Titel der Arbeit:

Scalable and Efficient Analysis of Large High-Dimensional Data Sets in the Context of Recurrence Analysis

D I S S E R T A T I O N
zur Erlangung des akademischen Grades

Doktor-Ingenieur
(Dr.-Ing.)

im Promotionsfach:
Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
M.Sc. Tobias Rawald

Tag der mündlichen Prüfung: 1. Dezember 2017

Abstract

Recurrence analysis is a method from nonlinear time series analysis to investigate the recurrent behaviour of a system, e.g., the Earth's climate system. Among others, it comprises a technique to quantitatively assess the contents of binary similarity matrices. This recurrence quantification analysis (RQA) relies on the identification of line structures within such recurrence matrices and provides a set of scalar measures. Existing computing approaches to RQA are either not capable of processing recurrence matrices exceeding a certain size or suffer from long runtimes considering time series that contain hundreds of thousands of data points. This thesis introduces scalable recurrence analysis (*SRA*), which is an alternative computing approach that subdivides a recurrence matrix into multiple sub matrices. Each sub matrix is processed individually in a massively parallel manner by a single compute device. This is implemented exemplarily using the OpenCL framework. *SRA* further enables the parallel processing of multiple sub matrices using a set of compute devices. It is shown that this approach delivers drastic performance improvements in comparison to state-of-the-art recurrence analysis software by exploiting the computing capabilities of many-core hardware architectures, in particular graphics cards. This reduces the runtime for analysing time series exceeding one million data points from hours or days to minutes. The usage of OpenCL allows to execute identical *SRA* implementations on a variety of hardware platforms having different architectural properties. On major challenge is that an implementation may expose varying performance characteristics across different compute devices. An extensive evaluation analyses the impact of applying concepts from database technology, such memory storage layouts, to the recurrence analysis processing pipeline. It is investigated how different realisations of these concepts, e.g., row-store vs. column-store layout, affect the performance of the computations on different types of compute devices. This does not only include the runtime behaviour but also additional performance counters, such as the amount of data fetched from compute device memory. Finally, an approach based on automatic performance tuning is introduced that autonomously selects well-performing RQA implementations for a given analytical scenario on a specific computing hardware. The corresponding evaluation compares the performance of a set of greedy selection strategies while analysing a real-world time series from climate impact research. Among others, it is demonstrated that the customised auto-tuning approach allows to drastically increase the efficiency of the processing by adapting the implementation selection.

Kurzfassung

Die Rekurrenzanalyse ist eine Methode aus der nicht-linearen Zeitreihenanalyse, die es erlaubt das wiederkehrende Verhalten von Systemen zu analysieren, bspw. des Klimasystems der Erde. Sie umfasst einen Ansatz zur quantitativen Analyse des Inhalts von binären Ähnlichkeitsmatrizen. Im Mittelpunkt der sogenannten Recurrence Quantification Analysis (RQA) steht dabei die Identifikation von Linienstrukturen in solchen Rekurrenzmatrizen. Hierzu stellt die RQA eine Reihe von skalaren Maßen bereit. Bestehende Berechnungsansätze zur Durchführung der RQA können entweder nur Zeitreihen bis zu einer bestimmten Länge verarbeiten oder benötigen viel Zeit zur Analyse von sehr langen Zeitreihen. Diese Dissertation stellt die sogenannte skalierbare Rekurrenzanalyse (*SRA*) vor. Sie ist ein neuartiger Berechnungsansatz, der eine gegebene Rekurrenzmatrix in mehrere Submatrizen unterteilt. Jede Submatrix wird von einem Berechnungsgerät in massiv-paralleler Art und Weise analysiert. Dieser Ansatz wird unter Verwendung der OpenCL-Schnittstelle implementiert. Darüber hinaus erlaubt es die *SRA*, mehrere Berechnungsgeräte gleichzeitig für die Analyse einer Rekurrenzmatrix zu verwenden. Anhand mehrerer Experimente wird demonstriert, dass dieser Ansatz massive Leistungssteigerungen im Vergleich zu existierenden Berechnungsansätzen ermöglicht, insbesondere durch den Einsatz der Vielkernprozessen von Grafikkarten. Dies reduziert die Laufzeit zur Analyse von Zeitreihen bestehend aus mehr als einer Million Datenpunkten von Stunden oder Tagen zu wenigen Minuten. Die Verwendung von OpenCL ermöglicht es, identische *SRA*-Implementierungen auf einer Reihe unterschiedlicher Hardware-Plattformen mit verschiedenen Architekturen auszuführen. Die Herausforderung ist hierbei das potentiell variierende Laufzeitverhalten einer konkreten Implementierung auf unterschiedlichen Berechnungsgeräten. Es wird eine ausführliche Evaluation durchgeführt, die den Einfluss der Anwendung mehrerer Datenbankkonzepte, wie z.B. die Repräsentation der Eingangsdaten, auf die RQA-Verarbeitungskette analysiert. Es wird untersucht, inwiefern unterschiedliche Ausprägungen dieser Konzepte, bspw. die zeilenorientierte gegenüber der spaltenorientierten Speicherung der Eingangsdaten, Einfluss auf die Effizienz der Analyse auf verschiedenen Berechnungsgeräten haben. Dies umfasst nicht nur die Betrachtung der Laufzeiten, sondern berücksichtigt ebenfalls weitere Leistungsindikatoren, wie z.B. die Menge der aus dem Speicher geladenen Daten. Abschließend wird ein automatischer Optimierungsansatz vorgestellt, der performante RQA-Implementierungen für ein gegebenes Analyseszenario in Kombination mit einer Hardware-Plattform dynamisch bestimmt. Die dazugehörige Evaluation vergleicht die Leistung einer Menge von empirischen Auswahlstrategien am Beispiel einer Klimazeitreihe. Neben anderen Aspekten werden drastische Effizienzgewinne durch den Einsatz des Optimierungsansatzes aufgezeigt.

Contents

1 Introduction

"Starting today, the performance lunch isn't free any more. ... [If] you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written concurrent ... application. And that's easier said than done, because not all problems are inherently parallelizable and because concurrent programming is hard."
– Excerpt from [Sutter, 2005]

The project underlying this thesis started with a simple question: "Can you make this faster?". It was posed by Norbert Marwan, who is a senior researcher at the Potsdam Institute for Climate Impact Research. In early 2013, he was using *recurrence analysis* [Marwan et al., 2007], a method from nonlinear time series analysis, to investigate the *Potsdam temperature profile* [Potsdam Institute for Climate Impact Research, 2016], a time series consisting of more than one million measurements. Existing software tools, only less than a handful of them even capable of processing time series of such length, consumed several hours to conduct the analysis for a given input parameter configuration. Varying the configuration multiplied the time for retrieving the analytical results. This induced the need for exploring novel computing approaches to accelerate the computations related to recurrence analysis, which is the topic of this thesis.

Performing recurrence analysis requires to compare the states of a system, e.g., the Earth's climate system, changing over time. The states are represented by multi-dimensional vectors consisting of floating point values. They refer to one or more observational variables, such as the temperature measured at a local weather station. The pairwise similarity of those vectors is computed using a specific measure and captured within a quadratic matrix. The matrix elements are transformed into binary values, using a similarity threshold. Connected matrix elements referring to the same value form small-scale structures, in particular diagonal and vertical lines, that have specific semantics in the context of recurrence analysis. Those structures may either be displayed by converting the matrix into a monochrome image (*recurrence plot*) or quantified using a set of scalar measures (*recurrence quantification analysis*) [Marwan et al., 2007]. An example recurrence plot based on real-world data from the climate domain is depicted in Fig. 1.1.

Investigating the dynamic behaviour of systems using recurrence analysis experienced an increasing interest in the recent past, expressed by the growing number of publications released each year [Marwan, 2017]. The method has proven its applicability in a variety of domains, including:

Geosciences: The Earth's climate system exposes recurrent behaviour, such as seasonal changes. Investigating the systems behaviour in the past allows to draw implications regarding future developments [Donges et al., 2011, Zhao and Li, 2011]. Recurrence analysis is

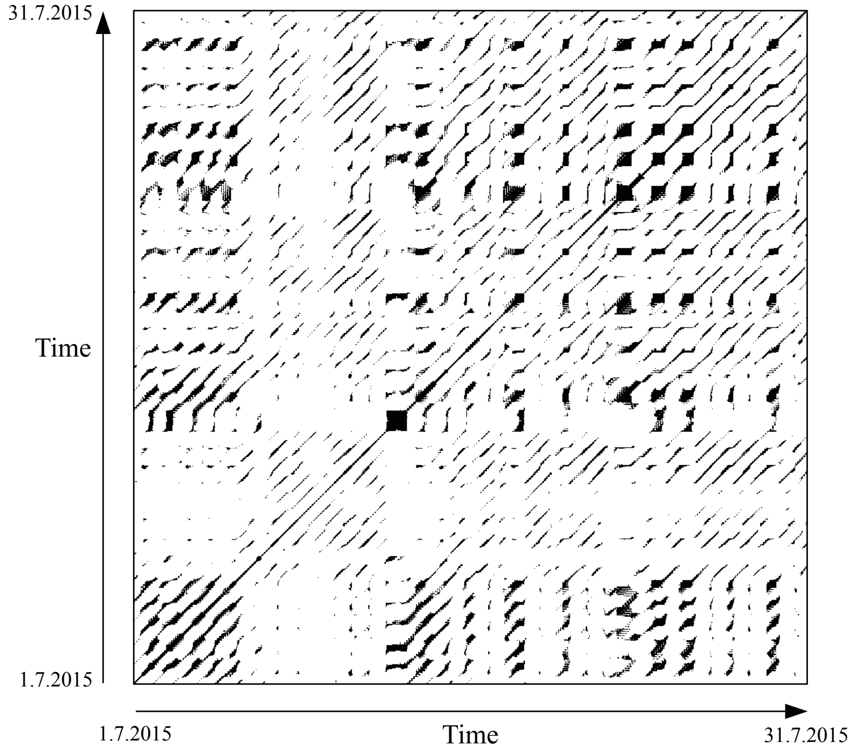


Figure 1.1: Real-world recurrence plot. The recurrence plot displayed refers to the dry-bulb temperature in degree Celsius measured at the Asheville Regional Airport in North Carolina during July 2015 with an hourly resolution. The measurements belong to the *Quality Controlled Local Climatological Data* (QCLCD) provided by the *National Oceanic and Atmospheric Administration* (NOAA) of the United States of America [National Oceanic and Atmospheric Administration, 2017]. Multi-dimensional vectors are extracted from the time series by applying the time delay method, using an embedding dimension of 5 and a time delay of 3. The pairwise similarity of the vectors is determined using the Maximum norm in combination with a threshold of 3.0. The semantics of the individual parameters are explained in detail in Sect. 2.1. The plot comprises small-scale structures formed by black and white dots. They characterise the dynamic behaviour of the climate system at the corresponding geographic location during the observation period.

further used to describe earthquake dynamics [Chelidze and Matcharashvili, 2015] and to get a better understanding of solar variability [Ponyavin and Zolotova, 2005].

Medicine: Recurrence analysis is applied to investigate brain activity and to enable the early detection of epileptic states [Carrubba et al., 2010, Chua et al., 2008]. Moreover, it is employed to observe functions and dysfunctions of the cardiorespiratory system [Zbilut et al., 1990, Van Leeuwen et al., 2009].

Mechanical engineering: The recurrent behaviour of gas turbines is examined to assess their functionality [Bassily and Wagner, 2008]. Furthermore, cutting processes are monitored using recurrence analysis [Litak et al., 2009]. The method is also employed to detect cracks within aluminum plates [Iwaniec et al., 2012].

Recurrence quantification analysis (RQA) addresses in particular the investigation of large matrices, constructed from more than one million vectors. Such matrices can hardly be displayed without loss of information. RQA comprises a set of quantitative measures based on line length histograms capturing data on small-scale structures. The values of those measures characterise the dynamic behaviour of a system within a certain period of time. In early 2013, state-of-the-art software conducting RQA suffered from several limitations, hampering or preventing the analysis of sets of multi-dimensional vectors comprising hundreds of thousands of elements. This is due to the quadratic relationship between the number of input vectors and the extent of the similarity matrix.

1.1 Contributions

This thesis presents three major contributions. First, it introduces *scalable recurrence analysis* (*SRA*), a computing approach that allows to process matrices exceeding $10^6 \times 10^6$ elements. It is heavily based on concepts from parallel processing and focusses especially on the computations of the quantitative analysis. *SRA* subdivides the binary similarity matrix into multiple sub matrices and distributes their processing across multiple compute devices within a computing system. Each sub matrix is processed individually by a single device. To enable the detection of line structures across multiple sub matrices, *carryover buffers* are introduced. These global data structures store the length of lines that reach the outer borders of sub matrices. The corresponding data is shared among all sub matrices, while investigating diagonals and columns of the global similarity matrix. The sub matrices are processed in a specific order, to compute valid global RQA results. This cutomised *processing order* is designed such that multiple sub matrices can be analysed simultaneously. That allows to leverage the capabilities of multiple compute devices at the same time.

Second, this thesis proposes to separate the RQA processing into a set of analytical operators. For each operator, a single type of atomic task is defined such that each task instance is fully independent of any other instance of the same operator. This allows to conduct the computations in a *massively parallel manner*. This approach is implemented using the *OpenCL* framework for heterogeneous computing, which allows to offload computations to parallel compute devices from various hardware vendors, in particular graphics processors. Furthermore, several concepts from database engineering are assessed regarding their applicability to RQA processing. This includes for example different representations of the multi-dimensional input vectors as well as whether the data captured in the recurrence matrix is materialised or not. An extensive evaluation investigates the impact of the concepts to the performance of conducting RQA. A key insight of the evaluation is that the performance of a specific implementation highly depends on the combination of analytical scenario and hardware platform.

Third, this thesis introduces an *automatic performance tuning* approach that allows to dynamically select a well-performing implementation. The development of this auto-tuning approach was driven by OpenCL providing functional portability, which ensures the compilation and execution of identical source code on multiple platforms. In this regard, a major challenge is that OpenCL does not guarantee that the compiled code delivers a good performance across various platforms [Trevett, 2017, p. 14]. Hence, a set of implementations with varying properties is provided. The performance of each implementation is assessed on sub matrix level, executing all operators required to compute the RQA results. The selection of implementations is conducted based on a set of greedy selection strategies that aim at minimising to total runtime.

These contributions allow to drastically reduce the runtime for performing recurrence analysis computations. As an example, the time for analysing the Potsdam temperature profile with RQA could be reduced from more than a day, using state-of-the-art software in combination with server CPU hardware, to roughly one hundred seconds. This is achieved by applying *SRA* while leveraging the computing capabilities of four GPU processors at the same time and dynamically selecting the best-performing implementation.

1.2 Structure

This thesis is structured as follows. Chapter 2 gives an overview over the basic principles behind recurrence analysis. The first section, referring to the theoretical foundations, is heavily based on the work by Norbert Marwan and his colleagues from the recurrence analysis community [Marwan et al., 2007, Marwan and Webber, 2015]. The subsequent section highlights the computational aspects of the method. This includes the description of basic algorithms that are required to create recurrence plots and to perform RQA. Subsequently, the properties of state-of-the-art software conducting recurrence analysis computing are explored. The focus of those considerations is specifically on their computational limitations. The third section gives an introduction to the underlying concepts of the OpenCL framework that is used to implement *SRA*.

Chapter 3 presents alternative computing approaches that aim at overcoming the limitations of existing recurrence analysis software. This includes parallel computing using multi-core devices, such as CPUs, and using many-core devices, such as GPUs. The basic approach regarding the construction of the binary matrix refers to the exhaustive computation of all pairwise vector similarities. Alternatively, index data structures, such as multi-dimensional search trees, can be applied. The remaining sections further elaborate on approaches based on compacting the input data and approximating the RQA measures.

Chapter 4 gives detailed information on scalable recurrence analysis, which is based on the concept *divide and recombine* [Guha et al., 2012]. The chapter describes strategies on how to subdivide the full recurrence matrix into sub matrices and to recombine the individual analytical results. Moreover, the functionality of the carryover buffers as well as the resulting sub matrix processing order is described. It is highlighted, how these concepts enable the concurrent processing of multiple sub matrices. The chapter further includes descriptions in which way the symmetry of recurrence matrices can be exploited to balance the workload for detecting diagonal lines among sub matrices.

Chapter 5 addresses the application of advanced concepts from database technology to the processing within the analytical operators. The design dimensions considered in the first section include the representation of the multi-dimensional vectors serving as input data, the representation of the quadratic recurrence matrix, the representation of the binary similarity values, the recycling of intermediate values and whether the data captured in a recurrence matrix is materialised or not. Two realisations are considered for each of those design dimensions. The second section elaborates on using index data structures, in particular grid directories and multi-dimensional search trees, to determine the similarity of multi-dimensional vectors.

Chapter 6 presents a comprehensive evaluation regarding the design dimensions mentioned in Chap. 5. Based on a set of initial experiments, the impact of each realisation on the performance is investigated. The analysis considers the execution of the individual analytical operators as well as the whole RQA processing pipeline. The experiments are conducted on different hardware platforms, leveraging the functional portability provided by OpenCL. The goal is to draw conclusions regarding the suitability of certain realisations on specific hardware platforms given a concrete analytical scenario. The second part of the evaluation concentrates on the impact of the index data structures.

Chapter 7 addresses the problem of selecting well-performing *SRA* implementations. It introduces an automatic performance tuning approach that considers the individual performance characteristics of a set of implementations that are extended using tuning parameters. The selection is conducted based on greedy strategies, considering the runtime for processing sub matrices. The chapter further includes three experiments, investigating different effects of auto-tuning. The first experiment investigates the impact of applying different greedy selection strategies on the overall performance of conducting RQA. The second experiment highlights the impact on the efficiency of the computations in comparison to state-of-the-art RQA software. The third experiment demonstrates the scalability of the *SRA* approach by determining the speed-up gained from increasing the number of compute devices used during the processing.

Chapter 8 concludes the findings of the thesis. It summarises the properties of *SRA* that enable drastic performance improvements regarding the processing of very long time series. Furthermore, the limitations of this novel computing approach are examined to deduce research questions potentially addressed in the future.

Appendix A contains source code that refers to an approximation approach described in Chap. 3. Appendix B contains a set of mathematical equations that refer to a specific database concept described in Chap. 5. Appendix C presents the hardware and software configurations of the computing systems applied during the experiments. Appendix D describes the experimental setups referring to the evaluations from Chap. 6 and 7. Appendix E contains the corresponding experimental results, including visual and tabular representations. Note that the appendices are only available in the digital version of the manuscript.

1.3 Prior Work

Chapter 4 is based on [Rawald et al., 2014a], which was written by Tobias Rawald, in the following referred to as the author, Mike Sips, Norbert Marwan and Doris Dransch. This conference paper focusses specifically on the computations related to RQA. It presents the principles behind *SRA*, which was developed and implemented by the author under the supervision of Mike Sips and Doris Dransch. The potential of this novel computing approach is demonstrated by Norbert Marwan, who applied it to the Potsdam temperature profile from 1893 until 2011. He further provided background on recurrence analysis.

The first sections of Chap. 5 and Chap. 6 are based on [Rawald et al., 2015], which was written by the author, Mike Sips, Norbert Marwan and Ulf Leser. This workshop paper considers a subset of the database concepts presented in Chap. 5. The corresponding implementations were designed and implemented by the author. He further conducted an experiment comparing their runtime performance for varying analytical scenarios. This work has been conducted under the supervision of Mike Sips and Ulf Leser. Again, Norbert Marwan provided the link to the recurrence analysis domain.

Additional publications have been released to create awareness regarding *SRA* within the recurrence analysis community [Rawald et al., 2014b,c].

1.4 Software

SRA has further been implemented using the Python programming language, leveraging the capabilities provided by OpenCL framework. This also includes the automatic performance tuning approach. A corresponding Python package named *PyRQA*, which was designed and implemented by the author, is released under version 2.0 of the Apache License [Rawald, 2015]. The contents of the package and application examples are described in [Rawald et al., 2016]. This journal paper was written by the author under the supervision of Mike Sips. Norbert Marwan contributed the analysis of a real-world analysis scenario. A focus of this publication is on describing the application programming interface and enabling the reproducibility of the experimental results.

2 Background

This chapter gives background information on multiple topics and is structured into three sections; each of them is largely based on related work. The first section addresses the theoretical foundations of recurrence analysis. It focuses explicitly on those parts that are relevant in the following chapters. The second section considers the computational aspects of recurrence analysis, including basic algorithms and state-of-the-art software. The focus is especially on describing the limitations of existing computing approaches to recurrence analysis. The approach presented in this thesis considers offloading massively parallel computations on accelerators, such as GPUs. This is achieved by using the OpenCL framework for heterogeneous computing. Its components and functionalities are summarised in the third section of this chapter.

2.1 Theoretical Foundations of Recurrence Analysis

Recurrence analysis is a method from nonlinear time series analysis, which allows to investigate the recurrent behaviour of systems. The corresponding research field comprises a variety of different topics. A continuously updated list of publications is maintained at [Marwan, 2017]. As of June 2017, it contains over 1,900 publications regarding theoretical foundations, applications, software and other publications. The following overview does not claim to be complete but rather highlights relevant aspects.

This thesis is based on a collaboration with Dr. Norbert Marwan of the Potsdam Institute for Climate Impact Research (PIK). Unless indicated otherwise, this condensed introduction refers to [Marwan et al., 2007] and [Marwan and Webber, 2015].

2.1.1 Recurrence and Time Delay Embedding

Recurrence is an important concept regarding the the analysis of dynamic system behaviour. The corresponding mathematical foundations were introduced by Henri Poincaré in 1890, resulting in the *Poincaré recurrence theorem* [Poincaré, 1890]. He stated that a "system recurs infinitely many times as close as one wishes to its initial state" under certain conditions. If the behaviour of the system is deterministic, this property allows to draw conclusions regarding its future development.

The state of a system can be described by a set of variables. As an example, the impact of the Earth's climate system at a specific location is among others expressed through the air pressure and temperature, relative humidity as well as wind speed. The state of a system can be captured by a vector residing in d-dimensional space, where each of the d vector components refers to a single observational variable. As time progresses, the values of the vector components change, resulting in different system states.

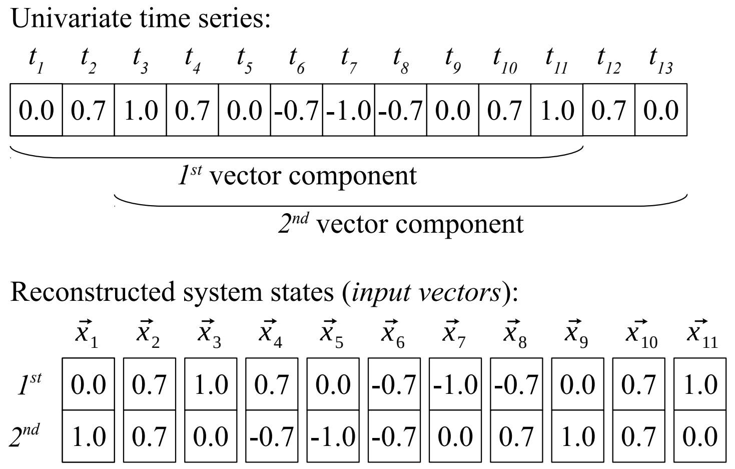

Univariate time series:
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
0.0 0.7 1.0 0.7 0.0 -0.7 -1.0 -0.7 0.0 0.7 1.0 0.7 0.0
1st vector component
2nd vector component
Reconstructed system states (input vectors):
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
1st 0.0 0.7 1.0 0.7 0.0 -0.7 -1.0 -0.7 0.0 0.7 1.0
2nd 1.0 0.7 0.0 -0.7 -1.0 -0.7 0.0 0.7 1.0 0.7 0.0

Figure 2.1: Time delay method. Given an discretisation of the *sine* wave within the interval 0 and 3π. The corresponding univariate time series consists of thirteen data points (t_1 to t_{13}). Eleven input vectors ($\vec{x}_1$ to $\vec{x}_{11}$) are reconstructed by employing an embedding dimension $m = 2$ and a time delay $t = 2$.

Usually, it is not possible to obtain data for all relevant variables. Often, only a single variable, such as the air temperature, may be observed. However, the data collected for this single variable contains information about the dynamics of the whole system. The *Takens' theorem* [Takens, 1980] and corresponding extensions [Sauer et al., 1991] ensure that it is possible to reconstruct the topological structure of the trajectory formed by the state vectors, given only data for a single variable. For this purpose, multi-dimensional vectors are extracted from an univariate time series that captures the observations of the single variable at discrete points in time. In the remaining parts of this thesis it is assumed that only a single univariate time series is provided as input for the recurrence analysis processing, unless stated otherwise.

The *time delay* method is commonly used to reconstruct system states from an univariate time series [Kantz and Schreiber, 2003, pp. 30–36]. The dimensionality of the reconstructed vectors corresponds to the number of relevant variables and varies between different systems. In Fig. 2.1, the application of the time delay method to a time series capturing the sine wave is depicted. Vectors comprising two components are reconstructed for the purpose of demonstration.

The state reconstruction relies on the parameters:

- Embedding dimension (m), and
- Time delay (t).

The *embedding dimension* describes the dimensionality of the reconstructed vectors. The *time delay* parameter specifies the temporal offset of the vector components within the time series. Note that it requires substantial knowledge of a domain expert to assess the suitability of corresponding parameter values, although there are methods to detect the assignments automatically. Appropriate assignments are usually determined manually using iterative refinement, evaluating embedding results for a number of parameter value combinations.

Recommendations for setting the parameter values regarding different systems are available. In [Webber Jr. and Zbilut, 2005, p. 36] it is proposed to set the embedding dimension between 10 and 20 regarding the analysis of biological systems. In principle, values between 1 and 20 are considered to be reasonable regarding the embedding dimension as well as the time delay[1]. There are also approaches neglecting the embedding within multi-dimensional space by operating on the scalar values of a given time series [Thiel et al., 2004, Iwanski and Bradley, 1998].

Given a time series consisting of l elements, an embedding dimension m and a time delay t, the number of multi-dimensional vectors N is calculated as defined in Equation 2.1.

$$N = l - ((m-1) * t) \tag{2.1}$$

Assuming that the time series consists of hundreds of thousands of data points and that m as well as t are within the boundaries as described above, the number of vectors corresponds roughly to the number of measurements [Kantz and Schreiber, 2003, p. 35].

2.1.2 Recurrence Plot

A method to visualise the dynamics of a system is the *recurrence plot*. A synthetic example is presented in Fig. 2.2. The method was originally introduced in [Eckmann et al., 1987] as follows.

Definition 2.1 (Recurrence plot)**.** Let $\vec{x}_i$ be the i-th point on the orbit[2] describing a dynamical system in d-dimensional space, for $i = 1, ..., N$. The recurrence plot is an array of dots in a $N \times N$ square, where a dot is placed at (i, j)[3] whenever $\vec{x}_j$ is sufficiently close to $\vec{x}_i$.

A recurrence plot allows to visually explore the binary similarities of pairs of system states. In this regard, $\vec{x}_1$ to $\vec{x}_N$ refer to the multi-dimensional vectors extracted from the input time series. Each of those vectors references a specific point in time. The axes of the recurrence plot capture the temporal progress. The visual representation is based on a squared similarity matrix, the *recurrence matrix* as defined in Equation 5.3.

$$R_{i,j} = \begin{cases} 1 : \vec{x}_i \approx \vec{x}_j, \\ 0 : \vec{x}_i \not\approx \vec{x}_j, \end{cases} \quad i, j = 1, ..., N \tag{2.2}$$

$\vec{x}_i \approx \vec{x}_j$ indicates the similarity of a pair of vectors. A column of the matrix captures the similarities of a specific vector with respect to all system states extracted from the time series. As a result, the matrix captures a total of N^2 binary similarity values.

Pairwise Vector Similarities

A distance measure, typically an L_p-norm, is applied to determine the pairwise similarity of two vectors. Candidates regarding the creation of a recurrence matrix are [Webber Jr. and Zbilut, 2005, p. 38]:

[1]Private conversation with Norbert Marwan in July 2013.
[2]The trajectory that is formed by the input vectors in d-dimensional space.
[3]It holds that $1 \leq i \leq N$ and $1 \leq j \leq N$.

- the L_1-norm (Manhattan distance),
- the L_2-norm (Euclidean distance), and
- the L_∞-norm (Maximum distance).

A *neighbourhood condition* is applied to transform the pairwise similarities into binary values. In this regard, either the *fixed radius* or the *fixed amount of nearest neighbours* condition are used. The similarity is determined by a threshold ϵ in both cases. All vectors that lie within the ϵ-neighbourhood of a query vector $\vec{x}_q$ are considered similar to $\vec{x}_q$. Choosing $\epsilon > 0$ ensures that dissimilarity up to a certain error is permitted, as required by recurrence according to Poincaré [Poincaré, 1890].

The fixed radius condition, which is most commonly used, specifies the same threshold value for each input vector, leading to equally sized ϵ-neighbourhoods. The shape of a neighbourhood is determined by the distance metric applied. The amount of vectors that lie within the neighbourhoods varies. The original concept introduced in [Eckmann et al., 1987] proposed to apply an individual threshold for each query vector, so that each neighbourhood contains a minimum number of vectors. This resembles the fixed amount of nearest neighbours condition. Here, the k closest neighbours are determined for each query vector, creating variable-sized neighbourhoods.

Applying the fixed radius condition in combination with a distance metric leads to recurrence matrices that are symmetric along the middle diagonal. The symmetry results from the fixed-sized neighbourhoods as well as the transitivity property of the distance metric. The middle diagonal captures the self-similarities of the multi-dimensional vectors and is commonly referred to as *line of identity* (LOI). In contrast, it is not guaranteed that a recurrence matrix is symmetric, if the fixed amount of nearest neighbours condition is applied.

In addition to the two types of neighbourhood conditions introduced before, there exist other types, e.g. the selection of a radius corridor [Iwanski and Bradley, 1998], which will not be discussed here. Furthermore, the unthresholded similarity values can be organised using the layout of a recurrence plot, which is then similarly referred to as *unthresholded recurrence plot* [Iwanski and Bradley, 1998]. In the context of this thesis, it is assumed that recurrence plots are constructed based on thresholded recurrence matrices.

Similar to the selection of the values for embedding dimension and time delay, the choice of an appropriate ϵ is essential regarding the meaningfulness of a recurrence plot. There exist several strategies, e.g., setting the value to a small portion of the diameter of the space spanned by the multi-dimensional vectors [Mindlin and Gilmore, 1992]. This is commonly referred to as *maximum phase space diameter*. Another strategy is to choose a radius based on the density of the recurrence plot [Zbilut et al., 2002]. Again, the knowledge of the domain expert is required to conduct an appropriate parameter selection.

Characteristic Structures

A recurrence plot is usually encoded as a monochromatic image. Commonly, cells of the recurrence matrix referring to similar vector pairs are represented by black dots, whereas cells referring to dissimilar pairs of vectors are represented by white dots. The black dots are referred to as *recurrence points*. A recurrence point reflects that the system under investigation recurs

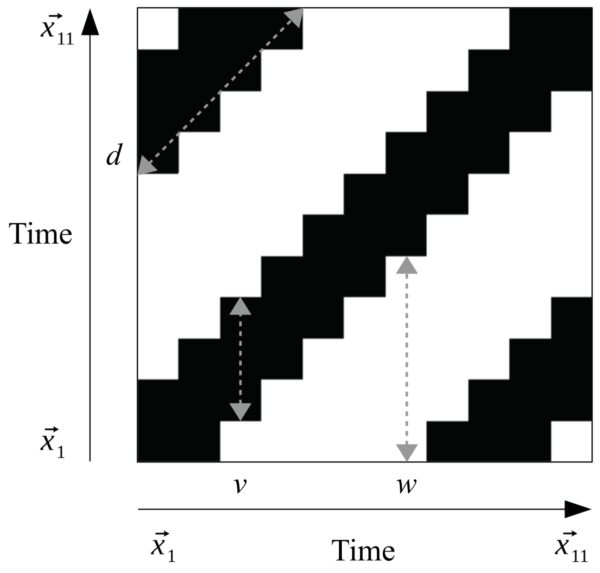


Figure 2.2: Recurrence plot. The plot depicted is based on the example from Fig. 2.1. Both axes refer to the vectors $\vec{x}_1$ to $\vec{x}_{11}$ in temporal order. Regarding the pairwise vector similarity, the Euclidean norm and a similarity threshold $\epsilon = 1.0$ are employed. A diagonal line d of length $l_d = 4$, a vertical line v of length $l_v = 3$ and a white vertical line w of length $l_w = 5$ are highlighted.

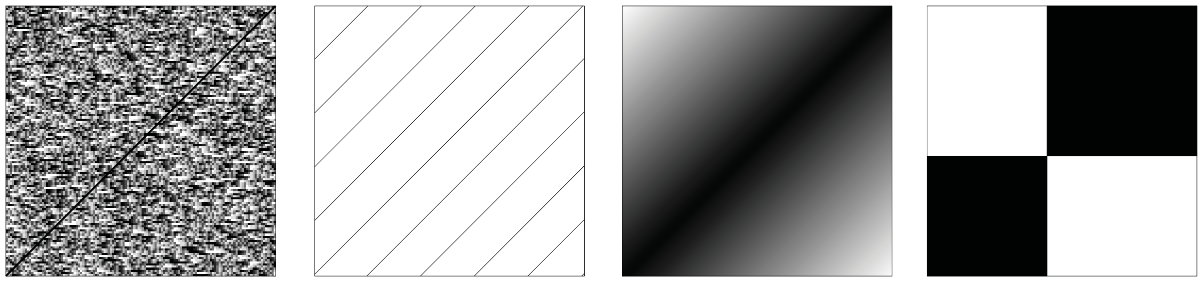
to a similar state. A recurrence plot that is constructed based on the two-dimensional vectors from Fig. 2.1 is depicted in Fig. 2.2.

Dots within a recurrence plot form small-scale and large-scale structures. These structures are the basis for its interpretation. There exist three types of small-scale structures:

- Diagonal lines formed by black dots (*diagonal lines*),
- Vertical lines formed by black dots (*vertical lines*), and
- Vertical lines formed by white dots (*white vertical lines*).

By convention, a line consists of at least two points of the same colour. It is stressed that each line in the recurrence plot has a numerical counterpart within the recurrence matrix. A line is either delimited by points of different colour or the borders of the recurrence matrix. Hence, only the lines with maximum extent are considered, excluding partial lines from the investigation.

Each line type mentioned above has different semantics. The meaning of a *diagonal line* depends on its alignment, which is either parallel or orthogonal to the LOI. The state of the system evolves similarly at different periods in time, if the diagonal line is parallel to the middle diagonal. The system states evolve similarly but in reverse temporal order, if the diagonal line is orthogonal to the LOI. In the following, only diagonal lines parallel to the LOI are considered.



(a) Homogenous (b) Periodic (c) Drift (d) Disrupted

Figure 2.3: Recurrence plot types. A pictogram is given for each topological type, conveying prototypical contents of corresponding plots.

A *vertical line* defines a period during which the state of a system does not change or the change progresses very slowly. It appears as if the system state is trapped for a certain period of time.

White vertical lines are the inverse of vertical lines. They serve as an estimator for *recurrence times*, which capture the time that elapses until a system recurs to a similar state. Similar to the vertical lines, each white vertical line has a mirrored counterpart of the same length but with horizontal alignment. This property is fulfilled, if the recurrence matrix is symmetric.

Sets of occurrences of the line structures described above form large-scale patterns within a recurrence plot, influencing its topological structure. There is a distinction between the following types of recurrence plots. Simplified representatives for each of the types are presented in Fig. 2.3.

Homogenous: The recurrence plot contains a large number of recurrence points that are homogeneously distributed.

Periodic: Long uninterrupted diagonal lines occur with the recurrence plot, indicating a periodic system. These lines are usually distributed regularly.

Drift: The recurrence point density decreases gradually from the LOI to the outer corners of the recurrence plot.

Disrupted: White bands or areas appear within the recurrence plot, signalling extreme events or drastic changes in the system dynamics.

2.1.3 Recurrence Quantification Analysis

Recurrence plots are attached with specific limitations, despite providing means to analyse the recurrence properties of system behaviour. This includes that plots with increasing size can hardly be depicted on graphical displays as a whole. Either, only a part of the original plot can be displayed at once or the whole plot has to be resized. The necessary resampling of the plot is very likely to distort patterns or create new artefacts, which may cause incorrect interpretations [Marwan, 2011].

The second major limitation is the subjective impression of the structures within a recurrence plot. Varying interpretations are likely to be retrieved for the same plot, when presenting it to two different researchers. To enable an objective assessment, Zbilut and Webber introduced the quantification of the structures within a recurrence plot by defining measures based on diagonal lines [Zbilut and Webber Jr., 1992, Webber Jr. and Zbilut, 1994]. This quantitative analysis, which is referred to as *recurrence quantification analysis* (RQA), allows to capture the complexity of the plot.

Following the initial work of Zbilut and Webber, RQA has been extended by additional measures. Those quantitative measures were later associated with concrete semantics regarding the recurrence properties of systems. In this thesis, the focus lies on RQA measures in the following categories:

- Recurrence point density,
- Diagonal line measures,
- Vertical line measures, and
- White vertical line measures.

There also exist approaches that conduct the quantitative analysis based on complex networks, which are not discussed here [Marwan et al., 2009].

Below, RQA measures are defined based on [Marwan and Webber, 2015, pp. 13–18], [Webber Jr. and Zbilut, 2005, pp. 46–50] and [Marwan et al., 2007, pp. 263–274], given set of N input vectors and a corresponding recurrence matrix R. $r_{i,j}$ refers to a single cell of the matrix.

Recurrence Point Density

Recurrence point density, or *recurrence rate* (RR), is defined in Equation 2.3.

$$RR = \frac{1}{N^2} \sum_{i,j=1}^{N} r_{i,j} \tag{2.3}$$

It captures the probability that a system recurs to a similar state, assuming that the amount of multi-dimensional vectors approaches infinity (see Equation 2.4). Mathematically, P expresses the probability that a cell of the recurrence matrix is assigned with the value one.

$$P = \lim_{N \to \infty} RR(N) \tag{2.4}$$

The remaining measures rely on frequency distributions of line lengths H_D (diagonal lines), H_V (vertical lines) and H_W (white vertical lines), referred to as *histograms*. They capture the number of occurrences that can be found within the recurrence plot for each line length l. Similar to the recurrence rate, they express the probability that a line of a certain length can be found.

Diagonal Line Measures

A fundamental measure based on diagonal lines is *determinism* (DET) (see Equation 2.5) that refers to the portion of recurrence points that form diagonal lines. By convention, a line consists of at least two consecutive matrix elements of the same colour. Hence, only diagonal lines with a length $l \geq d_{min}$ where $d_{min} = 2$ are considered regarding the quantitative analysis. It might be desirable to select $d_{min} > 2$ for specific scenarios [Webber Jr. and Zbilut, 2005, p.41].

$$DET = \frac{\sum_{l=d_{min}}^{N} l H_D(l)}{\sum_{i,j=1}^{N} r_{i,j}} \tag{2.5}$$

The *average diagonal line length* (D_{mean}) refers to the mean length of all diagonal lines within the recurrence plot with $l \geq d_{min}$ (see Equation 2.6). It captures the mean time that a segment of the trajectory is within the ϵ-tube of another trajectory [Marwan and Webber, 2015, p. 15].

$$D_{mean} = \frac{\sum_{l=d_{min}}^{N} l\, H_D(l)}{\sum_{l=d_{min}}^{N} H_D(l)} \tag{2.6}$$

The *maximum diagonal line length* (D_{max}) describes the maximum time that two segments of the trajectory are close to each other (see Equation 2.7).

$$D_{max} = max\{l \mid H_D(l) > 0\} \tag{2.7}$$

The *entropy diagonal lines histogram* (D_{entr}) uses the Shannon entropy of the histogram of diagonal lines to determine the complexity of the diagonal structures within the recurrence plot (see Equation 2.8).

$$D_{entr} = - \sum_{l=d_{min}}^{N} p(l)\, ln\, p(l) \tag{2.8}$$

$p(l)$ captures the probability that a diagonal line of length l occurs (see Equation 2.9).

$$p(l) = \frac{H_D(l)}{\sum_{l=d_{min}}^{N} H_D(l)} \tag{2.9}$$

An additional parameter regarding the analysis of diagonal structures is the *Theiler corrector* c. It states that only diagonal lines that are located on diagonals that have a distance of at least c from the LOI are captured within the frequency distribution. As an example, $c = 1$ excludes the the middle diagonal of the recurrence matrix from the inspection regarding diagonal lines. Conceptually, the Theiler corrector reduces the impact of *tangential motion*. This phenomenon occurs when the ϵ parameter is set too large.

Vertical Line Measures

Extending the original RQA, there exist measures to quantify vertical line structures. The *laminarity* (LAM) captures the amount of recurrence points that form vertical lines. The corresponding formula is similar to Equation 2.5, despite that the vertical line length histogram

Table 2.1: Summary of basic RQA measures.

Measure Category	*RP Density*	*Diagonal Lines*	*Vertical Lines*	*White Vertical Lines*
Fraction	RR	DET	LAM	-
Average line length	-	D_{mean}	TT	W_{mean}
Maximum line length	-	D_{max}	V_{max}	-
Most frequent line length	-	-	-	W_{mode}
Entropy	-	D_{entr}	V_{entr}	W_{entr}

is applied. Likewise, a minimum length v_{min} is defined, excluding shorter lines. The laminarity expresses the portion of laminar states within the recurrence plot.

The *trapping time* (TT) is the vertical counterpart to D_{mean}. It refers to the mean vertical line length and is calculated similar to Equation 2.6, using the vertical line length histogram and v_{min}. The trapping time reveals how long a system remains within a similar state in average.

The *maximum vertical line length* (V_{max}) captures the longest period of time that a system remains within a similar state. It refers to the highest index of the vertical line length histogram where the corresponding value is not 0. The *entropy vertical lines histogram* (V_{entr}), which is calculated similarly to Equation 2.8, captures the complexity of the vertical line length histogram.

White Vertical Line Measures

Measures estimating recurrence times based on the histogram of white vertical lines have first been introduced in [Ngamga et al., 2007]. Among others, it defines the *mean recurrence time* T_{MRT} and the *number of recurrence* N_{MPRT}, indicating the *number of occurrences of the most frequent white vertical line length.*

More measures have been added based on this initial effort. This includes the relabelled mean recurrence time W_{mean} and the entropy of the white vertical line length histogram W_{entr}, which has been first introduced as *recurrence probability density entropy* in [Little et al., 2007]. Apdapting N_{MPRT}, W_{mode} (see Equation 2.10) captures the *most frequent white vertical line length.*

$$W_{mode} = \arg\max_{l} H_W(l) \tag{2.10}$$

2.2 Computational Aspects of Recurrence Analysis

This section gives an overview over the computational aspects of recurrence analysis. In the beginning, basic recurrence analysis algorithms are introduced, e.g., regarding the construction of the binary similarity matrix. The second part refers to existing software that employs the basic algorithms to retrieve analytical results. A particular focus is on the limitations of those tools, hampering or preventing the analysis of long time series.

2.2.1 Basic Recurrence Analysis Algorithms

A number of analytical operations has to be performed to conduct recurrence quantification analysis. This includes:

1. Creating a recurrence matrix,
2. Detecting diagonal lines within the recurrence matrix,
3. Detecting vertical lines within the recurrence matrix,
4. Detecting white vertical lines within the recurrence matrix, and
5. Computing the RQA measures based on the individual histograms.

In the following, basic algorithms for those operations are investigated, except for the computation of the specific RQA measures. The algorithms depicted are the basis of the RQA processing and implemented by state-of-the-art software. All of those algorithms assume that the processing is conducted in a sequential manner.

The focus of the following descriptions lies especially on examining the time complexity and the space complexity of each algorithm, based on the total number of multi-dimensional input vectors N. Moreover, the content of the line length histograms regarding specific recurrence point densities is analysed.

The first four analytical operations are mapped to corresponding functions:

1. $\textsc{CreateRecurrenceMatrix}(X, N, \epsilon)$
2. $\textsc{DetectDiagonalLines}(R, N, c)$
3. $\textsc{DetectVerticalLines}(R, N)$
4. $\textsc{DetectWhiteVerticalLines}(R, N)$

The first function returns the binary similarity matrix R, whereas the latter three return the corresponding line length histograms H_D, H_V and H_W. It is assumed that the recurrence matrix is created based on the set of input vectors X with a size of N and by applying a fixed-sized ϵ neighbourhood. The binary similarity matrix serves as input for the detection of line structures. The resulting histograms are used during the computation of the RQA measures. The detection of diagonal lines additionally relies on the Theiler corrector c, potentially excluding diagonals of the recurrence matrix from the inspection.

Algorithm 1 Create recurrence matrix.

```
function CreateRecurrenceMatrix($X, N, \epsilon$)
    $R \leftarrow$ Empty2DArray()
    for $i \leftarrow 1$ to $N$ do
        for $j \leftarrow 1$ to $N$ do
            if Distance($\vec{x}_i, \vec{x}_j$) $\leq \epsilon$ then
                $r_{i,j} \leftarrow 1$
            else
                $r_{i,j} \leftarrow 0$
            end if
        end for
    end for
    return $R$
end function
```

Create Recurrence Matrix

Algorithm 1 presents the construction of the recurrence matrix R, if a fixed radius neighbourhood condition is applied. The recurrence matrix is represented using a two-dimensional array, which results in a space complexity of $\mathcal{O}(N^2)$. Note that this uncompressed representation can be replaced by a compressed representation (see Sect. 5.1.2). The following line detection algorithms assume an uncompressed representation.

A nested loop traverses each matrix element, with both loop counters i and j running from one to N. A distance measure is applied to each pair of multi-dimensional vectors (x_i, x_j). The value 1 is assigned to the corresponding matrix element $r_{i,j}$, if the distance is smaller or equal to the similarity threshold ϵ. Otherwise, the value 0 is assigned. Algorithm 1 has a time complexity of $\mathcal{O}(N^2)$, when abstracting from the detailed computations related to the pairwise vector similarities.

Detect Diagonal Lines

Algorithm 2 depicts the detection of diagonal lines, assuming that the recurrence matrix is symmetric. The histogram of diagonal line lengths H_D is initialised as an one-dimensional array of size N containing zero values. A nested loop forms the body of the algorithm. The outer loop iterates over the last $N-1$ input vector indices, excluding the LOI that contains a diagonal line of length N. Each loop index maps directly to a diagonal within the lower half of the recurrence matrix. Given the value i, the corresponding diagonal comprises $N-(i-1)$ matrix elements.

The inner loop is only executed, if the diagonal index i is greater or equal to the Theiler corrector c. If this condition is fulfilled, the elements of the corresponding diagonal are examined sequentially. The line length l is incremented by one, if the current matrix element $r_{k,j}$ is a recurrence point. Otherwise, the line length histogram H_D is updated and l is reset to zero. The corresponding histogram element $H_D(l)$ is incremented by two, due to inspecting only the lower half of the symmetric matrix. The histogram is only updated, if the current diagonal line

Algorithm 2 Detect diagonal lines.

```
1: function DetectDiagonalLines($R, N, c$)
2:     $H_D \leftarrow$ Zeros1DArray($N$)
3:     for $i \leftarrow 2$ to $N$ do
4:         if $i \geq c$ then
5:             $l \leftarrow 0$
6:             $j \leftarrow 1$
7:             for $k \leftarrow 1$ to $(N - (i - 1))$ do
8:                 if $r_{k,j} = 1$ then
9:                     $l \leftarrow l + 1$
10:                else
11:                    if $l \geq 2$ then
12:                        $H_D(l) \leftarrow H_D(l) + 2$
13:                    end if
14:                    $l \leftarrow 0$
15:                end if
16:                $j \leftarrow j + 1$
17:            end for
18:            if $l \geq 2$ then
19:                $H_D(l) \leftarrow H_D(l) + 2$
20:            end if
21:        end if
22:    end for
23:    if $1 \geq c$ then
24:        $H_D(N) \leftarrow H_D(N) + 1$
25:    end if
26:    return $H_D$
27: end function
```

consists of at least two recurrence points.

Diagonal lines may stretch to the outer borders of the recurrence matrix, which requires to perform post-processing. The lengths of those lines are added to H_D, after having finished the execution of the inner loop. The length of the middle diagonal is added only, if its index 1 is greater than or equal to the Theiler corrector. The histogram H_D is returned in a final step.

Algorithm 2 has a time complexity of $\mathcal{O}(N^2)$. More precisely, up to $(N * (N-1))/2$ matrix elements are traversed. The histogram H_D captures diagonal line lengths ranging from 2 to N, resulting in a space complexity of $\mathcal{O}(N)$.

Detect Vertical Lines

Algorithm 3 Detect vertical lines.

```
1:  function DetectVerticalLines($R, N$)
2:      $H_V \leftarrow$ Zeros1DArray($N$)
3:      for $i \leftarrow 1$ to $N$ do
4:          $l \leftarrow 0$
5:          for $j \leftarrow 1$ to $N$ do
6:              if $r_{i,j} = 1$ then
7:                  $l \leftarrow l + 1$
8:              else
9:                  if $l \geq 2$ then
10:                     $H_V(l) \leftarrow H_V(l) + 1$
11:                 end if
12:                 $l \leftarrow 0$
13:             end if
14:         end for
15:         if $l \geq 2$ then
16:             $H_V(l) \leftarrow H_V(l) + 1$
17:         end if
18:     end for
19:     return $H_V$
20: end function
```

Algorithm 3 depicts the detection of vertical lines. The line lengths are captured in the histogram H_V, which is initialised as an empty one-dimensional array. The central component of the algorithm is a nested loop. The outer loop iterates over all columns of the recurrence matrix. Each column is referred to by a specific vector index. The inner loop iterates over all matrix elements of the column. The line length l is zero-initialised, before the first matrix element of a column is evaluated. It is incremented by one, if the current matrix element $r_{i,j}$ is a recurrence point. Otherwise, the histogram H_V is updated and l is reset to zero. The vertical lines that reach the outer border of the recurrence matrix are captured in a post-processing step, which is performed subsequently to the execution of the inner loop.

The algorithm for detecting vertical lines has a time complexity of $\mathcal{O}(N^2)$, since all N^2 elements of the recurrence matrix are accessed exactly once. It returns the histogram H_V that captures the amount of lines with a length ranging from 2 to N, similar to the detection of diagonal line structures. Hence, Alg. 3 also has a space complexity of $\mathcal{O}(N)$.

Detect White Vertical Lines

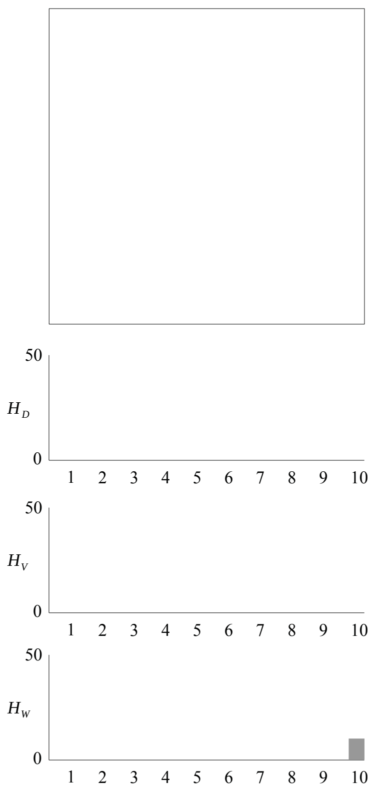
Algorithm 4 Detect white vertical lines.

```
1:  function DetectWhiteVerticalLines($R, N$)
2:      $H_W \leftarrow$ Zeros1DArray($N$)
3:      for $i \leftarrow 1$ to $N$ do
4:          $l \leftarrow 0$
5:          for $j \leftarrow 1$ to $N$ do
6:              if $r_{i,j} = 1$ then
7:                  if $l \geq 2$ then
8:                      $H_W(l) \leftarrow H_W(l) + 1$
9:                  end if
10:                 $l \leftarrow 0$
11:             else
12:                 $l \leftarrow l + 1$
13:             end if
14:         end for
15:         if $l \geq 2$ then
16:             $H_W(l) \leftarrow H_W(l) + 1$
17:         end if
18:     end for
19:     return $H_W$
20: end function
```

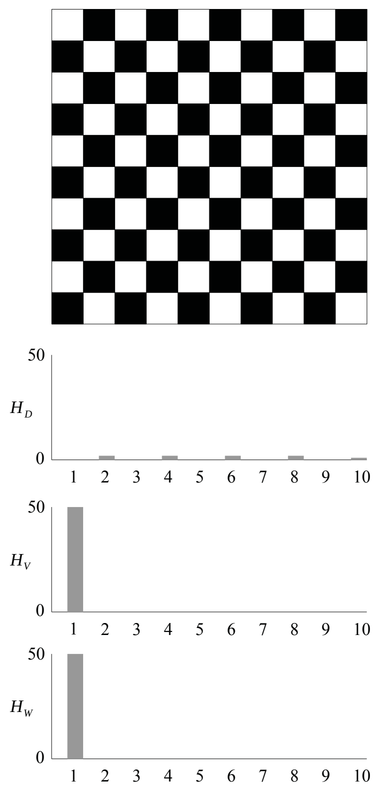
Algorithm 4 is largely consistent to Alg. 3. The detection of white vertical lines has the same time as well as space complexity as its vertical counterpart. The two algorithms differ regarding the operations performed, if a recurrence or non-recurrence point occurs. The functionality executed in the corresponding branches is inverse. The variable l refers to the length of the current white vertical line and the histogram H_W captures the white vertical line length histogram. The similarities of Alg. 3 and Alg. 4 allow to merge the detection of vertical and white vertical lines into a single algorithm, which is not depicted separately.

Histogram Density

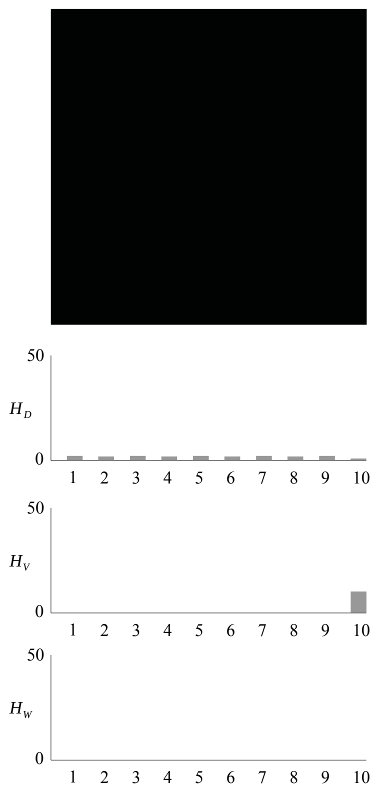
Each algorithm detecting line structures returns one histogram, resulting in a total of three histograms. The amount of memory consumed by each histogram depends on the number of input vectors in two ways. First, each histogram contains $N - 2$ elements, since lengths ranging from 2 to N are captured. This results in a space complexity of $\mathcal{O}(N)$, as stated before.



(a) Empty recurrence matrix.



(b) Checkerboard recurrence matrix.



(c) Completely filled recurrence matrix.

Figure 2.4: Histograms. Given a recurrence matrix created based on ten vectors. The histograms for (a) having an empty recurrence matrix, (b) a recurrence matrix containing a chequerboard pattern and (c) a completely filled recurrence matrix are presented. The minimum length one is chosen regarding diagonal, vertical and white vertical lines for the purpose of demonstration.

Second, the data type employed to encode the number of line length occurrences depends on the maximum line length N.

Figure 2.4 presents three extreme cases of recurrence plots, to elaborate on the impact of data type selection. The highest total number of line occurrences appears regarding the chequerboard pattern. Here, $N^2/2$ vertical and white vertical lines of length one are detected, indicating a quadratic relationship. As a result, a 64-bit unsigned integer value is required to encode a single histogram element, given a set of 2^{32} multi-dimensional input vectors.

2.2.2 State-of-the-Art in Recurrence Analysis Software

In the following, an overview of software that allows to create recurrence plots or to conduct recurrence quantification analysis is given[4]. The focus lies on software that is freely available and open source. Tools are distinguished regarding their functionality and their conformity with respect to the algorithms presented in 2.2.1. Their computational limitations are explained in Sect. 2.2.3, to motivate a novel computing approach for conducting RQA on very long time series.

RQA Software / RQA X

The tool *RQA Software* was published by Charles L. Webber Jr. [Webber Jr., 2016]. It comprises several command line programs to create recurrence plots and to conduct RQA. This software is only available in compiled form and requires a MS-DOS [Microsoft Corporation, 2016] environment. *RQA X*, an open source version of RQA Software for the OS X operating system, is provided by Andrew Keller [Keller, 2016]. RQA X is written in *Objective C* and has similar functionality.

The software by Andrew Keller supports the construction of recurrence matrices based on the fixed radius neighbourhood condition. Given an input time series, RQA X creates a matrix containing the raw similarity values in a first step. A number of similarity measures, such as the Euclidean and the Maximum norm, are available to compare pairs of multi-dimensional vectors. The software supports the creation of recurrence plots and their quantitative analysis. Regarding the latter, RQA X does only compute measures based on diagonal and vertical lines.

RQA X adheres to the algorithms presented in Sect. 2.2.1. It contains a modified version of Alg. 1, which computes the unthresholded similarity value for each pair of vectors only once and stores it in an intermediate matrix. The threshold condition is applied during the creation of the recurrence plot and during the inspection of the matrix regarding line structures. RQA X runs the computations in a single thread on a CPU. Additionally, it allows to execute batch jobs, with each job running in a separate thread.

There is a restriction regarding the size of recurrence matrices that can be processed by RQA X. The software is only capable of handling matrices created from up to 40,000 vectors[5], presumably to avoid running out of memory. If the edge lengths of a recurrence matrix exceed this amount, it can not be processed as a whole. It is possible to specify *epochs*, to compen-

[4]A continuously maintained list of software is available at `http://www.recurrence-plot.tk/programmes.php`.
[5]This value is hardcoded in the file `RQAPrefsController.m` as `maximumBatchWindowSize`.

sate this restriction. An epoch is a fixed-sized window along the middle diagonal of the full matrix [Webber Jr. and Zbilut, 2005, p. 52].

TISEAN

TISEAN, an acronym derived from *time series analysis*, is a collection of command line tools that allow to analyse numerical time series. The latest version 3.0.1 comprises utilities, e.g., for generating time series or performing noise reduction as well as conducting linear and nonlinear time series analysis [Hegger et al., 2016].

To compute the contents of a recurrence matrix, TISEAN includes two versions of the program `recurr`. One is written in C and one is written in FORTRAN. In the following, the C version is investigated. A single or multiple time series serve as input, combined with a set of additional parameters. The output of `recurr` is a list containing recurrence points, which are represented as pairs of integer values. This list is either written to *standard output* or written to a file. It becomes considerable long, given a large recurrence matrix and a relaxed threshold condition. For this reason, `recurr` allows to specify a percentage of recurrence points that are considered as output.

