive mapping, or spatial projections to create an index. Ball partitioning can also be combined with hyperplane partitioning to create an index that uses both to divide the space [14, 55]. Most of these methods rely on the triangle inequality defined as $d(x, y) + d(y, z) \geq d(x, z)$ where $d()$ is the distance function and x, y, z are points. The triangle inequality is useful because it gives a basis for a relationship between any three points and states that the sum of any two distances between the points must be greater than a single distance, or rather any two sides of a triangle must be greater or equal to the remaining side. This can be used to establish bounds on the distance between two points x and y, if $d(x, z)$ and $d(y, z)$ are already known without having to calculate $d(x, y)$.

The main purpose of metric-based indexing is to establish lower or upper bounds on the distance between points, or both [55]. By bounding the distances, the number of distance calculations can be pruned. In the case of a join operation on a dataset, the establishment of a lower bound on the distance between points can be used to remove any point from the candidate set if the lower bound is greater than the distance threshold, ϵ. While self-joins and semi-joins do not typically make use of an upper bound on the distance, finding the k-nearest neighbors often needs the upper bound in order to ensure that all points that could be a neighbor have been considered.

Some of the earliest metric-based indexing methods were proposed by Burkhard and Keller in 1973 [19]. The proposed methods focused on a single query into a file system. The authors noted that pruning could be done based on a distance to a distinguished feature (a pivot point) by using the triangle inequality, and that this would reduce the number of calculations that would need to be done. The authors proposed the Burkhard-Keller Tree (BKT) which used these principles. Their idea has been extrapolated on by authors in

the years since to solve not only nearest neighbor problems but also joins and other range queries [43].

2.6 Summary of Challenges for Distance Similarity Searches

In this section we briefly review the challenges of distance similarity searches.

Curse of Dimensionality: As dimensionality increases, traditional search methods become inefficient due to the tendency of data points to become equidistant. This phenomenon reduces the effectiveness of indexing techniques.

Indexing Limitations: Indexing with a coordinate-based approach results in poor index performance in high-dimensions due to the reduction in the number of partitions across individual dimensions. Using a metric-based approach requires the placement of reference points which has a substantial impact on performance. Since the optimal reference point placement is dependent on the dataset characteristics there is no overall solution.

Balancing Overhead and Distance Calculations Performance: There is a trade-off between the amount of partitioning that an index performs and the number of distance calculations. The overhead and index search time increases as the partitioning increases, but the number of distance calculations decreases. Balancing the overhead and search time with the amount of distance calculations is also dependent on dataset characteristics, requiring a data aware approach to index construction for the best overall performance.

In this dissertation we seek to address these challenges with the introduction of both COSS and MiSTIC.

2.7 Searching Hamming Space

Hamming space is the set of all possible bits of a fixed length. It is used to measure the distance between sets of bits, where the Hamming distance quantifies the number of differing positions between two sets. We can use a cube to visualize a 3 dimensional Hamming space

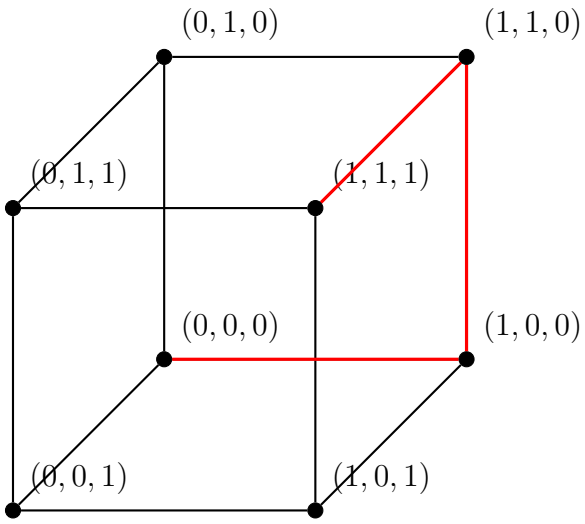


Figure 2.3: This figure is an example cube (a Hamming space with 3 dimensions) with a path between vertices $(0, 0, 0)$ and $(1, 1, 1)$. The distance between the two vertices is 3, which corresponds to the shortest number of edges between the two vertices.

as shown in Figure 2.3. The distance between two points (each point is also a vertex of the cube) in the Hamming space is also the number of edges that need to be traversed from one vertex to another. While a 3-dimensional Hamming space only has 2^3 possible points in its Hamming space, the number of points in the space increases exponentially with the increase in dimensionality which can make searching the high-dimensional spaces intractable.

When searching a Hamming space you start with a set of bits and then find all of the other sets of bits that are within a given Hamming distance, where the Hamming distance is the number of bits in each set which are not the same (i.e. there is a Hamming distance of two between 1001 and 1010). The number of possible sets in a given Hamming space is 2^n where n is the fixed length of the sets of bits. In the context of computer security, the size of the search space is called the *bits of security*, with a Hamming space of 118-dimensions corresponding to *118-bits of security*. As n increases, the search space becomes intractable, when $n \geq 128$ a brute force search is generally considered unfeasible even for the largest supercomputers [5].

Instead of a brute force search we can search increasing the the sets around the query set by permuting the query set. The magnitude of the permuted sets is given by n choose

k, denoted as $\binom{n}{k}$, where k is the Hamming distance from the query set.

As we will discus in the next section, Hamming space searches are critical to Response-based Cryptography (RBC) where bit sets (the keys) are rapidly permuted and hashed to match a transmitted hash generated from another bit set which is some Hamming distance from the one being permuted.

2.8 Response Based Cryptography (RBC)

Response-based Cryptography (RBC) is an innovative approach to securing communications and data, particularly in environments where traditional cryptographic methods may fall short. This method leverages the unique responses of physical unclonable functions (PUFs) to generate cryptographic keys and authenticate devices. PUFs are hardware-based security primitives that exploit the inherent manufacturing variations in electronic components to produce unique, unpredictable responses. These responses can be used as cryptographic keys, providing a robust layer of security that is resistant to cloning and tampering [24].

PUFs are central to the functioning of RBC. They act as hardware fingerprints, providing a unique identifier for each device. The unpredictability and uniqueness of PUF responses make them ideal for cryptographic applications. Unlike traditional cryptographic keys, which are stored in non-volatile memory and can be extracted by attackers, PUF responses are generated on-the-fly and are not stored, making them much harder to compromise [22, 74].

One of the key features of RBC is the introduction of controlled noise into the cryptographic process. This noise makes it difficult for attackers to predict or replicate the cryptographic keys. This is measured as the Hamming distance between the registered PUF fingerprint (often stored on a server) and the response of the PUF to a challenge (the challenge is often generated by the same server which stores the registered PUF's fingerprint). This measured Hamming distance is the drift between the two fingerprints which is a subset of the PUF image called a PUF seed.

RBC has a wide range of applications, particularly in the field of Internet of Things (IoT) and post-quantum cryptography. IoT devices often have limited computational resources and cannot afford the overhead of traditional cryptographic methods. RBC provides a lightweight and secure alternative that can be implemented on these devices. Additionally, RBC is well-suited for post-quantum cryptography (PQC), as it does not rely on mathematical problems that could be easily solved by quantum computers.

Despite its advantages, RBC also faces several challenges. One of the main challenges is the inherent bit error rates in PUF responses. If keys do not match in cryptography then authentication will fail, therefore RBC must find and then correct the erroneous bits. Additionally, the introduction of noise and probabilistic responses can complicate the design, implementation, and robustness of RBC systems.

2.8.1 Challenges of RBC

RBC using PUFs is a relatively new method in the field of computer security and as such does not have as large a body of work to support it as metric-based indexing does. Despite this, we are able to utilize previous work in related fields to solve some of the issues which occur with RBC. When searching there are two main challenges that need to be addressed:

- Secure Hashing
- PUF Seed Permutation

The first, secure hashing, is mostly solved for our purposes. The National Institute of Standards and Technology (NIST) has published guidelines for what Secure Hashing Algorithm (SHA) methods to use [95]. Optimized versions of SHA are available for most hardware platforms and can be easily adapted to new protocols.

The second, key permutation, is used as input into SHA. The PUF seed, which is stored on the server, has to be permuted to account for a drift in the client device's PUF. A combination is generated to select which bits in the PUF seed to flip, and then those bits need to be flipped

in the seed before it inputted to SHA. The output of SHA is then compared to the revived hash/message digest before a new combination is generated and the seed further permuted. Generating the combinations is non-trivial and can take a significant portion of the overall runtime of the protocol. This is in part due to how well SHA has been optimized to run on GPUs and as such takes a relatively short amount of time to compute each hash compared to combination generation.

In this dissertation we seek to address this challenge with the following:

- Reducing the number of seeds that have to be searched.
- Reducing the combination generation time.
- Making the protocol more robust to communication interference.

2.9 Complexity and Evaluation

There are two approaches to evaluating search algorithms. The first is to examine the theoretical asymptotic time complexity of the algorithms and compare them based on the amount of calculations/work they perform. Using this approach we would rank the lower complexity as better. For example, consider finding an element in a sorted list of integers where a binary search has a time complexity of $O(\log n)$ while a scan has a time complexity of $O(n)$, we would consider a binary search to be faster since it has a lower complexity. This approach is platform agnostic and is appropriate for the general evaluation of methods for a simple comparison. One of the major difficulties with asymptotic time complexity analysis is that some searches have a large range in bounds. For example DBSCAN, a popular clustering algorithm that performs searches in Euclidean spaces does not have a well defined time complexity [46, 103] with a generally accepted time complexity of $O(n \log n)$ but with an upper bound of $O(n^2)$. This leaves a large amount of uncertainty in the time complexity analysis which is not well constrained. When comparing algorithms that accomplish the same task, this uncertainty makes it difficult to determine which algorithm should be employed

based on performance. In the similarity search algorithms we evaluate, the upper bound is $O(n^2)$ with a lower bound of $O(n)$ for querying all pints in a dataset with n points.

The second approach for evaluating search algorithms is to examine empirical metrics, such as the runtime (i.e. the time from start to finish). This raises a number of problems, the foremost of which is that the runtimes are dependent on the hardware platform used to execute the search algorithm. This adds a level of difficulty to evaluating algorithms which can only be addressed through excess testing on a wide range of hardware platforms. The benefit of this approach is that we obtain a more practical performance evaluation and based on hardware platform specifications. Additionally, time complexity analysis uses asymptotic notation which often obfuscates overheads that are present in many algorithms. For expensive searches, the overhead tends to be negligible but on smaller workloads the asymptotic notation may not be reflective of an algorithm's real-world performance. By evaluating the search algorithms on a range of dataset characteristics (i.e. distribution, dimensionality etc.), we can show the effects of overhead as well as illustrate the actual performance of the searches.

In this dissertation research, we focus on timed evaluations though we do consider the time complexities of our algorithms and how those relate to the observed runtimes.

2.10 Parallel Computing and GPUs

Part of the focus of this dissertation is on designing data structures and algorithms that take advantage of modern hardware. We use both parallel CPU and discrete GPU (i.e. a GPU on a separate chip from the CPU) algorithms to accelerate our searches. Parallel computing has a set of paradigms encapsulated by Flynn's Taxonomy which we describe as follows.

Flynn's Taxonomy: In parallel computing there are four main ways of issuing instructions: (i) Single Instruction Single Data (SISD), where a single processor executes a single instruction on a single data stream; (ii) Single Instruction Multiple Data (SIMD), where

multiple data elements are processed simultaneously using one instruction; (iii) Multiple Instruction Single Data (MISD), where multiple instructions operate on a single data stream; and Multiple Instruction Multiple Data (MIMD), where multiple processors execute multiple instructions on multiple data streams independently. Specific to GPUs, there is also Single Instruction Multiple Threads (SIMT) where single instructions operate in lockstep across a warp of threads.

GPUs have a substantially different architecture from CPUs with some of the following benefits highlighted below for an NVIDIA A100 GPU [90] and an AMD EPYC 9004 series CPU [4]:

- High memory bandwidth (GPU has $\geq$ 2 TB/s [90] versus up to 512 GB/s with the CPU).
- High compute throughput (GPU has 312.5 TFLOPS for single-precision versus up to 11 TFLOPS with the CPU).
- Large number of compute cores (GPU has 6912 CUDA cores versus up to 128 cores on the CPU).

While theoretically GPUs outperform CPUs available today, GPUs do come with some substantial downsides which may significantly reduce comparative performance. Some of these drawbacks are as follows:

- Warps of threads (32 threads with NVIDIA GPUs) must operate in lockstep because of the single instruction multiple thread (SIMT) architecture.
- Limited global memory (GPU has up to 80 GB versus typically 512 to 1024 GB with the CPU).
- Communication bottleneck over the interconnect from the CPU to the GPU (this is often PCIe).

- Overhead results in high latency when launching a large number of GPU kernels.
- Dynamic memory is largely unsupported (some recent advances allow for dynamic memory but at the expense of highly reduced performance).

We discuss the strengths and weaknesses of both multi-core CPUs and GPUs in detail below.

Multi-core CPUs: Multi-core CPUs tend to have higher clock speed than GPUs but have far fewer cores. They have a lower memory latency than GPUs but also a lower overall bandwidth. Additionally, each core on a CPU can operate independently from the other cores allowing for flexibility when designing parallel algorithms. These attributes of multi-core CPUs make them well suited to algorithms which have a large number of branching instructions, since each core can operate independently, and scenarios in which a single thread or process needs to complete before other work because a single thread executing on a core on the CPU is going to compute faster than a single core on a GPU. CPUs will have reduced performance when running into memory bandwidth limitations, a problem that GPUs were specifically designed to address. CPUs will also have less overall computational throughput when compared to a GPU because they have fewer overall cores. CPUs pair well with GPUs for pre-processing data and building data structures which can then be transferred to the GPU for the bulk of the computations as we show to be the case with COSS, MiSTIC and npRBC-CPU.

GPUs: While GPUs where originally created for computer graphics their uses have expanded over the years. Most supercomputers have GPUs for general purpose computing which have more memory and cores but lower clock speed compared to consumer grade GPUs which are still mostly used for graphics applications. GPUs have an inherent latency because the all of the memory needs to be transferred from the host (by the CPU) to the GPU. Additionally, creating and launching the number of threads necessary to saturate the high core count on the GPU increases the latency due to these overheads. Because of the high latency when using the GPU it is better for high throughput oriented problems

which lends itself very well to searching high-dimensional spaces that tend to require a large amount of computation and high memory bandwidth. GPUs have sets of threads (32 for NVIDIA GPUs) called warps that must perform the same instructions at the same time but can process different data. This is not a problem for range queries which perform simultaneous distance calculations, nor is it a problem for most cryptographic algorithms which can perform multiple encryptions or hash on different data at the same time.

In this dissertation we examine similarity searches designed specifically for the GPU, which are compared to existing multi-core CPU methods. Additionally, we evaluate npRBC-CPU on multi-core CPU hardware platforms and compare to npRBC-GPU, a GPU optimized version of npRBC-CPU.

Now that we have introduced the key concepts needed to understand the content in this dissertation, we begin with our first research area, similarity searches on Euclidean distance spaces in the following Chapter.

Chapter 3

Coordinate Oblivious Similarity Search (COSS)

This chapter presents COSS, which is a method for high dimensional similarity searches. The goal with this work was to adapt modern approaches for similarity searches on the GPU and adapt them to work with metric-based indexing in order to mitigate the *curse of dimensionality* problem described in Section 1.1.1.

This work originally appeared in the reference below and has been adapted for this dissertation from its original format.

Donnelly, B., & Gowanlock, M. Proceedings of the 34th ACM International Conference on Supercomputing (ICS 2020), Barcelona, Spain, Article No. 8, pp 1–12, 2020.

3.1 Abstract

We present COSS, an exact method for high-dimensional distance similarity self-joins using the GPU, which finds all points within a search distance ϵ from each point in a dataset. The similarity self-join can take advantage of the massive parallelism afforded by GPUs, as each point can be searched in parallel. Despite high GPU throughput, distance similarity self-joins exhibit irregular memory access patterns which yield branch divergence and other performance limiting factors. Consequently, we propose several GPU optimizations to improve self-join query throughput, including an index designed for GPU architecture. As data dimensionality increases, the search space increases exponentially. Therefore, to find a reasonable number of neighbors for each point in the dataset, ϵ may need to be large.

The majority of indexing strategies that are used to prune the ϵ-search focus on a spatial partition of data points based on each point's coordinates. As dimensionality increases, this data partitioning and pruning strategy yields exhaustive searches that eventually degrade to a brute force (quadratic) search, which is the well-known curse of dimensionality problem. To enable pruning the search using an indexing scheme in high-dimensional spaces, we depart from previous indexing approaches, and propose an indexing strategy that does not index based on each point's coordinate values. Instead, we index based on the distances to reference points, which are arbitrary points in the coordinate space. We show that our indexing scheme is able to prune the search for nearby points in high-dimensional spaces where other approaches yield high performance degradation. COSS achieves a speedup over CPU and GPU reference implementations up to 17.7× and 11.8×, respectively.

3.2 Introduction

Similarity searches are fundamental database operations and are used in data analysis. For example, similarity searches [78, 96, 98, 115] are used in clustering algorithms [103], and k-nearest-neighbors searches [3, 25]. This paper examines the distance similarity self-join problem [42, 48, 78, 92], defined as searching a distance ϵ around each point in a dataset and returning all of the neighbors within this search distance. We focus on a GPU-efficient, coordinate-oblivious index that prunes the search for nearby points. While we use the index for the distance similarity self-join, the index can be employed in other spatial search algorithms.

A semi-join on two datasets $A \ltimes_\epsilon B$ involves comparing every point in A to every point in B with a complexity $O(|A| \cdot |B|)$. Comparatively, self-joins ($A \bowtie_\epsilon A$) involve comparing all of the points in a single dataset with a complexity $O(|A|^2)$. In this paper, we examine the self-join, but note that the method and most optimizations proposed can be employed for the semi-join as well.

The brute force approach to the distance similarity self-join computes the distance from

every point to every other point yielding a time complexity of $O(n^2)$, where n is the number of data points in a dataset, making the approach impractical for large datasets. Index-trees use the data's coordinate values to build a hierarchical data structure of partitions. For example, kd-trees [51, 123], R-trees (and R-tree variants) [8, 26, 52, 53, 57, 58, 65, 83, 87, 94, 120], and X-trees [12] are all types of trees that prune the search for nearby objects and are optimized for specific application scenarios. Grid-based indexes with fixed length cells [47, 48, 62, 92, 96] have also been proposed to partition the dataset. The major difference between index-trees and grids is that many index-trees construct the index based on the positions of the points, whereas static grids partition the space independently of the data distribution.

Both trees [65, 94] and grids [47, 48, 62, 92, 96] have been designed for the GPU. Searching an index on the GPU introduces several challenges related to both index types. For example, searches on trees require tree traversals which may lead to divergent execution paths that degrade performance on GPUs [65, 90]. Depending on the type of query, a static grid may perform worse than a tree, because the data partitions are of equal size. For the self-join problem with a fixed search radius, static grids are an attractive option because ϵ-length cell sizes can be utilized, which bound the search to neighboring cells [62]. Additionally, grids may have less branch divergence than trees, since trees require many branch conditions in their traversals [65, 94].

The volume of the space that needs to be searched grows exponentially with data dimensionality. To find points near each other, the search distance ϵ needs to increase proportionately to the increase in dimensionality. Figure 3.1 shows the ϵ needed to find one average neighbor on a uniformly distributed dataset. We observe that as the dimensionality of the data increases, ϵ has to increase to maintain finding a single neighbor. In a grid-based index the search within a unit hypercube becomes brute force when $\epsilon = 0.5$, which occurs at only 18 dimensions (Section 3.4.1). This illustration shows that the pruning efficacy of methods that index based on the coordinate space of the data (e.g., grids and trees) causes most index searches to degrade rapidly into a brute force search. This is known as the *curse*

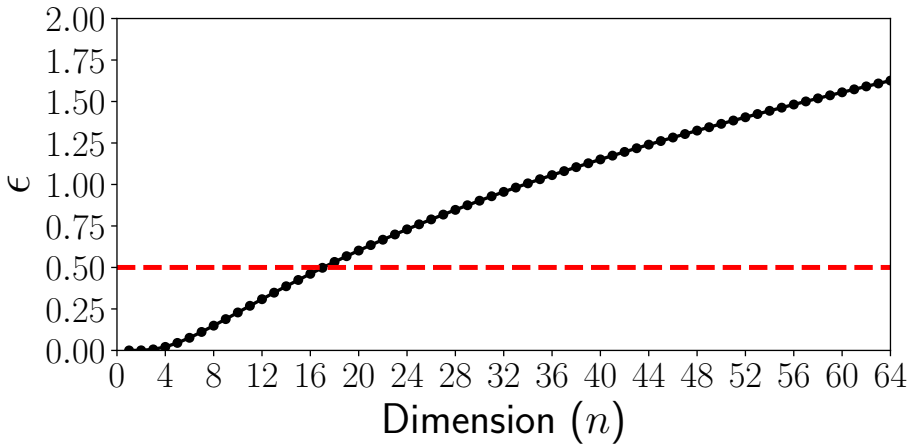


Figure 3.1: ϵ vs. n showing the minimum value of ϵ required to find a single neighbor (on average) within a unit n-dimensional hypercube, $\beta = [0, 1]^n$, where the $|D| = 10^6$ data points are uniformly distributed. When $\epsilon \geq 0.5$ (red dashed line), a grid-based index degrades to a brute-force search.

of dimensionality problem [9].

The ability to efficiently use the GPU makes grid-indexing a good solution for large dataset analysis. With a higher memory bandwidth, and a massive throughput for floating point calculations [89], GPUs provide the ability to replace large multi-core systems with a single device [119].

We propose COSS– a GPU algorithm for high-dimensional similarity searches. GPUs have high memory bandwidth and high throughput for floating point calculations [89], which are needed to compute Euclidean distances between neighbors. COSS is designed to address high-dimensional similarity searches by constructing a coordinate-oblivious index in distance space. COSS indexes based on the distance to an arbitrary point in space, that we denote as a reference point. Data points are then ordered and assigned to bins based on this distance. COSS has a similar instruction flow as searches on grids, but does not partition on coordinate space. By indexing based on the distance to a reference point, we can construct an index that does not rely on the individual coordinates of the data, but instead utilizes the entire set of coordinate values for indexing. This reduces the curse of dimensionality problem described above. We show that COSS is a more efficient algorithm than other

state-of-the-art methods for high-dimensional distance similarity self-joins. We outline the major contributions of this paper as follows:

- We propose a novel coordinate-oblivious indexing method, COSS, tailored to exact similarity searches on high dimensional data using the GPU.
- By using our coordinate-oblivious indexing scheme, we optimize pruning power by selecting the number of reference points and their location.
- We leverage several optimization that improve index performance and memory management, including dimensional ordering, short circuiting the distance calculations, and batching the computation across multiple kernel invocations.
- We evaluate COSS on 3 real-world datasets, 2 synthetic datasets and compare to other state-of-the-art methods, SUPER-EGO and GPU-JOIN.

The paper is organized as follows. Section 3.3 presents the problem statement, Section 3.4 discusses the curse of dimensionality problem and related work, Section 3.5 presents our coordinate-oblivious indexing scheme, Section 3.6 presents the optimizations used in COSS, Section 3.7 presents our results, and finally, Section 3.8 concludes the paper.

3.3 Problem Statement

We outline the distance similarity self-join problem, denoted as $D \bowtie_\epsilon D$, as follows. Let D be a dataset, containing $|D|$ points (or feature vectors), in n dimensions. Each point is defined as $p_i \in D$, where $i = 1, 2, \ldots, |D|$. We denote the coordinates of each point, $p_i \in D$, as $p_i = (x_1, x_2, \ldots, x_n)$. Like other works [12, 47, 48, 58, 62, 92, 96] we use the Euclidean distance similarity measure. The Euclidean distance between points $r \in D$ and $s \in D$ is defined as $dist(r, s) = \sqrt{\sum_{j=1}^{n}(r_j - s_j)^2}$. The self-join performs similarity searches on all points in the dataset, $p_i \in D$. A pair of points r and s are added to the result set

if $dist(r,s) \leq \epsilon$. The value of ϵ directly controls the selectivity of the self-join, where the selectivity refers to the average number of neighbors found per point in the dataset, D.

In this paper, all processing occurs in-memory. We consider the case where the result set size may exceed the GPU's global memory capacity instead of limiting our work to the case where the result set must fit within global memory on the device. Since the result set size is typically much larger than the input dataset size, we do not allow for the case where the input dataset exceeds global memory capacity.

3.4 Background

In this section, we provide an overview of the motivation and literature. We use CUDA terminology throughout this section and paper.

3.4.1 Motivation: Selectivity and the Curse of Dimensionality

We illustrate the relationship between dimensionality (n), ϵ, and selectivity, where selectivity refers to the average number of neighbors found by each point. We denote selectivity as $S = (|R| - |D|)/|D|$ where R is the result set and D is the input dataset. We draw on the example given by Kalashnikov [62], and refer the reader to that paper for a comprehensive discussion of selectivity. Consider a unit hypercube containing $|D| = 10^6$ uniformly distributed data points in the bounding volume defined by $\beta = [0,1]^n$. Because points are uniformly distributed in the hypercube, as the dimensionality, n, increases, the search distance ϵ will need to increase to find neighboring points. Assume that we wish to find 1 neighbor on average (i.e., a selectivity $S = 1$).

We compute the value of ϵ needed to find $S = 1$ using geometric arguments. First, we define the volume of an n-dimensional sphere with radius ϵ as follows: $V(n,\epsilon) = g(n)\epsilon^n$, where $g(n) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$. If the volume needed to search a query point lies entirely within β, and is not positioned near the edge of the bounding volume, β, then the point is more likely to find neighbors within its search radius. We consider this best case scenario for a given

search.

To find a selectivity of $S = 1$ point on average, we want to find the value of ϵ where $|D| \cdot g(n)\epsilon^n \geq 1$. Solving for ϵ, we obtain $\epsilon \geq (|D| \cdot g(n))^{-1/n}$. Figure 3.1 plots ϵ vs. dimension (n), where a value of ϵ below that, plotted for a given value of n, yields $S < 1$ (less than 1 neighbor found per point on average). Since grid indexing schemes constrain the search to adjacent grid cells, then a search in β with $\epsilon \geq 0.5$ will degrade to a brute force search because searching adjacent grid cells will span the entire bounding volume, β [62]. From the plot, we find that at $n = 18$, $\epsilon \geq 0.53$ is needed to find a single neighbor. Consequently, for uniformly distributed data, indexing the data based on their coordinate values will degrade to a brute force search when $n \geq 18$ dimensions. This illustrative example shows the pitfalls of using grid-indexing schemes for high-dimensional data. Additionally, other methods that index the data based on a point's coordinate values, such as index-trees (e.g., R-tree [52], X-tree [12], kd-tree [123]), suffer from the same curse of dimensionality problem.

3.4.1.1 Dimensionality Reduction and Approximate Solutions

One method of processing high-dimensional datasets and counteracting the curse of dimensionality is to use a feature extraction method like Principle Component Analysis [116] or Map Analysis [25]. While reducing the effective dimensions of the data is a straightforward method for reducing the computation time, there is a loss of data that results in approximate solutions. For an exact solution, a larger amount of computation is needed [48], and we focus on exact similarity searches in this paper.

3.4.2 Related Work

Indexing methods can reduce the runtime of a distance similarity self-join by reducing the total number of distance calculations needed [58]. Indexing methods partition the input dataset, allowing the algorithms to prune the search space by only evaluating nearby searched query points. There are two main approaches to indexing: one is to construct an

index using the coordinate values of the points (e.g., kd-trees [123]), and the other is a data oblivious approach that creates the index by partitioning the space (e.g., statically partitioned grids [62]). Our COSS algorithm is differentiated by its coordinate-oblivious index; COSS does not rely on either the data coordinate values or partitioning the coordinate space to construct the index.

3.4.2.1 Index-trees

Trees construct a hierarchical index that partitions the coordinate space. [8, 12, 26, 51, 52, 53, 57, 58, 65, 83, 87, 94, 120, 123]. For example, in an R-tree, when a query point is being searched, the tree is traversed to find points within the query point's minimum bounding box. When concurrently searching the tree, traversals cause thread divergence on the GPU because of irregular instruction flow [65, 94]. Several methods have been developed to improve tree searching performance on the GPU. For example, Kim et al. [65] propose a technique that allows trees to search on the GPU, minimizing the divergence and avoiding back-tracking.

While most CPU index-trees use a depth first search, GPU implementations use a breadth first search to help reduce branching [94]. The downside of the breadth first search is that it can require a large amount of dynamic storage. [94] Large storage requirements are problematic because of the limited amount of memory available on the GPU. Even with a number of optimizations made for index-trees that use the GPU, the architecture of the GPU may not be well-suited to index tree searches.

3.4.2.2 Grid-based Indexes

Grid-based indexing methods [47, 62, 92] build a structure that partitions the space and then assigns points to a cell based on their coordinate values. The index itself is constructed in a data oblivious manner, but the points are assigned to the cells based on the coordinate values of the points. In contrast to the R-tree, grid-based index searches use a deterministic

instruction flow when checking adjacent cells. This makes grid-based index searching more well-suited to the GPU architecture. ϵ Grid Order [15] indexes use cells that have edges of length ϵ, when ϵ becomes a large portion of the range in a single dimension, the number of total cells decreases along with pruning efficiency. We discuss two grid-based implementations in the following section.

3.4.2.3 The iDistance Method

Jagedish et al. [59] propose the iDistance method which creates an adaptive B^+-tree by indexing the points on distance to a reference point in the coordinate space. Each point in the dataset is assigned to the closest reference point. A one-dimensional B^+-tree is constructed using the distance from each data point to its assigned reference point. This indexes the data on a single dimension based on distance to the nearest reference point. In contrast to indexing directly on the coordinate space of the data, points that are adjacent in the iDistance B^+-tree may not be adjacent in the coordinate space.

Similarly to COSS, the iDistance method uses the distance to reference points to construct an index. In contrast to our proposed algorithm, iDistance creates a one-dimensional tree using multiple reference points, while COSS creates a multi-dimensional grid-like index. The data points are assigned locations in the index based on their distance to every reference point in the COSS algorithm, while iDistance only uses the distance to a single reference point for each data point. Therefore, COSS has the ability to increase pruning capability compared to iDistance. While iDistance is not implemented on the GPU, the B^+-tree structure would have the same problems as other tree-indexes as discussed above, while COSS is designed specifically to exploit GPU hardware.

3.4.3 Reference Implementations

We compare COSS to two state-of-the-art reference implementations SUPER-EGO and GPU-JOIN. We review the two methods below.

3.4.3.1 Super-EGO

Kalashnikov's SUPER-EGO [62] is a CPU-only grid-indexing method that indexes dimensions intelligently. By carefully selecting which dimension to index, the Super-EGO algorithm is able to increase pruning. One weakness of the algorithm pointed out by the authors is that, for datsets normalized to $[0, 1]^n$, any $\epsilon \geq 0.5$ causes the runtime to become quadratic. This is a similar weakness shared by other grid-based indexes and index-trees which our method addresses. In this paper we use SUPER-EGO as a reference for evaluating the performance of COSS.

3.4.3.2 GPU-Join

Gowanlock and Karsin [47] introduce a GPU grid-index for joins that has several optimizations to improve performance on high-dimensional datasets. GPU-JOIN reduces the number of indexed dimensions to avoid increasing the cost of index searches, while this increases the number of distance calculations, it reduces the overall work. When reducing the amount of partitioning, the algorithm uses statistics to decide which subset of the dimensions to index on, thereby maximizing the pruning effectiveness of the index. These optimizations allow GPU-JOIN to address high-dimensional datasets.

3.5 Indexing on Distance Spaces for High-Dimensional Data

3.5.1 Overview: Indexing by Distance to Points

To mitigate the curse of dimensionality (see Section 3.4.1), we can construct a coordinate-oblivious index. Our proposed index uses the distance to an arbitrary point in the coordinate space. We call this arbitrary point a reference point and find the distance between it and every other point in the dataset. Figure 3.2 shows the *distance space* with a reference point RP and 10 data points. The distance from the reference point is segmented into ϵ-width bins. The point p_4 in Figure 3.2 is in the second bin, so we know that there is no possibility

of it being within ϵ of p_8 which is in the fourth bin. Searches for points within ϵ of p_4 can prune any points that are not in bins 1, 2 or 3, because points in other bins exceed the search distance ϵ.

We refer to the *distance space* as the location of each $p_i \in D$ based on its distance to the reference points (e.g., Figure 3.2 is a 1-D distance space, and Figure 3.3 shows a 2-D distance space). We refer to the *coordinate space* as the typical Cartesian space that contains the input dataset point coordinates of each $p_i \in D$.

The index stores the bins that each point is located in. The number line in Figure 3.2 shows how the points would be placed into bins based on their distance to the reference point (RP). In Section 3.5.2.1 we show how we use the distance to a reference point to construct the index.

By indexing with the distance to a reference point, we avoid relying on the coordinate space to partition the data. This method directly addresses the issue that arises from selectivity and the curse of dimensionality discussed in Section 3.4.1. This index is still affected by the increase in the dimensionality of the data, but only insomuch as that it affects the distances between points. The distance space is entirely independent of the dimensionality of the data. Consequently, this yields an opportunity to have a higher pruning capacity than methods that index on the coordinate space (e.g., index-trees and grids).

3.5.2 Bin and Index Construction

3.5.2.1 Reference Point Bin Construction

We create a set of W reference points, where the reference points are denoted as l_t, where $W = (l_1, l_2, \ldots, l_{|W|})$ and has the coordinates $l_t = (x_1, x_2, \ldots, x_n)$. We construct all reference point bins, $B = (B_1, B_2, \ldots, B_{|W|})$, by computing the the Euclidean distance between all $p_i \in D$ to all reference points, W. We define an array Q, where $|Q| = |D|$, which contains the point ids. We then stable sort the arrays B (keys) and Q (values) as key-value pairs. This is repeated $|W|$ times, each time using a different subset of B. Consequently, points

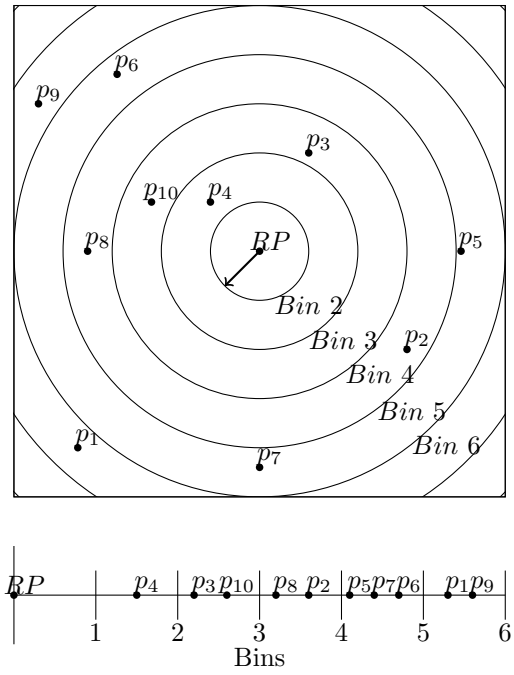


Figure 3.2: This figure shows a graphical example of an index with a single reference point, RP. The number line in the bottom of the image shows where points fall into ϵ-width bins based on their distance to RP.

that are within the same bin are stored contiguously in Q.