The current version of `recurr` only supports computing pairwise distances based on the fixed radius neighbourhood condition in combination with the Maximum norm. It employs a similar implementation of Alg. 1 as RQA X. `recurr` calculates only one half of the recurrence matrix excluding the LOI. This results in $N * (N - 1)/2$ similarity comparisons, which corresponds to a quadratic time complexity.

Commandline Recurrence Plots

Commandline Recurrence Plots allows to compute recurrence plots and to conduct recurrence quantification analysis [Marwan, 2016]. Version 1.13z of the tool is available in compiled form for a variety of platforms, including Linux, Mac OS X, Windows, HP-UX and Solaris. Access to the source code of version 1.14, which is written in C++, was granted by Norbert Marwan for the purpose of this thesis. Both versions are similar regarding the functionality implemented. Version 1.14 supports the calculation of recurrence plot density as well as diagonal line, vertical line and white vertical line measures.

The focus of Commandline Recurrence Plots is on the quantitative analysis. The identification of line structures is not performed based on a recurrence matrix persisted in the memory of the computing device. The similarity values referring to pairs of multi-dimensional vectors are computed while sequentially inspecting the elements within the diagonals and columns of the matrix. This allows to analyse recurrence matrices of large size. The result of the computations is a set of measures similar to Tab. 2.1.

The algorithms employed to detect line structures correspond to Alg. 2, 3 and 4. The application of the fixed radius neighbourhood condition allows to neglect one half of the recurrence matrix during the detection of diagonal lines. The detection of vertical and white vertical lines is combined into a single nested loop, which avoids traversing the same column twice.

Not persisting the recurrence matrix reduces space complexity to $\mathcal{O}(N)$. On the other hand, it requires to compute each element of the recurrence matrix one and a half times on average, one time during the detection of vertical and white vertical lines as well as half times during the detection of diagonal lines. In total, $(N^2) + (N * (N-1)/2)$ similarity comparisons have to be conducted, resulting in a quadratic time complexity.

2.2.3 Limitations of Existing Computing Approaches

All algorithms presented in Sect. 2.2.1 have a quadratic time complexity and perform the computations in a sequential manner. The corresponding implementations presented in Sect. 2.2.2 serve well for time series consisting of thousands of data points. However, there is a trend towards time series consisting of hundreds of thousands or millions of data points. This may either results from ongoing observations, e.g., in the context of environmental series, or the increased temporal resolution of observations, e.g., due to improved measurement methods. Here, state-of-the-art software conducting recurrence analysis suffers from several limitations.

Runtime limitation: The runtime of the implementations described in Sect. 2.2.2 increases drastically for very long time series, due to the time complexity of $\mathcal{O}(N^2)$. It takes for instance six hours and 18 minutes to quantitatively analyse a sample of the *Potsdam temperature profile*[6], using the Commandline Recurrence Plot software[7].

Memory limitation: Computing approaches that build on persisting recurrence matrices in the main memory are hardly able to cope with the increasing length of time series. There is a physical limit regarding the size of the recurrence matrices that can be stored, depending on the memory space available in the computing system. This limitation could be overcome by employing disk-based algorithms. Their application would slow down the processing heavily, due to higher access latencies. Therefore, such algorithms are out of scope regarding this thesis.

Resource limitation: Existing computing approaches to recurrence analysis only exploit a limited part of the computing capabilities of state-of-the-art hardware platforms. Current CPU designs use multiprocessor architectures that incorporate multiple cores on a single silicon die. It is also possible to leverage the parallel processing capabilities of graphics processors or other accelerators for general purpose computing tasks. Furthermore, there exist computing systems that contain multiple devices or architectures. Those advancements are currently only partially considered in the context of recurrence analysis computing (see 3).

Those limitations motivate the development of a novel computing approach to recurrence analysis, which is presented in the following chapters. Ideally, this approach should allow to:

1. process time series of almost arbitrary size,

[6]The sample employed spans from 1893 to 2011, resulting in 1,043,112 data points. It captures the temperature anomalies, which are defined as the deviation from the hourly average. The following parameter assignments have been found optimal after investigating the anomaly series: $m = 5$, $t = 3$, $\epsilon = 1.0$ and $d_{min} = v_{min} = w_{min} = 2$.

[7]The experiment was conducted on an Intel(R) Core(TM) i7-3820 CPU running at 3.60GHz. The runtime was determined as the average of five runs.

2. conduct the computations in an appropriate amount of time,
3. use all information provided by the input time series,
4. provide exact results, and
5. compute all quantitative measures.

Except from the last property, these constraints hold for creating recurrence plots as well as conducting recurrence quantification analysis.

2.3 The OpenCL Framework for Heterogeneous Computing

This thesis proposes to conduct recurrence analysis by performing the processing on parallel hardware architectures. A major goal is to enable support for devices from different vendors. OpenCL, short for *Open Computing Language*, allows to exploit the parallel computing capabilities of a variety of parallel computing hardware architectures, including [Khronos OpenCL Working Group, 2015, p. 25][8]:

- Many-core GPU designs,
- Multi-core CPU designs,
- *Field-programmable gate arrays* (FPGAs), and
- Other architectures, such as the Intel Many Integrated Core Architecture [Intel Corporation, 2016c].

Different versions of the OpenCL specification are available, defining the following contents of the framework:

- An *application programming interface* (API) that allows to offload computations to compute devices,
- Platform-specific runtimes that provide implementations of the OpenCL API,
- The programming languages *OpenCL C* (since version 1.0) and *OpenCL C++* (since version 2.1),
- The intermediate language *SPIR-V* (since version 2.1), and
- A set of libraries that provide common functionality.

In the following, relevant aspects of the OpenCL framework regarding this thesis are explained in detail.

[8]A continuously updated list of supported devices is available at [Khronos Group, 2016b].

Table 2.2: Architecture terminology. Comparison of the OpenCL terminology with the terminology used by the two GPU hardware vendors NVIDIA and AMD.

OpenCL	*Nvidia (since Fermi Architecture)*	*AMD (since Graphics Core Next 1.0)*
Compute Device	GPU processor	GPU processor
Compute Unit	*Streaming Multiprocessor*	*GCN Compute Unit*
Processing Element	*CUDA Core*	Single lane of *SIMD Unit*

2.3.1 Architecture

OpenCL introduces a specific architecture that abstract from the specific design of a *computing system*. There is a distinction between *host device* and *compute device*. A computing system comprises exactly one host device, typically a CPU, and one or more compute devices. Host and compute device may be identical, for example given a computing system that contains a single CPU. A compute device is further subdivided into one ore more *compute units*. Each compute unit contains one ore more *processing elements*.

The mapping from the abstract OpenCL concepts, e.g., compute unit, to parts of the actual hardware architecture is device-specific. In general, the subdivision of a compute device is in alignment with state-of-the-art GPU design. Table 2.2 depicts the mapping for two selected GPU architectures from the hardware vendors Nvidia [NVIDIA Corporation, 2009a, pp. 7–11] and AMD [Advanced Micro Devices, Inc., 2012, pp. 2–10].

The OpenCL API defines an interface on how the host device communicates with the compute devices. Each compute device is associated to a computing platform, which is vendor specific. The platform vendor provides an implementation of the OpenCL API as well as device drivers that enable OpenCL processing.

An OpenCL application is subdivided into *host program* and *kernels*. The host program runs on the host device and is responsible for steering the processing on the compute devices. The communication with the compute devices is realised via the OpenCL API. Bindings are available for different languages, including Python [Khronos Group, 2016c]. Among others, the API contains functionality to initialise an OpenCL environment, transfer data from and to the memory of the compute devices, and execute *kernels*.

A kernel encapsulates the compute intensive tasks meant for parallel processing within kernel functions, which are executed on the compute devices. Previous to OpenCL version 2.1, it was only possible to write kernel functions in OpenCL C, a subset of the *C99* programming language. Those kernels are compiled using a platform-specific compiler. The resulting binaries can only be executed by a certain device.

Starting from version 2.1, kernel functions can also be implemented in OpenCL C++, a subset of the *C++14* programming language [Guillon, 2015]. OpenCL C++ code can be converted into an intermediate representation using the language SPIR-V. This representation obfuscates implementation details but is at the same time portable across different platforms.

The RQA implementations referred to in the following employ functionality that is defined in version 1.1 of the OpenCL specification [Khronos OpenCL Working Group, 2011]. This version has been released in June 2011. The reason for relying on this outdated OpenCL specification is that it takes time until hardware vendors release implementations for specific versions. As an example, Nvidia enabled support for version 1.2 not before April 2015 [NVIDIA Corporation, 2015b, pp. 3–5]. Furthermore, selected Nvidia devices were considered to be compliant to version 1.2 by the Khronos OpenCL Working Group not before September 2015.

At this point, it is reasonable to assume that OpenCL 1.1 is the most wide-spread version of the standard. This is also due to the backwards compatibility, which is ensured until OpenCL 2.1 [Khronos OpenCL Working Group, 2015].

2.3.2 Memory Model

OpenCL introduces a specific memory model, that distinguishes between host and compute device memory. The host memory is equivalent to the main memory of the computing system and can only be accessed by the host program. Furthermore, the host can access specific parts of the memory of a compute device. Other parts of the compute device memory can only be accessed by kernel functions. The compute device memory is subdivided into:

Global memory: Available to all work-items of all work-groups.

Local memory: Available to all work-items of a work-group.

Private memory: Available to a single work-item.

Those memory regions can be accessed via reads and writes. The *constant memory* is a dedicated region within the global memory that allows to store immutable data. The three memory regions listed above are organised in a hierarchy. The global memory can be seen as the main memory of the compute device. It is usually the largest of the three memory regions, comprising several gigabytes. At the same time, it is associated with the highest access latency. In contrast, the local and private memory are considerably smaller, comprising only dozens of kilobytes, while at the same time providing considerably faster data access [Advanced Micro Devices, Inc., 2012]. Global memory should therefore be accessed as seldom as possible.

The host can only transfer data to the global memory of the compute device. Nevertheless, it is possible to allocate memory space within all three memory regions while executing a kernel function. If the space allocated in private memory is to large, data spills to the global memory. A compute device is only able to access its own memory. It can not access data that resides in the memory of other compute devices. Exchanging information between compute devices has to be realised via transferring data to the host memory during an intermediate step.

2.3.3 Command Execution

Each compute device is equipped with at least one *command queue*. The host program communicates with the compute devices by enqueuing commands. Among others, these commands refer to kernel execution, transfer of data to and from the global memory of the compute device and synchronisation between running tasks. There are several states that a command can enter:

Queued: The command resides within the command queue.

Submitted: The command has been submitted for execution on the device.

Ready: The command is scheduled for execution.

Running: The execution of the command has started.

Ended: The execution of the command has ended.

Complete: The execution of the command and its child commands has ended (new since version 2.0).

The transitions between these states are captured within an *event*. The event provides profiling information regarding the transition times, including:

Queued: *None* → *Queued*

Submit: *Queued* → *Submitted*

Start: *Ready* → *Running*

End: *Running* → *Ended*

Complete: *Running* → *Complete* (new since version 2.0)

The enqueuing of commands to a queue can be conducted *blocking* or *non-blocking*. If the enqueuing is blocking, the corresponding function call does not return until the command has reached the status *Complete*. If the enqueuing is non-blocking, the corresponding function call returns immediately. The latter requires that the dependencies between relevant resources are synchronised manually, e.g., that the processing of the input data starts after all relevant input data has been transferred. It can be enforced by flushing all commands that reside within the command queue. The corresponding function call does not return until all commands currently in the queue have reached the status *Complete*. A compute device processes commands either using *in-order* or *out-of-order* execution. The execution strategy applied depends on the concrete OpenCL implementation.

One particular command initiates kernel execution, which takes the number of *work-items* as an input argument. It is equivalent to the amount of processing tasks that should be executed concurrently. The actual number of kernel function instances running in parallel depends on the hardware architecture of the compute device and is usually smaller than the number of work-items. According to the taxonomy of Flynn, there exist the following execution models [Flynn, 1972]:

Single instruction, single data (SISD): A single instruction is performed on a single data item at the same time.

Single instruction, multiple data (SIMD): A single instruction is performed on multiple data items at the same time.

Multiple instructions, single data (MISD): Multiple instructions are performed on a single data item at the same time.

Multiple instructions, multiple data (MIMD): Multiple instructions are performed on multiple data items at the same time.

Modern multi-core CPUs adhere to the concept of MIMD by executing different threads on each core at the same time. Often they are also equipped with a dedicated SIMD unit that comprises a set of registers. Relevant technologies are for example Intel AVX [Intel Corporation, 2016b] or ARM Neon [ARM Limited, 2014]. Here, a single sequence of instructions is performed on multiple data items that reside in the same register. SIMD units are also integrated in GPU designs such as AMDs Graphics Core Next architecture [Advanced Micro Devices, Inc., 2012]. Nvidia proposes a concept that is referred to as *single instruction, multiple threads* (SIMT), executing a single thread per work-item.

Work-items are referenced by coordinates in up to three dimensions. The actual number of available dimensions is specific for each compute device. OpenCL refers to this d-dimensional index as *ND range*. The extent of the ND range for a specific kernel is defined within the function call that performs its enqueue.

The set of work-items is organised in *work-groups*, which have the same dimensionality as the ND range. The size of the work-groups is defined prior to kernel execution and can not exceed a maximum value that is device-specific. The creation of work-groups decomposes the full ND range into subsets of work-items. There exist *work-sizes* on two different levels:

Global work-size: The total number of work-items in each dimension.

Local work-size: The number of work-items within each dimension of a work-group.

The global work-size has to be specified explicitly, while the local work-size can be determined automatically by the OpenCL runtime. The work-sizes are kernel-specific and part of the kernel enqueue command. The compiled kernel along with its parameterisation is referred to as *kernel instance*. The execution of a kernel instance is finished, when the processing of all of its work-groups has finished.

The set of work-groups that are ready for execution are referred to as *work-pool*. Multiple work-groups of the same work-pool can run in parallel, depending on the OpenCL implementation provided by the device vendor. It may be the case that the execution of the work-groups is serialised, processing only one work-group at a time. The OpenCL specification does not enforce a specific order regarding the processing of the work-groups. Furthermore, it does not provide means to synchronise the execution of work-groups.

Work-items within the same work-group are executed in parallel. Each work-item may make independent progress. Depending on the implementation of the OpenCL runtime, only work-items following the same path of instructions are executed at the same time. The processing of work-items across multiple work-groups can be synchronised using *barriers*. The processing beyond a barrier continues only after all work-items belonging to a kernel instance have reached it.

2.3.4 Basic Workflow

The basic workflow of executing an OpenCL program is separated into multiple steps. For the purpose of simplification, the process is described for a host device and a single compute device. First, the host program identifies the OpenCL platforms available within the computing system. The host program creates a command queues for the compute device selected. After having set up the OpenCL environment, the host program reads the input data from its source, e.g., a file or a database. This data is transferred to a memory region within the global memory of the compute devices. Assuming a dedicated GPU that is attached to a PCI Express slot, it is for example required that the data is transferred over the PCI bus.

The host program initiates the kernel execution after the transfer of all relevant input data has finished. The compute device conducts the computations that are captured within the kernel function based on the parameters specified. Output data is generated by transforming the input data or synthesising new data. The kernel processing is finished, if the related operations have been applied to all work-items. Afterwards, the output data is transferred from the global memory of the compute device to the host memory.

3 Advanced Computing Approaches to Recurrence Analysis

The limitations presented in Sect. 2.2.3 hamper the analysis of very long time series, commonly exceeding one million data points. Approaches to enable their analysis have been developed in the recent past. They can be assigned to either one of the following categories:

- Parallelisation of the brute-force processing,
- Compaction of the input time series, and
- Approximation of the RQA measures.

Additionally, this chapter considers methods to obtain the pairwise vectors similarities using index data structures. To the best of the knowledge of the author, those data structures have not been applied to recurrence analysis computing in the past.

Approaches building on parallel processing address the quadratic relationship between the length of the time series to analyse and the resulting recurrence matrix by executing similar computations at the same time. In this way, the parallel computing resources of modern multi-core and many-core processors are exploited. The usage of index data structures, such as grid directories and multi-dimensional search trees, aim at reducing the number of pairwise vector similarity comparisons conducted during the creation of the recurrence matrix.

The number of computations conducted during the analysis can be reduced by decreasing the length of the input time series. This is achieved by applying compaction techniques, like the temporal aggregation of measurements. Note that this approach does not consider the modification of the brute-force operations. The last approach approximates the quantitative measures, in contrast to the exact results retrieved by the previous approaches. In this regard, the expensive operations to compute exact results are replaced by less expensive operations delivering inexact results.

The general ideas behind each of those computing approaches are presented in detail. Note that the parallel brute-force processing and the usage of index data structures are of particular interest regarding the remaining chapters of this thesis.

3.1 Parallel Brute-Force Processing

Recurrence analysis offers several opportunities to apply concepts from parallel computing. Again, the focus is especially on the recurrence quantification analysis. The corresponding processing is subdivided into three basic steps:

Table 3.1: Recurrence quantification analysis operators. For each operator, a category of atomic tasks is defined. Each task is considered atomic, being the smallest unit of execution without interfering with any other task of the same category. The maximum degree of parallelism (DOP) refers to the largest amount of tasks that can be processed simultaneously. It depends on the total number of multi-dimensional vectors N.

Operator	*Atomic Task*	*Maximum DOP*
create_recurrence_matrix	Similarity comparison and thresholding of a *single pair of input vectors*.	N^2
detect_diagonal_lines	Inspection of a *single diagonal* of the recurrence matrix regarding diagonal lines.	$2N-1$ (non-symmetric) / $N-1$ (symmetric)
detect_vertical_lines	Inspection of a *single column* of the recurrence matrix regarding vertical and white vertical lines.	N

1. Creation of the recurrence matrix (*create_recurrence_matrix*),
2. Detection of diagonal line structures (*detect_diagonal_lines*), and
3. Detection of vertical and white vertical line structures (*detect_vertical_lines*).

In the following, these processing steps are referred to as *operators* (see Tab. 3.1). The third operator subsumes the detection of vertical and white vertical lines, since they only differ regarding the interpretation of matrix element values. Unless stated otherwise, the term *vertical lines* hereafter refers to vertical structures consisting of ones and vertical structures consisting of zeros within the recurrence matrix.

Creating a recurrence plot does only require to compute the pairwise input vector similarities. To conduct RQA, it is also necessary to execute the *detect_diagonal_lines* and *detect_vertical_lines* operator. There are dependencies regarding the execution of the line detection operators and the *create_recurrence_matrix* operator. The former require that the recurrence matrix has been computed previously. If the binary matrix is available, it can be inspected regarding diagonal as well as vertical lines at the same time, since there are no mutual dependencies between the tasks of the two categories.

Regarding the parallel execution, a category of *atomic tasks* is specified for each operator. Each atomic task has the property of being fully independent, such that its execution can not be interfered by any other task of the same category running simultaneously. A *create_recurrence_matrix* task comprises computing and thresholding the pairwise similarity of two vectors. Computing a single matrix element does not require to consider data regarding any other element of the recurrence matrix. Up to N^2 similarity comparisons can be conducted concurrently, due to the quadratic structure of the matrix. This amount of atomic tasks per operator is referred to as maximum *degree of parallelism* (DOP).

The maximum DOP regarding the detection of line structures is selected such that it depends on the number of diagonals or columns within the matrix. The inspection of a single column is fully independent from any other column of the recurrence matrix. The evaluation of the matrix elements within the same column is performed using a sequential scan, to identify vertical and white vertical lines. As a result, data has only to be shared among elements belonging to the same column. This similarly applies to the inspection of diagonals regarding line structures.

Up to N vertical line detection tasks can be executed simultaneously, which matches the number of columns of the recurrence matrix. At most $N-1$ or $2N-1$ diagonal line detection tasks have to be executed, depending on whether the recurrence matrix is symmetric or not. This is equivalent to the maximum DOP of the operator.

Line detection tasks need to update the corresponding line length histograms to enable the computation of the RQA measures. The access to the histograms has to be synchronised to prevent race conditions between multiple tasks of the same category running in parallel.

3.1.1 Using Multi-Core CPUs

The algorithms presented in Sect. 2.2.1 are largely based on nested loops. These loops iterate over the elements of columns and diagonals of the recurrence matrix. Each column or diagonal is represented by an iteration of the outer loop. An iteration is referenced by a specific value of the corresponding loop variable.

As stated before, there are no dependencies between the processing of different columns and diagonals, which enables to conduct different iterations of the outer loops simultaneously. Using the *OpenMP* [OpenMP Architecture Review Board, 2016] framework, it is possible to chunk the set of outer loop iterations and distribute them across multiple computing resources, such as CPU cores. It implements a *fork-join-model* that is based on *shared memory processing* [Barney, 2016].

OpenMP distinguishes between two types of threads, one master thread and multiple worker threads. The latter are created on demand by the master thread and are executed in parallel, running within the same operating system process. Using OpenMP, parallel regions are specified within the source code by adding compiler directives. The master thread forks a set of worker threads that execute the corresponding code segment in parallel. When the execution within all worker threads has reached the end of the parallel region, a join operation is performed and the master thread continues. To exchange data, the worker threads access shared main memory.

OpenMP mainly addresses the parallel computing capabilities of multi-core CPUs. Starting from version 4.0, OpenMP allows to offload computations to accelerators, such as GPUs [OpenMP Architecture Review Board, 2013, Cramer et al., 2012]. However, the compiler support by hardware vendors is limited, for example in the *GNU Compiler Collection* (GCC). Viable support is only enabled for the *Intel Xeon Phi* platform [APPLIED PARALLEL COMPUTING LLC, 2015]. Although approaches to support Nvidia devices exist, according implementations are still rather immature [Beyer and Larkin, 2016, p. 51]. OpenACC is a competitive framework to OpenMP, providing similar functionality to offload computations to accelerators [OpenACC-standard.org, 2016]. The project recently received broader audience by AMD announcing its support [Advanced Micro Devices, Inc., 2015].

Since the support of accelerators regarding both approaches improved only recently, according features are out of the scope of this thesis. Note that employing the OpenMP framework is only one possibility to enable parallel processing on CPUs. It is considered here, since it allows to adapt existing implementations without great effort and is actively used in practice.

Commandline RQA Multithreaded

A first approach introducing parallel processing strategies to recurrence analysis has been conducted by *Commandline RQA Multithreaded*[1]. The source code is written in C++ and extended by OpenMP compiler directives.

Commandline RQA Multithreaded focusses explicitly on conducting "RQA for very long time series"[2]. This is achieved without persisting the recurrence matrix. All similarity values are computed while inspecting the diagonals and columns of the recurrence matrix. The software allows to apply the fixed radius neighbourhood condition in combination with a set of metrics.

The processing is structured as a nested loop, as depicted in Alg. 5. The outer loop iterates over all input vector indices, using the variable i. Each value of i identifies a specific diagonal and column of the matrix. As an example, the value one refers to the middle diagonal of the matrix and its first column. The functions to inspect a specific diagonal or column of the recurrence matrix require:

- the set of input vectors (X),
- the number of input vectors (N),
- the Theiler corrector (c),
- the loop iteration index (i), and
- the line length histograms (H_D or H_V and H_W).

The line inspection functions either iterate over all elements of the diagonal or column referred to by i. Thus, each iteration of the outer loop triggers the execution of two inner loops.

Parallel execution is achieved by inserting the OpenMP pragmas `parallel` and `for` front of the outer loop. This splits the set of diagonals and columns regularly into multiple chunks. Each chunk is processed within a separate worker thread. A single thread iterates over all indices i of its chunk. Distributing the chunks across multiple compute resources, e.g., CPU cores, leads to processing multiple columns and diagonals of the recurrence matrix simultaneously. For this reason, the access to the line length histograms has to be synchronised.

[1]The tool is developed by Norbert Marwan and has not yet been made publicly available. Access to the source code of version 1.1 has been granted for the purpose of this thesis.

[2]A comment in the source code file.

Algorithm 5 Processing as conducted by Commandline RQA Multithreaded.

```
 1: function ConductRQA($X, N, c$)
 2:     $H_D \leftarrow$ Zeros1DArray(N)
 3:     $H_V \leftarrow$ Zeros1DArray(N)
 4:     $H_W \leftarrow$ Zeros1DArray(N)
 5:     for $i \leftarrow 1$ to $N$ do
 6:         InspectDiagonal($X, N, c, i, H_D$)
 7:         InspectColumn($X, N, i, H_V, H_W$)
 8:     end for
 9:     return ComputeRQAMeasures($H_D, H_V, H_W$)
10: end function
```

3.1.2 Using Many-Core GPUs

Initial efforts to conduct recurrence analysis on hardware architectures beside CPUs have been presented in [Rybak, 2010]. It proposes to offload the parallel computations of pairwise vector similarities onto a graphics processor. The results are transformed into binary values using a similarity threshold and stored within the GPU memory as a recurrence matrix. The maximum degree of parallelism applied corresponds to the *create_recurrence_matrix* operator introduced before.

Evaluation results presented in [Rybak, 2010] indicate that the usage of GPUs is not reasonable for recurrence matrices constructed from only a few thousand input vectors. Here, a GPU implementation may be much slower than a CPU implementation. It is stated that the compilation of the GPU code is the dominant factor regarding the processing of small recurrence matrices. Hence, the application of a GPU computing is appropriate for recurrence matrices with increasing size. Additionally, [Rybak, 2010] states that it is possible to conduct the quantitative analysis by inspecting diagonals and columns of a recurrence matrix regarding line structures in parallel.

There are several limitations attached to the approach presented in [Rybak, 2010]. First, it does only allow to process recurrence matrices that fit in the memory of a GPU, rendering the analysis of very long time series impossible. This corresponds directly to the *memory limitation* described in Sect. 2.2.3. Second, the author uses an implementation based on the CUDA framework [NVIDIA Corporation, 2015a], which does only provide access to Nvidia GPUs. There is only few information given regarding the properties of this parallel implementation. Important details are missing, e.g., which storage format for the input and output data has been chosen. In addition, the approach for conducting RQA has only been sketched roughly and has not been implemented.

The implementation used to conduct the runtime analysis has not been made publicly available, preventing the verification of the runtime results. The main focus of the runtime analysis is on comparing a compiled pure CPU implementation with an implementation, whose GPU code is compiled during runtime. This leads to incomparable results due different overhead costs. This effect is further enhanced by only providing information regarding the GPU employed, not the CPU.

[Rybak, 2010] acknowledges the problem of memory limitation. The experiments conducted either use recurrence matrices that fit in the main memory of the computing system or the memory of the GPU. Recurrence matrices with a maximum size of $16,384^2$ elements regarding the CPU and $10,000^2$ elements regarding the GPU are chosen, rendering a direct comparison impossible.

In addition to [Rybak, 2010], [de Lima Prado, 2012] considers the quantitative analysis of recurrence matrices in the context of image analysis using GPUs. [de Lima Prado, 2012] focusses solely on the computation of the recurrence point density, as presented in Sect. 2.1.3. It comprises an evaluation that compares the performance of a CUDA implementation using different parameterisations with a C implementation extended by OpenMP pragmas that either runs on one or eight CPU cores. The experiment captures the runtime of computing the single RQA measure with increasing size of the recurrence matrix. The results indicate significantly lower runtimes regarding the CUDA implementation.

The considerations of [de Lima Prado, 2012] are attached with specific limitations. Although it addresses RQA, the publication does not provide information on how to compute the quantitative measures based on line structures. In addition, it does not give any insights on the properties of the CUDA or the C implementation used during the performance evaluation. The experiment presented does only consider small recurrence matrices constructed from hundreds of input vectors. [de Lima Prado, 2012] does furthermore not mention, which hardware architectures have been applied for evaluation. This renders an objective assessment of the runtime results impossible.

The approaches presented in [Rybak, 2010] and [de Lima Prado, 2012] demonstrate the applicability of GPU computing to the problem of recurrence analysis. Nonetheless, they miss to provide implementation details, important aspects regarding the processing, in particular the computation of the quantitative measures based on line structures, as well as a profound performance analysis. The CUDA framework used in both cases to leverage the computing capabilities of GPUs does only support devices of a single hardware vendor. Employing this framework would restrict the execution to a limited set of architectures. As a result, the *OpenCL* framework for heterogeneous computing is chosen regarding this thesis, to implement recurrence analysis in a parallel manner.

3.2 Index Data Structures

The computing approaches presented in the previous sections rely on computing the binary similarity of all pairs of multi-dimensional vectors. This exhaustive strategy conducts N similarity comparisons for each of the N multi-dimensional vectors, which results in a computational complexity of $\mathcal{O}(N^2)$. The quadratic complexity can be overcome by employing *index data structures*, which organise the set of input vectors in a specific manner. The algorithms to identify neighbouring objects using those index data structures allow to prune the number of similarity comparisons conducted and therefore have a lower computational complexity.

Index data structures are applied to recurrence analysis as follows. The input vectors reside in m-dimensional space, with m being the embedding dimension. The neighbourhood of a vector is either defined by a fixed threshold ϵ or a fixed number of neighbours. A specific property of recurrence analysis is that the set of *query points* is identical to the set of *data points*.

The problem of neighbour search in multi-dimensional space has been studied widely in the past decades. A vast number of methods that solve the problem for point data, as required by recurrence analysis, exist [Samet, 2006]. Prominent methods can be consolidated into two classes:

- Grid directories, and
- Multi-dimensional search trees.

Both classes of methods rely on partitioning the multi-dimensional space into subspaces. Given a multi-dimensional object, only those subspaces are inspected that might contain its neighbours. The set of subspaces that have to be inspected regarding neighbours depends on the partitioning strategy and the neighbourhood condition applied.

The partitioning into subspaces is attached with specific limitations. First, it requires to preprocess the set of input vectors to assign each of them to a specific partition, which increases the computational effort. The assignment highly depends on the distribution of the vectors in multi-dimensional space. Second, the number of partitions created correlates with the embedding dimension. Increasing the value of m results in a larger number of subspaces. This restricts the applicability of index data structures in general only to small embedding dimensions. The latter is commonly referred to as *curse of dimensionality* [Bellman, 2015, p. 94].

In the following, an overview over grid-based methods and tree-based approaches is given. The general concepts behind each of these categories are explained. This also includes the corresponding neighbour search algorithms.

3.2.1 Grid Directories

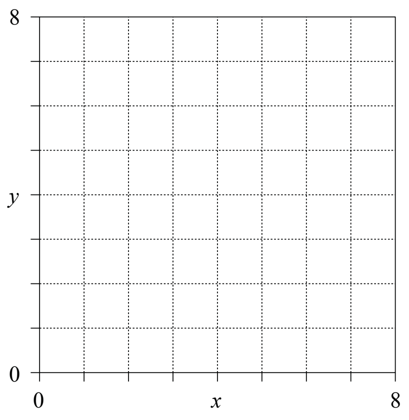
The following explanations are based on [Samet, 2006, pp. 130–163]. Grid directories were originally designed for the storage of data on hard disk drives. The partitions of the multi-dimensional space are referred to as *grid cells*. Each data point is assigned to a specific grid cell. A grid cell is defined by enclosing grid lines, where each grid line refers to specific dimension. There exist several grid types, which differ with respect to the distribution of grid lines within the multi-dimensional space:

Cartesian grid: The grid lines are placed at equal distance within all dimensions. The distance between two adjacent grid lines of the same dimension is one.

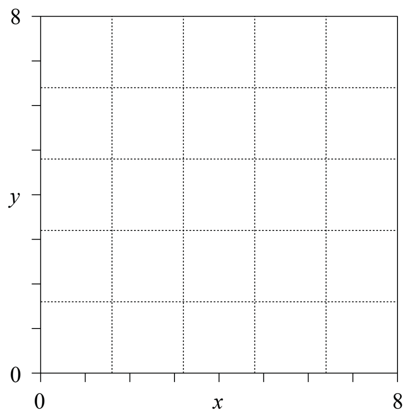
Uniform grid: The grid lines are placed at equal distance within all dimensions. The distance between two adjacent grid lines is less or greater than one.

Regular grid: The grid lines are placed at equal distance within a particular dimension. The distance applied varies between the different dimensions.

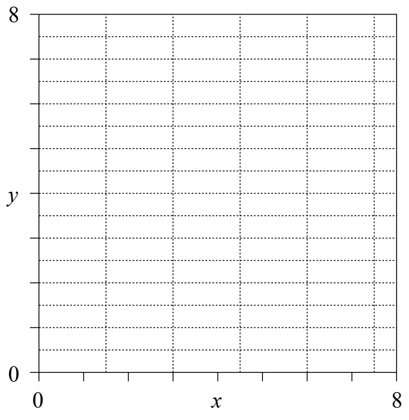
Irregular grid: The grid lines are placed at arbitrary distances within a particular dimension.



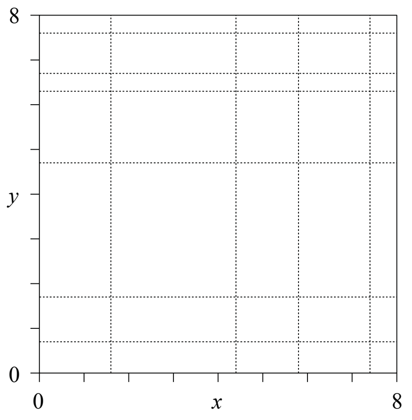
(a) Cartesian grid



(b) Uniform grid

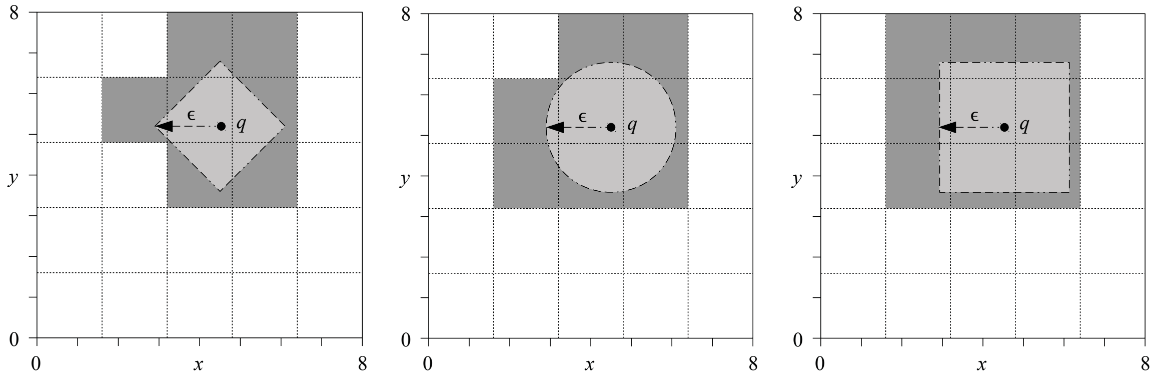


(c) Regular grid



(d) Irregular grid

Figure 3.1: Grid types.



(a) L_1 norm (b) L_2 norm (c) L_∞ norm

Figure 3.2: ϵ-neighbourhood search using a uniform grid. The example refers to a two-dimensional space with a fixed ϵ. The impact of the distance measure selected to the set of overlapping grid cells is demonstrated using three specific L_p norms. The number of grid cells that have to be inspected to determine the neighbours of the query point q increases with growing p.

Figure 3.1 depicts an example in two-dimensional space for each of the four grid types. The grid cell to which a data point belongs can be inferred from its coordinates. Given at least a regular grid, each coordinate value is divided by the equidistance of the corresponding dimension. The irregular grid requires that the distances of the grid lines are added up cumulatively or to perform a lookup in a separate data structure.

The density of data points in the grid cells depends on their distribution in the multi-dimensional space as well as the grid type employed. As an example, an uniform distribution of data points in combination with a uniform grid leads to roughly a constant number of data points per grid cell. Given an arbitrary distribution of data points, the alignment of the grid lines can be optimised, such that each grid cell contains almost the same number of data points. This procedure can be motivated by allocating a fixed size of memory for each grid cell.

Neighbour Search

The overall goal regarding neighbour search using a grid data structure is to visit only those grid cells that may contain neighbours of a query point given. Applying a fixed radius condition, a virtual ϵ-neighbourhood can be formed around a query point. Only those grid cells have to be inspected regarding relevant data points, which overlap with this neighbourhood. The extent of the neighbourhood depends on the distance measure applied. Figure 3.2 visualises the impact of selected L_p norms that are relevant in the context of recurrence analysis (see Sect. 2.1.2).

The search for k nearest neighbours, which resembles the fixed amount of nearest neighbours condition in the context of recurrence analysis, begins with identifying the grid cell in which the query point is located. This cell is referred to hereafter as *source cell.* First, the source cell is inspected regarding neighbours of the query point. The neighbour search is finished, if the source cell already contains the k closest data points. If not, the search is expanded to the grid

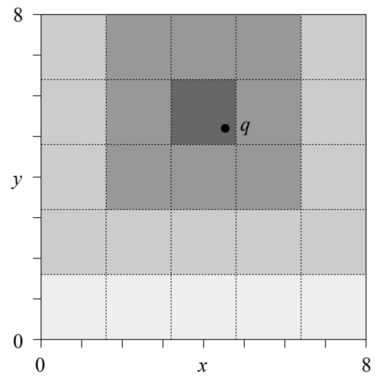


Figure 3.3: k-Nearest neighbour search using a uniform grid. The probability that an object belongs to the k nearest neighbours of query point q decreases with growing distance. The decreasing probability is depicted using paling colouring of the grid cells. Grid cells having the same colouring belong to the same probability level.

cells adjacent to the source cell.

The probability that a grid cell contains at least one of the k neighbours decreases with increasing distance from the query point. This property is demonstrated in Fig. 3.3 using a 2-dimensional uniform grid. Each ring of grid cells around the source cell is hereafter referred to as *probability level*. Strategies based on these probability levels have been presented in [Piegl and Tiller, 2002] and [Franklin, 2005]. Both approaches traverse the probability levels in order of increasing distance and inspect all cells of a probability level. In [Di Angelo and Giaccari, 2011], an approach is presented that uses the distance between the query point and the borders of its source cell to exclude parts of probability levels from neighbour search.

3.2.2 Multi-Dimensional Search Trees

The following explanations are based on [Samet, 2006, pp. 14–89]. A search tree partitions the multi-dimensional space into subspaces and organises them in a hierarchical structure. Each subspace may be partitioned further into a set of sub-subspaces, and so on. There exist a variety of tree types, including:

- the *range tree* [Bentley, 1979, Bentley and Maurer, 1980],
- the *quad tree* [Finkel and Bentley, 1974], and
- the *k-d tree* [Bentley, 1975].

Each of those types has specific advantages and disadvantages, which have been discussed widely in the according literature. For each of those types, there exist variations of the original tree definitions, modifying their structure and behaviour. We have chosen the k-d tree as one representative that has several advantages but also limitations that are explained in detail hereafter.

K-d Tree

The k-d tree is a binary tree data structure, where each non-leaf node has exactly two children, the *low son* and the *high son*. The labelling of the children is in alignment with the partitioning strategy of the k-d tree, where each tree level corresponds to a dimension of the multi-dimensional space. Given a dimension d, all nodes stored within the sub tree for which low son serves as root have smaller values regarding d than the parent node of low son. This concept is applied vice versa to the high son.

There are several strategies for assigning a dimension to a specific tree level. Regarding this thesis, it is assumed that the dimensions are assigned in ascending order of their index, beginning at the root level. If the k-d tree contains more levels than dimensions available, the assignment strategy is executed in a rolling fashion. This procedure has been initially proposed in [Bentley, 1975].

Concrete implementations of the k-d tree may differ for example regarding the semantics of non-leaf nodes, which can either be elements of the set of data points or points created artificially. The latter may only specify the value that partitions the set of data points within one dimension. In addition, there may also be different strategies for assigning dimensions to tree levels.

The usage of k-d trees for recurrence analysis requires to consider the following two phases:

- the *construction*, and
- the *neighbour search*.

Both phases are described below.

Construction. A k-d tree can be created using two different methods, either:

1. by inserting data points sequentially, or
2. by inserting the set of data points as a bulk.

In the following, both methods are examined for the case that data points serve as leaf and non-leaf nodes.

Regarding the sequential approach, the first data point inserted serves as root node. Data points inserted afterwards require to traverse additional tree levels. As stated before, each level is mapped to a specific dimension. Traversing a level, the attribute value of the node currently visited is compared to the corresponding value of the data point to insert. The traversal is continued either in the left or right subtree, depending on the result of the comparison. The traversal stops, if a leaf node of the tree is reached. The new data point is either inserted as low son or high son of the leaf node.

The sequential insertion method is suitable for data sets that grow during the process of creating the k-d tree. However, the order in which data points are inserted influences the tree balancing. As a result, the structure of two k-d trees may differ completely, although the same set of data points is captured. Moreover, the sequential method does not guarantee fully balanced trees, which is an essential property for conducting neighbour search efficiently.

Balanced k-d trees can be created by inserting the complete set of data points as a bulk. This requires that all data points are known prior to the tree construction. An approach commonly used is the median-based tree construction. The median either refers:

1. to the value range regarding the dimension considered at a specific tree level, or
2. to the number of data points to organise in a subtree.

The tree construction starts with the complete set of data points. The data point considered to be the median of the full data set is assigned as the root node. The remaining nodes are assigned either to the left or right subtree of the root node. The determination of the median is applied likewise to the set of data points assigned to both subtrees. This process is continued until all leaf nodes of the k-d tree are determined.

The decision regarding the assignment of a data point to a subtree differs between the two median-based approaches. The first one computes the median of the attribute values of the dimension considered at the current tree level. The data point containing the value closest to the median value may be selected as root node of the subtree. A data point is assigned to the left sub-subtree, if its attribute value is smaller than the median value. If not, it is assigned to the right sub-subtree.

The second method refers to the size of the set of data points to organise in a subtree. Here, the data points are ordered regarding their attribute value of the dimension considered at the current tree level. The data point that is located at the median index of the ordered set serves as the root node of the subtree. The data points with a lower index are assigned to the left sub-subtree. The data points with a higher index are assigned to the right sub-subtree. Figure 3.4 demonstrates the application of this method using a synthetic example.

Approaches based on bulk insertion are suitable for recurrence analysis, since all N input vectors are known prior to the analysis. This allows to create balanced k-d trees, e.g., using median-based approaches, enabling efficient neighbour search.

Neighbour Search. The k-d tree subdivides the multi-dimensional space into a set of partitions, similar to grid files. Given a query point, only those partitions have to be inspected, which potentially contain neighbours. Again, there is a distinction between ϵ-neighbourhood and fixed amount of nearest neighbours. The search using a fixed radius inspects all subtrees that intersect with the ϵ-neighbourhood. Depending on the similarity measure applied, the set of subtrees to inspect may vary (see Fig. 3.5).

The L_∞-norm creates a rectangular neighbourhood around the query point, as depicted in Fig. 3.5c. It has been shown that the worst case complexity of finding neighbours within such rectangular regions is $\mathcal{O}(d * N^{1-1/d} + k)$ [Lee and Wong, 1977], where d refers to the dimensionality of the search space, N to the number of data points and k to the amount of neighbours found. However, the complexity of an ϵ-neighbourhood search within a balanced k-d tree is $\mathcal{O}(log_2 N + k)$.

The size of the set of data points N has to be at least $(2^{d+1} - 1)$, to partition the multi-dimensional space in all of the d dimensions. This condition holds under the assumption that data points also serve as non-leaf nodes.

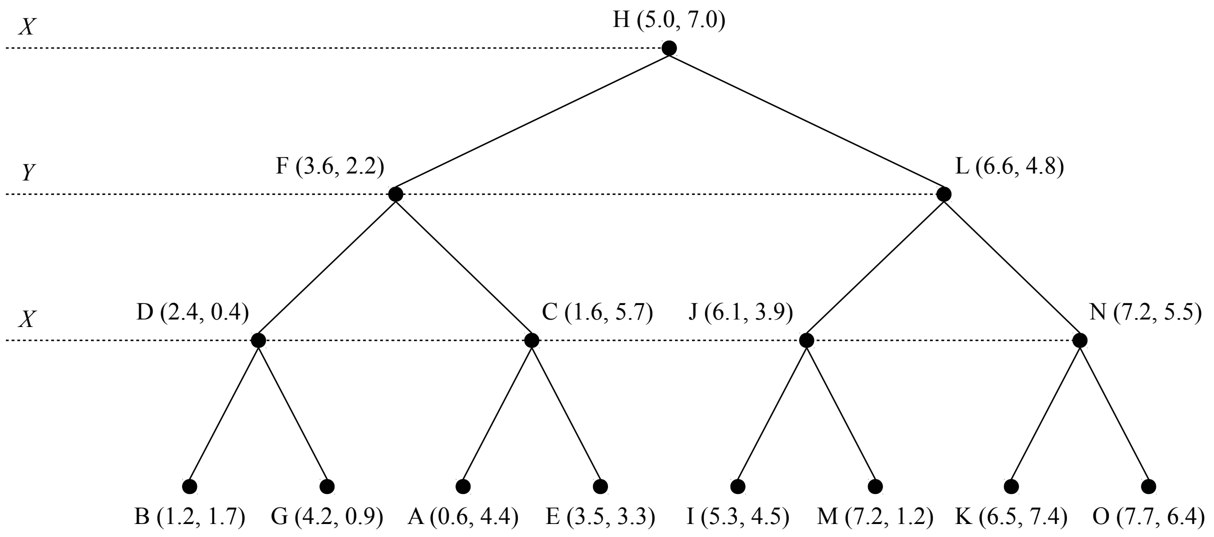


Figure 3.4: Balanced k-d tree. The tree is based on the data points from Tab. 3.2. It in constructed using a median-based approach referring to the number of data points to organise in a subtree. This results in a fully balanced binary structure. Data points serve as leaf and non-leaf nodes. The split dimension on each tree level is indicated on the left side of the k-d tree.

Table 3.2: Data points. The table contains the data points from which the k-d tree in Fig. 3.4 is constrcuted. The data points reside in two-dimensional space.

Label	x	y
A	0.60	4.40
B	1.20	1.70
C	1.60	5.70
D	2.40	0.40
E	3.50	3.30
F	3.60	2.20
G	4.20	0.90
H	5.00	7.00
I	5.30	4.50
J	6.10	3.90
K	6.50	7.40
L	6.60	4.80
M	7.20	1.20
N	7.20	5.50
O	7.70	6.40

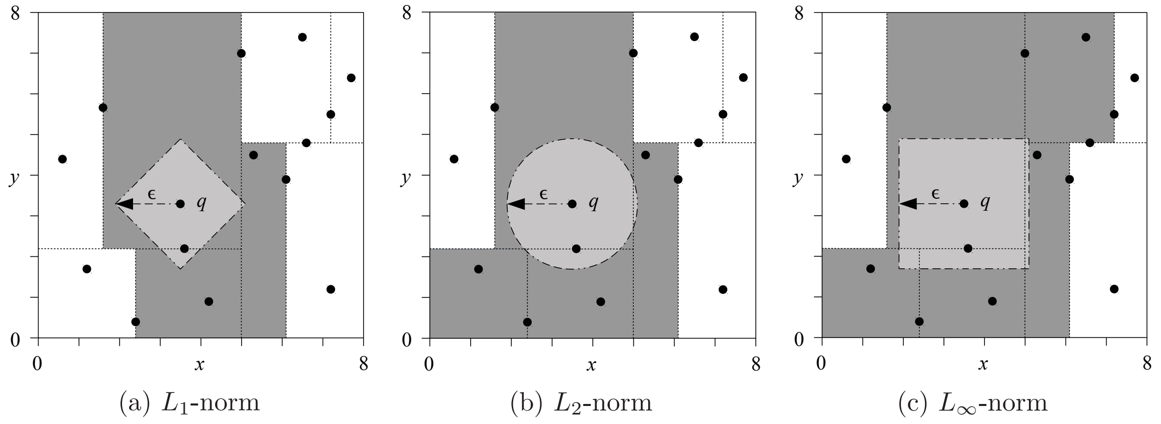


Figure 3.5: K-d tree ϵ-neighbourhood. The k-d tree from Fig. 3.4 is depicted in a two-dimensional coordinate system. The dotted lines represent the one-dimensional split planes. The data points located on such lines are the root nodes of the corresponding subtrees. The data point E serves as query point q. The extent of the ϵ-neighbourhoods around q influences the set of subtrees that have to inspected regarding potential neighbours.

3.3 Compaction of Time Series Data

There have been made attempts to circumvent the limitations described in Sect. 2.2.3 by compacting the time series data that serves as input for the recurrence analysis. A common approach is lowering the temporal resolution of the input series by using average values, e.g., daily averages instead of hourly measurements. This considerably reduces the amount of input data, but also causes a loss of information.

In [Chen and Yang, 2012], an approach is presented that compacts the input time series by applying *wavelet decomposition*. The input signal is decomposed into two parts, one referring to high frequencies and the other to low frequencies. Each of those decomposed signals has roughly half of the length of the original signal, depending on the wavelet function applied. The computational effort can be reduced roughly by a factor of four, if the recurrence analysis is performed only on one of the decomposed signals. This effect can be amplified by decomposing the decomposed signals again.

This approach has several issues. First, the application of the wavelet decomposition leads to a loss of information, since the full frequency band of of the input signal is separated. This property may also serve in favour of the analysis, since it allows to remove noise. Second, there exist numerous wavelet functions with different characteristics, which requires a sensible selection. Third, the most relevant frequency band has to be identified, requiring a prior inspection.

3.4 Approximation of RQA Measures

Instead of computing the exact values of the quantitative measures, approximate approaches may be applied. The goal is to reduce the runtime for retrieving the analytical results. This is either achieved by conducting the same operations but enforcing a limited precision or by conducting less expensive computations. The first branch refers to the application of data types that encode a value using a reduced number of bits, e.g., using 16-bit floating point numbers instead of representing values using 32 bits [Konsor, 2012], thus enabling an increase in computational throughput.

The second branch comprises several approaches relying on less expensive computations, including for example the activation of *relaxed math operations* using the `-cl-fast-relaxed-math` option during the compilation of OpenCL kernels (see Sect. 2.3). A completely different approach is proposed in [Schultz et al., 2015] to approximate the recurrence rate and diagonal line based measures such as determinism. The basic steps conducted are:

1. Concatenating input vectors to capture sequences of system states,
2. Counting the number of occurrences of each unique sequence, and
3. Discretising the state space by applying a regular grid.

As explained in Sect. 2.1, a diagonal line within the recurrence plot states that a system evolves similarly in two different epochs over a number of consecutive time steps. In the most extreme case, the evolution may be identical in both epochs, referring to the same sequence of unique system states. The approach introduced in [Schultz et al., 2015] captures sequence instances by concatenating input vectors. These instances are represented by matrices with a dimensionality of d-by-v, with d being the dimensionality of the input vectors and v being the length of the sequences. Figure 3.6 depicts a concatenation example for $d = 2$ and $v = 3$.

According to [Schultz et al., 2015], the number of instances of each unique sequence correlates with the number of diagonals of length v. It is proposed to sort the list of d-by-v matrices, so that all instances referring to the same unique sequence are stored consecutively. The sorted list of instances is scanned to create a histogram that captures the amount of instances per unique input vector sequence. It allows to calculate the number of *pairwise proximities* regarding a sequence length v ($\mathcal{PP}^{(v)}$). It is defined as the sum of squares regarding the number of instances per unique sequence.

It is shown that the recurrence rate is equal to $\mathcal{PP}^{(1)}$, assuming that the fixed radius neighbourhood condition in combination with $\epsilon = 0$ is applied. It is also proven that it is possible to calculate the exact value for determinism using pairwise proximities for different values of v. Here, the same assumptions as for the recurrence rate are applied.

The space in which the multi-dimensional vectors reside is discretised to approximate $\mathcal{PP}^{(v)}$ for $\epsilon > 0$. Each sequence M_i is divided element-wise by the parameter δ, which is commonly set to $\delta = 2\epsilon$. The division results are furthermore rounded off. This aligns the individual vectors to the structure of a uniform grid. This reduces the number of potential unique sequences and increases the number of instances per unique sequence, if the vectors are not already aligned.

M_1

0.0	0.7	1.0
1.0	0.7	0.0

M_4

0.7	0.0	-0.7
-0.7	-1.0	-0.7

M_7

-1.0	-0.7	0.0
0.0	0.7	1.0

M_2

0.7	1.0	0.7
0.7	0.0	-0.7

M_5

0.0	-0.7	-1.0
-1.0	-0.7	0.0

M_8

-0.7	0.0	0.7
0.7	1.0	0.7

M_3

1.0	0.7	0.0
0.0	-0.7	-1.0

M_6

-0.7	-1.0	-0.7
-0.7	0.0	0.7

M_9

0.0	0.7	1.0
1.0	0.7	0.0

Figure 3.6: Input vector concatenation. The concatenations depicted refer to the recurrence vectors $\vec{x}_1$ to $\vec{x}_{11}$ from Fig. 2.1. Each of the matrices M_1 to M_9 is a concatenation of three consecutive vectors, e.g., M_1 concatenates $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$. M_1 and M_9 refer to the same unique sequence since $M_1 = M_9$. The matrices M_2 to M_8 are instances of different unique sequences. In total, this results in eight unique input vector sequences.

[Schultz et al., 2015] claims to drastically reduce the runtime for computing the recurrence rate as well as diagonal line based measures. As an example, the calculation of the approximate pairwise proximities for a particular time series consisting of one million data points allegedly consumes only 0.19 seconds. That is a runtime reduction of more than five magnitudes compared to analysing the almost equally long anomaly series of the Potsdam temperature profile using the Commandline Recurrence Plots software (see Footnote 6 in Sect. 2.2.3). It is stressed that the MATLAB implementation used to conduct the experiment is only capable to process vectors with a dimensionality of one, which simplifies the reconstruction of the state sequences as well as their ordering. Hence, the evaluation results can hardly be generalised.

Appendix A presents an implementation based on Python 2.7 that conducts the approximation of the measures *RR* and *DET* for input vectors of arbitrary size. The approximate pairwise proximities are computed by calling the function `pp_approx` (see Line 35). Each matrix, representing a instance of a state sequence, is hashed using the function `hash_matrix` in Line 48. This function (see Line 30) transforms the matrix into its character representation and maps it to an integer value. The unique hash values are determined by calling the function `unique` from the Python package *NumPy* (see [NumPy developers, 2014b]). This function returns the unique entries of an array as well as the number of their occurrences based on sorting (see [NumPy developers, 2014a]). The sorting is by default conducted using the quicksort algorithm, as proposed by [Schultz et al., 2015].

Executing the Python implementation on the anomaly series of the Potsdam temperature profile takes 61.47 seconds on average[3]. This is a runtime reduction of still more than three magnitudes. Table 3.3 compares the exact values for the RQA measures recurrence rate and

[3]The optimal parameter assignments for the analysis of the anomaly series previously described have been employed. The experiment was conducted on an Intel Core i5-4288U CPU running at 2.60GHz. The time elapsed refers to the average of five executions.

Table 3.3: Exact vs. approximate RQA results. The values depicted refer to the anomaly series of the Potsdam temperature profile from 1893 to 2011. An embedding dimension $m = 5$, a time delay $t = 3$ and a similarity threshold $\epsilon = 1.0$ in combination with the L_2 norm are applied.

RQA Measure	*Exact*	*Approximate*	*Deviation (%)*
RR	0.12089	0.19817	63.92
DET	0.93575	0.93694	0.12

determinism with their approximations. There is deviation of more than 60% regarding the recurrence rate. The approximate value for determinism is almost identical to the exact result.

The authors of [Schultz et al., 2015] state that the mean error regarding the measures computed increases with growing values of ϵ as well as growing minimum lengths of diagonal lines. Mean errors of more than 80% could be observed. They miss to provide means to assess the quality of the approximate RQA results, given an arbitrary analysis scenario. Hence, the approximation approach may serve as an initial estimation but can hardly replace the exact results. Furthermore, the approach does not provide information regarding vertical and white vertical line based measures.

4 Scalable Recurrence Analysis (*SRA*)

This chapter introduces a novel computing approach to recurrence analysis, which is in the following referred to as *Scalable Recurrence Analysis* (*SRA*). It overcomes the limitations of existing computing approaches described in Sect. 2.2.3, including the runtime limitation, the memory limitation and the resource limitation. This is achieved by applying concepts from parallel processing on two levels.

On the *top level*, the processing of a recurrence matrix is subdivided into a set of sub matrices. This division enables to process multiple sub matrices using multiple compute devices at the same time. The concept *divide and recombine* is applied to organise the analytical processing. *Carryover buffers*, which are additional global data structures, are used to share data regarding line lengths among adjacent sub matrices. A custom sub matrix processing order guarantees that the line structures are detected correctly. The combination of those concepts ensure scalability by being able to employ multiple compute devices simultaneously. These concepts are addressed in the first section of this chapter.

The *bottom level* parallelisation is heavily based on the concepts presented in Sect. 3.1. Here, the computations related to RQA are subdivided into multiple operators. For each operator, a single type of atomic task is defined, so that the processing of each task is fully independent of any other task of the same type. This allows to conduct the processing of every operator in a massively parallel manner. Depending on the maximum degree of parallelism, scalability is achieved by being able to distribute each set of tasks among a varying number of parallel cores. The mapping of the low level parallelisation to OpenCL is described in the second section of this thesis, whereas the concrete execution pipeline is the topic of the third section.

In the following, the two parallel processing strategies are described in detail. Contrary to the computing approaches presented before, *SRA* enables the processing of very long time series in a runtime-efficient manner. This is achieved by leveraging massively parallel computing hardware, in particular GPUs. *SRA* allows to analyse all data captured in a time series, without any loss of information. The computations performed deliver exact results regarding all quantitative measures presented in Tab. 2.1.

4.1 Divide and Recombine

Existing computing approaches to recurrence analysis either rely on materialising the full recurrence matrix within the memory of a compute device or do not persist the matrix at all. The first option restricts the size of recurrence matrices processable according to the amount of memory available. The latter allows to compute recurrence matrices of arbitrary size at the cost of having to compute matrix elements multiple times. *SRA* divides the recurrence matrix into multiple sub matrices to eliminate both restrictions. Each sub matrix is processed individually by a single compute device, providing intermediate results. They are merged into global data structures that serve as input for the computation of the final results.

SRA can be treated as a specific instance of the concept *divide and recombine* (D&R), which was initially presented in [Guha et al., 2012]. Its origin lies in the problem of applying statistical methods to large data sets. Processing those data sets as a whole using traditional computing approaches is inefficient. Furthermore, the aggregation of input data would likely cause a loss of information. D&R is subdivided into:

- Computations to create *subsets* (S),
- Analytical computations *within* subsets (W), and
- Recombination computations *between* the subset-specific outputs (B).

The S computations split the input data set into subsets. D&R distinguishes several division strategies, e.g., using variable-based conditions. The W computations perform the actual analytical tasks on the subdivided data. D&R suggests that each subset is processed without transferring information between subsets. As an additional criterion, the analytical computations applied should be *embarrassingly parallel*, requiring a large number of independent tasks. The B computations are responsible for condensing the individual subset results. There exist multiple recombination strategies, including:

- *analytical* recombination, and
- *visual* recombination.

The first performs analytical operations on the recombined subset results. The latter refers to a detailed visualisation of the analytical subset results.

4.1.1 Division Strategy

A key aspect of applying D&R to recurrence analysis is separating the full recurrence matrix into multiple sub matrices (see Fig. 4.1), which is achieved by applying an uniform grid to the recurrence matrix (see Sect. 3.2.1). The axes of the matrix are chunked at multiples of a fixed number of input vectors k, with $k \leq N$ and N being the total number of input vectors. This value represents the edge length of the sub matrices. Similar concepts have already been applied for example to the problem of *nearest neighbour search* [Kato and Hosino, 2009].

As a consequence, sub matrices with a uniform extent of $k \times k$ are created. If $(N \bmod k) \neq 0$, sub matrices with different extents appear at the outer borders of the recurrence matrix. This includes sub matrices with an extent of $(N \bmod k) \times k$, $k \times (N \bmod k)$ and $(N \bmod k) \times (N \bmod k)$, which are hereafter referred to as *residual sub matrices*.

The analytical computations performed on each sub matrix match the operators defined in Sect. 3.1. There is a large number of independent tasks that can be be executed simultaneously for each operator, which corresponds to the requirements of D&R. The original concept assumes that there is no communication between the individual analytical operations. This

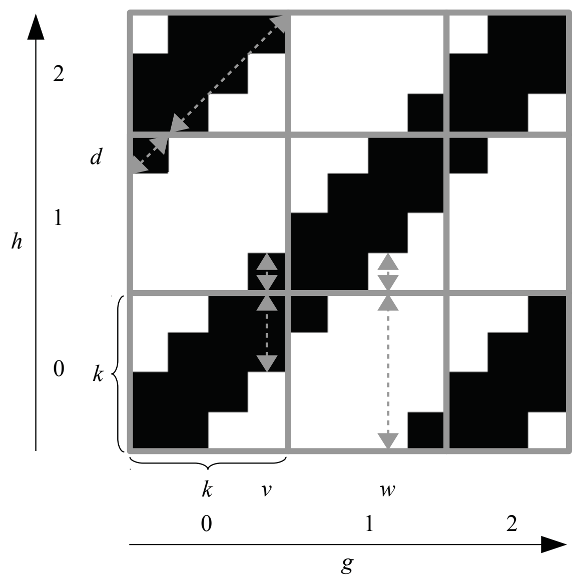


Figure 4.1: Recurrence matrix division. The recurrence matrix from Fig. 2.2 is divided into multiple sub matrices. g and h refer to two sub matrix indices. An uniform grid with the edge length $k = 4$ is applied. This results in sub matrices with an extent of 4×4, 4×3, 3×4 and 3×3. The highlighted line structures d, v and w stretch over multiple adjacent sub matrices. Each of those structures is split into two partial lines.

constraint has to be relaxed in the context of RQA, due to the interdependencies between the *create_recurrence_matrix* and the line detection operators. Furthermore, there has to be an exchange of data between sub matrices, regarding the detection of diagonal and vertical lines.

4.1.2 Carryover Buffers

Line structures may cross the borders of adjacent sub matrices (see Fig. 4.1), affecting the quantitative analysis. This makes it necessary to share data regarding line lengths between sub matrices. Individual histograms are created for each sub matrix by scanning its diagonals and columns sequentially. If the inspection reaches the borders of a sub matrix, the current line lengths are stored within *carryover buffers*. After having finished the processing of a sub matrix, the buffer values are used as an input to detect line structures within the adjacent sub matrices.

There exists a separate buffer for each of the three line types:

(I) Diagonal lines carryover buffer (C_D),

(II) Vertical lines carryover buffer (C_V), and

(III) White vertical lines carryover buffer (C_W).

The first is employed within the *detect_diagonal_lines* and the second and third within the *detect_vertical_lines* operator. The size of each buffer corresponds to the amount of line detection tasks that can at most be executed simultaneously. For each of those tasks, a separate carryover buffer element is provided. Hence, the number of elements per carryover buffer is equal to the maximum degree if parallelism regarding the corresponding line detection operator:

C_D : $2N - 1$ (non-symmetric) / $N - 1$ (symmetric)

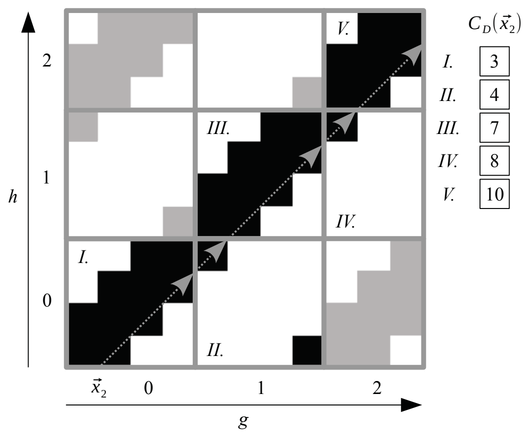
C_V : N

C_W : N

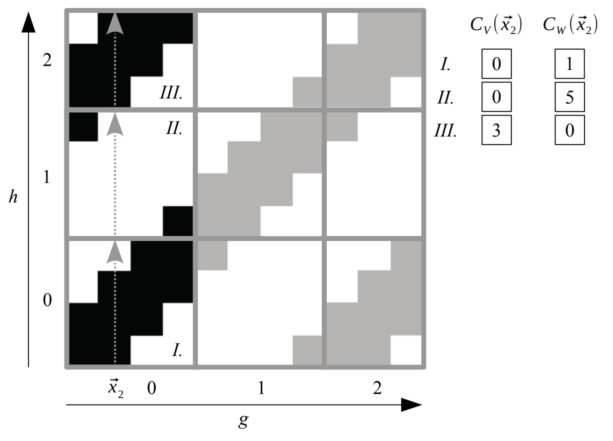
Note, there is a linear correlation between the number of input vectors N and the size of the carryover buffers. Their functionality is demonstrated in Fig. 4.2. Here, only those carryover buffer elements are displayed, which are relevant for the specific line detection tasks.

The data type selected for representing carryover buffer elements depends on the maximum line length, which is equal for diagonal and vertical lines. This is due to the quadratic structure of the recurrence matrix. Note that the number of elements per carryover buffer does only depend on the number of global line detection tasks. This allows to create sub matrices of arbitrary extent, without having to modify the structure of the carryover buffers.

The carryover buffers are designed to be space efficient. Assuming a time series consisting of one million data points, less than 16 MiB are required to store all three carryover buffers. It is assumed that every carryover buffer element is represented by a 32-bit integer value. The vertical and white vertical carryover buffer each consume ≈ 4 MiB. The diagonal lines carryover buffer consumes either ≈ 8 MiB or ≈ 4, depending on the presence of matrix symmetry. In

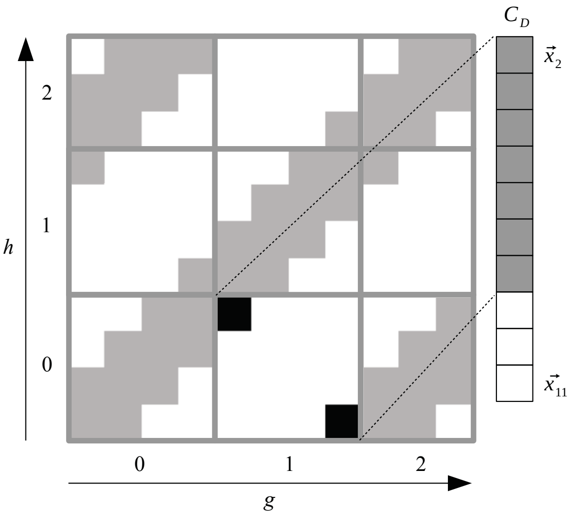


(a) Diagonal

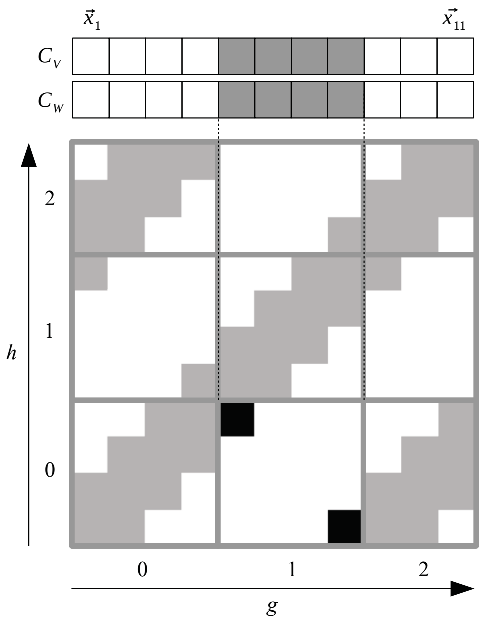


(b) Column

Figure 4.2: Carryover buffer functionality. The same partitioning scheme as depicted in Fig. 4.1 is used. The inspection of a single (a) diagonal and (b) column of the recurrence matrix is demonstrated. Each inspection corresponds to a single detection task within the full matrix. These tasks are referenced by the input vector $\vec{x}_2$. The diagonal line detection task considers to the carryover buffer element $C_D(\vec{x}_2)$, whereas the vertical line detection task refers to $C_V(\vec{x}_2)$ and $C_W(\vec{x}_2)$. The global detection tasks are separated into multiple subtasks, due to the uniform grid. Each subtask, highlighted by a dotted arrow, belongs to a specific sub matrix. The indices (a) $I.$ to $V.$ and (b) $I.$ to $III.$ indicate all relevant sub matrices as well as their processing order. The content of the carryover buffer elements after processing each sub matrix is captured on the right side of the recurrence plots.



(a) Diagonal Carryover Buffer Segment



(b) Vertical Carryover Buffer Segments

Figure 4.3: Carryover buffer segments. The relevant segments within the carryover buffers C_D, C_V and C_W for the sub matrix with the indices $g = 1$ and $h = 0$ are highlighted. The diagonal carryover buffer C_D refers to a symmetric recurrence matrix and therefore contains only $N - 1$ elements.

contrast, storing the full recurrence matrix consisting of 10^{12} elements would require ≈ 954 GiB, assuming a byte is used to encode a single binary similarity value. Representing each similarity value by a single bit, the size could further be reduced to ≈ 119 GiB. Due to their small size, all three carryover buffers can be stored simultaneously within the memory of a computing system. Access to memory layers with a higher latency, e.g., hard disk drives, can therefore be avoided.

Each sub matrix maps directly to a consecutive segment within all of the three carryover buffers (see Fig. 4.3). This allows to transfer only those segments to a compute device that are relevant for processing the current sub matrix, which further reduces the amount of data transferred. The starting point of a each carryover buffer segment depends on the values of the indices g and h. The size of each segment is determined by the extent of the corresponding sub matrix.

4.1.3 Sub Matrix Processing Order

A carryover buffer provides one element either for every global diagonal or vertical line detection task. To ensure correct global RQA results, it is required to process the sub matrices in a specific order. The dependencies between sub matrices regarding the detection of diagonal and vertical lines differ. Based on Fig. 4.2, two sub matrix processing orders are defined:

Diagonal processing order: Given an arbitrary sub matrix $S_{g,h}$ with $g \geq 0$ and $h \geq 0$. It is

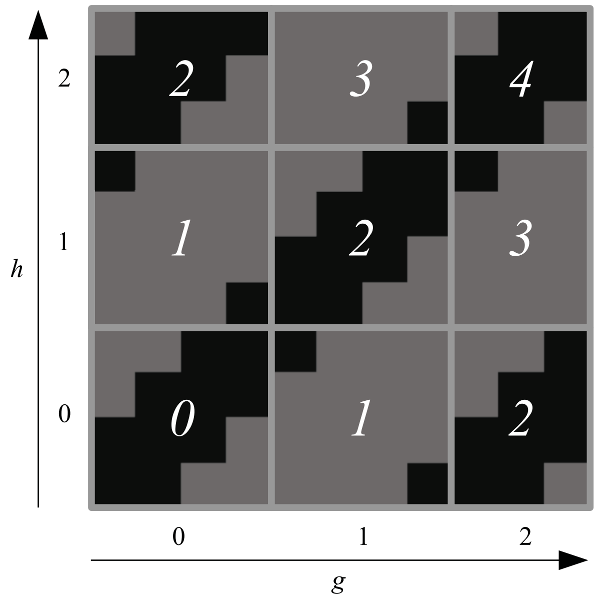


Figure 4.4: Sub matrix processing levels. The processing levels are defined based on the diagonal processing order. Each sub matrix is assigned to exactly one processing level. Each level is identified by a specific index, which increases with greater distance from the point of origin.

required to process the sub matrices $S_{g,h-1}$, $S_{g-1,h}$ and $S_{g-1,h-1}$, before processing $S_{g,h}$.

Vertical processing order: Given an arbitrary sub matrix $S_{g,h}$ with $g \geq 0$ and $h \geq 0$. It is required to process the sub matrix $S_{g,h-1}$, before processing $S_{g,h}$.

Note that sub matrices referenced by one or more negative indices do not exist. The diagonal processing order is more restrictive compared to the vertical one, by requiring the fulfilment of additional constraints. The latter also allows to detect vertical and white vertical lines correctly, when applying the diagonal processing order.

Based on the diagonal processing order, a set of *processing levels* is defined (see Fig. 4.4). Each sub matrix is assigned to a exactly one level, which is referred to by an ascending index. Each line detection subtask within a sub matrix does not have share data with another subtask of any other sub matrix belonging to the same processing level. Hence, it is guaranteed that all line detection subtasks of the same processing level are fully independent of each other.

The introduction of processing levels allows to compute multiple sub matrices belonging to the same level in parallel. This is an important requirement, since it enables to distribute sub matrices across multiple compute devices. The amount of sub matrices contained by a specific processing level determines its maximum degree of parallelism. It varies between 1 and $\lceil N/k \rceil$, depending on the specific level.

4.1.4 Diagonal Lines Symmetric Workload Balancing

As stated in Sect. 2.1.2, a recurrence matrix may be symmetric along the middle diagonal. This requires to inspect only less than one of its halves regarding diagonal lines and multiplying the content of the corresponding histogram by two. This half only a subset of sub matrices, resulting in three sub matrix types (see Fig. 4.5a):

- sub matrices that do *not* have to be inspected regarding diagonal lines,
- sub matrices that have to be inspected *partially* regarding diagonal lines, and
- sub matrices that have to be inspected *fully* regarding diagonal lines.

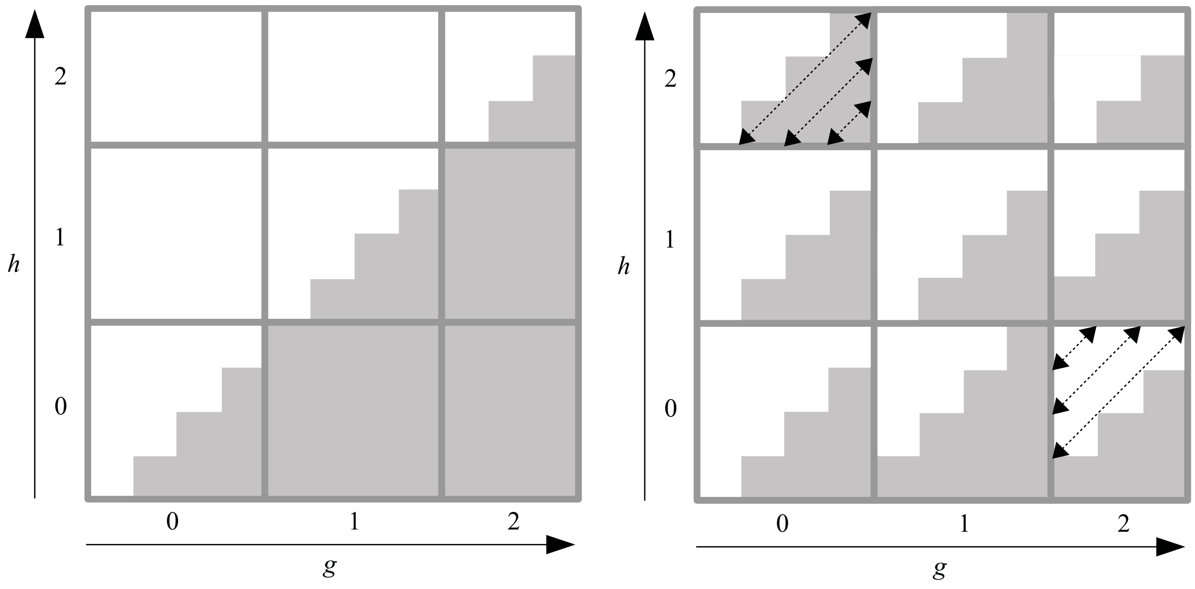
These different types lead to an unbalanced distribution of workload regarding the detection of diagonal lines, which causes drastic runtime variations between sub matrices. Those variations likely create computational bottlenecks, if the processing of sub matrices is distributed across multiple compute devices.

As an example, assume two compute devices A and B that have the same computational capabilities. Assume further that each of those devices starts to process one of two equally-sized sub matrices at the same time. Compute device A processes a sub matrix that is not inspected regarding diagonal lines, whereas compute device B processes a sub matrix that is fully inspected regarding diagonal lines. A finishes its processing, while B is still running. A has to wait for B to finish its processing, assuming that there are no other sub matrices of the current processing level left (see Sect. 4.1.3). This leaves the compute resources of A unused in the meantime. The impact of this unbalanced processing increases with increasing number of compute devices as well as increasing size of the sub matrices.

To reduce the waiting time of compute devices, *SRA* introduce a processing scheme that distributes the amount of work regarding the detection of diagonal lines more evenly across all sub matrices (see Fig. 4.5b). It exploits the property that each sub matrix that belongs to the lower half of the recurrence matrix and that contains no part of the LOI has a peer within the upper half that is mirrored along the middle diagonal. This means that parts of the global diagonals of the lower half are also present in sub matrices that belong to the upper half. The order of those parts is identical to the ordering in the lower half, which ensures the correct detection of line structures. As a result, each sub matrix may contribute to the detection of diagonal lines by inspecting its lower half. This approach also satisfies the diagonal processing order.

4.1.5 Recombination Strategy

Individual line length histograms are computed for each of sub matrix. Such a local histogram stores lengths of up to N, since lines may cross the borders of adjacent sub matrices. If such a line ends within a sub matrix, the corresponding element within the local histogram is incremented. Otherwise, the corresponding carryover buffer element stores the intermediate length. The contents of the three local histograms are merged into the global frequency distributions H_D, H_V and H_W, after having finished the processing of a particular sub matrix. The merging is synchronised across the sub matrices belonging to the same processing level.



(a) Unbalanced. (b) Balanced.

Figure 4.5: Distribution of work regarding the detection of diagonal lines in symmetric recurrence matrices. The extent of the recurrence plot displayed as well as its partitioning refers to Fig. 4.1. The contents of the plot are not displayed, for the purpose of simplification. It is assumed that the underlying recurrence matrix is symmetric. The matrix elements evaluated during the inspection of diagonal lines are highlighted with a dark overlay. In (a), the sub matrix $S_{0,2}$ is not inspected regarding diagonal lines at all, whereas in $S_{2,0}$ all matrix elements have to be evaluated. This leads to an unbalanced distribution of work between the sub matrices. Nonetheless, $S_{2,0}$ contains mirrored parts of the full diagonals contained by $S_{2,0}$, denoted as dashed arrows in (b). This property is exploited by the balanced processing scheme, where $S_{0,2}$ and $S_{2,0}$ contribute evenly to the detection of diagonal lines by inspecting both of their lower halves. Note that the LOI does not have to be inspected, because it contains a single diagonal line.

The carryover buffers C_D, C_V and C_W contain the lengths of the lines that reach the outer borders of the full recurrence matrix, after having investigated all sub matrices. These lengths have to be added to the global histograms in a post-processing step. The set of quantitative measures is computed based on the final states of the global histograms. This is the last step in the RQA processing pipeline using D&R.

4.2 Mapping to OpenCL

As mentioned earlier, an essential goal of *SRA* is to enable RQA processing on massively parallel hardware architectures. For this purpose, the OpenCL framework is employed, whose foundations have been laid out in Sect. 2.3. OpenCL distinguishes between host device and compute devices. Regarding *SRA*, the host device steers the processing of the individual sub matrices on the set of compute devices. Among others, this includes:

- the transfer of data to and from compute device memory,
- the launch of the parallel analytical tasks on the compute devices, and
- the maintenance of the global histograms and carryover buffers.

The host device is furthermore responsible for assigning sub matrices to compute devices. Each sub matrix is processed by a single compute device. Only sub matrices that belong to the same processing level are processed simultaneously. Considering a single sub matrix, the compute device executes the tasks of the *create_recurrence_matrix* as well as the two line detection operators. The kernel functions implement the analytical operations that have to be conducted by a single atomic task. Depending on the operator, a work-item corresponds either to:

- the similarity computation of a single pair of input vectors,
- the inspection of a single diagonal of a sub matrix regarding diagonal lines, or
- the inspection of a single column of a sub matrix regarding vertical lines.

The OpenCL framework is well-suited in the context of *SRA*, due to several reasons. First, *SRA* allows to leverage the parallel computing capabilities of massively parallel hardware architectures, because recurrence analysis can be separated into large number of independent tasks. Second, only those input vectors have to be transferred to a compute device that are required to process the current sub matrix. The same holds for the corresponding segments of the diagonal and vertical carryover buffers. In contrast, the sub matrices materialised consume vast parts of the compute device memory, depending on their edge lengths. Nonetheless, their storage does not require communication with the host device. The histograms of diagonal and vertical line lengths are comparatively small. They are transferred from compute device memory to the main memory of the host device, together with the updated carryover buffer segments. Subsuming, the amount of data to transfer between host and compute devices is fairly small, while the intermediate results stored in the compute device memory are comparatively large.

4.3 Execution Pipeline

The *SRA* execution pipeline building on OpenCL is depicted in Fig. 4.6. It focuses on the interaction between the host device and the compute devices. In the following, the different segments of the *SRA* execution pipeline are explained in detail, including initialisation, execution of analytical operators and final processing.

4.3.1 Initialisation

Initialisation (see step *2* to *4* in Fig. 4.6) is conducted exclusively by the program that runs on the host device. Among others, it comprises:

- the discovery of the OpenCL environment,
- the retrieval of the time series data,
- the creation of the global data structures, and
- the generation of abstract sub matrix descriptions.

At first, the available OpenCL platforms within the computing system are discovered. Having selected a specific platform, the user can further narrow the set of compute devices used during the processing. One OpenCL command queue is created per compute device selected. A compiled version of all kernel functions is created for each device. Compiling the kernel code only once during the initialisation phase reduces the computational overhead.

The host program reads time series data from its source, which serves as input for the RQA. Time series are usually stored within a single file. The length of the time series in combination with the embedding parameters are the basis for creating the global carryover buffers as well as the line length histograms. Furthermore, they are used to generate *abstract sub matrix descriptions* (see Fig. 4.7). They comprise indices referring those input vectors that represent the starting points of a sub matrix. In addition, a description contains the edge lengths regarding the X and Y dimension. The abstract descriptions are stored within an array of queues, with each queue representing a processing level. The array of queues is processed in sequential order. The descriptions within a single queue are processed in parallel. If the processing of the last description of the current queue has finished, the processing of the subsequent queue begins.

The host program creates a host thread for each compute device employed. The communication between the host device and the compute devices is realised solely via those threads. They iteratively dequeue sub matrix descriptions from the current queue. Note that all host threads access the current queue simultaneously, which requires synchronisation. Each sub matrix is processed in a single loop iteration within the host thread. It steers the execution of the analytical operators on the compute device.

4.3.2 Execution of Analytical Operators

The analytical operators applied to a single sub matrix are processed sequentially (step *5* to *19*). OpenCL also offers the opportunity to process multiple operators at the same time. This

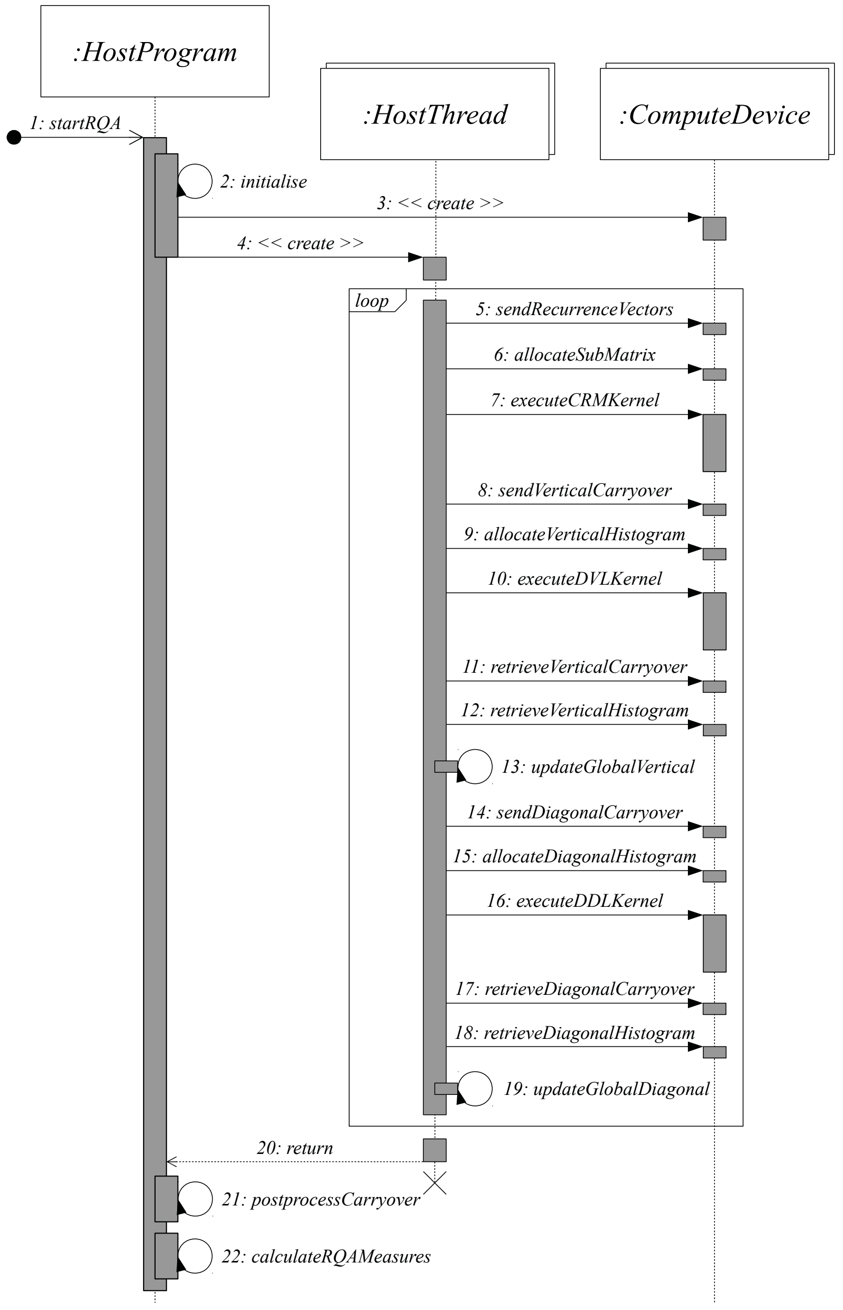


Figure 4.6: OpenCL execution pipeline. The communication between the actors is depicted in a sequence diagram according to version 2.5 of the *Unified Modeling Language* (UML) [Object Management Group, Inc, 2015]. First, the host program discovers the OpenCL devices available and creates matching threads that run on the host device. Each host thread interacts with exactly one compute device and is structured as a loop. It iteratively obtains the next sub matrix description from the queue and steers the execution of the analytical operators on the corresponding compute device. Finally, the host program post-processes the diagonal and vertical carryover buffers and computes the RQA measures, after all sub matrices are processed. Note that the acronyms CRM, DVL and DDL refer to the creation of the recurrence matrix, the detection of vertical lines and the detection of diagonal lines.

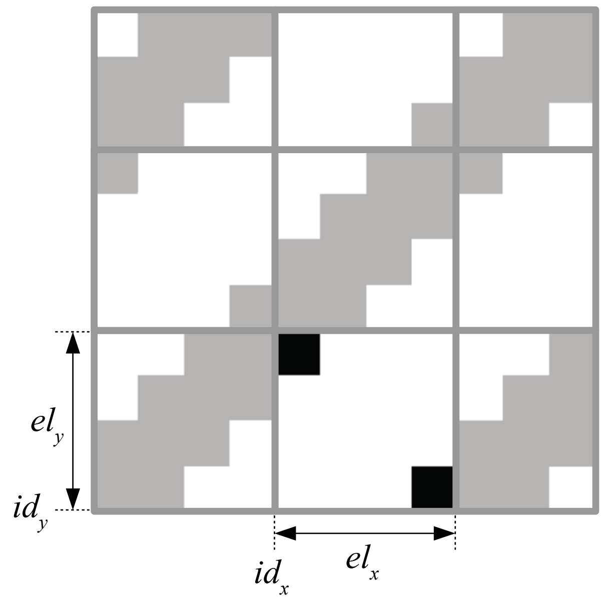


Figure 4.7: Abstract sub matrix description. Each sub matrix description comprises the indices of the input vectors representing its starting points (id_x and id_y) and its edge lengths regarding each of the dimensions X and Y (el_x and el_y).

is achieved by interleaving the execution of the set of work-groups belonging to different kernel instances. Interleaved processing changes the order but not the amount of work conducted. More severe, it hampers valid time measurements. Profiling events do not capture the cumulative time of processing all related work-groups. They rather deliver the time interval between starting to process the first work-group and having finished the processing of the last one of a kernel instance (see Fig 4.8).

The process steps *5* to *19* are executed for each sub matrix. It comprise the execution of the three analytical operators presented in Sect. 3.1. The execution of the host threads terminates, if those operators have been applied to all sub matrices (step *20*).

Create Recurrence Matrix

The execution of the *create_recurrence_matrix* operator (step *5* to *7*) is a precondition for executing the vertical and diagonal line detection tasks. The host thread allocates regions within the global memory of the compute device regarding the storage of the relevant input vectors and their pairwise similarities. The host thread transfers the recurrence vectors from host to compute device memory. The allocated region representing the sub matrix may be initialised on demand, e.g., with zeros.

The kernel instance computing the binary similarities is executed, after the initialisation of the compute device memory is completed. A work-item is created for each of the pairwise similarity comparisons. Work-items belonging to the same work-group are executed in parallel. The work-group size is specified by the OpenCL runtime, determining the degree of parallelism.

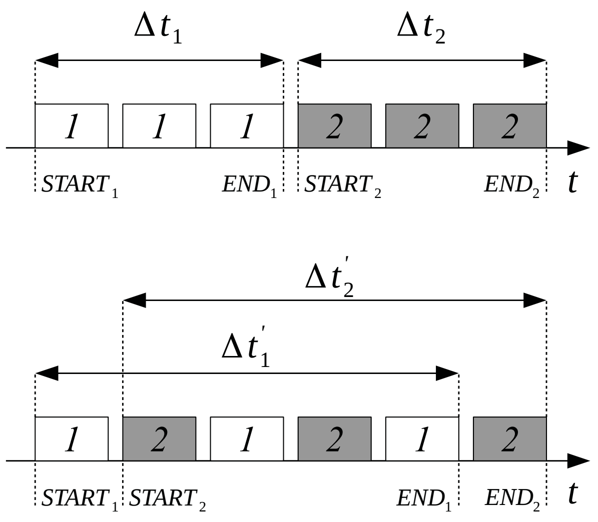


Figure 4.8: Sequential vs. interleaved processing of work-groups. The execution times of sets of work-groups belonging to the two kernel instances *1* and *2*, with three groups per instance, is depicted. The top of the figure displays the sequential processing, whereas the bottom displays the interleaved processing. The total execution time is equal in both cases. Note that the time delta between starting the first work-group and finishing the last work-group is larger regarding the interleaved execution: $\Delta t_1' > \Delta t_1$ and $\Delta t_2' > \Delta t_2$.

The similarity values are written to the matching cells in the global memory of the compute device.

The creation of the sub matrix does not require to perform any post-processing after the execution of the operator is completed. The reference to the memory region storing the binary similarity values is kept by the host thread. It is passed as a parameter to the *detect_vertical_lines* and *detect_diagonal_lines* operator.

Detect Vertical Lines

Executing the *detect_vertical_lines* operator (step *8* to *13*) inspects the binary similarity values regarding vertical and white vertical line structures. The host thread allocates space within the global memory of the compute device for the line length histograms and the relevant carryover buffer segments. Note that vertical and white vertical lines are detected within the same kernel instance, since both rely on inspecting columns of the sub matrix. As a result, vertical and white vertical lines within a column are detected using a single work-item.

Initially, the host thread copies the carryover buffer segments belonging the current sub matrix from the host memory to the global memory of the compute device. The values stored in those segments are used as input for detecting vertical and white vertical lines. The local histograms of vertical and white vertical lines are initialised with zeros before the detection of lines starts.

The kernel instance encapsulating the vertical line detection tasks is executed, after the initialisation of the carryover buffer segments and the histograms is completed. Each work-item inspects the elements of a single column of the sub matrix sequentially. The amount of occurrences of each line length is updated concurrently. The access to the histograms is synchronised, since multiple columns are inspected at the same time. The access to the carryover buffer elements does not have to be synchronised, since each carryover buffer element belongs exclusively to a single work-item. Similar to the *create_recurrence_matrix* operator, the amount of work-items executed in parallel is determined by the OpenCL runtime.

The execution of the *detect_vertical_lines* operator is finished, if the execution of its last work-item is finished. Afterwards, the host thread copies the modified carryover buffer segments from the global memory of the compute device to the host memory. The local modifications are used to update the global carryover buffers. Note that this access does not need to be synchronised, since only sub matrices belonging to the same processing level are processed simultaneously.

Following, the host thread retrieves the filled histograms of vertical and white vertical line lengths from compute device memory. The content is added to their global counterparts. This access has to be synchronised, since multiple host threads may perform an histogram update simultaneously. This restriction can be avoided by keeping separate histograms for each compute device employed. The latter requires to merge the device-specific into global histograms in a post-processing step.

Detect Diagonal Lines

The procedure of executing the *detect_diagonal_lines* operator (step *14* to *19*) is similar to the detection of vertical lines. This includes the allocation and initialisation of memory regions for segments of the diagonal carryover buffer and the local diagonal line length histogram. A work-item referring to the kernel instance encapsulating the detection of diagonal lines inspects a single diagonal of the sub matrix. Multiple diagonals are inspected simultaneously, likewise to the detection of vertical lines. The corresponding degree of parallelism also depends on the OpenCL runtime. The synchronisation constraints regarding the update of the local and global data structures are identical to the *detect_vertical_lines* operator.

4.3.3 Final Processing

After all sub matrices have been processed, the carryover buffers contain the lengths of the lines that reach the outer borders of the full recurrence matrix. Those line lengths have to be captured in the global histograms, to ensure valid RQA results. For this purpose, the final content of the carryover buffers is used to update the histograms in a post-processing step (step *21*). This is conducted by the host program, after all host threads have terminated. Based on the final state of the vertical, white vertical and diagonal histogram, the RQA measures as presented in Sect. 2.1.3 are calculated by the host program (step *22*). The host program terminates, after all measures have been computed.

5 Analytical Operator Variants

Chapter 4 describes *scalable recurrence analysis*, a novel computing approach to recurrence analysis based on the concept divide and recombine. Whereas the previous chapter focussed on steering the processing of a set of sub matrices, this chapter addresses the computations performed to analyse the contents of a single sub matrix. Chapter 3 introduced several computing approaches to improve the performance of RQA computations, in particular:

1. parallel brute-force processing, and

2. index data structures.

The index-based approaches are only employed regarding the construction of the binary similarity matrices, whereas the parallel brute-force processing is applied to all analytical operators.

Given an arbitrary algorithm, it is usually assumed that the performance characteristics of a corresponding implementation are optimised. This claim can not be made in the context of heterogeneous computing. OpenCL provides *functional portability* but does not guarantee *performance portability.* Identical OpenCL kernels can be compiled and executed on different hardware architectures. Compute devices may adhere to varying execution models and likely have different parallel computing capabilities. Hence, an implementation most probably has varying performance characteristics on different computing platforms. This trade-off has been widely acknowledged in the scientific literature [Rul et al., 2010, Pennycook et al., 2013, Zhang et al., 2013]. A first study reflecting the performance variations of several parallel brute-force RQA implementations using OpenCL on different hardware platforms has been presented in [Rawald et al., 2015].

The following two sections describe the design space of implementing RQA using parallel brute-force processing (see Sect. 5.1) and index data structures (see Sect. 5.2). The design dimensions considered, such as input data format, are heavily based on research in the field of database technology. A set of concrete realisations is investigated for each of dimension. Here, the focus lies especially on describing the concepts instead of examining concrete source code. Note that the set of dimensions is not claimed to be complete. It rather serves as an entry point for additional design space explorations.

Again, there is no claim of developing a single best-performing implementation. The goal is rather to come up with a pool of implementations having different characteristics regarding the execution of the analytical RQA operators. The performance of those implementations is evaluated in Chap. 6, employing different analytical workloads and hardware architectures.

5.1 Parallel Brute-Force Processing

Parallel brute-force processing means that all pairwise inout vector similarities are computed, which are captured in a sub matrix. The inspection of the binary similarity matrix regarding diagonal and vertical line structures is also conducted in a parallel fashion. The design dimensions considered in this section are:

- input data format,
- recurrence matrix representation,
- similarity value representation,
- intermediate results recycling, and
- recurrence matrix materialisation.

Specific realisations regarding each of those dimensions are investigated in detail, including their impact to the RQA processing. It is stressed that a major contribution of this thesis is the adaption of those concepts to the requirements of recurrence analysis.

5.1.1 Input Data Format

Time series are the basis for recurrence analysis. The following descriptions assume that time series data resides in the main memory of a computing system. A given time series is embedded into multi-dimensional space with an dimensionality of m, leading to the reconstruction of N input vectors. Note that the embedding dimension is equal for all vectors.

Memory Storage Formats

A common technique to capture a set of vectors, where each vector component refers to specific variable, is the tabular structure. Regarding recurrence analysis, a table consists of N rows and m columns. Each row refers to the data of a single input vector, whereas each column represents a specific component of the m-dimensional space. Table 5.1 presents a concrete example.

There exist two common formats to store tabular data in memory (see Fig. 5.1):

Row-wise: All data values belonging to a single row are stored consecutively in memory. The individual rows are stored sequentially (see Fig. 5.1a).

Column-wise: All data values belonging to a single column are stored consecutively in memory. The individual columns are stored sequentially (see Fig. 5.1b).

Each storage format has proven its applicability in different scenarios. The row-wise storage format adheres to transactional use cases, accessing all data related to a single instance of an entity at once. This type of workload is dominated by a high number of write operations. The column-wise storage format has been designed to improve the responsiveness of analytical workloads, such as aggregations over a single column a database table [Stonebraker et al., 2005].

Table 5.1: Input vector components. The eleven input vectors from Fig. 2.1 are captured in a table. Each vector consists of two components. The values regarding each component refer to a consecutive segment within the input time series. The values with dark hightlights refer to an identical sequence of values in the two component columns.

Input Vector	1^{st} *Component*	2^{nd} *Component*
$\vec{x}_1$	0.0	1.0
$\vec{x}_2$	0.7	0.7
$\vec{x}_3$	1.0	0.0
$\vec{x}_4$	0.7	-0.7
$\vec{x}_5$	0.0	-1.0
$\vec{x}_6$	-0.7	-0.7
$\vec{x}_7$	-1.0	0.0
$\vec{x}_8$	-0.7	0.7
$\vec{x}_9$	0.0	1.0
$\vec{x}_{10}$	0.7	0.7
$\vec{x}_{11}$	1.0	0.0

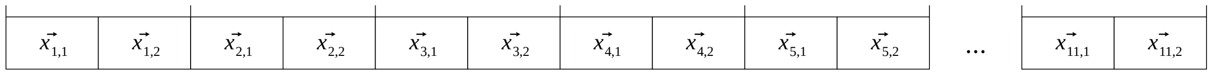
These workloads focus on the inspection of single columns and require a high amount of read accesses.

Several database management systems supporting column-wise storage have emerged in the recent past. This includes disk-based systems such as *C-Store* [Stonebraker et al., 2005] and in-memory databases such as *SAP HANA* [Färber et al., 2012]. The relevance of column-wise storage formats is furthermore illustrated by the *Apache Parquet* project [Kestelyn, 2013].

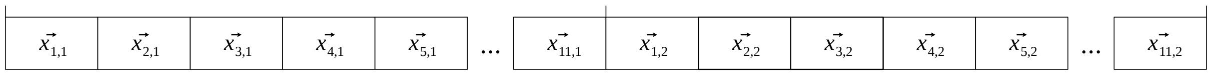
Regarding RQA, both storage formats presented require to transform the input time series, which is represented by an one-dimensional array. Such transformations are attached with specific computational costs. Data values have to be read either one or multiple times and written to a different memory location to implement a particular storage format. In addition, the amount of memory occupied by the row-wise and column-wise format is $m*N$ data elements, with m being the dimensionality and N being the number of input vectors.

The transformation costs as well as the increase in memory consumption can be eliminated by leveraging the properties of input vector reconstruction using the time delay method (see Sect. 2.1.1). Vector components refer to data values from the input time series. A single data value is used as a component of up to m input vectors, leading to duplicate storage. This thesis introduces the *overlapped column-wise* memory storage format to reduce the memory footprint (see Fig. 5.1c).

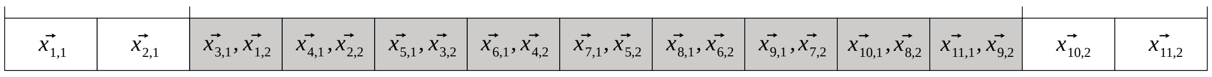
Input vectors are compounds of m values referring to succeeding points in time with a temporal offset of t. This property enables to store the multi-dimensional vectors in columnar fashion without having to perform any transformation. A sub series of the input time series is equivalent to two segments the component columns, as demonstrated in Tab. 5.1. As pre-



(a) Row-wise. The data values belonging to a single row are stored consecutively. The rows are stored in ascending order of the input vector index.



(b) Column-wise. The data values belonging to a single column are stored consecutively. The columns are stored in ascending order of the component index.



$\vec{x}_{1,1}$
$\vec{x}_{2,1}$
$\vec{x}_{3,1}, \vec{x}_{1,2}$
$\vec{x}_{4,1}, \vec{x}_{2,2}$
$\vec{x}_{5,1}, \vec{x}_{3,2}$
$\vec{x}_{6,1}, \vec{x}_{4,2}$
$\vec{x}_{7,1}, \vec{x}_{5,2}$
$\vec{x}_{8,1}, \vec{x}_{6,2}$
$\vec{x}_{9,1}, \vec{x}_{7,2}$
$\vec{x}_{10,1}, \vec{x}_{8,2}$
$\vec{x}_{11,1}, \vec{x}_{9,2}$
$\vec{x}_{10,2}$
$\vec{x}_{11,2}$

(c) Overlapped column-wise. Segments of the input time series map directly to segments of the component columns, assuming that the input vectors are extracted using the time delay method. Those columns are stored virtually in overlapping fashion. The dark highlights indicate the overlapping segments, similar to Tab. 5.1. In this concrete example, single data values are part of two input vectors.

Figure 5.1: Memory storage formats. The figure refers to the tabular structure presented in Tab. 5.1. Note that the column that stores the input vectors indices is omitted. A data value is referenced by $\vec{x}_{v,c}$, with v referring to the input vector index and c to the component index.

sented in Fig. 5.1c, the columns are already stored virtually in an overlapped manner when using the time series representation. This overlapped column-wise storage format consumes the same amount of memory as the input time series, since its original structure is not modified. There are no additional costs attached to implement this storage format, due to the absence of transformations.

In addition, the overlapped column-wise storage format complements well with the division of the recurrence matrix into sub matrices. Segments of the input time series map directly to the edges of sub matrices, which simplifies data extraction. In the following, the overlapped column-wise storage format is preferred over the column-wise storage format, due to its advantages in the context of recurrence analysis.

Discussion

The column-wise storage format is favoured regarding data access by several parallel hardware architectures. The labelling of the column-wise storage format varies between the different vendors:

AMD: *Coalesced access patterns* [Advanced Micro Devices, Inc., 2013b, pp. 6–16]

Intel: *Structure of arrays* (SOA) [Intel Corporation, 2011, p. 11]

Nvidia: *Coalesced memory access* or *non-strided memory access* [NVIDIA Corporation, 2009b, pp. 13–18]

Reasons for favouring the column-wise storage format stem from the parallel execution and data acquisition model. The execution model usually adheres to SIMD or SIMT, which perform the same operation of different data elements at the same time. Among others, this affects reading data from the memory of the compute device during the execution of the *create_recurrence_matrix* operator in the context of RQA. The impact of choosing a particular storage format on the memory access is depicted in List. 5.1 and List. 5.2 as OpenCL C code. Note that the input vectors are stored using a different format in each case. Furthermore, *t* refers to the time delay parameter.

Listing 5.1: Row-wise Data Access.

```
for (uint i = 0; i < m; ++i)
{
    input_vectors[(vector_id * m) + i];
}
```

Listing 5.2: Overlapped Column-wise Data Access.

```
for (uint i = 0; i < m; ++i)
{
    input_vectors[vector_id + (i * t)];
}
```

The code in both listings iterates over the *m* components of a input vector that is identified by a specific index. Multiple work-items access the same component of multiple vectors with ascending indices simultaneously, while performing the same loop iteration. The corresponding data elements are transferred from the global memory of the compute device to its *L1* cache, using *cache lines*.

A cache line contains a consecutive segment of data from global memory. To minimise the amount of cache lines transferred, the number of values usable by the work-items currently executing the read operation has to the maximised. This portion of usable values is referred to as *cache line saturation* (see Fig. 5.2). Optimising this saturation has been found to be a key driver regarding the performance of OpenCL kernels.

5.1.2 Recurrence Matrix Representation

The previous considerations of this thesis assume that sub matrices are stored in the global memory of the compute devices. Those sub matrices can be represented in different ways. A property to distinguish the suitability of a particular representation is the filling rate of a sub matrix. In general, there is the distinction between *dense* and *sparse* matrices. A dense matrix contains a high amount of non-zero values, referred to as recurrence points in the context of recurrence analysis. Contrastingly, a sparse matrix contains a low amount of non-zero values. Representations for both types of matrices are explained in the following. Dense matrices are commonly encoded using an uncompressed representation, while sparse matrices offer the potential to apply compression.

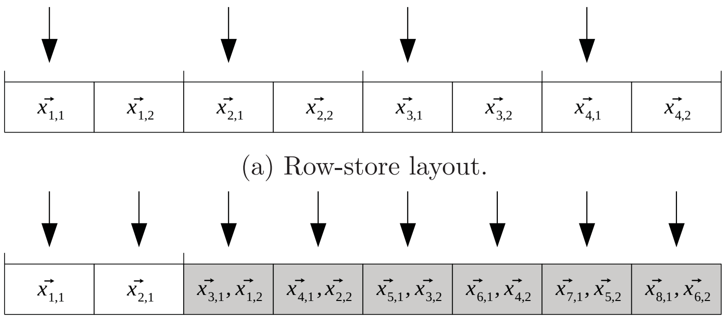

$\vec{x}_{1,1}$
$\vec{x}_{1,2}$
$\vec{x}_{2,1}$
$\vec{x}_{2,2}$
$\vec{x}_{3,1}$
$\vec{x}_{3,2}$
$\vec{x}_{4,1}$
$\vec{x}_{4,2}$
(a) Row-store layout.
$\vec{x}_{1,1}$
$\vec{x}_{2,1}$
$\vec{x}_{3,1}, \vec{x}_{1,2}$
$\vec{x}_{4,1}, \vec{x}_{2,2}$
$\vec{x}_{5,1}, \vec{x}_{3,2}$
$\vec{x}_{6,1}, \vec{x}_{4,2}$
$\vec{x}_{7,1}, \vec{x}_{5,2}$
$\vec{x}_{8,1}, \vec{x}_{6,2}$
(b) Overlapped column-store layout.

Figure 5.2: Cache line saturation. The example refers to the storage formats presented in Fig. 5.1. The saturation regarding the row-store and overlapped column-store layout is compared to each other. A cache line with a size of eight data elements is applied for the purpose of demonstration. The usable data elements regarding the processing of the first vector component are indicated by vertical arrows. The overlapped column-store layout delivers a saturation of 100%, whereas the row-store layout delivers a saturation of only 50%.

Uncompressed Matrix Representation

The general approach to represent dense recurrence matrices is to store the value of every matrix element explicitly. Each recurrence and non-recurrence point is represented by a one or a zero. The offset for accessing a particular matrix element in the global memory of the compute device can be directly inferred from its X and Y indices. Similar to the representation of input data, a row-wise or column-wise storage format can be applied. The first one stores all values of a single row of the matrix consecutively. The latter stores all values of a single column of the matrix consecutively.

The importance of coalesced memory access regarding the storage of the input data has been highlighted in Sect. 5.1.1. The same holds for the representation of the recurrence matrix. A matrix element is referenced by a two-dimensional index. By convention, the X coordinate is the first part of the index, whereas the Y coordinate is the second part. The OpenCL runtime iterates over the first part of the index before iterating over the second part [Intel Corporation, 2011, p. 29]. Data elements accessed in successive iterations therefore have to be stored consecutively to ensure coalesced memory access. As a result, the row-wise storage format is advantageous, as presented in Fig. 5.3.

Compressed Matrix Representation

Compressed matrix representations assume that a high proportion of matrix elements is zero. The amount of memory consumed by the matrix is reduced, by storing only information about non-zero matrix elements. The *coordinate format* [Saad, 2003, p. 92] is a simple representation to compress sparse matrices. It stores the X and Y coordinate as well as the value of each non-zero matrix element. A single matrix element is therefore represented by three numbers.

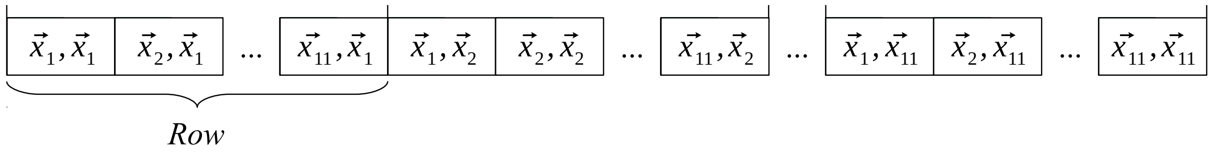


Figure 5.3: Uncompressed representation of the recurrence matrix. The recurrence vectors $\vec{x}_1$ to $\vec{x}_{11}$ depicted in Fig. 2.1 serve as an example. Each matrix cell corresponds to a pair of the eleven input vectors, where the first vector refers to its X and the second to the Y coordinate. The recurrence matrix is represented using a row-wise storage format to ensure coalesced memory access.

The size of the datatype to represent the position of a non-zero element depends on the extent of the recurrence matrix. The size of the datatype to store its value depends on the range of values.

Compressed Sparse Formats. The *compressed sparse row* (CSR) and the *compressed sparse column* (CSC) format are extensions of the coordinate format [Saad, 2003, pp. 92–93]. Both formats comprise three data structures. Each is represented by an one-dimensional array.

AA: stores the values of the non-zero matrix elements.

JA: stores the column indices (CSR) or row indices (CSC) of all non-zero matrix elements.

IA: stores the start index of each row (CSR) or column (CSC) within the AA and JA array.

The AA array is omitted in the context of recurrence matrices, since each non-zero matrix element has the value one. The IA array contains $N+1$ values, independent of the sparseness of the recurrence matrix, one value regarding the start index of each row (CSR) or column (CSR) and a boundary value at the end. The amount of values stored in the JA array depends on the sparseness of each row (CSR) or column (CSC).

Detection of Vertical Line Structures using the CSC Storage Format. Changing the in-memory storage format for representing the recurrence matrix requires to modify the algorithms for detecting vertical and diagonal line structures. The algorithms presented in Sect. 2.2.1 assume that an uncompressed matrix representation is applied. Note that the corresponding modifications are an important contribution of this thesis to the research field of recurrence analysis computing.

The CSC storage format is favourable regarding the sequential detection of vertical line structures, since indices of recurrence points within one column are stored consecutively. They are represented by a consecutive segment within the JA array. Note that in the following it is assumed that the row indices within this segment are in ascending order. Two successive row indices within this segment are part of the same vertical line, if they have a difference of one. They frame a white vertical line, if the difference is larger than one. The boundaries of each

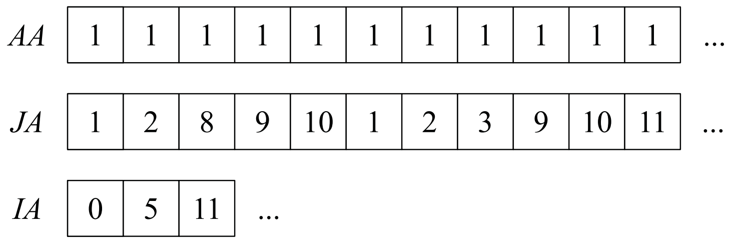


Figure 5.4: Compressed sparse matrix. The compressed matrix refers to the recurrence plot from Fig. 2.2. Its representations using the CSR and CSC format are identical, since the underlying matrix is symmetric. Only the first two rows (CSR) or columns (CSC) are displayed. All elements of the AA array have the value 1, since the recurrence matrix stores binary information. The JA array stores the original vector indices, starting at one. The IA array stores the start indices of the rows (CSR) or columns (CSC), starting at zero.

column are captured in the IA array. The algorithm for detecting vertical lines within the CSC storage format is depicted in Alg. 6, which is an adaption of Alg. 3.

The arrays JA and IA as well as the total number of input vectors N is passed as parameters to the function DetectVerticalLinesCSC. The algorithm contains a single nested loop. The outer loop iterates over all columns of the recurrence matrix. The inner loop iterates over the row indices of the current column that are stored within the consecutive segment of JA. The boundaries of the column are stored in the IA array. The variable lst refers to the last row index evaluated. If the distance between the last and the current row index is equal to one, the current line length is incremented by one. If not, the current line length is used to update the corresponding histogram element.

Algorithm 7 depicts the detection of white vertical lines using the CSC format. The parameters of the function DetectWhiteVerticalLinesCSC are identical to the function for detecting vertical lines. Algorithm 4 is adapted to inferring the positions of non-recurrence points implicitly. The first row index of a column segment corresponds to the length of the first white vertical line. The variable flg indicates, whether it has already been evaluated. The ordering of the inner and outer loop as well as the semantics of the variable lst are equal to Alg. 6.

Algorithm 6 and 7 both evaluate each element of the matrix only once, resulting in a time complexity of $\mathcal{O}(N^2)$. They furthermore do only require to maintain an histogram of size N, leading to a linear space complexity depending on the amount of input vectors N. This can be likewise mapped to the concept of sub matrices.

Detection of Diagonal Line Structures using the CSC Storage Format. The uncompressed matrix storage formats guarantee that each matrix element can be accessed with a fixed offset based on its coordinates. The CSC storage format does not adhere to this property. The information, whether a specific row index is contained within a column can only be obtained by inspecting the specific segment in the JA array.

Algorithm 6 Detect vertical lines using the CSC format.

```
1: function DetectVerticalLinesCSC($JA, IA, N$)
2:     $H_V \leftarrow$ Zeros1DArray($N$)
3:     for $i \leftarrow 1$ to $N$ do
4:         $lst \leftarrow 0$
5:         $l \leftarrow 0$
6:         for $j \leftarrow IA(i)$ to $IA(i+1)$ do
7:             if $JA(j) - lst = 1$ then
8:                 $l \leftarrow l + 1$
9:             else
10:                if $l \geq 2$ then
11:                    $H_V(l) \leftarrow H_V(l) + 1$
12:                end if
13:                $l \leftarrow 1$
14:            end if
15:            $lst \leftarrow JA(j)$
16:        end for
17:    end for
18:    return $H_V$
19: end function
```

Algorithm 7 Detect white vertical lines using the CSC format.

```
1: function DetectWhiteVerticalLinesCSC($JA, IA, N$)
2:     $H_W \leftarrow$ Zeros1DArray($N$)
3:     for $i \leftarrow 1$ to $N$ do
4:         $flg \leftarrow 0$
5:         $lst \leftarrow 0$
6:         $l \leftarrow 0$
7:         for $j \leftarrow IA(i)$ to $IA(i+1)$ do
8:             if $flg = 0$ then
9:                 $l \leftarrow JA(j)$
10:                $flg \leftarrow 1$
11:            else
12:                $l \leftarrow JA(j) - lst - 1$
13:            end if
14:            if $l \geq 2$ then
15:                $H_W(l) \leftarrow H_W(l) + 1$
16:            end if
17:            $lst \leftarrow JA(j)$
18:        end for
19:    end for
20:    return $H_W$
21: end function
```

Each diagonal of the recurrence matrix can be identified by an index, which is computed as the difference between the X and the Y coordinate of its origin. A recurrence point belongs to a diagonal, if the difference between its X and Y coordinate is equal to the diagonal index. This calculation can be applied to symmetric as well as to non-symmetric recurrence matrices.

An approach for detecting diagonal lines using the CSC storage format based on diagonal indices is presented in Alg. 8, which assumes a symmetric recurrence matrix. It can be adapted to non-symmetric matrices with limited effort. Algorithm 8 contains a nested loop, where the outer loop iterates over all columns of the recurrence matrix. The inner loop iterates over the row indices of its recurrence points, computing the individual diagonal index. For each diagonal index d, the difference between the row index of the current and the previous recurrence point is calculated. The current diagonal line length as captured in L is incremented, if the difference is equal to one. If not, the histogram of diagonal line lengths is updated. The array LST stores the row index evaluated at last regarding each diagonal index. c refers to the Theiler corrector (see Sect. 2.1.3).

Algorithm 8 evaluate each recurrence point stored exactly once. At most N^2 recurrence points have to be evaluated, resulting in a quadratic time complexity. The number of recurrence points actually evaluated may be smaller, depending on the sparseness of the recurrence matrix. The algorithm further has a space complexity of $\mathcal{O}(N)$, because only three arrays of size N have to be maintained.

The algorithm iterates over the columns in ascending order, which ensures valid line detection results. This property is likewise applied to the OpenCL processing. A loop evaluates all columns of the current sub matrix in ascending order. An OpenCL work-item is created for each recurrence point stored in the current column. Those work-items can be executed in parallel since they belong to different diagonals of the recurrence matrix. Note that this procedure requires extended communication between the host and the compute devices.