Example Bin Construction: In Figure 3.3 we construct an array B for the bins using two reference points RP_1, and RP_2, for an example dataset $D = (p_1, p_2, \ldots, p_{10})$, where $|D| = 10$. In step 1, we start with RP_2, and find the distance from every point $p_i \in D$ to RP_2, finding which bin each point falls into. We store the bin number for each point in array B_2, and store the corresponding point ids in Q. After B_2 has been computed, we sort Q and B_2 with a stable key-value sort that uses the bin numbers in B_2 as the key. This gives us an ordered array Q that starts with points in the lowest bin number and ends with points in the highest bin number. Arrays B_2 and Q in step 1 of Figure 3.3 show this sorted state.

In step 2 we consider RP_1 and repeat the procedure in step 1. When we use the stable sort on B_1 and Q, the points will maintain the order from B_2 in step 1 within the individual bins of B_1. This gives a final array B that is sorted from lowest to highest bin. Note that while p_2, p_6 and p_9 are spatially far apart, we can see that they have the same final bin numbers in the B array. This illustrates that points may be within the same bin in our

index (nearby in the distance space) but distant in the coordinate space.

After B has been constructed, B contains duplicate bin ids. We reduce B to remove the duplicate bin ids to create B', such that we do not store redundant bin ids in the array; therefore, B' contains all of the non-empty and unique bin ids. To keep track of which points are in each bin, we construct a range array C as will be discussed in Section 3.5.2.2.

3.5.2.2 Index Construction

We compute the Euclidean distance between each reference point in W and $p_i \in D$, yielding the bin that contains each point. Using this information, we sort the points based on bin and store this information in Q. Next, we store B' (constructed as described in Section 3.5.2.1) which contains the unique bin ids (since many points may fall within a single bin, and some bins are empty, we only store the ids of the non-empty bins). We construct an array A that maps Q to B', which indicates the bin id of each point id in Q. For example, the point in $Q[i]$ is stored in the bin at $B'[A[i]]$. We construct an array C where $|C| = |B'|$ and $C[A[i]]$ contains the range of points in Q that are stored in bin $B'[A[i]]$. We illustrate the components of the index, when we show an example search in the next section.

3.5.3 Searching the Index

Each $p_i \in D$ is located within a single bin, where each bin is defined by $|W|$ bin numbers. Each bin has an address corresponding to the bin numbers. For example, a bin constructed with two reference points has bin numbers y_1 and y_2; therefore, adjacent bins are in the ranges $[y_1 - 1, y_1 + 1]$ and $[y_2 - 1, y_2 + 1]$. All the points in a bin will only need to evaluate the distance to points in the same, or adjacent bins as non-adjacent bins are separated by a distance $\geq \epsilon$.

We refer to a *query point* as a point in the dataset that is being searched. To find the adjacent bins for a query point, we take the query point's bin and compute the adjacent bin numbers (described above). We then do a binary search on array B' for those bin numbers.

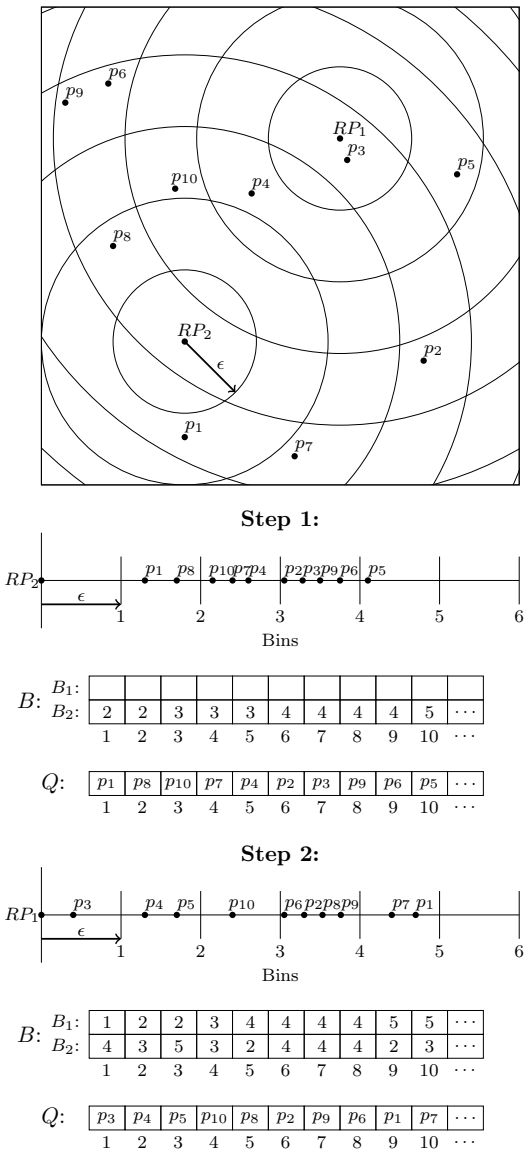


Figure 3.3: Example showing the construction of B and Q for two reference points. Every additional reference point adds an additional construction step. Note that the distance to a reference point within a bin does not impact the sorting, but the stable sort maintains the ordering from previous steps within a bin.

Note that B' only contains non-empty bins, so only a fraction of searches find a non-empty bin. Increasing the number of reference points increases the number of binary searches, as each query point executes $3^{|W|}$ binary searches.

Since we compute the self-join, we can eliminate duplicate distance calculations using

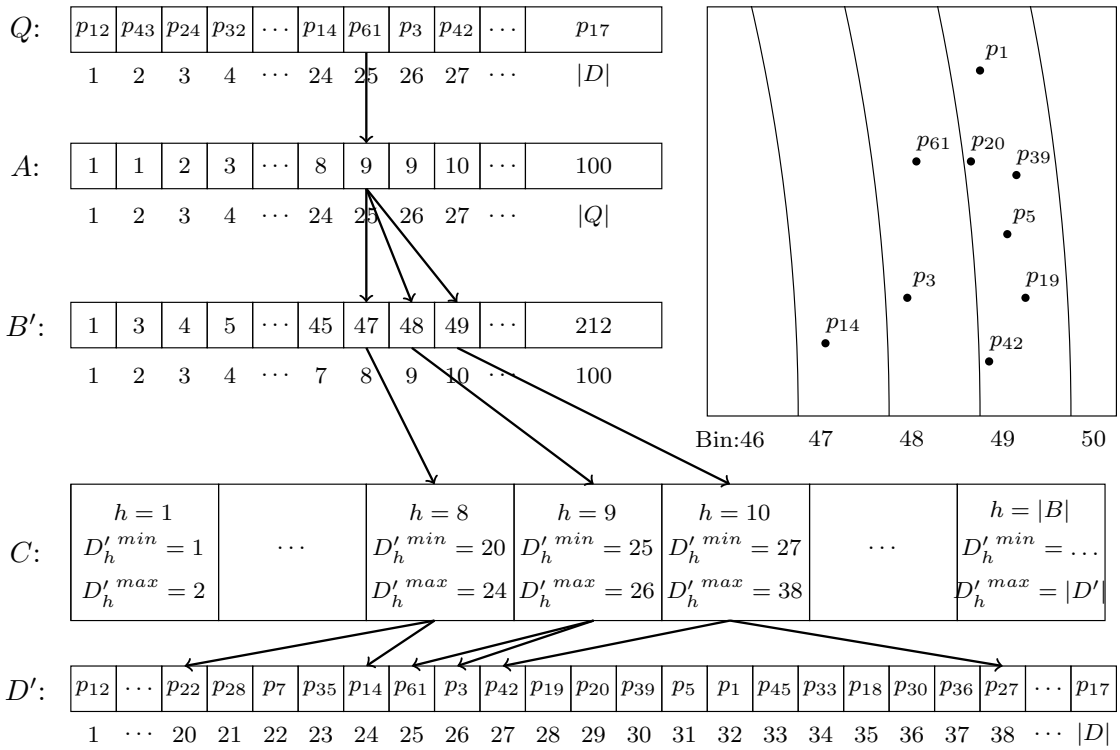


Figure 3.4: Grid indexing example, where Q is the point array, A is the lookup array, B' is the unique bin array, C is the range array, and D' is the sorted data array. For clarity, only one reference point is shown. When there are multiple reference points, B' will be a multidimensional array, constructed as described in Section 3.5.2.

the reflexive property (i.e., $dist(r, s) = dist(s, r)$), which reduces the total work by roughly half. To eliminate these distance calculations, we use the Unidirectional Comparison strategy developed by Gowanlock and Karsin [48] to select which bin numbers each query point will need to search. In short, the method halves the average number of adjacent bin searches. We refer the reader to Gowanlock and Karsin [48] for more detail. This optimization does not apply to the semi-join problem, only the self-join. All other optimizations (described in Section 3.6) can be applied to both the self-join and semi-join problems.

Example Search: Immediately after index construction, we transform D into D' by key-value sorting based on Q. This causes all of the coordinate data in D' to be mapped to the indices of Q.

For clarity, we outline an example search of our index without the Unidirectional Com-

parison [48] strategy and only index using a single reference point. Figure 3.4 shows an example of a search to find those points within ϵ of the query point $p_{61} \in D$. Point p_{61} at $Q[25]$ maps to $B'[9]$ using mapping array A. Since p_{61} is found in bin 48, we need to search adjacent bins, yielding a bin range of $[47, 49]$ (the three arrows from A to B'). Bins 47, 48, and 49 are found in $C_h = 8, 9, 10$, respectively. Note that the non-empty bins are not stored in B' or C, which is why indices of B' correspond to the indices of C. Bins 47, 48, and 49 contain the following candidate points and comprise the candidate set K, where $K = \{D'[20], \ldots, D'[24]\} \cup \{D'[25], D'[26]\} \cup \{D'[27], \ldots, D'[38]\}$. The Euclidean distance is computed between p_{61} and each point in its candidate set, K.

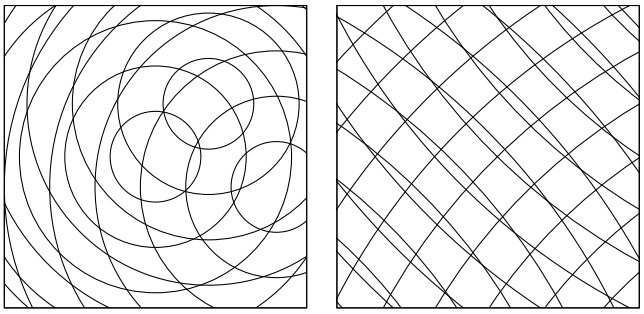
With multiple reference points the search only needs to consider more bins in B'. These are additional binary searches whose effects on the performance of COSS is discussed in Section 3.7.4.1.

3.5.4 Selecting the Location of Reference Points

We propose two reference point placement heuristics. While the selected position of each reference point in the coordinate space is arbitrary, the positions will impact the pruning efficiency of the algorithm. We note that finding the optimal positions of reference points that minimize the number of point comparisons is intractable.

RP-Inner: This strategy places multiple reference points close to the center of the data. We take the average value of the data in each dimension and place the first reference point at that location. The subsequent reference points are placed around the centered reference point in an expanding area. This creates a large number of small bins near the average center of the data, with bins that grow in size with distance from the center. Figure 3.5(a) shows an example of this placement strategy and the pattern of bins that develops near the center of the data.

RP-Outer: This strategy places the reference points at the edges of the data. The first reference point will be placed at the farthest range in every coordinate. To place the subse-



(a) (b)

Figure 3.5: (a) shows the pattern generated with the RP-Inner placement strategy. (b) shows the pattern generated by the RP-Outer placement strategy.

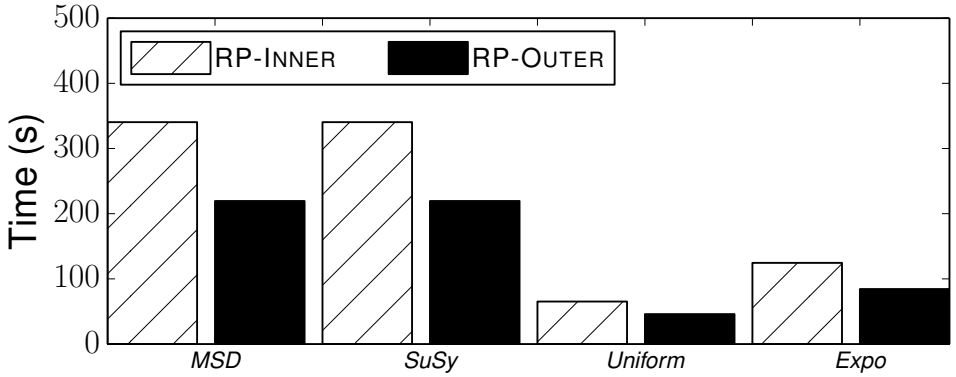


Figure 3.6: We compare the runtimes for RP-Inner and RP-Outer reference point placement strategies with *MSD* ($n = 90, \epsilon = 0.007$), *SuSy* ($n = 18, \epsilon = 0.015$), *Uniform* ($n = 10, \epsilon = 0.35$), *Expo* ($n = 16, \epsilon = 0.04$).

quent reference points, we take the number of dimensions, n, and divide that by the number of remaining reference points $v = n/(|W| - 1)$. The reference points will have v max range values, with the rest of their coordinate values being 0 (every reference point, besides the first, has v unique non-zero values). This scatters all of the reference points around the outside of the data distribution. Figure 3.5(b) shows what three reference points on the outskirts of the data looks like. The bins made by the expanding rings are fairly consistent in size. Note that as the distance from one reference point increases, the distances to the other reference points decrease. The larger bins from the increased distance are offset by the smaller bins from the decreased distances, creating bins with a more even point distribution than RP-Inner.

Placement Method Comparison: Figure 3.6 shows the difference in total runtime for two real-world and two synthetic datasets. In every experiment, the RP-Outer placement strategy outperforms the RP-Inner placement strategy. Therefore, in all following evaluations of COSS we use the RP-Outer placement strategy.

3.6 GPU Algorithm and Optimizations

In this section we present an overview of COSS and algorithm optimizations.

3.6.1 Algorithm Overview

We present the pseudocode of COSS in Algorithm 1 and refer to the optimizations outlined later in this section. The COSSSelfJoin procedure begins by loading in the dataset D on line 2 and then ordering D according to the variance in each dimension (line 3 see Section 3.6.6). We then select our reference point placement based on the values in D' (line 4, see Section 3.5.4), set the number of threads per point (line 5, see Section 3.6.2), and construct our index (line 6, see Section 3.5.2). We compute the number of batches on line 7, initialize the max result size to zero (line 8) and then begin looping through every batch on line 9. For every batch we; execute COSSKernel on the GPU (line 10) as described on lines 17–30, check if the result size is smaller than the max results size (line 11) and pin memory for the result set buffer (line 12) if the result size was larger, and finally transfer and store the results on the host (lines 14 and 15).

The COSSKernel begins by storing the global thread id, and the query point's id and bin (lines 18 to 20). For every possible adjacent bin (line 21), we get the bin number to search (line 22) and search B' with a binary search to find which index that bin is at in B' (line 23). Note that on line 22, to avoid duplicate calculations by exploiting the reflexive property of the distance calculation, we apply the unidirectional comparison strategy [48] described in Section 3.5.3. If the bin is found in B' (line 24) we retrieve the min and max index into Q from C and store those points as the candidate set of the bin, Z (lines 25

Algorithm 1 COSS Algorithm

```
 1: procedure COSSSelfJoin
 2:     D ← inputData()
 3:     D′ ← dimensionalOrdering(D)
 4:     W ← placeReferencePoints(D′)
 5:     t ← setNumberThreadsPerPoint()
 6:     Q, A, B′, C, D′ ← constructIndex(D′, W)
 7:     g ← computeNumberOfBatches(D′)
 8:     maxSize ← 0
 9:     for i ∈ (1, 2, . . . , g) do
10:         resultSize ← COSSKernel(Q, A, B′, C, D′, t)
11:         if resultSize > maxSize then
12:             pinMemory(resultSize)
13:             maxSize ← resultSize
14:         results ← tranferResultsToHost(resultSize)
15:         R ← R ∪ results
16:
17: procedure COSSKernel(Q, A, B′, C, D′, t)
18:     tid ← getThreadID()
19:     queryPointID ← Q[tid/t]
20:     queryPointBin ← A[tid/t]
21:     for i ∈ (1, 2, . . . , 3^|W|) do
22:         binToSearch ← generateNextBin(queryPointBin, i)
23:         binToSearchIndex ← searchBins(B′, binToSearch)
24:         if binToSearchIndex ≠ ∅ then
25:             minIndex, maxIndex ← C[binToSearchIndex]
26:             Z ← {Q[minIndex], . . . , Q[maxIndex]}
27:             for j ∈ (1, 2, . . . , |Z|) do
28:                 distance ← dist(Q[tid/t], Q[Z[j]], D′)
29:                 if distance ≤ ε then
30:                     results ← results ∪ (queryPointID, Z[j])
```

and 26). The pseudocode refers to assigning a single thread to process one query point ($t = 1$). When $t > 1$ we divide the $|Z|$ candidate points to be processed by t threads on line 27. In particular, for each query point, let $l = 1, \ldots, t$. Thread l is assigned candidate point j where $(l - 1) \bmod j = 0$. Then we compute the distance between the query point and all candidate points, checking if they are within ϵ (lines 28 and 29). If the distance is within ϵ we add the point pair to the result set (line 30).

3.6.2 GPU Thread Allocation

Modern GPUs have thousands of cores. We can make use of the cores by dividing the distance calculations for a query point across multiple threads. (The query point q is the

point that is evaluating a set of candidate points K found through the use of the index.) A query point q with an associated candidate set, K, will divide the work across multiple threads t. Where each thread has $|K|/t$ distance calculations to compute. Without the loss of generality and for illustrative purposes we assume t divides $|K|$. Figure 3.7 plots the runtime vs. the number of threads per a point on the *MSD* dataset (other datasets exhibit similar performance with the change in threads per a point). Every query point is assigned multiple threads to compute the distance calculations to refine the candidate set, K. From the experiments in Figure 3.7, we find that $t = 8$ achieves a good performance, and we use $t = 8$ threads in all the evaluation.

3.6.3 Batching Scheme

Depending on the search distance, ϵ, and data distribution, the result set size, $|R|$, may exceed global memory capacity. To ensure that the result set does not exceed global memory capacity, we divide the total computation into several batches. The batched execution allows us to concurrently execute tasks (e.g., pinning memory and host-GPU data transfers) in multiple CUDA streams. In this paper we use two CUDA streams.

The number of query points per a batch, h, is governed by the amount of global memory available to store results, which is contingent on the selectivity discussed in Section 3.4.1. The CUDA block size b and the number of blocks p determine the number of points that are evaluated in each batch/kernel invocation. We experimentally found that a block size $b = 1024$ yields the best performance. We use the number of blocks p per kernel invocation to determine h. The number of points evaluated per a batch is $h = b \cdot p$ and we can use h to find the total number of batches $g = \lceil |D|/h \rceil$. The total number of threads per a batch is $u = bpt$, where t is the number of threads per a point as discussed in Section 3.6.2.

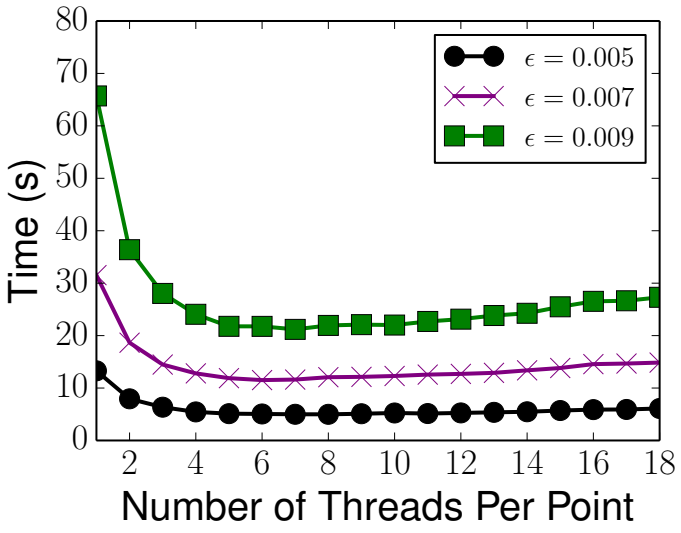


Figure 3.7: Runtime vs. the number of threads (t) on the *MSD* dataset ($n = 90, S = 4 - 1892$) shows that the while a small number of threads per a point has significantly longer runtime, $t = 5 - 10$ achieves good performance.

3.6.4 Concurrent Execution of Batches

The result sets generated on the GPU are large, it is more efficient to manually pin the memory needed and then reuse the pinned memory buffer. By pinning memory we can increase the effective bandwidth of the PCIe interconnect that connects the GPU to the host [90]. To determine how much memory needs to be pinned, each CUDA stream (COSS is evaluated with 2 streams) will execute one batch then take the size of the results and pin that much memory for the stream. The subsequent batches with smaller result set sizes can reuse the pinned memory buffer. After every batch is computed by a kernel invocation on the GPU we ensure that the pinned memory is sufficiently large to store the data, if not, we reallocate a larger pinned memory buffer.

High-dimensional distance similarity searches are compute bound. We can take advantage of the high computation time to hide memory transfers. Using two concurrent streams, one stream executes the kernel, and one sends results back to the host. Other host-side tasks are mostly hidden using two streams.

3.6.5 Short Circuiting the Distance Calculations

The distance calculation as defined in Section 3.3, is the summation of the distances in individual coordinate dimensions. When calculating this distance we compute the first term of the summation and add it to a running total distance, then compute and add the second term to the running total and so forth for all n terms. After we compute and add each term, we can check to see if the running total has exceeded the distance threshold, ϵ. If the running total exceeds ϵ we stop computing the distance, which reduces the total number of floating point operations computed.

3.6.6 Dimensional Ordering

We can increase the effectiveness of short circuiting (Section 3.6.5) by finding the variance of each dimension of the original data. We can then rearrange the point coordinates of the data so that the highest variance is first and the lowest variance is last. For example, the points $p_i \in D$ where $p_i = (x_1, x_2, \ldots, x_n)$, where n is the number of dimensions, will become $p_i = (x_{n_{max}}, x_{n_{max}-1}, \ldots, x_{n_{min}})$, where n_{max} is the dimension that had the most variance and n_{min} is the dimension that had the least amount of variance. When computing the distance calculations, this will result in the distance accumulating faster, leading to an earlier short circuit for most points. In high dimensions, this is especially effective because of the large number of dimensions and how early a distance calculation can exceed ϵ. Other grid-based algorithms use a similar dimensionality reordering method, including the two reference implementations, GPU-Join and Super-EGO.

3.7 Experimental Evaluation

3.7.1 Experimental Methodology

All host code is written in C/C++ and GPU code is written in CUDA and is compiled with the GNU compiler with the O3 optimization flag. GPU code is compiled using CUDA

Table 3.1: Datasets used in the evaluation.

Dataset	n	Size ($\|D\|$)	ϵ	Selectivity (S)
MSD [13]	90	$515,345$	$0.005 - 0.01$	$4 - 1892$
SuSy [6]	18	5×10^6	$0.01 - 0.02$	$5 - 780$
Higgs [6]	28	11×10^6	$0.035 - 0.045$	$5 - 91$
Uniform	10	2×10^6	$0.25 - 0.45$	$2 - 551$
Expo	16	2×10^6	$0.03 - 0.05$	$4 - 1226$

9. Our platform consists of 2x Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz, with a total of 16 physical cores, and 128 GiB of main memory, equipped with an Nvidia GP100 GPU with 16 GiB of global memory (Pascal generation).

In all experiments, we report the average runtime as averaged over 3 trials. As described in Section 3.7.3, we compare our algorithm, COSS, to GPU-Join, and Super-EGO. We refer to the total runtime of each algorithm using respective algorithm components described in Section 3.7.3.

To ensure that our experiments reflect real-world application scenarios, we report the selectivity of our searches as defined in Section 3.4.1.

3.7.2 Datasets

We select three real-world datasets from the literature and generate two synthetic datasets. *MSD* is a 90-D dataset containing song features, *SuSy* (18-D) and *Higgs* (28-D) are from particle physics. All datasets used in this paper are normalized for each dimension in the range $[0, 1]$. The range of dataset dimensions is consistent with other papers (real-world datasets in other works span $n = 9 - 32$ [75], $n = 2 - 784$ [62], and $n = 18 - 90$ [47]).

We selected uniformly and exponentially distributed datasets. The *Uniform* dataset represents the case where indexing on the data point coordinates leads to an increasingly exhaustive search (Section 3.4.1). The *Expo* dataset represents the opposite of the *Uniform* dataset, where there is one over-dense region and a large under-dense region. *Expo* was generated with $\lambda = 40$. The datasets are summarized in Table 3.1.

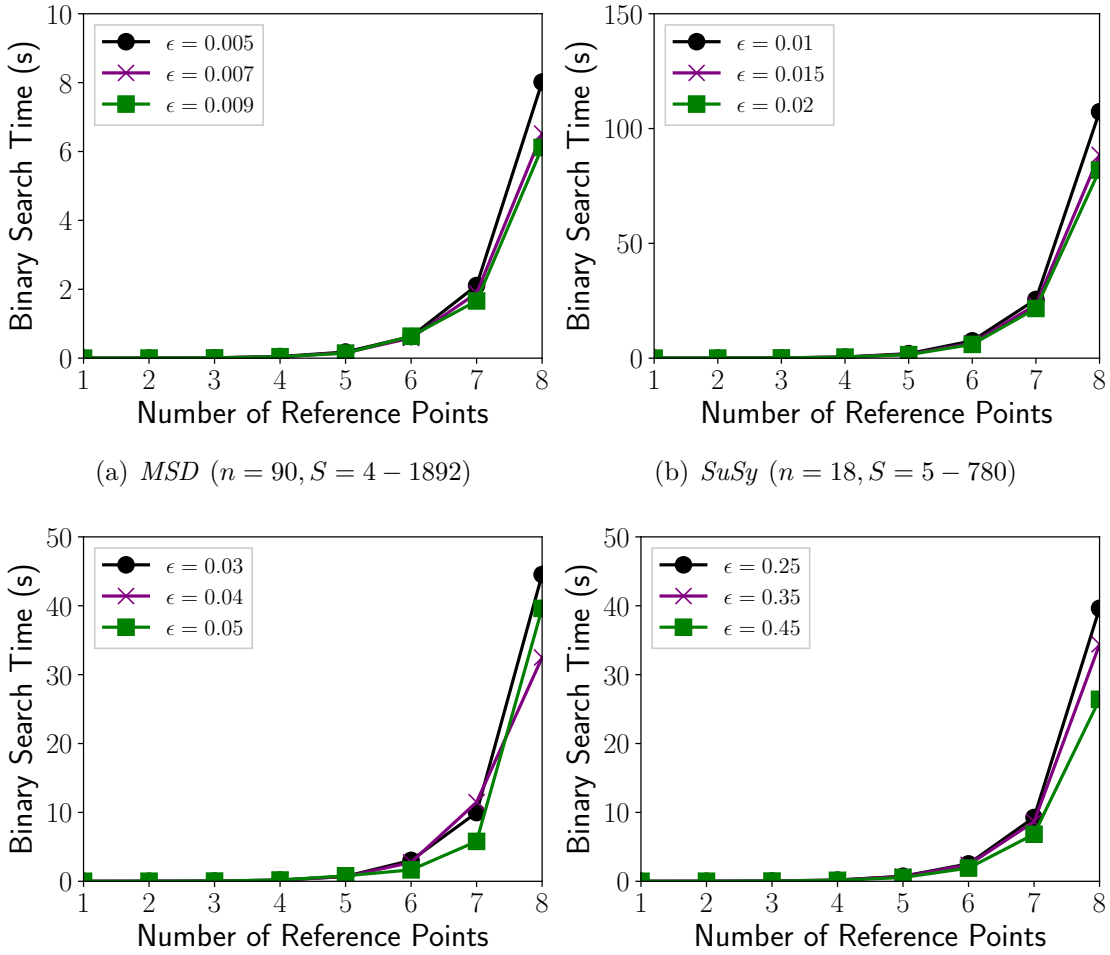
3.7.3 Implementations

COSS: COSS is evaluated when we enable all of the optimizations outlined in Section 3.6. We select a fixed set of parameters that achieve good performance across all experimental scenarios. COSS is configured with 2 CUDA streams, $t = 8$ threads per a point, 6 reference points and the RP-OUTER reference point placement strategy (Section 3.5.4). We include the time it takes to construct the index, pin and transfer memory, perform the distance calculations, and store the final results on the host. COSS is evaluated using 64-bit floating point values.

GPU-Join (GPU Reference Implementation): As described in Section 3.4.3.2, GPU-JOIN [47] uses a grid-based indexing scheme for the GPU. The algorithm uses several optimizations, including projecting the coordinates into $k < n$ dimensions, reordering the data by variance in each dimension, short circuiting the distance calculation, and reducing distance calculations by searching on an un-indexed dimension. We use the experimental parameters and configuration used in Gowanlock and Karsin [47] when executing GPU-JOIN. In particular, we enable all of their optimizations, and index on $k = 6$ dimensions, and use 256 threads per block. GPU-JOIN is executed using 64-bit floating point values which is consistent with COSS. Using the experimental methodology in Gowanlock and Karsin [47], the runtime excludes the time to index the dataset, but includes all GPU computation, and transferring the data and results to and from the GPU.

In contrast to COSS, GPU-JOIN does not eliminate duplicate searches for the same point, as GPU-JOIN [47] presents performance results that are directly applicable to both the self-join and the semi-join on two datasets (the self-join can eliminate duplicate work, but the semi-join on two datasets cannot). Therefore, we expect that GPU-JOIN will perform at least double the distance calculations as COSS.

Super-EGO (CPU Reference Implementation): As described in Section 3.4.3.1, SUPER-EGO indexes using a grid with ϵ-length cells, and prunes the search by employing a data reordering scheme. The algorithm is parallelized for multi-core CPUs. We execute



(a) *MSD* ($n = 90, S = 4 - 1892$)

(b) *SuSy* ($n = 18, S = 5 - 780$)

(c) *Expo* ($n = 16, S = 4 - 1226$)

(d) *Uniform* ($n = 10, S = 2 - 551$)

Figure 3.8: Index binary search time (s) vs. number of reference points.

SUPER-EGO using 16 threads (the number of physical cores on our platform). Since SUPER-EGO fails to execute when using 64-bit floating point values, we execute the algorithm with 32-bit values. This gives an advantage to SUPER-EGO over GPU-JOIN and COSS. The runtime is computed as the time to EGO-sort and join. The code is publicly available on the author's website.[1]

[1] `https://www.ics.uci.edu/~dvk/code/SuperEGO.html`.

3.7.4 Impact of COSS Parameters on Performance

Performance is evaluated on all datasets in this subsection except *Higgs* which was omitted due to space constraints. The results from *Higgs* are consistent with the datasets used in this subsection.

3.7.4.1 Binary Search Time

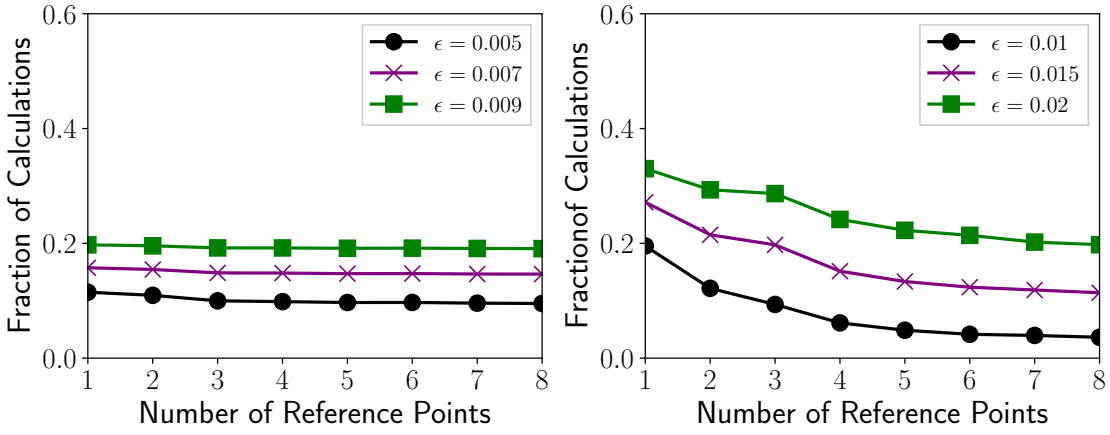
COSS uses binary searches to find adjacent non-empty bins in B'. We evaluate the binary search time vs. number of reference points and plot it in Figure 3.8. While the binary search times are insignificant in small numbers, when the number of reference points increases beyond 6 ,with $3^6/2$ searches per point (see Section 3.5.3), it begins to impact performance. In Figure 3.8 we observe the exponential growth in search time with the increase in the number of reference points. From this we conclude that it would be disadvantageous to use $\gtrsim 6$ reference points.

3.7.4.2 Pruning Efficiency

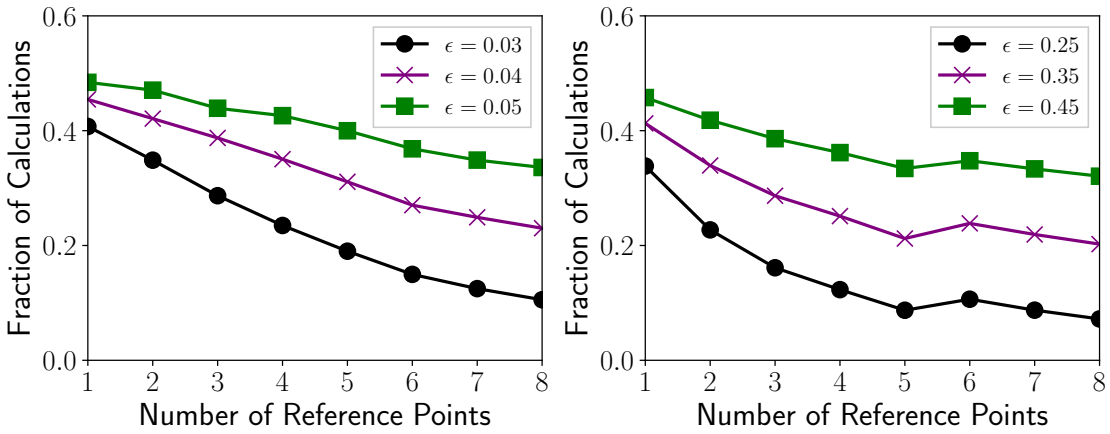
The efficiency of COSS is dependent on the amount of pruning that it can achieve. The amount of pruning is dependent on both the reference point placement strategy and number of reference points. Figure 3.9 plots the fraction of distance calculations vs. number of reference points where the fractions is calculated as $|K|/|D|^2$, where $|K|$ is the number of distance calculations made by COSS. Increasing the number of reference points greatly reduces the total number of distance calculations.