Relevant Carryover Buffers for CSC Storage Format. The CSC storage format has influence on the usage of the carryover buffers in the context of SRA. The vertical line length carryover buffer C_v is complemented by a *vertical index carryover buffer* C_{vi} that stores the row index evaluated at last for each of the N columns of the recurrence matrix. This is required, because a vertical sequence of recurrence points may or may not continue in an adjacent sub matrix. If it does not continue, the corresponding value of C_v is used to update the vertical line length histogram, during the inspection of the adjacent sub matrix.

The detection of white vertical lines does not require to utilise a line length carryover buffer. The length of white vertical lines crossing the borders of adjacent sub matrices can be directly inferred from the row indices of the recurrence points stored in JA. This information is shared across adjacent sub matrices using a *white vertical index carryover buffer* C_{wi}, which works similar to C_{vi}. An additional *white vertical flag carryover buffer* C_{wf} captures, whether the first row index of a column of the full recurrence matrix has already been evaluated. An element of this carryover buffer corresponds to the variable flg in Alg. 7. C_{wf} could also be omitted by initialising each element of C_{wi} with minus one. This would require to use a signed data type, which limits the maximum size of the sub matrices that can be processed.

Two carryover buffers are employed for detecting diagonal lines using the CSC format; the

Algorithm 8 Detect diagonal lines using the CSC format.

```
function DetectDiagonalLinesCSC($JA, IA, N, c$)
    $H_D \leftarrow$ Zeros1DArray($N$)
    $L \leftarrow$ Zeros1DArray($N$)
    $LST \leftarrow$ Zeros1DArray($N$)
    for $x \leftarrow 1$ to $N$ do
        for $j \leftarrow IA(x)$ to $IA(x+1)$ do
            $y \leftarrow JA(j)$
            $d \leftarrow x - y$
            if $d \geq c$ then
                if $y - LST(d) = 1$ then
                    $L(d) \leftarrow L(d) + 1$
                else
                    if $L(d) \geq 2$ then
                        $H_D(L(d)) \leftarrow H_D(L(d)) + 1$
                    end if
                    $L(d) \leftarrow 1$
                end if
                $LST(l) \leftarrow JA(j)$
            end if
        end for
    end for
    return $H_D$
end function
```

diagonal line length carryover buffer C_d and the *diagonal index carryover buffer* C_{di}. The semantics of C_d corresponds to C_v and the semantics of C_{di} to C_{vi}.

Discussion

The memory consumption of the CSR and CSC format is identical. The memory consumption compared to using the uncompressed matrix representation depends on the amount of non-zero values within the recurrence matrix. Up to a filling rate of $N^2 - (N + 1)$ recurrence points, a compressed sparse format requires less or an equal amount of data elements to represent a recurrence matrix. Above that threshold, the CSR and CSC format require more data elements, due to the fixed size of the IA array. The memory consumption is reduced under the assumption that the data type for representing similarity values within the uncompressed matrix representation is equal to the data type used for representing entries in the IA and JA array. Note that the size of uncompressed matrix representation can further be reduced by applying *bit compression* (see Sect. 5.1.3).

The CSC storage format allows a straight forward implementation of the detection of vertical line structures. Executing the algorithms in a parallel fashion ensures coalesced memory access while retrieving the column boundaries from the IA array. On the contrary, the access to the JA array is non-coalesced. An OpenCL work-item inspects a single column of a sub matrix, which requires to retrieve data referring to different columns simultaneously. The CSC storage format prevents to read consecutive segments from global memory that contain row indices from different columns, unless the columns are rather short or the column densities are low.

The algorithm for detecting diagonal lines using the CSC storage format can be adapted with little effort to the CSR storage format. Instead of iterating over the columns of a sub matrix, the iteration is performed regarding the rows. This property is independent of whether the full recurrence matrix is symmetric or not.

5.1.3 Similarity Value Representation

The previous section introduced different storage formats to represent recurrence matrices as well as the modifications regarding the algorithms to detect vertical and diagonal lines. This section considers the representation of a single similarity value.

Using a compressed sparse matrix representation (see Sect. 5.1.2), e.g., the CSC storage format, only those elements of the recurrence matrix are captured that refer to recurrence points. A recurrence point is represented by a combination of an entry within the IA and and entry within the JA array. The data type chosen to represent those two entries depends of the number of rows and columns of the matrix captured. For example, a recurrence matrix consisting of $2^{32} \times 2^{32}$ matrix elements requires to use an unsigned 32-bit data type.

Using an uncompressed matrix representation, each recurrence matrix element is captured explicitly, independent of its concrete value. In the following, this thesis considers two options to represent an element of an uncompressed matrix representation:

- a number of k bytes, and
- a single bit.

Modern computer systems provide memory that is byte-addressable, which means that the size of each standard data type is a multiple of eight bits. Hence, each recurrence matrix element consumes at least a single byte, when using such data types. Nonetheless, a recurrence matrix contains only binary information, allowing to reduce the memory consumption of a similarity value to a single bit.

Bit Compression

Bitmaps [Roth and Horn, 1993], a method from data compression, uses zero-bits to represent null data, whereas non-null data is captured by one-bits. In the case of recurrence analysis, the only non-null data that has to be stored are recurrence points. It is therefore not required to store the concrete values of each non-null data element. The size of a recurrence matrix, where each matrix element is represented using a single byte, can be reduced up to a factor of at most one eighth by applying such a bit compression approach.

A bit-compressed recurrence matrix is represented by a sequence of byte-aligned memory objects. The indiviudal bits, either assigned one or zero, within those objects refer to specific matrix elements. A *bit mask*, created using bit-shift operations, is applied to modify the value of a specific matrix element. The access to the memory object has to be synchronised using *atomic operations*, when calculating the values of multiple matrix elements in parallel. However, OpenCL does only support atomic operations using memory objects with a size of at least 32 bits [Khronos OpenCL Working Group, 2011, pp. 234–236].

Custom Bit-wise Storage Format. As explained in Sect. 5.1.1, there is a distinction between the row-wise and column-wise storage format. Considering the row-wise format, all bits referring to the same row are stored consecutively. This hampers reading values of matrix elements belonging to same column but to different rows at the same time using a single cache line. Considering the column-wise format, all bits referring to the same column are stored consecutively. Vice versa, this hampers reading values of matrix elements belonging to the same row but to different columns simultaneously. Therefore, this thesis proposes a hybrid solution combining both storage formats. This allows to exploit their individual advantages while at the same time mitigating their disadvantages.

The *custom bit-wise storage format* is depicted in Fig. 5.5. It consists of a two-level structure. Each 32-bit memory object stores similarity values that belong to 32 consecutive rows of a single column. The 32-bit values are stored in ascending order of their column index. This sequence of 32-bit memory objects is repeated until all row values have been assigned. The index of the 32-bit memory object id_{mem}, in which the bit of a matrix element is located, is computed as follows.

$$id_{mem} = \lfloor id_y/32 \rfloor + id_x. \tag{5.1}$$

id_x and id_y refer to the X and Y coordinate of the matrix element. The bit mask that is added to the memory object to mark a recurrence point is created by using a bit shift operation.

$$1 \ll (id_y \bmod 32). \tag{5.2}$$

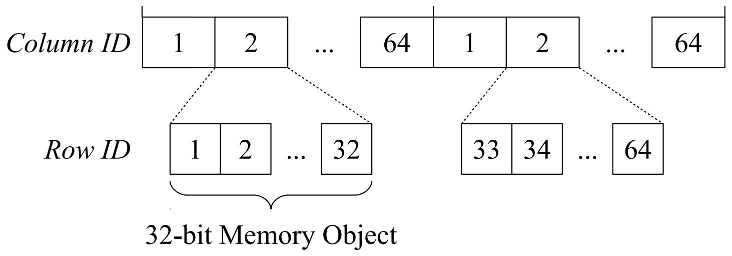


Figure 5.5: Custom bit-wise storage format. The example refers to a recurrence matrix containing 64×64 elements. Each memory object stores the similarity values of 32 matrix elements, resulting in a total of 128 memory objects. Each of those memory objects refers to a specific column of the recurrence matrix. Each bit stored within a memory object refers to a specific row within the corresponding column.

Discussion

The main benefit of bit compression is the reduction of memory consumption when applying an uncompressed matrix representation. Compared to using data types with a size of k bytes to represent a single similarity, the memory occupied can be reduced at most by a factor of $8k$. Nonetheless, bit compression is attached with computational overhead that might increases the overall runtime, e.g., the creation of bit-masks while updating and reading the values of recurrence matrix elements.

The synchronised access to memory objects using atomic operations leads to work-items waiting for the acquisition of locks. The impact of synchronisation can be mitigated by null-initialising the memory region in which the recurrence matrix or its sub matrices are stored. As a result, only those bits need to be modified that correspond to recurrence points. The OpenCL specification does not enforce that the memory allocated on compute devices is initialised with a certain value. Therefore, this thesis proposes to execute an additional kernel that performs zero-initialisation. Here, a work-item is started for each memory object of the recurrence matrix. This kernel-based approach prevents copying null-initialised arrays from the host device to the memory of the compute devices, which would drastically increase the data transfer. Although being the favourable approach, this kernel execution also consumes additional runtime.

The custom bit-wise storage format has significant advantages in comparison the the row-wise and column-wise storage format. It supports coalesced write access due to storing the similarity values belonging to the same row of successive columns consecutively. Furthermore, it allows to inspect successive rows of the same column, without having to reload data from the global memory of the compute device. This property reduces the number of cache lines transferred to the L1 cache (see Sect. 5.1.1).

5.1.4 Intermediate Results Recycling

As presented in Sect. 3.1, *SRA* structures the RQA processing into three analytical operators:

- *create_recurrence_matrix*,

- *detect_diagonal_lines*, and
- *detect_vertical_lines*.

Each of those operators has a specific maximum degree of parallelism, with the creation of the recurrence matrix having the highest one. The line detection operators require that the binary similarity values of the recurrence matrix are computed prior to the inspection of columns and diagonals. Although both line detection operators can be executed in parallel, their execution is serialised.

Inspecting a column or diagonal of the recurrence matrix does not require that all corresponding similarity values are computed previously. It is rather required that only those similarity values are computed, which belong to matrix elements evaluated up to the current iteration of the sequential scan of a column or diagonal. This relaxed constraint allows to include the computation of the pairwise input vector similarities within the line detection operators. In this way, it is possible to omit the *create_recurrence_matrix* operator.

The combination of serialising the execution of the line detection operators and being able to omit the separate *create_recurrence_matrix* operator allows the *recycling of intermediate results* [Ivanova et al., 2010]. The goal of this approach from database technology is to optimise query execution. It proposes to store the results of calculations, which are reused later on. This recycling strategy assumes that the execution model adheres to *operator-at-a-time*. Given the two operations A and B, the result of A serves as an argument for operation B. The *recycler*, a specific software component, dynamically decides, which intermediate results from A to store within or evict from memory. The approach has been proposed especially for workloads that are dominated by read accesses, such as analytical tasks.

The concept of intermediate results recycling fits RQA well, because there exist multiple operators that are related based on predefined dependencies. Only one of those operators is executed at a time. The concept of recycling is implemented by integrating the computation of the pairwise input vectors similarities into one of the line detection operators. Since the intermediate results are stored under all circumstances, no specific recycler module is employed. Note, executing a separate *create_recurrence_matrix* operator itself adheres to storing intermediate results.

The recycling of intermediate results is adapted to *SRA*. This thesis proposes to compute the binary similarity values during the inspection of columns and reuse them during the detection of diagonal lines. The reason for introducing this specific ordering regarding the execution of the analytical operators lies in the potential symmetry of the recurrence matrix. If the symmetry property is fulfilled, the *detect_diagonal_lines* operator does only require to evaluate $N(N-1)/2$ matrix elements. In this case, only one half of the recurrence matrix needs to be computed. However, it is required to compute the full recurrence matrix for the detection of vertical lines.

Discussion

The recycling of intermediate results allows to save overhead costs, e.g., for creating and executing OpenCL kernel instances, by eliminating the *create_recurrence_matrix* operator. This requires to include the computation of the pairwise input vector similarities within the *detect_vertical_lines* operator, whose maximum degree of parallelism N is much smaller than N^2 of executing a separate operator for creating the recurrence matrix. The tradeoff between operator elimination and reduction of parallelism determines, if intermediate results recycling allows to reduce the overall runtime of conducting RQA.

As stated before, the *detect_vertical_lines* operator is executed preceding to the detection of diagonal lines, because the latter does only inspect roughly one half of the recurrence matrix, if it is symmetric. This property can be exploited to minimise the amount of intermediate results written to the global memory of the compute devices. If a separate *create_recurrence_matrix* operator is executed, each matrix element has to be written once. Additionally, it has to be read one and a half times, one time during the detection of vertical lines and a half time during the detection of diagonal lines. Performing intermediate results recycling, the binary similarity values required for the detection of vertical lines are computed on the fly. Hence, only those elements need to be stored, which are required by the *detect_diagonal_lines* operator. Having a symmetric recurrence matrix, only $N(N-1)/2$ recurrence matrix elements have to be stored. The minimisation the amount of write accesses to global compute device memory allows to further reduce the overall runtime.

5.1.5 Recurrence Matrix Materialisation

All aspects presented in the previous sections assume that the sub matrix currently processed is stored completely or partially within the global memory of a compute device during the detection of line structures. This procedure is referred to as *materialised views* in database technology. Here, views on database tables are created during the processing of a query are materialised. This means that they are stored by the database management system, e.g., on hard disk. Materialised views are attached with specific limitations, although they have proven their applicability in different scenarios. In particular, there is a tradeoff between the memory latency for storing intermediate results and computing them multiple times [Gupta et al., 1993].

In Sect. 5.1.4, we demonstrated that it is feasible to include the computations of the pairwise input vector similarities in the *detect_vertical_lines* operator. This approach can be easily extended to the *detect_diagonal_lines* operator. In this way, the storage of recurrence matrix data in the global memory of a compute device can be omitted. Depending on whether the recurrence matrix is symmetric or not, N^2 or $N(N-1)/2$ similarity comparisons have to be conducted while inspecting its $2N-1$ or $N-1$ diagonals regarding line structures.

Discussion

Not-materialising the recurrence matrix in the memory of the compute devices influences the amount of data read and written by each operator. In the following, the amount of memory transferred while conducting RQA is investigated, considering materialising and not-materialising the recurrence matrix. For the purpose of simplification it is assumed that reading data from and writing data to the global memory has the same latency. In addition, only the amount of data elements is considered, not the actual size of the memory occupied.

Using a separate *create_recurrence_matrix* operator, the amount of data elements transferred comprises:

(i) $2mN^2$ data elements read during the computation of the pairwise similarity values,

(ii) N^2 data elements written during the materialisation of the recurrence matrix,

(iii) N^2 data elements read during the detection of vertical lines, and

(iv) N^2 (non-symmetric) or $N(N-1)/2$ (symmetric) data elements read during the detection of diagonal lines.

Computing the similarity of a single pair of multi-dimensional vectors requires to evaluate $2m$ data elements, each referring to a specific vector component (i). The corresponding binary similarity value is written once to the global memory of a compute device. This has to be performed for each of the N^2 elements of the recurrence matrix (ii). Similarly, the *detect_vertical_lines* operator requires to evaluate each matrix element during the inspection of the columns of the recurrence matrix (iii). Each or only every second matrix element is read once from the global memory of the compute device during the detection of diagonal lines, depending on whether the matrix is symmetric or not (iv).

Regarding not-materialising the binary similarity values, the amount of data elements transferred is structured as follows:

(a) $2mN^2$ data elements read during the detection of vertical lines.

(b) $2mN(N-1)/2$ (symmetric) or $2mN^2$ (non-symmetric) data elements read during the detection of diagonal lines.

Considering the non-materialisation approach, the write accesses in (ii) are omitted, since no materialisation is performed. The read accesses to the global memory in (iii) and (iv) do not have to be conducted either. At the same time, reading the input vector components during the computation of the pairwise similarities has to be included in the detection of vertical (a) and diagonal (b) lines.

Considering materialising the matrix data, the amount of memory accesses captured by (i) depends on the total number of input vectors N and the embedding dimension m, whereas the accesses of (ii), (iii) and (iv) only depend on N. On the contrary, the amount of memory accesses while detecting vertical (a) and diagonal lines (b) without materialisation both depend on m and N.

Appendix B contains equations that compare the amount of data elements transferred when using a separate *create_recurrence_matrix* operator to materialise the recurrence matrix (*CRM Operator*), not materialising the matrix (*Non-Materialisation*) and materialising the recurrence matrix through recycling (*Recycling*). Furthermore, a distinction is made whether the recurrence matrix is symmetric or not. An essential assumption is that the computational overhead to read data from and write data to memory is identical. Each formula is solved for the embedding dimension m to determine the position of the break-even point regarding the amount of data elements transferred of the corresponding approaches.

Appendix B.1 considers a symmetric recurrence matrix. Here, the equilibrium embedding dimension depends on the total number of input vectors N. Assuming Equ. 5.3, not-materialising the recurrence matrix requires less memory accesses for embedding dimensions $m \leq 2$. Having $m \geq 3$, materialising the matrix data within a separate *create_recurrence_matrix* operator is more efficient.

$$\lim_{N \to \infty} \frac{5N - 1}{2N - 2}, \tag{5.3}$$

Appendix B.2 compares employing a separate *create_recurrence_matrix* operator (left) and not-materialising matrix data (right) under the assumption that the recurrence matrix is not symmetric. Having a dimensionality of one, not-materialising the matrix data requires less memory accesses. Setting $m \geq 2$, the application of a separate operator that materialises the recurrence matrix requires less memory accesses.

Appendix B.4 and B.3 present similar formulas considering the recycling of intermediate results. Here, materialising the recurrence matrix consumes the same amount of memory accesses or less for embedding dimensions $m \geq 1$. Note that the amount of memory accesses is only one of the criteria that influences the performance of corresponding implementations. It rather serves as an indicator regarding performance comparisons.

5.2 Index Data Structures

Section 5.1 focuses on parallel brute-force processing in the context of RQA, which computes of all pairwise input vector similarities stored in the recurrence matrix. Nonetheless, Sect 3.2 demonstrated that index data structures, in particular grid directories and the application of multi-dimensional search tress, allow to increase the efficiency of RQA by pruning similarity comparisons. Note that this does only affect the processing of the *create_recurrence_matrix* operator.

The application of index data structures can be distinguished into:

1. *building* the corresponding data structures using a set of data points, and

2. *querying* the data structures regarding the neighbours of a set of query points.

The following sections describe the properties of the two types of index data structures mentioned above, focussing on the computational concepts as well as the implications regarding runtime and memory usage.

5.2.1 Grid Directories

In Sect. 3.2.1, this thesis demonstrates the application of grid data structures in the context of neighbour search. In this regard, a grid data structure is created based on a set of multi-dimensional input vectors. Each vector is assigned to a specific grid cell. Grid data structures offer the benefit of only inspecting those grid cells, which might contain the neighbours of given query vector. As a recap, the likelihood of containing neighbours is determined by the distance from grid cell, in which the query vector is located.

In [Green, 2012], it has been demonstrated that grid processing can be implemented efficiently on graphics cards. Here, the CUDA framework is employed to detect collisions between up to three-dimensional particles in a massively parallel manner. Two different approaches are presented, which both rely on partitioning the multi-dimensional space using a uniform grid. A particle consists of a centre, defined as multi-dimensional point, and a radius r. Its centre is located in a specific source grid cell. The extent of a particle traverses this source grid cell and zero or more adjacent cells, depending on the radius r. A collision between two particles is defined as an overlap between two particle extents.

[Green, 2012] proposes to apply a length of $2r$ to the edges of each grid cell. This ensures that the extent of a particle potentially traverses only those grid cells directly adjacent to its source grid cell. The number of adjacent grid cells depends on the dimensionality of the space in which the particle resides. [Green, 2012] claims that on GPUs it is simpler to identify those grid cells, which might be traversed by the extent of a particle, instead of determining the ones which are actually traversed.

These concepts are adapted to recurrence analysis. Instead of particles consisting of a centre and an extent, this thesis considers input vectors and their neighbourhood defined by the similarity threshold ϵ. In the following, we focus on the fixed-radius neighbourhood condition.

In [Green, 2012], the parallel creation of the grid data structure is either performed using:

1. Atomic operations, or
2. Sorting.

Both approaches are distinguished in the following and adapted to the properties of sub matrix processing in the context of *SRA*. It is assumed that a uniform grid with a grid cell edge length of 2ϵ is applied.

Grid Creation Using Atomic Operations

Using atomic operations, the grid data structure is represented by two one-dimensional arrays:

1. `grid_cells`, and
2. `grid_counters`.

`grid_cells` comprises fixed-sized segments of integer values, where each segment stores the indices of the objects located in a specific grid cell. `grid_counters` provides an integer value for each grid cell, storing the number of objects located in this cell. Both arrays reside in the global memory of a compute device.

Each multi-dimensional vector is assigned to a specific grid cell during grid creation. An OpenCL work-item is created for each vector. It computes the index of the grid cell in which the vector is located and updates the corresponding element of the `grid_counters` array, using an *atomic incrementation* operation. The work-item calls an atomic function, because multiple work-items may update the data referring to the same grid cell simultaneously. The atomic function returns the number of vectors located in the grid cell prior to the incrementation. This value serves as an input to store the index of the input vector in the corresponding segment of the `grid_cells` array.

Discussion. The approach using atomic operations assumes that segments in the `grid_cells` array are of fixed size. Ideally, this size is set to the maximum number of vectors assigned to a single grid cell. To determine this maximum value, the grid cell index of every vector has to be computed prior to the actual grid construction. In the following, the segment size is therefore set to the total number of input vectors that need to be stored during the processing of the current sub matrix. This allows to avoid pre-processing. It is the highest amount of vectors a single grid cell potentially has to store. The inherent problem of this approach is that the memory consumption of the `grid_cells` array increases enormously with an increasing number of grid cells, e.g., due to an increasing number of dimensions. A third approach proposed in [Green, 2012] is to employ heuristics to estimate the segment size. Note that those heuristics do not prevent segments storing large numbers of input vector indices.

Overall, creating the grid data structure using atomic operations adheres well to the properties of GPU computing, because all compute-intensive operations, such as computing the grid cell index, can be conducted in a massively parallel manner. Nonetheless, the usage of atomic operations introduces necessary synchronisation that creates a bottleneck regarding the parallel execution of work-items. Furthermore, the approach is only favourable for small amounts input vectors and grid cells, due to the potentially large size of the grid cell segments.

Grid Creation Using Sorting

[Green, 2012] proposes a second grid creation approach, which is based on sorting. It overcomes the restriction of using fixed-sized grid cell segments. Here, the grid data structure is represented by the following two one-dimensional arrays:

1. `grid_cells`, and
2. `grid_cells_start`.

The `grid_cells` array stores the indices of the objects located in each grid cell, similar to the approach using atomic operations. A grid cell is likewise represented by a consecutive segment of input vector indices. In contrast, those segments are of variable size. The start index of every segment in the `grid_cells` array is stored as an integer value in `grid_cells_start`.

The approach based on sorting is separated into two major steps. First, the grid cell index of each multi-dimensional vector is determined in a massively parallel manner. In [Green, 2012], it is proposed to use a hash function as an indirection to assign an object to a grid cell. This thesis proposes to compute the actual grid cell index in the context of *SRA*. The mapping is stored in an *intermediate array*, where each element captures the grid cell index of a specific multi-dimensional vector.

Second, the intermediate array is sorted in ascending order of the grid cell indices. As a byproduct, the input vector indices per grid cell are stored as segments in `grid_cells`. The size of the array scales linearly, depending on the number of data points. At the same time, the elements of `grid_cells_start` are assigned. Here, empty grid cells are marked by using the same start index as the previous grid cell.

[Green, 2012] proposes to use a parallel sorting algorithm that is executed on the GPU. This strategy is out of the scope of this thesis. Instead it is proposed to create only the intermediate

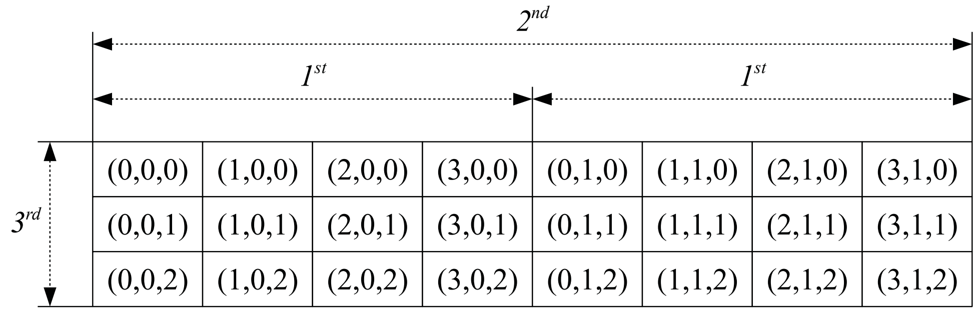


Figure 5.6: Grid cell order. The example refers to a three-dimensional space. Each grid cell is identified by a three-dimensional index (id_1, id_2, id_3), one component for each dimension. Each component is an integer value starting at zero. The 1^{st} dimension is separated into four segments, the 2^{nd} dimension into two segments, and the 3^{rd} dimension into three segments. This results in 24 grid cells. Note that all grid cells are stored consecutively in the memory with ascending multi-dimensional index starting at the 1^{st} component.

array in a parallel manner. To perform CPU-based sorting, the array is transferred from the global memory of the compute device to the main memory of the computing system. This introduces additional overhead regarding data transfer and avoids parallel execution, leading to an increased runtime. After the sorting is completed, `grid_cells` and `grid_cells_start` are transferred from main memory to the global memory of the compute device, to enable parallel neighbour search.

Discussion. Constructing the intermediate array in a massively parallel manner suits GPU processing. Furthermore, the size of `grid_cells` is independent of the number of grid cells. This property lets the approach being applicable even for large and irregular distributed sets of multi-dimensional vectors. Nonetheless, the minimisation of memory usage comes at the cost of conducting a separate sorting step that increases the overall runtime.

Determination of Grid Cell Index

Both grid data structures presented in [Green, 2012] use the array `grid_cells`, which stores the indices of objects belonging to the same grid cell consecutively. In [Böhm et al., 2001], it is stated that the numbering of grid cells in multi-dimensional space is "clumsy", unless the data space is limited. This requirement is fulfilled in the context of recurrence analysis. The extent of the input space is determined by the dimensionality of the input vectors as well as the minimum and maximum value for each dimension. The number of grid cells is computed by separating each dimension according to multiples of 2ϵ.

To identify a specific grid cell, we apply a multi-dimensional index that comprises a component id_i for each of the m dimensions of the data space. Each component represents the index of the segment regarding the i-th dimension. Given an input vector $\vec{x}$, the component id_i is computed as:

$$id_i = \frac{\vec{x}_i - min_i}{2\epsilon}. \tag{5.4}$$

$\vec{x}_i$ refers to the i-th component of $\vec{x}$, min_i to the minimum value of the i-th dimension and ϵ to the similarity threshold of the fixed-sized radius neighbourhood condition.

The grid cells are stored sequentially within the global memory of the compute device. They are organised as depicted in Fig. 5.6. As a result, the multi-dimensional index has to be serialised. To compute this serialised index $id_{serialised}$ of a grid cell, the individual components of its multi-dimensional index are multiplied with dimension-specific integer values, hereafter referred to as *multipliers*:

$$id_{seralised} = \sum_{i=1}^{m} u_i id_i, \tag{5.5}$$

with m being the embedding dimension, u_i being the multiplier of dimension i and id_i being the index of dimension i. The multiplier u_i is computed as the cumulated number of grid cell segments up to the previous dimension $i-1$, with $u_1 = 1$. Regarding the example from Fig. 5.6, those multipliers are $(u_1, u_2, u_3) = (1, 4, 8)$. The set of multipliers is identical for all multi-dimensional vectors.

Neighbour Search

The grid data structure created by using one of the two approaches presented above serves as an input for the neighbour search. Here, the X and Y axis of a sub matrix usually refer to different sub sets of the full set of multi-dimensional vectors. During the sub matrix creation, each set may either serve as *data points* or *query points*. The assignment is chosen such that for each vector belonging to the X axis, a set of neighbours regarding the vectors belonging to the Y axis is computed. The data points are organised in a uniform grid using the similarity threshold ϵ.

For each query point, the set of similar data points is determined as follows. First, the grid cell index of the query point is computed. Second, each grid cell adjacent to this source cell is inspected regarding neighbouring vectors. Again, the edge length of the uniform grid cells is chosen to be 2ϵ. This ensures that neighbouring vectors can only be located in grid cells directly adjacent to the source grid cell.

Given an arbitrary query vector, the set of adjacent grid cells is determined by incrementing and decrementing each component of the corresponding multi-dimensional grid cell index. As an example, a source grid cell with the two-dimensional index $(id_1, id_2) = (10, 23)$ is assumed. The set of indices referring to grid cells directly adjacent include:

- $(9, 22)$,
- $(9, 23)$,
- $(9, 24)$,

- $(10, 22)$,
- $(10, 24)$,
- $(11, 22)$,
- $(11, 23)$, and
- $(11, 24)$.

The search for neighbours in adjacent grid cells is implemented in a parallel manner. Given a query point, each work-item inspects a single adjacent grid cell regarding neighbours. The work-item computes the binary similarities between the query point and all data points located in the adjacent grid cell. The similarity results are stored in the binary sub matrix.

There are potential variations regarding the implementations of the grid-directory methods, similar to parallel brute-force processing. This includes the representation of the input data, by storing the input vectors either using a row-wise or column-wise layout (see Sect. 5.1.1), and the representation of the similarity values, either using a byte or bit representation (see Sect. 5.1.3). The impact of those variations on the runtime is examined in Sect. 6.3.1.

5.2.2 Multi-Dimensional Search Trees

This thesis considers the k-d tree as one representative of the multi-dimensional search trees. It is a well-studied data structure, for which various optimised implementations written in different programming languages exist. Therefore, it is reasonable to use existing k-d tree software instead of writing *yet another implementation*. This thesis focusses on k-d tree implementations that provide a Python API, since the *SRA* implementation is based on this programming language. There are two prevailing and continuously maintained k-d tree implementations available in the Python packages:

- *scipy* [Jones et al., 2001], and
- *scikit-learn* [Pedregosa et al., 2011].

The corresponding classes encapsulating the k-d tree functionality are:

- `scipy.spatial.cKDTree`, and
- `sklearn.neighbors.KDTree`

Both implementations contain source code written in *Cython* [Behnel et al., 2011], a Python language extension. The source code is statically compiled, which enables significant performance improvements in comparison to interpreted Python code.

Similar to grid data structures, the usage of a k-d tree is subdivided into two basic steps:

1. *building* the k-d tree using data points, and

2. *querying* the k-d tree using query points.

During the build phase, the data points are assigned to tree nodes. After the build phase is completed, the tree can be queried regarding the neighbours of the query points. The runtimes for conducting both phases are referred to as *build time* and *query time*.

`cKDTree` and `KDTree` provide functionality to query for a fixed number of neighbours and to apply a fixed radius neighbourhood. Here, the focus lies on identifying neighbours using a fixed ϵ. The k-d tree implementations mentioned differ from its original definition. The set of modifications includes:

1. the storage of more than one object within the leaf nodes of the tree, and
2. the usage of multiple trees during neighbour query execution.

The original definition of the k-d tree considers a leaf node to represent only a single data object. Neighbours of a query point are discovered by traversing the nodes of the tree and pruning unnecessary similarity comparisons. Nonetheless, a brute-force neighbour search may be more efficient for small sets of data points [scikit-learn developers, 2016]. Both approaches can be combined by storing more than one data object per leaf node. This presents several advantages regarding the runtime mentioned above. First, the build time is reduced, due to creating less tree nodes. Second, the query time is reduced, due to traversing less nodes and performing brute-force similarity comparisons in the leaf nodes.

The choice regarding the number of objects stored in the leaf nodes, referred to as *leaf size*, has major influence on the cumulated runtime. If it is chosen to small, the build time increases due creating more tree nodes and the query time increases due to traversing a higher number of tree nodes. If it is chosen too large, not enough similarity comparisons are pruned and brute-force neighbour search becomes inefficient. Detecting the optimal number of objects per leaf node is out of the scope of this dissertation. We rely on the default values of each implementation:

- `cKDTree`: 16, and
- `KDTree`: 30.

The original definition of the k-d tree only considers the data points to be organised in a single tree structure. This tree is traversed regarding neighbouring objects one query object at-a-time. The query point is compared to the nodes of the k-d tree. [Gray and Moore, 2000] describes an approach how to prune similarity comparisons of sets of query points simultaneously. For this purpose, a second k-d tree is constructed that captures the set of query points. Nodes of both trees are compared to each other, to perform neighbour search. The nodes stored in the corresponding sub trees are either compared to each other or not, depending their mutual similarity. This *dual tree approach* potentially allows to reduce the cumulated runtime. Up to this point, this modification is only supported by `KDTree` with respect to k-nearest neighbour queries. It is deactivated during the evaluation in Sect. 6.3.2. Note that the functionality of `KDTree` is accessed in the following via the wrapper `sklearn.neighbors.NearestNeighbours`.

Application to *SRA*

A sub matrix stores the binary similarities of pairs of two sets of input vectors, one representing the X and the other representing the Y axis. In general, those sets differ, except the sub matrices along the middle diagonal. By definition, the multi-dimensional vectors referring to the Y axis are treated as data points, whereas the vectors referring to the X axis are used as query points.

A k-d tree is build by calling the constructor of `cKDTree` or `NearestNeighbors`. The multi-dimensional vectors serving as data points are passed as arguments. Both classes provide methods to perform neighbour search:

- `cKDTree.query_ball_point`, and
- `NearestNeighbors.radius_neighbors_graph`.

Given a fixed radius, `query_ball_point` returns a list of neighbours for each object contained in a list of query points, which is passed as an argument. `radius_neighbors` returns a neighbour graph represented as a compressed sparse matrix (see Sect. 5.1.2). Both results are converted to an object of the type `scipy.sparse.csc_matrix`. In this way, the algorithms from Sect. 5.1.2 regarding the detection of line structures can be applied. `scipy.sparse.csc_matrix` provides built-in functionality to convert the compressed to an uncompressed matrix representation. Hence, the algorithms from Sect. 2.2.1 can also be employed. The conversion requires computational overhead, which leads to an increase in runtime[1]. Nonetheless, using `scipy.sparse.csc_matrix` allows to supply a consistent interface regarding the execution of the *create_recurrence_matrix* operator.

`cKDTree` and `NearestNeighbors` offer the possibility to perform neighbour search for multiple query objects in parallel. Considering the former, this feature is enabled regarding the application of a radius neighbourhood as well as a fixed amount of nearest neighbours. In contrast, the latter supports parallel query processing only for a fixed number of nearest neighbours. As a result, it is disabled during the evaluation in Sect. 6.3.2, to ensure comparable runtime results.

Referring to *SRA*, it is assumed that a k-d tree storing the data points is created while processing each sub matrix. This leads to organising identical sets of input vectors multiple times, because sub matrices with the same row index refer to the same segment of the Y axis of the full recurrence matrix. The corresponding multiplication of the build time could be avoided by creating exactly one k-d tree for each segment of the Y axis. This tree can be used by all sub matrices belonging to the same row. Nonetheless, the expected runtime savings are rather small, since the build time contributes only a small portion to the total runtime for executing the *create_recurrence_matrix* operator[2]. In addition, the memory footprint of conducting RQA increases. This would furthermore lead to a dependence on the size of the full recurrence matrix. Therefore, this modification is omitted.

[1]Regarding the analysis of an examplary sub matrix consisting of $10,000^2$ elements, the conversion took about twice the time as for conducting neighbour search.

[2]Experiments presented in [Vanderplas, 2013] indicate that the build time is up to two magnitudes smaller than the query time.

Previous Performance Analysis

[Vanderplas, 2013] has conducted a detailed performance analysis of `NearestNeighbors` and `cKDTree`. The benchmarking focusses on the fixed amount of nearest neighbour query, considering a single query object. The runtime for building as well as the querying a k-d tree is investigated. In this regard, different data point distributions are used:

Uniform distribution: Data points are distributed uniformly in the input vector space.

Digits distribution: Data points refer to pixel values retrieved from images.

Spectra distribution: Data points refer to flux observations from astronomical spectra.

The build and query time are analysed regarding the influence of:

- leaf size,
- number of neighbours,
- number of data points, and
- dimensionality of the input vector space.

Five experimental runs are conducted for each combination of input variables, computing the average runtime. The Euclidean distance is applied in each of those experiments.

Regarding the experimental results, the build time increases with increasing leaf size, due to the lesser amount of tree nodes that need to be created. There exists an optimal leaf size, which leads to minimum runtime regarding the query processing. Note that the optimal leaf sizes differ between the implementations. Increasing the number of neighbours does not infect the build time. Contrastingly, the query time increases super-linear, due to the maintenance of a priority queue containing the set of k neighbours. Increasing the number of data points, the build and query time scale $\mathcal{O}(NlogN)$. Increasing the dimensionality, the build and query time increase roughly logarithmically. `cKDTree` has significantly smaller build time in comparison to `NearestNeighbors`, up to two magnitudes. The query time is roughly similar.

Discussion. [Vanderplas, 2013] provides no information regarding the computing system employed, e.g., the type of CPU used and the size of the main memory. The analysis is restricted to the fixed amount of nearest neighbours neighbourhood using a single query object. Information regarding other scenarios is not provided.

In [Vanderplas, 2013], a brute force method is mentioned that computes the similarity between the query point and each data point. There is no information provided regarding this exhaustive implementation, e.g., which programming language has been used. More specifically, there are only estimations given regarding the runtime for performing the nearest neighbour query.

The analysis of [Vanderplas, 2013] gives only a hint regarding the performance characteristics of both k-d tree implementations, due to the issues mentioned before as well as the usage of only small sets of up to 10^4 data points. The applicability of `cKDTree` and `NearestNeighbors` in the context of RQA requires deeper investigation, which is conducted in Sect. 6.3.2.

6 Analytical Operator Evaluation

This chapter addresses the impact of selecting different realisations regarding the concepts from database technology presented in Chap. 5 on the performance of recurrence analysis processing. Each realisation either refers to parallel brute-force processing (see Sect. 5.1) or the usage of index data structures (see Sect. 5.2). The performance characteristics are examined and compared to other realisations of the same design dimension. For this purpose, the results of a set of experiments are presented. Note that the impact of each realisation is evaluated per analytical operator.

The structure of this chapter is as follows. Sect. 6.1 addresses foundations regarding all of the experiments conducted. This includes the description of the underlying experimental methodology and the computing environment used during the experiments. Sect. 6.2 considers a set of experiments referring to the parallel brute-force processing. A set of experiments discussing initial assumptions are presented prior to investigating the actual implementation dimensions, such as how the spatial distribution of the multi-dimensional vectors affects the performance. The experiments presented in Sect. 5.2 evaluate how the usage of multi-dimensional index data structures influences the runtime behaviour of executing the *create_recurrence_matrix* operator.

An initial evaluation on the impact of different implementation strategies on the performance characteristics of RQA processing has been published in [Rawald et al., 2015].

6.1 Foundations

This section describes the foundations of the experiments presented in the subsequent parts of the chapter. The following considerations hold for the evaluations referring to the parallel brute-force processing and the index data structures.

6.1.1 Experimental Methodology

The evaluation comprises a set of experiments. Each experiment refers to a specific *analytical scenario*, which is defined by the RQA input parameters:

- time series,
- embedding dimension,
- time delay,
- similarity measure, and
- similarity threshold.

Each experiment refers to the processing of a recurrence matrix that fits in the global memory of an OpenCL compute device, corresponding to the concept of sub matrices in the context of *SRA*. Those sub matrices are constructed from only thousands of multi-dimensional vectors.

If the vectors are reconstructed from multiple time series, the time delay method is not applicable. In this case, the time delay parameter is negligible. A ratio of the maximum phase space diameter between zero and one is used regarding the pairwise vector comparisons, instead of defining a fixed threshold (see Sect. 2.1.2). This approach relates to the practice that is used in real-world applications. The resulting absolute threshold depends on the properties of the time series analysed.

While conducting an experiment, usually one RQA input parameter is selected as independent variable. The impact of varying this parameter on the performance characteristics of a set of RQA implementations is observed. Those implementations differ regarding specific computational aspects, such as the representation of input data. The focus of the performance analysis is on the runtime of executing a RQA implementation. The corresponding measurements are collected for each operator individually. This enables a fine-grained investigation of the effects of each computing approach or implementation strategy.

The operator implementations relying on OpenCL measure the runtime using profiling events (see Sect. 4.3.2). Here, the time elapsed between the start and the end of the execution of commands is determined. Operators implementing RQA functionality using index data structures calculate the difference between timestamps at the end and the start of its execution, implemented using the Python module `time`.

In addition to the runtime, the *AMD Radeon RX 470* compute device, which is employed during this evaluation, allows to extract additional performance counters, such as the *cache hit rate* (`CacheHit` [Advanced Micro Devices, Inc., 2013a, pp. 15–17]). This information is obtained using the tool `CodeXLGpuProfiler` that is part of AMDs *CodeXL* suite. The values of such counters ideally allow to draw conclusions regarding the causes of the runtime behaviour.

The complete computing environment is described in detail in Sect. 6.1.2. In this regard, a specific computing system is composed of hardware and software components. Note that a computing system may contain one or more *compute devices*, such as CPUs and GPUs.

An *experimental configuration* refers to a combination of:

- analytical scenario,
- RQA implementation, and
- compute device.

Multiple experimental runs are executed for each configuration. The number of runs conducted is specific to each experiment. The impact of outliers regarding the runtime or performance counters measured is reduced, by repeating the computations for each configuration.

Prefaced by a short introduction, each of the following sections referring to an experiment consists of:

1. a description of the experimental *setup*,

2. a set of initial *hypotheses*, and

3. an analysis of the experimental *results*.

The setup comprises the experimental configurations used for this particular experiment. Furthermore, it specifies additional parameters, such as the amount of experimental runs conducted for each configuration. Based on this information, initial hypotheses regarding the performance behaviour of each RQA implementation are formulated. The experimental results are used to validate the initial hypotheses. The experimental setups of each of the following experiments are captured in App. D. The corresponding experimental results are depicted in App. E.

6.1.2 Computing Environment

This section describes the computing environment employed for conducting the following experiments. The detailed description of the hardware and software specifications of the computing systems can be found in App. C. The focus is particularly on the compute devices that are used for OpenCL processing. The set of computing systems includes:

(A) a workstation containing a single *AMD Radeon RX 470* GPU,

(B) a workstation containing two *Nvidia GeForce GTX 690* GPUs, and

(C) a server containing two *Intel Xeon E5620* CPUs.

The computing systems containing GPU hardware additionally comprise a single 64-bit CPU, supporting the *x86_64* instruction set. Each of those systems is equipped with 16GB of main memory. The GPU devices are attached using version 3.0 of the *Peripheral Component Interconnect Express* (PCI-E). Each GPU device is equipped with dedicated memory that is separated from the main memory of the computing system. This memory is referred to as global memory in the context of OpenCL.

A single Nvidia GeForce GTX 690 GPU comprises two *graphics processors*, resulting in a total of four processors. Each GPU processor is supplied with dedicated memory, that is not shared with the other processor. Each processor is treated as a separate compute device that is addressed individually.

The third computing system comprises two Intel Xeon E5620 server processors, that share 48GB of main memory. Each CPU, implementing Intels proprietary *hyper-threading* technology [Marr et al., 2002], contains four cores and runs eight threads. Note that the two CPUs are treated as a single compute device by the OpenCL runtime. The CPUs employed also support version 4.2 of Intels *Streaming SIMD Extensions*, enabling additional parallel computing capabilities. Here, separate registers for performing the same instruction on multiple data elements simultaneously are provided.

Regarding the software environment, every computing system runs an operating system based on the *Linux* kernel [Torvalds et al., 2017]. The kernel versions employed differ between the systems. The software environment includes an OpenCL runtime engine and corresponding device drivers. The relevant software packages provided by the hardware vendors are:

(A) the *AMDGPU-PRO Driver*,

(B) the *NVIDIA Accelerated Linux Graphics Driver*, and

(C) the *OpenCL Runtime for Intel Core and Intel Xeon Processors*.

The version of those software packages including the supported OpenCL version are listed for each compute device in App. C.

6.2 Parallel Brute-Force Processing

This section refers to experiments investigating the performance characteristics of RQA implementations based on parallel brute-force processing. Those implementations rely on conducting all pairwise vector similarities captured in a sub matrix in a massively parallel fashion using the OpenCL framework. The set of experiments is subdivided in two blocks. The first block comprises experiments that verify initial assumptions, including the impact of:

1. the *spatial distribution* of multi-dimensional vectors,
2. the *similarity measure* regarding the quantitative analysis, and
3. the status of the *default compiler optimisations*

on the performance of the massively parallel RQA implementations. The second block contains experiments referring to the design dimensions described in Sect. 5.1, including:

1. input data representation,
2. recurrence matrix representation,
3. similarity value representation,
4. intermediate results recycling, and
5. recurrence matrix materialisation.

The experiments are presented from Sect. 6.2.1 to Sect. 6.2.8, with each subsection referring to a specific experiment.

6.2.1 Input Vector Distribution

The first of the three initial experiments evaluates the impact of the distribution of the input vectors residing in m-dimensional space. The location of each vector heavily affects its similarity in relation to other vectors. It is analysed, how the spatial distribution influences the runtime characteristics of massively parallel implementations of the *create_recurrence_matrix* operator.

Setup

The experiment applies a set of random distributions to place the input vectors in multi-dimensional space, including:

- the *uniform* distribution,
- the *normal* distribution,
- the *exponential* distribution, and
- the *Cauchy* distribution.

Each of those distributions is assigned with *standardised parametrisations.* For the purpose of this experiment, the input vectors are not extracted from a time series using the time delay method, but rather generated explicitly using functions provided by the NumPy module `numpy.random`. Given a distribution as well as a dimensionality m, the individual vector components are created by applying the NumPy function m times. For each distribution, Tab. D.1 presents the corresponding `numpy.random` function as well as a short description and its parametrisations, if available. Example distribution instances referring to two-dimensional vectors are shown in Fig. D.1.

The distribution type serves as independent variable regarding this experiment. The dependent variable is the runtime for executing the operator that creates the recurrence matrix. The time measurements are restricted to the *create_recurrence_matrix* operator, because the computation of the binary similarity values is the central aspect of RQA that is influenced by the distribution of input vectors. Note that different distributions may lead to different topological characteristics within recurrence plots, e.g., their density regarding recurrence points. The impact of those characteristics on the performance of executing the line detection operators are considered in the following sections, for example in Sect. 6.2.5.

The values of the RQA input parameters apart from the distribution type remain constant, for example the embedding dimension and the similarity measure employed (see Tab. D.3). As mentioned before, the similarity threshold chosen is a fraction of the maximum phase space diameter, which depends on the extent of the space spanned by the full set of input vectors.

The properties of the RQA implementations employed during the experiment are depicted in Tab. D.2. The variant chosen relies on a row-based representation of the input data, because it stores each component of a vector separately, as required by the predefined distributions. This procedure is not applicable using the overlapped column-based representation, which does only refer to the extraction of values from a single time series.

Hypotheses

Parallel brute-force processing conducts all pairwise input vector comparisons stored within a recurrence matrix. There is no pruning of comparisons based on the spatial locality of the input vectors. Hence, the runtimes observed on a particular compute device should be independent of distribution type applied.

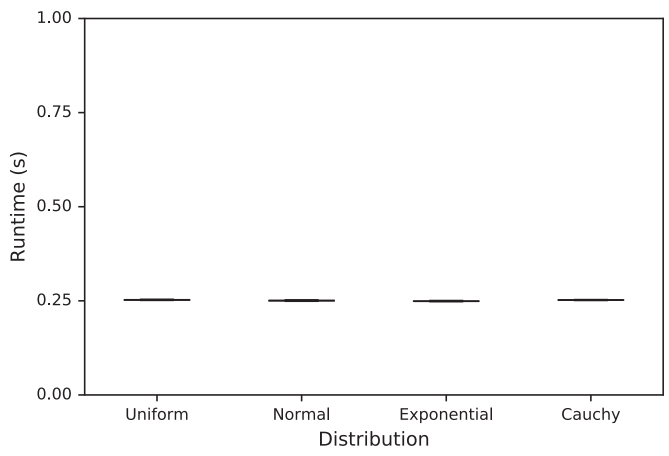


Figure 6.1: Multi-dimensional vector distribution - AMD Radeon RX 470.

Results

The experimental results are presented in App. E.1.1. Figure 6.1 depicts the runtime results regarding the execution of the experiment on the AMD Radeon RX 470 GPU. For each type of distribution, there is a boxplot [Tukey, 1977] that shows the spread of runtimes regarding 100 experimental runs. There are only small variances in relation to the mean runtime, signalled by the small extents of each boxplot. Confirming the initial hypothesis, the runtime performance of the *create_recurrence_matrix* operator does not depend on the spatial distribution of input vectors. This finding holds independent of the compute device employed.

The following experiments referring to parallel brute-force processing employ input data adhering to a uniform distribution of random values. Unless stated otherwise, the values are represented by a single series, from which the multi-dimensional vectors are reconstructed using the time delay method.

6.2.2 Similarity Measure

Different similarity measures can be employed to determine the pairwise input vector similarities. This experiment analyses the impact of the following measures on the runtime performance:

- the *Taxicab* metric,
- the *Euclidean* metric, and
- the *Maximum* metric.

Each measure has a specific computational overhead, which among others depends on the efficiency of built-in functions used to implement the corresponding OpenCL C kernels. The Taxicab metric requires to compute the absolute difference of vector component pairs. For this purpose, the built-in function `abs` is applied. The Maximum metric also uses this function.

Additionally, it requires to compute the maximum of vector component pairs. It is implemented by calling the built-in function `max`. In contrast, the Euclidean metric is the only measure that is implemented without calling any built-in function. This is achieved by squaring the value of ϵ during the application of the similarity threshold, preventing to take the second root of the sum of the squared distances of the individual vector components.

Setup

The experiment compares the runtime behaviour of three RQA implementations that differ only regarding the similarity measure used to calculate the similarities of pairs of multi-dimensional vectors. In this way, the influence of other factors is eliminated.

The selection of the similarity measure does only affect the computation of the pairwise vector similarities. Therefore, only the runtime of the *create_recurrence_matrix* operator is considered as observational variable. The properties of the RQA implementations employed during this experiment are depicted in Tab. D.4. Table D.5 captures the assignments of the RQA input parameters that are not varied.

Hypotheses

The implementations of the built-in functions are provided by the hardware vendors and are usually optimised for specific platforms. Hence, their performance should vary across the different compute devices. The runtimes of RQA implementations using the Euclidean metric are expected to be similar to or lower than the using the other two metrics, since its implementation omits the computational overhead for calling built-in functions.

Results

The complete results are presented in App. E.1.2. Regarding the two GPU compute devices, the runtimes of using the different similarity measures vary only slightly, as indicated by the narrow boxplots. This allows to conclude that employing the built-in functions mentioned has almost no effect on the runtime behaviour.

In contrast, runtime variations between the different similarity measures can be observed on the OpenCL compute device consisting of two Intel Xeon E5620 CPUs (see Fig. 6.2). Here, the runtimes while using the Taxicab and Euclidean metric are almost identical. The runtimes measured while applying the Maximum metric are considerably higher. It is reasonable to conclude that this effect stems from calling the built-in function `max`, being the major difference regarding the computations conducted. The increase in runtime does not necessarily imply that the corresponding function is implemented inefficiently, but rather that its usage introduces additional computational effort.

In the following, the Euclidean distance is selected as similarity measure, because the corresponding OpenCL C kernels do not rely on built-in functions. As a result, their impact on the performance measurements is eliminated.

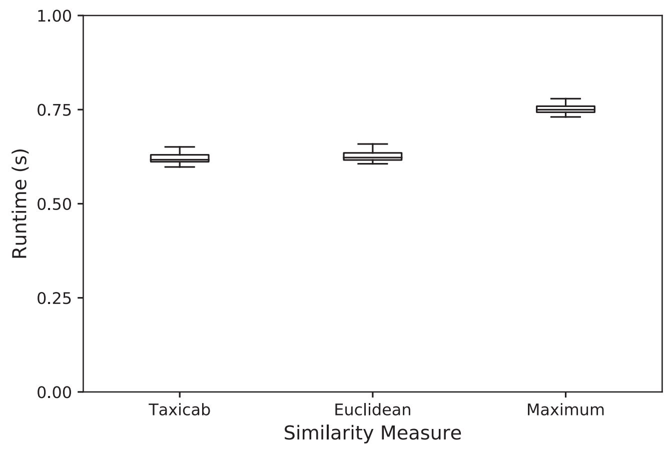


Figure 6.2: Similarity measure - Intel Xeon E5620.

6.2.3 Default Compiler Optimisations

The kernel functions written in OpenCL C are converted to machine code using a compiler that is usually provided by the vendor of the OpenCL platform. The prototype implementing RQA used in this evaluation performs the compiling during the execution of the host program. Here, the OpenCL runtime engine calls the compiler with a set of arguments triggering optimisations. Each OpenCL platform enables a specific set of *default compiler optimisations*, which aim at improving the performance of executing the compiled kernels on the respective platform.

Some compiler optimisations may come at the cost of inaccuracy regarding the computational results, e.g., due to the relaxed precision of mathematical functions. As an example, [Rawald et al., 2014a] presents an experiment conducting RQA on the Potsdam temperature profile. Although conducting the exact same analytical scenario, the experimental results computed on the Nvidia GeForce GTX 690 graphics card differed regarding the number of recurrence points, while having the default compiler optimisations enabled or disabled. This behaviour was observed when using version 331.49 of the Nvidia Accelerated Linux Graphics Driver. Having the default compiler optimisations enabled, $131,534,113,286$ recurrence points were detected within the whole recurrence matrix. Without default optimisations, the value drops slightly to $131,534,112,068$. The difference of $1,218$ points ($\approx 0.000001\%$) causes variations in the topology of the recurrence matrix that may result in diverging RQA measures.

The reason for those variations can most likely be explained by the usage of relaxed mathematical operations, e.g., regarding the processing of floating-point numbers [Zuras et al., 2008], that are activated by enabling default compiler optimisations. This issue has been resolved at least in the driver version 352.30. Although the differences in accuracy have a rather small impact, the user should be aware of their existence. It is especially important when comparing RQA results calculated by different devices.

The following experiment analyses the performance improvements, in particular the runtime reductions, gained by enabling default compiler optimisations. Note that all combinations of compute device and OpenCL runtime employed delivered identical computing results, indepen-

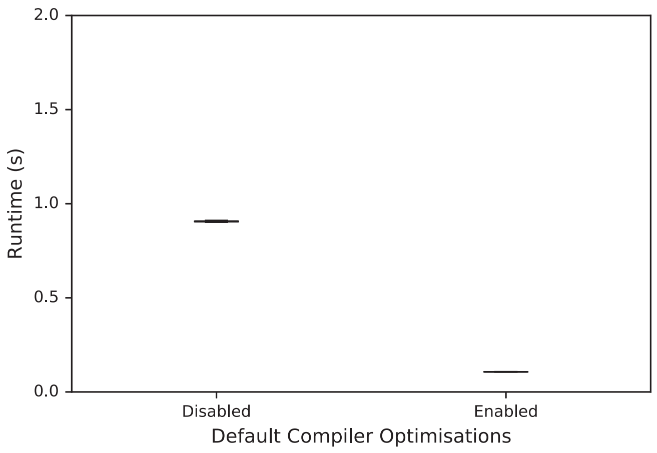


Figure 6.3: Default compiler optimisations - AMD Radeon RX 470.

dent of whether those optimisations are enabled or not.

Setup

For each default compiler optimisations status, *disabled* and *enabled*, the total runtime of executing all three analytical RQA parameters is measured on each compute device. The same RQA implementation is used in each experimental run. Its properties, for example the representation of the binary similarity values, are summarised in Tab. D.6. The assignments of the RQA input parameters, e.g., the dimensionality of the input vectors, are depicted in Tab. D.7.

Hypotheses

The total runtime of conducting RQA while having the default compiler optimisations enabled should be lower than having them disabled. This property should hold across all compute devices. The absolute and relative impact of enabling the default optimisations on the runtime performance should vary from device to device.

Results

The complete results are presented in App. E.1.3. As expected, enabling the default compiler optimisations delivers considerable performance improvements across all compute devices. The runtime can at least be reduced by a factor of two. The execution on the AMD Radeon RX 470 GPU experiences the most dramatic performance improvements. Here, a relative improvement of a factor of ≈ 9 is observed (see Fig. 6.3).

Based on these findings, the default compiler optimisations are enabled in each of the following experiments, unless stated otherwise.

6.2.4 Input Data Format

This experiment is the first one that explores the impact of the design dimensions presented from Sect. 5.1.1 to Sect. 5.1.5. In the following, the impact of the *row-wise* and *overlapped column-wise*, or short *column-wise*, input data format on the performance of the RQA processing is analysed.

Setup

The selection of the input data format influences the computation of the pairwise recurrence vector similarities. Therefore, the runtime of the *create_recurrence_matrix* operator execution regarding two RQA implementations is observed; one using the row-wise and the other using the column-wise input data format. The remaining properties, as depicted Tab. D.8, are equal independent of the input data format employed.

The embedding dimension is selected as independent variable, because it highly influences the layout of the input data within the global memory of a compute device. The range of dimensionalities applied in this experiment varies between one to twenty. This corresponds to real-world applications of recurrence analysis (see Sect. 2.1.1). The remaining input parameter assignments are depicted in Tab. D.9.

Hypotheses

In general, the runtime of computing the pairwise input vector similarities should increase while increasing the embedding dimension. Ideally, the slope of this increase should be linear. The runtime of using the column-wise input data format should be smaller than the ones using the row-wise input data format for all embedding dimensions $m > 1$. This is expected due to the high importance of coalesced memory access regarding massively parallel hardware architectures.

Results

Appendix E.1.4 depicts the runtime results for each compute device. In contrast to the previous experiments, the runtimes captured are the mean of all of the 100 runs conducted for each experimental configuration.

Using the column-wise input data format results in runtimes that are either similar or lower than the runtimes of using the row-wise format. This holds except for an embedding dimension of one on the NVIDIA GeForce GTX 690 compute device. Furthermore, the runtime increase of the column-wise input data format is linear with respect to the embedding dimension across all compute devices employed. This does also hold for the runtimes regarding the row-wise input data format on the Intel Xeon E5620. Here, while increasing the embedding dimension, there is only a slight increase regarding the runtime ratio, whereas the slopes of the runtime ratio curves are much greater regarding the GPU devices (see Fig. 6.4).

To identify the cause of the supra-linear increase in runtime, the performance counters on the AMD Radeon RX 470 are investigated [Advanced Micro Devices, Inc., 2013a, pp. 15–17]. The `CacheHit` is expected to be the dominant impact factor, as suggested in Sect. 5.1.1. This is not confirmed by the results of this experiment. Surprisingly, the value of `CacheHit` is higher

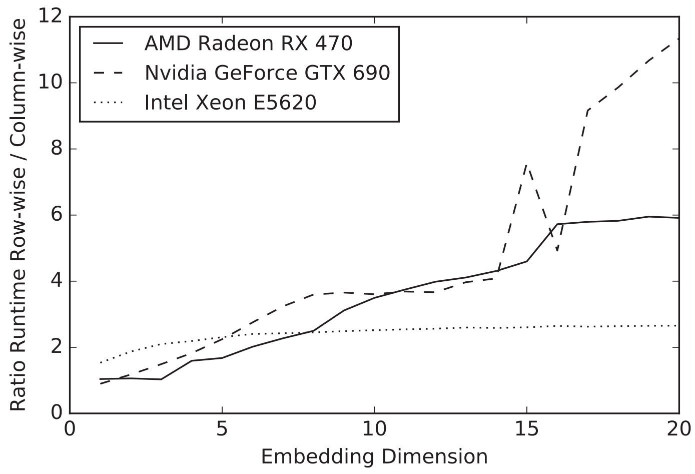


Figure 6.4: Input data format - Ratios between the runtime for using the row-wise versus the column-wise input data format.

regarding the row-wise input data format, increasing from 84.48% to 99.22%, in comparison to the column-wise layout, decreasing from 84.66% to 81.91% (see Tab. E.11).

The `FetchSize`, a performance counter capturing the total amount of data fetched from the global memory of the compute device, gives an indication for the increase in runtime while using the row-wise input data format. Regarding the column-wise layout, its value remains almost constant at approximately $157kB$, when varying the dimensionality of the input vectors from one to twenty. Using the row-wise format, the value of `FetchSize` grows from roughly $157kB$ ($m = 1$) to more than $2.6GB$ ($m = 20$), which is an increase of more than four magnitudes.

It is reasonable to assume, that the value of `FetchSize` also explains that the runtimes of the row-wise and column-wise layout are similar for embedding dimensions of $1 \leq m \leq 3$. Up to a dimensionality of three, the amount of data fetched does not exceed $512kB$ for both formats. This is equivalent to the total size of the L1 cache available on the AMD Radeon RX 470[1]. Having an amount of data that exceeds the size of the L1 cache, additional *evict* and *fetch* operations have to be performed. Apparently, this procedure increases the value of `FetchSize` drastically, which leads to a corresponding increase in runtime.

6.2.5 Recurrence Matrix Representation

This experiment evaluates the impact of selecting either an *uncompressed* or a *compressed* recurrence matrix representation (see Sect. 5.1.2). An uncompressed representation stores each result of the pairwise vector similarity comparisons, independent of its concrete value. The compressed sparse representation stores only the indices of those matrix elements that refer to recurrence points. Hence, the sparser the recurrence matrix, the fewer indices have to be stored.

[1]The Polaris 10 graphics processor mounted on the AMD Radeon RX 470 comprises 32 compute units [Burke, 2016], with each of them being equipped with $16kB$ of L1 cache. This results in $512kB$.

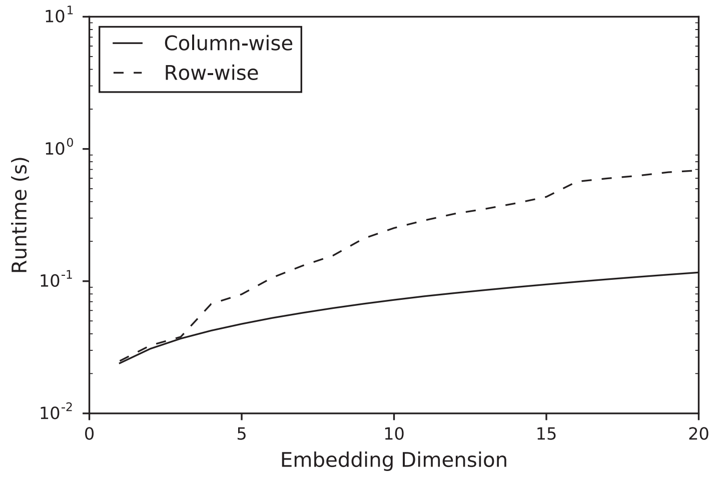


Figure 6.5: Input data format - AMD Radeon RX 470.

A separate parallel brute-force variant of the *create_recurrence_matrix* operator that represents the recurrence matrix in compressed format is not implemented. Rather, the Python packages SciPy and ScikitLearn are used to create recurrence matrices represented in the compressed sparse column format. Those compressed matrices are transferred to the global memory of the compute device and analysed in a massively parallel manner. As explained in Sect. 5.1.2, using the compressed representation requires to adapt the algorithms for detecting vertical and diagonal lines. As a major objective, the dependency of the performance of those algorithms regarding the sparseness of the recurrence matrix are investigated.

Setup

The sparseness of a recurrence matrix is influenced by several parameters, in particular the *similarity threshold*. Since it is cumbersome to determine the exact threshold achieving a certain recurrence rate, thresholds that are fractions of the maximum phase space diameter are selected. The ratios used range between 0.0 and 1.0 with an offset of 0.05. Applying the RQA input parameter assignments used in this experiment, the RQA measure recurrence rate, which reflects the sparseness of a recurrence matrix, is progressing as depicted in Fig. 6.6.

There is a super-linear increase in the number of recurrence points up to a recurrence rate of approximately 50%. This value is reached at a maximum phase space diameter ratio of 0.4. This progression results from the polynomial increase of the volume of the neighbourhoods around each input vector. The amount of neighbours is increasing likewise, due to the uniform distribution of input vectors.

Exceeding a maximum phase space diameter ratio of 0.4, the progression inverts until a recurrence rate of 100% is reached at a ratio between 0.70 and 0.75. The inversion can be explained by the fact, that the region of the m-dimensional space, in which the input vectors reside, is limited. With increasing the maximum phase space diameter ratio, a growing number of neighbourhoods exceed the limits of the fixed-sized region, leaving parts of the spheres empty.

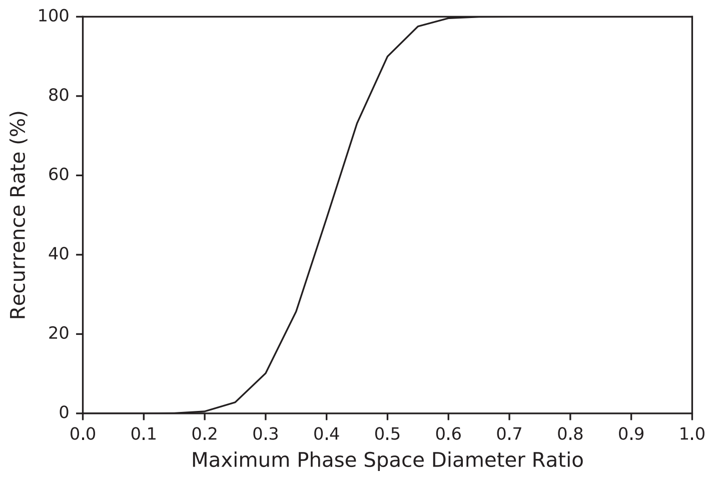


Figure 6.6: Recurrence rates. The progression of the recurrence rate results from applying a random input series adhering to the uniform distribution in combination with an embedding dimension of $m = 10$ and the Euclidean metric as similarity measure. The tabular results are captured in Tab. D.12.

This eventually results in a decline of the growth of the number of input points.

In this experiment, the maximum phase space diameter ratio is selected as independent variable. The runtimes of the *detect_vertical_lines* and *detect_diagonal_lines* operators are observed, while varying its value between 0.0 and 1.0. The properties of the RQA implementations used are depicted in Tab. D.10. Note that the RQA implementation based on the compressed sparse representation represents the coordinates of recurrence points as 32-bit integer values. The assignments of the full set of RQA input parameters are captured in Tab. D.11.

Hypotheses

An almost constant runtime across all maximum phase space diameter ratios is expected regarding the line detection using an uncompressed recurrence matrix representation. The reason for this assumption is that only the content of the recurrence matrix is changing, not the corresponding memory layout. Hence, the number of data elements evaluated remains constant in every case. This should hold regarding all compute devices, although the absolute runtimes should be specific to each device. Ideally, the runtimes of detecting diagonal lines should be lower than the ones of executing the *detect_vertical_lines* operator, because only one half of the matrix needs to be evaluated.

Using the compressed sparse representation, the runtime for line detection should grow with increasing maximum phase space diameter ratio. This results from the increasing number of recurrence points located in the columns and diagonals of the recurrence matrix. The *detect_vertical_lines* operator should perform well for small maximum phase space diameter ratios. This is due to the much smaller number of data elements evaluated during the sequential scan of matrix columns. There should be a break-even point regarding the runtimes of using the uncompressed and compressed matrix representation. Its location is expected to be specific to each compute device.

The implementation of the *detect_diagonal_lines* operator using the compressed representation should have higher runtimes in comparison to its uncompressed counterpart, since the underlying algorithm has a higher worst-case time complexity of $\mathcal{O}(N^3)$. Similar to the detection of vertical lines, the runtime deltas between the uncompressed and compressed implementation should differ across the compute devices.

Results

The experimental results, as shown in App. E.1.5, are only partially in alignment with the initial hypotheses. The set of confirmed hypotheses includes increasing runtimes regarding the detection of vertical lines using the compressed matrix representation. Furthermore, the compressed representation leads to higher runtimes regarding the detection of diagonal lines. In contrast to the initial hypotheses, the runtimes using the uncompressed representation experience a peak at specific maximum space diameter ratios. In addition, a continuous runtime increase using the compressed representation for detecting diagonal lines can only be observed regarding the Intel Xeon E5620 compute device. In the following, the divergent behaviour is distinguished in detail according to the two line detection operators.

Detection of Vertical Lines. Considering the runtime observations regarding the execution of the *detect_vertical_lines* operator, the following major deviations from the initial hypotheses exist:

1. The runtimes of using the uncompressed recurrence matrix representation are not constant.
2. The runtime curves of both representations intersect at almost the same maximum phase space diameter ratio.
3. The compressed representation results in lower runtimes regarding larger maximum phase space diameter ratios on the Intel Xeon E5620 compute device.

It was assumed that there is an almost constant runtime regarding the detection of vertical and diagonal lines while using the uncompressed matrix representation. This assumption does not hold, in particular not for the *detect_vertical_lines* operator. Within the ratio range from 0.2 until 0.6 an increase in runtime is observed across all compute devices. The runtime peak is either located at a ratio of 0.35 (Nvidia GeForce GTX 690) or 0.4 (AMD Radeon RX 470, Intel Xeon E5620).

Those elevations most likely result from using atomic operations to update the vertical line length histogram. The positions of the peaks correlate with a recurrence rate of approximately 50%. This density leads to a high number of lines having the same length, considering the uniform distribution of input vectors, which causes a high number of work-items to simultaneously update the same elements of the line length histogram. The execution of many work-items is blocked, due to the synchronised access. This eventually increases the overall runtime.

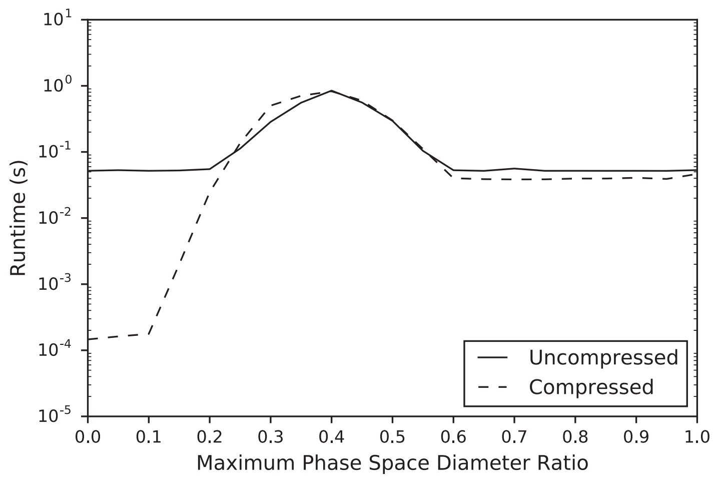


Figure 6.7: Recurrence matrix representation - Detect vertical lines - Intel Xeon E5620.

The negative impact of synchronisation affects both recurrence matrix representations and has the highest impact on the Intel compute device (see Fig. 6.7). Further increasing the recurrence rate, fewer lines of the same length are created, leading to a reduced synchronisation overhead. Independent of the compute device, the runtimes stabilise at a plateau starting at a maximum phase space diameter ratio of 0.6.

Using the compressed matrix representation, the runtimes correlate with the increase in recurrence rate across all compute devices. This leads to similar positions of the break-even points, in combination with similar runtime ratios with respect to the dense representation. The intersection is located between a maximum phase space diameter ratio of 0.2 and 0.3, across all compute devices.

On the Intel compute device, the implementation of the *detect_vertical_lines* operator using the compressed representation has lower runtimes than the one using the uncompressed representation for all ratios greater than or equal to 0.6, although these recurrence matrices contain large number of recurrence points. This behaviour may result from the diverging data formats used to represent the recurrence matrices in the global memory.

Detection of Diagonal Lines. The runtime results obtained regarding the execution of the *detect_diagonal_lines* operator are only partially in alignment with the initial hypotheses (see App. E.1.5). As expected, the runtimes while using the compressed representation are always higher, independent of the maximum phase space diameter ratio and compute device applied. The largest difference of more than one magnitude is observed on the Intel Xeon E5620 compute device.

Differently than expected, runtime peaks are observed around a maximum phase space diameter ratio of 0.4, using the uncompressed representation. This behaviour likely stems from the synchronised access to the diagonal line length histogram, similar to the detection of vertical lines. The impact of the compressed recurrence matrix representation is specific to each compute device. Whereas the runtimes regarding the AMD GPU remains almost constant, there is

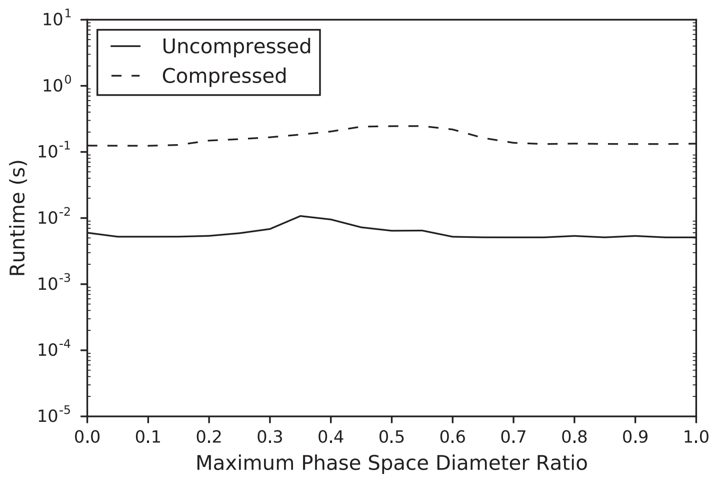


Figure 6.8: Recurrence matrix representation - Detect diagonal lines - Nvidia GeForce GTX 690.

a peak at a ratio of 0.55 regarding the Nvidia GeForce GTX 690 (see Fig. 6.8). In contrast, the progression of the runtime curve regarding the CPU device corresponds to the recurrence rates from Fig. 6.6.

6.2.6 Similarity Value Representation

The following experiment refers to the representation of similarity values within uncompressed recurrence matrices, as explained in Sect. 5.1.3. Either one *byte* or *bit* is used to represent the binary similarity value referring to a single pair of multi-dimensional vectors. In this regard, this thesis introduces a custom bit-wise recurrence matrix representation. The bit representation uses atomic operations to update individual bits of 32-bit integer values. Only bits that refer to recurrence points are modified during recurrence matrix construction, to reduce the negative impact of synchronisation.

The representation of the similarity values influences the processing of each analytical RQA operator, including the detection of vertical and diagonal lines. The bit values have to be extracted from the 32-bit integer values, to inspect a column or diagonal of the recurrence matrix. In this regard, multiple bits of the same integer value can be extracted simultaneously without having to perform synchronisation. The following experiment compares the performance characteristics of executing all three analytical RQA operators, either using the bit-wise or byte-wise representation of similarity values.

Setup

The embedding dimension is selected as independent variable, which is varied between one and twenty. A single series adhering to a uniform distribution of floating-point values is used as input data. A constant ratio regarding the maximum phase space diameter is used to specify the similarity threshold. This leads to decreasing recurrence rates, with an increasing embedding

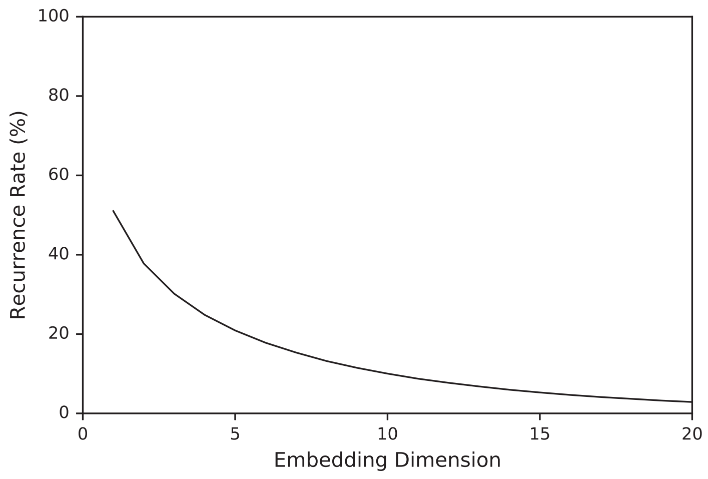


Figure 6.9: The development of the recurrence rate using a uniform distribution of input values in combination with a fixed maximum phase space diameter ratio of 0.3, while increasing the embedding dimension from one to twenty.

dimension (see Fig. 6.9). The properties of the analytical operator implementations is presented in Tab. D.13. The input parameter assignments are shown in Tab. D.14.

Hypotheses

The computational load regarding the calculation of the pairwise vector similarities grows when increasing the embedding dimension. This results from processing more pairs of input vector components. Hence, there should be an increase in runtime for executing the *create_recurrence_matrix* operator, independent of the similarity value representation. The recurrence rate drops gradually with increasing embedding dimension, while starting from a value of more than 50%. This should result in lower runtimes regarding small embedding dimensions when using the byte-wise representation, in comparison to the representing a similarity value by a single bit. The synchronisation overhead regarding the latter should mitigate with increasing dimensionality of the input vectors.

The runtimes for detecting vertical lines should remain constant, independent of the embedding dimension and similarity value representation. This is due to the white vertical lines being the inverse of the vertical lines consisting of recurrence points. As a result, the performance improvements of having to update the vertical line length histogram fewer times is compensated by an increasing number of updates to the white vertical line length histogram. In contrast, the runtime for detecting diagonal lines should decrease with an increasing embedding dimension, since fewer updates of the corresponding line length histogram have to be performed.

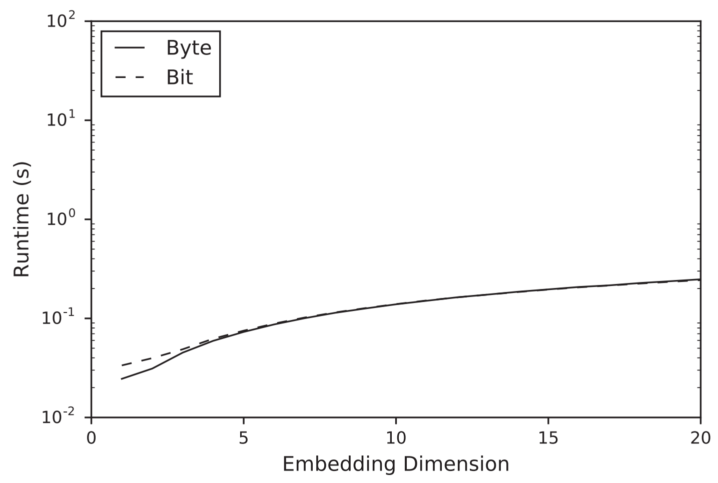


Figure 6.10: Similarity value representation - Create recurrence matrix - Nvidia GeForce GTX 690.

Results

The runtime results of the experiment are depicted in tabular fashion as well as using diagram in App. E.1.6. They are largely in alignment with the initial hypotheses regarding the creation of the recurrence matrix and the detection of diagonal lines. This particularly refers to increasing runtimes during the *create_recurrence_matrix* operator execution and decreasing runtimes during the *detect_diagonal_lines* operator execution. Diverging behaviour can be observed regarding the detection of vertical lines, where the runtime decreases instead of remaining constant across all embedding dimensions. In the following, the runtime results are analysed separately for each analytical operator.

Creation of the Recurrence Matrix. The runtimes observed regarding the execution of the *create_recurrence_matrix* operator grow with increasing embedding dimension, independent of the similarity value representation. For small embedding dimensions, the runtimes using the bit-wise similarity value representation are higher than the ones using the byte-wise representation. Considering the AMD and the Intel compute devices, the runtimes using the byte-wise representation are smaller with respect to the bit-wise representation (see Fig. E.19 and Fig. E.25). This behaviour is observed independent of the concrete embedding dimension used. Furthermore, the two runtime curves converge with increasing vector dimensionality. In contrast, using the bit-wise representation leads to slightly smaller runtimes on the Nvidia GPU device for embedding dimensions with $m \geq 12$ (see Fig. 6.10).

Detection of Vertical Lines. Regarding the *detect_vertical_lines* operator it was assumed that the runtimes remain constant, independent of the similarity value representation employed. The runtimes observed do not confirm this hypothesis. Instead, the runtimes decrease with increasing vector dimensionality across all compute devices. This behaviour likely results from counting the

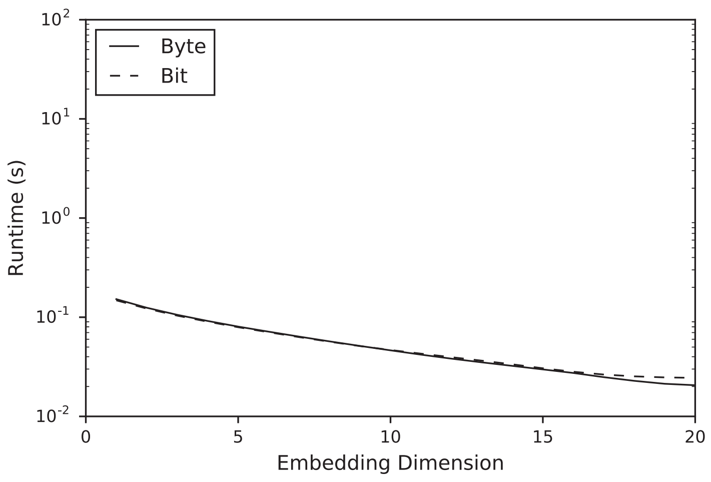


Figure 6.11: Similarity value representation - Detect vertical lines - AMD Radeon RX 470.

number of recurrence points, required for the computation of the recurrence rate, per column of the recurrence matrix during the detection of vertical lines. In this regard, one integer variable is kept per column, that is incremented while scanning the corresponding matrix elements. This implementation detail allows to avoid the application of atomic operations, in particular during the execution of the *create_recurrence_matrix* operator. Since fewer increment operations have to be performed, as indicated by Fig. 6.9, the computational effort decreases with increasing embedding dimension.

The runtimes of using the byte-wise or bit-wise similarity value representation almost completely overlap, independent of the compute devices employed. An exception is the AMD Radeon RX 470. Here, the runtimes using the bit-wise representation are slightly smaller for all embedding dimensions with $m < 10$. Regarding $m \geq 10$, applying the byte-wise representation results in lower runtimes. The gap between the two runtime curves increases with increasing embedding dimension (see Fig. 6.11).

Detection of Diagonal Lines. Regarding the detection of diagonal lines, only the runtimes regarding the Intel Xeon E5620 compute device are in alignment with the initial hypothesis (see Fig. 6.12). Here, the runtime decreases continuously while increasing the embedding dimension. In contrast, there are runtime peaks at a dimensionality of two, considering the two GPU devices. The performance characteristics of the AMD Radeon RX 470 and the Nvidia GeForce GTX 690 differ regarding the relation between the runtime curves of both similarity value representations. Executing the *detect_diagonal_lines* operator on the AMD compute device, using the bit-wise representation results in lower runtimes for embedding dimensions that are greater or equal than six. In contrast, the byte-wise representation is favourable regarding the Nvidia compute device, independent of the embedding dimension applied.

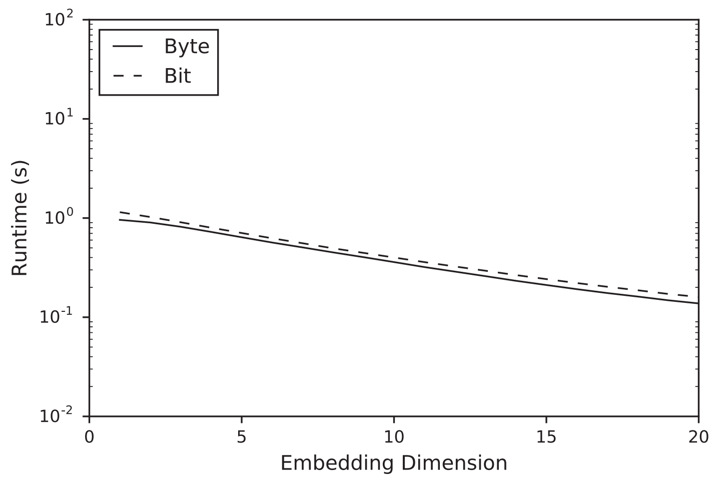


Figure 6.12: Similarity value representation - Detect diagonal lines - Intel Xeon E5620.

Summary. The effects of representing similarity values using a single bit or byte differs between the compute devices employed. Although the bit-wise representation introduces computational overhead, corresponding RQA operator implementations may have a better performance regarding sparse recurrence matrices, e.g., regarding the execution of the *create_recurrence_matrix* operator on the Nvidia GPU. Furthermore, there may be diverging behaviour when comparing the execution of the different analytical operators on the same compute device. As an example, the byte-wise representation is favourable while detecting vertical lines on the AMD GPU, whereas the bit-wise representation is favourable regarding the detection of diagonal lines.

6.2.7 Intermediate Results Recycling

The recycling of intermediate results refers to including the computations of the pairwise vector similarities within the *detect_vertical_lines* operator (see Sect. 5.1.4). The corresponding binary similarity values are persisted within the global memory of the compute device and used during the detection of diagonal lines.

A major benefit of recycling is saving the overhead for executing a separate operator to create the recurrence matrix. Moreover, computing the pairwise similarities during the inspection of columns allows to eliminate the reading of N^2 similarity values. On the downside, each atomic task instance of the *detect_vertical_lines* operator has to compute N binary similarity values, instead of only one. This limits the potential for parallel processing, although the total amount of computational effort remains the same. The goal of this experiment is to investigate whether there are conditions, under which the benefits of recycling exceed its disadvantages.

Setup

The suitability of recycling is strongly influenced by the number of input vectors, from which the corresponding recurrence matrix is constructed. Changing the value of this parameter allows to steer the total number of work-items executed per operator. During the experiment, the number of input vectors N is varied between $1,000$ and $20,000$ with an offset of $1,000$. Increasing N linearly leads a quadratic increase in the number of similarity comparisons and a linear increase in vertical line detection tasks.

A one-dimensional series capturing values that adhere to a uniform distribution serves as input data. The density of the recurrence matrix processed is set to roughly 50%, by selecting a maximum phase space diameter ratio of 0.4. The remaining input parameter assignments are presented in Tab. D.16.

The runtimes of the *create_recurrence_matrix* and *detect_vertical_lines* operator of the non-recycling RQA implementation are cumulated. This ensure comparability with the operator *detect_vertical_lines* using recycling. The detection of diagonal lines is excluded from the performance analysis, because the corresponding computations are identical in both cases. The remaining operator properties are captured in Tab. D.15.

Hypotheses

The performance behaviour of the recycling and the non-recycling implementation is likely to be specific to each computing device. This is for example due to the different parallel processing capabilities, e.g., the total number of processing elements. One of the following three runtime behaviours is expected to be observed:

The benefits of recycling outweigh its disadvantages: The elimination of the separate operator *create_recurrence_matrix* leads to a reduction in runtime, independent of the size of the recurrence matrix. The access to the global memory of the compute device is very time consuming. The overhead for initialising and executing separate work-items regarding *create_recurrence_matrix* is considerable. The larger amount of work performed by each work-item does not have considerable effect.

The disadvantages of recycling outweigh its benefits: Leveraging the full parallel processing capabilities of the compute device is the key to a reasonable runtime performance. The access to the global memory is very cheap from a computational perspective. The overhead for work-item initialisation and execution is comparatively small. The amount of work performed by each work-item has a significant influence on the runtime.

Neither the benefits nor the disadvantages of recycling are dominant: There exists a break-even point regarding the runtime curves of both approaches, depending on the amount of input vectors.

Results

Appendix E.1.7 compares runtime performance of using either the recycling and non-recycling approach. As stated before, the runtimes referring to the non-recycling approach are a cumulation of the individual runtimes of the *create_recurrence_matrix* and the *detect_vertical_lines* operator. There is no consistent runtime behaviour across the three compute devices used during the evaluation. On a macro level, the performance of the two GPU compute devices is comparable, whereas the CPU exposes diverging runtime results. Therefore, the analysis is conducted separately for each compute device type, GPU and CPU.

Device Type GPU. There exist break-even points regarding the two runtime curves for the AMD Radeon RX 470 and the Nvidia GeForce GTX 690. The non-recycling approach performs well for small amounts of vectors, starting with comparatively lower runtimes for $1,000$ input vectors. Apart from that, the runtimes regarding the non-recycling approach grow stronger than the runtimes referring to recycling. The location of the resulting intersection of both curves differs between the two devices. Considering the AMD GPU, the break-even point lies between $11,000$ and $12,000$ input vectors. The runtime curves of the Nvidia GPU intersect between $4,000$ and $5,000$ vectors.

Performance counters regarding the execution on the AMD GPU are gathered, to investigate the reason for this runtime behaviour. As indicated before, a benefit of intermediate results recycling is omitting read accesses to the global memory of the compute device, regarding N^2 similarity values. Correspondingly, this affects the amount of data transferred from and to the global memory, measured by `FetchSize` and `WriteSize` [Advanced Micro Devices, Inc., 2013a, pp. 16–17].

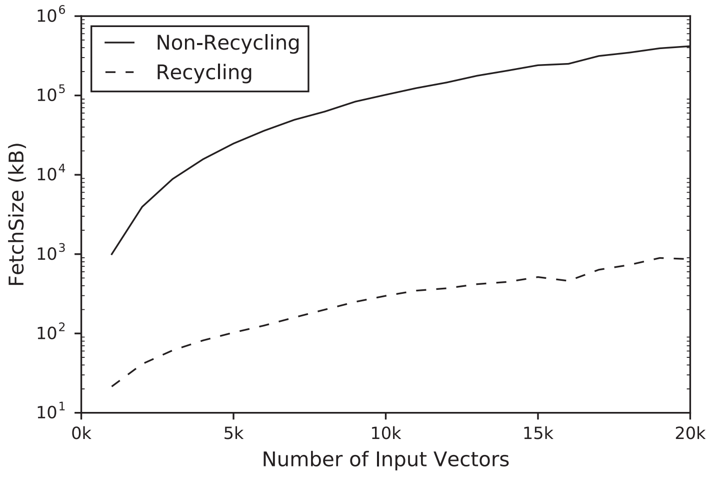


Figure 6.13: Intermediate results recycling - AMD Radeon RX 470 - `FetchSize`. The `FetchSize` values referring to the non-recycling approach are an aggregate of the individual values of the *create_recurrence_matrix* and *detect_vertical_lines* operator. The tabular data is shown in Tab. E.30.

The progression of `FetchSize` is depicted in Fig. 6.13. The amount of data fetched from the global memory differs greatly, whether recycling is applied or not. Only the N vectors serving as input data are read during the execution of the *detect_vertical_lines* operator using recycling. The non-recycling approach requires to additionally read the quadratic recurrence matrix, containing N^2 binary similarity values. This quadratic relationship with respect to the

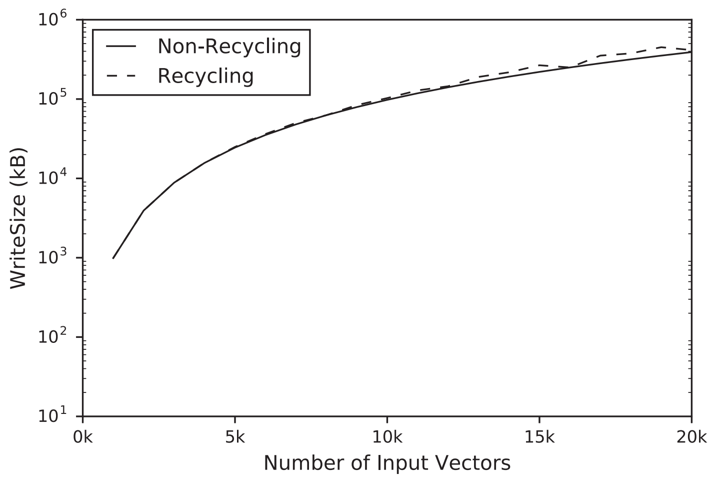


Figure 6.14: Intermediate results recycling - AMD Radeon RX 470 - `WriteSize`. The `WriteSize` values referring to the non-recycling approach are an aggregate of the individual values of the *create_recurrence_matrix* and *detect_vertical_lines* operator. The tabular data is shown in Tab. E.30.

number of input vectors is also reflected by the quadratic increase in the amount of data fetched from global memory.

The value of the performance counter `WriteSize` is almost independent of intermediate results recycling (see Fig. 6.14). This results from persisting the N^2 binary results of the similarity comparisons regarding each of the two cases. Hence, the amount of data written to the global memory is comparable.

Device Type CPU. Applying recycling on the Intel Xeon E5620 CPU results in larger runtimes (see Fig. 6.15). The corresponding runtime curve progresses more steeply than the non-recycling counterpart, independent of the number of input vectors used. The amount of data fetched from the global memory does not seem to have considerable effect on the runtime. It is reasonable to assume that the higher degree of parallelism of the *create_recurrence_matrix* operator suits the processing on the Intel CPU.

6.2.8 Recurrence Matrix Materialisation

All of the previous experiments assume that the recurrence matrix is stored in the global memory of the compute device. Either the *create_recurrence_matrix* or the *detect_vertical_lines* operator computes its contents, which are used to detect line structures. The following experiment investigates the impact of not-persisting the matrix at all, but rather to compute the binary similarity values while inspecting columns and diagonals regarding line structures.

The theoretical considerations referring to this experiment are presented in Sect. 5.1.5. The focus is on a set of equations that capture the impact of the amount of data transferred from and to the global memory on the overall runtime of RQA (see App. B).

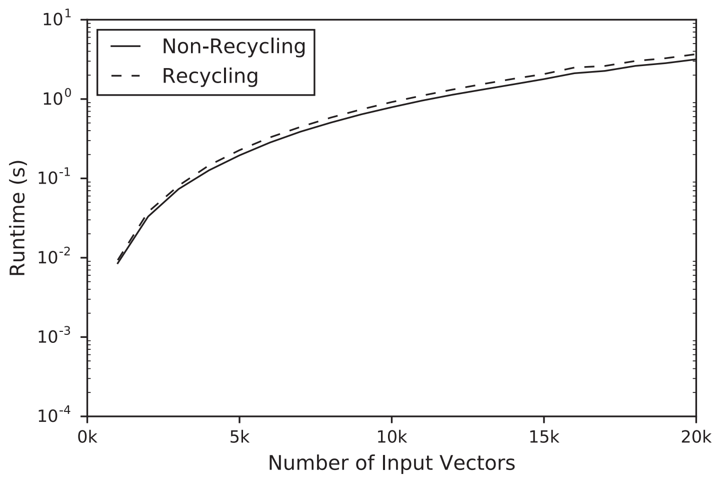


Figure 6.15: Intermediate results recycling - Intel Xeon E5620. The runtimes referring to the non-recycling approach are an aggregate of the individual runtimes of the *create_recurrence_matrix* and *detect_vertical_lines* operator.

Setup

The dimensionality of the input vectors is expected to have a large influence on the runtime performance of implementations relying on non-materialisation, as indicated by the equations in App. B. Hence, the embedding dimension is selected as independent variable. Its value is varied between one and twenty with an offset of one.

The influence of other factors potentially distorting the runtime results is reduced, to investigate solely the impact of the embedding dimenson. As an example, the maximum phase space diameter is set to 0.0, which results in a recurrence matrix containing only zero values. In this way, the computational effort to update the line length histograms is minimised. The assignments of the remaining input parameters is depicted in Tab. D.18.

Each of the analytical RQA operators is influenced by the choice of materialising or not-materialising the recurrence matrix. Hence, the cumulative runtime of all RQA operators is measured. Note that the implementation that materialises the recurrence matrix only consists of the operators for detecting vertical and diagonal lines. The operator properties except from the materialisation status are depicted in Tab. D.17. Note that the values of the categories *similarity value representation* and *intermediate results recycling* do only apply to the implementation that persists the binary similarity values.

Hypotheses

The equations presented in App. B compare the amount of data elements transferred when applying different materialisation and non-materialisation approaches. The assumption is that the number of data elements transferred from and to the global memory of a compute device has significant impact on the runtime. The equations are solved for the embedding dimension m, having a high influence on the amount of data transferred. This experiment compares using

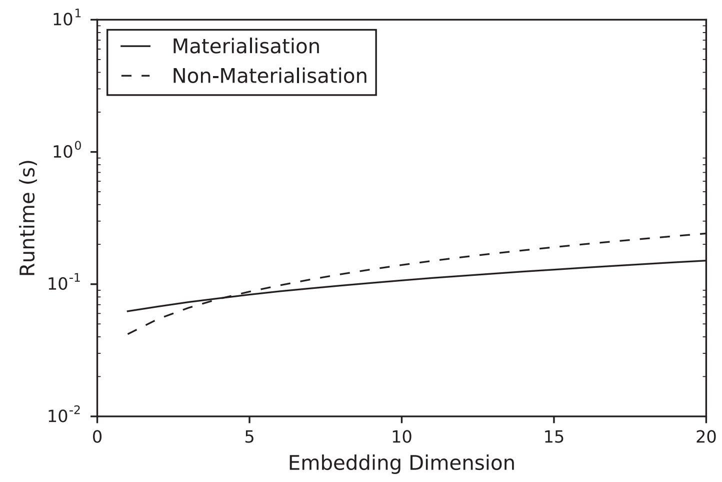


Figure 6.16: Recurrence matrix materialisation - AMD Radeon RX 470.

a separate *create_recurrence_matrix* operator with not materialising a symmetric recurrence matrix. The corresponding equation that is depicted in App. B.1 indicates a break-even point at an embedding dimension between two and three. For $m \leq 2$, not-materialising the recurrence matrix results in fewer data elements transferred. For $m \geq 3$, not-materialising the recurrence matrix results in more data elements transferred. The development of the corresponding runtime curves should correlate with these theoretical considerations. This should hold, independent of the compute device used.

Results

The comprehensive results regarding all compute device employed are presented in App. E.1.8. They vastly support the initial hypotheses. The runtime increases almost linearly, both for materialising and not-materialising the recurrence matrices. However, the slope of the runtime curves referring to non-materialisation are steeper compared to materialising the matrix, causing the break-even points. The progression of the runtime curves is similar across all compute devices. As an example, the execution on the AMD GPU is considered (see Fig. 6.16). For $1 \geq m \geq 3$, the runtime regarding the non-materialisation approach is smaller than its counterpart referring to materialisation. This behaviour inverts for $4 \geq m \geq 20$.

The position of the break-even point with respect to the embedding dimension m is device-specific.

AMD Radeon RX 470: $3 < m < 4$

Nvidia GeForce GTX 690: $5 < m < 6$

Intel Xeon E5620: $4 < m < 5$

The comparatively small values support the equation from App. B.1. Deviations from the theoretical results may stem from multiple reasons. First, the equations presented in App. B assume that read accesses and write accesses have the same computational overhead. However, write operations are usually more demanding than read operations. Second, the number of data elements considered in the equations does not match the actual amount of data transferred. This heavily depends on the encoding of particular data elements, e.g., the binary similarity values. Third, not-materialising the recurrence matrix allows to save overhead costs for executing a separate *create_recurrence_matrix* operator. This may also have a positive impact on the runtime, which is expected to be specific for each compute device.

6.3 Index Data Structures

This section evaluates the performance of index data structure implementations in the context of RQA processing. Each of the following subsections refers to a specific type of index data structures, either grid directories or multi-dimensional search trees. Each subsection comprises a single experiment, focussing on the the execution of the *create_recurrence_matrix* operator. It is investigated, how the pruning of pairwise similarity comparisons influences the performance characteristics of corresponding operator implementations.

6.3.1 Grid Directories

This section investigates the applicability of grid directories to RQA (see Sect. 5.2.1). In this regard, the multi-dimensional vectors extracted from the input time series are indexed using an uniform grid. This grid is used to prune similarity comparisons by investigating only those grid cells that are adjacent to the source grid cell of a query vector. The process of employing a grid directory is subdivided into two steps:

1. Creating the grid directory, and
2. Querying the grid directory regarding neighbouring vectors.

Both of those steps are performed while executing the *create_recurrence_matrix* operator. Two strategies to create grid directories have been presented in Sect. 5.2.1. This includes grid creation using:

- Atomic operations, and
- Sorting.

Implementations based on the OpenCL framework are provided for each of those approaches. Note that the approach using sorting requires to transfer the mapping of vector index to grid cell index from the global memory of the compute device to the main memory of a computing system. The corresponding *create_recurrence_matrix* implementation conducts the sorting of the indices on the host device. This is done to avoid additional effort to implement parallel sorting. Instead, functionality provided by the Python package *NumPy* is employed. Here, the functions `numpy.argsort`, `numpy.bincount` and `numpy.cumsum` are called.

The goal of using grid directories is to reduce the number of pairwise similarity comparisons by investigating only those grid cells that potentially contain neighbours of a query vector. For this purpose, a uniform grid with an edge length of $2*\epsilon$ in each of the m dimensions, with ϵ being the similarity threshold, is applied. As a result, 3^m grid cells have to be inspected regarding neighbours for each multi-dimensional vector. This relation indicates that the number of grid cells to inspect increases exponentially when increasing the embedding dimension linearly. For this reason, the experiment evaluates the impact of the following parameters on the performance characteristics of the grid directory implementations:

- Embedding dimension,

- Similarity threshold, and
- Spatial distribution of input vectors.

Setup

The experiment is conducted solely on computing system *(A)* (see App. C.3). The AMD Radeon RX 470 GPU provides the largest parallel computing capabilities, among the set of OpenCL compute devices available. In addition, the global memory of 8GB is larger than the memory of a single Nvidia GeForce GTX 690 processor contained in computing system *(B)*.

Initial tests have shown, that this amount of memory is sufficient to store grid directories referring to roughly 10^3 multi-dimensional vectors. This limitation results from the approach using atomic operations. Here, the amount of memory each grid cell occupies is fixed. Instead of using a heuristic regarding the potential neighbours to store, the size of the corresponding memory region is selected such that each grid cell can potentially store all vector indices from the dataset. This property amplifies the impact of the exponential increase regarding the number of grid cells. On the positive side, this limitation restricts the computational overhead performed by the host device.

The embedding dimension is varied only between 1 and 15, to address the memory limitation described above. Again, a fraction of the maximum phase space diameter is selected as similarity threshold. The ratio ranges from 0.1 to 1.0 with an offset of 0.1. The value 0 is left out, because it would lead to grid cell edge lengths of 0. Selecting a ratio smaller than 0.1 would also increase the total number of grid cells, overstraining the global memory of the compute device. The vectors are scattered in multi-dimensional space either using the uniform, normal, exponential and Cauchy distribution. The utilisation of synthetic data guarantees a certain spatial distribution of input vectors.

A fixed number of ten instances are created for each combination of embedding dimension and distribution. The full range of maximum phase space diameter ratios is applied to each of those instances, leading to different total numbers of grid cells. The following three *create_recurrence_matrix* implementations are employed during the experiment:

1. Parallel brute-force implementation,
2. Parallel grid directory implementation using atomic operations, and
3. Parallel grid directory implementation using sorting.

All of those implementations rely on OpenCL functionality and share common properties as shown in Tab. D.19. The implementations differ only regarding the procedure to obtain the contents of the recurrence matrix. The assignments of the input parameters that are kept constant are presented in Tab. D.20.

Hypotheses

The set of hypotheses considers multiple aspects of the performance analysis. Each of them is captured individually.

Parallel Brute-Force Implementation. The parallel brute-force implementation is expected to be insensitive to the spatial distribution of input vectors, as indicated by the experiment in Sect. 6.2.1. This should lead to comparable runtimes across all vector distribution types employed. The runtimes of the implementation selected should further not depend on the amount of recurrence points, since an uncompressed recurrence matrix representation is applied. Hence, the spreads between maximum and minimum runtimes should be very narrow. In addition, the runtime should increase linearly, while linearly increasing the dimensonality m of the input vectors.

Grid Directory Implementations. As explained earlier, 3^m grid cells have to be investigated regarding potential neighbours of a query vector. Among other aspects, this exponential increase contributes to the curse of dimensionality. Each combination of vector and grid cell is represented by a single OpenCL work-item during the computation of the pairwise similarities. $3^m * N$ work-items are created, with N being the total number of multi-dimensional vectors, independent of whether the grid directory is created using atomic operations or sorting. Each of those work-items introduces additional overhead for initialisation and execution. As a consequence, the runtimes are expected to increase roughly according to $\mathcal{O}(N^3)$.

Creating the grid directory using sorting should yield in higher runtimes compared to using atomic operations. This assumption is based on the increased effort to sort the indices, including data transfer from and to the global memory as well as computations conducted by the host device.

The efficiency of the pruning similarity comparisons largely depends on the spatial distribution of the input vectors, as indicated before. It should decrease in the order of uniform, normal, exponential and Cauchy distribution. Increasing the similarity threshold increases the number of similarity comparisons actually conducted, although the amount of grid cells inspected for each input vectors stays the same.

Parallel Brute-Force vs. Grid Directory Implementations. The two grid directory implementations are expected to deliver competitive runtime results with respect to the parallel brute-force approach for small embedding dimensions. This is due to performing computations in a massively parallel manner using OpenCL. The exponential increase in work-items should lead to a drastic increase in runtime considering larger embedding dimensions.

Results

The experimental results are depicted in a comprehensive manner in App. E.2.1. They are partially in alignment with the initial hypotheses. This includes the execution of the parallel brute-force and the grid directory implementations. The results of the experiment are subdivided according to the hypotheses presented before.

Parallel Brute-Force Implementation. The runtime of executing the parallel brute-force implementation of the *create_recurrence_matrix* operator increases, depending on the embedding dimension (see Fig. 6.17). Contradicting the initial hypothesis, this increase is only partially

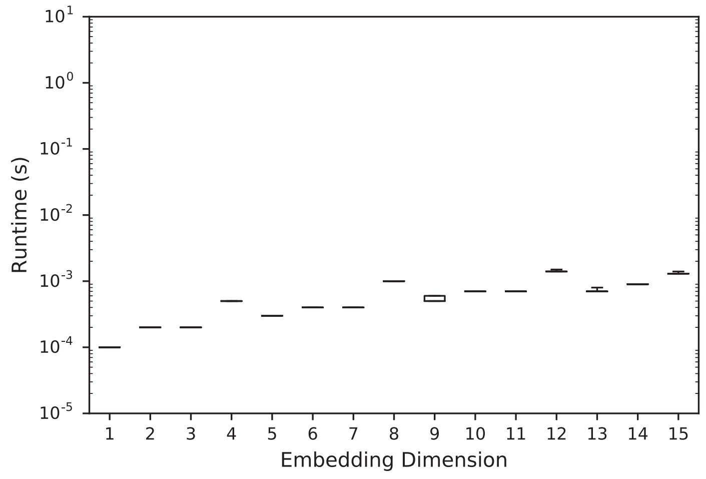


Figure 6.17: Grid directories - Parallel brute-force implementation on exponential distribution.

linear. Runtime peaks at embedding dimensions referring to multiples of four ($m = 4$, $m = 8$ and $m = 12$) are observed. It is reasonable to assume that this behaviour results from specific architectural properties of the AMD compute device. Nonetheless, there are only small spreads between the minimum and maximum runtimes, independent of the dimensionality of the input vectors.

The evolution of runtimes is comparable across all types of input vector distribution. The runtimes for the same combinations of quantile and embedding dimension are almost identical. This also includes the position as well as the extent of the runtime peaks.

Grid Directory Implementations. The two grid directory implementations used in this experiment are sensitive regarding the spatial distribution of input vectors as well as the similarity threshold employed. In Fig. 6.18, the runtimes of the grid directory implementation using atomic operations regarding the uniform distribution are shown. A boxplot is displayed for each embedding dimension. The upper and lower whiskers refer to the 10% and 90% quantiles. The outliers are depicted using the symbol $+$.

The runtimes referring to the maximum phase space diameter ratio of 0.1 are of particular interest, because they expose diverging behaviour. For embedding dimensions with $1 \geq m \geq 7$ they represent the lower 10% of the runtimes collected. In contrast, for embedding dimensions with $9 \geq m \geq 15$ they represent the top 10% runtimes. The runtimes drop to almost the average runtime, when increasing the maximum phase space diameter ratio to 0.2. This leads to two conclusions. First, the benefit of having smaller grid cells outweighs the exponential growth regarding their amount for all embedding dimensions with $m < 8$. This behaviour inverses for all embedding dimensions with $m > 8$. Second, a further decrease in the maximum phase space diameter ratio to a value greater than 0.2 does not have a considerable impact on the runtime. Note that this behaviour might differ using other OpenCL compute devices.

The diverging behaviour regarding a maximum phase space diameter ratio of 0.1 is observed across all distribution types applied. Nonetheless, the intensity of the variations differs among

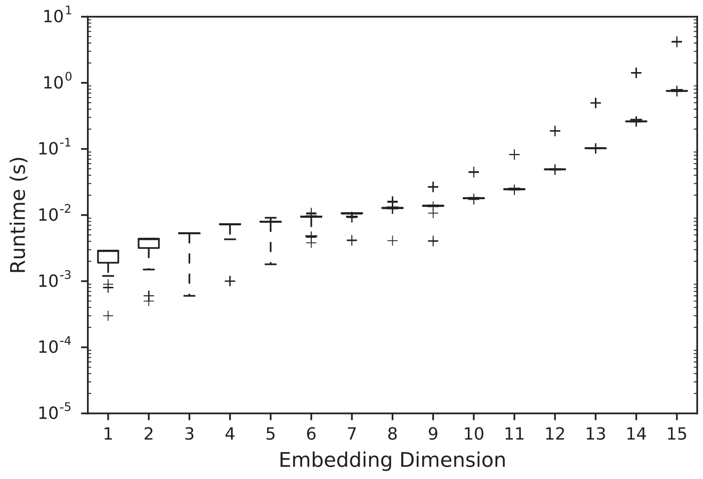


Figure 6.18: Grid directories - Grid directory implementation using atomic operations on uniform distribution.

Table 6.1: Grid directories - AMD Radeon RX 470 - Performance counters. The counters reflect average values out of ten experimental runs.

Distribution	`Time` (ms)	`VALUInsts`	*Ratio*
Uniform	4092.00	393.44	10.40
Normal	2651.47	249.01	10.65
Exponential	1676.92	150.78	11.12
Cauchy	771.48	77.57	9.95

the distribution types. Considering a uniform distribution of multi-dimensional vectors, it is the highest. Considering the Cauchy distribution, it is the lowest. This contradicts the expected behaviour regarding the pruning of similarity comparisons. Additional performance counters regarding an embedding dimension $m = 15$ and a maximum phase space diameter ratio of 0.1 are obtained for further investigation.

A strong correlation between the average number of *vector arithmetic logic unit instructions* and the total runtime of executing the *create_recurrence_matrix* operator is observed [Advanced Micro Devices, Inc., 2013a, p. 17]. In the following, the performance counter `Time` is compared to `VALUInsts`. The runtime is roughly ten times the size of `VALUInsts`, independent of the distribution type (see Tab. 6.1). Vector instructions are particularly executed when computing the pairwise similarities between a query vector and the vectors in adjacent grid cells. Therefore, it is reasonable to assume that concentrating the multi-dimensional vectors in regions with small extent results in a larger number of work-items investigating empty neighbouring grid cells. This reduces the average number of instructions executed per work-item.

The progression of the 50% quantile runtimes of the two grid directory implementations while

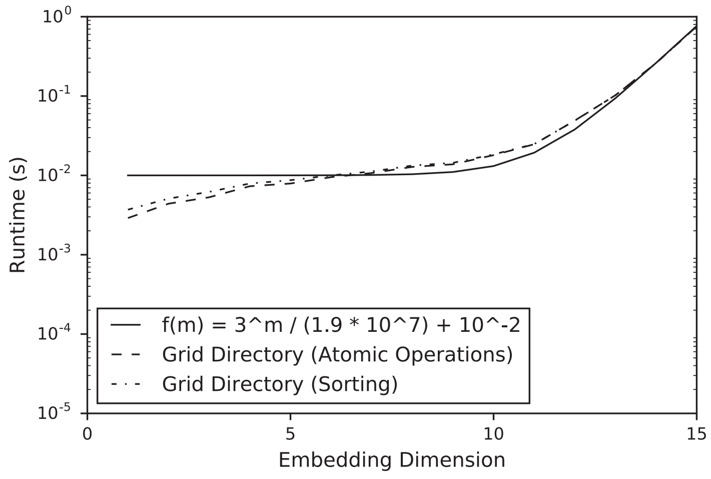


Figure 6.19: Grid directories - Grid directory implementations on normal distribution.

increasing the embedding dimension is in alignment with the exponential increase in the total number of grid cells. The runtime curves presented in Fig. 6.19 confirm the initial hypothesis. A fitted curve $f(m)$ is displayed to emphasise the exponential relationship. It can be concluded that the pairwise vector similarity comparisons are not the major impact factor to the overall runtime. It is rather the exponentially growing number of grid cells that have to be inspected regarding neighbouring vectors.

Parallel Brute-Force vs. Grid Directory Implementations. The average runtimes of the parallel brute-force implementation are at least one magnitude smaller than the runtimes of the parallel grid directory implementations (see Fig. 6.20). This contradicts the initial assumption that suggested lower runtimes of the grid directory implementations regarding small embedding dimensions. The runtime difference between the parallel brute-force and the two grid directory implementations increases up to more than two magnitudes.

Overall, the grid directory implementations suffer from higher runtimes in comparison to the brute-force implementation, although they also leverage the massively parallel computing capabilities of the AMD Radeon RX 470. This is independent of the combination of embedding dimension and type of spatial distribution.

6.3.2 Multi-Dimensional Search Trees

This section comprises a single experiment that investigates the applicability of tree-based index data structures to RQA. The focus is on two implementations of the k-d tree as presented in Sect. 5.2.2. This includes:

- `scipy.spatial.cKDTree` as part of the Python package *Scipy*[2] and

[2]The function `scipy.spatial.cKDTree.query_ball_tree` is applied to conduct ϵ-nearest neighbour queries.

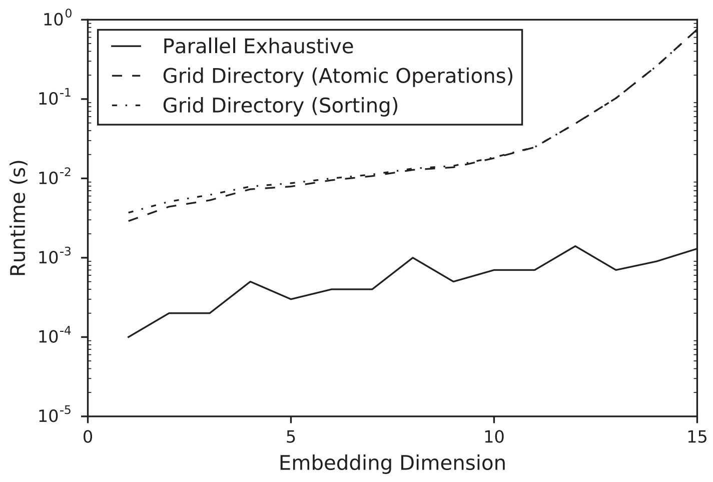


Figure 6.20: Runtimes - Normal Distribution.

- `sklearn.neighbors.KDTree` as part of the Python package *scikit-learn*[3].

The functionality required in the context of RQA processing is the creation of the structure and its querying regarding nearest neighbours. Insights on a prior performance analysis are given in Sect. 5.2.2. It mentions several factors that affect the runtime performance of those k-d tree implementations. Here, the impact of the distribution type and the dimensionality of the input vectors organised within the tree are investigated further, since they heavily influence the efficiency of the pruning of similarity comparisons.

Setup

Similar to grid directories, the multi-dimensional tree data structures are used during the construction of a recurrence matrix. Hence, only their impact on the performance of the *create_recurrence_matrix* operator is analysed. The runtime of a parallel exhaustive implementation is used as a *baseline.*

The execution of the k-d tree implementations is restricted to CPUs. As a consequence, all implementations are executed solely on the Intel Xeon E5620 compute device that is part of computing system *(C).* This includes a parallel brute-force implementation representing the baseline. It ensures the comparability of the runtime results. Nonetheless, the baseline implementation is capable of using the computing capabilities of all sixteen CPU threads available, whereas the execution of `cKDTree` and `KDTree` is limited to a single CPU thread.

The independent variables, including the embedding dimension, the spatial distribution of the multi-dimensional vectors and the similarity threshold, whose impact is observed in this experiment as well as the corresponding value ranges are similar to the setup in Sect. 6.3.1.

[3]The function `sklearn.neighbors.NearestNeighbors.radius_neighbors_graph` is applied to conduct ϵ-nearest neighbour queries.

The latter is especially applied to steer the density of the recurrence matrix. Apart from those assignments, the full set of embedding dimensions is applied, ranging from one to twenty.

The implementations of the *create_recurrence_matrix* operator relying on the k-d tree represent the resulting recurrence matrix in compressed sparse format. Implementations that transform this compressed into an uncompressed matrix representation are excluded from the evaluation, since they do only introduce conversion overhead that distracts from the computation of the pairwise input vector similarities. The runtimes of the k-d tree implementations are compared to a parallel brute-force implementation. Its properties are depicted in Tab. D.21.