3.7.4.3 Effect of Number of Reference Points on Runtime

From the previous experiments we can see that the number of reference points impacts the performance significantly. Figure 3.10 shows the response time based on the number of reference points used. Figure 3.8 combined with Figure 3.9 explains how the response time in Figure 3.10 increases after 6 reference points. While the percentage of distance calculations



(a) *MSD* $(n = 90, S = 4 - 1892)$ (b) *SuSy* $(n = 18, S = 5 - 780)$



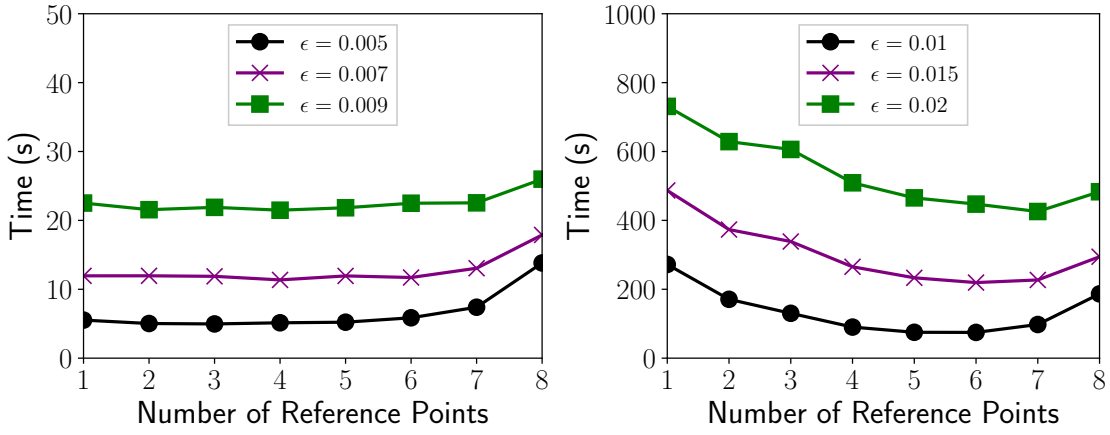
(c) *Expo* $(n = 16, S = 4 - 1226)$ (d) *Uniform* $(n = 10, S = 2 - 551)$

Figure 3.9: Fraction of distance calculations vs. number of reference points.

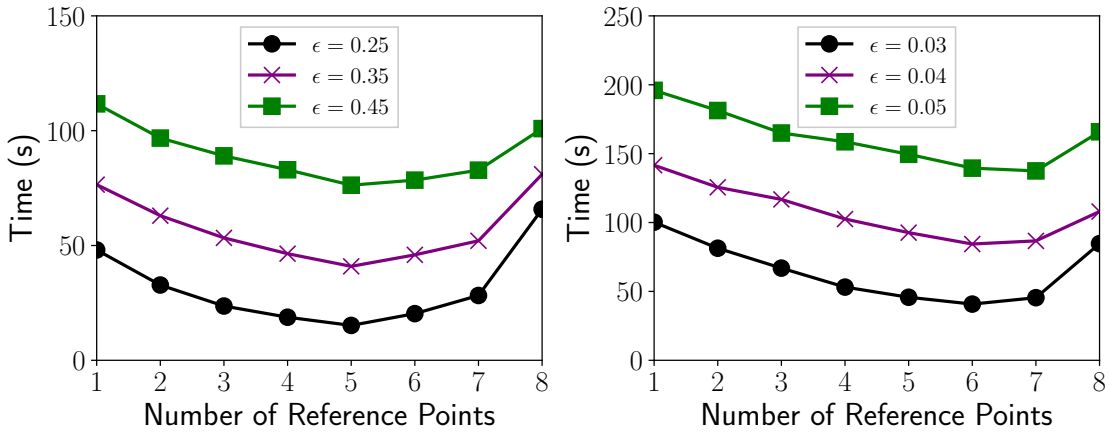
decrease, the number of binary searches increases rapidly. There is a trade off between time spent on the searches and time spent on the distance calculations. We find that 6 reference points performs well on all experimental scenarios.

3.7.5 Comparison to Reference Implementations

In this section we look at the experimental results across three real world data sets and one synthetic dataset. Table 3.1 shows a summary of the datasets used. We choose to use 6 reference points for making comparisons to other methods to maintain consistency across different datasets.



(a) *MSD* ($n = 90, S = 4 - 1892$) (b) *SuSy* ($n = 18, S = 5 - 780$)



(c) *Expo* ($n = 16, S = 4 - 1226$) (d) *Uniform* ($n = 10, S = 2 - 551$)

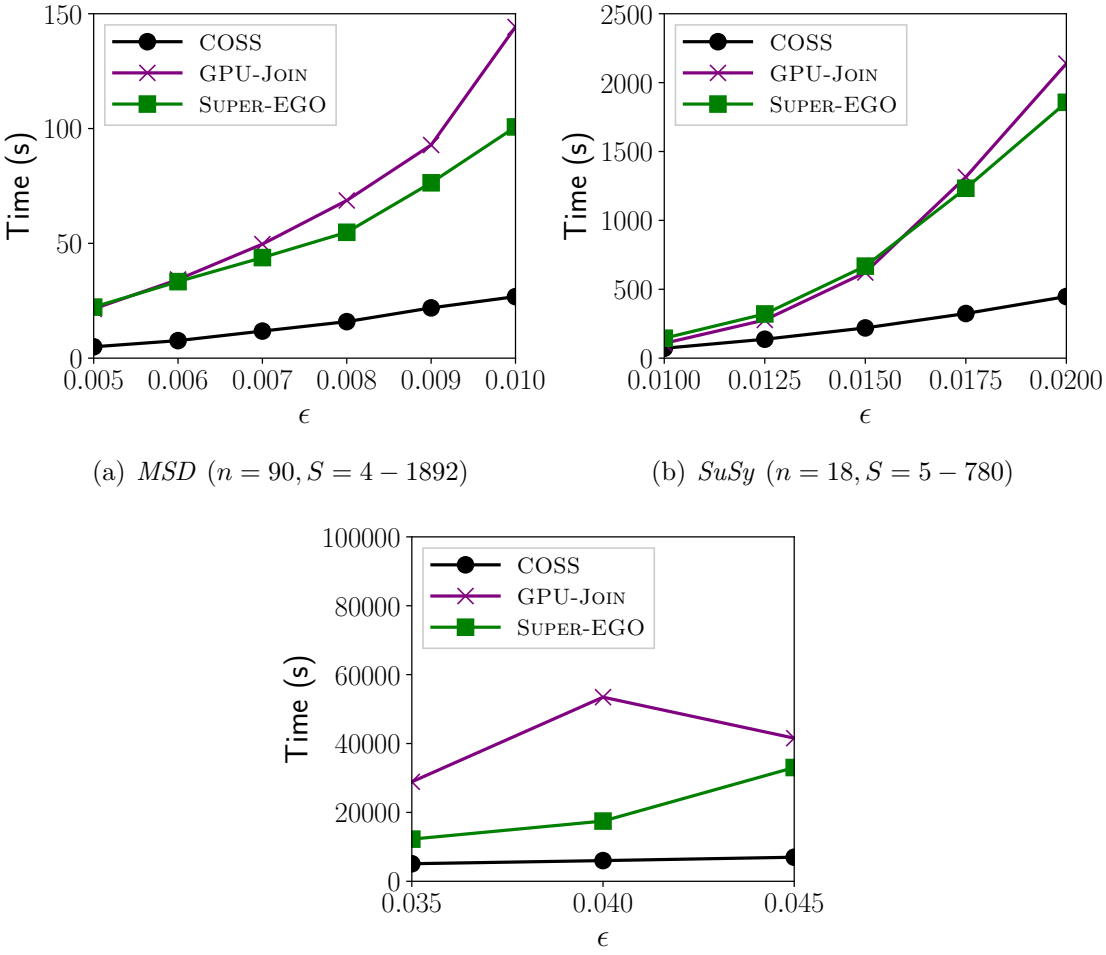
Figure 3.10: Runtime (s) vs. number of reference points.

3.7.5.1 Real World Datasets

The real-world datasets (*MSD*, *SuSy* and *Higgs*) are used to compare the performance of COSS with Super-EGO and GPU-Join. The datasets dimensions' span $n = 18 - 90$ and are evaluated on a large range of search distances and selectivity values.

***MSD* Dataset:** Figure 3.11(a) shows the runtime vs. ϵ on the *MSD* dataset. The performance of COSS degrades gracefully with increasing ϵ. COSS significantly outperforms the reference implementations. We find that COSS has a speedup of up to 5.38× and 3.76× over GPU-Join and Super-EGO, respectively.

***SuSy* Dataset:** In Figure 3.11(b) we plot the runtime vs ϵ on the *SuSy* dataset. From the



(a) *MSD* ($n = 90, S = 4 - 1892$)

(b) *SuSy* ($n = 18, S = 5 - 780$)

(c) *Higgs* ($n = 28, S = 5 - 91$)

Figure 3.11: Runtime (s) vs. ϵ on real-world datasets. Comparing COSS, GPU-Join, and Super-EGO.

plot we observe that while all three methods (COSS, GPU-Join, and Super-EGO) have similar runtimes at $\epsilon = 0.01$, both GPU-Join and Super-EGO suffer a rapid increase in runtime with the increasing ϵ values. COSS yields a speedup of up to 4.78× over GPU-Join and 4.15× over Super-EGO.

***Higgs* Dataset:** Figure 3.11(c) plots the runtime vs ϵ on the *Higgs* dataset. We observe that COSS has better performance at all ϵ values with a speedup of up to 8.85× over GPU-Join and 4.73× over Super-EGO.

3.7.5.2 Synthetic Datasets

On the synthetic datasets (*Expo* and *Uniform*), the data has the same variance in each dimension. Therefore, all three algorithms are unable to use their respective optimizations that exploit the statistical properties of the data (e.g., dimensional ordering in Section 3.6.6).

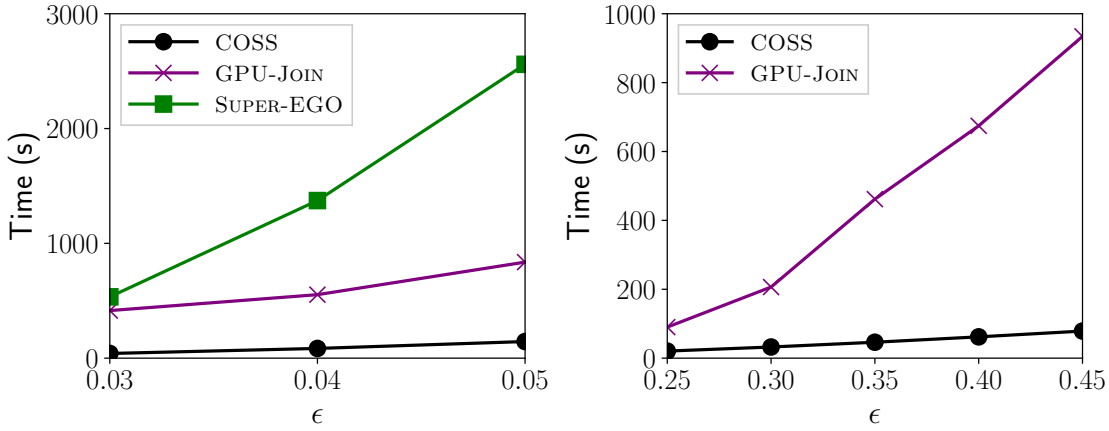
Exponentially distributed data allows the self-join to find a reasonable number of neighbors with a moderate search radius. Whereas uniformly distributed data requires a large search radius to find many neighboring points (Section 3.4.1). The selectivity yielded by *Expo* is more similar to real-world data distributions than *Uniform*.

Exponentially Distributed Data: Figure 3.12(a) plots the runtime vs. ϵ on the *Expo* datasets. From the figure, we observe that COSS significantly outperforms both GPU-JOIN and SUPER-EGO. For example, at $\epsilon = 0.05$, we obtain a speedup of 5.78× and 17.69×, over GPU-JOIN and SUPER-EGO, respectively.

Grid Killer – Uniformly Distributed Data: Figure 3.12(b) plots the runtime vs. ϵ on the *Uniform* dataset. Note that SUPER-EGO failed to execute on this dataset. COSS achieves a speedup of 11.8× over GPU-JOIN at $\epsilon = 0.45$. From the figure we observe that while the performance of COSS degrades gracefully with increasing ϵ, the pruning efficiency of GPU-JOIN decreases rapidly.

As described in Section 3.4.1 on uniformly distributed datasets, to achieve a reasonable average number of neighbors per point, ϵ needs to be sufficiently large. In Figure 3.11(b), GPU-JOIN has 4, 4, 3, 3, and 3 cells in each indexed dimension at $\epsilon = 0.25$, 0.30, 0.35, 0.40, and 0.45, respectively. Consequently, the grid used in GPU-JOIN is unable to prune a large fraction of the points, and the algorithm approaches the brute force quadratic complexity. For example, in the worst case, if there are 3 cells in each dimension, then a point located in the center of the grid is compared to all $|D|$ points in the dataset. Similarly, all multidimensional data access methods that directly index on the coordinates of the input data will suffer from the curse of dimensionality.

As discussed in Kalashnikov [62] (SUPER-EGO), when the search distance exceeds half



(a) *Expo* ($n = 16, S = 4 - 1226$) (b) *Uniform* ($n = 10, S = 2 - 551$)

Figure 3.12: Runtime (s) vs. ϵ on synthetic datasets. Comparing COSS, GPU-Join, and Super-EGO.

Table 3.2: Average speedup of COSS over GPU-Join and Super-EGO across all values of ϵ in Section 3.7.5.

Dataset	*MSD*	*SuSy*	*Higgs*	*Expo*	*Uniform*
GPU-Join	4.50	3.04	6.84	7.49	8.79
Super-EGO	3.88	3.08	3.35	15.66	-

of the bounding volume, the algorithm degrades to brute force. While optimizations such as short circuiting the distance calculation are able to reduce point comparison cost, only a better pruning strategy, such as that employed by COSS, is able to significantly improve performance.

3.8 Discussion and Conclusions

In this paper, we propose COSS, a GPU-efficient coordinate-oblivious similarity self-join algorithm. To our knowledge, no other indexing methods have been proposed that utilize distance space for the GPU. We summarize the performance of COSS in Table 3.2 which plots the average speedup obtained on all datasets in Table 3.1. This shows that our novel index mitigates the curse of dimensionality problem on datasets up to 90 dimensions. While the reference implementations degrade to brute force searches on uniformly distributed

data, COSS is still able to prune the search in this scenario. Overall, our index significantly outperforms the two reference implementations which index on the coordinate space.

Future work includes transforming coordinate space into distance space for other related similarity search problems, such as k-nearest neighbor searches. While we proposed two heuristics for reference point placement in this paper, a future direction is to investigate other placement strategies.

Chapter 4

Multi-Space Tree with Incremental Construction (MiSTIC)

This chapter introduces MiSTIC which is a continuation of the work in the previous chapter which introduced COSS. MiSTIC addresses two main goals. The first was to mitigate the cost of binary searches which the previous chapter highlighted as a bottleneck in performance, especially with a higher number of reference points. The second goal of MiSTIC was to address the difficulty with placing the reference points. The previous chapter introduced two reference point placement strategies which had a large difference in performance indicating that this was a potential area for improvement. MiSTIC uses an incremental approach to creating the index structure which allows for flexibility regarding reference point placement. Additionally, we sample the data and attempt to place reference points in response to the data itself instead of the bounding volume of the data space (data-aware as opposed to the previous data-oblivious approach). While this increased the complexity of MiSTIC as compared to COSS, it also increased the robustness of MiSTIC to different datasets and improved performance in almost every scenario we examined.

This work originally appeared in the reference below and has been adapted for this dissertation from its original format.

Brian Donnelly and Michael Gowanlock. Multi-Space Tree with Incremental Construction for GPU-Accelerated Range Queries. *2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 132-142, 2024.

4.1 Abstract

Performing range queries is prohibitively expensive as the dimensionality of the data increases. Indexing data structures reduce the time complexity of these searches by eliminating superfluous distance calculations. The state-of-the-art utilizes the GPU due to its high distance calculation throughput as compared to multi-core CPUs. Previous state-of-the-art indexes fall into two categories: metric- and coordinate-based indexes, both of which partition the space using different approaches. The indexes partition the space to generate a set of candidate points for a given query which are later refined by distance calculations. Popular metric-based indexes partition the data based on distances to reference points, where the placement of the reference points determines the partitioning of the data space but the effectiveness depends on the distribution of the data. In high-dimensions, coordinate-based indexes typically partition the data based on a subset of the coordinate dimensions. Regardless of the index type there is a tradeoff between index search overhead and the number of distance calculations, where increasing the number of partitions will increase the search overhead but will decrease the number of distance calculations computed. In this paper, we propose Multi-Space Tree with Incremental Construction (MiSTIC), a blended approach which uses both metric-based and coordinate-based partitioning strategies coupled with incremental index construction. We evaluate MiSTIC on 5 real-world datasets and compare performance to both a state-of-the-art metric-based index, COSS, and a state-of-the-art coordinate-based index, GDS-Join. We find that MiSTIC outperforms the state-of-the-art methods with an average speedup of 2.53× over COSS and 2.73× over GDS-Join.

4.2 Introduction

Advances in science and technology are producing quantities of data that have surpassed our analysis capabilities [100]. Range queries are an important tool that data scientists use to process large volumes of data, as they answer a fundamental question: Which objects in

a dataset are similar to my query object(s)? However, range queries are computationally expensive [3, 25], and so reducing the cost of this operation is key for extracting information from large datasets.

Indexing Multi-dimensional Data Points: Indexes are data structures that store the dataset (D) and partition the data space. The index is then *searched* which produces a candidate set of points that may be within the search distance (ϵ), this candidate set is then *refined* using distance calculations. This is the *search-and-refine* strategy and is used for efficiently querying large datasets [63, 80, 85].

Increasing data dimensionality necessitates a corresponding increase in the search distance because the data space grows exponentially with dimensionality [124]. For a uniformly distributed dataset of a fixed size, an increase in the dimensionality will result in an exponentially larger separation between points, thus the search distance must be increased to find nearby points. This problem with high-dimensional spaces has been termed the *Curse of Dimensionality* [14], where an index may be completely ineffective at pruning searches and degrade into a brute force search (i.e., all of D will need to be examined for a given search).

Coordinate- vs. Metric-based Indexes: Coordinate-based indexes directly use the coordinates of each point in a dataset for partitioning (e.g., a canonical index is a kd-tree [10]). Using the kd-tree as an example, as the search distance increases, an increasing number of partitions in the tree will need to be examined, and in the extreme case, the entire kd-tree will need to be searched, thus degrading into a brute force search. To address this problem, a metric-based indexing strategy should be used instead [28, 80, 85].

A metric-based approach uses a contractive mapping function to embed the dataset into a lower dimensional space [28]. For a metric-based index, the contractive mapping uses distances from points in a dataset to a set of reference points. These distances are the new coordinates in the mapped space. Contractive mapping guarantees that the distances between points in the dataset only decrease so all points within the search radius are obtained and there is no accuracy loss.

Metric-based indexes maintain effectiveness as the distance threshold increases because each coordinate in the mapped space uses all of the coordinate information in the original data space. This creates more partitions which still allow for pruning the search in instances where coordinate-based methods degrade to brute force.

GPU-Acceleration and Distance Calculations: The total work computed is proportional to the search distance (ϵ). Also, increasing the data dimensionality will increase the cost of each individual distance calculation. Higher cost distance calculations incentivize more aggressive index partitioning tailored to each query. To this end, we propose an index with ϵ-width partitions where the index construction cost is offset by a substantial increase in performance by decreasing the number of distance calculations. In terms of peak performance, GPU hardware has exceeded the capacity of multi-core CPUs. Range queries are an excellent algorithm for GPU acceleration for the following reasons: (*i*) the algorithm is throughput-oriented, as we are interested in computing a batch of range queries; (*ii*) each query point can be computed independently by one or more threads, although this leads to other issues regarding the Single Instruction Multiple Thread (SIMT) architecture; and, (*iii*) the GPU has superior distance calculation throughput compared to the CPU. For these reasons, with the exception of small workloads, the GPU outperforms multi-core CPU range query algorithms [35, 45, 47].

Drawbacks and Contributions: We outline several drawbacks of prior work in this area (**D1-3**):

D1 There is a vast quantity of work on the CPU outlining efficient indexes but many of those structures, particularly trees, do not perform well on the GPU.

D2 Due to the *curse of dimensionality* problem outlined above, some areas of research have instead focused their attention on approximate range queries [61, 122], which avoids many of the problems associated with searching high dimensional datasets. However, they do not return an exact result, which is often required in scientific and engineering domains.

D3 Previous indexes have used either a metric- or coordinate-based approach, leading to indexes tailored to dataset characteristics which reduce the overall robustness of the methods.

We address the drawbacks with contributions **C1-6**:

C1 We propose a novel multi-space index, the Multi-Space Tree with Incremental Construction (MiSTIC), which combines metric- and coordinate-based approaches which are more robust than using a single indexing type.

C2 The index uses incremental construction to increase the pruning efficiency of the index when coupled with a heuristic for determining the effectiveness of candidate partitions.

C3 We propose a new reference point placement strategy that exploits dataset characteristics, yielding good partitioning.

C4 The index exploits several facets of GPU architecture including good locality and caching behavior and uses instruction level parallelism (ILP) to hide accesses to global memory.

C5 We compare MiSTIC to one metric- and one coordinate-based GPU reference implementation on five real-world datasets. We show that MiSTIC is robust to different dataset characteristics and consistently outperforms the state-of-the-art methods COSS and GDS-Join with a speedup of 2.53× and 2.73×, respectively.

C6 Contrary to other work, we find that minimizing distance calculations does not necessarily lead to the best performance, rather load balancing may be more important.

This paper is organized as follows: Section 4.3 outlines the problem statement and related work. Section 4.4 presents MiSTIC and associated optimizations. Section 4.5 presents the experimental evaluation. Lastly, Section 4.6 concludes the work.

4.3 Background

4.3.1 Problem Statement

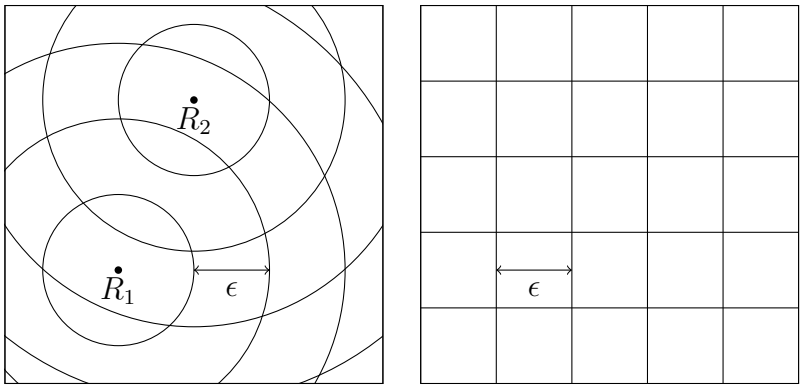
We define a dataset, D, which contains $|D|$ points (or feature vectors), where each point has n dimensions. Each point is denoted as $x_i \in D$ where $i = 1, 2, \ldots, |D|$. Each point, x_i, is defined by a set of coordinate values in n dimensions denoted as $\{x_i^1, x_i^2, \ldots, x_i^n\}$. A range query search finds all of the $x_i \in D$ which are within a distance threshold, ϵ, of a query point.

In a self-join operation all of the data points in D are compared to each other. The operation returns $\{q_1, q_2, \ldots, q_i\}$, where q_i contains the points in D which are within ϵ of x_i. The total number of returned points, $|Q| = \sum_{i=1}^{|D|} |q_i|$, is most often greater than $|D|$ such that the memory required to store the results from a self-join query exceeds the memory capacity of a GPU, requiring batched computations where intermittent results are transferred back to the host. In the case that D also exceeds the memory capacity of a GPU, the batched computations will only transfer a partition of D which is required for that batch of computations. This enables the self-join to be computed on the GPU regardless of $|Q|$ and $|D|$ and the global memory capacity of a particular GPU model.

We define the Euclidean distance between two points, a and b as $dist(a, b) = \sqrt{\sum_{j=1}^{n}(a^j - b^j)^2}$ and we add a result to the result set when $dist(a, b) \leq \epsilon$. We use the Euclidean distance because it is the standard distance metric that is employed in the literature [35, 45, 47, 62].

4.3.2 Index Supported Range Queries

A distance similarity self-join is straightforward to implement using a nested loop, and when including the dimensionality of the data, n, it has a time complexity of $O(n \cdot |D|^2)$. For even moderately sized datasets, the quadratic time complexity becomes an intractable problem, particularly in high dimensions [14]. To improve performance, indexing methods have been developed to reduce the number of distance calculations needed at the cost of



(a) Metric-Based (b) Coordinate-Based

Figure 4.1: (a) An example of a metric-based index (similar to COSS [35]) partitioning a two-dimensional space with two reference points R_1, and R_2. (b) An example of partitioning with a grid (similar to GDS-Join [48]) in two dimensions where each of the dimensions are used for indexing. Both methods use the triangle inequality to exclude points in non-adjacent ϵ-width partitions from the search.

preprocessing overhead [19]. These methods prune the search space so that points only calculate the distance to a subset of other points that are nearby in the data space. This allows range queries to be performed on larger datasets by reducing the total computational cost.

Most indexes use either a grid or a tree structure to store the partitions of the index [14, 35, 55]. Trees create a hierarchical structure where nodes in the tree are decomposed into smaller subsequent nodes. In contrast, a grid typically partitions data using axis-aligned regions [48, 63, 64]. Both trees and grids can be data-agnostic, where the index is constructed statically without using information about the data points (i.e., only using the bounding volume), or they can be data-aware and use the data points during index construction, as exemplified by a kd-tree [10].

As shown in Figure 4.1(a), metric-based indexes (also referred to as distance-space, pivot-point, or coordinate-oblivious indexes), use a set of points in the same space as the data as references for partitioning the dataset [15, 55, 80]. These methods tend to have better performance than coordinate-based indexes in high-dimensional spaces because they incorporate every coordinate into a single mapped dimension. Metric-based indexes create indexing

structures on a transformed space [14]. Pruning the search occurs in this transformed space but the distance calculations occur in the original data space. While metric-based indexes have proven to be effective in high-dimensional spaces, their additional complexity leads to higher overheads that make them less attractive for low dimensional searches [35].

The time complexity of range queries using an index (such as an R*-tree, kd-tree, or MiSTIC) is an open problem. The number of distance calculations for the self-join operation has an upper bound of $O(|D|^2)$ and a lower bound of $O(|D|)$. Because the time complexity is a function of the search radius, ϵ, and is unconstrained, it is possible that all points will need to be compared to each other. There has been some discussion in the literature regarding practical values of the search radius and the lack of a more robust complexity analysis [46, 103].

4.3.3 State-of-the-art & Reference Implementations

We outline the range query algorithms from the literature that we will compare with MiSTIC in Section 4.5. As described in Section 4.2, recent advances in GPU hardware have surpassed the capabilities of multi-core CPUs and so range queries are best carried out on the GPU. Consequently, we do not compare to any multi-core CPU algorithms, as they are typically only advantageous on small workloads for which GPU acceleration is unwarranted as illustrated by previous work [45].

We compare our work to COSS [35] and GDS-Join [49] which are GPU-accelerated range query algorithms that use metric- and coordinate-based indexes, respectively. We summarize the algorithms as follows and note that the authors have made their code publicly available such that we can make a comparison to their work[1]. We also compare to a highly optimized brute force implementation, Brute, which is a baseline for comparison.

GDS-Join performs range queries using the GPU [47, 48, 49]. It constructs a compact coordinate-based grid index on the data as shown in Figure 4.1(b). The algorithm includes

[1] `https://github.com/bwd29/Coordinate-Oblivious-Similarity-Search/` and `https://github.com/mgowanlock/gpu_self_join/`.

several optimizations, such as reordering the sequence in which query points are processed to limit load imbalance within warps[2]. It indexes the data in $r < n$ dimensions to reach a trade-off between index search overhead and the number of points that need to be refined using distance calculations. The distance calculation kernel takes advantage of the GPU's instruction level parallelism to hide global memory access latency [111]. We compare MiSTIC to GDS-Join as it is a state-of-the-art coordinate-based index and is therefore suitable for different dataset properties than metric-based indexes.

COSS or Coordinate-Oblivious Index for Similarity Searches, is a metric-based index, designed for GPU acceleration and partitions the space as shown in Figure 4.1(a). The index design was motivated by the drawbacks of coordinate-based grid indexes to be better for high-dimensional range queries [35].

Brute is a brute-force implementation we created for comparison which is highly optimized to use coalesced memory access patterns, good locality to increase cache hits, and reorders the dimensions of the data based on variance to increase the efficiency of short circuiting the distance calculations (which has a substantial impact on brute-force algorithms). Brute lacks index construction overhead allowing it to outperform indexing methods on small datasets.

4.3.4 Limitations of the State-of-the-Art

We show in Section 4.4 that MiSTIC addresses several limitations of the state-of-the-art methods, which include:

- COSS and GDS-Join search the index to find non-empty partitions on the GPU using binary searches instead of using tree traversals. A drawback of this is that a single binary search has to perform $\log_2(|G|)$ accesses to global memory, where $|G|$ is the number of non-empty partitions in the index (empty space is not indexed to limit global memory usage). In contrast, a tree traversal aborts early when children do not exist, which typically results

[2]Warps are groups of 32 threads on the GPU that execute the same instruction in lockstep. Throughout this paper, we use CUDA terminology, but the concepts are the same across GPUs from different vendors.

in fewer accesses to global memory compared to binary searches.

- The reference point placement in COSS and the dimension selection in GDS-Join determines how the partitions are generated; however, the strategies are static. MiSTIC addresses this limitation by using incremental index construction to examine several sets of candidate partitions that when combined together produce an efficient index structure that prunes the search better than a static method.
- Metric- and coordinate-based indexes have fundamentally different approaches with performance dependent on different dataset characteristics [49]. This necessitates the selection of an indexing method based on dataset characteristics, like intrinsic dimensionality, which are non-trivial to discover. MiSTIC merges the two approaches to yield an algorithm which dynamically adapts to the dataset without foreknowledge of dataset characteristics. This leads to performance gains over both metric- and coordinate-based indexing methods regardless of the dataset.

4.4 MiSTIC

As described in Section 4.3.4 there are three goals for MiSTIC, firstly to replace a potently expensive binary search with a tree traversal, secondly to allow for incremental construction, and finally to combine both metric- and coordinate-based partitioning strategies.

Our tree is constructed using a combination of the intersections of ϵ-width shells centered on each reference point (the metric-based method) and a grid of ϵ-width axis-aligned cells (the coordinate-based method) which create partitions. Each query requires a search over the index to identify nearby partitions containing data points. There are two methods for searching the partitions; (i) a binary search or (ii) a depth first tree traversals.

A binary search (which is used in the reference implementations) can be used on the last layer of the tree and requires $\log_2(|G|)$ memory accesses [35], where $|G|$ is the number of non-empty index locations (also the size of the last layer of the tree). The worst case for

a binary search is if the searched-for partition is empty and therefore not in the array of non-empty partitions, as this requires a full search of the array.

A depth first tree traversal will have a maximum of r memory accesses into the data structure, where r is the number of layers of the tree representing partitions which are either metric- or coordinate-based. The best case for a tree traversal is if the searched-for partition is empty because the search terminates when a branch of the tree has no partitions. This is the opposite of the binary search and results in the tree traversals performing fewer memory access when there are more empty partitions than non-empty. Additionally, tree traversals require fewer memory accesses as compared to a binary search when $\log_2(|G|) > r$ which occurs when the index partitions the data effectively (i.e., the data is separated into numerous partitions which yields good pruning). The number of non-empty partitions of ϵ-width is dependent on the ϵ used in the search, so large ϵ values result in fewer partitions and may therefore be more efficient using binary searches.

In Figure 4.1, an example of how MISTIC partitions a two-dimensional coordinate space is shown. The non-empty partitions that are created by the overlapping shells from the reference points or the non-empty cells in the grid become leaves on the tree. In high dimensional space the number of partitions will increase to such a level that it becomes intractable to store each partition, therefore only the non-empty partitions are used in the last layer of the tree (as shown by L_r in Figure 4.2). The non-terminal layers of the tree keep track of the empty shells generated by that layer's reference point, but those nodes on the tree do not have any children nodes, which allows for depth first searches to terminate early.

As ϵ increases, the width of the shells and cells also increase resulting in fewer and larger partitions. While a high dimensional space will have more partitions for a given ϵ, in order to have a practical query the ϵ value will have to increase as the dimensionality increases. This leads to a small subset of partitions which contain the majority of the dataset. This is offset by effective partitioning of the data using a combination of metric- and coordinate-based

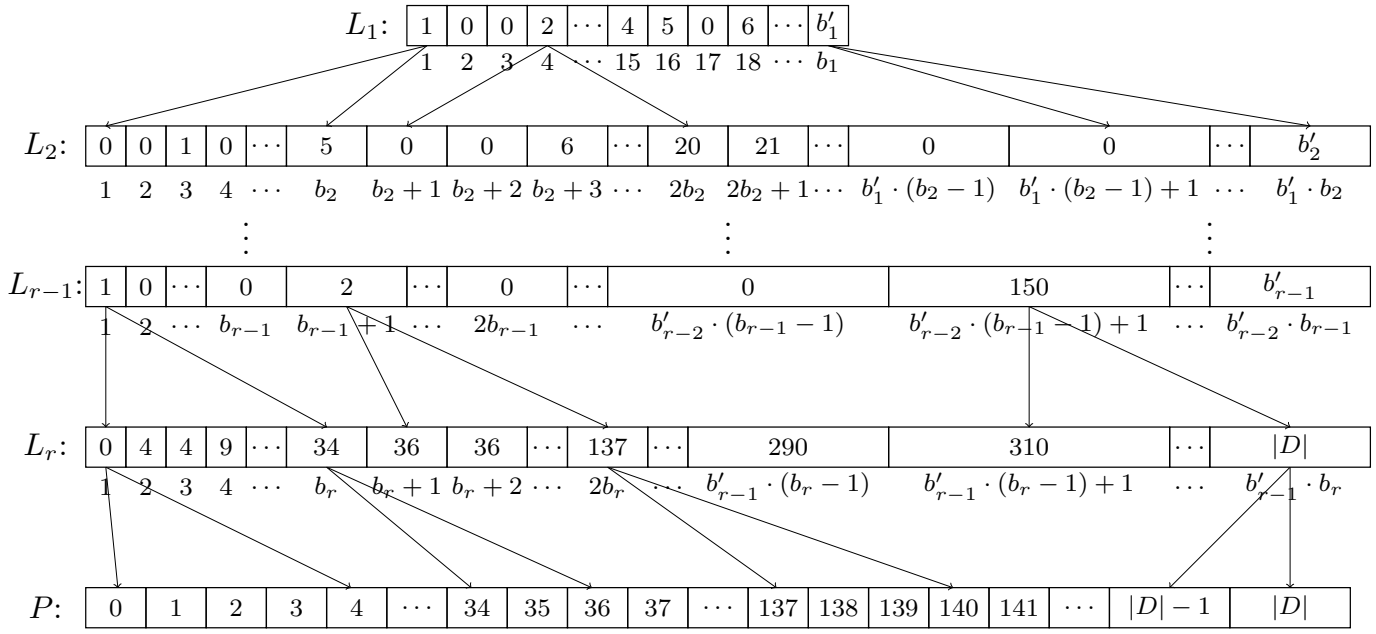


Figure 4.2: Tree indexing example, where L is the tree structure with r layers/reference points. b_x is the maximum partition range in that layer, b'_x is the number of non-empty partitions where x is the layer number. P is the point array and $|D|$ is the number of points in the dataset.

partitioning strategies.

4.4.1 Tree Structure

We outline the tree structure of MiSTIC which contains both metric-based partitions created with reference points and coordinate-based partitions created by partitioning dimensions in the coordinate space (see Figure 4.1). The tree construction is performed on the CPU while the final data structure is transferred to the GPU for searching and distance calculations.

First Layer – Figure 4.2 shows the structure of an example tree with r reference points/indexed dimensions. The tree has one layer, L, for every reference point or indexed dimension. The first layer of the tree, L_1, is constructed by first calculating the maximum distance from the first reference point to all of the points or the maximum coordinate value of all of the points in the data. This maximum value is used to find the total number of possible partitions for that layer, b_1, by dividing the maximum value by the distance threshold ϵ used in the search. An array of size b_1 is then allocated. The layer L_1 is then populated with an incrementing counter and zeroes to represent non-empty and empty partitions, respectively. A partition is non-empty if the distance from a point to the reference point associated with L_1 falls within the range of a partition or if the coordinate values of a point for an indexed dimension falls within that range. For the first layer, the index of a partition in the array is the distance from the reference point or the coordinate value divided by ϵ. This is not true for all subsequent layers. b'_1 is the number of non-empty partitions in L_1, and is the sum of the array L_1 and $b'_1 \leq b_1$ since there cannot be more non-empty partitions than the total number of potential partitions.

The values in layers L_1 through L_{r-1} represent the current non-empty partition count such that the value in L_{x-1} at a non-empty index will point into the next layer L_x, where x is the layer number. The value does not directly correspond to the index in the next layer but is rather a multiple of the range of partitions for the reference point/indexed dimension associated with the next layer.

Middle Layers – Subsequent layers after L_1 are generated using the previous layer's values for b_x and b'_x. L_x allocates $b'_{x-1} \cdot b_x$ partitions. The idea is to have a section of L_x which has b_x partitions for each non-empty partition in the previous layer L_{x-1}. To populate L_x, each point is compared to the reference point associated with L_x, and this will yield the partition number that will be non-empty based on the previous layer's partition for that point. A point in partition y in the previous layer L_{x-1} and in partition z of layer L_x will be in the index $y \cdot b_x + z$. This index will be switched to the current non-empty partition count from

zero if no other point was found in that partition previously, otherwise the partition will have already been set, and no update is required.

Final Layer – The last layer of the tree, L_r, is unique as it contains the counts of how many points are in each partition of the final layer. The layer is constructed similarly to the previous layer but instead of only noting if the partition is non-empty, a count is incremented in that partition every time a point is located there, and all subsequent partitions are also incremented. This creates a mapping from L_r to P where each value in L_r points to a starting location in P which contains the first point in that non-empty partition. The points in P which belong to that partition are found by accessing the next starting location in L_r which denotes where the next partition begins. The final partition in layer L_r will have a maximum range $|D|$ which corresponds to the size of the dataset D, and will be the maximum index of array P.

4.4.1.1 Partitioning Strategy

The placement of the reference points (metric-based layers) and the selected dimensions for indexing (coordinate-based layers) has a significant impact on the performance of the algorithm. However, finding the optimal reference point placement or selection of dimensions to index that minimize the total number of distance calculations, is intractable. Many attempts have been made to try to model reference point placement strategies [20, 29, 30, 80] and determine which dimensions to index [49] but because of the complexity of the problem and the dependency on the data distribution there is no single best solution. MISTIC chooses metric- or coordinate-partitions using a partitioning strategy and evaluates them using a heuristic. Selecting reference points and indexed dimensions to consider for tree construction is carried out with three different partitioning strategies (PS):

PS_1: We give the intuition of an approach that examines the variance of points within partitions. Consider that the greater the variance, the better pruning a reference point should have in general because the points will be separated into more partitions which

reduces the total number of distance calculations. To this end, we select initial reference points from a random distribution of values to create a reference point set, R', containing q reference points $R_1, R_2, \ldots, R_q$. The initial set of randomly generated reference points R' are then evaluated using a sample of the data set to find the variance in the distances between R' and a subset of data points, C or $\delta = S^2 = \frac{\sum_{a=1}^{a=|C|}(dist(R'_a, C_a) - \overline{X})^2}{|C| - 1}$, where $\overline{X}$ is the mean of distances between R'_a and all points in C.

PS_2: The reference points are generated using the strategy outlined in previous work [35] which places the reference points around the outside of the data. This is effective at reducing partitions that are adjacent in the index but are spatially distant in the original coordinate-space.

PS_3: We select the dimension to index for the coordinate-based partitions based on the variance of each dimension. Previous work has shown that the dimensions with the highest variance are the best dimensions to construct an index [47, 48]. We only evaluate the $r = 6$ highest variance dimensions for each layer to reduce the amount of work needed for index construction. This corresponds to the dimensions used for indexing in the reference implementation, GDS-Join.

Regardless of the strategy used, in the best case the points will be evenly spread throughout the partitions associated with R'_x and in the worst case they will be distributed into only a few partitions. The variance of R'_x with respect to C is a good indication of how clustered the points will be into the partitions, with higher variance correlated with more evenly distributed points.

4.4.2 Incremental Tree Construction

To construct the tree index incrementally as described by Algorithm 2, the procedure takes as input the dataset (D), the distance threshold (ϵ), an array of reference points or dimensions to index R' (generated as described in Section 4.4.1.1) and the number of layers of the tree (r), as shown on line 1. The algorithm constructs each layer of the tree on the

Algorithm 2 Incremental tree construction.

```
1: procedure TreeConstruction(D, ϵ, R′, r)
2:     for i ∈ {1, 2, ..., r − 1, r} do
3:         for j ∈ {1, 2, ..., |R′| − 1, |R′|} do
4:             M ← ParValue(R′_j, D)
5:             b_i ← ⌈(max(M)/ϵ)⌉ · b′_{i−1}
6:             for k ∈ {1, 2, ..., |D| − 1, |D|} do
7:                 ParNum[j, k] ← ⌊(M[j, k]/ϵ)⌋
8:                 Ofs ← (Par[i, ParOfs[i − 1, k]] − 1) · b_i
9:                 ParOfs[i, k] ← Ofs + ParNumbers[j, k]
10:                if ParCnts[i, ParOfs[j, k]] = 0 then
11:                    b′_i ← b′_i + 1
12:                ParCnts[i, ParOfs] ← ParCnts[i, ParOfs] + 1
13:            Heuristic[j] ← CalcHeuristic(ParCnts)
14:        R_i ← R′[minimum(Heuristic)]
15:        Count ← 0
16:        for j ∈ {1, 2, ..., b_i − 1, b_i} do
17:            if ParCnts[i, j] ≠ 0 then
18:                Count ← Count + 1
19:                L_i[j] ← Count
20:            else
21:                L_i[j] ← 0
22:    Sum ← 0
23:    for i ∈ {1, 2, ..., b_r − 1, b_r} do
24:        Sum ← Sum + L_r[i]
25:        L_r[i] ← Sum
26:    Return L
```

CPU starting with layer L_1 and proceeding to layer L_r in a loop starting on line 2. For each layer of the tree, each potential reference point or indexed dimension in R' needs to generate a layer L_i (line 3). To generate a layer for a given *reference point* in R' first the distance to all the points in D is used to generate a distance vector M with the function $ParValue()$ on line 4. To generate a layer for an *indexed dimension* in R', $ParValue()$ will return the coordinate value for each point which matches the dimension being indexed. The total number of partitions for that layer b_i for a given reference point R'_j are calculated by finding the maximum value in M, dividing it by ϵ, and multiplying it by the previous layer's non-empty bins, b'_{i-1} (line 5).

Each potential layer needs to iterate through every point in D to find: (i) the partition number, $ParNum$, from the floor of the distance from D_k to R'_j divided by ϵ (line 7); (ii) the offsets, Ofs (line 8), of the point from the beginning of L_i which is found from the point's

previous partition in L_{i-1} multiplied by the b_i value calculated on line 5; (iii) the offset into the layer for each partition, $ParOfs$, is the previously calculated Ofs added to the $ParNum$ on line 9; (iv) the partition counters, $ParCnts$, which tracks how many points are in each partition for a given layer L_i. If a partition goes from empty to non-empty (line 10), b'_i is incremented to track the non-empty partitions for L_i (line 11).

After each point has been assigned a partition, a heuristic for that potential layer is calculated using $ParCnts$ on line 13. The best layer for L_i is selected based on which potential reference points or indexed dimension in R' generated the lowest heuristic value (line 14). Once R_i has been selected, layer L_i is constructed as outlined in Section 4.4.1 (lines 15 – 21). The final layer, L_r is a special case and needs to have a running total to track the number of points in each partition which are found by keeping a running total of the values in the partitions of L_i when $i = r$. This replaces the values in the array with the running total as shown on lines 22 – 25.

Parallel Incremental Construction – Constructing the tree incrementally (see Algorithm 2) requires that each layer be constructed $|R'|$ times. While this increases the amount of work needed to construct the tree, each layer of the tree is dependent on the previous layer and the amount of parallelization is limited when statically constructing the tree. When constructing the tree incrementally, each layer must be evaluated for each potential reference point or indexed dimension. This allows for parallelization such that a thread is assigned to generate each potential layer on line 3. The overhead from evaluating potential layers of the tree is mostly hidden with concurrent construction, resulting in a negligible increase in the overall construction time.

Heuristic for Incremental Construction – When constructing the tree incrementally there needs to be a heuristic to determine which potential layer will lead to the best performance. Without a good heuristic, the tree may select a given layer which may be effective on an individual basis, but not when considering overlapping partitions with the other layers. This would increase both the construction overhead and search time while failing to offset

Algorithm 3 Searching the tree.

```
1: procedure TreeSearch(A, B, r, I)
2:     if I[A[0]] = 0 then
3:         return False
4:     B_Total ← B[0]
5:     Loc ← A[1] + B[0] + (I[A[0]] − 1)
6:     for i ∈ {1, 2, ..., r − 1} do
7:         if I[Loc] = 0 then
8:             return False
9:         B_Total ← B_Total + B[i]
10:        Loc ← B_Total + (I[Loc] − 1) + A[i + 1]
11:    if I[Loc] = I[Loc − 1] then
12:        return False
13:    return (I[Loc − 1], I[Loc])
```

these costs with increased distance calculation pruning. We define StdDev as the standard deviation which is calculated as follows: $\sqrt{\sum_{i=0}^{b_r} |L_r[i] - \overline{X}|^2/|D|}$, where $\overline{X}$ is the average number of points in each partition of the layer, L_r. We select the layer that has the lowest standard deviation to minimize the difference in the number of points between partitions. We examined several heuristics but found that StdDev outperformed them, so we omit describing them.

4.4.3 Searching the Tree

While tree construction occurs on the CPU, searching the tree occurs on the GPU. The search requires four inputs (line 1 in Algorithm 3). A, B, r, and I refer to an array indicating a partition to find, an array of the size of each layer of the tree, the number of layers, and an array that stores all of the tree layers in adjacent memory as $L_1, L_2, \ldots, L_{r-1}, L_r$, respectively. A, has r values each representing an index in I which corresponds to a partition adjacent to the partition of the point that is initiating the search. For each layer of the tree as depicted in Figure 4.2, the search finds if the value of I corresponding to partition indicated by A that matches the layer to be searched is zero, which indicates an empty partition and terminates the search.

In Algorithm 3, the search starts at the first layer on line 2 which evaluates if the adjacent partition is empty and terminates the search if true. If not, then the location of the adjacent

partition for the next layer is calculated on line 5. The algorithm also starts a counter for the total partition sizes B_{Total} on line 4 which will be used for determining the offset into array I.

On line 6 the algorithm enters into a loop to evaluate if the adjacent partition continues to be non-empty (line 7) then calculates the offset (line 9) and location (line 10) for the next layer of the tree. If the adjacent partition is found to be empty for any layer (including the last on line 11 which is an exception because L_r stores the location of points in P, so an empty partition is indicated by no change in the value compared to the previous index) then the search terminates, otherwise it returns the lower and upper bound of points within the adjacent partition on line 13. These bounds correspond to points in the array P (as shown in Figure 4.2) and define the candidates for the query point for that particular adjacent partition indicated by A.

Binary Searches – Instead of using tree traversals, MiSTIC can be configured to use a binary search to locate non-empty partitions in the final layer of the tree. Since the last layer of the tree is sorted we represent it with a linear (one dimensional) ID which a binary search uses to find a specific non-empty partition. As stated in Section 4.3.4, binary searches have drawbacks compared to tree traversals, especially while searching a large number of non-empty partitions. We investigate MiSTIC with both search methods and evaluate their impact on performance.

4.4.4 Other Optimizations

While the main contribution of this paper is our efficient index, MiSTIC, we outline several other optimizations that are needed to ensure that we do not overflow the result set buffer on the GPU, and perform efficient distance calculations.

4.4.4.1 Batching the Computations

The distance calculation kernels need to be batched because of GPU memory limitations. Most result sets, Q, from a self-join operation on a large dataset will need more memory than is available on the GPU to store $|Q|$ pairs of points. The number of batches needed depends on the number of points in the dataset and ϵ, with a larger number of points (or ϵ) requiring a larger number of batches to compute. The limiting factor for how many points are computed in each batch is the global memory needed to store the result set for each batch. Between each batch/kernel invocation, the result set (Q) is sorted and transferred to the host, freeing space for the next batch. The global memory on the GPU is allocated once and data is transferred to the host using a pre-pinned memory buffer to reduce the overhead due to data transfers over PCIe.

4.4.4.2 Instruction Level Parallelism and Short Circuiting

The kernel utilizes instruction level parallelism (ILP) for higher computation throughput by hiding accesses to global memory. This is done by unrolling iterations of the loop which computes the distances between points. The number of iterations unrolled limits the amount of ILP that is possible. We experimentally determined that unrolling by four loop iterations had the best performance with MiSTIC on the platform and so the parameter is fixed to four in the experimental evaluation.

In addition to using ILP for computing distance calculations, we also allow for the distance calculations to short circuit and abort early. We use a variable to keep a running total of the distance accumulated, and with ILP this total gets updated every four dimensions. At each update we terminate the distance computation if it has exceeded ϵ. This reduces the amount of work needed to compute distance calculations between points that are far away from each other.

4.5 Experimental Evaluation

4.5.1 Evaluation Platform

Experiments are executed on a platform with 2× AMD EPYC 7542 CPUs (64 total physical cores) clocked at 2.9 GHz with 512 GiB of main memory and a 40 GiB NVIDIA A100 GPU. The code uses CUDA v.11 and is compiled with the O3 optimization flag. Multi-threaded CPU code is parallelized using the OpenMP library.

4.5.2 Implementation Configurations

In the results, we report the response time and variant metrics, such as speedup. The total response time (or end-to-end time) excludes loading the dataset into main memory as this is the same for all experiments, and includes all data transfers to/from the GPU, and host-side overheads such as storing and organizing the final result set in main memory. Each method uses batching on the GPU to allow for result sets that exceed the GPU's memory capacity, which is a necessity for moderately large datasets. All indexing methods are configured to use $r = 6$ reference points/indexed dimensions. While this will not always be the optimal configuration across all datasets, it allows for a fair comparison across all methods. Similarly, prior work showed that indexing in $r = 6$ dimensions (for coordinate-based indexes) and using $r = 6$ reference points (for metric-based indexes) was found to yield good performance on a range of workloads [35, 48, 49].

MiSTIC: We use a kernel block size of 256 with 1024 blocks per kernel for 262,144 threads per kernel invocation. MiSTIC uses $r = 6$ layers, with 38 potential layers (16 reference points using PS_1, 16 reference points using PS_2 and 6 indexed dimensions using PS_3) per layer as discussed in Section 4.4.1.1. The code repository used for evaluation is available to the public [3].

COSS: We experimentally found that a kernel block size of 1024 with 128 blocks per kernel

[3]Code available at https://github.com/bwd29/self-join-MiSTIC

Table 4.1: The five real-world datasets (normalized to the range $[0, 1]$) that we use in our evaluation where the data dimensionality n, intrinsic dimensionality i (rounded) and dataset size $|D|$ are shown along with the values of ϵ that correspond to the three target selectivity values.

Dataset	n	i	$\lvert D\rvert$	S_s	S_m	S_l
Wave [97]	49	15	287,999	0.0054	0.00702	0.008358
MSD [13]	90	29	515,345	0.0076	0.00913	0.011334
Bigcross [1]	57	3	11,620,300	0.0131	0.01994	0.0281
SuSy [6]	18	9	5,000,000	0.01703	0.02078	0.025555
Higgs [6]	28	19	11,000,000	0.049186	0.05558	0.063117

launch (131,072 threads total) with 2 concurrent GPU streams has the best performance.

GDS-Join: A kernel block size of 32 with a dynamic number of blocks per launch with 3 concurrent GPU streams was found to have the best performance on our platform.

Brute: We configured BRUTE to run with a kernel block size of 256 with 512 blocks per kernel launch for a total of 131,072 threads per launch, which was found to have the best performance on our platform.

4.5.3 Selection of the Search Distance ϵ

We will show how performance scales as a function of the search distance (ϵ). To compare the performance between datasets having different sizes and dimensions ($|D|$ and n), the typical convention is to use search distances that span the same range of selectivity values across all datasets. The selectivity, S, refers to the mean number of points found by all searches carried out on a dataset. For the self-join this refers to $|D|$ searches, and so the selectivity $S = (|Q| - |D|)/|D|$, where $|Q|$ is the total number of results returned. In this paper, we select three target (small, medium, and large) selectivity values, corresponding to $S_s = 2^8$, $S_m = 2^{10}$, and $S_l = 2^{12}$, respectively. These selectivity values span several orders of magnitude and so they allow us to examine how the algorithms scale with increasing search distance and thus workload.

Note that there is not an analytical method to calculate what the search distance, ϵ, should be to yield the abovementioned selectivity values because the real-world datasets

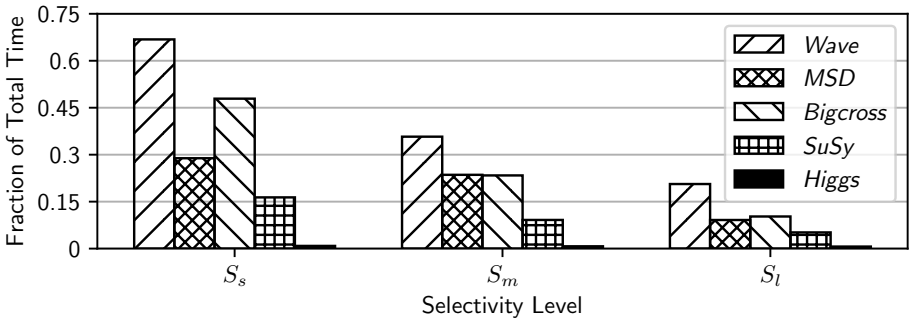


Figure 4.3: The fraction of the total response time spent constructing the tree for each of the selectivity values S_s, S_m, and S_l across the five real-world datasets for $r = 6$.

Table 4.2: The partitioning strategy (1-3) dynamically selected for each layer of MiSTIC as described in Section 4.4.1.1 for the five real-world datasets and each selectivity level.

Dataset	S_s						S_m						S_l					
	L_1	L_2	L_3	L_4	L_5	L_6	L_1	L_2	L_3	L_4	L_5	L_6	L_1	L_2	L_3	L_4	L_5	L_6
Wave	2	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1
MSD	2	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1
Bigcross	3	2	1	1	1	1	3	2	1	1	1	1	3	2	1	1	1	1
SuSy	3	1	1	1	1	1	3	1	1	1	1	1	3	2	1	1	1	1
Higgs	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

do not follow known data distributions (e.g., uniform, exponential, or normal) and so we perform a search on all datasets to first determine the ϵ values that correspond to each of the selectivity values above. All of the ϵ values yield a selectivity that is within 1% of the target selectivity values (S_s, S_m, and S_l) and are given in Table 4.1.

4.5.4 MiSTIC Performance Analysis

MiSTIC: Tree Construction – We examine the fraction of time spent on tree construction for each selectivity level in Figure 4.3. Tree construction overhead has a large impact on performance on the smaller datasets and lower selectivity levels which are correlated with lower total response times. As the number of distance calculations increase with higher selectivity levels and larger datasets, the tree construction time fraction becomes negligible. Tree construction accounts for up to 65% of the total time; however, we will show that MiSTIC maintains a competitive performance level despite the tree construction overhead. Reference implementations COSS and GDS-Join have negligible index construction overhead.

Table 4.3: Speedup of using the StdDev heuristic over the SumSqrs heuristic, where the speedup is the total response time of SumSqrs over StdDev.

Dataset	S_s	S_m	S_l
Wave	0.96	1.08	1.10
MSD	0.86	1.05	1.86
Bigcross	19.64	13.10	9.34
SuSy	3.65	2.66	2.13
Higgs	1.02	0.96	0.98

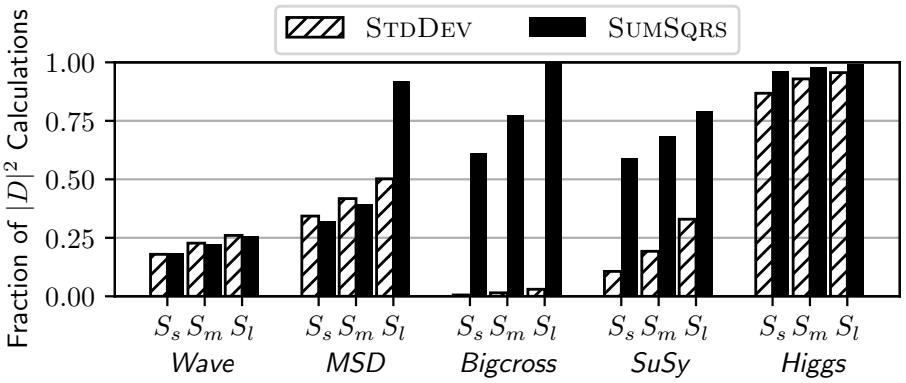


Figure 4.4: The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$.

The partitioning strategies (PS) used for each layer of the tree are given in Table 4.2. Only L_1 and L_2 did not always select our novel partitioning strategy, PS_1. L_1 uses the reference point placement strategy also used by COSS (PS_2) for *Wave* and *MSD* while MiSTIC uses PS_3 (the PS used by GDS-Join) on *Bigcross* and *SuSy* for L_1. Additionally, there are 4 cases where PS_2 is used for L_2 and *Higgs* always uses PS_1. The partitioning of the first few layers has a larger impact on overall performance than subsequent layers because of how the heuristics select subsequent layers. Even though PS_2 and PS_3 are rarely selected, they substantially improve MiSTIC's performance. Forcing MiSTIC to use a single PS reduces performance regardless of which PS is used.

Heuristic Comparison for Construction – MiSTIC is evaluated using two different heuristics for incremental construction to reduce total distance calculations as discussed in Section 4.4.2. The incremental construction evaluates 32 reference points for each layer as well as the 6 highest variance dimensions and then selects which reference point or dimension

to use for that index layer based on which had the minimum heuristic value. It should be noted that MiSTIC is a greedy algorithm and so the heuristics are not guaranteed to find the global minima.

In Figure 4.4, the fraction of $|D|^2$ distance calculations (the number of distance calculations required of a brute force approach) needed by both heuristics is plotted for each dataset and selectivity level. The number of distance calculations is an estimate for the amount of work that will be performed on the GPU; therefore, a lower number of distance calculations is correlated with higher overall performance. For all selectivity levels of the *Wave* dataset and the first two selectivity levels of the *MSD* dataset, the SumSqrs heuristic leads to fewer distance calculations. For every other dataset and selectivity level, the StdDev heuristic results in fewer calculations. The SumSqrs method fails to create a large number of partitions which yields poor performance particularly on the *Bigcross* and *SuSy* datasets. This is supported by Table 4.3 where the speedup of StdDev over SumSqrs is up to 19.64×.

The overall performance of MiSTIC with each heuristic is not solely based on the number of distance calculations however, as observed in Table 4.3 where the speedup of StdDev over SumSqrs exceeds 1 even while performing more distance calculations on S_m and S_l of *Wave* and S_m of *MSD*. This is due to the StdDev heuristic resulting in partitions which have similar number of points and therefore yielding similar workloads for the GPU threads that will search and refine the points in these partitions. On the GPU, if the amount of work between threads is unbalanced then the throughput of the device will be reduced due to the SIMT architecture. This makes it imperative to have a balanced workload and therefore an even distribution of points in the partitions is ideal for achieving the best performance on the GPU. StdDev is a better heuristic than SumSqrs because it creates more partitions resulting in fewer distance calculations and it distributes the points evenly across partitions resulting in better load balancing.

MiSTIC: Binary Search vs. Tree Traversal – In Section 4.4.3 we describe the two possible methods for searching the tree; tree traversals or binary searches. The average speedup

Table 4.4: The average speedup across selectivity levels of performing tree traversals searches over binary searches for each dataset, where speedup is the ratio of the response time for binary searches over tree traversals. The average number of non-empty partitions is included for analysis.

Dataset	Speedup	Partitions
Wave	1.05	2,152
MSD	1.02	5,537
Bigcross	1.04	21,057
SuSy	1.36	27,284
Higgs	0.96	902

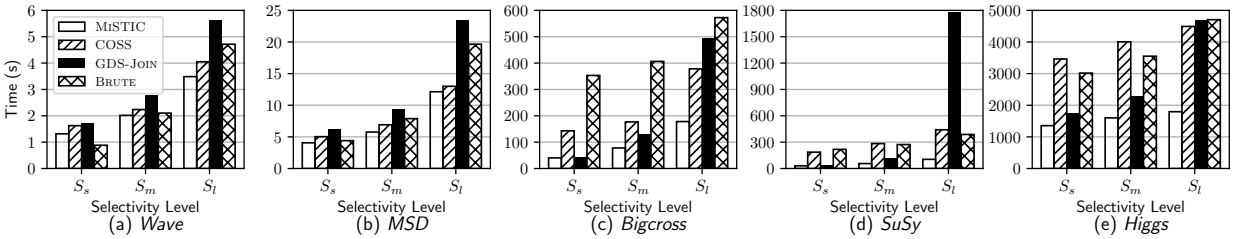


Figure 4.5: Response time as a function of the three selectivity values across the five real-world datasets for each reference implementation.

across the selectivity levels of the self-join using MiSTIC which uses the tree traversal described by Algorithm 3 as compared to a binary search is described in Table 4.4. The tree traversals are more efficient than the binary searches when the number of non-empty partitions in the index is higher due to the increased costs of binary searches as the size of the array being searched increases. A secondary factor in the efficiency of the searches is the

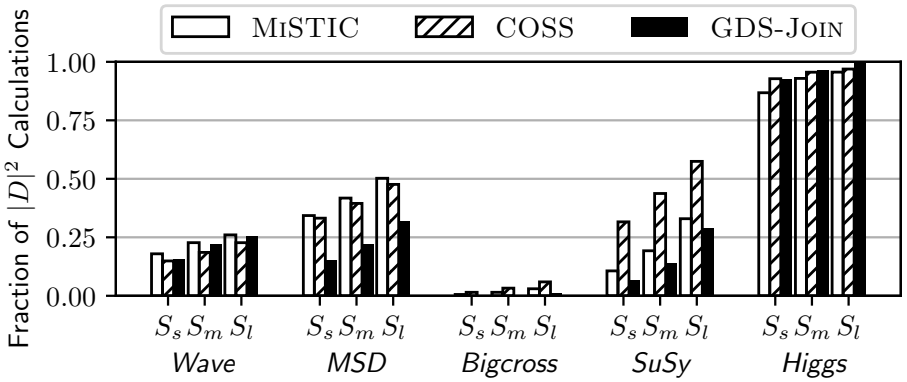


Figure 4.6: The fraction of $|D|^2$ distance calculations vs. selectivity values across the five real-world datasets for $r = 6$.

Table 4.5: The average speedup across selectivity levels for MiSTIC over the reference implementations, where speedup is the ratio of response time of the reference implementation over MiSTIC.

Dataset	COSS	GDS-Join	Brute
Wave	1.17	1.43	1.02
MSD	1.17	1.70	1.36
Bigcross	2.65	1.84	5.74
SuSy	5.15	6.89	5.26
Higgs	2.52	1.78	2.36

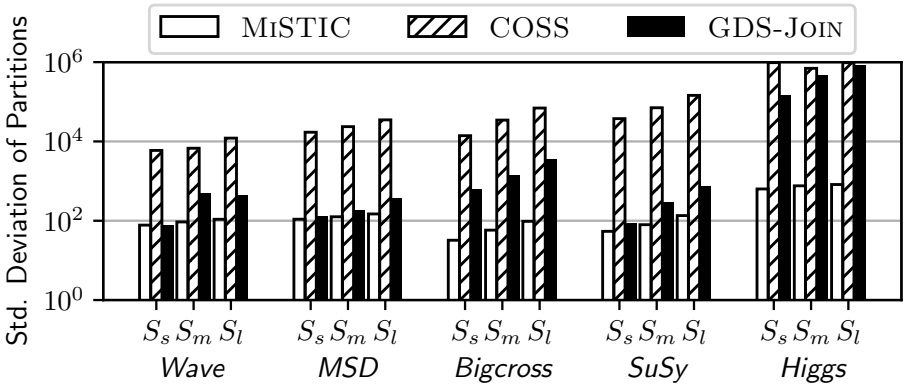


Figure 4.7: Standard deviation of the number of points in each partition vs. selectivity values for the five real-world datasets.

ability of the tree traversal to short-circuit and terminate before checking each layer of the tree. This happens in datasets with over-dense regions where there is a higher chance of adjacent partitions being empty, as opposed to a more uniformly distributed dataset where the non-empty partitions are more likely adjacent to other non-empty partitions. We observe from Table 4.4 that *SuSy* has the greatest speedup using tree traversals while also having the largest number of non-empty partitions on average across the selectivity levels. Only *Higgs* is faster with binary searches because it has few non-empty partitions.

4.5.5 Comparison to the Reference Implementations

Now that we have demonstrated key facets of the performance of MiSTIC we compare it to the reference implementations (COSS, GDS-Join, and Brute). Figure 4.5 shows the total response time (s) vs. selectivity level across five real-world datasets. The average

speedup for S_s, S_m, and S_l for MiSTIC over the other implementations is given in Table 4.5. We observe that MiSTIC has the lowest response time in every instance except for Figure 4.5(a) where Brute has a lower response time due to the index construction overhead shown in Figure 4.3.

Measuring Distance Calculations – In Figure 4.6, we plot the fraction of $|D|^2$ distance calculations that are performed for MiSTIC, COSS and GDS-Join (Brute is omitted because it performs $|D|^2$ calculations). The fraction of $|D|^2$ calculations represents the reduction of distance calculations due to the index as compared to Brute. Measuring the number of distance calculations needed is a good representation of the amount of work that each algorithm performs but does not directly correspond to the response time. In Figure 4.6, we observe that MiSTIC performs more distance calculations than either COSS or GDS-Join for the *Wave* and *MSD* datasets but has a lower response time (Figure 4.5). This behavior is also observed for the *SuSy* dataset at S_l where MiSTIC performs more calculations than GDS-Join but yields a speedup of 17.07× over GDS-Join.

We also observe in Figure 4.6 that the number of distance calculations needed for the highest selectivity level on the *Higgs* dataset approaches $|D|^2$. This is the effect of the *curse of dimensionality*, which is difficult to mitigate for these selectivity levels, even for the metric-based index COSS.

Workload Balancing – The GPU uses a Single Instruction Multiple Thread (SIMT) execution model which requires all threads within a warp to execute operations in lockstep. MiSTIC assigns one thread to each point in the dataset; therefore, threads belonging to separate partitions that are assigned to the same warp may have load imbalance due to differing numbers of comparisons between candidate points. Therefore having a similar number of points assigned to each partition improves workload balancing, and reduces the time it takes for all threads in a warp to complete their distance calculations. We hypothesize that one reason MiSTIC is faster than COSS and GDS-Join is because the variance of points in each partition is lower for MiSTIC than these reference implementations. To examine load balancing, Figure 4.7 shows the standard deviation of the number of points in each non-empty partition. MiSTIC has the lowest standard deviation, and therefore the best load balancing and most even distribution of work among threads. This is in part due to the StdDev heuristic used during incremental construction which prioritizes this

distribution of points among the partitions. COSS has the highest standard deviation in every scenario but as described in prior work [35], workload imbalance is partially offset by assigning multiple threads to each point and batching the computations based on the amount of work.

Selecting Between Brute and MiSTIC– There are some cases where MiSTIC performs worse than the brute force approach, Brute (Figure 4.5). This is due to index construction overhead which requires a large fraction of the total time when there at low selectivity levels and/or when $|D|$ is small. Brute has a time complexity of $O(|D|^2)$ which is penalized less by smaller datasets. In these scenarios, the response times are low and occur when GPU acceleration is unwarranted.

Why is MiSTIC Faster? – As described above, MiSTIC sometimes performs more distance calculations than COSS or GDS-Join, but is faster than those algorithms across all datasets and selectivity levels. We summarize why MiSTIC is faster than the other approaches as follows: (*i*) MiSTIC uses tree traversals instead of binary searches to perform index searches. (*ii*) The StdDev heuristic leads MiSTIC to distribute the points evenly across the partitions, which results in a more balanced workload and increased performance. (*iii*) MiSTIC is robust to data characteristics because of the blending of metric- and coordinate-based partitioning strategies.

4.6 Conclusion

MiSTIC demonstrates that a blended approach to partitioning yields a more robust index than either a metric- or coordinate-based indexing method. Consequently, MiSTIC can be used instead of selecting a metric- or coordinate-based index based on the dataset characteristics. We show that MiSTIC's incremental construction substantially improves performance when coupled with the StdDev heuristic. Additionally, the novel reference point placement strategy improves the pruning efficiency of MiSTIC by intelligently placing reference points to maximize the variance of the points in the associated partitions. The

experimental evaluation shows that MiSTIC is the best GPU index for range queries on large high-dimensional datasets, with an average speedup over the state-of-the art reference implementations of 2.53× and 2.73× for COSS and GDS-Join, respectively.

Chapter 5

Parallel Combination Generation on the CPU and GPU for Secure Key Retrieval

COSS and MiSTIC were both introduced in the previous two chapters. One of the problems with similarity searches shared by both COSS and MiSTIC is searching for partitions in the index. To find points in the index that may be adjacent to a query point, the algorithm needs to search adjacent partitions to the partition the query point was assigned to. This requires that the address for each partition will have to search for partitions in the index which vary by up to 1 in each dimension of the index (where the dimension of the index is k, the number of reference points or indexed dimensions used in construction of the index). For example an index with $k = 4$ may have a partition with address $\{3, 1, 4, 1\}$. A subset of adjacent partitions include partitions with addresses: $\{2, 1, 4, 1\}$, $\{3, 1, 5, 1\}$ and $\{2, 1, 5, 1\}$. In total, each partition would have $3^k - 1$ adjacent partitions. To generate which partitions to search for in the index, a mask is generated and combines the query partitions address. For the example the masks used would be: $\{-1, 0, 0, 0\}$, $\{0, 0, 1, 0\}$ and $\{-1, 0, 1, 0\}$. Generating these masks is a non-trivial combinatorial operation. COSS uses an even more complicated version of this which allows partitions to only compare in one direction which reduces computation for self-join operations.

These combinatorial operations are necessary for performing searches in a Hamming space which requires a starting address and then a mask is used to modify the starting address. In the following chapter, different combinatorial methods are evaluated for performance using

the retrieval of a corrupted hash input as an example application. While the applications discussed in this chapter are different than those in the previous two chapters the searching strategy and fundamental problem is similar.

This work originally appeared in the reference below and has been adapted for this dissertation from its original format.

Brian Donnelly and Michael Gowanlock. Performance Characterization of Parallel Combination Generators on CPU and GPU Systems. To appear in the Proceedings of the 15th International Workshop on Accelerators and Hybrid Emerging Systems (AsHES 2025).