Initial experiments have shown that the k-d tree implementations deliver considerable higher runtimes than using parallel brute-force processing. Hence, the number of multi-dimensional vectors that are compared to each other regarding their mutual similarity is restricted to 10^3, which still results in a recurrence matrix consisting of 10^6 elements. The remaining input parameter assignments are shown in Tab. D.22.

Hypotheses

The set of hypotheses considers multiple aspects of the performance analysis. It is assumed that the parallel brute-force implementation exposes the similar behaviour as anticipated in Sect. 6.3, having a linear increase in runtime while increasing the embedding dimension. Initial experiments have shown that the parallel brute-force implementations have a drastic performance surplus in comparison to the k-d tree implementations. In average, the runtime differences are expected to exceed one magnitude.

The k-d tree implementations are expected to be sensitive regarding the distribution of multi-dimensional vectors and the density of the recurrence matrix. The efficiency of pruning pairwise vector similarity comparisons should descend from uniform, normal, exponential to Cauchy distribution. Correspondingly, the runtimes regarding small maximum phase space diameter ratios are expected to be the lowest regarding the uniform distribution and the highest regarding the Cauchy distribution. This assumption should be independent of the k-d tree implementation used.

Executing the *create_recurrence_matrix* operator implementation using the `query_ball_tree` function should take longer compared to executing `radius_neighbors_graph`. This is due to the former transforming the query results by additionally calling the function `csc_matrix` (see Sect. 5.2.2). Based on prior experiments, the size of this overhead is expected to consume up to one half of the total runtime.

Results

The complete set of experimental results is captured in App. E.2.2. The experimental results are largely in alignment with the initial hypotheses. There is an almost linear increase in runtime with respect to the embedding dimension, except from isolated outliers regarding the maximum runtimes (see Fig. 6.21). In general, the spread of the runtimes for each combination of distribution type and embedding dimension is relatively small. This emphasises the fact, that the performance of the parallel exhaustive implementation is independent of the spatial distribution of input vectors. In the following, the runtime behaviour of the k-d tree implementations is investigated in more detail.

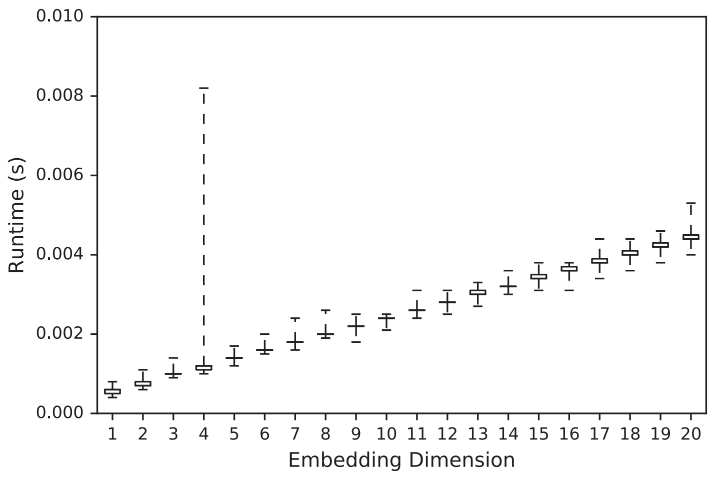


Figure 6.21: Multi-dimensional search trees - Parallel brute-force implementation on Cauchy distribution.

K-d Tree Implementations. The performance of both k-d tree implementations heavily depends on the spatial distribution of multi-dimensional vectors. It is reasonable to assume that large spreads between the minimum and maximum runtimes for executing the operator *create_recurrence_matrix* indicate different efficiencies regarding the pruning of similarity comparisons.

The efficiency of pruning that is observed regarding the individual distribution types is also in alignment with the initial hypotheses. The largest spread between minimum and maximum runtimes was measured for the uniform distribution (see Fig. 6.22 and Fig 6.23). Recall that the large spreads regarding the runtimes measured originate from the high sensitivity regarding the distribution type as well as the similarity threshold applied. Their impact varies between the different distribution types. Regarding the Cauchy distribution, the input vectors concentrate along the axes of the multi-dimensional coordinate system. Recurrence rates approaching 100% are already achieved for maximum phase space diameter ratios considerably smaller than 0.01. As a result, the spread between minimum and maximum runtimes is much smaller with respect to the other distribution types.

The *create_recurrence_matrix* operator implementation relying on the `cKDTree` is considerably slower than the one using `KDTree`, as expected. The average runtime of the former is always higher than the 50% quantile of the latter. The runtime ratio varies from 4.19 (uniform distribution, $m = 3$) to 5.37 (normal distribution, $m = 4$). As explained before, the overall runtime of using `cKDTree` includes significant overhead for transforming the computing results into the compressed sparse format. Hence, only a fraction of the runtime difference stems from the actual computation of pairwise input vector similarities.

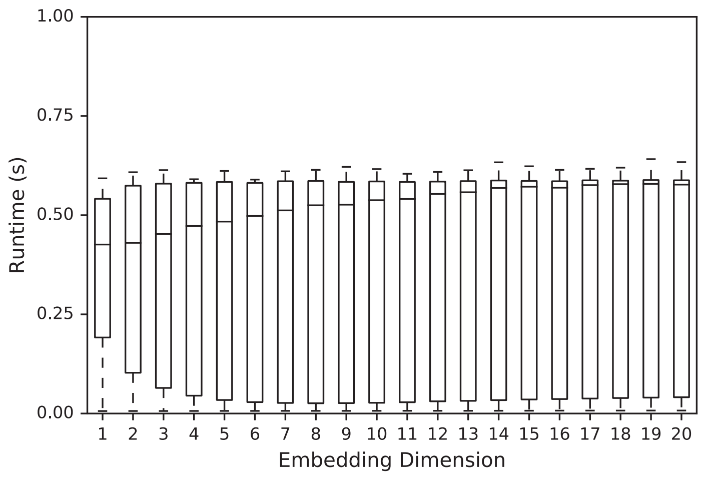


Figure 6.22: Multi-dimensional search trees - `cKDTree` on uniform distribution.

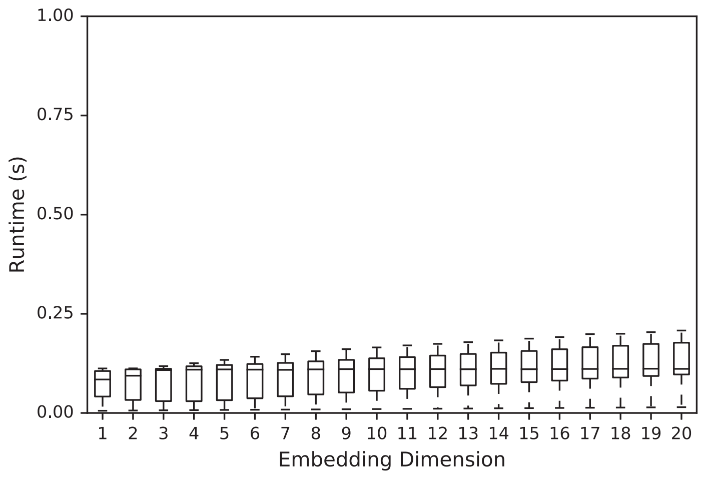


Figure 6.23: Multi-dimensional search trees - `KDTree` on uniform distribution.

Parallel Brute-Force vs. K-d Tree Implementations. The k-d tree operator implementations suffer from higher runtimes, compared to the parallel brute-force implementation. The ratio between the runtimes of the k-d tree based implementations and the parallel brute-force implementation decreases with increasing embedding dimension, different than expected. This behaviour is illustrated in Fig. 6.24, which compares the minimum runtimes of two specific implementations for all distribution types applied.

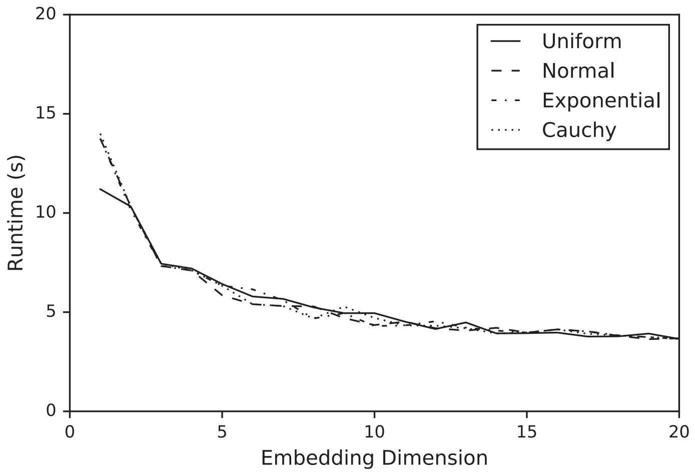


Figure 6.24: Runtime Ratios - `KDTree` vs. Parallel Brute-Force Impelementation.

It holds that the minimum runtime regarding the k-d tree implementations is higher than its parallel brute-force equivalent, for each combination of distribution type and embedding dimension. For `cKDTree` (see Tab. E.60), the runtime ratio considering the minimum runtime is at least 3.65 (uniform and exponential distribution, $m = 20$). Regarding `KDTree` (see Tab. E.60), the runtime ratio considering the minimum runtime is at least 1.88 (uniform and exponential distribution, $m = 20$). Hence, the parallel brute-force *create_recurrence_matrix* operator implementation outperforms the k-d tree based implementations, even for sparse recurrence matrices.

7 Automatic Performance Tuning for Implementation Selection

Chapter 4 presents *scalable recurrence analysis* (*SRA*), a computing approach based on dividing a recurrence matrix into a set of sub matrices and distributing their analysis across multiple compute devices. Intermediate results are computed for each sub matrix separately. They are recombined into a global RQA result in a final processing step. *SRA* proposes to conduct the computation of the pairwise input vector similarities as well as the detection of vertical and diagonal lines in a massively parallel manner. Chapter 5 introduces several concepts from database technology to the *SRA* processing pipeline. Based on these considerations, a set of analytical operator implementations is provided.

Chapter 6 investigates the impact of specific implementation properties in detail. In this regard, the significance of hardware architecture and analysis parameters on the runtime performance has been highlighted. As an example, not-materialising the recurrence matrix delivers appropriate results for small embedding dimensions, while materialisation is favourable with increasing input vector dimensionality. In conclusion, there is no single implementation conducting RQA that delivers the best performance under all circumstances. There are rather one or more implementations delivering appropriate performance results for a given analytical scenario on a certain hardware platform. This set is likely to contain different implementations, considering varying scenarios and compute devices.

This chapter presents an approach to automatically select well-performing combinations of operator implementations. This is achieved by using the concept of *automatic performance tuning* [Pankratius et al., 2011, pp. 239–263]. First, a concrete problem description is given. In the following, the theoretical background behind the automatic optimisation approach is presented. This includes a clarification of the terminology used. Concluding, the impact of the tuning approach on the runtime performance is investigated using three experiments. The first one focusses on the impact of applying specific implementation selection strategies. The second experiment evaluates the increase in efficiency provided by automatic performance tuning in the context of RQA processing. The third one presents the speed-up gained by employing more than one compute device during the analysis.

7.1 Problem Description

The initial motivation for introducing an approach to optimise RQA implementation selection stems from a fundamental principles behind the OpenCL framework. It allows to compile and execute identical source code on a variety of compute devices. Nonetheless, OpenCL does not guarantee that an implementation delivers appropriate runtime performance across different hardware architectures. In that sense, OpenCL provides *functional portability* but does not ensure *performance portability*. Among others, this dilemma has been pointed out for example by [Rul et al., 2010] and [Rosenfeld et al., 2015]. It will be aggravated by future hardware architectures supporting OpenCL.

Chapter 6 demonstrates that finding an appropriate operator implementation does not only depend on the hardware architecture applied but also on the RQA analysis scenario, which is defined by the input parameter assignments. The impact on the performance of selecting specific implementation properties, e.g., column-wise input data representation, has been analysed on several compute devices using different experimental setups. The focus of these experiments was on investigating the performance of individual operator implementations. However, conducting RQA requires the execution of a set of operators. A specific property may influence multiple operators and restricts the operator implementations applicable. In contrast to the prior considerations, this chapter focusses on the cumulative performance of all operator implementations that are required to retrieve the RQA results.

7.2 Theoretical Background

To select well-performing RQA implementations, the concept of automatic performance tuning, or short *auto-tuning*, is applied. It aims at exploiting the computing capabilities of a given hardware architecture as much as possible. The capabilities are measured using a *performance metric*, such as the computing time. Its value can be steered by a set of *tuning parameters*. The goal is to find a configuration of tuning parameter assignments, such that the value of the performance metric is optimised, e.g., the minimisation of the computing time.

According to [Pankratius et al., 2011, pp. 243–244], auto-tuning approaches can be distinguished along three dimensions, given a program whose execution has to be optimised.

Tuning type: Auto-tuning can either be performed *online* or *offline*. Online tuning refers to modifying the values of the tuning parameters while running a program. Multiple parameter configurations are applied during the program execution. Conducting the tuning offline, the tuning parameter configuration is modified between two program executions.

Time of tuning: Auto-tuning can either be performed during *development time* or during *production time*. Regarding the first, the auto-tuning is performed at the end of the development process. Suitable tuning parameter configurations are determined for a set of target platforms. Regarding the latter, a suitable tuning parameter configuration is either identified during the initial program execution or determined periodically.

Search strategy: Either *empirical* or *model-based* approaches can be applied to identify a suitable tuning parameter configuration. Empirical strategies require to execute a program or parts of it multiple times, while applying different tuning parameter configurations. Potential search strategies include *random sampling*, *local search* and *global search*. Note that there is no guarantee that the optimal tuning parameter configuration is found. Model-based strategies use analytical performance models to predict the optimal tuning parameter configuration. The models are specific to each combination of program, hardware architecture and other criteria. It highly depends on how good the model fits a specific computing scenario, whether a parameter configuration actually represents to the optimum.

The adaptation of tuning parameter configurations can be distinguished into [Whaley et al., 2001]:

Parameterised adaption: The performance characteristics of the program are modified by changing parameter values, e.g., compile-time variables.

Source code adaption: The performance characteristics of the program are modified by adapting its source code. This is done either by providing *multiple implementations* of the same algorithm, having the same semantics, or using *code generation* techniques, where the source code of a program is generated by another program.

In the following, the auto-tuning approach applied to *SRA* is characterised according to the previous distinctions.

7.2.1 Micro Adaptivity

The auto-tuning approach applied in the context of *SRA* follows the framework presented in [Raducanu et al., 2013] and [Rosenfeld et al., 2015]. [Raducanu et al., 2013] introduces *micro adaptivity*, a method to optimise the execution of database operators. Using the taxonomy presented previously, it can be classified as:

- an *online* auto-tuning approach
- performed during *production time*
- by applying *empirical* search strategies.

It is assumed that a database operator is applied to a set of tuples. Furthermore, there exist several implementations of this operator. The performance of multiple implementations is monitored while processing the stream of tuples. The most suitable implementation is subsequently selected based on an empirical analysis. This approach has been further extended by [Rosenfeld et al., 2015]. The following terminology is used to ensure comparability between the two publications.

Operator algorithm: The abstract description of the operator processing. This description may be given as pseudocode.

Operator implementation: The implementation of the operator algorithm. Regarding OpenCL, this includes source code that is executed on the host device and the kernel functions.

Flavour: The operator implementation combined with a specific tuning parameter configuration. The host code, if required, and the kernel functions are stored in compiled form.

Note that several flavours may be created based on the same operator implementation. Vice versa, every flavour refers to a single operator implementation.

[Raducanu et al., 2013] expresses the selection of the flavours as a *multi-armed bandit* problem, as described in [Robbins, 1952]. A set of flavours and a concrete computing scenario comprising a set of tuples is given. Each flavour is treated as single arm of the bandit. Each tuple is processed by exactly one flavour. The performance of processing the tuples is monitored using a specific measure, such as the computing time. Each tuple is assigned with a certain *reward*, such as the inverse of the computing time. The overall performance of the processing is expressed as the sum of the individual tuple rewards. The *regret* expresses the difference between the actual and the maximum possible reward. The goal is to optimise the flavour selection, such that the regret is minimal.

The flavour selection approach presented in [Raducanu et al., 2013] subdivides the execution of a program into *exploration* and *exploitation* phases. During exploration, knowledge about the performance of the individual flavours is gathered. During exploitation, the flavour with the maximum reward is applied. In this regard, flavours are treated as non-stationary processes. This results from a potentially varying workload while processing the set of tuples. Hence, algorithms solving the problem optimally, that means finding the best-performing flavour, can not be applied. There exist several algorithms to solve the multi-armed bandit problem approximately. This includes various *greedy* algorithms, finding the best local solution based on the data previously gathered [Black, 2005, Raducanu et al., 2013].

ϵ-greedy: Random flavours are selected with a probability of ϵ, using a uniform distribution. Their execution allows to gather performance data. The current best-performing flavour is selected with a probability of $1 - \epsilon$. Note that this definition does not claim a fixed number of tuples lying between two random flavour selections.

vw-greedy: Regarding the selection of the best-performing flavour, the ϵ-greedy algorithm considers all performance data gathered up to the processing of the current tuple. This limits the adaptability to changing workloads, since the knowledge previously gathered likely becomes invalid. Using the vw-greedy strategy, the knowledge is reseted after a certain amount of tuples has been processed. The algorithm considers fixed-sized segments, in which exploration and exploitation alternate.

ϵ-first: A fixed-sized exploration phase is conducted in the beginning of program execution. Its length is steered by changing the value of ϵ. The performance data gathered in this period is exploited afterwards, until the program execution terminates.

ϵ-decreasing: This strategy is very similar to the ϵ-greedy approach, but the value of ϵ is gradually decreased. This leads to a higher amount of exploration at the beginning and a higher amount of exploitation at the end of the program execution.

[Rosenfeld et al., 2015] extends the micro adaptivity approach to heterogenous computing environments. Here, the conflict between functional and performance portability in the context of OpenCL serves as a motivation to introduce strategies for flavour selection. It further addresses the problem of having a large amount of possible flavours, due to a large tuning parameter space.

Instead of instantiating all flavours, only a small pool of active flavours is kept. The size of the pool is set prior to program execution.

[Rosenfeld et al., 2015] divides the process of flavour selection into two parts. During the execution of a single query, the vw-greedy strategy, as introduced by [Raducanu et al., 2013], is used to select the best-performing flavour from the pool of active flavours. Between the execution of two queries or after a fixed amount of queries, the pool of active flavours is reorganised. For this purpose, a number of active flavours are evicted from the pool. They are replaced either by applying *greedy* or *genetic* search strategies. Using the greedy approach, new flavours are selected based on a random distribution. Regarding the genetic approach, all flavours except the two best-performing are evicted from the pool. Genetic propagation is used to create new flavours by combining the features of both flavours. Mutations with varying properties are introduced occasionally to avoid local reward maxima.

[Rosenfeld et al., 2015] comprises several experiments, which highlight the influence of the size of the flavour pool as well as the selection strategy on the overall query performance. In general, there seems to be a sweet spot regarding the size of the flavour pool. If the pool size is too small, the amount of flavours explored may not be sufficient. If the pool size is too large, the overhead for conducting explorations outweighs its benefits during exploitation. Comparing the two search strategies, the genetic approach outperforms the greedy approach with an increasing amount of queries. This behaviour is independent of the computing device employed.

The approach presented in [Rosenfeld et al., 2015] aims at reducing the overhead for handling large sets of potential flavours. Performance data is only gathered for active flavours, which is attached with several limitations. As explained earlier, the pool of flavours is only reorganised between the execution of queries. Hence, there is an emphasis on the selection of the pool members. Randomly selecting a small number of active flavours only before the execution of a query reduces the probability that a particular flavour is actually applied. This might cause high variations regarding the query runtimes, depending on the size of the pool and the portion of well-performing flavours. The genetic strategy either assumes that the performance of the individual flavours are independent of the query workload or that the workloads do not change. If either one of those conditions is not fulfilled, the approach might deliver worse results than selecting flavours randomly. Due to those issues, the following considerations focus on the original concept of micro adaptivity, as presented in [Raducanu et al., 2013].

7.2.2 Automatic Performance Tuning in *SRA*

Regarding *SRA*, the performance of the individual operator implementations highly depends on the concrete analysis scenario and hardware architecture employed, as demonstrated within the experiments presented in Chap. 6. Exploring all potential configurations is infeasible, because of their vast quantity. Therefore, an online auto-tuning approach gathering performance data during production time is the means of choice.

The concept of micro adaptivity is adapted to the constraints of *SRA*. In the following, the specific adjustments are presented in detail. To begin with, the terminology introduced before is extended. As stated previously, several analytical operators have to be executed to conduct RQA. Each operator implementation has specific properties, which may influence the processing in the subsequent operators. For this reason *variants* are introduced, which combine the functionality of implementations referring to multiple analytical operators. This leads to a modification of the semantics of *flavour*.

Variant: A set of operator implementations required to compute RQA results, which are coupled based on shared properties, e.g., an uncompressed recurrence matrix representation. A variant does not necessarily comprise implementations of all three analytical operators. Based on the implementation properties, it may only be required to execute the *detect_vertical_lines* and the *detect_diagonal_lines* operator, e.g., if the recurrence matrix is not materialised.

Flavour: A variant combined with a set of tuning parameter assignments. The same parameter assignments are applied to every operator implementation of the variant.

The *loop unrolling factor* is selected as a tuning parameter for the purpose of demonstration. Loop unrolling is applied to iterations over sets of items, to save computational overhead for evaluating loop headers. It is assigned with a value greater than or equal to one. Regarding *SRA*, this concept is employed while inspecting the columns and diagonals regarding line structures. The same loop unrolling factor is applied to the kernel functions of each of the corresponding operator implementations. The assumption behind using the same factor is, that the impact of loop unrolling on the performance highly depends on the hardware architecture. Loop unrolling is implemented using the OpenCL compiler directive `#pragma unroll <FACTOR>`, which is placed in front of the relevant loop within the kernel function. Note that loop unrolling may not be supported by each compute device providing an OpenCL interface.

In the context of *SRA*, the performance of individual flavours is not investigated on individual tuples or chunks of tuples but rather on sub matrix level. This is a contrast to the approaches presented in [Raducanu et al., 2013] and [Rosenfeld et al., 2015]. The processing of a sub matrix requires the execution of all analytical RQA operators contained by a single variant. They are executed sequentially according to the operator-at-a-time principle, hampering tuple-wise performance measurements. Instead, the cumulative runtime for executing all operators on a sub matrix is observed. The selection of flavours is optimised, such that the total runtime for processing all sub matrices is minimised.

Search Strategies

Regarding *SRA*, the greedy search strategies presented in [Raducanu et al., 2013] are used for flavour selection. The execution of the program conducting RQA is likewise subdivided into exploration and exploitation phases. For all search strategies except ϵ-first, the exploration phase comprises only the processing of single sub matrix to which a specific flavour is applied. The reason for this procedure is the limited amount of sub matrices usually available. Their amount highly depends on the ratio between total number of multi-dimensional vectors and edge lengths of the sub matrices. Furthermore, the processing is scattered among multiple compute devices. This may only leave tens of sub matrices for each device. This requires either a cautious choice of the edge length or restricting the number of compute devices. In the following, the set of greedy search strategies are modified.

ϵ-greedy: A fixed offset between two sub matrices employed for exploration is defined.

vw-greedy: After a specific number of exploration phases, the complete knowledge previously gathered is deleted.

ϵ-first: The initial exploration is performed on a fixed number of sub matrices.

ϵ-decreasing: After each exploration phase, the fixed offset between two sub matrices used for exploration is increased.

A strategy that randomly selects a flavour before processing a sub matrix is introduced additionally. Note that each flavour is selected once, before another flavour is used repeatedly. A new flavour is created by randomly selecting a combination of variant and tuning parameter assignments, using a uniform distribution.

As explained in Sect. 4.1.1, not all sub matrices have the same extent. The shape and size of the sub matrices at the outer borders of the recurrence matrix diverges. To ensure comparability, the runtime for processing each individual sub matrix is normalised to an amount of 10^8 matrix elements. For the purpose of simplification, this normalisation abstracts from the specific shape of each sub matrix. Note that this procedure neglects its potential influence on the runtime.

The flavour selection is performed individually for each compute device. It requires to conduct the performance analysis per device used during the processing. This approach enables to optimise the overall runtime performance of the program by optimising the execution on each compute device. This allows to exploit the full computing capabilities of the computing system, while utilising compute devices with different performance characteristics.

7.3 Evaluation

In this section, the outcome of three experiments referring to flavour selection are presented. The first experiment evaluates the impact of the different selection strategies on the overall performance of conducting RQA. The second experiment investigates the impact of automatic performance tuning on the efficiency of exploiting parallel computing resources. The third experiment focusses on the application of multiple compute devices, investigating horizontal scalability. The same real-world time series is employed in each experiments. It captures the Potsdam temperature profile (see Chap. 1) at an hourly resolution from 1.1.1893 to 31.12.2014. This results in $1,069,416$ data points. The time series captures the anomaly temperatures, meaning the derivations from the hourly average on a yearly scale.

Similar RQA input parameter assignments as applied in an experiment referring to the period up to 2011 are used during the analysis [Rawald et al., 2014a]. This includes the embedding dimension, time delay and similarity measure. The similarity threshold is set to 5% of the maximum phase space diameter. The RQA input parameter assignments are given in Tab. D.23. The corresponding RQA measures are presented in Tab. D.24, using minimum line lengths of two for diagonal, vertical and white vertical lines. The values are identical across all compute devices employed.

7.3.1 Selection Strategies

This experiment compares the overall runtimes for conducting RQA using a set of selection strategies. In this regard, the runtimes for performing all computations from reading the input data to computing the RQA measures is compared.

Setup

The following compute devices are employed during the experiment:

- the single graphics processor of the AMD RX 470 GPU in computing system *(A)*,
- the four graphics processors of the two Nvidia GeForce GTX 690 GPUs in computing system *(B)*, and
- the two Intel Xeon E5620 CPUs in computing system *(C)*.

The set of implementations used during this experiments are restricted to parallel brute-force processing, due to the large increases in runtime using k-d tree as well as grid directory approaches. The combination of possible operator implementation results in ten variants. Each variant is supplemented by a specific loop unrolling factor between 2^0 and 2^5, which results in a pool of 60 flavours. The compilation of the kernel functions is conducted the first time a flavour is selected. The default compiler optimisations are activated for each flavour. The experiment compares the overall runtime of conducting RQA, while using the set of selection strategies presented above. The parameters relevant for each selection strategy, including their assignments, are presented in Tab. D.25.

An experimental configuration comprises a selection strategy in combination with a set of compute devices. Each configuration is applied in five experimental runs, to generate different flavour selection instances. The random generation of flavours should lead to variations in the overall runtimes. The size of the sub matrices is kept constant across all sets of compute devices.

Hypotheses

The individual performances of the different greedy selection strategies are compared against each other. Chapter 6 showed that there are differences in the runtime performance between the different flavours, resulting from the varying properties of the analytical operator implementations. Hence, the different selection strategies should result in varying performance characteristics.

random: The runtimes while picking flavours randomly serve as a reference, since all flavours have the same probability of being selected.

ϵ-**greedy:** A balanced runtime performance should be observed. Using the parametrisation from Tab. D.25, one tenth of the set of sub matrices is used to explore random flavours. It allows to gather performance about all flavours, considering a total of 2,916 sub matrices.

vw-greedy: The runtime performance depends on the contents of the sub matrices and their assignments to the set of compute devices. Assuming a homogeneous workload, the runtimes should be higher compared to the ϵ-greedy strategy. This is due to resetting the performance evaluation on a regular basis.

ϵ-first: The runtime performance depends on the portion of well-performing flavours and the length of the initial exploration phase. Given only a small number of well-performing flavours and a short exploration phase, higher runtimes compared to randomly selecting flavours might be observed.

ϵ-decreasing: The runtime performance depends on the workloads of the individual sub matrices and the exploration phase in which a well-performing flavour is selected. The latter does also hold for the ϵ-greedy strategy, although the impact enhances while gradually increasing the delta between two exploration phases. The runtimes are lower compared to the ϵ-greedy strategy, if the workloads remain homogeneous and a well-performing flavour is selected in an early exploration phase.

Results

The complete set of experimental results, including figures and numerical data, are presented in App. E.3.1. The impact of each greedy selection strategy on the runtime performance is evaluated on each of the sets of compute devices. Each boxplot depicted in the figures in App. E.3.1 refers to five experimental runs. The highest and the lowest runtimes are highlighted as outliers. The experimental results differ drastically between the three sets of compute devices, requiring a set-specific analysis. This includes a comparison of the runtimes using the selection strategies with always using the flavour with the lowest runtime in average, referred to as *baseline*.

The random selection strategy delivers the highest runtimes across all compute device. Nonetheless, the relative difference to the runtimes of the remaining strategies varies. Considering the runtimes referring to the 50% quantile, the highest differences can be observed regarding the Nvidia GTX 690 GPUs, with up to a factor of 1.58 (ϵ-decreasing), followed by the Intel Xeon E5620 CPUs, with a factor of up to 1.57 (ϵ-first). The lowest ratio can be observed on the AMD Radeon RX 470 GPU, with a factor of only 1.12 (vw-greedy). In the following, the device-specific results are investigated in detail.

AMD Radeon RX 470: The ϵ-greedy, ϵ-first and ϵ-decreasing strategy deliver comparable runtime results. They are almost overlapping with the baseline. The vw-greedy strategy delivers slightly higher runtimes, which are still lower compared to picking random flavours.

Nvidia GeForce GTX 690: All strategies have similar performance characteristics, except the random flavour selection. The ϵ-decreasing strategy delivers the lowest, whereas the ϵ-first strategy delivers the highest runtimes. It is of particular interest that there is a considerable gap between the minimum runtimes of each selection strategy and the baseline. This may stem from a large spread regarding the runtimes of the individual flavours.

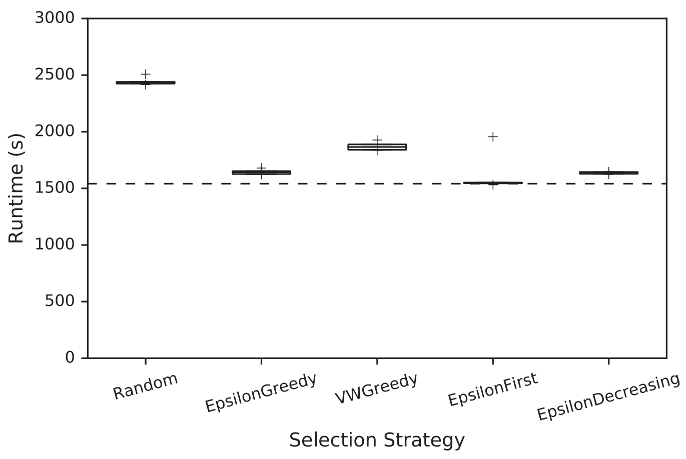


Figure 7.1: Selection strategies - Intel Xeon E5620.

Intel Xeon E5620: The relation between the runtimes of the selection strategies is comparable to the AMD GPU. Highlighting the ϵ-first strategy, it delivers contradictory result (see Fig. 7.1). On the one hand, it contains the overall lowest runtime, similar to the baseline. On the other hand, it comprises the highest runtime of all selection strategies, except from randomly selecting flavours. This results from not-selecting a flavour that has the properties of a well-performing variant, during the corresponding experimental run. These properties are captured in Tab. E.67.

7.3.2 Efficiency

Increasing the efficiency in using the computing resources available is one of the key drivers in introducing automatic performance tuning to *SRA*. The previous experiment compares the runtime efficiency of several flavour selection strategies. This experiment compares the efficiency of the parallel *SRA* implementation using OpenCL to two state-of-the-art RQA implementations. It is investigated, whether the former delivers efficiency improvements compared to existing implementations.

Setup

The experimental setup is equivalent to Sect. 7.3.1. Among others, this includes using the Potsdam temperature profile from 1893 until 2014 as input data (see Chap. 1). The input parameter assignments are presented in Tab. D.26. Note that the ϵ-first strategy is applied regarding the selection of flavours while applying the the *SRA* implementation based on OpenCL.

The runtime performance of the *SRA* implementation is compared to Commandline Recurrence Plots (CRP)[1] (see Sect. 2.2.2) and Commandline RQA Multithreaded (CRM)[2] (see

[1]The i686 binaries of version 1.13z are employed during the experiment.
[2]The source code is compiled using the compiler optimisation level *O3* to ensure legitimate performance com-

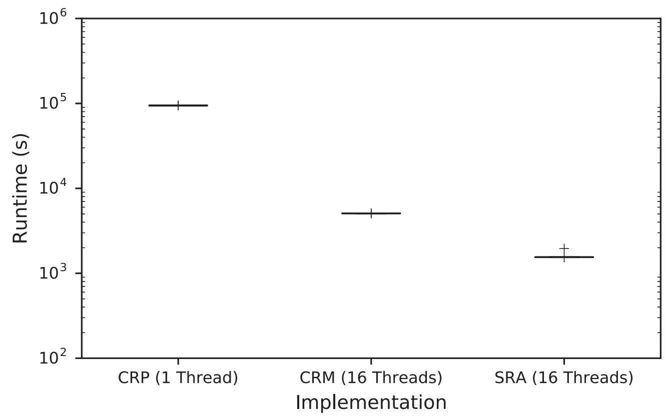


Figure 7.2: Efficiency - Intel Xeon E5620.

Sect. 3.1.1). The evaluation is restricted to computing system *(C)*, since the two Intel Xeon E5620 CPUs comprise the highest total number of CPU threads. Note that Commandline Recurrence Plots uses only a single CPU thread, while the CRM and *SRA* implementation use all sixteen CPU threads available. Every implementation applied in this experiment covers the complete process from reading the input data to having computed the final RQA measures.

Hypotheses

The CRP implementation is expected to have the highest runtimes, since it uses only a fraction of the computing resources available. The CRM and the *SRA* implementation leverage the maximum of 16 CPU threads, which allows to compare their individual efficiency directly. It is expected that the *SRA* implementation using OpenCL is in general at least as fast as the CRM implementation using OpenMP. This is due to the fact that several operator implementations with varying properties are explored using the ϵ-first strategy. At least one of them should match the architecture of the Xeon CPUs better than the CRM implementation.

Results

The experimental results are fully in alignment with the initial hypotheses, including Commandline Recurrence Plots delivering the highest runtimes (see App. E.3.2). Furthermore, the runtimes of the *SRA* implementation are lower than using Commandline RQA Multithreaded.

Regarding the 50% quantiles, the *SRA* implementation consumes only $\approx 30\%$ of the runtime of the CRM implementation (see Fig. 7.2). Again, it is emphasised that both implementations leverage the exact same computing resources. This represents a drastic increase in efficiency achieved by combining *SRA* with automatic performance tuning.

parisons [Bailey, 1991].

7.3.3 Scalability

One of the major properties of *SRA* is enabling the usage of multiple computing devices at the same time. This experiment investigates the performance improvements gained by increasing the number of compute devices within a single computing system. According to [Michael et al., 2007], there is a distinction between two types of scalability.

Scale-up: Adding more compute resources, such as CPUs, to to a single computing node, typically a shared memory system.

Scale-out: Adding more interconnected computing nodes to a computing landscape, typically a cluster.

It is assumed that each computing node runs its own operating system instance and has access to a certain amount of dedicated memory. The set of computing nodes are usually connected using a specific network technology, e.g., Ethernet. Regarding OpenCL, adding compute devices to a computing system does neither satisfy the properties of scale-up nor scale-out. It is rather a hybrid approach, that combines properties from both categories.

A computing system, equivalent to a compute node, contains a single host device, which is attached with main memory to store data shared among multiple compute devices. A compute device is attached with dedicated global memory that can only be accessed by itself. Furthermore, each compute device runs its own instance of the kernel functions. Therefore, the scaling type referring to *SRA* can rather be described as *scale-out-in-a-box* [Michael et al., 2007].

Setup

The experiment is restricted to computing system *(B)*, containing two Nvidia GeForce GTX 690 GPUs. A single GPU contains two graphics processors, each attached with 2GB of dedicated global memory. This results in a total of four compute devices. The runtime of conducting RQA, from reading the input data to computing the final RQA measures, is observed while increasing the number of compute devices applied during the computations from one to four.

The remaining experimental setup is similar to the evaluation in Sect. 7.3.1. The parameterisation is summarised in Tab. D.27. The flavour selection is solely conducted using the ϵ-greedy strategy, since it delivers balanced runtime results across all sets of compute devices.

Hypotheses

All compute devices employed have similar computing capabilities. Therefore, a reduction in runtime is expected while increasing their amount. Using multiple compute devices creates an overhead for maintaining device specific data, such as the compilation of kernel functions and the merging of line lengths histograms. It is expected that this overhead eats up the performance improvements gained by distributing the processing of the sub matrices among a large number of compute devices. This behaviour result in a presence of a global runtime minimum at a certain number of devices.

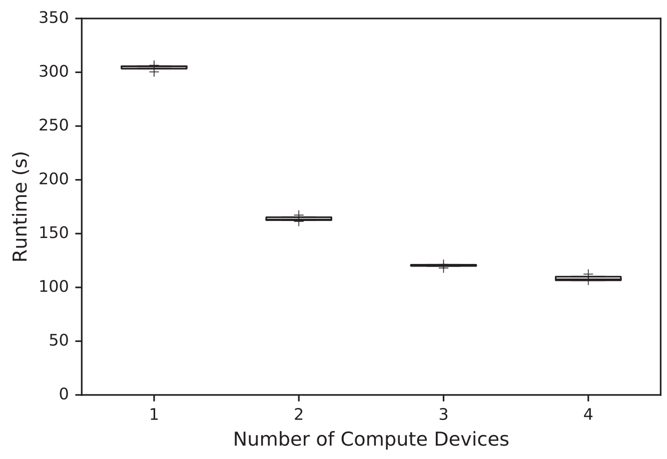


Figure 7.3: Scalability - Nvidia GeForce GTX 690.

According to *Amdahl's law*, the processing of a parallel application is subdivided into segments either referring to sequential or parallel execution [Amdahl, 1967]. The portion of segments belonging to either one of those categories determines the potential performance gains by adding more compute devices. The sequential processing includes for example the reading of input data and the final computation of the RQA measures. The processing of the set of sub matrices is distributed across the set of compute devices and conducted in parallel fashion. As a consequence, a sublinear decrease in runtime is expected while increasing the number of compute devices linearly.

Results

The experimental results, presented in App. E.3.3, are in alignment with the initial hypotheses. The overall runtimes decrease when increasing the number of devices, as shown in Fig. 7.3.

The results confirm that using multiple compute devices enables considerable performance improvements. An effective speedup of 2.84 is observed, when comparing the 50% quantiles of using a single and using four compute devices. Nonetheless, the results also indicate a considerable impact of the sequential processing in combination with the overhead for maintaining device specific data structures. The individual amount of contribution regarding each of these factors is not further analysed.

The significant flattening regarding the transition of the runtimes from three to four compute devices implies that only marginal performance improvements are gained by using more than three compute devices. It is reasonable to assume that using four compute devices results in runtimes that are close to the minimum runtime regarding this specific experimental setup.

8 Conclusion

The following sections summarise the findings of this thesis, present limitations of the *SRA* computing approach introduced and provide perspectives on future research.

8.1 Summary

The previous chapters introduced a novel computing approach to recurrence analysis, a method from nonlinear time series analysis, conducting the related computations in an scalable and efficient manner. It allows to overcome several limitations inherent in state-of-the-art computing approaches. This includes in particular the property of being able to process time series of arbitrary size.

Scalable recurrence analysis (*SRA*) introduces parallel processing on two levels. The global recurrence matrix is divided into a set of sub matrices. Each sub matrix is processed by an individual compute device. The focus of this novel computing approach is especially on the computation of scalar measures based on the contents of recurrence matrices, which is referred to as recurrence quantification analysis (RQA). In this regard, global data structures allow to share data among multiple sub matrices. This includes the carryover buffers that store the length of line structures detected at the outer borders of sub matrices. A specific order regarding the processing of the sub matrices ensures that valid global RQA results are computed. The line detection within a single sub matrix is performed in a massively parallel manner. In this regard, functionality provided by the OpenCL framework is used to offload processing to accelerators, such as GPUs.

The second part of the manuscript focusses on the efficiency of the computations related to recurrence analysis. This was driven by a basic property of OpenCL, providing functional portability by enabling the compilation and execution of identical source code on a vast number of different devices. A major drawback of OpenCL is that it does not guarantee that a given implementation delivers appropriate performance results across different compute devices. For this purpose, several concepts from database technology, such as different types of input data representation, have been applied to the RQA processing pipeline. An extensive evaluation investigated, how the application of those concepts influence the performance of analytical operator implementations. The experimental results highlight that the runtime performance does not only depend on the combination of RQA implementation and hardware platform but also on the parameterisation of the quantitative analysis. This aspect is a contribution to the field of recurrence analysis computing. It motivated the combination of *SRA* with an automatic performance tuning approach that selects well-performing RQA implementations based on greedy selection strategies.

Based on a set of experiments, it is demonstrated that automatic performance tuning allows to drastically improve the performance of conducting recurrence quantification analysis. Consider the Potsdam temperature profile from 1893 to 2014, a real-world time series containing more than one million data points. Here, the runtime for performing RQA could be reduced from more than a day, running singled-threaded state-of-the-art software on a server CPU, to roughly 100 seconds, executing the OpenCL-based *SRA* implementation on four GPU processors. *SRA* introduces only limited synchronisation overhead, which allows to scale well with increasing the number of compute devices. As a result, the runtime for conducting the specific analysis mentioned before could be reduced by a factor of 2.8, when increasing the number of GPU processors from one to four.

The automatic performance tuning approach allows to optimise the execution to the properties of the individual compute devices. Hence, it is possible to employ compute devices with varying hardware architectures, while still optimising the performance. It allows to increase the number of sub matrices processed by a specific device in a certain amount of time. This presents a major advantage over existing computing approaches. As an example, the massively parallel *SRA* implementation that incorporates OpenCL and auto-tuning outperforms a state-of-the-art RQA implementation using OpenMP by a factor of roughly three, considering the previous analytical scenario that is conducted on two server CPUs. It is stressed that both implementations exploit the exact same parallel computing resources.

8.2 Limitations

Despite its benefits, *SRA* is attached with specific limitations. The most severe limitation is that the approach does not reduce the quadratic complexity of the parallel algorithms, depending on the number of multi-dimensional vectors extracted from the input time series. Implementations regarding two types of index data structures, including grid directories and multi-dimensional search trees, have been evaluated regarding their potential for performance improvements during the computation of the pairwise vector similarities. The implementations considered do not allow to reduce the runtime for recurrence matrix creation, although they expose a lower time complexity. Furthermore, no approaches to reduce the time complexity of the algorithms to determine the line structures in an exact manner could be identified. The latter remains on open research question.

The *SRA* approach leveraging the massively parallel computing capabilities of multiple compute devices allows to analyse time series with a length between 10^6 and 10^7 data points in an appropriate amount of time. It is assumed that a single computing system containing ten or less accelerators is applied. This is due to only flattening the quadratic increase in runtime by performing the computations in a parallel manner.

The decomposition in multiple parallel analytical operators allows to profit from future hardware developments. This especially includes the increase in the number of compute units and processing elements per compute device of upcoming architectures, e.g., Nvidia Volta [Durant et al., 2017] and AMD Vega [Advanced Micro Devices, Inc., 2017]. Research in fields such as artificial intelligence and machine learning drive the development of computing systems adhering to scale-out-in-a-box, such as Nvidia DGX-1 [NVIDIA Corporation, 2017]. This computing system contains up to eight graphics cards that provide a total of 120 TFLOPS regarding single precision floating point computations.

Although there is a trend towards an increased number of processing elements per compute device, it will not be possible to analyse time series containing more than 10^8 data points in the foreseeable future, when using only a single computing system. Time series of such a size require different computing approaches, including the ones presented in Sect. 3.3 and Sect. 3.4. Note that those approaches either do not use all information provided by the time series or only return approximate results. Among others, the following section drafts an approach on how to obtain exact results for such time series by extending *SRA*.

8.3 Perspectives

The *SRA* approach as presented in Chap. 4 divides a given recurrence matrix into multiple sub matrices. They are distributed across multiple compute devices in a single computing system. This procedure can easily be adapted to support not only a single but multiple computing systems or *compute nodes*, that are organised in a network. Each of those compute nodes is equipped with one or more compute devices. A multi-level partitioning and processing scheme is introduced to address this environment, which comprises the following steps:

1. Partition the recurrence matrix into multiple sub matrices.
2. Distribute the sub matrices across multiple compute nodes.
3. Partition a sub matrix in multiple sub-sub matrices.
4. Distribute the sub-sub matrices across the compute devices of the compute node.
5. Recombine the individual sub-sub matrix results on compute node level.
6. Recombine the individual sub matrix results on network level.

The structure and the semantics of the carryover buffers and the line length histograms can be likewise applied to the multi-level approach. Moreover, the processing order can be used to synchronise the computations of more than one compute node. This highlights that the underlying concepts of *SRA* can easily be applied to multiple processing levels. The challenges related to this extension rather address engineering aspects, such as steering the communication between the individual compute nodes and supplying global data on network level.

Despite extending *SRA* to employ multiple computing nodes, there are other aspects that might be considered in future work. This includes evaluating the performance of the *SRA* implementation running on a single computing system on other hardware platforms, e.g., Field-programmable arrays or Intel Xeon Phi accelerators. In addition to focus solely on the runtime of conducting an analysis, aspects like power usage could be investigated in detail.

Previous evaluations either considered a single compute device or multiple compute devices of the same type, for example multiple GPU processors of the same model. Nonetheless, a computing system may contain compute devices with different computing capabilities, such as combinations of various GPU models or GPUs and CPUs. It is worthwhile to analyse the distribution of work among the individual devices and the impact of such hardware configurations on the overall performance.

Regarding the implementation of *SRA*, the internal representation of the data points of the input time series may be changed from 32-bit to 16-bit floating point numbers. The increasing importance of reduced precision is mainly driven by applications from machine learning. This could also be leveraged regarding RQA processing. It would allow reduce the runtime for conducting the RQA computations up to 50%, given devices providing half-precision support. An open research question is the impact of the reduced precision on the structures within recurrence matrices.

Subsuming, the *SRA* computing approach represents a break-through in recurrence analysis computing, providing exact computing results in short time for series exceeding one million data points. This enables applications previously unfeasible, either considering very long or large amounts of time series. Relying on open-source software and frameworks supported by a variety of vendors, *PyRQA* will heavily profit from future developments in the field of parallel computing hardware.

A Approximation of RQA Measures - Source Code

```python
import numpy as np
import sys


def read_file(f_path):
    t_series = []

    with open(f_path, 'r') as f:
        for line in f.readlines():
            if line:
                t_series.append(float(line))

    return np.array(t_series)


def concatenate(ind,
                t_series,
                m,
                t,
                v):
    rows = []

    for i in np.arange(m):
        start = ind + (i*t)
        rows.append(t_series[start:start + v])

    return np.mat(rows)


def hash_matrix(mat):
    mat.flags.writeable = False
    return hash(mat.data)


def pp_approx(t_series,
              n,
              m,
              t,
              e,
              v):
    number_of_matrices = n - (v - 1)
    hashed = np.zeros(number_of_matrices, dtype=np.int)
    if e > 0:
        t_series = np.floor(t_series / (2 * e))

    for i in np.arange(number_of_matrices):
        matrix = concatenate(i, t_series, m, t, v)
        hashed[i] = hash_matrix(matrix)

    unique, counts = np.unique(hashed,
                               return_counts=True)

    return np.sum(np.square(counts))


if __name__ == "__main__":
    arguments = sys.argv[1:]

    file_path = arguments[0]
    embedding_dimension = int(arguments[1])
    time_delay = int(arguments[2])
    epsilon = float(arguments[3])
    d_min = int(arguments[4])

    time_series = read_file(file_path)
    number_of_recurrence_vectors = len(time_series) - \
        ((embedding_dimension - 1) * time_delay)


    pp_one          = pp_approx(time_series,
                                number_of_recurrence_vectors,
                                embedding_dimension,
                                time_delay,
                                epsilon,
                                1)

    pp_mu           = pp_approx(time_series,
                                number_of_recurrence_vectors,
                                embedding_dimension,
                                time_delay,
                                epsilon,
                                d_min)

    pp_mu_plus_one  = pp_approx(time_series,
                                number_of_recurrence_vectors,
                                embedding_dimension,
                                time_delay,
                                epsilon,
                                d_min + 1)

    print "PP^1: %.0f" % pp_one
    print "PP^%d: %.0f" % (d_min, pp_mu)
    print "PP^%d: %.0f" % (d_min + 1, pp_mu_plus_one)
    print ""
    rr = float(pp_one) / pow(number_of_recurrence_vectors, 2)

    det = float(d_min * pp_mu - (d_min - 1) * pp_mu_plus_one) / pp_one

    print "RR: %.5f" % rr
    print "DET: %.5f" % det
```

B Recurrence Matrix Materialisation - Data Transfer

This appendix contains mathematical equations capturing the amount of data elements transferred while choosing specific approaches regarding recurrence matrix materialisation. There is a distinction between approaches materialising the matrix, either by applying a separate *create_recurrence_matrix* operator (*CRM Operator*) or using intermediate results recycling (*Recycling*), and computing the matrix values on-the-fly during the detection of line structures (*Non-Materialisation*). Each section compares the amount of data elements transferred of two specific approaches. The equations are solved for the embedding dimension m. Further descriptions can be found in Sect. 5.1.5.

B.1 CRM Operator vs. Non-Materialisation (Symmetric)

$$2mN^2 + N^2 + N^2 + \frac{N(N-1)}{2} = 2mN^2 + 2m\frac{N(N-1)}{2} \tag{B.1}$$

$$N^2 + N^2 + \frac{N(N-1)}{2} = 2m\frac{N(N-1)}{2} \tag{B.2}$$

$$2N^2 = 2m\frac{N(N-1)}{2} - \frac{N(N-1)}{2} \tag{B.3}$$

$$2N^2 = (2m-1)\frac{N(N-1)}{2} \tag{B.4}$$

$$\frac{4N^2}{N(N-1)} = 2m - 1 \tag{B.5}$$

$$\frac{4N^2}{N(N-1)} + 1 = 2m \tag{B.6}$$

$$\frac{4N^2 + N(N-1)}{N(N-1)} = 2m \tag{B.7}$$

$$\frac{N(4N + (N-1))}{N(N-1)} = 2m \tag{B.8}$$

$$\frac{4N + N - 1}{N - 1} = 2m \tag{B.9}$$

$$\frac{5N - 1}{N - 1} = 2m \tag{B.10}$$

$$m = \frac{5N-1}{2N-2}. \tag{B.11}$$

B.2 CRM Operator vs. Non-Materialisation (Non-Symmetric)

$$2mN^2 + N^2 + N^2 + N^2 = 2mN^2 + 2mN^2 \tag{B.12}$$
$$N^2(2m + 1 + 1 + 1) = N^2(2m + 2m) \tag{B.13}$$
$$2m + 3 = 4m \tag{B.14}$$
$$3 = 2m \tag{B.15}$$
$$m = 3/2. \tag{B.16}$$

B.3 Recycling vs. Non-Materialisation (Symmetric)

$$2mN^2 + \frac{N(N-1)}{2} + \frac{N(N-1)}{2} = 2mN^2 + 2m\frac{N(N-1)}{2} \tag{B.17}$$
$$2\frac{N(N-1)}{2} = 2m\frac{N(N-1)}{2} \tag{B.18}$$
$$N(N-1) = mN(N-1) \tag{B.19}$$
$$m = 1. \tag{B.20}$$

B.4 Recycling vs. Non-Materialisation (Non-Symmetric)

$$2mN^2 + N^2 + N^2 = 2mN^2 + 2mN^2 \tag{B.21}$$
$$2N^2 = 2mN^2 \tag{B.22}$$
$$2 = 2m \tag{B.23}$$
$$m = 1. \tag{B.24}$$

C Computing Environment

C.1 Computing System (A)

Table C.1: Central Processing Unit (CPU).

Property	*Value*
Model	Intel Core i5-3570
Architecture	Ivy Bridge
Number of Processors	1
Number of Cores	4
Number of Threads	4
Base Frequency	3.4 GHz
Max Turbo Frequency	3.8 GHz
Cache SIze (L3)	6 MB

Table C.2: Main Memory.

Property	*Value*
Type	DDR3
Size	2 * 8 GB = 16 GB
Clock Rate	1333 MHz

Table C.3: Graphics Processing Unit (GPU).

Property	*Value*
Model	Sapphire Radeon RX470 Nitro+
Architecture	Polaris 10 / Graphics Core Next (GCN) v1.4
Number of Processors	1
Number of Stream Processors	2048
Core Clock Rate	1121 MHz
Boost Clock Rate	1260 MHz
Memory Type	GDDR5
Memory Size	8 GB
Memory Bandwidth	211 GB/s
Memory Bus Type	PCI-E 3.0
Memory Bus Width	256 bit
Driver Type	AMDGPU-PRO
Driver Version	16.40-348864
OpenCL Version	1.2

Table C.4: Operating System.

Property	*Value*
Distribution	Ubuntu
Version	16.04.1 LTS
Linux Kernel Version	4.4.0

C.2 Computing System (B)

Table C.5: Central Processing Unit (CPU).

Property	*Value*
Model	Intel Core i7-3820
Architecture	Sandy Bridge-E
Number of Processors	1
Number of Cores	4
Number of Threads	8
Base Frequency	3.6 GHz
Max Turbo Frequency	3.8 GHz
Cache Size (L3)	10 MB

Table C.6: Main Memory.

Property	*Value*
Type	DDR3
Size	2 * 8 GB = 16 GB
Clock Rate	1600 MHz

Table C.7: Graphics Processing Unit (GPU).

Property	*Value*
Model	Nvidia GeForce GTX 690
Architecture	Kepler
Number of Processors	2
Number of CUDA Cores (per Processor)	1536
Core Clock Rate	915 MHz
Boost Clock Rate	1019 MHz
Memory Type	GDDR5
Memory Size (per Processor)	2 GB
Memory Bandwidth (per Processor)	192 GB/s
Memory Bus Type	PCI-E 3.0
Memory Bus Width (per Processor)	256 bit
Driver Type	NVIDIA Accelerated Linux Graphics Driver
Driver Version	352.20
CUDA Version	5.5
CUDA Compiler Type	nvcc
CUDA Compiler Version	Wed_Jul_17_18:36:13_PDT_2013
OpenCL Version	1.2

Table C.8: Operating System.

Property	*Value*
Distribution	openSUSE
Version	12.2 (x86_64)
Linux Kernel Version	3.4.63

C.3 Computing System (C)

Table C.9: Central Processing Unit (CPU).

Property	*Value*
Model	Intel Xeon Processor E5620
Architecture	Westmere-EP
Number of Processors	2
Number of Nodes	2
Node Connection	QuickPath Interconnect (QPI)
Number of Cores (per Processor)	4
Number of Threads (per Processor)	8
Base Frequency	2.4 GHz
Max Turbo Frequency	2.66 GHz
Cache Size (L3) (per Processor)	2 * 12 MB
OpenCL Version	1.2

Table C.10: Main Memory.

Property	*Value*
Type	DDR3
Number of Channels (per Node)	3
Number of Banks (per Node)	3 * 2 = 6
Size (per Node)	6 * 4 GB = 24 GB
Clock Rate	1067 MHz

Table C.11: Operating System.

Property	*Value*
Distribution	openSUSE
Version	13.2 (x86_64)
Linux Kernel Version	3.16.7

D Experimental Setups

D.1 Parallel Brute-Force Processing

D.1.1 Input Vector Distribution

Table D.1: Types of input vector distributions. Overview of the distributions employed as well as the `numpy.random` used to generate the data.

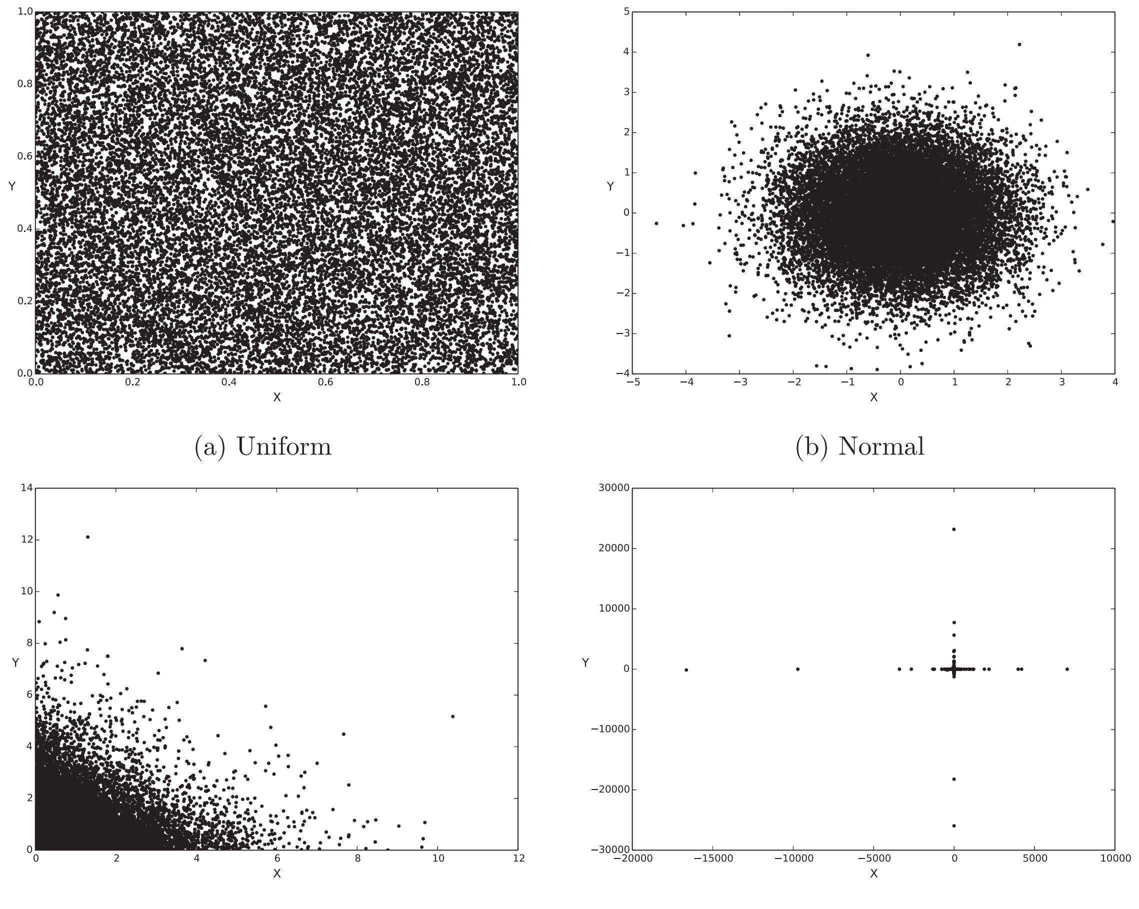
Distribution	`numpy.random` *Function*	*Description*	*Parameters*
Uniform	`uniform`	Uniform distribution.	`low=0`, `high=1`
Normal	`standard_normal`	Normal distribution with a mean of 0 and a standard deviation of 1.	
Exponential	`standard_exponential`	Exponential distribution with a scale parameter beta of 1.	
Cauchy	`standard_cauchy`	Cauchy (Lorentz) distribution with a mode of 0.	