5.1 Abstract

Combinatorics is a fundamental aspect of computer science and encompasses many aspects from graph theory to automata. One often overlooked aspect of combinatorics is combination generation which is key to several applications, and critically key to several security protocols. Despite this, there has been an insignificant research dedicated to improving combination generating algorithms since their original introduction roughly 50 years ago. Since their introduction, a number of aspects of computing have changed including the introduction of multi-core CPUs and GPUs. This work examines combination generating algorithms which create combinations for all possible values of n objects taken k at a time. In this paper, we parallelize seven existing combination generation algorithms and optimize one for improved performance on modern parallel architectures. We perform a thorough evaluation of all eight algorithms on a wide range of parameter spaces to gauge their scalability. Combination generators are typically used as a subroutine in a larger application, and so we also examine the performance when combinations are used as input into another routine. We show that the architectural differences between the CPU and GPU influence the performance of the algorithms and include analyses to explain these differences. From our evaluation, we recommend specific combination generators that have the best performance based on architecture and application scenario.

5.2 Introduction

In this paper, we examine generating combinations of n objects taken k at a time. Applications that demand large combination spaces (typically large n) yield high computational cost to generate these combinations. Combination generation is important for many fields including communications, molecular biology, computer architecture, engineering, cybersecurity, among others [84]. In the field of cybersecurity alone, combination generation is used largely for encryption tasks, including homomorphic encryption [101], block ciphers [50, 121], image encryption [40, 88], in addition to digital signature schemes [41].

History and Background: Early combination algorithms were developed in the 1960s and 1970s to generate combinations of n objects taken k at a time, which are still used today. A survey by Payne and Ives in 1979 [91] compared a number of these algorithms. Since that time, there have been changes in computer architecture and how these algorithms are used, most notably *in the size of the combination spaces being explored*. The comparison in Payne and Ives's work evaluated algorithms in two cases: $n = 16$ where $k \leq 8$, and $n = 32$ where $k \leq 4$; each having a maximum combination count of 12,870 and 35,960, respectively. In later work by Akl [2], the combination spaces were expanded by a small margin.

Batched/Throughput-oriented Combination Generation: We focus on evaluating algorithms based on their efficiency of bulk (or batched) combination generation. This is the typical use case for combination generators where a given application requires producing large combination spaces [74, 84]. This implies that throughput-oriented architectures, such as the GPU, may be proficient at computing combinations for large combination spaces.

Performance Characterization Under Two Application Scenarios: We examine two application scenarios. In the first, we examine combination generation in an isolated setting where where combinations are generated and are not used as input into another routine. In the second, we examine adding our combination generators to an application to observe potential changes in overall performance behavior.

The first application scenario yields insight into application behavior when there are no

other tasks that need to share compute resources. The second application scenario yields insight into the case where combination generation is used as a subroutine within a larger computational task. The performance of GPUs are very sensitive to the coupling of tasks within a single GPU kernel because resources need to be carefully managed to achieve good performance. For instance, adding a second task within a GPU kernel is likely to increase register pressure which could impact the kernel's theoretical occupancy, or having two tasks occupy shared memory may limit the number of threads that can be executed in a CUDA block[1].

Case Study Application: To examine performance behavior in the second application scenario above, we employ a case study application. In this scenario, we correct a corrupted cryptographic key based on the hash (using SHA3 [38]) of the original key, which is used in protocols such as response-based cryptography [74]. We use this case study to illuminate the performance differences between the combination generating algorithm when there is contention for compute resources.

Reevaluating the State-of-the-art: To the best of our knowledge, there have not been any survey papers in the literature that perform extensive benchmarks on these algorithms since Akl's work in 1981 [2], and as we will discuss in Section 5.4, parallel algorithms in this area are limited. We assess the performance of parallel combination generation algorithms, which addresses a significant gap in the literature that is *long overdue for reevaluation*, as many applications, including our motivating case study application in cybersecurity, rely on fast combination generators.

Summary of Novel Contributions: (*i*) We categorize seven combination algorithms based on their properties to explain the performance behavior of each algorithm and to identify characteristics which make them suitable for specific application scenarios. (*ii*) We compare the performance of the seven algorithms and show which performs the best on the multi-core CPU and GPU. (*iii*) We optimize a variant of Algorithm 515. Our table-based variant

[1]We use CUDA terminology throughout this paper.

achieves a mean speedup of 7.07× and 5.01× over the original algorithm when executed on the CPU and GPU, respectively.

The paper is organized as follows. The binomial coefficient calculation and the properties of combination generating algorithms are defined in Section 5.3. The history and background of combination generating algorithms is described in Section 5.4. All of the evaluated combination generating algorithms are described in Section 5.5. The experimental evaluation is presented in Section 5.6. Finally, the discussion and conclusions are presented in Section 5.7.

5.3 Definitions and Properties

In this section we define the binomial coefficient and outline the properties of combination generation algorithms.

5.3.1 Calculating the Binomial Coefficient

The binomial coefficient $\binom{n}{k}$ defines a set of combinations that are unique and we refer to this unique set of combinations as a combination space. The combination space consists of combinations in an ordered sequence where the order in the sequence is dependent on the algorithm used for combination generation. The number of combinations of n objects taken k at a time is calculated as follows:

$$C(n,k) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}. \tag{5.1}$$

$C(n,k)$ is typically calculated with a commonly used iterative algorithm described by Algorithm 4.

5.3.2 Combination Generating Algorithm Properties

We summarize the properties of combination generating algorithms using terminology from the literature. We compare algorithms and summarize their properties in Table 5.1.

Algorithm 4 Computing the Binomial Coefficient

```
1: procedure C(n, k)
2:     x ← 1
3:     for i ∈ 1, ..., k do
4:         x ← x · (n − i)
5:         x ← x / (i+1)
6:     Return x
```

We use these properties when describing the eight algorithms that we evaluate in Section 5.5.

5.3.2.1 Ranking vs. Unranking Algorithms

Ranking algorithms generate a sequence of combinations such that generating the next combination requires the current combination [68] and these algorithms often have specific starting conditions that are required to generate the first combination. In contrast, an *unranking* algorithm needs the iteration number of the combination and generates combinations out of order. All *Unranking* algorithms must compute multiple binomial coefficients (Equation 5.1) for each combination generated [67].

Comparing the two approaches, *ranking* algorithms must generate combinations in the order they occur in a sequence but are very computationally efficient, whereas *unranking* algorithms are more flexible and generate any combination of a sequence in any order, albeit at an increased cost for each generated combination. It should be noted that *ranking* and *unranking* algorithms do not generate combinations in the same order, thus the i^{th} combination in a *ranking* algorithm's sequence will not be the same as the i^{th} combination in the *unranking* algorithm's sequence.

5.3.2.2 Historyless Algorithms

A combination generating algorithm is *historyless* if the algorithm only requires the previous combination to generate a subsequent combination [102]. An algorithm is not *historyless* if it requires additional information beyond the previous combination. *Historyless* algorithms have the benefit of requiring less memory and typically fewer memory accesses.

Table 5.1: Properties of the eight combination generating algorithms that we evaluate.

Algorithm	Unranking	Ranking	Historyless	Cyclical	Reversible	Pointer	Storage (bytes per thread)	Ref.
Algorithm 515	✓		N/A	N/A	N/A	✓	$k \cdot 4$	[18]
Algorithm 515 (Table)	✓		N/A	N/A	N/A	✓	$k \cdot 4 + 8 \cdot (n \cdot k)/t$	
Pnxcb		✓	✓	✓	✓	✓	$k \cdot 4$	[91]
Algorithm 154		✓	Semi			✓	$k \cdot 4 + 1$	[81]
Algorithm 94		✓	✓	Semi		✓	$k \cdot 4$	[71]
Chase's Sequence		✓				✓	$(k+1) \cdot 4 + (k+1) + 4$	[27]
Cool-Lex Branchless		✓		✓			$n + 8$	[99]
Cool-Lex Loopless		✓		✓			$n + 8$	[99]

5.3.2.3 Cyclical Algorithms

An algorithm is *cyclical* if the last combination generated can be used to generate the first combination in the sequence [99]. *Cyclical* algorithms are able to generate the entire combination sequence regardless of the initial combination.

5.3.2.4 Reversible Algorithms

Reversible algorithms allow for generating the previous combination based on the current combination [91]. A *cyclical* and *reversible* algorithm is able to generate the last combination in the sequence from the first combination in the sequence.

5.3.2.5 Pointer Algorithms

A *pointer* algorithm has k pointers to indicate the locations in n that are permuted [91]. The pointers are used to define the combination rather than storing n values. For example, the binary combination 01001001 in the set of $\binom{8}{3}$ is represented by the pointers $1, 4, 7$ instead of needing all 8 values (i.e., storing both the permuted and non-permuted bits). This guarantees memory conservation since $k < n$ ($k = n$ is trivially computed).

5.4 Background and Related Work

There are numerous algorithms for generating a set of $\binom{n}{k}$ combinations, all of which generate combinations in a specifically ordered sequence which may be useful some problem

constraints [37, 70, 84, 102, 106, 114]. In contrast, we focus on generating all possible combinations $C(n, k)$, where the ordering of the combination sequence is irrelevant. This addresses the problem of generating all possible combinations that are within a hamming distance k of a given sequence of n objects which is required for our case study outlined in Section 5.2. Since combination ordering is irrelevant, with the only requirement being that the combination space is explored exhaustively, the best algorithm has the lowest response time for a given set of input parameters n and k.

Combination generation with ordered sequences: While we do not focus on problems where the ordering of a sequence is important, we do observe that using any parallel architecture will disrupt the ordering since the sequence will be generated non-deterministically. This is especially true for massively parallel architectures such as GPUs, where the number of threads generating combinations may be greater than the number of combinations being generated by each thread. Thus controlling the exact order in which the sequence generates is nearly impossible. For application scenarios where strict ordering is necessary (which is by definition a sequential process) it may be impossible to efficiently parallelize sequence generation. An alternative is to use an *unranking* algorithm (such as ALGORITHM 515 [18]) which generates combinations in any order at the expense of performance.

Recursive Algorithms: One algorithm for general combination generation is to use a recursive function. Recursive functions are not well-suited to the GPU due to its limited stack size; thus they are not considered in this work. Also, recursive algorithms have never been popular because of their low performance and were not evaluated in previous works as well [2, 91].

De Facto Algorithm: A commonly used combination generation algorithm is reported in Knuth's The Art of Computer Programming [66], which is called Gosper's Hack and it generates combinations rapidly and in place with only a few bit-wise operators. The drawback of Gosper's Hack is that it performs well for native data types. The bit-wise operators require significant modification for non-native data types (where the value n is

not either 8, 16, 32, or 64 for most systems) and this significantly degrades the performance relative to other algorithms [74].

Parallel Algorithms: There has been limited work on designing parallel combination generating algorithms. One such algorithm by Torres et. al. [108] is limited to $n - k + 1$ processors, which makes it usable on the CPU but not on the GPU where the number of active threads will be greater than most n values. For example, given typical parameters, such as $n = 256$ and $k = 5$, only a maximum of 252 threads could be created, which might be suitable for a CPU, but not a GPU. Furthermore, Torres et. al. [108] does not compare their combination algorithm to any other algorithms.

Another set of algorithms by Kokosínski [68] introduces a number of *unranking* algorithms which generate a different sequence than the *unranking* algorithm we evaluate but with similar computations. Kokosínski proposes that all *unranking* algorithms are straightforward to parallelize since they generate combinations out of order; however, they do not offer details on parallel performance nor consider hardware limitations that are typical of parallel algorithms.

Pioneering works: Previous works have evaluated some of the same algorithms that we examine here. Payne and Ives work [91] contradicts later work by Akl [2] where the former work found the PNXCB algorithm to have the best performance while Akl's work found Misfud's ALGORITHM 154 [81] has the best performance. One discrepancy between these two evaluations is that they compared on different n and k combinations as well as different hardware platforms. We examine many of the same algorithms as the two contradicting evaluations and expand on them to evaluate the parallel performance of these algorithms on larger combination spaces using the CPU and GPU.

5.5 Summary of Evaluated Algorithms

We select seven algorithms from the literature to evaluate. We chose one *unranking* algorithm and two *non-pointer* algorithms as well as four *pointer-ranking* algorithms. This

yields an extensive performance evaluation that includes algorithms that have different performance characteristics and properties which may be useful in certain contexts. The seven selected algorithms best represent the highest performing state-of-the-art combination generating algorithms as supported by previous benchmarks [2] and other works [37]. The properties of these algorithms are compared and summarized in Table 5.1. All of the algorithms were translated to the C and CUDA C programming languages. In addition to the seven algorithms in the literature, we optimize the *unranking* algorithm, Algorithm 515, for increased performance and compare to the seven other algorithms.

To generate sequences in parallel we save a combination in each algorithm's sequence and use that as a starting combination for each thread. In what follows, the only algorithms that do not need to save starting combinations are Algorithm 515 and Algorithm 515 (Table), which are both *unranking* algorithms.

Algorithm 515: This algorithm was proposed by Buckles and Lybanon in 1977 [18] and was an early *unranking* algorithm to generate combinations in a lexicographical sequence. Instead of needing the previous combination to generate the subsequent combination, Algorithm 515 instead repeatedly calculates the binomial coefficient described in Section 5.3. Algorithm 515 is straightforward to parallelize as it only requires the sequence number of the combination, thus negating the need to save starting combinations.

Algorithm 515 (Table): While Algorithm 515 is useful because of its low memory requirements, it is the slowest algorithm evaluated in this work due to the cost of repeatedly calculating the binomial coefficient described by Algorithm 4. To increase the performance of Algorithm 515, while also maintaining the benefits of an *unranking* algorithm, we precompute a table of binomial coefficients. This table is n by k elements and replaces the binomial coefficient calculation in Algorithm 515 at the cost of increased memory accesses. This table is shared across all threads. We find that the Algorithm 515 (Table) always outperforms Algorithm 515 in our evaluation.

Pnxcb: This algorithm is a modification of Liu and Tang's NXCB algorithm [76]. Pnxcb

is an optimized pointer version of NXCB introduced by Payne and Ives [91]. PNXCB is reversible, which allows a parallel algorithm to double the number of threads per starting combination by having one thread work forward in the sequence while another thread works backwards in the sequence. This is important when the number of starting combinations is large because it decreases the memory footprint of the algorithm. PNXCB is also able to start generating the sequence from any iteration in the sequence since it is a *cyclical* algorithm. Since PNXCB is historyless, cyclical, reversible, and uses pointers, it is the most adaptable to different scenarios of the *ranking* algorithms evaluated.

Algorithm 154: ALGORITHM 154 is a combination generator introduced by Mifsud in 1963 [81]. This algorithm generates combinations in a lexicographical sequence with a special case for the first combination generated. ALGORITHM 154 is semi-historyless since it requires a single Boolean variable to track if the combination is the first; otherwise it requires no other additional memory.

Algorithm 94: Kurtzberg proposed ALGORITHM 94 in 1962 [71] as one of the first combination generating algorithms. The starting conditions for the algorithm requires an array of zeros while the last iteration in the sequence will generate the first combination, thus making the algorithm cyclical after the first combination is generated. If there are pre-computed iterations in the sequence, then any of those iterations could be used as a starting point to generate the entire sequence.

Chase's Sequence: CHASE'S SEQUENCE (also known as Algorithm 382) is a *near-perfect* Gray code [27, 66] that guarantees that the difference between each combination is as small as possible. This makes each iteration in CHASE'S SEQUENCE as similar as possible to both the previous and the next iteration. While CHASE'S SEQUENCE has increased memory overhead due to the required history for each iteration, it is still useful in implementations where the similarity between each iteration is important. CHASE'S SEQUENCE requires not only the k pointers to the permuted values but also additional variables to keep track of the history, thus adding $k + 9$ bytes to be saved between each generated combination.

Cool-Lex Branchless and Cool-Lex Loopless: Ruskey and Williams introduced a new series of combination generating algorithms in 2009 [99] that use the Cool-Lex ordering algorithm. The ordering of the sequence is fully cyclical; thus any of the Cool-Lex algorithms can start at any iteration in the sequence. Ruskey and Williams introduced several algorithms, where the two most suited to parallel architectures, particularly the GPU, are the branchless and loopless algorithms. Both the branchless and loopless algorithms require n bytes to store the current combinations since they are not pointer algorithms. Additionally they require two integers of history for each combination which results in an additional 8 bytes. Because the Cool-Lex algorithms do not use pointers, they are not well suited to problems with high n values, as storage cost scales with n.

5.5.1 Parallelizing the Algorithms

Algorithm 5 GPU Combination Generation.

```
1: procedure CombinationGPUKernel(n, k, p, S)
2:     tid ← (blockIdx.x * blockDim.x) + threadIdx.x
3:     b ← C(n,k)/p
4:     if C(n,k) mod p ≠ 0 and tid ≤ C(n,k) mod p then
5:         b ← b + 1
6:     for i ∈ 1, 2, ..., b − 1, b do
7:         CombinationAlgorithm(n, k, S[tid])
```

To parallelize the algorithms on the GPU, each thread is assigned a starting combination stored in S, where subsequent combinations for each thread are generated from this starting combination (*unranking* algorithms use the iteration number $tid*b+i$ on line 7 instead of S). Because we examine much larger workloads than prior work (up to 1.86×10^{11} combinations), we simply assign sufficient threads, p, such that each thread computes roughly $b = 10,000$ combinations, with the $C(n,k) \mod p$ remaining combinations assigned evenly across the threads. Although this is straightforward to parallelize, the starting combinations are non-trivial to create, with most algorithms requiring unique starting conditions.

Algorithm 5 describes the distribution of work across all of the threads on the GPU and is applied to each combination algorithm by updating the combination algorithm as appropriate

Table 5.2: The AMD and Intel platforms used in our experimental evaluation, where the number of physical CPU cores are reported. PlatformA contains the GPU used in the evaluation.

Platform	**CPU**					**GPU**			
	Model	# Cores	Clock	L3 Cache	Memory	Model	# Cores	Clock	Memory
PlatformA	2×Epyc 7542	64	2.9 GHz	2×128 MB	512 GiB	Nvidia A100	6912	1.4 GHz	40 GiB
PlatformB	2×Xeon Platinum 8358	64	2.6 GHz	2×48 MB	512 GiB	-	-	-	-

on line 7. The distribution of work to threads on the CPU is done similarly except that the CPU thread ID is assigned on line 2. Algorithm 5 requires the n and k values used for the combination generation as well as the number of threads, p, and the starting position saved for each thread stored in S (with the exception of *unranking* algorithms which do not require S). The number of combinations assigned to each thread, b, is computed on line 3 and modified if the number of combinations cannot be evenly divided among the threads on lines 4 and 5.

5.6 Experimental Evaluation

To understand performance behavior on multi-core CPUs and the GPU, we examine the performance of the algorithms on two platforms as outlined in Table 5.2. The platforms use AMD (PlatformA) and Intel (PlatformB) CPUs such that our evaluation is robust and conclusions apply to platforms from two CPU vendors. PlatformA also contains the Nvidia A100 GPU used in our evaluation.

5.6.1 Experimental Methodology

The GPU code is written in CUDA v12.3 for NVIDIA GPUs and we use a kernel block size of 32, which was experimentally determined to be the optimal block size across all combination generating algorithms. On the CPU we use 64 threads (corresponding to the number of physical cores on each platform) and parallelize the algorithm with OpenMP. The CPU code was written in C++ and compiled with GNU v8.5.0 using the -O3 optimization

flag.[2]

CPU Performance Metrics: We define the parallel speedup of CPU algorithms as $s = T_1/T_p$, where T_1 and T_p are the algorithm response times with $p = 1$ and $p = 64$ threads, respectively. The parallel efficiency is given by $e = s/p$.

GPU Performance Metrics: We compare the performance of the GPU algorithms to the CPU using $p = 64$ threads/cores. We compute the GPU speedup over the multi-core CPU as: $s_{GPU} = T_{64}/T_{GPU}$, where T_{64} is the response time of the parallel CPU algorithm, and T_{GPU} is the response time of the GPU algorithm.

Application Scenarios: As described in Section 5.2, we examine combination generation under two scenarios: the first generates combinations in isolation to understand raw throughput, and the second is where the combinations are used as a subroutine in a larger application. Our case study application couples the SHA3 hashing algorithm with combination generation in the same GPU kernel which is required in the cybersecurity application outlined in Section 5.2. We perform experiments with both SHA3-256 and SHA3-512 to highlight the impacts of increasing compute resource requirements when using SHA3-512 as compared to SHA3-256. We denote the different workloads as W_0, W_1 and W_2 which correspond to isolated permutation generation, SHA3-256, and SHA3-512, respectively.

5.6.2 Combination Sets Evaluated

The performance of the algorithms vary with n and k; therefore, we present most results as an average of numerous executions with differing n and k values and we refer to these as combination sets. In particular, we evaluate all of the algorithms on the two combination sets (CS) outlined in Table 5.3 which have a wide range of combination parameters (n and k) with values that address modern workloads. The combination spaces for each CS has up to seven orders of magnitude more combinations than that of the pioneering literature [2, 91].

We also perform a more detailed examination of $n = 100, k = 7$ from CS-1 and $n =$

[2]All code will be made publicly available upon acceptance and the link reported in this footnote.

Table 5.3: Overview of combination sets (CS) where the maximum combinations column corresponds to the n and k values that yield the greatest number of total combinations. |CS| refers to the number of combination spaces for a set.

CS	**|CS|**	n **Range**	k **Range**	**Max. Combinations**
CS-1	80	n=10, 20, . . . , 100	k=1, 2, . . . , 8	$186 \cdot 10^9$
CS-2	32	n=128, 256, . . . , 1024	k=1, 2, 3, 4	$46 \cdot 10^9$

Table 5.4: The mean response time (s) for workload W_0 (raw throughput workload) across all combinations in CS-1 and CS-2 for each algorithm as executed on the two multi-core CPU and GPU platforms. The lowest response times are highlighted in bold face.

Algorithm	**PlatformA CPU** $p = 64$ **cores/threads**		**PlatformB CPU** $p = 64$ **cores/threads**		**GPU: A100**	
	CS-1 Mean Time	CS-2 Mean Time	CS-1 Mean Time	CS-2 Mean Time	CS-1 Mean Time	CS-2 Mean Time
Algorithm 515	189.56	667.40	88.33	310.64	5.58	14.76
Algorithm 515 (Table)	7.50	41.03	4.06	22.40	0.65	2.46
Pnxcb	0.82	1.35	0.66	0.87	**0.08**	**0.03**
Algorithm 154	0.64	0.80	**0.56**	0.28	0.09	**0.03**
Algorithm 94	**0.60**	**0.76**	0.61	**0.26**	0.09	0.04
Chase's Sequence	96.84	101.43	45.77	52.55	0.64	0.07
Cool-Lex Branchless	8.39	5.60	5.64	3.69	0.40	0.44
Cool-Lex Loopless	4.33	2.60	5.20	3.83	0.38	0.43

$512, k = 4$ from CS-2. Additionally, CS-2 corresponds to commonly used values for the cybersecurity case study application, as key sizes are multiples of 128-bits.

5.6.3 CPU Results: Raw Throughput Workload (W_0)

In what follows, we highlight the raw combination generation throughput (W_0) using the two multi-core CPU platforms. We refer to the results reported in Tables 5.4–5.5.

***Unranking* Algorithms (Algorithm 515 and Algorithm 515 (Table)):** As we will show, despite a higher response time than the *ranking* algorithms evaluated, Algorithm 515 and Algorithm 515 (Table) offer significant advantages in terms of flexibility to dynamic application scenarios (Section 5.5), therefore we highlight the performance of these algorithms here (and in Section 5.6.5 where we compare them on the GPU).

From Table 5.4, Algorithm 515 has the highest response time across all algorithms and both permutation sets for the CPU on PlatformA and PlatformB. This is directly the result of repeated binomial coefficient calculations described by Algorithm 4. Algorithm 515 is compute-bound and so the response time is directly related to the number of

Table 5.5: Scalability of multi-core CPU and GPU algorithms on PlatformA and PlatformB. The parallel speedup (s) and parallel efficiency (e) of the CPU algorithms is shown using $p = 64$ threads on PlatformA and PlatformB; we omit showing T_1 due to space constraints, but it can be calculated using $T_1 = s \cdot T_p$. The response time of the GPU algorithms (T_{GPU}) and speedup over the CPUs in PlatformA and PlatformB are shown.

Algorithm	**PlatformA CPU** $p = 64$ **cores/threads**			**PlatformB CPU** $p = 64$ **cores/threads**			**GPU: A100**		
	Time (T_p) (s)	Speedup (s)	Par. Eff. (e)	Time (s) (T_p)	Speedup (s)	Par. Eff. (e)	Time (s) (T_{GPU})	Speedup (s_{GPU}) over PlatformA CPU	Speedup (s_{GPU}) over PlatformB CPU
$n = 100, k = 7$									
Algorithm 515	664.38	40.02	62.53%	319.73	40.57	63.39%	19.36	34.32	16.52
Algorithm 515 (Table)	42.82	16.37	25.58%	37.74	19.90	31.09%	2.41	17.77	15.66
Pnxcb	4.83	10.01	15.64%	2.97	8.04	12.56%	0.28	17.25	10.61
Algorithm 154	4.36	10.76	16.81%	2.56	9.59	14.98%	0.30	14.53	8.53
Algorithm 94	3.76	8.67	13.55%	2.62	8.71	13.61%	0.29	12.97	9.03
Chase's Sequence	397.23	0.16	0.25%	178.95	0.25	0.39%	0.51	778.88	350.88
Cool-Lex Branchless	30.87	4.29	6.70%	17.53	6.73	10.52%	1.55	19.92	11.31
Cool-Lex Loopless	15.13	3.30	5.16%	15.84	2.04	3.19%	1.49	10.15	10.63
$n = 512, k = 4$									
Algorithm 515	336.86	32.18	50.28%	163.48	32.67	51.05%	9.28	36.30	17.61
Algorithm 515 (Table)	27.02	16.75	26.17%	12.29	39.26	61.34%	1.47	18.38	8.36
Pnxcb	1.08	4.52	7.06%	0.80	3.67	5.73%	0.03	36.00	26.66
Algorithm 154	0.70	10.13	15.83%	0.35	14.83	23.17%	0.03	23.33	11.66
Algorithm 94	0.70	4.02	6.28%	0.30	11.61	18.14%	0.03	23.33	10.00
Chase's Sequence	97.06	0.11	0.17%	52.98	0.15	0.23%	0.06	1617.67	883.00
Cool-Lex Branchless	5.17	4.54	7.09%	3.16	6.59	10.30%	0.34	15.21	9.29
Cool-Lex Loopless	2.30	4.31	6.73%	3.40	1.79	2.80%	0.29	7.93	11.72

combinations, $C(n, k)$, that are generated. Our optimized version, Algorithm 515 (Table) has a mean speedup of 21.76× and 13.87× over Algorithm 515 for CS-1 and CS-2, respectively. Algorithm 515 and Algorithm 515 (Table) achieve the highest parallel efficiency of any algorithms because they have the greatest compute to memory access ratio.

Parallel Scalability: The CPU parallel performance of each algorithm is examined for $p = 64$ cores/threads on both platforms in Table 5.5. All of the algorithms benefit from increasing p from 1 to 64 cores with the exception of Chase's Sequence. Chase's Sequence has a lower response time when $p = 1$ than when $p = 64$ across both platforms and both combination sets. Chase's Sequence requires the largest amount of storage to create each combination (see Table 5.1), resulting in a large number of memory accesses. This leads to a slowdown when threads compete for memory bandwidth. However, even when $p = 1$, it has a greater response time than the other *ranking* algorithms with the exception of Cool-

Lex Branchless; therefore, we consider it ill-suited for combination generation on the CPU.

From Table 5.5, we find that with the exception of Algorithm 515 and Algorithm 515 (Table), the algorithms do not yield very high parallel scalability/efficiency. This is largely because the algorithms use more memory operations to generate combinations resulting in memory/cache-bound performance behavior which yields low parallel efficiency. Despite this, the fastest CPU algorithm on PlatformA, Algorithm 94, yields a moderate parallel speedup of 8.67×. Algorithm 94 yields the second to lowest response time on PlatformB and it achieves a similar speedup. Overall, the 8.67–8.71× speedup for Algorithm 94 is moderate but respectable.

Intel vs. AMD CPUs: While both PlatformA and PlatformB have similar CPU metrics (64-cores with a clock speed of 2.9 GHz and 2.6 GHZ for PlatformA and PlatformB, respectively), PlatformB consistently outperforms PlatformA with few exceptions. For this reason, in all that follows, we focus on comparing the algorithms on PlatformB.

Comparison to prior work: Overall, of the eight algorithms on the CPU, Algorithm 94 has the highest performance in most scenarios which contradicts the results of both Payne and Ives', and Akl's previous works [2, 91]. This contradiction is explained by three factors: (*i*) the evolution of computer architectures over the last fifty years, (*ii*) the algorithms are executed in parallel instead of sequentially; and, (*iii*) we examine larger (modern) workloads compared to those pioneering works.

Observation #1

The fastest algorithms use optimizations that reduce computation, which yields a low compute to memory access ratio. Despite moderate multi-core scalability, for best performance on multi-core CPUs, we recommend that Algorithm 94 is employed. Other architectures that have higher aggregate memory bandwidth and cache throughput will be well-suited to parallel combination generation algorithms.

5.6.4 GPU Preprocessing Overhead

We now examine combination generation on the GPU. As described in Section 5.5.1, starting combinations are required to parallelize the algorithms on the GPU.[3] The greatest amount of preprocessing overhead occurs for both Cool-Lex algorithms (see Table 5.1), which have the greatest memory footprint, with a maximum data size of 4.38 GB (for CS-2 when $n = 1024, k = 4$). The overhead is solely related to data transfer and we find that the upper bound time needed to transfer the data between the host to the GPU is < 0.15 s using PCIe 4.0 with a bandwidth of 32 GB/s, which is at most 2.63% of the total response time for the Cool-Lex algorithms when $n = 1024, k = 4$. The median memory required by the Cool-Lex algorithms is 17 MB when $n = 1024, k = 4$, yielding negligible data transfer time. All other algorithms will have less overhead so we do not elaborate here.

Observation #2

The preprocessing overhead for assigning starting combinations to GPU threads does not negatively impact performance.

5.6.5 GPU Results: Raw Throughput Workload (W_0)

In this section we discuss the results of our experiments on the GPU and focus on the optimized version of Algorithm 515 and the best performing algorithm on the GPU,

[3]Starting combinations are also required on the CPU; however, because the CPU supports far fewer threads than the GPU, the overhead is negligible.

Pnxcb.

***Unranking* Algorithms (Algorithm 515 and Algorithm 515 (Table)):** From Table 5.4 we observe that Algorithm 515 has the highest response time and Algorithm 515 (Table) has the second highest response time. Recall from Section 5.6.3 which compared Algorithm 515 to Algorithm 515 (Table) that Algorithm 515 (Table) yielded a speedup of 21.76× and 13.87× on CS-1 and CS-2, respectively. Similarly, on the GPU Algorithm 515 (Table) achieves a speedup of 8.59× and 6.00× on CS-1 and CS-2, respectively. Therefore, our optimized Algorithm 515 (Table) maintains a significant performance advantage over Algorithm 515 on the GPU.

Pnxcb: The algorithm with the lowest overall response time on the GPU is Pnxcb. Other *ranking* algorithms with similar performance on the GPU, Algorithm 94 and Algorithm 154, also have low storage requirements of either k or $k + 1$ bytes, respectively (Table 5.1). This low space complexity allows for higher cache throughput on the GPU. We elaborate on the cache usage of these algorithms in Section 5.6.9.

Observation #3

A comparison of combination generators yielded highly differentiated performance on legacy architectures. However, on modern architectures, the performance difference between the fastest algorithms is often negligible, particularly on modern workloads. We find that Pnxcb is the best algorithm for combination generation on the GPU across large parameter ranges.

5.6.6 Comparison of CPU and GPU: Raw Throughput Workload (W_0)

In this section we compare the experimental results for the GPU and CPU. Overall, the GPU achieves a speedup over the CPU ranging from 6.22× to 71.51× and from 6.50× to 750.71× for CS-1 and CS-2, respectively. We attribute this massive speedup to the increased memory bandwidth of the GPU for the *ranking* algorithms (see Observation #1) and the increased computational throughput for the *unranking* algorithms as compared to

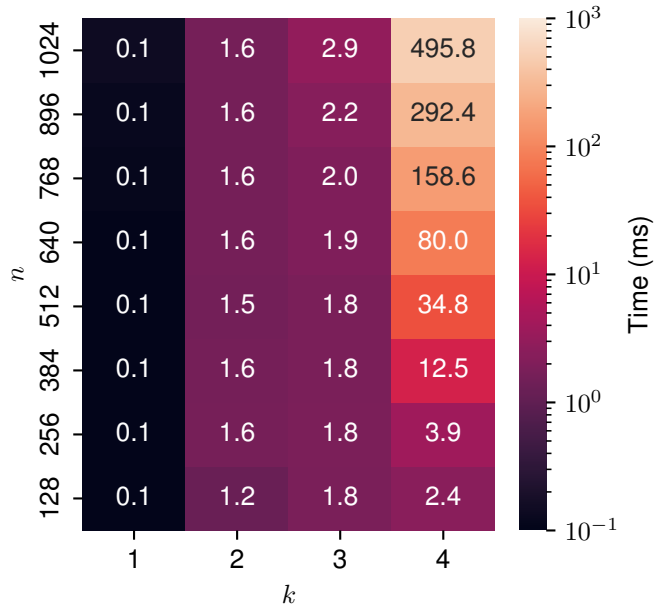


Figure 5.1: The response time (T_{GPU}) in milliseconds for W_0 showing all n and k combinations in CS-2 for Pnxcb on the GPU.

the CPUs.

5.6.7 Examination of Individual Parameters in CS-2: Raw Throughput Workload (W_0)

In Sections 5.6.5 and 5.6.6, we examined performance on the combination sets, CS-1 and CS-2, as averaged over all parameters in those sets. Here, we show the performance across each individual value of n and k in CS-2 to demonstrate how the response time increases with these parameters. Due to page limitations, we cannot show the results for all algorithms, so we focus on Pnxcb and Algorithm 515 (Table), similarly to that in Section 5.6.5.

Figures 5.1 and 5.2 show the results of this experiment on the GPU for Pnxcb and Algorithm 515 (Table), respectively. From both figures, we observe that the performance is more sensitive to k rather than n. We attribute this to the impact k has on not only the number of combinations generated, but also the amount of work needed to generate each combination. This analysis shows that due to the large differences in response times between

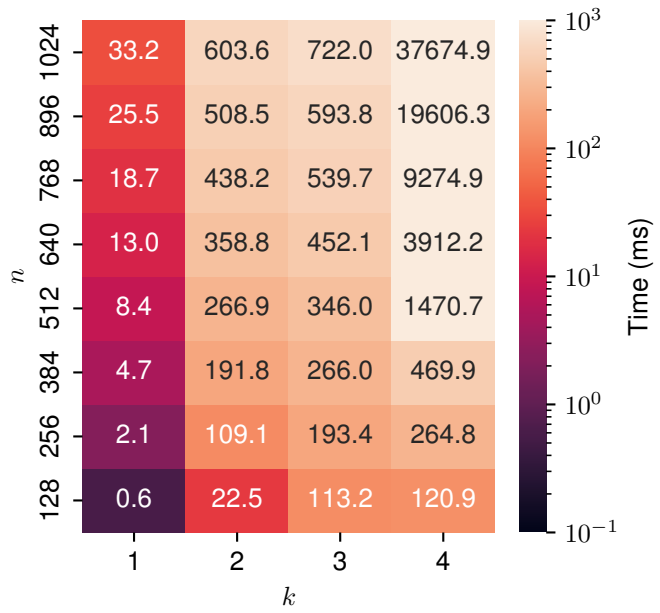


Figure 5.2: The response time (T_{GPU}) in milliseconds for W_0 showing all n and k combinations in CS-2 for ALGORITHM 515 (TABLE) on the GPU.

n and k values, the average response times reported for CS-1 and CS-2 can be misleading, particularly when assessing whether an algorithm is suitable for a fixed value of n and k. This is because the average response times are dominated by the largest n and k values for a given algorithm, e.g., in Figure 5.1 at $C(n,k) = (128, 1)$, $T_{GPU} = 0.1$ ms, whereas it is 495.8 ms when $C(n,k) = (1024, 4)$.