Table D.2: RQA implementation properties.

Property	*Value*
Input data format	Row-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No
Recurrence matrix materialisation	Yes

Table D.3: RQA input parameter assignments.

Parameter	*Value*
Embedding dimension	10
Similarity measure	Euclidean metric
Similarity threshold	10% of the maximum phase space diameter
Number of input vectors	20, 000
Default compiler optimisations	Enabled
Number of experimental runs	100



(a) Uniform

(b) Normal

(c) Exponential

(d) Cauchy (Lorentz)

Figure D.1: Example instances. Each diagram depicts an instance for a specific distribution type from Tab D.1. Each instance refers to a set of 20, 000 randomly generated vectors that reside in two-dimensional space.

D.1.2 Similarity Measure

Table D.4: RQA implementation properties.

Property	*Value*
Input data format	Column-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No
Recurrence matrix materialisation	Yes

Table D.5: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Embedding dimension	10
Time delay	2
Similarity threshold	10% of the maximum phase space diameter
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of experimental runs	100

D.1.3 Default Compiler Optimisations

Table D.6: RQA implementation properties.

Property	*Value*
Input data format	Column-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No
Recurrence matrix materialisation	Yes

Table D.7: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Embedding dimension	10
Time delay	2
Similarity measure	Euclidean
Similarity threshold	10% of the maximum phase space diameter
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of experimental runs	100

D.1.4 Input Data Format

Table D.8: RQA implementation properties.

Property	*Value*
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No
Recurrence matrix materialisation	Yes

Table D.9: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Time delay	2
Similarity measure	Euclidean
Similarity threshold	10% of the maximum phase space diameter
Number of input vectors	20,000
Default compiler optimisations	Enabled
Number of experimental runs	100

D.1.5 Recurrence Matrix Representation

Table D.10: RQA implementation properties.

Property	*Value*
Input data format	Row-wise
Similarity value representation	Byte / 32-bit integer
Intermediate results recycling	No
Recurrence matrix materialisation	Yes

Table D.11: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Embedding dimension	10
Similarity measure	Euclidean
Number of input vectors	$10,000$
Default compiler optimisations	Enabled
Number of experimental runs	100

Table D.12: Recurrence rate. The tabular data refers to Fig. 6.6.

Maximum Phase Space Diameter Ratio	*Recurrence Rate*
0.00	0.0000
0.05	0.0001
0.10	0.0001
0.15	0.0005
0.20	0.0050
0.25	0.0277
0.30	0.1004
0.35	0.2576
0.40	0.4884
0.45	0.7304
0.50	0.8995
0.55	0.9755
0.60	0.9960
0.65	0.9997
0.70	0.9999
0.75	1.0000
0.80	1.0000
0.85	1.0000
0.90	1.0000
0.95	1.0000
1.00	1.0000

D.1.6 Similarity Value Representation

Table D.13: RQA implementation properties.

Property	*Value*
Input data representation	Column-wise
Recurrence matrix representation	Uncompressed
Recurrence matrix materialisation	Yes
Intermediate results recycling	No

Table D.14: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Time delay	2
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.3
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of experimental runs	100

D.1.7 Intermediate Results Recycling

Table D.15: RQA implementation properties.

Property	*Value*
Input data representation	Column-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Recurrence matrix materialisation	Yes

Table D.16: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Embedding dimension	10
Time delay	2
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.4
Default compiler optimisations	Enabled
Number of experimental runs	100

D.1.8 Recurrence Matrix Materialisation

Table D.17: RQA implementation properties.

Property	*Value*
Input data representation	Column-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No

Table D.18: RQA input parameter assignments.

Parameter	*Value*
Distribution	Uniform
Time delay	2
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.0
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of experimental runs	100

D.2 Index Data Structures

D.2.1 Grid Directories

Table D.19: Parallel brute-force implementation properties.

Property	*Value*
Input data representation	Row-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No

Table D.20: RQA input parameter assignments.

Parameter	*Value*
Similarity measure	Euclidean
Number of input vectors	$1,000$
Default compiler optimisations	Enabled
Number of distribution type instances	10

D.2.2 Multi-Dimensional Search Trees

Table D.21: Parallel brute-force implementation properties.

Property	*Value*
Input data representation	Row-wise
Recurrence matrix representation	Uncompressed
Similarity value representation	Byte
Intermediate results recycling	No

Table D.22: RQA input parameter assignments.

Parameter	*Value*
Similarity measure	Euclidean
Number of input vectors	$1,000$
Default compiler optimisations	Enabled
Number of distribution type instances	10

D.3 Automatic Performance Tuning for Implementation Selection

D.3.1 Selection Strategies

Table D.23: RQA input parameter assignments.

Parameter	*Value*
Embedding dimension	5
Time delay	3
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.05
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of distribution type instances	5

Table D.24: RQA measures. Note that instead of showing percentages regarding the values of RR, DET and LAM, the actual ratios are captured.

Measure	*Value*
Recurrence rate (RR)	0.170708
Determinism (DET)	0.954954
Average diagonal line length (D_{mean})	10.296124
Longest diagonal line length (D_{max})	7504
Entropy diagonal lines (D_{entr})	3.002752
Laminarity (LAM)	0.975276
Trapping time (TT)	12.179679
Longest vertical line length (V_{max})	463
Entropy vertical lines (V_{entr})	3.309513
Average white vertical line length (W_{mean})	58.928727
Entropy white vertical lines (W_{entr})	4.618407

Table D.25: Selection strategies parameter assignments. Different parameters are applied, but not each parameter is required by each strategy. The parameter `explore_length` refers to the consecutive amount of random flavour selections during an exploration phase. The parameter `exploit_length` refers to the consecutive amount of sub matrices, to which the best-performing flavours is applied. The parameter `explore_period` is specific to the vw-greedy method. It refers to the consecutive amount of sub matrices, after which the performance data previously gathered is reset. The parameter `delta` is specific to the ϵ-decreasing strategy. It refers to the increase in `exploit_length` after each exploration phase.

Selection Strategy	*Parameter Assignments*

ϵ-greedy	`explore_length = 1`
	`exploit_length = 9`

vw-greedy	`explore_length = 1`
	`exploit_length = 9`
	`epxlore_period = 150`

ϵ-first	`explore_length = 15`

ϵ-decreasing	`explore_length = 1`
	`exploit_length = 9`
	`delta = 1`

random	`explore_length = 1`
	`exploit_length = 0`

D.3.2 Efficiency

Table D.26: RQA input parameter assignments.

Parameter	*Value*
Embedding dimension	5
Time delay	3
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.05
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of distribution type instances	5

D.3.3 Scalability

Table D.27: RQA input parameter assignments.

Parameter	*Value*
Embedding dimension	5
Time delay	3
Similarity measure	Euclidean
Maximum phase space diameter ratio	0.05
Number of input vectors	$20,000$
Default compiler optimisations	Enabled
Number of distribution type instances	5

E Experimental Results

E.1 Parallel Brute-Force Processing

E.1.1 Input Vector Distribution

AMD Radeon RX 470

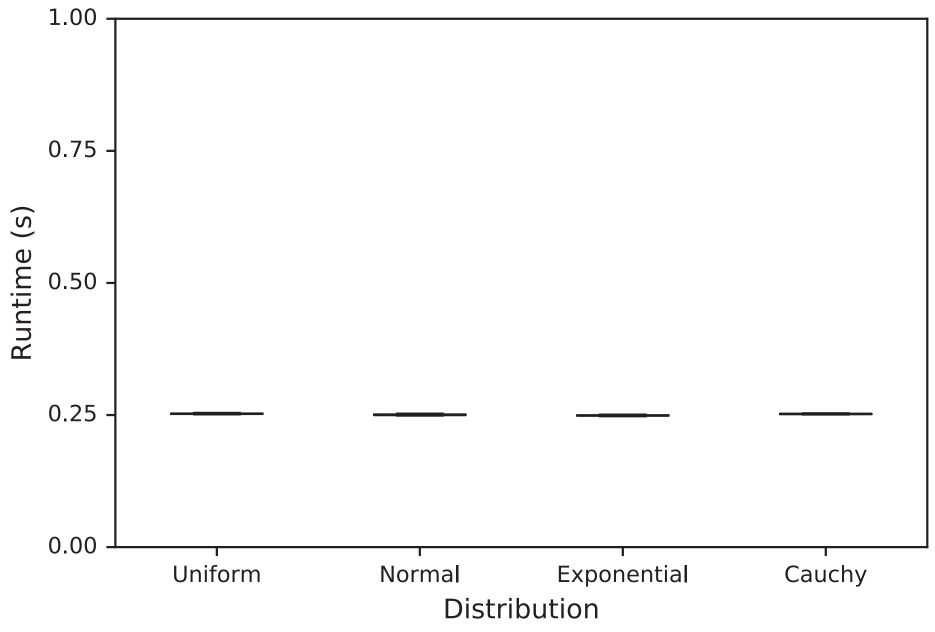


Figure E.1: Runtimes. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.1.

Table E.1: Runtimes. The tabular runtime results referring to Fig. E.1.

Distribution	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Uniform	0.2512	0.2521	0.2524	0.2529	0.2541
Normal	0.2489	0.2498	0.2505	0.2511	0.2526
Exponential	0.2477	0.2487	0.2491	0.2496	0.2512
Cauchy	0.2510	0.2517	0.2520	0.2524	0.2535

Nvidia GeForce GTX 690

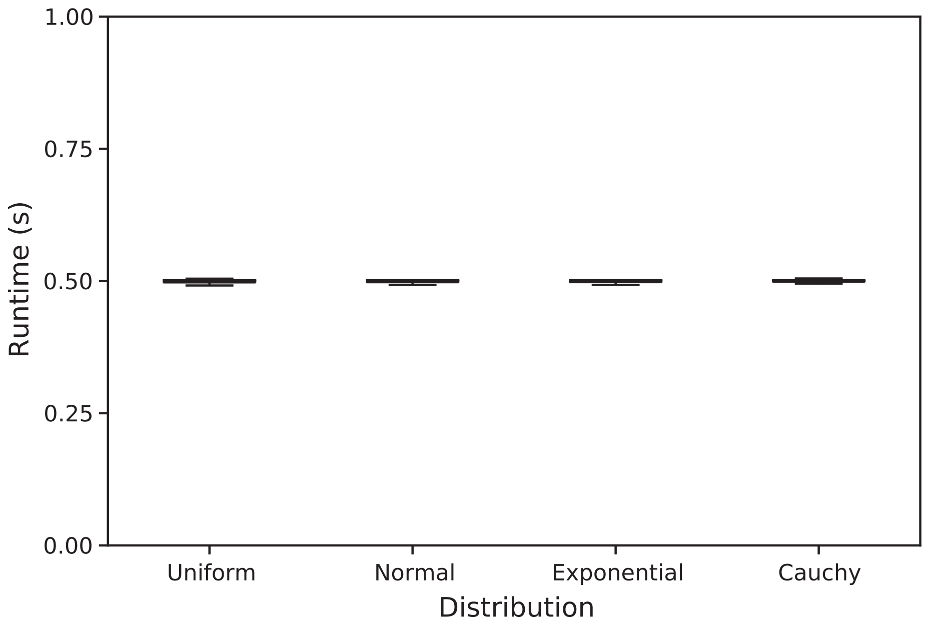


Figure E.2: Runtimes. The OpenCL kernels are executed on the Nvidia GeForce GTX 690 compute device. A single boxplot is depicted for each distribution. The runtimes are captured in tabular fashion in Tab. E.2.

Table E.2: Runtimes. The tabular runtime results referring to Fig. E.2.

Distribution	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Uniform	0.4916	0.4975	0.5015	0.5015	0.5271
Normal	0.4916	0.4978	0.5015	0.5015	0.5015
Exponential	0.4923	0.4979	0.5015	0.5015	0.5015
Cauchy	0.4924	0.4990	0.5011	0.5015	0.5049

Intel Xeon E5620

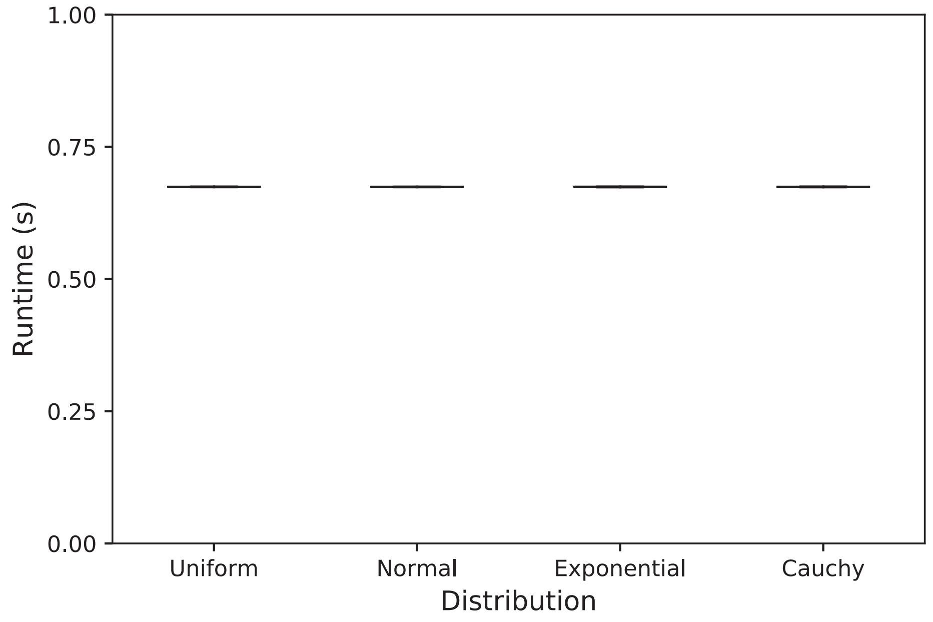


Figure E.3: Runtimes. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.3.

Table E.3: Runtimes. The tabular runtime results referring to Fig. E.3.

Distribution	*Runtime* (*s*)				
	Min	25%	50%	75%	*Max*
Uniform	0.6737	0.6742	0.6743	0.6745	0.6751
Normal	0.6738	0.6741	0.6743	0.6745	0.6757
Exponential	0.6736	0.6742	0.6744	0.6746	0.6759
Cauchy	0.6737	0.6742	0.6743s	0.6745	0.6762

E.1.2 Similarity Measure

AMD Radeon RX 470

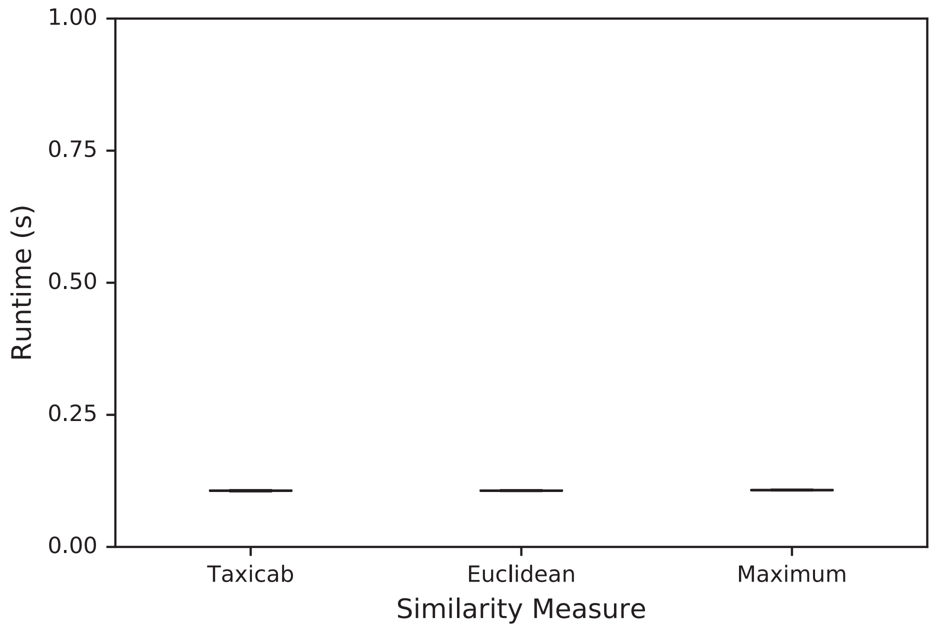


Figure E.4: Runtimes. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.4.

Table E.4: Runtimes. The tabular runtime results referring to Fig. E.4.

Similarity Measure	*Runtime* (*s*)				
	Min	25%	50%	75%	*Max*
Taxicab	0.0943	0.1061	0.1066	0.1068	0.1074
Euclidean	0.0953	0.1063	0.1065	0.1068	0.1079
Maximum	0.0955	0.1073	0.1076	0.1079	0.1084

Nvidia GeForce GTX 690

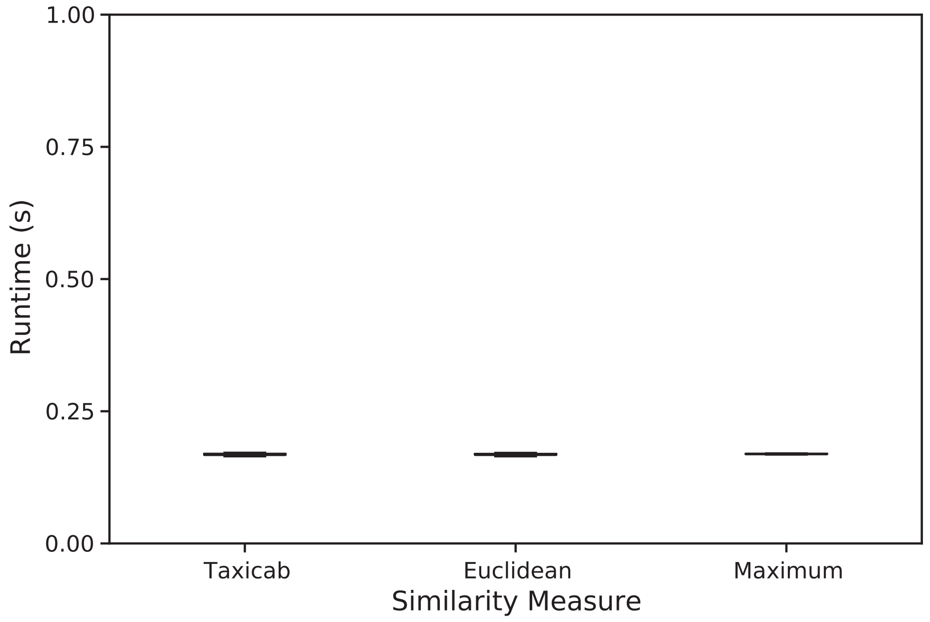


Figure E.5: Runtimes. The OpenCL kernels are executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.5.

Table E.5: Runtimes. The tabular runtime results referring to Fig. E.5.

Similarity Measure	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Taxicab	0.1648	0.1673	0.1692	0.1694	0.1888
Euclidean	0.1648	0.1674	0.1692	0.1693	0.1734
Maximum	0.1653	0.1689	0.1694	0.1695	0.1736

Intel Xeon E5620

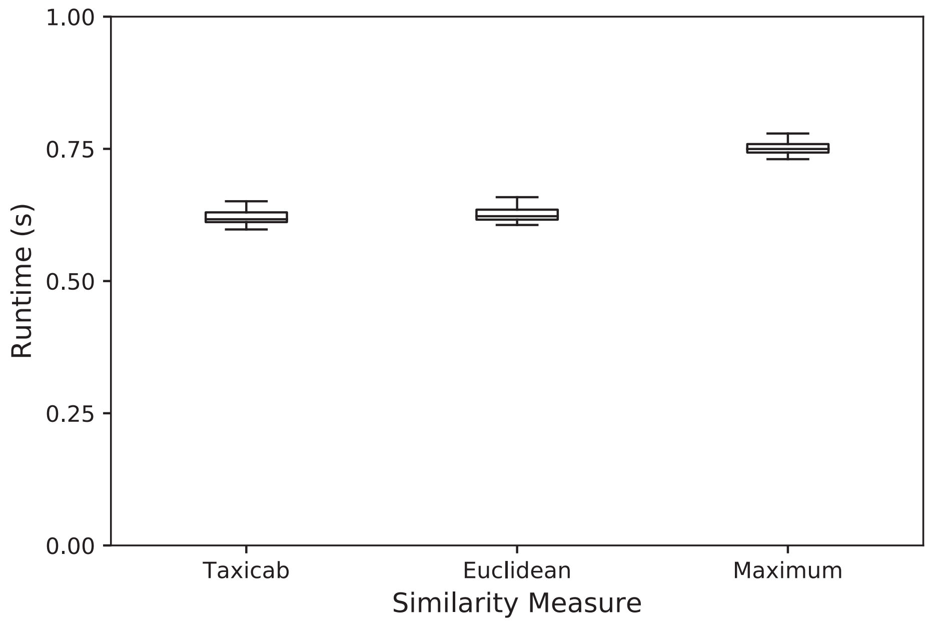


Figure E.6: Runtimes. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.6.

Table E.6: Runtimes. The tabular runtime results referring to Fig. E.6.

Similarity Measure	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Taxicab	0.5976	0.6115	0.6169	0.6299	0.6788
Euclidean	0.6061	0.6161	0.6226	0.6350	0.6586
Maximum	0.7305	0.7430	0.7498	0.7590	0.7984

E.1.3 Default Compiler Optimisations

AMD Radeon RX 470

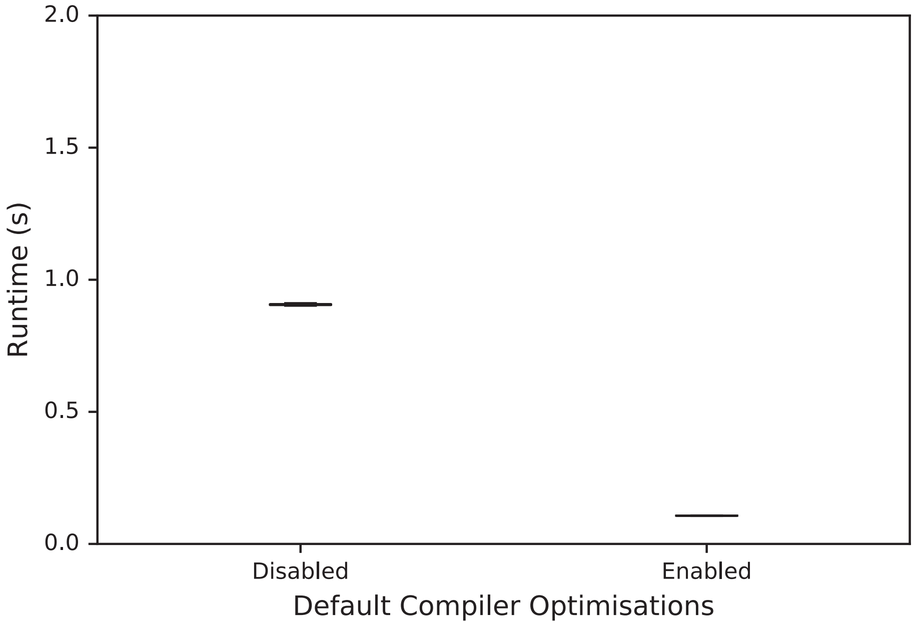


Figure E.7: Runtimes. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.7.

Table E.7: Runtimes. The tabular runtime results referring to Fig. E.7.

Default Compiler Optimisations	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Disabled	0.9019	0.9044	0.9059	0.9077	0.9178
Enabled	0.1060	0.1067	0.1069	0.1071	0.1075

Nvidia GeForce GTX 690

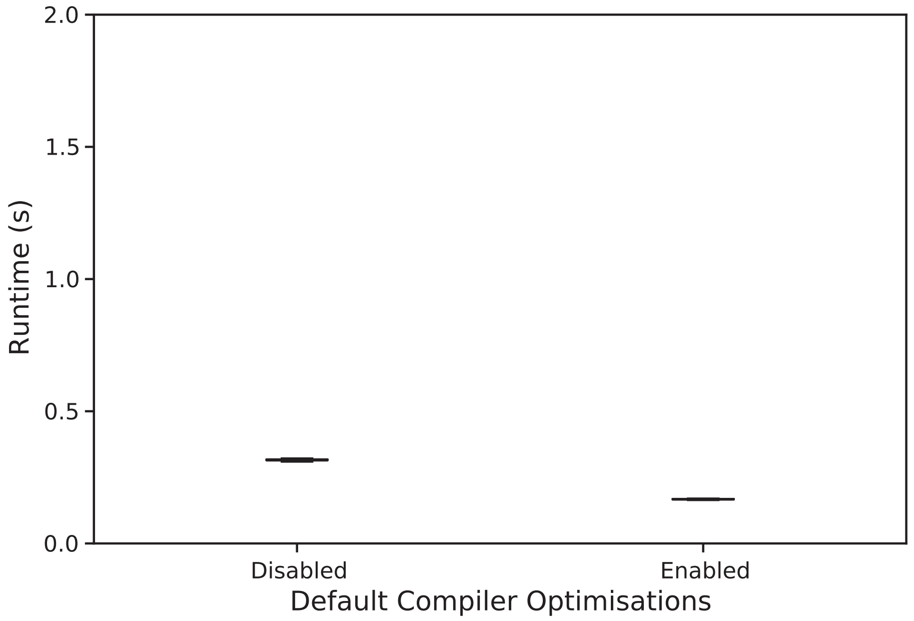


Figure E.8: Runtimes. The OpenCL kernels are executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.8.

Table E.8: Runtimes. The tabular runtime results referring to Fig. E.8.

Default Compiler Optimisations	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Disabled	0.3078	0.3141	0.3167	0.3177	0.3486
Enabled	0.1643	0.1661	0.1678	0.1681	0.1696

Intel Xeon E5620

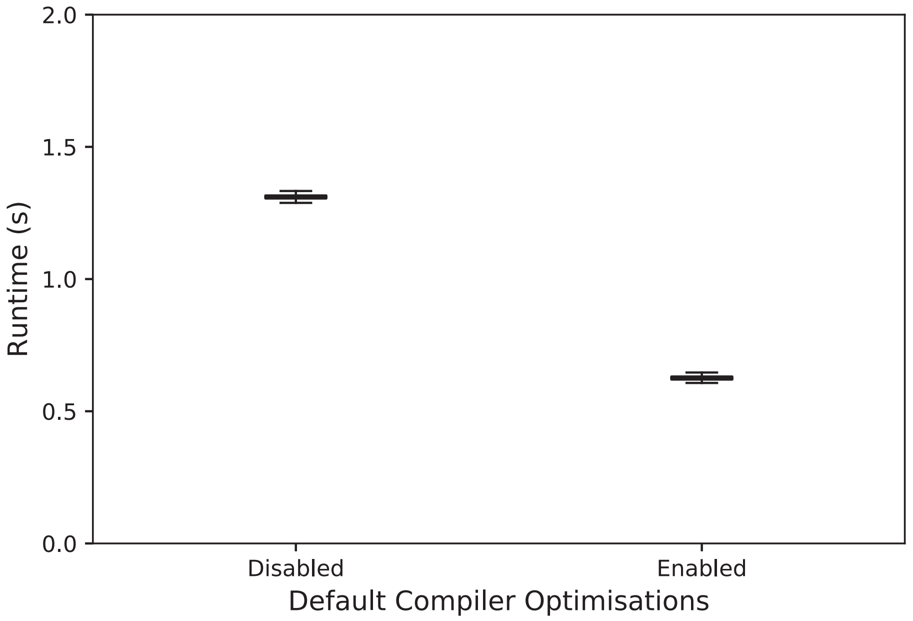


Figure E.9: Runtimes. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.9.

Table E.9: Runtimes. The tabular runtime results referring to Fig. E.9.

Default Compiler Optimisations	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Disabled	1.2875	1.3048	1.3106	1.3162	1.3365
Enabled	0.6070	0.6199	0.6255	0.6314	0.6706

E.1.4 Input Data Format

AMD Radeon RX 470

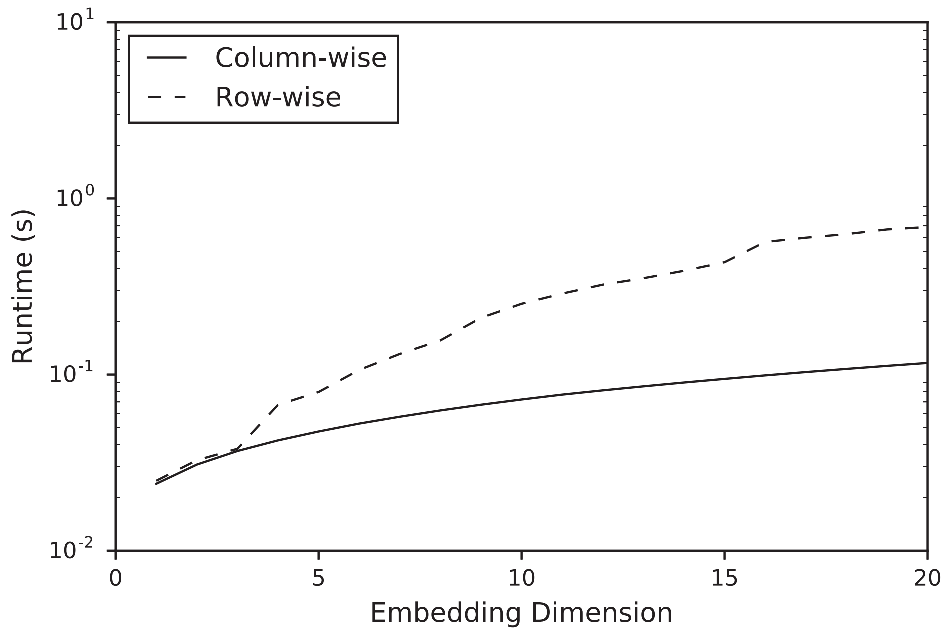


Figure E.10: Runtimes. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.10.

Table E.10: Runtimes. The tabular runtime results of Fig. E.10. t_{Column} and t_{Row} refer to the runtimes while applying the column-wise and row-wise input data format.

Embedding Dimension	*Runtime (s)*		$\frac{t_{Row}}{t_{Column}}$
	t_{Column}	t_{Row}	
1	0.0240	0.0249	1.0399
2	0.0308	0.0326	1.0586
3	0.0368	0.0379	1.0296
4	0.0423	0.0673	1.5918
5	0.0475	0.0796	1.6760
6	0.0527	0.1062	2.0165
7	0.0576	0.1310	2.2728
8	0.0626	0.1563	2.4982
9	0.0674	0.2100	3.1149
10	0.0722	0.2523	3.4953
11	0.0769	0.2883	3.7464
12	0.0813	0.3240	3.9828
13	0.0857	0.3524	4.1110
14	0.0901	0.3883	4.3089
15	0.0945	0.4344	4.5973
16	0.0989	0.5662	5.7236
17	0.1033	0.5986	5.7948
18	0.1077	0.6272	5.8258
19	0.1120	0.6668	5.9522
20	0.1164	0.6884	5.9144

Table E.11: Additional performance counters. The counters are retrieved while executing the OpenCL kernel that refers to the *create_recurrence_matrix* operator on the AMD Radeon RX 470 compute device.

Embedding Dimension	`CacheHit` (%)		`FetchSize` (kB)		`MemUnitStalled` (%)	
	c_{Column}	c_{Row}	f_{Column}	f_{Row}	m_{Column}	m_{Row}
1	84.66	84.48	156.81	156.81	19.18	17.70
2	84.33	90.75	156.94	313.06	12.27	13.24
3	82.99	93.72	156.94	469.31	8.10	8.57
4	81.56	98.47	156.94	625.56	5.20	21.72
5	80.81	98.83	156.94	781.81	3.86	15.41
6	80.51	99.22	156.94	938.06	3.10	16.41
7	80.40	99.40	156.94	1,094.31	2.64	14.78
8	80.35	99.50	156.94	1,252.06	2.33	12.71
9	80.35	99.66	156.94	1,694.31	2.09	11.90
10	81.07	99.72	157.06	4,318.06	1.90	11.71
11	81.07	99.73	157.06	42,814.62	1.74	10.28
12	81.10	99.74	157.06	89,768.44	1.61	11.34
13	81.12	99.78	157.06	48,143.00	1.50	8.84
14	81.12	99.76	157.06	156,703.50	1.40	8.04
15	81.14	99.69	157.06	395,528.00	1.32	6.87
16	81.15	99.67	157.06	709,689.12	1.24	6.16
17	81.16	99.59	157.06	1,057,299.56	1.17	5.77
18	81.88	99.55	157.19	1,291,943.12	1.11	6.20
19	81.88	99.38	157.19	2,033,561.50	1.06	5.88
20	81.91	99.22	157.19	2,780,607.06	1.01	7.03

Nvidia GeForce GTX 690

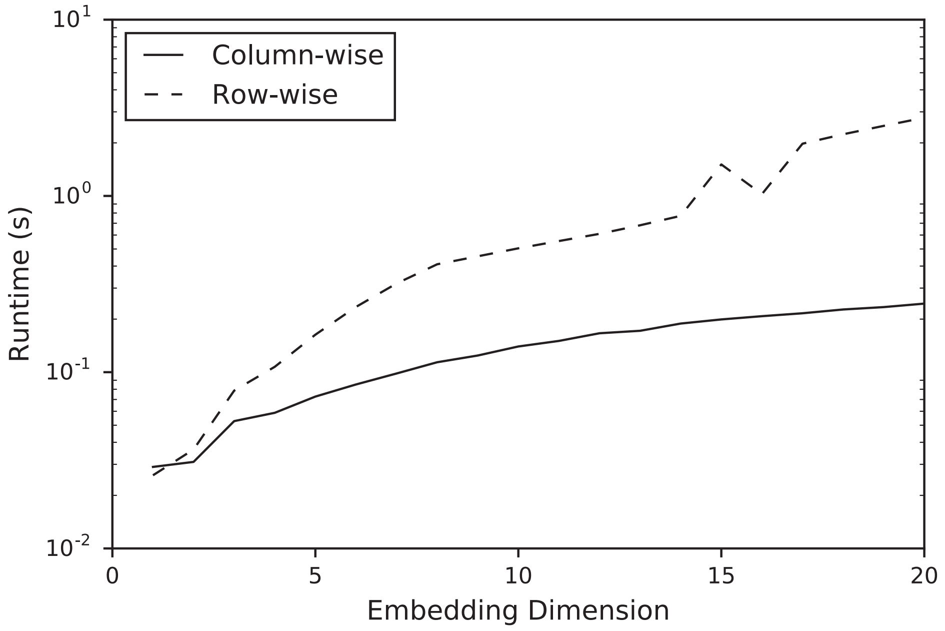


Figure E.11: Runtimes. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.12.

Table E.12: Runtimes. The tabular runtime results of Fig. E.11. t_{Column} and t_{Row} refer to the runtimes while applying the column-wise and row-wise input data format.

Embedding Dimension	*Runtime* (s)		$\frac{t_{Row}}{t_{Column}}$
	t_{Column}	t_{Row}	
1	0.0290	0.0260	0.8963
2	0.0310	0.0364	1.1761
3	0.0528	0.0786	1.4893
4	0.0588	0.1073	1.8239
5	0.0727	0.1632	2.2446
6	0.0852	0.2345	2.7536
7	0.0983	0.3185	3.2400
8	0.1139	0.4094	3.5959
9	0.1243	0.4544	3.6549
10	0.1399	0.5044	3.6052
11	0.1505	0.5547	3.6869
12	0.1663	0.6094	3.6648
13	0.1717	0.6816	3.9687
14	0.1888	0.7708	4.0832
15	0.1992	1.5088	7.5737
16	0.2078	1.0236	4.9250
17	0.2159	1.9787	9.1646
18	0.2269	2.2377	9.8641
19	0.2340	2.4974	10.6712
20	0.2449	2.7783	11.3424

Intel Xeon E5620

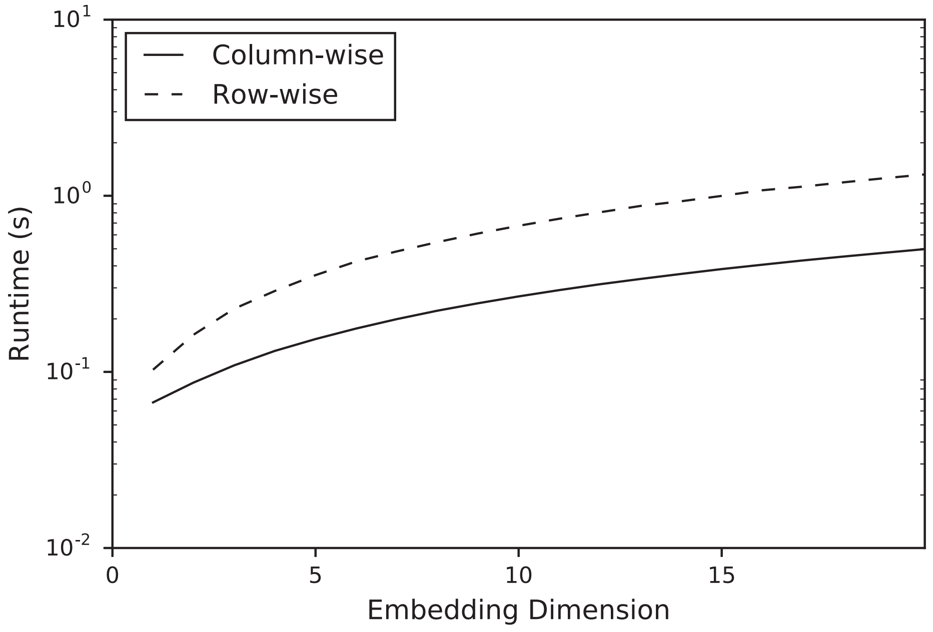


Figure E.12: Runtimes. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.13.

Table E.13: Runtimes. The tabular runtime results of Fig. E.12. t_{Column} and t_{Row} refer to the runtimes while applying the column-wise and row-wise input data format.

Embedding Dimension	*Runtime (s)*		$\frac{t_{Row}}{t_{Column}}$
	t_{Column}	t_{Row}	
1	0.0671	0.1029	1.5331
2	0.0870	0.1627	1.8700
3	0.1090	0.2285	2.0973
4	0.1316	0.2882	2.1897
5	0.1536	0.3543	2.3060
6	0.1763	0.4235	2.4030
7	0.1993	0.4831	2.4238
8	0.2226	0.5454	2.4507
9	0.2453	0.6105	2.4891
10	0.2683	0.6754	2.5176
11	0.2913	0.7406	2.5424
12	0.3145	0.8057	2.5621
13	0.3370	0.8755	2.5978
14	0.3599	0.9307	2.5858
15	0.3835	0.9984	2.6035
16	0.4060	1.0749	2.6473
17	0.4294	1.1269	2.6246
18	0.4518	1.1922	2.6388
19	0.4745	1.2577	2.6503
20	0.4979	1.3214	2.6540

E.1.5 Recurrence Matrix Representation

AMD Radeon RX 470

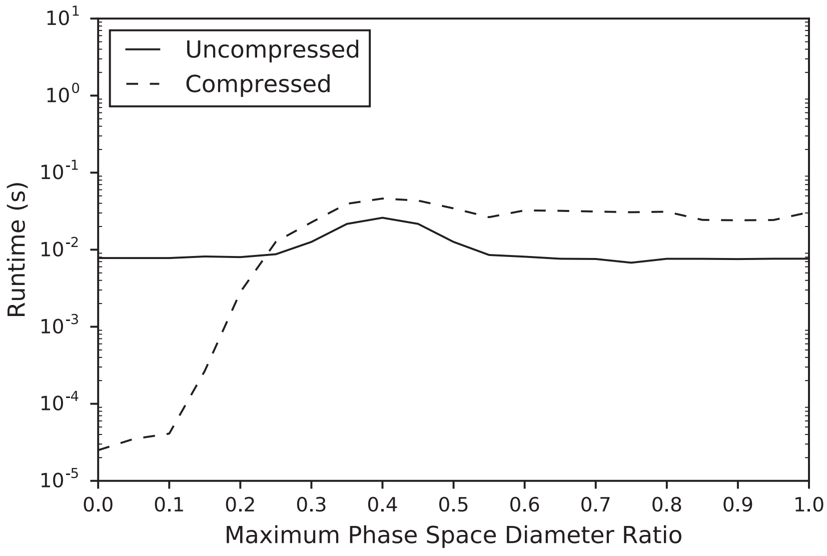


Figure E.13: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.14.

Table E.14: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.13. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime* (s)	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0078	0.00003
0.05	0.0078	0.00004
0.10	0.0078	0.00004
0.15	0.0082	0.0003
0.20	0.0080	0.0028
0.25	0.0087	0.0127
0.30	0.0126	0.0227
0.35	0.0216	0.0395
0.40	0.0260	0.0461
0.45	0.0217	0.0434
0.50	0.0126	0.0344
0.55	0.0085	0.0264
0.60	0.0081	0.0324
0.65	0.0076	0.0320
0.70	0.0076	0.0313
0.75	0.0068	0.0307
0.80	0.0076	0.0311
0.85	0.0076	0.0244
0.90	0.0076	0.0241
0.95	0.0076	0.0243
1.00	0.0076	0.0306

Table E.15: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.14. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime (s)*	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0071	0.2342
0.05	0.0078	0.1520
0.10	0.0078	0.1475
0.15	0.0075	0.2102
0.20	0.0075	0.1700
0.25	0.0078	0.1979
0.30	0.0080	0.1666
0.35	0.0100	0.2132
0.40	0.0103	0.2202
0.45	0.0085	0.2618
0.50	0.0078	0.2428
0.55	0.0062	0.2606
0.60	0.0075	0.2599
0.65	0.0075	0.2389
0.70	0.0071	0.2494
0.75	0.0047	0.2527
0.80	0.0075	0.2459
0.85	0.0071	0.2482
0.90	0.0050	0.2598
0.95	0.0071	0.2327
1.00	0.0071	0.2498

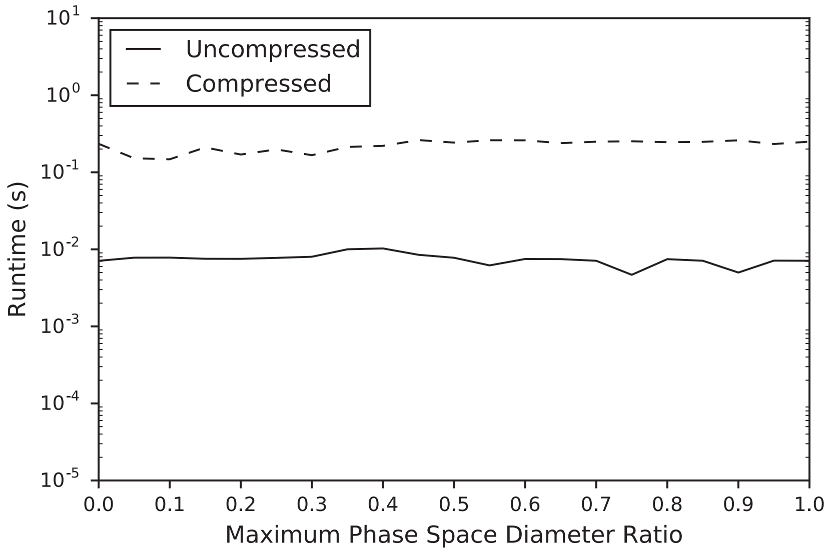


Figure E.14: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.15.

Nvidia GeForce GTX 690

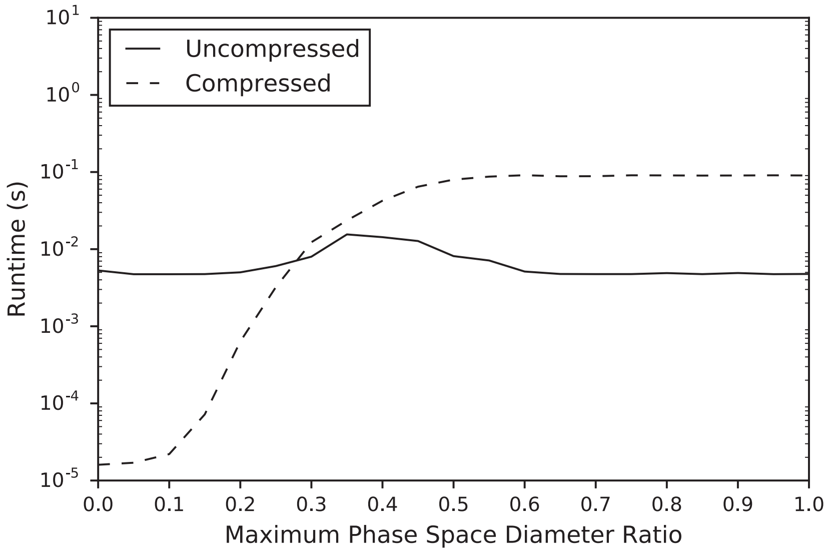


Figure E.15: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.16.

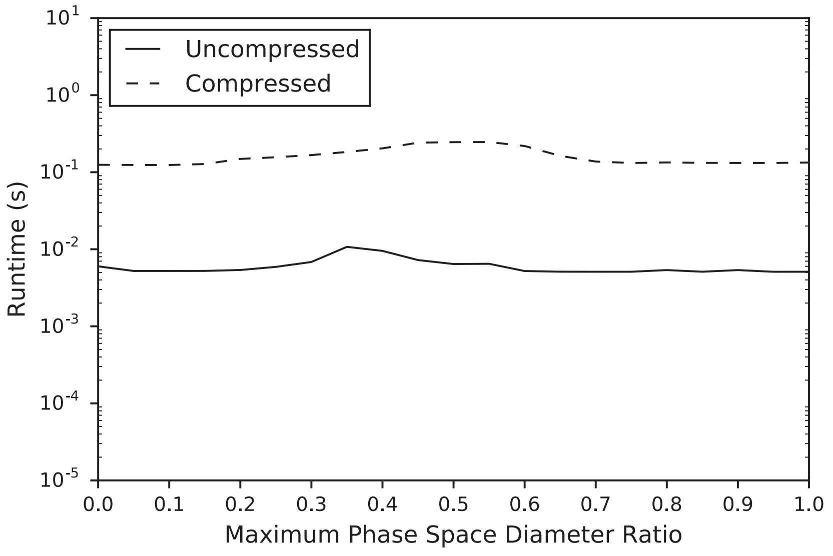


Figure E.16: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.17.

Table E.16: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.15. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime* (*s*)	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0053	0.00002
0.05	0.0047	0.00002
0.10	0.0047	0.00002
0.15	0.0048	0.00007
0.20	0.0050	0.0006
0.25	0.0060	0.0032
0.30	0.0080	0.0123
0.35	0.0155	0.0236
0.40	0.0143	0.0425
0.45	0.0128	0.0646
0.50	0.0081	0.0797
0.55	0.0071	0.0872
0.60	0.0051	0.0910
0.65	0.0048	0.0881
0.70	0.0048	0.0883
0.75	0.0048	0.0907
0.80	0.0049	0.0904
0.85	0.0048	0.0900
0.90	0.0049	0.0903
0.95	0.0047	0.0907
1.00	0.0048	0.0903

Table E.17: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.16. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime (s)*	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0060	0.1247
0.05	0.0052	0.1242
0.10	0.0052	0.1239
0.15	0.0052	0.1277
0.20	0.0054	0.1486
0.25	0.0059	0.1563
0.30	0.0068	0.1668
0.35	0.0107	0.1832
0.40	0.0095	0.2039
0.45	0.0072	0.2415
0.50	0.0064	0.2454
0.55	0.0065	0.2464
0.60	0.0052	0.2188
0.65	0.0051	0.1629
0.70	0.0051	0.1374
0.75	0.0051	0.1316
0.80	0.0054	0.1338
0.85	0.0051	0.1322
0.90	0.0054	0.1318
0.95	0.0051	0.1318
1.00	0.0051	0.1335

Intel Xeon E5620

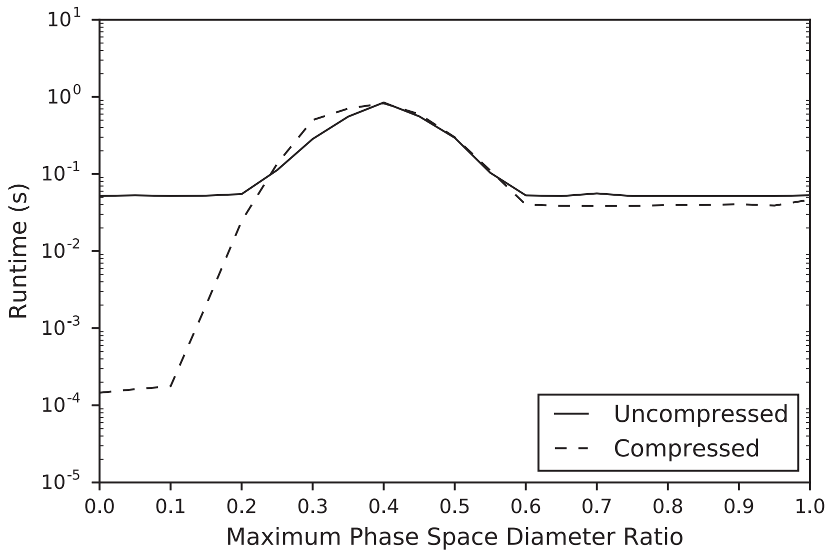


Figure E.17: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.18.

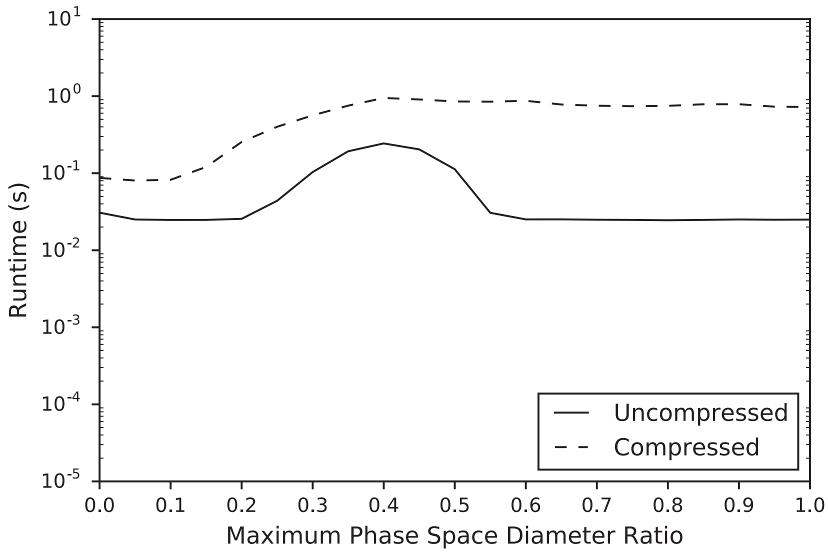


Figure E.18: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.19.

Table E.18: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.17. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime (s)*	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0519	0.0002
0.05	0.0530	0.0002
0.10	0.0518	0.0002
0.15	0.0525	0.0020
0.20	0.0550	0.0246
0.25	0.1126	0.1362
0.30	0.2851	0.5022
0.35	0.5558	0.7073
0.40	0.8467	0.8266
0.45	0.5596	0.5974
0.50	0.2951	0.2995
0.55	0.1038	0.1109
0.60	0.0529	0.0401
0.65	0.0518	0.0388
0.70	0.0561	0.0384
0.75	0.0518	0.0385
0.80	0.0519	0.0396
0.85	0.0518	0.0396
0.90	0.0518	0.0406
0.95	0.0517	0.0391
1.00	0.0531	0.0464

Table E.19: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.18. $t_{Uncompressed}$ and $t_{Compressed}$ refer to the runtimes while using the uncompressed as well as the compressed sparse recurrence matrix representation.

Maximum Phase Space Diameter Ratio	*Runtime (s)*	
	$t_{Uncompressed}$	$t_{Compressed}$
0.00	0.0308	0.0869
0.05	0.0251	0.0803
0.10	0.0247	0.0821
0.15	0.0248	0.1205
0.20	0.0256	0.2544
0.25	0.0439	0.4004
0.30	0.1035	0.5639
0.35	0.1926	0.7528
0.40	0.2435	0.9470
0.45	0.2038	0.9055
0.50	0.1127	0.8527
0.55	0.0306	0.8473
0.60	0.0251	0.8704
0.65	0.0252	0.7787
0.70	0.0249	0.7509
0.75	0.0248	0.7407
0.80	0.0245	0.7482
0.85	0.0248	0.7847
0.90	0.0251	0.7857
0.95	0.0249	0.7301
1.00	0.0250	0.7228

E.1.6 Similarity Value Representation

AMD Radeon RX 470

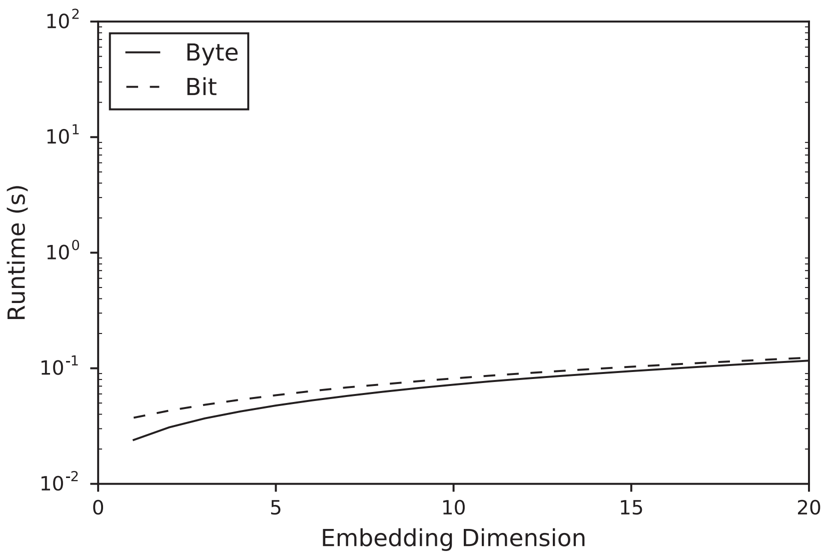


Figure E.19: Runtimes for creating the recurrence matrix. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.20.

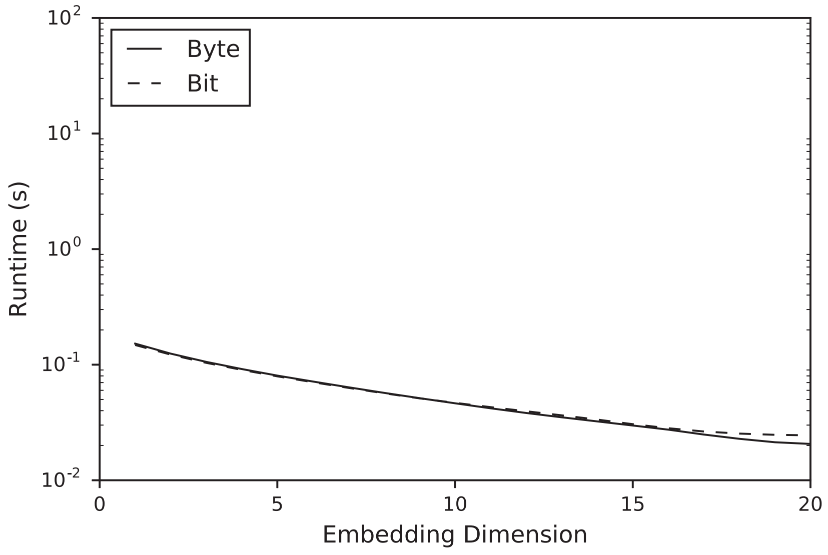


Figure E.20: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.21.

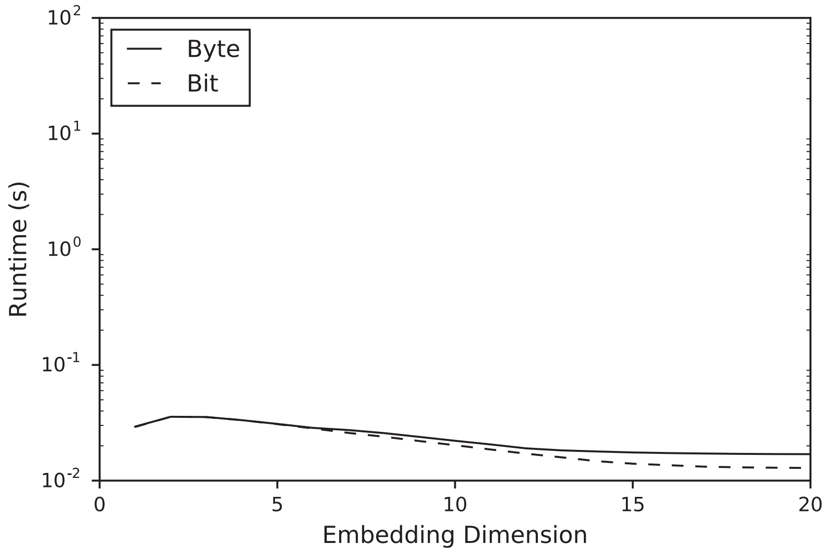


Figure E.21: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.22.