5.6.8 Case Study using Key Retrieval on the GPU: Workloads W_1 and W_2

Based on the significant performance gains yielded by the GPU over the CPU, in this section, we focus solely on the GPU.

Raw combination generation throughput does not adequately describe application performance when the combination generator is used as a subroutine in a larger application. As described in Section 5.2, for our case study, we examine the retrieval of a cryptographic key that has been corrupted by using a stored hash of the original key. We note here that we implement this in the response-based cryptography protocol [74], where the two tasks, combination generation and hashing, are executed within the same GPU kernel, and thus both

of the tasks require their own resources (e.g., registers, memory bandwidth, shared memory). The k value in this case corresponds to the Hamming distance between a corrupted key with n bits and the original key. For example, when $k = 3$, three bits of the n bits in the key have flipped (have been corrupted), and we retrieve the original key by searching all $\binom{n}{k}$ possible keys and match the hash output from each possible key to the stored hash. We can then recover the original key by choosing the combination which generated a matching hash.

We focus on CS-2 because the values of n are multiples of 128 bits, which are needed in the security application. We present the mean response times across all parameter values in CS-2 for the raw throughput workload (W_0), hashing with SHA3-256 (W_2), and hashing with SHA3-512 (W_3) in Table 5.6, where the lowest times are highlighted in bold face.

It is clear that the performance of the algorithms change when coupled with the hashing workloads (W_1 and W_2). However, the relative performance of the algorithms remain the same with the exception of Algorithm 515 (Table). Algorithm 515 (Table) is more robust to workload requirements than the other algorithms which we attribute to Algorithm 515 (Table)'s balance of compute and memory requirements. While our optimized Algorithm 515 does not have the best mean performance on W_2, is still maintains a respectable relative performance with $< 12\%$ greater response time than the best algorithms.

Observation #4

Benchmarking algorithms in isolation on GPU architectures may result in prematurely discarding algorithms that may have great utility when used as a subroutine in a larger application.

5.6.9 GPU Profiling Results

To understand each algorithm's performance behavior and to demonstrate that the algorithms are well optimized for the GPU, we profile their execution using the Nvidia Nsight Compute tool. Instead of averaging results over all combinations in a combination space,

Table 5.6: The mean response time (s) across all combinations in CS-2 for each algorithm as executed on the GPU with varying workloads. The lowest mean times in each column are highlighted in bold face. Note that there are several identical mean times, or times that are very close to each other and have insignificant differences in performance.

Algorithm	W_0 **(No Hashing)**	W_1 **(SHA3-256)**	W_2 **(SHA3-512)**
Algorithm 515	14.76	27.80	38.25
Algorithm 515 (Table)	2.46	**7.83**	10.10
Pnxcb	**0.03**	8.70	**8.92**
Algorithm 154	**0.03**	8.69	**8.92**
Algorithm 94	0.04	8.69	8.93
Chase's Sequence	0.07	8.72	8.95
Cool-Lex Branchless	0.44	11.38	13.21
Cool-Lex Loopless	0.43	11.32	13.15

Table 5.7: Profiler results using Nvidia Nsight Compute for $n = 512, k = 4$ on the A100 GPU. The computational throughput, memory throughput, L1 and L2 Cache hit rates are reported as a percentage where the maximum for each is 100%.

Algorithm	**Time (s)**			**Comp. Throughput (%)**			**Mem. Thoughtput (%)**			**L1 Cache Hit Rate (%)**			**L2 Cache Hit Rate (%)**			**Num. Registers**		
	W_0	W_1	W_2	W_0	W_1	W_2	W_0	W_1	W_2	W_0	W_1	W_2	W_0	W_1	W_2	W_0	W_1	W_2
Algorithm 515	9.28	17.01	23.05	63.61	43.66	32.78	0.79	21.73	16.30	100.00	99.95	99.93	99.85	100.00	99.99	44	157	173
Algorithm 515 (Table)	1.47	5.51	6.85	13.33	42.13	33.96	64.32	37.76	30.60	100.00	99.87	99.86	99.88	99.96	100.00	29	157	173
Pnxcb	0.03	4.88	5.06	12.05	29.71	28.76	68.68	75.99	74.44	99.73	99.99	100.00	100.00	100.00	100.00	24	157	173
Algorithm 154	0.03	4.91	5.12	8.71	29.49	28.34	69.54	75.36	73.25	99.61	99.99	100.00	99.99	100.00	100.00	32	157	173
Algorithm 94	0.03	4.86	5.08	15.57	29.87	28.68	69.00	76.31	74.14	99.65	99.99	100.00	100.00	99.99	100.00	32	157	173
Chase's Sequence	0.06	4.90	5.08	19.62	29.82	28.93	74.00	76.12	74.72	99.42	99.98	99.99	99.80	99.99	100.00	30	157	173
Cool-Lex Branchless	0.34	6.56	7.73	4.94	43.77	37.25	75.59	58.87	50.76	79.98	99.23	99.75	98.96	99.99	100.00	26	157	173
Cool-Lex Loopless	0.29	6.53	7.65	4.19	44.03	37.67	79.21	59.03	51.17	88.12	99.37	99.78	99.76	99.97	100.00	24	157	173

we select $C(512, 4) = 2.8 \times 10^9$ combinations from CS-2 because this is the configuration typically used in the real-world security application outlined in Section 5.6.8. We show the performance without a workload (W_0) and with a workload (W_1 and W_2).

In our results below, we focus on several metrics. Ideally we would include a roofline analysis that illustrates whether a given algorithm is compute-bound or memory-bound. However, because our data types are integers, we are unable to collect this information using the Nvidia Nsight Compute tool, as it only supports roofline statistics for floating point operations. However, as we will show below, most algorithms are cache bound, and so roofline statistics are not of great consequence.

We summarize the profiling results shown in Table 5.7. The table shows the computational throughput, global memory throughput, L1 and L2 cache hit rates and the number of registers that each thread requires in the kernel. All of the algorithms have low computational throughput, except for Algorithm 515 (as described in Section 5.5, it needs to repeatedly calculate the binomial coefficient for each combination). However, the response

time is still greater than the other memory-bound *ranking* algorithms. All of the algorithms achieved >98% cache hit rate for both the L1 and L2 caches with the exception of Cool-Lex Branchless and Cool-Lex Loopless which had an L1 cache hit rate of only ≈80% and ≈88% for W_0, respectively.

The number of registers used by the combination generating algorithms on the GPU is low and does not decrease the occupancy for W_0, where occupancy is a measure of the percentage of time that the streaming multiprocessors (SMs) are active. When the combination generation is coupled with hashing for the case study (W_1 and W_2), the number of registers used is dependent on the SHA3 version, with 157 registers for SHA3-256 and 173 registers for SHA3-512. This becomes the limiting factor for occupancy and subsequently fewer threads are active at any given time. This results in a higher cache hit rate for all of the *ranking* algorithms as shown in Table 5.7.

5.6.10 Discussion: Optimized Version of Algorithm 515

We report on our optimized implementation of Algorithm 515 by highlighting several results examined throughout the evaluation and discussing its capabilities as outlined in Section 5.5.

Performance: Algorithm 515 (Table) replaces the binomial coefficient calculations in Algorithm 515 with a table of binomial coefficients. This decreases the compute throughput and significantly increases the memory throughput as shown in Table 5.7 from 0.79% to 64.32% for $n = 512, k = 4$ with W_0. However, Algorithm 515 (Table) outperforms Algorithm 515 despite using more memory. On W_0, the optimization results in a mean speedup of 21.76× and 13.87× over Algorithm 515 on PlatformB for CS-1 and CS-2, respectively, and a mean speedup of 8.58× and 6.00× over Algorithm 515 on the GPU for CS-1 and CS-2, respectively (Table 5.4). Despite having relatively poor performance in terms of raw combination generation throughput (W_0) relative to the *ranking* algorithms, Algorithm 515 (Table) is comparable and in some cases even faster when coupled with

a workload as is the case for W_1 which uses SHA3-256 as described in Table 5.6. Specifically, our evaluation determined that Algorithm 515 (Table) is the fastest algorithm when $n \geq 768$ and $k = 4$ for W_1 which are the scenarios in which the largest number of combinations are generated.

Adaptability to Application Scenarios: Beyond comparing the performance of the algorithms, it is also beneficial to examine the ease of implementation and flexibility to application scenarios. All of the *ranking* algorithms require pre-computing the starting conditions for each thread, whereas the *unranking* algorithms do not. This is why Algorithm 515 (Table) is the best algorithm for most scenarios. Algorithm 515 (Table) has comparable performance to the *ranking* algorithms while maintaining the flexibility of Algorithm 515.

Observation #5

We recommend Algorithm 515 (Table) for most scenarios due to robust performance when coupled with a workload, the ease of implementation and the flexibility of the algorithm to dynamic application scenarios.

5.7 Conclusions

The performance of combination generating algorithms has changed significantly in the last fifty years since they were first comprehensively evaluated. The benchmarks presented in this paper highlight the impact that different architectures have on the performance of combination generation algorithms. By providing robust benchmarks on two multi-core CPUs and one GPU, this paper offers a comprehensive examination of performance behavior. This indicates which algorithm should be selected for a given application that employs combination generators.

While our results are inconsistent with the pioneering works [2, 91], we note that these works were conducted before the proliferation of multi- and many-core architectures. These contradictions between the works illuminate the need for an updated evaluation of combi-

nation generation algorithms provided in this paper.

We summarize several key findings as follows. (*i*) Algorithm 94 is the fastest algorithm on the CPU, which directly contradicts the findings of Payne and Ives', and Akl's previous works [2, 91]. (*ii*) Pnxcb is the fastest algorithm on the GPU. (*iii*) Algorithm 515 (Table) achieves a significant speedup over the original and performs exceptionally well when coupled with a workload on the GPU.

Our evaluation focuses on characterizing the performance of batched combination generation algorithms. These approaches can be employed in several applications; for example, the response-based cryptography protocol presented by Lee et al. [74] requires generating over 8 billion combinations, where $n = 256$ and $k = 5$. In our case study we show how choosing the correct combination generating algorithm can impact the overall performance of their protocol.

Chapter 6

Authentication in High Noise Environments using PUF-Based Parallel Probabilistic Searches

The previous chapter examined the problem of matching a corrupted input to a hash with the original input. This is a fundamental part of larger cryptographic protocols such as Response Based Cryptography (RBC). One of the main bottlenecks in RBC is the time required to search through so many possible hash inputs. The previous chapter examined the best methods for performing this task as quickly as possible. In this chapter other optimizations are discussed which reduces the total workload needed for RBC rather than accelerating a specific component of the search.

This work originally appeared in the reference below and has been adapted for this dissertation from its original format.

Donnelly, B. & Gowanlock, M. Authentication in High Noise Environments using PUF-Based Parallel Probabilistic Searches. To appear in the Proceedings of the 28th IEEE High Performance Extreme Computing Conference (HPEC).

6.1 Abstract

Enabling secure communication in noisy environments is a major challenge. In these environments, the outputs of cryptography algorithms undergo error where several bits change states and since these algorithms cannot tolerate any error, authenticating and securing

communication between parties is disrupted. We propose a noise-resistant public key infrastructure protocol that employs physical unclonable functions (PUFs). PUFs act as a unique fingerprint for each device in a network; however, their state may drift over time due to fluctuations in temperature and other factors. Using a PUF requires a search to identify flipped bits which is conducted on a secure server that has the benefit of removing error correction on low-powered client devices. We exploit the probabilistic nature of PUF bit error rates (BERs) and use this information to aid in the search process that resolves the noise imparted by the environment. We show that using a 256-bit PUF-generated seed (a PUF response) our protocol is robust to a PUF BER of $\approx$11% (or 30 of 256 bits) and a transmission bit error rate (TBER) of 30%. In this scenario, on average the authentication mechanism on a secure server requires $\lesssim$5 s. We also show results for higher PUF BERs which have a $<$100% authentication success rate which indicates the upper limit on the PUF BER tolerance of our protocol.

6.2 Introduction

One drawback of Public Key Infrastructure (PKI) is that all devices in the network use public/private key pairs, and the private key is typically stored in non-volatile memory (e.g., disk). If a private key is recovered from a device by an attacker, then they are able to masquerade as the user of the private key.

Several efforts have been proposed to mitigate this drawback by equipping client devices with Physically Unclonable Functions (PUFs) for the purposes of authentication [21, 22]. PUFs are volatile memory and are unique due to variance in the manufacturing process, allowing each PUF to act as a unique fingerprint for a device. PUFs are employed to generate random numbers (hereafter referred to as seeds) that are used as input into cryptography algorithms. The cells in a PUF are unstable and drift over time due to environmental factors such as temperature [107]. This error needs to be corrected, otherwise authentication will fail. One method that has been proposed to mitigate this error is to use error correction

codes (ECC), data helpers, and fuzzy extractors [7, 34, 44, 56] but they have two major drawbacks: (*i*) many devices are low-powered and are unable to correct for the error due to latency and/or power constraints; and, (*ii*) these mechanisms leak information [33, 69, 110] about the PUF if the device is compromised.

To address the drawbacks of error correction codes, the Response-Based Cryptography (RBC) protocol was developed [21, 23, 72, 93, 117, 118] which places the computational burden of authenticating PUF-equipped client devices on a secure server such that low-powered client devices do not need to employ error correction codes, data helpers, or fuzzy extractors. In the RBC algorithm, during enrollment, a server records the PUF images that are deployed in client devices. During the handshake between the server and client, the server instructs the client to read a subset of its PUF cells (this is denoted a PUF challenge), and the client uses this to generate a seed (a PUF response) which is used to create public/private key pairs. The client sends the public key to the server for authentication and the server generates a public key from the client PUF image. Because the seed generated by the PUF may have drifted and has a bit error rate (BER) >0% relative to when the PUF was enrolled, the server performs a search by flipping bits in the seed generated from the PUF image such that it reproduces the client's public key. If it does so within a Hamming distance threshold, the client is authenticated.

Lee et al. [73] proposed an efficient search method for iterating over PUF seed spaces. The search method employs the probability that a given cell in the PUF is likely to flip (this is information obtained during the PUF enrollment process), such that an ordering of candidate seeds is exploited. This allows searching the PUF seed space intelligently instead of searching without any knowledge of the probability that a PUF cell will flip. This significantly improves the performance of the RBC search by reducing the total number of seeds that need to be searched and allows for authentication even when PUFs may have drifted significantly from their initial state.

The research on RBC to date, including the most recent by Lee et al. [73] does not

permit any error in the transmission between the client and server. Securing communication in noisy or low signal environments is critical for many applications. Examples include unmanned autonomous vehicle (UAV) drone package delivery [82] in low signal environments, or wireless communication at a music concert where numerous people are accessing the same network. Securing communication in these settings is challenging for cryptography because by definition, cryptography primitives do not tolerate any error. A *motivating example* is as follows: Consider the following canonical scenario employing *Alice* and *Bob* that wish to communicate. Assume an asymmetric public key infrastructure (PKI) system where Alice wishes to receive an encrypted message from Bob. Alice generates a public/private key pair and sends her public key to Bob but the public key contains an error (i.e., one bit has flipped) due to noise in the transmission environment. Bob encrypts and sends Alice a message using her public key; however, Alice is unable to decrypt the message with her private key due to the erroneous public key received by Bob. This example illustrates that if any of the bits are incorrect, then secure communication is impossible between two parties. While ECC can be used to address this issue it burdens the low-powered device with more work while also enabling attackers to exploit vulnerabilities in ECC information leakage. By correcting for the noisy environment on the server and not the client, we gain an additional layer of security.

To address communication in noisy environments, the contributions of this paper are two-fold. First, we propose an authentication protocol that is resilient to noisy environments. Second, we optimize the system through algorithmic innovations and the parallelization potential of modern multi-core CPUs which are needed to reduce authentication latency. The same benefits of RBC are retained in our protocol, which include: requiring that the secure server perform error correction, thus allowing client devices to be lightweight/low-powered, and employ a probabilistic search of the PUF seed space. Our protocol bolsters the prior work on RBC by extending it to noisy or low-signal communication environments. To summarize, we make the following contributions.

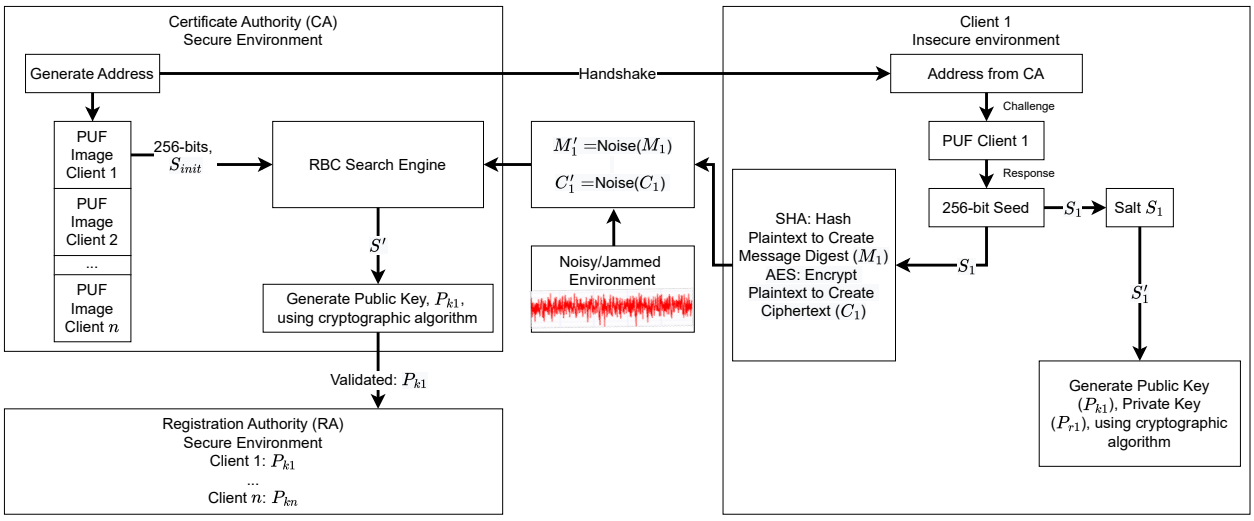


Figure 6.1: The response-based cryptography protocol that is robust to noise (npRBC). The RBC search engine is shown in Figure 6.2. This inspiration for this figure is from similar figures in the literature [73, 74].

- We introduce Noisy Probabilistic Response-Based Cryptography, npRBC, a protocol to authenticate low powered devices in high noise environments.
- We evaluate the protocol using a Monte Carlo approach that spans PUF BER noise levels between 20-45 bits and transmission errors (TBER) up to 44% (the upper bound is 50% [104]). The PUF BER is derived from enrollment data of an SRAM PUF and the TBER is selected in intervals that represent varying levels of noise in the environment.

The paper is organized as follows: Section 6.3 outlines the proposed protocol, npRBC, Section 6.4 presents the experimental evaluation, and finally, Section 6.5 concludes the paper and discusses future work directions.

6.3 npRBC

Our response-based cryptography protocol, npRBC, builds on the work of Lee et al. [73] which showed that response-based cryptography can employ knowledge of enrollment data which quantifies the bit error rate (BER) of each cell in a client's PUF and stores this information in the PUF image on a secure server. Our proposed protocol differs from Lee et al. [73] as it is robust to noise and requires that the client send a SHA3-512 message digest

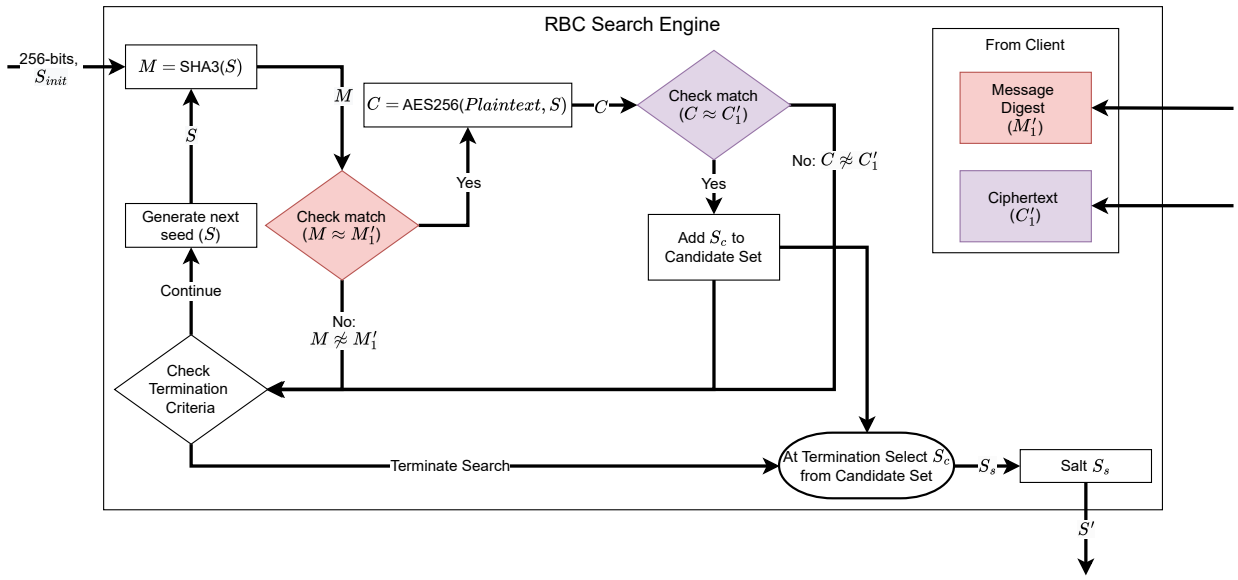


Figure 6.2: The probabilistic RBC search engine. Colors represent comparisons between client and server information.

(M_1) of the hashed 256-bit seed and also a ciphertext (C_1) generated with AES256 using the same 256-bit seed as the key. Sending both M_1 and C_1 allows the protocol to better identify the candidate seed which the client used to generate M_1 and C_1 rather than using M_1 alone, as was the case in Lee et al. [73].

We outline the steps of our protocol, npRBC, which are illustrated in Figures 6.1 and 6.2. The steps in the protocol pertaining to Figure 6.1 are as follows:

1. Client/Server: The server/certificate authority (CA) performs a handshake with the client and tells it which cells to challenge in its PUF.
2. Client: The client uses this information to generate a 256-bit seed (S_1) for two purposes in Steps 3–4 below.
3. Client: S_1 is salted and then is used to generate a public/private key pair (P_{k1}/P_{r1}).
4. Client: S_1 is used as input to create a message digest using SHA3 (M_1) and ciphertext generated by AES256 (C_1).
5. Client: M_1 and C_1 are sent to the server for authentication.
6. Server: The server performs the RBC search (outlined in the steps below). After the

search finds the most likely seed, S', generated by the client it is used as input to a cryptographic algorithm (e.g., ECC, or a post-quantum cryptography algorithm), the client's public key is generated (P_{k1}) and registered on the registration authority.

The steps in the protocol pertaining to Figure 6.2 (the RBC Search Engine in Figure 6.1) are as follows:

1. The server receives M_1' and C_1' from the client.
2. The server reads the initial seed from the client's PUF image (S_{init}).
3. The server hashes S_{init} to create a message digest M and checks if it matches M_1' ($M \approx M_1'$) received from the client. Recall that M_1' is the corrupted variant of M_1 (see Section 6.3.3 for details).
4. If $M \not\approx M_1'$, then go to Step 10.
5. If the algorithm continues, then the next seed is generated and the process restarts at Step 3 above, but S is permuted and then hashed (S_{init} is only used on the first iteration).

6. If $M \approx M_1'$ then we generate the ciphertext, C.
7. The ciphertext, C, is checked for a match with the ciphertext received by the client ($C \approx C_1'$). Recall that C_1' is the corrupted variant of C_1 (see Section 6.3.3 for details).
8. If $C \approx C_1'$ then the seed, S_c, is added to the candidate set and the termination criteria are checked and if the algorithm continues then go to Step 5 above, otherwise go to Step 10.
9. If $C \not\approx C_1'$ then go to Step 10.
10. The termination criteria are checked (see Section 6.3.4 for details). If the algorithm terminates, then all the candidate seeds are checked and those with the highest probability of being correct are selected (see Section 6.3.5 for details). A seed is selected (S_s) and is salted to create S'.

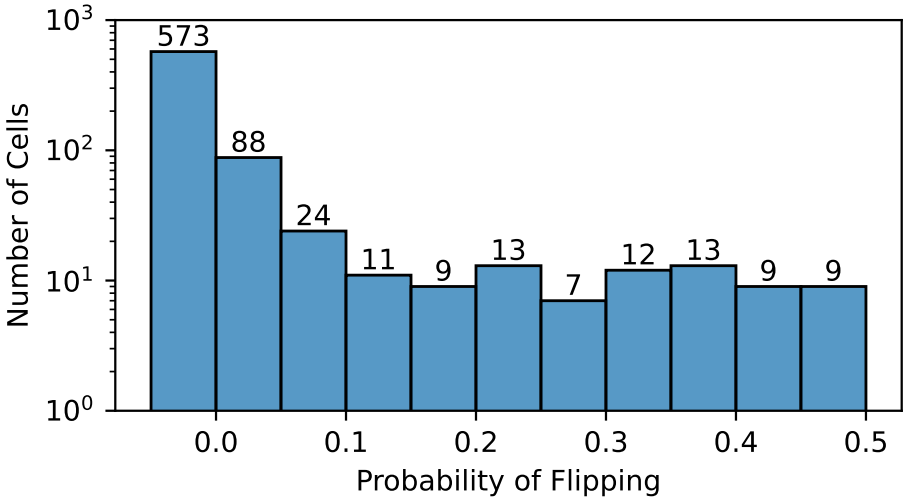


Figure 6.3: This histogram shows the probability of a bit flipping for the PUF used in our experimental evaluation, which is an ISSI 61-64WV6416BLL SRAM chip. There are a total of 768 enrolled cells that are stored in the PUF image on the secure server. While 573 of those cells are stable and will not change between authentication sessions, the other 185 cells have up to a 0.5 (or 50%) probability of their state changing when challenged during authentication.

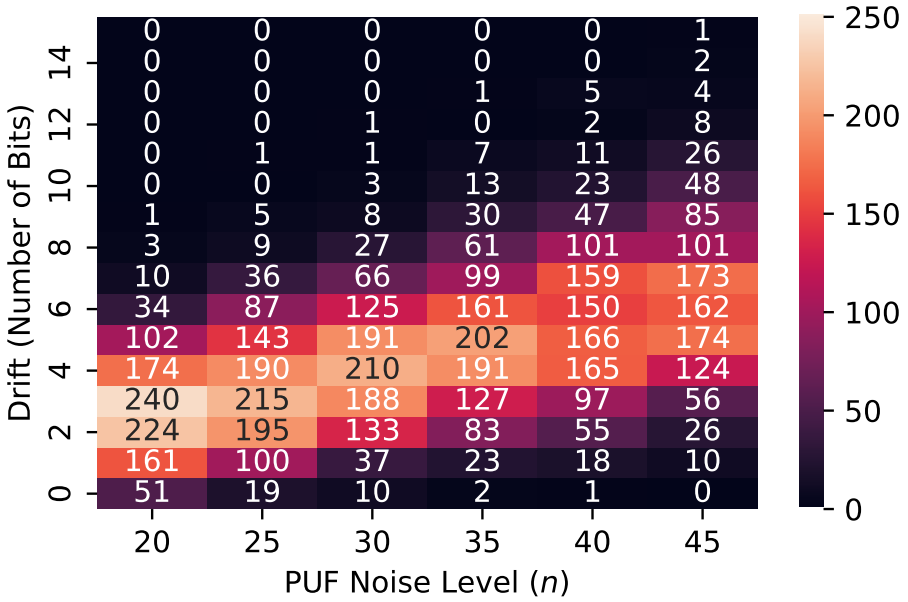


Figure 6.4: This heatmap plots the number of samples at each drift value for each PUF noise (n) using the PUF outlined in Figure 6.3. There are 1,000 trials for each n value with each trial selecting unstable cells at random from the PUF. The intensity of the heat map is the number of trials for each selected n value that had the corresponding amount of drift.

6.3.1 Probabilistic Searches of the PUF Seed Space

The cells of the PUF generate a response when challenged, where each cell's response is either a 0 or 1. During enrollment, the cells of the PUF are challenged numerous times and the responses are recorded to produce a PUF image. In the PUF used in our evaluation, most cells are stable; Figure 6.3 shows that the PUF has 573 bits that do not change between consecutive challenges, but some cells have a porbability of yielding either 0 or 1. During the authentication process, both the server and the client select the same cells (and in the same order) for their respective seeds based on the handshake that is initiated by the server. If all of the cells selected are stable, then the client and the server share the same initial seed. If cells are selected that vary between challenges then there is a probability that the server's and client's seeds will not match. In such a case, a search is performed on the server to determine which bits have drifted in the client's PUF relative to the image stored on the server. The drift of a seed is the number of bits that differ between the client's seed and the seed image on the server (this is the Hamming distance between the bits of the client's seed and the bits of the seed image).

PUF Noise Level (n) – As described above, the server selects which addresses the client reads from its PUF. Therefore, the server selects some number of unstable cells (cells having a non-zero probability of flipping). We denote the number of unstable cells selected to be n (which is the PUF noise level). For example a PUF noise level of $n = 20$ indicates that a seed is generated using 20 cells from the PUF that are not stable and have a chance of producing a different state compared to the image on the server. Figure 6.4 shows the amount of drift that occurs with varying PUF noise level values (n) for the PUF used in our evaluation. For example, for $n = 20$, there were 240 seeds out of 1000 where there was 3 bits of drift from the server's seed recorded in the PUF image (this is a Hamming distance of 3 between the PUF image and client's seed). As n increases, the amount of drift increases as well. Individual seeds with higher drift are less likely to occur, but at each level of drift there are exponentially more possible seeds, so while the individual seed with that level of drift has

a low probability of occurring, the set of seeds with that level of drift may be more likely. This explains why higher noise levels ($n > 30$) rarely have seeds with low (≤ 1) drift levels.

Accumulating Probability – The probability of each bit flipping in the client's seed is determined from the enrollment data on the server. This information is used to determine the likelihood of an individual seed being generated from the set of selected cells. The single most likely seed to be correct is the one where no bits have drifted relative to the server's image. As the probabilistic search proceeds, the probability of each searched seed is added to a sum. This sum yields the total probability of the correct seed having been found. Individual seeds that have more drift, i.e. the number of bits in the client's seed that have flipped, are less likely to occur than seeds with low drift.

Search Order – The probabilistic search checks seeds in order of highest to lowest probability of matching. This allows the search to prioritize the seeds which have a higher chance of being correct while disregarding seeds that have an infinitesimally small probability of being correct (e.g., a seed with 15 bits that drifted would never be searched because the probability of that occurring is too low to be worth considering). This search order leads to probability accumulating quickly at the beginning of the search and then to diminishing returns as the search continues. This search strategy is much more work efficient than prior work that assumes all bits in a seed have the same probability of flipping [21, 23, 72, 93, 117, 118].

6.3.2 Noisy Transmission

The transmission from the client to the server is vulnerable to corruption by environmental noise. Our protocol allows for the client to still be authenticated despite high environmental noise which flips bits in both the message digest and ciphertext. The protocol accepts a transmission bit error rate (TBER) up to a set threshold, t, in both the message digest and the ciphertext to account for corruption during transmission.

6.3.3 Match Criteria

A noisy/jammed environment will corrupt the signal from the client to the server resulting in M_1' and C_1' which are corrupted variants of the intended transmissions. The Hamming distance, d, between the corrupted variants and server generated variants is used to determine their similarity such that $d_M = \text{dist}(M, M_1')$ and $d_C = \text{dist}(C, C_1')$ where the dist() function is used to compute the Hamming distance between two sets of bits. The match criteria sets the threshold value, t, so that if $d \leq t$, then the match is considered true and the seed, S which is was used to generate M and C is added to the candidate set. This allows the RBC Search Engine to account for a given amount of noise during transmission and to still authenticate the client without having a perfect match, where $d_M = d_C = 0$. The server continues to generate seeds and match them until a termination criterion has been met.

6.3.4 Termination Criteria

There are two factors that determine when the search terminates. First, a probability threshold of 0.999 is used to terminate the search once the sum of the probabilities of the searched seeds reaches the threshold. Second, a time limit is used to terminate the search in cases where the sum of the probabilities has not accumulated to the threshold of 0.999. After the termination criteria has been met, the server then selects the best seed from the candidate set.

6.3.5 Candidate Seed Refinement

When the server accepts a high level of noise during transmission a large number of seeds are added to the candidate set. To choose the correct seed, S, from all of the candidates the server selects the seed with the lowest overall transmission error, $d_{total} = d_M + d_C$. The probabilistic search only examines a small fraction of the total possible seeds and so the probability of an incorrect seed on the server creating both a message digest (M) and ciphertext (C) that is corrupted during transmission to be closer to the client's M_1' and C_1'

Table 6.1: Parameter values used in the evaluation. Varied refers to whether the parameter is varied in the evaluation.

Parameter	Value	Varied
Probability Threshold (t)	0.999	
Time Limit	5 s	
TBER	10-44%	✓
PUF Noise Level (n)	20-45 bits (7.81-17.6% BER)	✓

Table 6.2: Average execution times and authentication rates for PUF noise levels $n = 20 - 45$, TBER levels 10 – 40% and time limit of 5s.

PUF Noise (n)	20 Bits				25 Bits				30 Bits				35 Bits				40 Bits				45 Bits			
TBER (%)	10	20	30	40	10	20	30	40	10	20	30	40	10	20	30	40	10	20	30	40	10	20	30	40
Time (s)	0.37	0.36	0.37	0.37	1.60	1.60	1.62	1.61	4.56	4.56	4.56	4.56	4.56	4.56	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
Succ.	0.99	0.99	0.99	0.88	0.99	0.99	0.99	0.77	0.98	0.98	0.98	0.67	0.84	0.84	0.84	0.57	0.58	0.58	0.58	0.39	0.28	0.28	0.28	0.00

than the correct seed's M and C is negligible. Requiring C_1' and lengthening M_1' and C_1' increases the probability of identifying the correct seed in high TBER scenarios.

6.4 Experimental Evaluation

6.4.1 Experimental Methodology

All experiments are conducted on a platform containing 2× AMD EPYC 7542 CPUs (64 total physical cores) clocked at 2.9 GHz with 512 GiB of main memory. All code is written in C/C++. All executions of the program are parallelized using OpenMP and employ 64 threads/cores as we found this to achieve the best performance on our platform.

PUF Used in the Evaluation – We use the SRAM PUF with characteristics outlined in Figures 6.3 and 6.4. In the protocol we select up to $n = 45$ unstable bits out of a 256 bit seed, where the remaining bits are stable. Note that while there are numerous PUF technologies, the SRAM PUF employed here has very high levels of noise. For instance, a Magnetic RAM PUF yielded a 7.7% BER [86], which is lower than the 17.6% BER here (or $n = 45$ of 256 bits). Also, a Resistive RAM (ReRAM) PUF has been reported to have a BER of 0.001 (only 1 in 1000 bits are unstable) [22]. Thus, our PUF is representative of having a substantial BER which is the worst case scenario for the protocol. Consequently

the results are applicable to PUFs with similar or lower levels of noise.

Fixed Parameters for Experimentation – A number of parameters are fixed for experimentation to allow for a detailed examination of parameters that significantly change the authentication rate of the protocol. The threshold, t, of transmission noise acceptance is fixed to 50% to account for TBER $\leq 44\%$ (see Section 6.3.2). Lower threshold values do not improve authentication rates (nor do they degrade it as long as they are $\geq 5\%$ above the TBER) while higher threshold values ($t \geq 50\%$) result in a lower authentication rate. A fixed time limit of 5 seconds and an accumulated probability of 0.999 is used as the termination criteria (see Section 6.3.4). A ciphertext of 1024-bits is used as it was experimentally found to have the best results on our platform. Additionally, SHA3-512 is used instead of SHA3-256 because the longer message digest increased the authentication rate in our experiments.

Monte Carlo Parameter Sampling – We employ a Monte Carlo approach where each combination of parameters given in Table 6.1 is trialed 1000 times. Each trial uses the trial number (1 to 1000) to set the seed for the random number generator. The random number generator determines which cells are selected for the seed and which bits flip during transmission (based on the selected TBER). The number of bits that flip during transmission follows a binomial distribution centered on the average number of bits that flip due to the TBER. This is modeled as Additive White Gaussian Noise (AWGN) resulting in a maximum TBER of 50%, which is Shannon's Limit [104].

6.4.2 Experimental Results

We simulate different PUF seeds with the Monte Carlo approach described above and report the results in Table 6.2. The results show that, up to a TBER of 30%, the determining factor for a successful authentication is the amount of PUF noise (n). Noise levels $n = 20-30$ bits with TBER $\leq 30\%$ successfully authenticate $\geq 98\%$ of the time. As the n value increases to 35, 40 and 45 bits the fraction of successful authentications with TBER of 30% drops from 0.98 to 0.84, 0.58, and 0.28, respectively. Additionally, the average time to complete

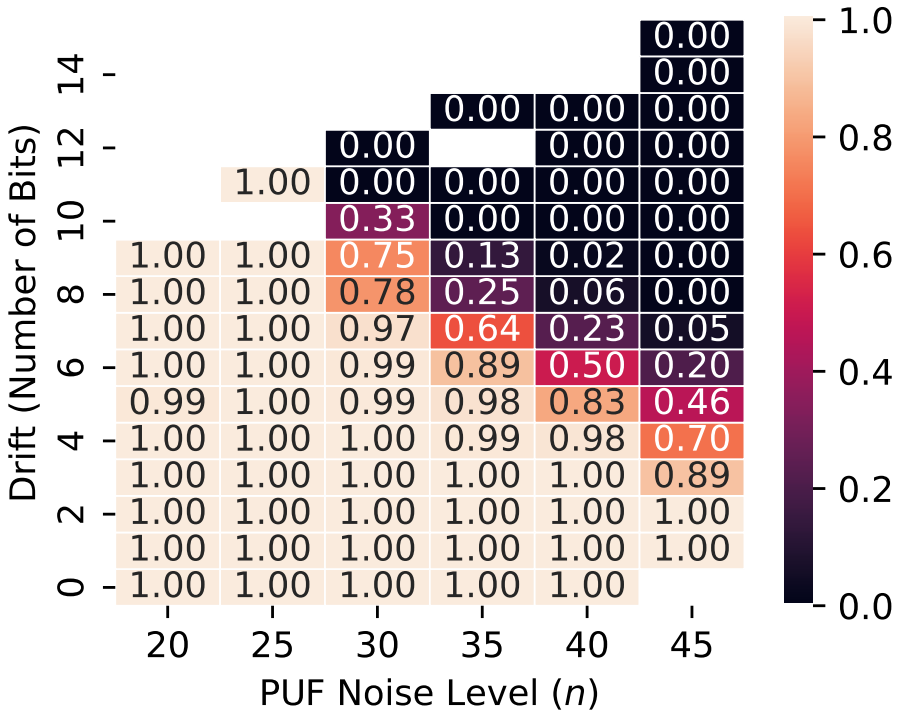


Figure 6.5: The heatmap plots the success rate of authentication for each PUF bit error rate (n) and drift with a TBER of 30%. Blank values in the heatmap indicate that there were no trials that had that combination of n and drift (see Figure 6.4). A success rate of 1.00 indicates that it always succeeded to authenticate with the given parameters, while a success rate of 0.00 indicates that it never succeeded to authenticate.

the search increases as a function of n. The time limit only impacts the success rate of the searches when $n \geq 35$. Recall from Figure 6.4 that the higher the PUF BER (n), the higher the average drift. The increase in drift requires more computation to find the correct seed. This is reflected in the average execution time values in Table 6.2. In the scenarios where the search is terminated by the time limit, the probability of finding the correct seed is lower because an insufficient number of seeds are searched.

In Figure 6.5 the success rate is plotted for each n and corresponding drift value. Every seed with a drift of 2 or lower is found while every seed with a drift of 12 or higher is not found. Additionally, the higher n values fail to find the correct seed at drift levels where lower n values succeed. This is due to the probabilistic search having to search through more possibilities at each drift value. From Table 6.2 we observe that this results in the time limit

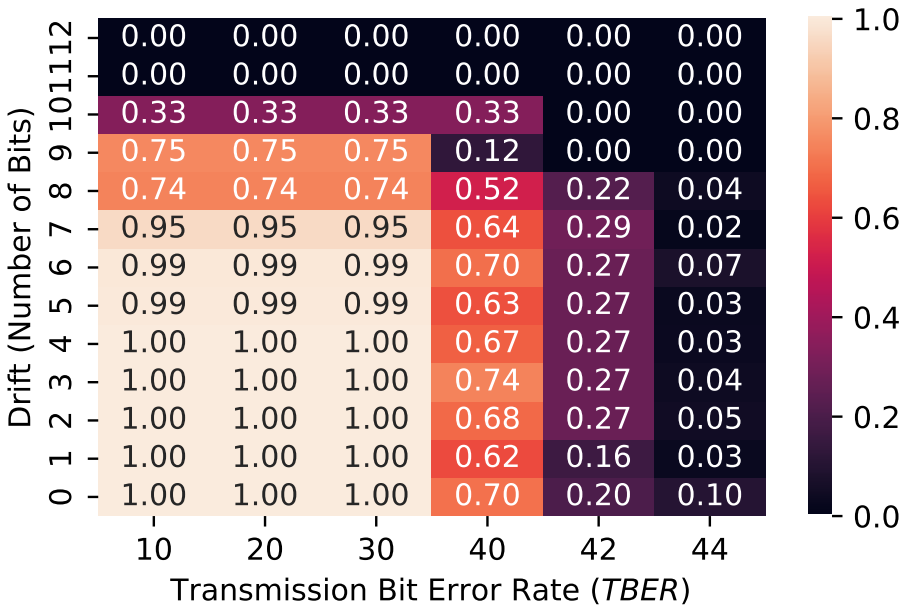


Figure 6.6: The fraction of successful authentications are plotted vs. the TBER (the chance of each transmitted bit flipping in the message digest or ciphertext) for a PUF noise level (n) of 30%.

of 5 seconds being reached, and the search terminating before the correct seed is identified.

While a TBER below 30% does not impact the authentication rate for a given n value, as the TBER increases past 30% it begins to be the limiting factor in authentication as opposed to n. Recall from Section 6.3.5 that the seed which is selected from the candidate set is the one with the lowest transmission error. In Figure 6.6 we observe that as the TBER approaches 44% the success rate rapidly decreases. We found that this is because we cannot identify and select the correct seed from the candidate set because of the high transmission error. High TBER compounds the issue with high n, resulting in poor authentication rates for seeds generated with a high n coupled with a high noise environment. The protocol mitigates this by selecting PUF cells that generate seeds with lower noise levels (n) in environments with high transmission bit error rates.

Leveraging Compute Power for Security – The two factors contributing to the success rate of NPRBC are the PUF Noise and the TBER. Increasing the compute power available increases the number of seeds searched which increases authentication rates at higher n

values. This has the added affect of increasing the overall security because a higher n value (corresponding to a noisier PUF) is harder to attack. Increasing the time limit also allows more seeds to be searched but gives attackers additional opportunity to compromise the system and adds latency to time sensitive communications. The more compute power available on the server, the more secure npRBC becomes. We reiterate that because the server has secret information regarding the client (the PUF image), compared to the number of seeds searched by the server, the search space for an attacker is intractable (2^{256} seeds).

6.5 Discussion & Conclusion

npRBC enables rapid device authentication in congested electromagnetic environments that have been previously found to be too noisy for canonical (zero-noise tolerance) cryptographic protocols [21, 23, 72, 74, 93, 117, 118]. We evaluated npRBC using a range of PUF noise levels $n = 20 - 45$ and transmission bit error rates (TBER) of 10–44%. Recall that the 44% limit refers to where each bit in the transmission of the message digest or ciphertext has a 44% chance of flipping. We evaluated the protocol by using a Monte Carlo approach that varied which bits are selected from the PUF and which bits flip in the transmitted data (message digest and ciphertext). With low PUF noise ($n \leq 30$), the protocol is successful in almost every trial. Non-probabilistic search methods are unable to authenticate in as noisy an environment as npRBC because they search too many seeds which increases the difficulty of distinguishing the correct seed from all of the candidate seeds. Additionally, non-probabilistic searches are intractable for seeds with higher drift values as shown in previous work [74]. npRBC successfully authenticates with a drift of up to 11-bits which for previous works requires more than 6.2×10^{18} seeds to be searched and would require >20 years to authenticate on a modern multi-GPU server node.

Chapter 7

GPU-Accelerated Authentication in High Noise Environments

The previous chapter introduced a protocol for RBC which uses probabilistic searching in a high noise environment. One of the benefits of npRBC-CPU is that increasing the compute power available on the secure server can increase both the level of security and ability to authenticate in noisy environments. This chapter examines using hardware accelerators to increase the performance of npRBC-GPU and assesses the impacts this has on authentication rates and security.

The contents of this chapter are currently submitted for publication.

7.1 Abstract

The use of parallel processing, and in particular, General Purpose Computing on Graphics Processing Units (GPGPU) can be exploited to expand the design space of security protocols. Higher computational throughput allows for the design of protocols that require significant computing power and are thus intractable for low-powered client devices that are susceptible to attacks by opponents. In this paper, we accelerate the Response-Based Cryptography protocol that authenticates low-powered client devices in environments with high levels of noise. Our approach offers (*i*) faster authentication, (*ii*) authentication that is robust to high levels of noise in the environment, and (*iii*) increased levels of security.

7.2 Introduction & Background

The field of computer security has yet to realize many of the benefits of general purpose computing on graphics processing units (GPGPU). The parallel processing capabilities of the GPU can be used to improve the performance of particular operations, where reduced latency yields greater levels of security. A recent successful example is the GPU parallelization of post-quantum cryptography algorithms [60, 72, 105, 112] which are well-known to be computationally expensive.

In this paper, we examine exploiting the GPU to reduce the latency of the response-based cryptography (RBC) protocol [117], and as we will show, while the major computational task that the protocol carries out is a probabilistic search using Hamming distance, the probabilistic nature of the search makes several aspects of the algorithm challenging to efficiently design for GPU architectures.

7.2.1 Response-Based Cryptography (RBC)

We present a high-level description of the probabilistic RBC protocol but do not provide in-depth detail because the aspect most relevant to this paper is the search that corrects error. See Donnelly & Gowanlock [36] for more information, which conducted the probabilistic RBC search on multi-core CPUs.

The RBC protocol authenticates client devices using a secure server. The protocol departs from many other cryptographic protocols by generating the inputs (hereafter referred to as seeds) of cryptographic methods using Physically Unclonable Functions (PUFs) [54] which are hardware devices that are embedded in client devices (e.g., phones, drones, UAVs, laptops, among others). In contrast, most cryptography algorithms store private keys in non-volatile memory, and so they are susceptible to being appropriated by an attacker. This is the benefit of PUFs — they enable a wider range of protocols that can be designed for use cases where client devices are at risk of being acquired by an opponent.

Before a PUF is integrated into a client device, an image of it is produced, which contains

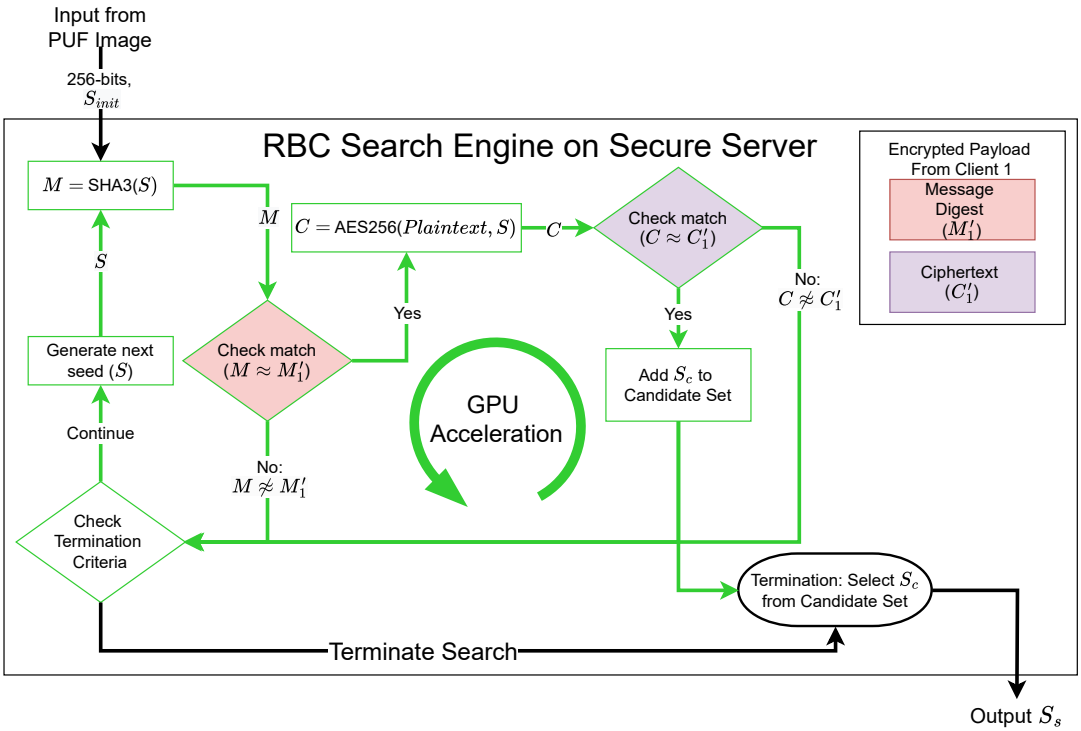


Figure 7.1: Probabilistic RBC search engine executed on the GPU used to authenticate Client 1. The major task for the search engine is hashing a seed using SHA3 to create a message digest (M) and comparing it to the client's message digest (M'_1) and if these are equivalent, a second check is performed using AES256. Client 1's encrypted payload that was received by the server will vary due to drift in the client's PUF and/or due to transmission error. Tasks computed on the GPU are outlined by the green lines. Terminating the search using a $T = 5$ s time limit occurs on both the host and GPU.

a representation of each of its memory cells. PUFs exploit small manufacturing variations in memory cells to uniquely identify each PUF device. PUFs have been developed using many memory technologies including SRAM [32], ReRAM [69], and MRAM [86].

Regardless of the PUF technology employed, the RBC protocol requires that a client device generate encrypted data using a seed generated by the PUF. This encrypted data could be ciphertext (e.g., using AES [39]), a public key (e.g., using CRYSTALS-Kyber [16]), and/or a hash/message digest (e.g., using SHA3 [38]). In this paper, we use a message digest generated with SHA3, and ciphertext generated by AES, but we refer to this simply as the "encrypted payload". Then, to authenticate a client device, the encrypted payload is transmitted to a secure server that authenticates the client by reading the client's associated PUF image to produce a PUF seed, which it then uses to generate the same information

contained in the encrypted payload. Next, the server determines if the encrypted data are the same, which would indicate that the client is to be authenticated. However, PUFs are susceptible to variation and a given bit in a memory cell has a probability that it will flip (a zero becomes a one and vice versa). This is the major challenge of RBC, as the server must perform a similarity search on a Hamming distance space to find the correct seed. Also, the RBC search is used to correct noise during transmission between the client and server in challenging electromagnetic environments.

The probability of each bit flipping can be exploited to perform a *targeted search* [36] instead of a *brute-force search* that assumes all bits have the same probability of flipping [72, 74, 117]. However, a *targeted search* requires partitioning the search space (the total space is 2^{256} seeds) and we outline the challenges of GPU algorithm optimization in this context.

7.2.2 Drawbacks and Opportunities

We outline two drawbacks (D1-D2) of accelerating the probabilistic RBC search using the GPU which stem from its throughput-oriented architecture.

D1 The targeted search attempts to search seeds that are likely to be correct based on the probability that a bit stored at a PUF memory address is likely to flip. Thus, the search space is partitioned based on these probabilities. These targeted searches produce a range of workloads that cannot be processed by a single (or few) long duration kernel invocation(s). Thus, while the GPU excels at throughput-oriented workloads (i.e., kernels with significant work), the targeted search generates work that varies in terms of the number of seeds searched, where small workloads risk underutilizing GPU resources.

D2 As described above, the total search space is practically infinite, so search time limits are imposed, and in this paper, we set a time limit of T=5 s, consistent with prior work [36, 73]. The search time limit means that the search terminates early if the seed

is not found, and so the algorithm needs to actively monitor the time limit on the host and GPU. To efficiently terminate the search early, it is best to have fine-grained workloads on the GPU, such that a long kernel invocation does not delay termination. Consequently, fine-grained workloads are preferred, but these may underutilize GPU resources.

To address D1-D2, this paper proposes Noisy Probabilistic Response-Based Cryptography on the GPU (npRBC-GPU) and makes the following contributions (C1-C2).

C1 We optimize npRBC-GPU for the spectrum of workload sizes described above. Optimizations are as follows: (*i*) Using a large number of CUDA streams[1] to increase workload granularity and to allow for concurrent execution of host (CPU/main memory) and GPU tasks. (*ii*) We employ unified memory for synchronizing between the host and GPU to signal when the search should be terminated early. (*iii*) We exploit constant and shared memory in the algorithms to reuse data and thread state.

C2 We demonstrate that despite the challenges above, our algorithm outperforms prior work that used multi-core CPUs, by increasing the amount of error that is corrected and decreasing authentication latency.

The paper is organized as follows. Section 7.3 outlines the proposed algorithm and associated optimizations. Section 7.4 presents the experimental evaluation. And lastly, Section 7.5 concludes the work and discusses future research directions.

7.3 npRBC-GPU: Accelerating Probabilistic Response Based Cryptography

Figure 7.1 shows the search process that occurs on a secure server that authenticates a client device. We highlight here that *we are not proposing a new security protocol*; rather,

[1]We use CUDA terminology throughout this paper.

we are accelerating the search process using the GPU as outlined by the steps with the green outlines in Figure 7.1. The main computation is hashing permuted seeds with SHA3, and a subset of the seeds that pass this filter ($M \approx M_1'$) will be encrypted using ciphertext. Of those, some will be added to the candidate set. We refer to Figure 7.1 when describing NPRBC-GPU below and provide brief summarizes of our optimizations. A single GPU kernel executes the components in Sections 7.3.1–7.3.3 below.

7.3.1 Hashing with SHA3-512

While hashing with SHA3-256 is secure for our purposes here, we use SHA3-512 because the increased length of the output (512 vs. 256 bits) increases robustness to higher transmission noise levels at minor expense to the response time.

GPU Optimization Summary: We optimize SHA3-512 on the GPU by storing the state needed by each thread (1600 bits) in shared memory, which improves performance because it limits access to global memory. While shared memory usage may seem like it might limit occupancy (the maximum number of active warps on a streaming multiprocessor), we find that the number of registers needed per thread limits occupancy rather than shared memory usage.

7.3.2 Encryption with AES256

We design an AES256 implementation that is efficient for encrypting small rather than large amounts of data (the latter is the typical use case for AES256). We use a Cipher Block Chaining (CBC) method for ciphertexts which are over 128 bits in length. We use a fixed ciphertext of 1024 bits which is 8 blocks in AES256. Each 128 bit block must be encrypted sequentially for CBC which is well-suited for our algorithm because we are using individual threads to encrypt a large number of ciphertexts rather than the more common use of encrypting a single large ciphertext with multiple threads.

GPU Optimization Summary: We use constant memory tables on the GPU that are

designed for short encryptions (few blocks) rather than for large encryptions (many blocks). Also, we use Instruction Level Parallelism (ILP) to improve AES256 performance. This is critical for reducing the number of registers required per thread, which can limit occupancy.

7.3.3 Fast Seed Permutation

We use Algorithm 515 [18] to permute the server's initial PUF image seed (S_{init}) and subsequent seeds (S).

GPU Optimization Summary: To prevent redundant calculations across threads, a lookup table for Algorithm 515 is used and stored in global memory. This reduces the computation needed to create each seed permutation.

7.3.4 Monolithic to Fine-grained Workloads & Kernels

In our initial algorithm design, we executed a small number of kernels as a function of the maximum anticipated PUF bit error rate (BER). The BER refers to the number of bits in a 256 bit seed that have flipped relative to the baseline in the PUF image (S_{init}). However, we find two major drawbacks of this approach: (i) The host sets a flag in GPU global memory to signal the GPU threads to return. This implies that even if threads launch and then immediately return, we still have to execute numerous (nearly) no-op CUDA blocks, which delays the kernel returning to the host. (ii) The monolithic kernels require a lot of accompanying information be sent to the device that outlines what each thread computes which caused significant register pressure.

GPU Optimization Summary: We decompose large kernels into several kernels with smaller CUDA grid sizes where the diversity of workloads assigned to the threads in a kernel are minimized. This decreased the amount of data that needs to be sent to each kernel and reduces register pressure.

Table 7.1: Parameter values and notation (Not.), where we outline if the parameter is varied or static.

Parameter	Not.	Value	Static	Varied
Probability Threshold	t	0.999	✓	
Time Limit	T	5 seconds	✓	
Transmission Bit Error Rate (TBER)	m	10-44%		✓
PUF Noise Level	n	20-45 bits		✓
Resulting PUF Drift	d	$d \propto n$	N/A	N/A

7.3.5 Concurrent Streams for Imbalanced Workloads

Fine-grained workloads are largely imbalanced, where some kernels execute for longer durations than others.

GPU Optimization Summary: We execute numerous (asynchronous) kernels across CUDA streams which allows for: (*i*) the concurrent execution of kernels; and, (*ii*) host-side tasks to overlap with host-GPU communication and computation.

7.4 Experimental Evaluation

7.4.1 Experimental Methodology

Platform & Software – Experiments are conducted on a platform with 2× AMD EPYC 7542 CPUs (64 physical cores total) clocked at 2.9 GHz with 512 GiB of main memory equipped with an Nvidia A100 GPU. The host code uses C/C++ and is parallelized with OpenMP with 64 threads and the GPU code is parallelized using CUDA v.12.4.

PUF Image – We select up to n=45 unstable bits out of a 256 bit seed, or a maximum bit error rate (BER) of 17.6% from a SRAM PUF image (the remaining ≥211 bits are stable and have zero chance of flipping). The SRAM PUF has a very high level of noise compared to other PUF technologies, such as a ReRAM PUF that has a 0.1% BER (1 in 1000 bits are unstable) [22]. Thus, the results are applicable to PUFs with similar or lower levels of noise and represent a worst-case scenario where significant error needs to be corrected.

Monte Carlo Sampling & Parameters – We outline the parameters used in the evaluation in Table 7.1. Due to the probabilistic nature of bits that flip in the PUF, we use Monte Carlo sampling of bits in the PUF for a selected transmission bit error rate (the number of bits that flip in the encrypted payload when sending the data from the client to the server). We select n bits that have a non-zero chance of flipping, where some number $d \leq n$ will flip, which is referred to as the PUF drift. The transmission bit error rate (m) is reported as a percentage of the number of flipped bits in the encrypted payload due to noise in the transmission environment, where the maximum possible noise level is 50% according to Shannon's limit [104]. We examine a TBER up to m=44% as authentication is unreliable beyond this threshold.

Termination of the Search – Each seed searched is assigned a probability that it will authenticate the client. We accumulate these probabilities and abort the search if the total accumulated probability exceeds t=0.999. We also abort if the search time exceeds T=5 s. We highlight that a timeout this does not imply that the search fails; rather, the seed found with the highest probability of being correct is selected which often authenticates the client.

7.4.2 Implementation Configurations

7.4.2.1 npRBC-CPU

Prior work using the CPU parallelized using OpenMP with 64 threads/physical cores [36].

7.4.2.2 npRBC-GPU

Our proposed algorithm executed on the GPU. GPU kernels are configured to launch with 32 threads per block and each thread is assigned 1000 seeds to search. Both of these parameters were selected as they were found to achieve the best performance on our platform. As we will discuss in Section 7.4.5, by default we use 64 CUDA streams to maximize GPU resource utilization.

Selection of the Candidate Seed – We determine the best candidate seed by evaluating

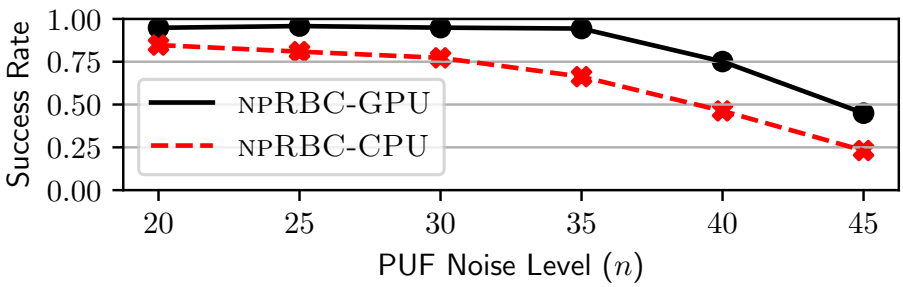
their Hamming distance to the encrypted payload received by the server. Both the SHA3-512 output (M) and AES256 output (C) generated by the seed, S, are compared with those received from the client. Unlike npRBC-CPU, GPU memory limitations necessitate selectively saving each potential candidate, S_c. To address this, we evaluate each candidate seed when it is generated and only save those that generate outputs that have the smallest Hamming distance to the encrypted payload. To select the candidate seed we use a semaphore on the GPU which was added to the libcuc++ library with CUDA v.11.0 [31].

Probability Accumulation & Kernel Invocation Search Order – As outlined in Section 7.2 compared to a brute-force search that assumes all bits in the PUF have an identical probability of flipping, the probabilistic search outlined here is a *targeted search*, and we exploit the information provided by each of the n bits selected from the PUF that have a non-zero probability of flipping. Thus, the search method proceeds by searching seeds with a high probability of generating the correct information in the encrypted payload. The probabilities of each individual seed being correct and authenticating the client are ordered from most to least probable, prioritizing seeds with a high chance of being correct. We batch the computation across numerous kernel invocations where each kernel invocation contains seeds to be evaluated that have a similar probability of authenticating the client.

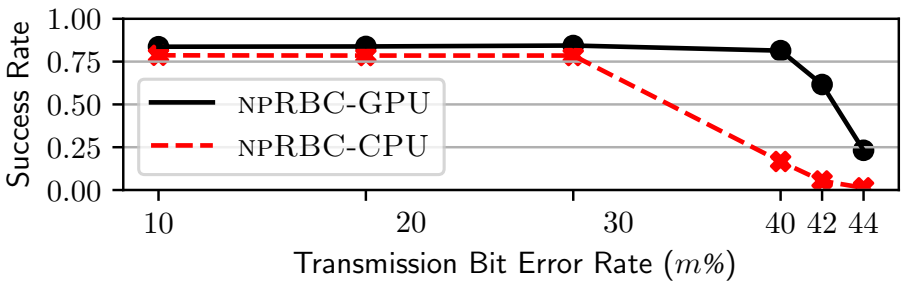
7.4.3 Determining PUF & Transmission Noise Limits

From Section 7.4.1, there are two parameters that are varied to examine the robustness of the system, which are the number of bits selected in the PUF with a non-zero probability of flipping (n) and the percentage of bits that flip due to transmission error in the environment (m). We examine at what PUF noise level (n) and transmission bit error rate (m) that communication becomes intractable with the T=5 s termination criterion using npRBC-CPU and npRBC-GPU.

Experimental Parameters: We use the cross product of n and m parameters where $n \in \{20, 25, \ldots, 45\}$ and $m \in \{10, 20, 30, 40, 42, 44\}\%$, yielding a total of 36 parameter



(a)



(b)

Figure 7.2: Limitations of mean search success rates between prior work (npRBC-CPU) and our work, npRBC-GPU. There are a total of 36k trials shown, see text for details. (a) Mean success rate as a function of n. (b) Mean success rate as a function of m (%), where the theoretical limit is $m = 50\%$.

combinations for n and m, and for each we carry out 1,000 Monte Carlo trials (36k trials total). We sample the n bits from the PUF, each of which has a non-zero probability of flipping, and sample differing bits that flip in the encrypted payload due to transmission error. For clarity, we selected $m \in \{42, 44\}$ to identify where the success rate of npRBC-CPU is $\approx$0%.

Figure 7.2 shows the results of the experiment, where Figure 7.2(a) shows the search success rate as a function of n and Figure 7.2(b) shows the search success rate as a function of m. npRBC-GPU outperforms the multi-core CPU implementation npRBC-CPU across all values of n and m. This is noteworthy as unlike npRBC-CPU, npRBC-GPU processes workloads that are not well-suited for GPU execution, particularly small workloads (low n and m) where GPU acceleration may be unwarranted. We elaborate on this in Section 7.4.5.

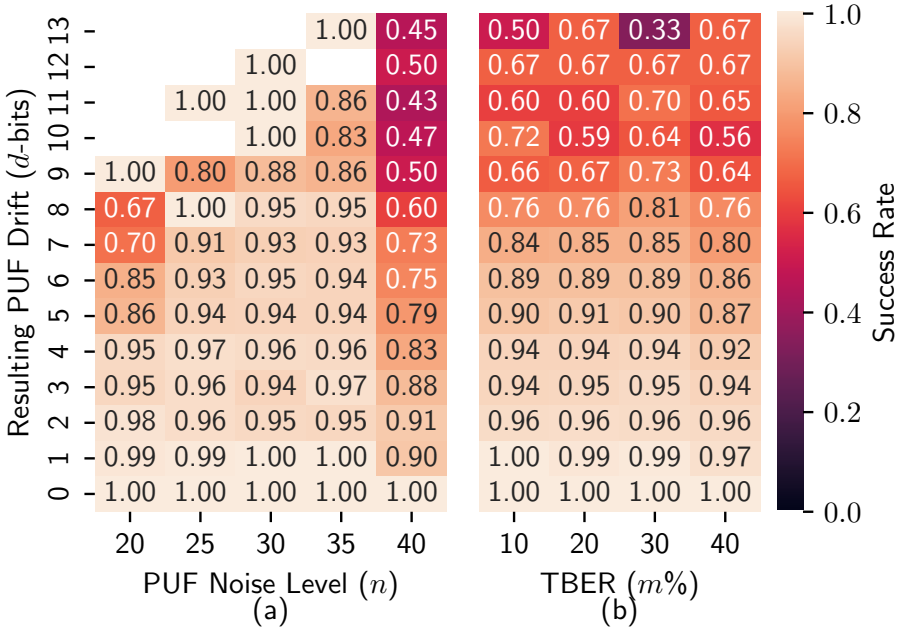


Figure 7.3: npRBC-GPU mean search success rate where we select $n \in \{20, 25, \ldots, 40\}$ and $m \in \{10, 20, \ldots, 40\}$. There are a total of 20 parameter combinations of n and m where each has 1,000 trials yielding a total of 20k trials. Each trial randomly samples PUF cells where there are n bits with a non-zero probability of flipping and m% of the bits are randomly selected to flip based on transmission error. (a) Search success rate as a function of PUF noise level (n) and the resulting PUF drift (d). (b) Search success rate as a function of TBER (m) and PUF drift (d).

Figure 7.2 also shows that npRBC-GPU does not have a 100% success rate for any value of n or m. This is somewhat misleading, because if we use Figure 7.2(a) as an example, at the data point for n=20 there are trials in the sample where m>40, which has a very low success rate as shown in Figure 7.2(b). Thus, if the values of n and m are restricted to more reasonable parameters, such as $n \leq 40$ and $m \leq 40\%$ then the success rate increases to $\approx$100% for many values of n and m for npRBC-GPU, but not for npRBC-CPU.

The PUF bit error rate (n) is controllable by the protocol and the maximum value for m=50%. Therefore, in all that follows, we examine $n \leq 40$ and $m \leq 40\%$.

7.4.4 GPU Search Success Rate as a Function of PUF and Transmission Error

The two parameters that are varied are n and m (Section 7.4.3). Some fraction of n PUF bits will have flipped, which is the resulting PUF drift, d; therefore, $d \propto n$. PUF drift is an

important statistic — high PUF drift implies there are more seeds to search compared to low PUF drift and intuitively, high PUF drift results in a lower authentication rate.

Figure 7.3(a) plots the success rate as a function of PUF noise level (n) and the resulting PUF drift (d). For each value of n, there are 4k total Monte Carlo trials. Observe from the plot that there are several cells that do not have any trials because $d \propto n$. For example, consider $(n, d) = (20, 13)$ which does not contain any trials. This is because it would be very improbable that of those 20 bits that have a non-zero probability of flipping, 13 flipped (see Figure 3 in Donnelly & Gowanlock [36] for the PUF probability histogram).

From Figure 7.3(a) we observe that npRBC-GPU has a reduced success rate when both the drift, d, and the noise, n, are high. This is owing to the increased number of seeds that need to be searched and the $T = 5$ s time limit being insufficient to find the seed. Note that the number of seeds that need to be searched increases proportional to n because the probability space is larger and therefore it is more challenging to find the correct seed even at lower drift, d. Similarly, seeds with high drift at low n are also challenging to find because the likelihood of such a high drift at low n is above the search probability threshold of t=0.999.

Figure 7.3(b) shows the search success rate is shown as a function of TBER (m). We observe that we achieve a $\sim$90% success rate when the PUF drift $d \leq 6$. Comparing Figure 7.3(b) to (a) it is clear that most of the drift values $d \geq 7$ are when the PUF noise $n \gtrsim 35$.

This is a major achievement as it demonstrates that high levels of noise (high n and m) can be corrected using GPU hardware within reasonable time constraints.

7.4.5 GPU vs. CPU Seed Search Throughput

The GPU has immense potential to improve the seed search throughput compared to the CPU. However, as described in Sections 7.2-7.3, the search requires numerous GPU optimizations to address the varying workloads associated with probabilistic searches. Several smaller kernels are needed to reduce the challenges associated with monolithic kernel invoca-

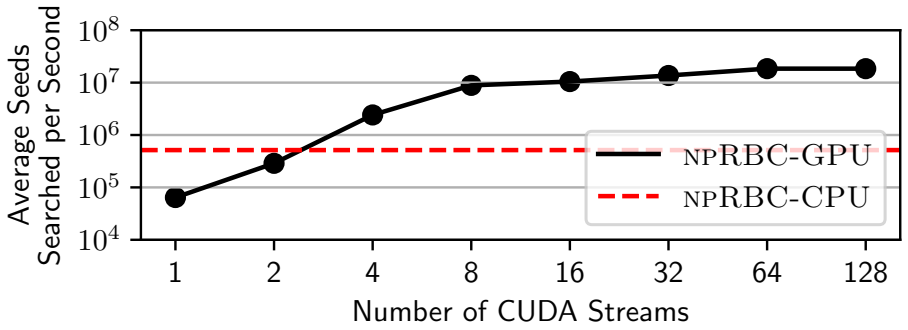


Figure 7.4: Seed search throughput (average number of seeds searched per second) plotted on a log scale for the experiment conducted in Section 7.4.4. The GPU curve shows search throughput vs. the number of CUDA streams. npRBC-CPU with 64 CPU threads/cores is shown for comparison.

tions; e.g., register pressure and reduced occupancy, load imbalance due to uneven workloads between threads, and higher overhead when terminating the search. Thus, we execute many smaller GPU kernels that compute fewer seeds per invocation using multiple asynchronous CUDA streams.

Figure 7.4 plots GPU seed search throughput as averaged across all 20k trials used in Figure 7.3. We find that one CUDA stream yields a throughput of $\lesssim 10^5$ seeds/s, whereas when using $\geq$16 streams, we achieve a throughput exceeding 10^7 seeds/s, which is an improvement of over two orders of magnitude. This enormous performance gain is counterintuitive, but it occurs for several reasons described as follows.

❶ The GPU kernels require a wide range of CUDA grid sizes and many of the kernels have small grid sizes and can execute concurrently on the GPU. Thus, using several CUDA streams allows for greater GPU resource utilization. ❷ The order that the kernels are executed are prioritized based on the likelihood that a given kernel contains the seed that will authenticate the client. Also, the search terminates after T=5 s. Thus, with only one stream the GPU may not get the opportunity to process very many kernels that yield high seed search throughput (kernels with larger CUDA grid sizes). In contrast, with numerous streams, the GPU processes more kernels, several of which compute significant work that increases the search throughput. ❸ The optimizations that cache data in constant or shared

memory (Sections 7.3.1–7.3.3) have a minimal impact on performance when a kernel has a small grid size. In contrast, when more streams are used it allows for processing more kernels with large grid sizes that benefit from data reuse across threads.

Figure 7.4 also compares the CPU and GPU. The search throughput using npRBC-CPU using 64 physical cores/threads are shown. The GPU with 64 streams achieves a throughput of 35.87× over the multi-core CPU implementation, indicating that despite uneven workloads, the GPU significantly outperforms the CPU in terms of raw search throughput.

7.4.6 Comparison of GPU and CPU Success Rates

Section 7.4.5 compared the CPU and GPU throughput, which provides an incomplete picture of the implications on search success rates. To address this, we execute the same experiment as that in Section 7.4.4 (summarized in Figure 7.3) but using npRBC-CPU. To highlight the differences in the success rates between the CPU and GPU, we subtract the GPU from the CPU success rate to create difference heatmaps as shown in Figure 7.5. Difference values $GPU-CPU>0$ imply that npRBC-GPU has a higher success rate than npRBC-CPU, whereas $GPU-CPU<0$ implies the opposite. We illustrate three trends from Figure 7.5 as follows.

❶ The major parameter ranges where the CPU yields a greater average success rate than the GPU occur when $n = 20-25$, $d \leq 5$, and $m \leq 30$. As described in Section 7.4.2.2, due to GPU memory limitations for storing candidate seeds, the algorithm makes a local rather than global decision as to whether a given seed should be added to the candidate set. Because all possible candidate seeds are not in the set, the GPU algorithm is sometimes unable to arrive at a global optimum — i.e., the correct seed is discarded when it should have been retained and the wrong seed is selected instead. ❷ The GPU has a much higher success rate at higher levels of PUF noise (n) and drift (d) than the CPU which is at n=30−40 and $d \geq 7$ (the upper right triangle in Figure 7.5(a)). This to due to the high search throughput of the GPU which is needed to find the correct seed within T=5 s. ❸ The GPU authenticates at

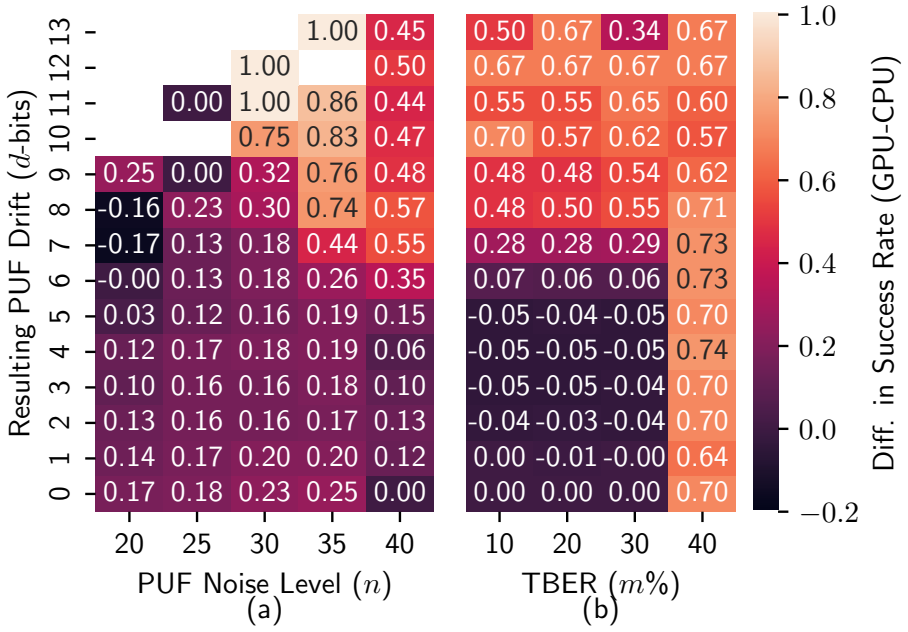


Figure 7.5: The same as that shown Figure 7.3, except that we show the difference in successful search rates between npRBC-CPU and npRBC-GPU. Positive values indicate that npRBC-GPU has a higher success rate than npRBC-CPU and negative values indicate the opposite.

much higher levels of transmission error (m) as shown when $d \geq 8$ or m=40% in Figure 7.5(b). Again, this is owing to the throughput advantage that the GPU has on searches with higher workloads. Specifically, when $d \geq 8$, the GPU has a 85.43× greater search throughput than the CPU.

Based on these results, it is clear that npRBC-GPU is superior to npRBC-CPU in cases where there is greater PUF noise and/or transmission error.

7.5 Conclusion

We have presented Noisy Probabilistic Response-Based Cryptography on the GPU (npRBC-GPU) which optimized the search process to correct error in PUF-generated seeds and transmission noise. Despite several aspects of the algorithm being poorly suited for GPU acceleration (e.g., register pressure/reduced occupancy, load imbalance due to uneven workloads, and overhead from search termination), we show that on average npRBC-GPU

enables much higher search success rates and enables communication at noise levels that are completely intractable for the multi-core CPU. A broader implication of this work is that the use of GPUs can expand the design space and use cases for security protocols; thus, future work includes investigating other security protocols that can benefit from GPU acceleration.

Chapter 8

Conclusion

This dissertation examines the challenges and innovations surrounding high-dimensional searches, focusing on both Euclidean and Hamming spaces. As data continues to grow in volume and complexity, traditional sequential search methods become increasingly inefficient due to the curse of dimensionality. To address this, we propose multiple novel indexing methods, computational optimizations, and parallel computing strategies to improve search performance in high-dimensional spaces.

Two of the primary contributions of this research is the development of the **Coordinate Oblivious Similarity Search (COSS)** and **Multi-Space Tree with Incremental Construction (MiSTIC)** methods. COSS introduced a metric-based indexing approach that reduces the reliance on coordinate values, making it particularly well-suited for high-dimensional Euclidean spaces where traditional coordinate-based tree and grid-based methods may perform poorly. MiSTIC further advanced this concept by combining metric-based and coordinate-based indexing strategies, leading to increased efficiency and robustness across different dataset characteristics. The experimental results demonstrate that these approaches significantly outperform existing state-of-the-art methods, particularly in scenarios where high-dimensionality renders traditional indexing techniques ineffective.

Combination generation is an intrinsic component of searching for both similarity searches and RBC. In our survey of combination generating methods we examine pioneering works and adapt them for modern multi- and many-core hardware platforms. We select the best

combination generating method for RBC and employ it in the works summarized in Chapters 6 and 7.

In addition to high-dimensional Euclidean searches and combination generation, this dissertation also explores **probabilistic searches in Hamming space**, specifically for applications in secure authentication and cryptographic key retrieval. **Noisy Probabilistic Response-Based Cryptography (npRBC-CPU)** is introduced as an approach to efficiently search for cryptographic keys in the presence of transmission errors and PUF (Physical Unclonable Function) noise. Given the growing importance of security in the digital age, this work contributes to the development of more robust and efficient authentication mechanisms that leverage probabilistic search strategies.

A major theme throughout this dissertation has been the role of **parallel and GPU computing** in enhancing search efficiency. Modern GPUs provide immense computational power, but their unique architecture requires careful algorithmic design to fully exploit their capabilities. In this dissertation, we demonstrate how parallelism is leveraged not only for accelerating similarity searches but also for cryptographic applications, where fast and efficient key retrieval is critical. By optimizing the algorithm, including memory usage, reducing computational overhead, and carefully managing GPU kernel executions, we achieved significant speedups compared to CPU-based implementations in addition to other state-of-the-art GPU methods.

The contributions of this dissertation have several implications for future research and practical applications:

- **Scalability for Large-Scale Data**: As data continues to grow in both size and dimensionality, the need for efficient search methods will only increase. The proposed methods provide a scalable foundation for indexing and searching high-dimensional data.

- **Hybrid Indexing Strategies**: The combination of metric-based and coordinate-based indexing, as demonstrated in MiSTIC, presents a promising direction for im-

proving search performance. Future work may explore dynamic indexing structures that adapt to specific dataset characteristics, optimizing both search efficiency and memory usage.

- **Combination Generation for High-Throughput Applications**: The categorization and comparison of combination generating methods enable the selection of methods which match application scenarios. This increases the efficiency and throughput across a wide range of applications which are reliant on combination generation.

- **Security and Cryptographic Applications**: The integration of high-speed search algorithms with cryptographic authentication highlights new possibilities for securing data in high-noise environments. Future research can explore ways to enhance security while maintaining computational efficiency, particularly in the era of post-quantum cryptography.

In conclusion, this dissertation contributes to the field of high-dimensional search algorithms by proposing novel indexing techniques, optimizing for parallel architectures, and paving the way for more efficient and secure search methodologies. By addressing key challenges in both Euclidean and Hamming space searches, this research provides a foundation for future advancements in similarity searches and cybersecurity.

Bibliography

[1] Marcel R Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. Streamkm++ a Clustering Algorithm for Data Streams. *Journal of Experimental Algorithmics*, 17:2.1–2.30, 2012.

[2] Selim G Akl. A comparison of combination generation methods. *ACM Transactions on Mathematical Software (TOMS)*, 7(1):42–45, 1981.

[3] Daichi Amagata, Takahiro Hara, and Chuan Xiao. Dynamic Set kNN Self-Join. In *IEEE 35th International Conference on Data Engineering (ICDE)*, pages 818–829. IEEE, 2019.

[4] AMD. 4th Gen AMD EPYC Processor Architecture. Technical report, Advanced Micro Devices, Inc., 2024. URL `https://www.amd.com/en/processor/epyc`. Accessed: 2024-11-25.

[5] Jean-Philippe Aumasson. Too Much Cypto. *Cryptology EPrint Archive*, 2019.

[6] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching For Exotic Particles in High-Energy Physics with Deep Learning. *Nature communications*, 5:4308, 2014.

[7] Georg T Becker, Alexander Wild, and Tim Güneysu. Security Analysis of Index-Based Syndrome Coding for PUF-Based Key Generation. In *2015 IEEE Intl. Symp. on Hardware Oriented Security and Trust*, pages 20–25, 2015.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings*

of the 1990 ACM SIGMOD international conference on Management of data, pages 322–331, 1990.

[9] Richard E Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton university press, 1961.

[10] Jon Louis Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[11] Jon Louis Bentley. Multidimensional Binary Search Trees in Database Applications. *IEEE Transactions on Software Engineering*, (4):333–340, 1979.

[12] S Berchtold, DA Keim, and HP Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking*, 451:28–39, 2001.

[13] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval* , 2011.

[14] Christian Böhm, Stefan Berchtold, and Daniel A Keim. Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[15] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. *ACM SIGMOD Record*, 30(2):379–388, 2001.

[16] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[17] Tolga Bozkaya and Meral Ozsoyoglu. Distance-Based Indexing for High-Dimensional Metric Spaces. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 357–368, 1997.

[18] Bill P. Buckles and Matthew Lybanon. Algorithm 515: Generation of a Vector from the Lexicographical Index. *ACM Transactions on Mathematical Software (TOMS)*, 3 (2):180–182, 1977.

[19] Walter A. Burkhard and Robert M. Keller. Some Approaches to Best-Match File Searching. *Communications of the ACM*, 16(4):230–236, 1973.

[20] Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot Selection Techniques for Proximity Searching in Metric Spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[21] B. Cambou. Unequally Powered Cryptography With Physical Unclonable Functions for Networks of Internet of Things Terminals. In *2019 Spring Simulation Conf.*, pages 1–13, 2019.

[22] Bertrand Cambou and Saloni Jain. Key Recovery for Content Protection Using Ternary PUFs Designed with Pre-Formed ReRAM. *Applied Sciences*, 12(4):1785, 2022.

[23] Bertrand Cambou, Michael Gowanlock, Bahattin Yildiz, Dina Ghanaimiandoab, Kaitlyn Lee, Stefan Nelson, Christopher Philabaum, Alyssa Stenberg, and Jordan Wright. Post Quantum Cryptographic Keys Generated with Physical Unclonable Functions. *Applied Sciences*, 11(6), 2021. ISSN 2076-3417.

[24] Bertrand Cambou, Christopher Philabaum, Jeffrey Hoffstein, and Maurice Herlihy. Methods to Encrypt and Authenticate Digital Files in Distributed Networks and Zero-Trust Environments. *Axioms*, 12(6):531, 2023.

[25] Přemysl Čech, Jakub Lokoč, and Yasin N Silva. Pivot-Based Approximate k-NN Similarity Joins for Big High-Dimensional Data. *Information Systems*, 87:101410, 2020.

[26] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid Tree: An Index Structure for High-Dimensional Feature Spaces. In *Proceedings of the 15th International Conference on Data Engineering*, pages 440–447. IEEE, 1999.

[27] Phillip J Chase. Algorithm 382: Combinations of m Out of n Objects. *Communications of the ACM*, 13(6):368, 1970.

[28] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in Metric Spaces. *ACM computing surveys (CSUR)*, 33(3):273–321, 2001.

[29] Lu Chen, Yunjun Gao, Xinhan Li, Christian S Jensen, and Gang Chen. Efficient Metric Indexing for Similarity Search. In *2015 IEEE 31st International Conference on Data Engineering*, pages 591–602. IEEE, 2015.

[30] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. Pivot-based Metric Indexing. *Proc. VLDB Endow.*, 10(10):1058–1069, June 2017.

[31] NVIDIA Corporation. Extended API Documentation for libcudacxx. Technical report, NVIDIA Corporation, 2024. URL `https://nvidia.github.io/cccl/libcudacxx/extended_api.html`. Accessed: 2024-12-12.

[32] Mafalda Cortez, Apurva Dargar, Said Hamdioui, and Geert-Jan Schrijen. Modeling SRAM Start-Up Behavior for Physical Unclonable Functions. In *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2012.

[33] Jerome Darbon, Bulent Sankur, and Henri Maitre. Error Correcting Code Performance

for Watermark Protection. In *Security and Watermarking of Multimedia Contents III*, volume 4314, pages 663–672. SPIE, 2001.

[34] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Helper Data Algorithms for PUF-Based Key Generation: Overview and Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(6):889–902, 2014.

[35] Brian Donnelly and Michael Gowanlock. A Coordinate-Oblivious Index for High-Dimensional Distance Similarity Searches on the GPU. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[36] Brian Donnelly and Michael Gowanlock. Authentication in High Noise Environments using PUF-Based Parallel Probabilistic Searches. In *To appear in the Proceedings of the 28th IEEE High Performance Extreme Computing Conference*, 2024.

[37] Stephane Durocher, Pak Ching Li, Debajyoti Mondal, Frank Ruskey, and Aaron Williams. Cool-lex Order and k-ary Catalan Structures. *Journal of Discrete Algorithms*, 16:287–307, 2012.

[38] Morris J Dworkin. SHA-3 standard: Permutation-based Hash and Extendable-Output Functions. *US Department of Commerce, NIST*, 2015.

[39] Morris J Dworkin, Elaine Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, and James F Dray Jr. Advanced Encryption Standard (AES). *US Department of Commerce, NIST*, 2001.

[40] MA Ben Farah, Ramzi Guesmi, Abdennaceur Kachouri, and Mounir Samet. A New Design of Cryptosystem Based on S-box and Chaotic Permutation. *Multimedia Tools and Applications*, 79:19129–19150, 2020.

[41] Thibauld Feneuil, Antoine Joux, and Matthieu Rivain. Shared Permutation for Syndrome Decoding: New Zero-Knowledge Protocol and Code-Based Signature. *Designs, Codes and Cryptography*, 91(2):563–608, 2023.

[42] Yilin Feng, Jie Tang, Meilin Liu, Chongjun Wang, and Junyuan Xie. Fast Document Cosine Similarity Self-Join on GPUs. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence*, pages 205–212. IEEE, 2018.

[43] Charles D Feustel and Linda G Shapiro. The Nearest Neighbor Problem in an Abstract Metric Space. *Pattern Recognition Letters*, 1(2):125–128, 1982.

[44] Benjamin Fuller, Xianrui Meng, and Leonid Reyzin. Computational Fuzzy Extractors. *Information and Computation*, 275:104602, 2020.

[45] Benoit Gallet and Michael Gowanlock. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. In *IEEE 29th International Conference on High Performance Computing, Data, and Analytics*, pages 135–144, 2022.

[46] Junhao Gan and Yufei Tao. DBSCAN Revisited: Mis-claim, Un-fixability, and Approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 519–530, 2015.

[47] Michael Gowanlock and Ben Karsin. Accelerating the similarity self-join using the GPU. *Journal of Parallel and Distributed Computing*, 133:107–123, 2019. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2019.06.005.

[48] Michael Gowanlock and Ben Karsin. GPU-Accelerated Similarity Self-Join for Multi-Dimensional Data. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–9, 2019.

[49] Michael Gowanlock, Benoit Gallet, and Brian Donnelly. Optimization and Comparison

of Coordinate- and Metric-Based Indexes on GPUs for Distance Similarity Searches. In *International Conference on Computational Science*, pages 357–364. Springer, 2023.

[50] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy. In *EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 674–704. Springer, 2020.

[51] Michael Greenspan and Mike Yurick. Approximate kd-Tree Search for Efficient ICP. In *Proceedings of the Fourth International Conference on 3-D Digital Imaging and Modeling*, pages 442–448. IEEE, 2003.

[52] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[53] Joseph M Hellerstein and Avi Pfeffer. The RD-tree: An Index Structure for Sets. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1994.

[54] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical Unclonable Functions and Applications: A Tutorial. *Proceedings of the IEEE*, 102(8): 1126–1141, 2014.

[55] Gisli R Hjaltason and Hanan Samet. Index-Driven Similarity Search in Metric Spaces (Survey Article). *ACM Transactions on Database Systems (TODS)*, 28(4):517–580, 2003.

[56] Maximilian Hofer and Christoph Böhm. Error Correction Coding for Physical Unclonable Functions. In *Austrochip, Workshop on Microelectronics*, 2010.

[57] Yun-Wu Huang, Ning Jing, Elke A Rundensteiner, et al. Spatial Joins using R-Trees: Breadth-First Traversal with Global Optimizations. In *VLDB*, volume 97, pages 25–29. Citeseer, 1997.

[58] Edwin H Jacox and Hanan Samet. Spatial Join Techniques. *ACM Transactions on Database Systems (TODS)*, 32(1):7, 2007.

[59] Hosagrahar V Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[60] Xinyi Ji, Jiankuo Dong, Tonggui Deng, Pinchang Zhang, Jiafeng Hua, and Fu Xiao. HI-Kyber: A Novel High-Performance Implementation Scheme of Kyber Based on GPU. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[61] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021. doi: 10.1109/TBDATA.2019.2921572.

[62] Dmitri V Kalashnikov. Super-EGO: Fast Multi-Dimensional Similarity Join. *The Very Large Database Journal*, 22(4):561–585, 2013.

[63] Dmitri V Kalashnikov and Sunil Prabhakar. Similarity Join for Low- and High-Dimensional Data. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings.*, pages 7–16. IEEE, 2003.

[64] Dmitri V Kalashnikov and Sunil Prabhakar. Fast Similarity Join for Multi-Dimensional Data. *Information Systems*, 32(1):160–177, 2007.

[65] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel Multi-Dimensional Range Query Processing with R-trees on GPU. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013.

[66] Donald E Knuth. *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[67] Zbigniew Kokosiński. Algorithms for Unranking Combinations and their Applications. In *Proc. 7th Intl. Conf. Parallel and Distributed Computing and Systems (PDCS)*, volume 95, pages 216–224, 1995.

[68] Zbigniew Kokosinski and Ikki-Machi Tsuruga. Unranking Combinations in Parallel. In *Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 79–82, 1996.

[69] Ashwija Reddy Korenda, Fatemeh Afghah, and Bertrand Cambou. A Secret Key Generation Scheme for Internet of Things using Ternary-States ReRAM-based Physical Unclonable Functions. In *14th Intl. Wireless Communications & Mobile Computing Conf.*, pages 1261–1266. IEEE, 2018.

[70] Vladimir Kruchinin, Yuriy Shablya, Dmitry Kruchinin, and Victor Rulevskiy. Unranking Small Combinations of a Large Set in Co-Lexicographic Order. *Algorithms*, 15(2): 36, 2022.

[71] Jerome Kurtzberg. Algorithm 94: Combination. *Communications of the ACM*, 5(6): 344, 1962.

[72] Kaitlyn Lee, Michael Gowanlock, and Bertrand Cambou. SABER-GPU: A Response-Based Cryptography Algorithm for SABER on the GPU. In *2021 IEEE 26th Pacific Rim Intl. Symp. on Dependable Computing*, pages 123–132, 2021.

[73] Kaitlyn Lee, Brian Donnelly, Michael Gowanlock, and Bertrand Cambou. Efficient Searches of Physical Unclonable Function Seed Spaces for Response-Based Cryptography. In *Autonomous Systems: Sensors, Processing and Security for Ground, Air, Sea, and Space Vehicles and Infrastructure 2023*, volume 12540, pages 112–125. SPIE, 2023.

[74] Kaitlyn Lee, Brian Donnelly, Tomer Sery, Dan Ilan, Bertrand Cambou, and Michael Gowanlock. Evaluating Accelerators for a High-Throughput Hash-Based Security Protocol. In *Proc. of the 52nd Intl. Conf. on Parallel Processing Workshops*, pages 40–49, 2023.

[75] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A Fast Similarity Join Algorithm using Graphics Processing Units. In *Proceedings of the 24th IEEE International Conference on Data Engineering*, pages 1111–1120. IEEE, 2008.

[76] C. N. Liu and D. T. Tang. Algorithm 452: Enumerating Combinations of m Out of n Objects. *Communications of the ACM*, 16(8):485, 1973.

[77] Jianquan Liu, Shoji Nishimura, and Takuya Araki. P-Index: A Novel Index Based on Prime Factorization for Similarity Search. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–8. IEEE, 2019.

[78] Youzhong Ma, Ruiling Zhang, Shijie Jia, Yongxin Zhang, and Xiaofeng Meng. An Efficient Similarity Join Approach on Large-Scale High-Dimensional Data using Random Projection. *Concurrency and Computation: Practice and Experience*, 31(20):e5303, 2019.

[79] Rui Mao, Willard L Miranker, and Daniel P Miranker. Pivot Selection: Dimension Reduction for Distance-Based Indexing. *Journal of Discrete Algorithms*, 13:32–46, 2012.

[80] Rui Mao, Peihan Zhang, Xingliang Li, Xi Liu, and Minhua Lu. Pivot Selection for Metric-Space Indexing. *International Journal of Machine Learning and Cybernetics*, 7 (2):311–323, 2016.

[81] Charles J Mifsud. Algorithm 154: Combination in Lexicographical Order. *Communications of the ACM*, 6(3):103, 1963.

[82] Syed Agha Hassnain Mohsan, Muhammad Asghar Khan, Fazal Noor, Insaf Ullah, and Mohammed H. Alsharif. Towards the Unmanned Aerial Vehicles (UAVs): A Comprehensive Review. *Drones*, 6(6), 2022. ISSN 2504-446X. doi: 10.3390/drones6060147.

[83] Marius Muja and David G Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. *International Conference on Computer Vision Theory and Applications*, 2(331-340):2, 2009.

[84] Toshinori Munakata and Roman Barták. Logic Programming for Combinatorial Problems. *Artificial Intelligence Review*, 33:135–150, 2010.

[85] Gonzalo Navarro and Nora Reyes. Dynamic Spatial Approximation Trees for Massive Data. In *2009 Second International Workshop on Similarity Search and Applications*, pages 81–88. IEEE, 2009.

[86] Arash Nejat, Frederic Ouattara, Mohammad Mohammadinodoushan, Bertrand Cambou, Ken Mackay, and Lionel Torres. Practical Experiments to Evaluate Quality Metrics of MRAM-Based Physical Unclonable Functions. *IEEE Access*, 8:176042–176049, 2020.

[87] Sameer A Nene and Shree K Nayar. A Simple Algorithm for Nearest Neighbor Search in High Dimensions. *IEEE Transactions on pattern analysis and machine intelligence*, 19(9):989–1003, 1997.

[88] Ying Niu and Xuncai Zhang. A Novel Plaintext-Related Image Encryption Scheme Based on Chaotic System and Pixel Permutation. *IEEE Access*, 8:22082–22093, 2020.

[89] NVIDIA. P100 The Most Advanced Data Center Accelerator Ever Built. Featuring Pascal GP100, the World's Fastest GPU, 2017. URL `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf`.

[90] NVIDIA. Pascal Tuning Guide, March 2018. URL `http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html`.

[91] William H Payne and Frederick M Ives. Combination generators. *ACM Transactions on Mathematical Software (TOMS)*, 5(2):163–172, 1979.

[92] Martin Perdacher, Claudia Plant, and Christian Böhm. Cache-Oblivious High-Performance Similarity Join. In *Proceedings of the International Conference on Management of Data*, pages 87–104, 2019.

[93] Christopher Philabaum, Christopher Coffey, Bertrand Cambou, and Michael Gowanlock. A Response-Based Cryptography Engine in Distributed-Memory. In Kohei Arai, editor, *Intelligent Computing*, pages 904–922, Cham, 2021. Springer Intl. Publishing. ISBN 978-3-030-80129-8.

[94] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627. IEEE, 2015.

[95] Penny Pritzker and Patrick D Gallagher. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Information Tech Laboratory National Institute of Standards and Technology*, pages 1–35, 2014.

[96] DA Rachkovskij. Fast Similarity Search for Graphs by Edit Distance. *Cybernetics and Systems Analysis*, 55(6):1039–1051, 2019.

[97] UCI Machine Learning Repository. Wave Energy Converters Data Set. `https://archive.ics.uci.edu/ml/datasets/Wave+Energy+Converters`. Accessed June 22, 2024.

[98] Chuitian Rong, Xiaohai Cheng, Ziliang Chen, and Na Huo. Similarity Joins for High-

Dimensional Data using Spark. *Concurrency and Computation: Practice and Experience*, 31(20):e5339, 2019.

[99] Frank Ruskey and Aaron Williams. The Coolest Way to Generate Combinations. *Discrete Mathematics*, 309(17):5305–5320, 2009.

[100] Seref Sagiroglu and Duygu Sinanc. Big data: A Review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, 2013.

[101] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.

[102] Carla Savage. A Survey of Combinatorial Gray Codes. *Society for industrial and Applied Mathematics (SIAM) Review*, 39(4):605–629, 1997.

[103] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.

[104] Claude Elwood Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[105] Shiyu Shen, Hao Yang, Wangchen Dai, Hong Zhang, Zhe Liu, and Yunlei Zhao. High-Throughput GPU Implementation of Dilithium Post-Quantum Digital Signature. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[106] Toshihiro Shimizu, Takuro Fukunaga, and Hiroshi Nagamochi. Unranking of Small Combinations from Large Sets. *Journal of Discrete Algorithms*, 29:8–20, 2014.

[107] Sachin Taneja, Anastacia B Alvarez, and Massimo Alioto. Fully Synthesizable PUF

Featuring Hysteresis and Temperature Compensation for 3.2% Native BER and 1.02 fJ/b in 40 nm. *IEEE Journal of Solid-State Circuits*, 53(10):2828–2839, 2018.

[108] Martha Torres, Alfredo Goldman, and Junior Barrera. A Parallel Algorithm for Enumerating Combinations. In *International Conference on Parallel Processing*, pages 581–588. IEEE, 2003.

[109] Jeffrey K Uhlmann. Metric Trees. *Applied Mathematics Letters*, 4(5):61–62, 1991.

[110] Scott A Vanstone and Paul C Van Oorschot. *An introduction to error correcting codes with applications*, volume 71. Springer Science & Business Media, 2013.

[111] Vasily Volkov. Better Performance at Lower Occupancy. `https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf`, 2010. Accessed June 22, 2024.

[112] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. A Novel High-Performance Implementation of CRYSTALS-Kyber with AI Accelerator. In *European Symposium on Research in Computer Security*, pages 514–534. Springer, 2022.

[113] Jinhua Wang, Qiuhong Li, Zhongsheng Li, Peng Wang, Yang Wang, Wei Wang, Ningting Pan, and Mingmin Chi. Similarity Join on Time Series by Utilizing a Dynamic Segmentation Index. *Knowledge and Information Systems*, 61(3):1517–1546, 2019.

[114] Terry Min Yih Wang and Carla D Savage. A Gray Code for Necklaces of Fixed Density. *SIAM Journal on Discrete Mathematics*, 9(4):654–673, 1996.

[115] David A White and Ramesh Jain. Similarity Indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523. IEEE, 1996.

[116] Svante Wold, Kim Esbensen, and Paul Geladi. Principal Component Analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[117] Jordan Wright, Zane Fink, Michael Gowanlock, Christopher Philabaum, Brian Donnelly, and Bertrand Cambou. A Symmetric Cipher Response-Based Cryptography Engine Accelerated Using GPGPU. In *2021 IEEE Conf. on Communications and Network Security*, pages 146–154, 2021.

[118] Jordan Wright, Michael Gowanlock, Chistopher Philabaum, and Bertrand Cambou. A CRYSTALS-Dilithium Response-Based Cryptography Engine Using GPGPU. In Kohei Arai, editor, *Proc. of the Future Technologies Conf. 2021, Volume 3*, pages 32–45, Cham, 2022. Springer Intl. Publishing.

[119] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering Billions of Data Points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6, 2009.

[120] Simin You, Jianting Zhang, and Le Gruenwald. Parallel Spatial Query Processing on GPUs using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–31, 2013.

[121] Amjad Hussain Zahid, Hamza Rashid, Mian Muhammad Umar Shaban, Soban Ahmad, Ehtezaz Ahmed, Muhammad Tallal Amjad, Muhammad Azfar Tariq Baig, Muhammad Junaid Arshad, Muhammad Nadeem Tariq, Muhammad Waseem Tariq, et al. Dynamic S-box Design using a Novel Square Polynomial Transformation and Permutation. *IEEE Access*, 9:82390–82401, 2021.

[122] Jingbo Zhou, Qi Guo, H. V. Jagadish, Lubos Krcal, Siyuan Liu, Wenhao Luan, Anthony K. H. Tung, Yueji Yang, and Yuxin Zheng. A Generic Inverted Index Framework for Similarity Search on the GPU. In *IEEE 34th International Conference on Data Engineering*, pages 893–904, 2018. doi: 10.1109/ICDE.2018.00085.

[123] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-Time kd-Tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 27(5):1–11, 2008.

[124] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 5(5):363–387, 2012.

Appendix A

Publications

- **2020**
  - A Coordinate-Oblivious Index for High-Dimensional Distance Similarity Searches on the GPU. **Donnelly, B.**, & Gowanlock, M. Proceedings of the 34th ACM International Conference on Supercomputing (ICS 2020), Barcelona, Spain, Article No. 8, pp 1–12, 2020.
- **2021**
  - Fast Period Searches Using the Lomb-Scargle Algorithm on Graphics Processing Units for Large Datasets and Real-Time Applications. Gowanlock, M., Kramer, D., Trilling, D. E., Butler, N. R., & **Donnelly, B**. Astronomy & Computing, 36, 100472, Elsevier.
  - A Symmetric Cipher Response-Based Cryptography Engine Accelerated Using GPGPU. Wright, J., Fink, Z., Gowanlock, M., Philabaum, C., **Donnelly, B.**, & Cambou, B. Proceedings of the IEEE Conference on Communications and Network Security 2021 (CNS 2021), pp. 146-154,
- **2023**
  - The Solar System Notification Alert Processing System (SNAPS): Design, Architecture, and First Data Release (SNAPShot1). Trilling, D. E., Gowanlock,

M., Kramer, D., McNeill, A., **Donnelly, B.**, Butler, N., & Kececioglu, J. The Astronomical Journal, 165, 111.

- Optimization and Comparison of Coordinate- and Metric-Based Indexes on GPUs for Distance Similarity Searches. Gowanlock, M., Gallet, B., & **Donnelly, B.** Proceedings of the International Conference on Computational Science 2023 (ICCS 2023), pp. 357-364. Cham: Springer Nature Switzerland.
- Evaluating Accelerators for a High-Throughput Hash-Based Security Protocol. Lee, K., **Donnelly, B.** Sery, T., Ilan, D., Cambou, B., & Gowanlock, M. 3rd International Workshop on Deployment and Use of Accelerators (DUAC) Proceedings of the 52nd International Conference on Parallel Processing Workshops (ICPPW) pp. 40–49.

- **2024**
  - Authentication in High Noise Environments using PUF-Based Parallel Probabilistic Searches. **Donnelly, B.** & Gowanlock, M. To appear in the Proceedings of the 28th IEEE High Performance Extreme Computing Conference (HPEC).
  - Multi-Space Tree with Incremental Construction for GPU-Accelerated Range Queries. **Donnelly, B.** & Gowanlock, M. 2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC 2024), pp. 132-142.

- **2025**
  - Performance Characterization of Parallel Combination Generators on CPU and GPU Systems. **Donnelly, B.** & Gowanlock, M. To appear in the Proceedings of the 15th International Workshop on Accelerators and Hybrid Emerging Systems (AsHES 2025).