Table E.20: Runtimes for creating the recurrence matrix. The tabular runtime results of Fig. E.19. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0240	0.0373
2	0.0308	0.0430
3	0.0368	0.0483
4	0.0423	0.0535
5	0.0476	0.0585
6	0.0527	0.0635
7	0.0576	0.0683
8	0.0626	0.0726
9	0.0674	0.0773
10	0.0721	0.0817
11	0.0769	0.0862
12	0.0814	0.0905
13	0.0858	0.0948
14	0.0901	0.0989
15	0.0945	0.1031
16	0.0989	0.1073
17	0.1033	0.1114
18	0.1077	0.1155
19	0.1120	0.1195
20	0.1164	0.1235

Table E.21: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.20. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.1521	0.1480
2	0.1246	0.1222
3	0.1058	0.1037
4	0.0917	0.0902
5	0.0804	0.0793
6	0.0715	0.0706
7	0.0638	0.0630
8	0.0571	0.0566
9	0.0513	0.0511
10	0.0463	0.0467
11	0.0419	0.0429
12	0.0382	0.0395
13	0.0350	0.0365
14	0.0323	0.0335
15	0.0298	0.0306
16	0.0274	0.0282
17	0.0249	0.0264
18	0.0228	0.0253
19	0.0213	0.0247
20	0.0206	0.0245

Table E.22: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.21. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0293	0.0291
2	0.0357	0.0357
3	0.0355	0.0354
4	0.0333	0.0333
5	0.0309	0.0308
6	0.0286	0.0283
7	0.0274	0.0259
8	0.0258	0.0240
9	0.0239	0.0220
10	0.0221	0.0202
11	0.0206	0.0186
12	0.0190	0.0171
13	0.0183	0.0159
14	0.0179	0.0147
15	0.0175	0.0140
16	0.0173	0.0136
17	0.0172	0.0132
18	0.0170	0.0130
19	0.0170	0.0129
20	0.0169	0.0129

Nvidia GeForce GTX 690

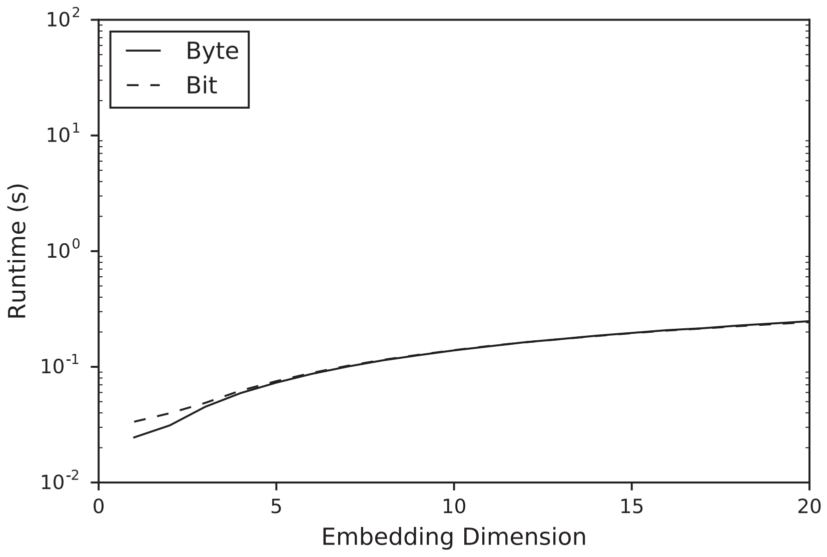


Figure E.22: Runtimes for creating the recurrence matrix. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.23.

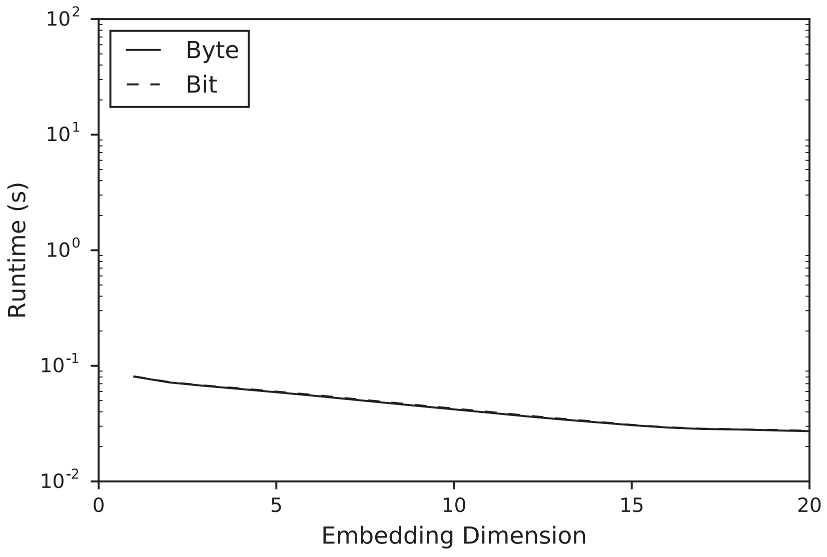


Figure E.23: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.24.

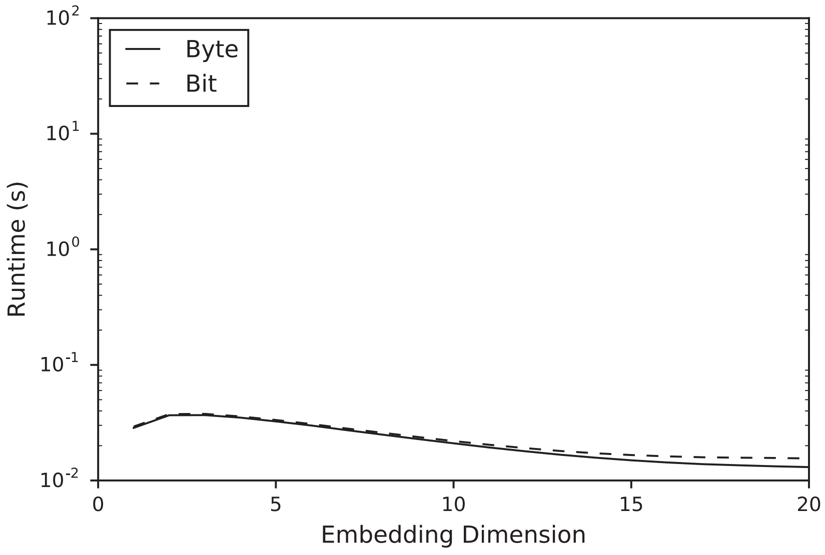


Figure E.24: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.25.

Table E.23: Runtimes for creating the recurrence matrix. The tabular runtime results of Fig. E.22. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0246	0.0335
2	0.0312	0.0397
3	0.0451	0.0488
4	0.0594	0.0623
5	0.0730	0.0752
6	0.0870	0.0886
7	0.1005	0.1020
8	0.1139	0.1149
9	0.1260	0.1271
10	0.1388	0.1394
11	0.1506	0.1516
12	0.1633	0.1630
13	0.1738	0.1736
14	0.1857	0.1847
15	0.1963	0.1954
16	0.2076	0.2059
17	0.2156	0.2146
18	0.2276	0.2247
19	0.2373	0.2341
20	0.2481	0.2436

Table E.24: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.23. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0806	0.0809
2	0.0718	0.0722
3	0.0670	0.0675
4	0.0630	0.0637
5	0.0591	0.0598
6	0.0553	0.0562
7	0.0516	0.0523
8	0.0481	0.0489
9	0.0449	0.0456
10	0.0420	0.0427
11	0.0393	0.0399
12	0.0367	0.0372
13	0.0344	0.0348
14	0.0325	0.0329
15	0.0307	0.0308
16	0.0293	0.0294
17	0.0284	0.0285
18	0.0281	0.0282
19	0.0276	0.0278
20	0.0272	0.0274

Table E.25: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.24. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0285	0.0293
2	0.0367	0.0375
3	0.0368	0.0377
4	0.0350	0.0358
5	0.0324	0.0334
6	0.0300	0.0308
7	0.0272	0.0282
8	0.0249	0.0259
9	0.0228	0.0238
10	0.0210	0.0219
11	0.0193	0.0204
12	0.0179	0.0191
13	0.0167	0.0180
14	0.0157	0.0172
15	0.0150	0.0166
16	0.0143	0.0162
17	0.0139	0.0159
18	0.0135	0.0158
19	0.0133	0.0157
20	0.0131	0.0155

Intel Xeon E5620

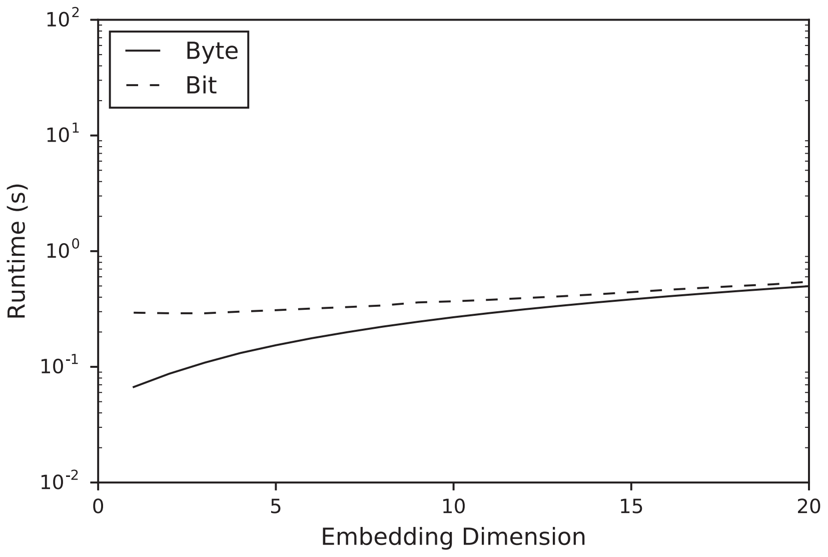


Figure E.25: Runtimes for creating the recurrence matrix. The OpenCL kernel that refers to the *create_recurrence_matrix* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.26.

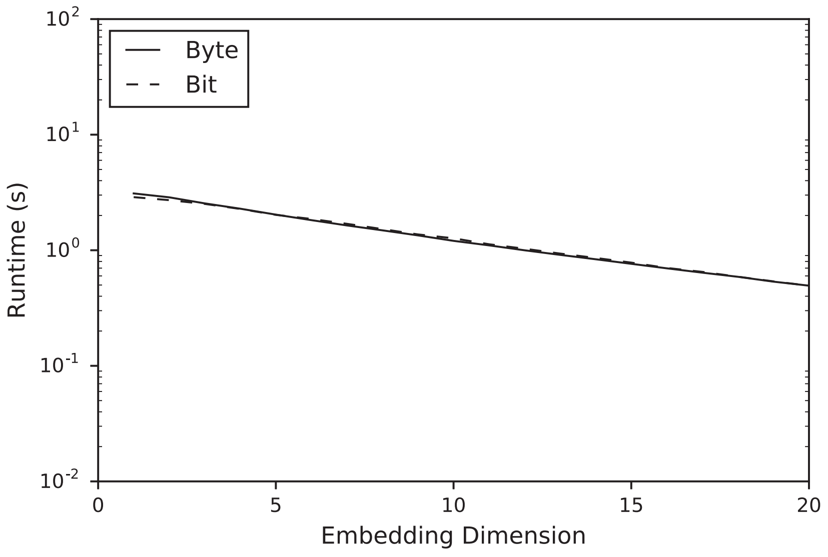


Figure E.26: Runtimes for detecting vertical lines. The OpenCL kernel that refers to the *detect_vertical_lines* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.27.

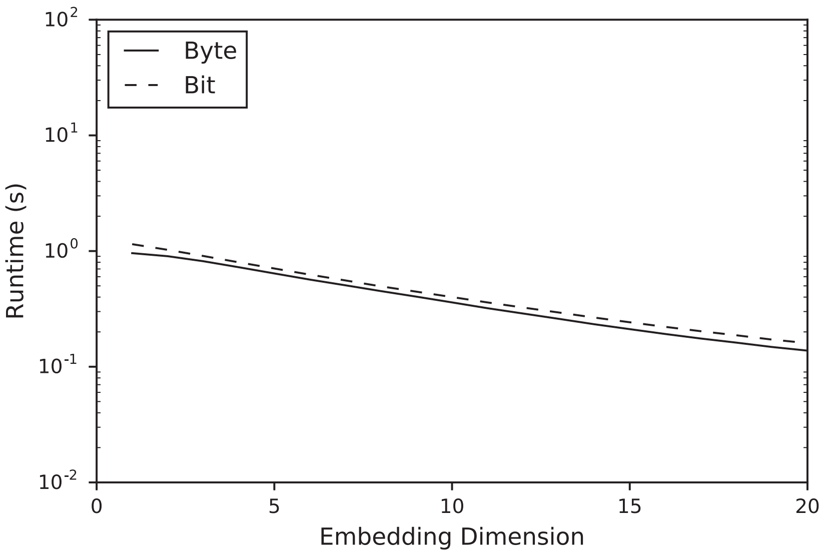


Figure E.27: Runtimes for detecting diagonal lines. The OpenCL kernel that refers to the *detect_diagonal_lines* operator is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.28.

Table E.26: Runtimes for creating the recurrence matrix. The tabular runtime results of Fig. E.25. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.0669	0.2939
2	0.0872	0.2905
3	0.1087	0.2903
4	0.1316	0.3002
5	0.1538	0.3088
6	0.1765	0.3191
7	0.1992	0.3286
8	0.2224	0.3392
9	0.2452	0.3605
10	0.2685	0.3686
11	0.2913	0.3796
12	0.3143	0.3930
13	0.3370	0.4068
14	0.3601	0.4231
15	0.3829	0.4421
16	0.4061	0.4633
17	0.4289	0.4813
18	0.4520	0.5002
19	0.4749	0.5173
20	0.4980	0.5442

Table E.27: Runtimes for detecting vertical lines. The tabular runtime results of Fig. E.26. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	3.1020	2.8803
2	2.8703	2.7130
3	2.5439	2.5171
4	2.2881	2.2763
5	2.0336	2.0234
6	1.8216	1.8656
7	1.6394	1.6930
8	1.4865	1.5238
9	1.3441	1.3704
10	1.2056	1.2692
11	1.1020	1.1266
12	0.9990	1.0304
13	0.9119	0.9355
14	0.8355	0.8576
15	0.7631	0.7816
16	0.6975	0.7028
17	0.6396	0.6499
18	0.5893	0.5903
19	0.5349	0.5379
20	0.4938	0.4936

Table E.28: Runtimes for detecting diagonal lines. The tabular runtime results of Fig. E.27. t_{Byte} and t_{Bit} refer to the runtimes while using the byte-wise and bit-wise similarity value representation.

Embedding Dimension	*Runtime (s)*	
	t_{Byte}	t_{Bit}
1	0.9590	1.1467
2	0.9025	1.0246
3	0.8170	0.9067
4	0.7246	0.7991
5	0.6404	0.7067
6	0.5665	0.6252
7	0.5058	0.5572
8	0.4506	0.4952
9	0.4034	0.4455
10	0.3598	0.4002
11	0.3204	0.3595
12	0.2884	0.3249
13	0.2595	0.2944
14	0.2330	0.2662
15	0.2113	0.2423
16	0.1919	0.2212
17	0.1753	0.2028
18	0.1616	0.1870
19	0.1482	0.1717
20	0.1378	0.1601

E.1.7 Intermediate Results Recycling

AMD Radeon RX 470

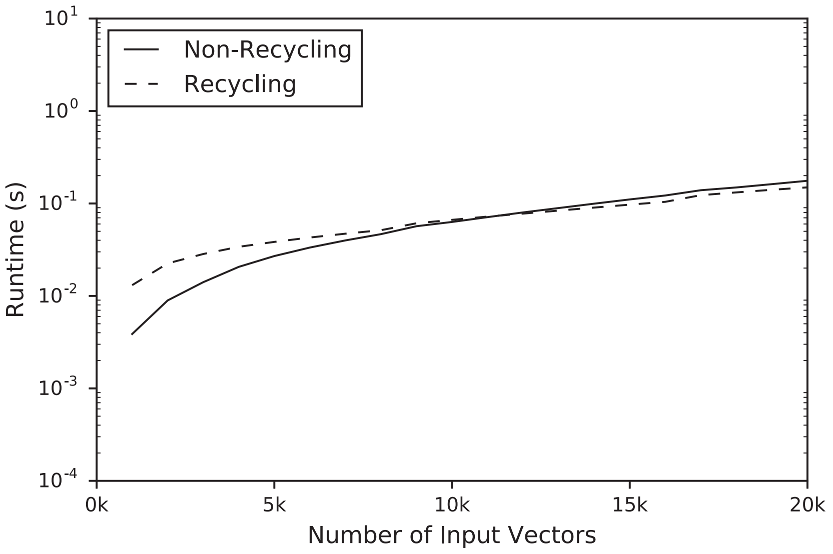


Figure E.28: Runtimes. The OpenCL kernels that conduct the computations of the pairwise input vector similarities as well as the detection of vertical lines are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.29.

Table E.29: Runtimes. The tabular runtime results of Fig. E.28. $t_{Non-Recycling}$ and $t_{Recycling}$ refer to the runtimes while using the non-recycling and recycling computing approach.

Number of Input Vectors	*Runtime (s)*	
	$t_{Non-Recycling}$	$t_{Recycling}$
1,000	0.0039	0.0130
2,000	0.0090	0.0225
3,000	0.0141	0.0284
4,000	0.0206	0.0339
5,000	0.0269	0.0383
6,000	0.0334	0.0427
7,000	0.0397	0.0471
8,000	0.0466	0.0514
9,000	0.0568	0.0611
10,000	0.0630	0.0665
11,000	0.0714	0.0722
12,000	0.0799	0.0777
13,000	0.0896	0.0839
14,000	0.0998	0.0902
15,000	0.1109	0.0970
16,000	0.1221	0.1041
17,000	0.1393	0.1234
18,000	0.1493	0.1313
19,000	0.1620	0.1407
20,000	0.1764	0.1493

Table E.30: Additional performance counters. The counters are retrieved while executing the OpenCL kernels that compute the pairwise input vector similarities as well as perform the detection of vertical lines on the AMD Radeon RX 470 compute device.

Number of Input Vectors	`FetchSize` (kB)		`WriteSize` (kB)	
	$f_{Non-Recycling}$	$f_{Recycling}$	$w_{Non-Recycling}$	$w_{Recycling}$
1,000	998.31	21.44	988.90	988.90
2,000	3,947.73	41.21	3,930.63	3930.67
3,000	8,850.48	61.23	8,825.76	8832.00
4,000	15,713.37	81.72	15,674.23	15,703.24
5,000	24,772.90	102.64	24,476.00	24,923.35
6,000	35,866.43	125.77	35,231.16	36,233.15
7,000	49,352.84	159.15	47,939.56	50,003.52
8,000	62,668.14	199.68	62,601.05	62,601.23
9,000	83,609.28	250.21	79,217.95	83,898.30
10,000	101,978.48	298.11	97,785.76	102,857.15
11,000	123,681.41	346.52	118,308.15	128591.42
12,000	145,599.23	370.77	140,783.36	145,012.94
13,000	177,175.85	418.31	165,212.41	190,066.55
14,000	205,552.94	446.69	191,594.73	217,255.65
15,000	240,397.96	514.00	219,929.88	266694.70
16,000	250,333.97	460.05	250,215.48	250,216.07
17,000	314,529.90	636.11	282,461.79	353008.47
18,000	346,788.02	731.41	316,657.65	376,084.91
19,000	392,960.08	894.28	352,806.76	450,290.47
20,000	417,950.60	864.19	390,906.05	414351.43

Nvidia GeForce GTX 690

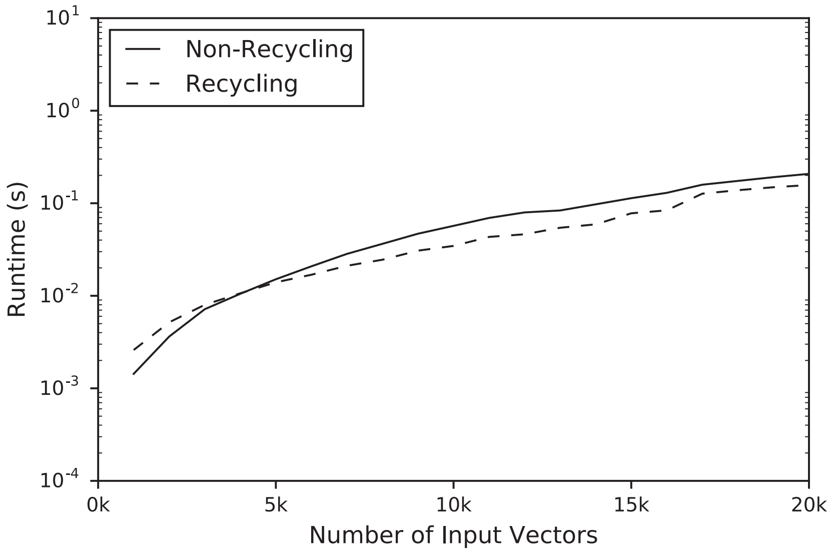


Figure E.29: Runtimes. The OpenCL kernels that conduct the computations of the pairwise input vector similarities as well as the detection of vertical lines are executed on the Nvidia GeForce GTX 690 compute device. The runtimes are captured in tabular fashion in Tab. E.31.

Table E.31: Runtimes. The tabular runtime results of Fig. E.29. $t_{Non-Recycling}$ and $t_{Recycling}$ refer to the runtimes while using the non-recycling and recycling computing approach.

Number of Input Vectors	*Runtime* (*s*)	
	$t_{Non-Recycling}$	$t_{Recycling}$
1,000	0.0014	0.0026
2,000	0.0036	0.0052
3,000	0.0072	0.0080
4,000	0.0105	0.0106
5,000	0.0151	0.0140
6,000	0.0208	0.0169
7,000	0.0284	0.0211
8,000	0.0364	0.0245
9,000	0.0469	0.0307
10,000	0.0570	0.0346
11,000	0.0694	0.0433
12,000	0.0796	0.0462
13,000	0.0836	0.0544
14,000	0.0973	0.0591
15,000	0.1134	0.0779
16,000	0.1296	0.0838
17,000	0.1586	0.1270
18,000	0.1743	0.1383
19,000	0.1913	0.1489
20,000	0.2078	0.1572

Intel Xeon E5620

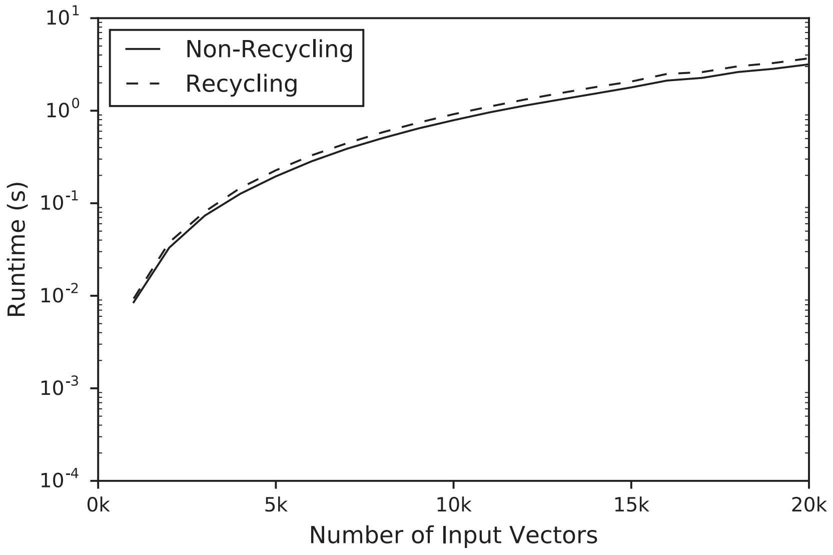


Figure E.30: Runtimes. The OpenCL kernels that conduct the computations of the pairwise input vector similarities as well as the detection of vertical lines are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.32.

Table E.32: Runtimes. The tabular runtime results of Fig. E.30. $t_{Non-Recycling}$ and $t_{Recycling}$ refer to the runtimes while using the non-recycling and recycling computing approach.

Number of Input Vectors	*Runtime* (s)	
	$t_{Non-Recycling}$	$t_{Recycling}$
1,000	0.0085	0.0093
2,000	0.0331	0.0379
3,000	0.0734	0.0810
4,000	0.1266	0.1463
5,000	0.1952	0.2269
6,000	0.2832	0.3285
7,000	0.3877	0.4450
8,000	0.5050	0.5841
9,000	0.6414	0.7403
10,000	0.7889	0.9166
11,000	0.9568	1.1056
12,000	1.1344	1.3178
13,000	1.3202	1.5474
14,000	1.5333	1.8013
15,000	1.7851	2.0656
16,000	2.1150	2.4950
17,000	2.2628	2.6137
18,000	2.6172	3.0190
19,000	2.8358	3.2726
20,000	3.1685	3.6813

E.1.8 Recurrence Matrix Materialisation

AMD Radeon RX 470

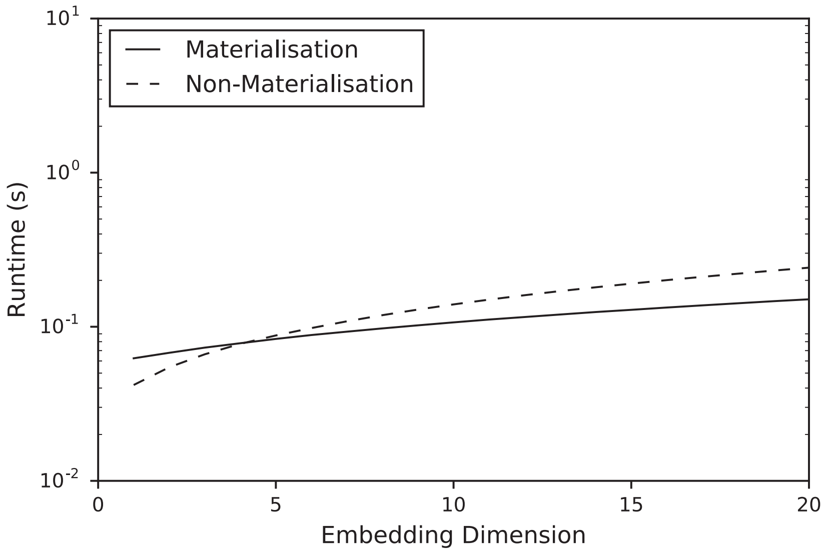


Figure E.31: Runtimes. The OpenCL kernels referring to each of the analytical RQA operators are executed on the AMD Radeon RX 470 compute device. The cumulated runtimes are captured in tabular fashion in Tab. E.33.

Table E.33: Runtimes. The tabular runtime results of Fig. E.31. $t_{Materialisation}$ and $t_{Non-Materialisation}$ refer to the runtimes while materialising and not materialising the recurrence matrix.

Embedding Dimension	*Runtime (s)*	
	$t_{Materialisation}$	$t_{Non-Materialisation}$
1	0.0624	0.0419
2	0.0678	0.0546
3	0.0732	0.0663
4	0.0782	0.0775
5	0.0834	0.0878
6	0.0884	0.0981
7	0.0929	0.1085
8	0.0976	0.1189
9	0.1022	0.1293
10	0.1068	0.1397
11	0.1114	0.1500
12	0.1157	0.1602
13	0.1201	0.1704
14	0.1246	0.1804
15	0.1288	0.1905
16	0.1333	0.2007
17	0.1376	0.2108
18	0.1420	0.2210
19	0.1465	0.2314
20	0.1507	0.2418

Nvidia GeForce GTX 690

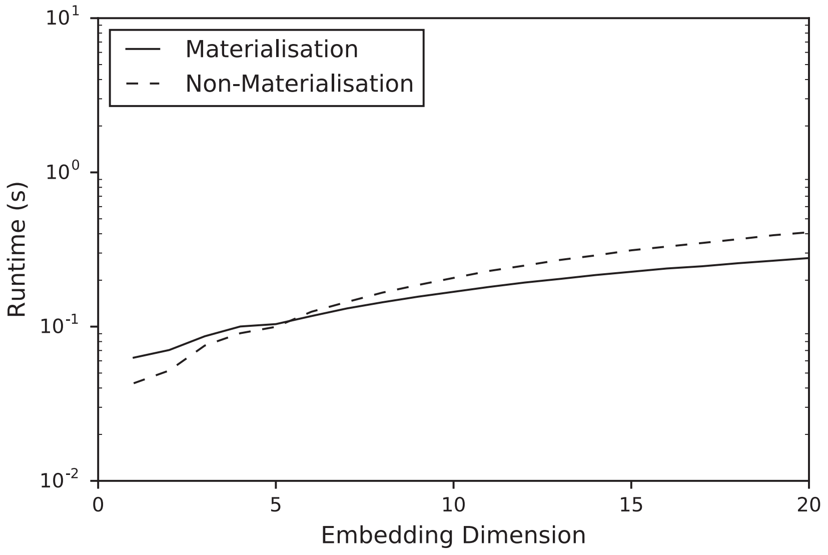


Figure E.32: Runtimes. The OpenCL kernels referring to each of the analytical RQA operators are executed on the Nvidia GeForce GTX 690 compute device. The cumulated runtimes are captured in tabular fashion in Tab. E.34.

Table E.34: Runtimes. The tabular runtime results of Fig. E.32. $t_{Materialisation}$ and $t_{Non-Materialisation}$ refer to the runtimes while materialising and not materialising the recurrence matrix.

Embedding Dimension	*Runtime (s)*	
	$t_{Materialisation}$	$t_{Non-Materialisation}$
1	0.0629	0.0429
2	0.0705	0.0519
3	0.0866	0.0753
4	0.1002	0.0906
5	0.1038	0.0997
6	0.1171	0.1248
7	0.1312	0.1445
8	0.1438	0.1662
9	0.1564	0.1862
10	0.1682	0.2069
11	0.1809	0.2296
12	0.1931	0.2488
13	0.2039	0.2707
14	0.2161	0.2896
15	0.2268	0.3125
16	0.2382	0.3303
17	0.2463	0.3487
18	0.2577	0.3687
19	0.2676	0.3915
20	0.2783	0.4091

Intel Xeon E5620

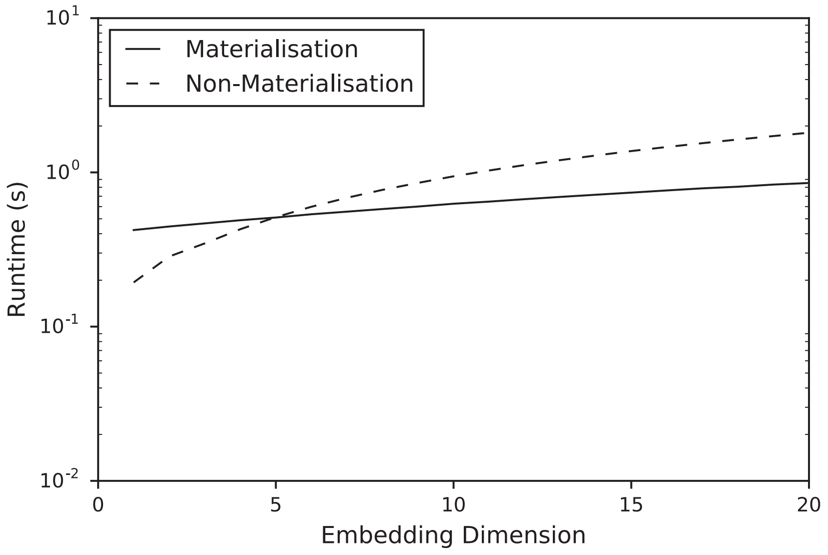


Figure E.33: Runtimes. The OpenCL kernels referring to each of the analytical RQA operators are executed on the Intel Xeon E5620 compute device. The cumulated runtimes are captured in tabular fashion in Tab. E.35.

Table E.35: Runtimes. The tabular runtime results of Fig. E.33. $t_{Materialisation}$ and $t_{Non-Materialisation}$ refer to the runtimes while materialising and not materialising the recurrence matrix.

Embedding Dimension	*Runtime (s)*	
	$t_{Materialisation}$	$t_{Non-Materialisation}$
1	0.4228	0.1935
2	0.4458	0.2850
3	0.4671	0.3464
4	0.4897	0.4281
5	0.5098	0.5124
6	0.5352	0.5986
7	0.5564	0.6845
8	0.5787	0.7703
9	0.6006	0.8550
10	0.6263	0.9431
11	0.6464	1.0288
12	0.6704	1.1158
13	0.6934	1.2006
14	0.7162	1.2864
15	0.7393	1.3750
16	0.7638	1.4611
17	0.7881	1.5465
18	0.8074	1.6342
19	0.8337	1.7201
20	0.8538	1.8082

E.2 Index Data Structures

E.2.1 Grid Directories

Uniform Distribution

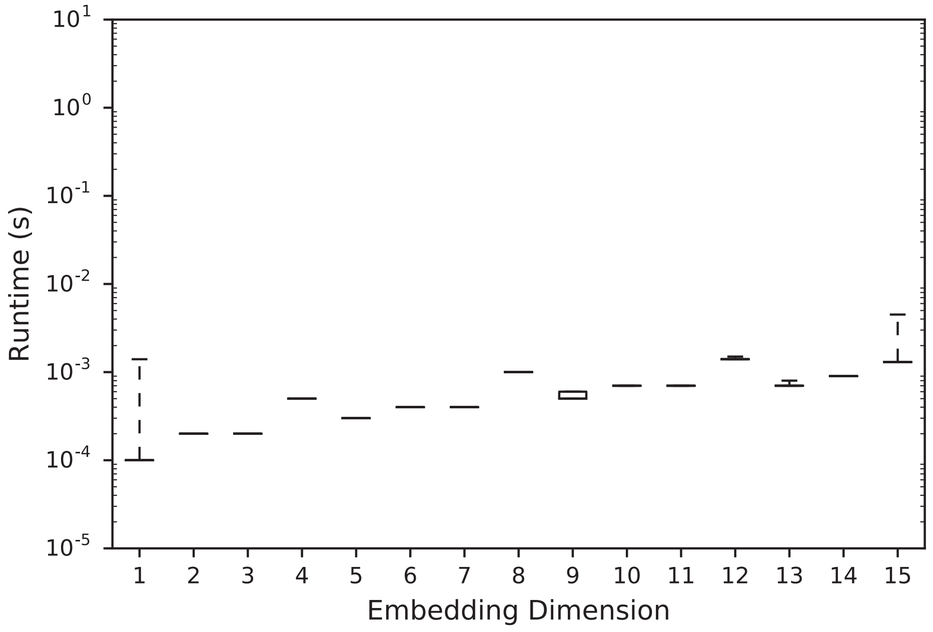


Figure E.34: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.36.

Table E.36: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.34.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0001	0.0001	0.0001	0.0001	0.0014
2	0.0002	0.0002	0.0002	0.0002	0.0002
3	0.0002	0.0002	0.0002	0.0002	0.0002
4	0.0005	0.0005	0.0005	0.0005	0.0005
5	0.0003	0.0003	0.0003	0.0003	0.0003
6	0.0004	0.0004	0.0004	0.0004	0.0004
7	0.0004	0.0004	0.0004	0.0004	0.0004
8	0.0010	0.0010	0.0010	0.0010	0.0010
9	0.0005	0.0005	0.0005	0.0006	0.0006
10	0.0007	0.0007	0.0007	0.0007	0.0007
11	0.0007	0.0007	0.0007	0.0007	0.0007
12	0.0014	0.0014	0.0014	0.0014	0.0015
13	0.0007	0.0007	0.0007	0.0007	0.0008
14	0.0009	0.0009	0.0009	0.0009	0.0009
15	0.0013	0.0013	0.0013	0.0013	0.0014

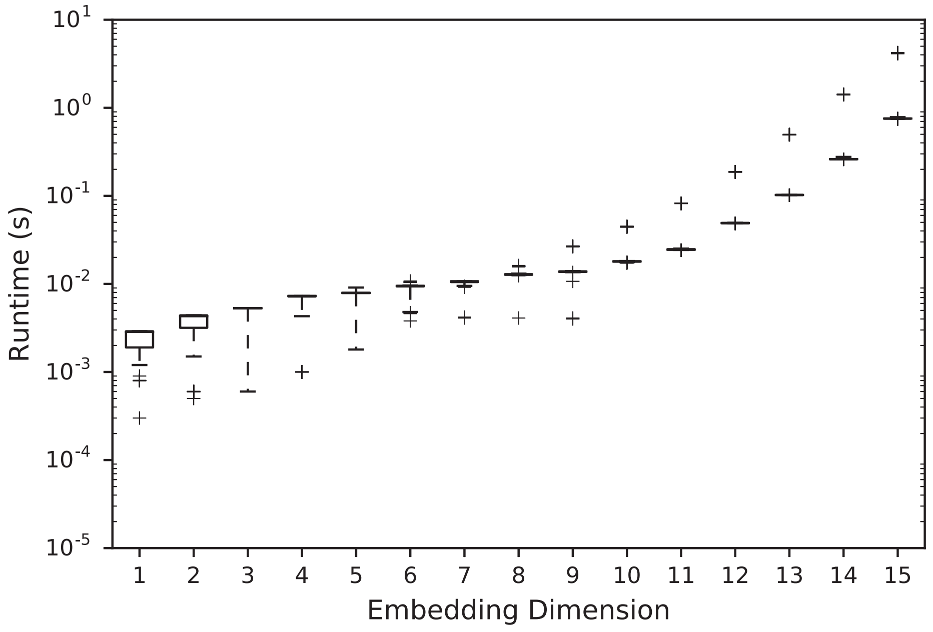


Figure E.35: Runtimes - Grid directory (atomic operations). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.37.

Table E.37: Runtimes - Grid directory (atomic operations). The tabular runtime results referring to Fig. E.35.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0003	0.0019	0.0029	0.0029	0.0029
2	0.0005	0.0032	0.0043	0.0044	0.0044
3	0.0006	0.0053	0.0053	0.0053	0.0053
4	0.0010	0.0072	0.0073	0.0073	0.0073
5	0.0018	0.0079	0.0079	0.0079	0.0091
6	0.0038	0.0094	0.0095	0.0095	0.0108
7	0.0041	0.0105	0.0107	0.0107	0.0107
8	0.0041	0.0127	0.0128	0.0129	0.0162
9	0.0040	0.0137	0.0138	0.0139	0.0270
10	0.0172	0.0179	0.0180	0.0181	0.0452
11	0.0241	0.0244	0.0246	0.0248	0.0824
12	0.0485	0.0489	0.0491	0.0493	0.1881
13	0.1017	0.1021	0.1024	0.1026	0.4989
14	0.2601	0.2608	0.2612	0.2616	1.4286
15	0.7429	0.7549	0.7554	0.7561	4.2121

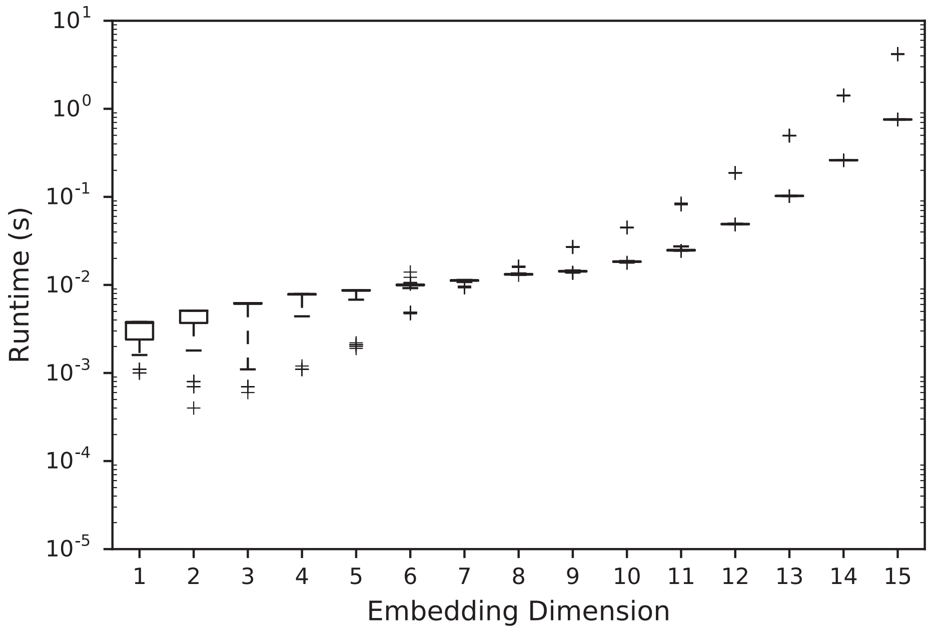


Figure E.36: Runtimes - Grid directory (sorting). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The Python code regarding the sorting of the grid cell indices is executed on the Intel Core i5-3570 compute device. The runtimes are captured in tabular fashion in Tab. E.38.

Table E.38: Runtimes - Grid directory (sorting). The tabular runtime results referring to Fig. E.36.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0010	0.0024	0.0037	0.0038	0.0038
2	0.0004	0.0037	0.0051	0.0051	0.0051
3	0.0006	0.0061	0.0062	0.0062	0.0062
4	0.0011	0.0078	0.0078	0.0079	0.0079
5	0.0019	0.0086	0.0087	0.0087	0.0087
6	0.0047	0.0099	0.0100	0.0101	0.0140
7	0.0093	0.0112	0.0112	0.0113	0.0114
8	0.0129	0.0131	0.0132	0.0133	0.0163
9	0.0136	0.0142	0.0143	0.0144	0.0272
10	0.0177	0.0183	0.0184	0.0185	0.0453
11	0.0243	0.0246	0.0248	0.0250	0.0850
12	0.0486	0.0489	0.0491	0.0493	0.1880
13	0.1018	0.1023	0.1025	0.1027	0.4994
14	0.2599	0.2608	0.2613	0.2616	1.4302
15	0.7535	0.7556	0.7563	0.7570	4.2139

Normal Distribution

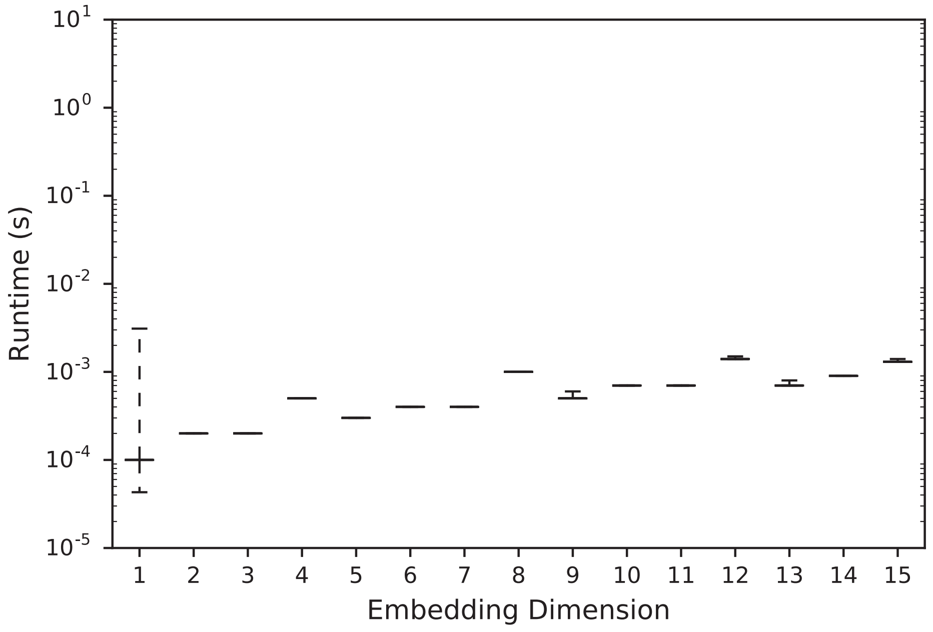


Figure E.37: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.39.

Table E.39: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.37.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0001	0.0001	0.0001	0.0001	0.0031
2	0.0002	0.0002	0.0002	0.0002	0.0002
3	0.0002	0.0002	0.0002	0.0002	0.0002
4	0.0005	0.0005	0.0005	0.0005	0.0005
5	0.0003	0.0003	0.0003	0.0003	0.0003
6	0.0004	0.0004	0.0004	0.0004	0.0004
7	0.0004	0.0004	0.0004	0.0004	0.0004
8	0.0010	0.0010	0.0010	0.0010	0.0010
9	0.0005	0.0005	0.0005	0.0005	0.0006
10	0.0007	0.0007	0.0007	0.0007	0.0007
11	0.0007	0.0007	0.0007	0.0007	0.0007
12	0.0014	0.0014	0.0014	0.0014	0.0015
13	0.0007	0.0007	0.0007	0.0007	0.0008
14	0.0009	0.0009	0.0009	0.0009	0.0009
15	0.0013	0.0013	0.0013	0.0013	0.0014

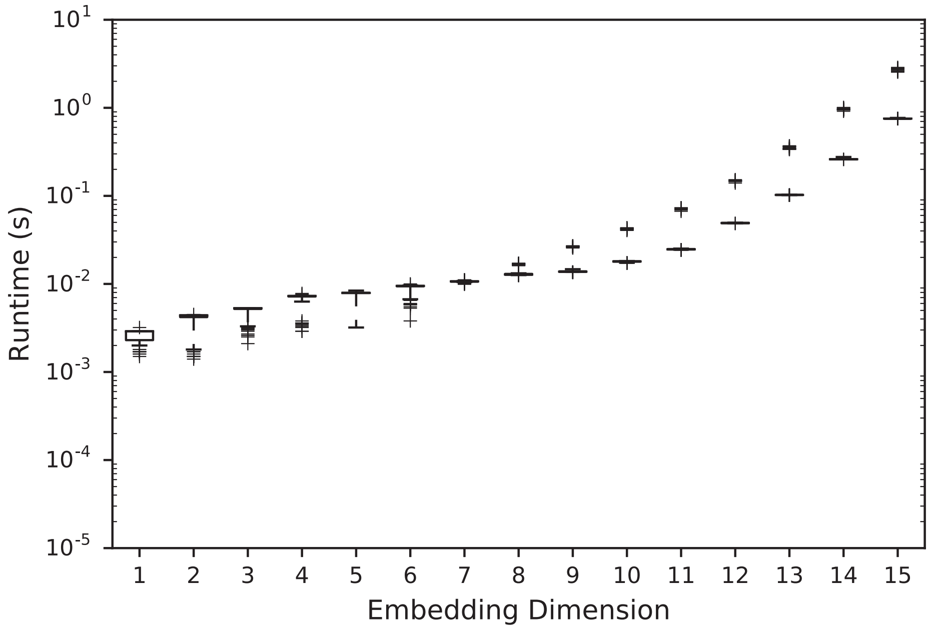


Figure E.38: Runtimes - Grid directory (atomic operations). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.40.

Table E.40: Runtimes - Grid directory (atomic operations). The tabular runtime results referring to Fig. E.38.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0015	0.0023	0.0029	0.0029	0.0032
2	0.0014	0.0042	0.0044	0.0044	0.0045
3	0.0021	0.0052	0.0053	0.0053	0.0053
4	0.0029	0.0072	0.0073	0.0073	0.0078
5	0.0032	0.0079	0.0079	0.0079	0.0084
6	0.0038	0.0094	0.0095	0.0095	0.0100
7	0.0099	0.0106	0.0107	0.0107	0.0111
8	0.0125	0.0127	0.0128	0.0130	0.0172
9	0.0135	0.0137	0.0138	0.0139	0.0271
10	0.0172	0.0179	0.0180	0.0181	0.0434
11	0.0242	0.0246	0.0247	0.0249	0.0732
12	0.0486	0.0489	0.0492	0.0494	0.1527
13	0.1018	0.1022	0.1025	0.1027	0.3705
14	0.2596	0.2601	0.2606	0.2611	1.0102
15	0.7530	0.7539	0.7542	0.7546	2.8778

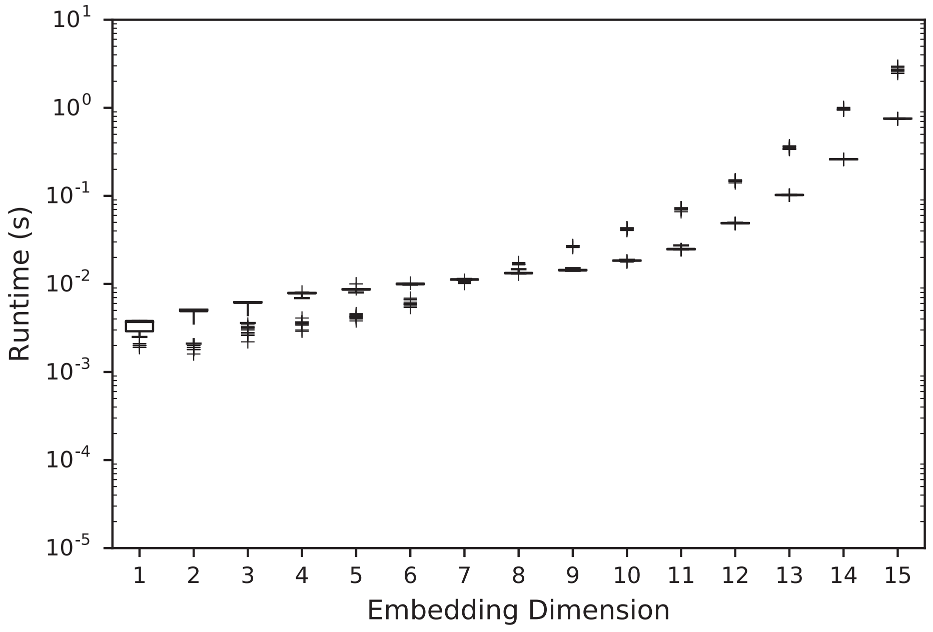


Figure E.39: Runtimes - Grid directory (sorting). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The Python code regarding the sorting of the grid cell indices is executed on the Intel Core i5-3570 compute device. The runtimes are captured in tabular fashion in Tab. E.41.

Table E.41: Runtimes - Grid directory (sorting). The tabular runtime results referring to Fig. E.39.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0019	0.0029	0.0037	0.0038	0.0038
2	0.0016	0.0049	0.0051	0.0051	0.0051
3	0.0022	0.0061	0.0062	0.0062	0.0062
4	0.0029	0.0078	0.0079	0.0079	0.0081
5	0.0038	0.0086	0.0087	0.0087	0.0100
6	0.0054	0.0099	0.0100	0.0101	0.0102
7	0.0101	0.0112	0.0112	0.0113	0.0114
8	0.0130	0.0132	0.0133	0.0133	0.0175
9	0.0141	0.0142	0.0144	0.0144	0.0273
10	0.0177	0.0184	0.0184	0.0185	0.0434
11	0.0244	0.0247	0.0248	0.0250	0.0733
12	0.0485	0.0488	0.0490	0.0492	0.1524
13	0.1018	0.1021	0.1024	0.1026	0.3704
14	0.2597	0.2600	0.2604	0.2609	1.0114
15	0.7414	0.7542	0.7547	0.7556	2.8780

Exponential Distribution

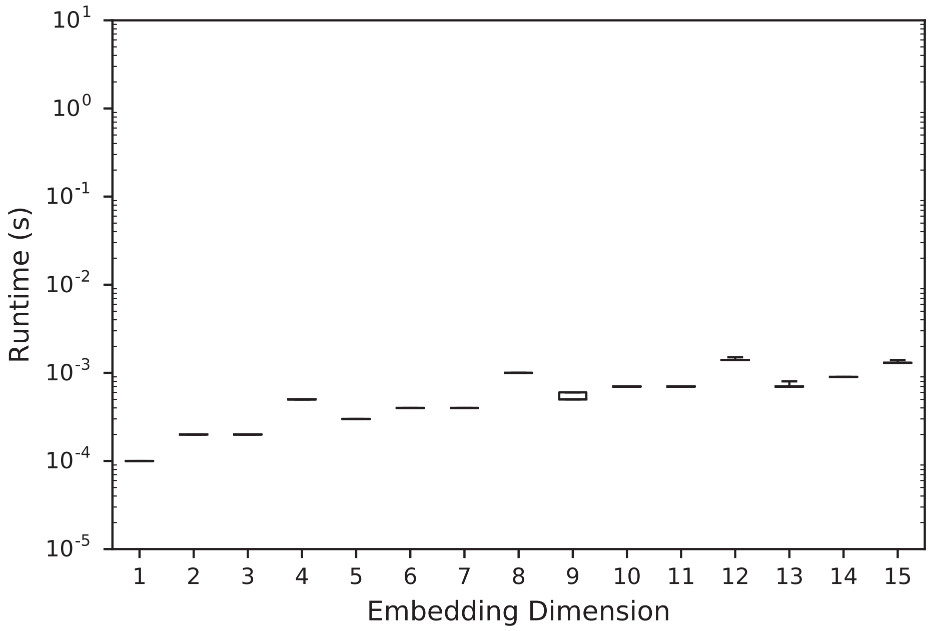


Figure E.40: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.42.

Table E.42: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.40.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0001	0.0001	0.0001	0.0001	0.0001
2	0.0002	0.0002	0.0002	0.0002	0.0002
3	0.0002	0.0002	0.0002	0.0002	0.0002
4	0.0005	0.0005	0.0005	0.0005	0.0005
5	0.0003	0.0003	0.0003	0.0003	0.0003
6	0.0004	0.0004	0.0004	0.0004	0.0004
7	0.0004	0.0004	0.0004	0.0004	0.0004
8	0.0010	0.0010	0.0010	0.0010	0.0010
9	0.0005	0.0005	0.0005	0.0006	0.0006
10	0.0007	0.0007	0.0007	0.0007	0.0007
11	0.0007	0.0007	0.0007	0.0007	0.0007
12	0.0014	0.0014	0.0014	0.0014	0.0015
13	0.0007	0.0007	0.0007	0.0007	0.0008
14	0.0009	0.0009	0.0009	0.0009	0.0009
15	0.0013	0.0013	0.0013	0.0013	0.0014

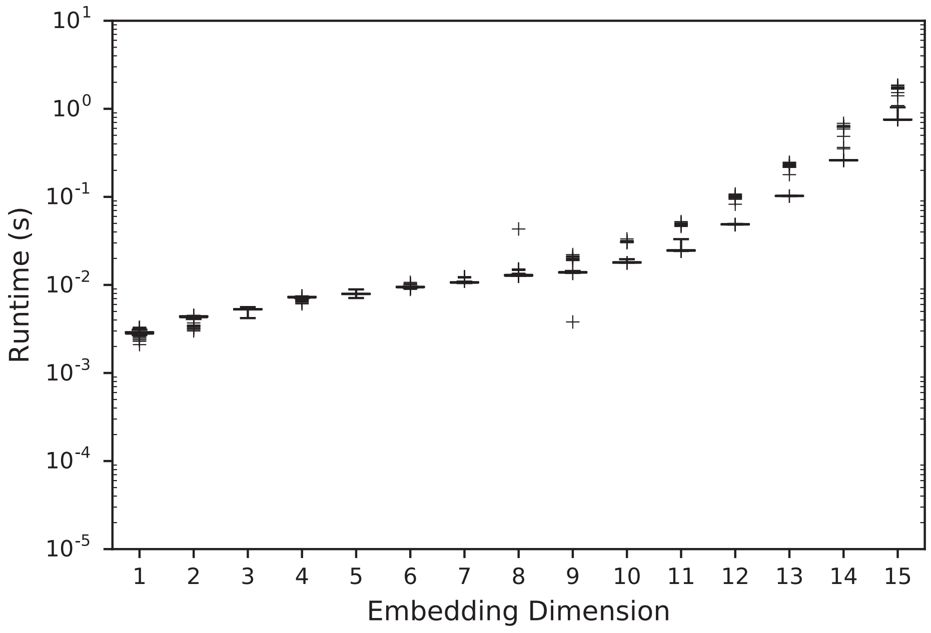


Figure E.41: Runtimes - Grid directory (atomic operations). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.43.

Table E.43: Runtimes - Grid directory (atomic operations). The tabular runtime results referring to Fig. E.41.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0021	0.0028	0.0029	0.0029	0.0033
2	0.0030	0.0043	0.0044	0.0044	0.0045
3	0.0042	0.0053	0.0053	0.0053	0.0056
4	0.0061	0.0072	0.0073	0.0073	0.0075
5	0.0071	0.0079	0.0079	0.0079	0.0089
6	0.0089	0.0094	0.0095	0.0095	0.0107
7	0.0105	0.0106	0.0107	0.0107	0.0124
8	0.0125	0.0127	0.0129	0.0130	0.0432
9	0.0038	0.0138	0.0139	0.0140	0.0221
10	0.0178	0.0179	0.0180	0.0181	0.0333
11	0.0241	0.0245	0.0246	0.0248	0.0524
12	0.0483	0.0486	0.0488	0.0490	0.1078
13	0.1020	0.1023	0.1025	0.1027	0.2475
14	0.2593	0.2602	0.2605	0.2609	0.6845
15	0.7518	0.7530	0.7533	0.7542	1.8604

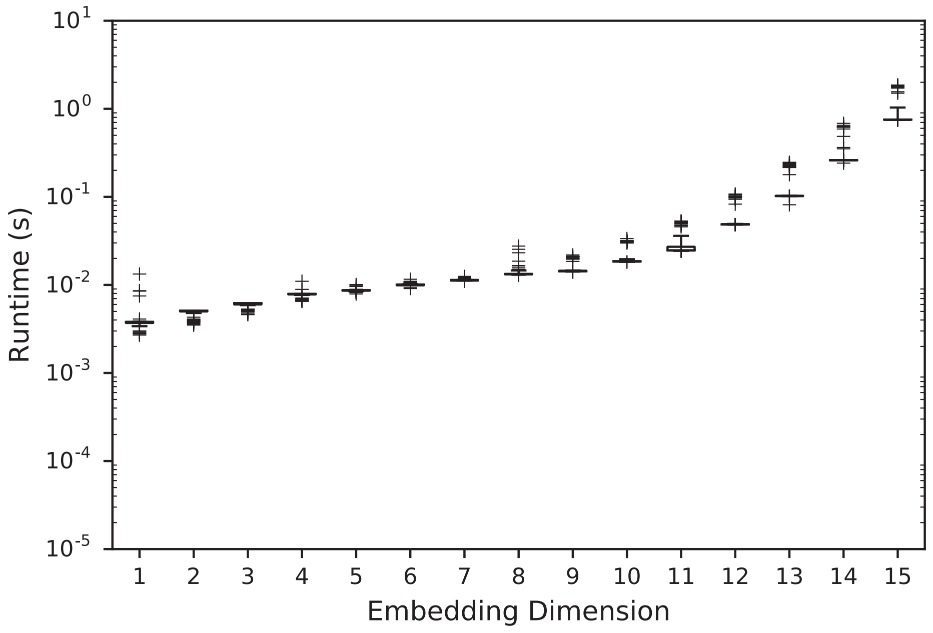


Figure E.42: Runtimes - Grid directory (sorting). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The Python code regarding the sorting of the grid cell indices is executed on the Intel Core i5-3570 compute device. The runtimes are captured in tabular fashion in Tab. E.44.

Table E.44: Runtimes - Grid directory (sorting). The tabular runtime results referring to Fig. E.42.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0027	0.0037	0.0038	0.0038	0.0133
2	0.0035	0.0050	0.0051	0.0051	0.0051
3	0.0046	0.0060	0.0062	0.0062	0.0062
4	0.0065	0.0078	0.0079	0.0079	0.0110
5	0.0079	0.0086	0.0087	0.0087	0.0101
6	0.0091	0.0099	0.0101	0.0101	0.0116
7	0.0111	0.0112	0.0113	0.0114	0.0125
8	0.0129	0.0132	0.0133	0.0133	0.0276
9	0.0140	0.0143	0.0143	0.0145	0.0219
10	0.0182	0.0184	0.0185	0.0186	0.0336
11	0.0243	0.0246	0.0247	0.0271	0.0532
12	0.0482	0.0485	0.0487	0.0489	0.1077
13	0.0814	0.1022	0.1024	0.1026	0.2470
14	0.2417	0.2604	0.2607	0.2610	0.6840
15	0.7416	0.7533	0.7539	0.7546	1.8789

Cauchy Distribution

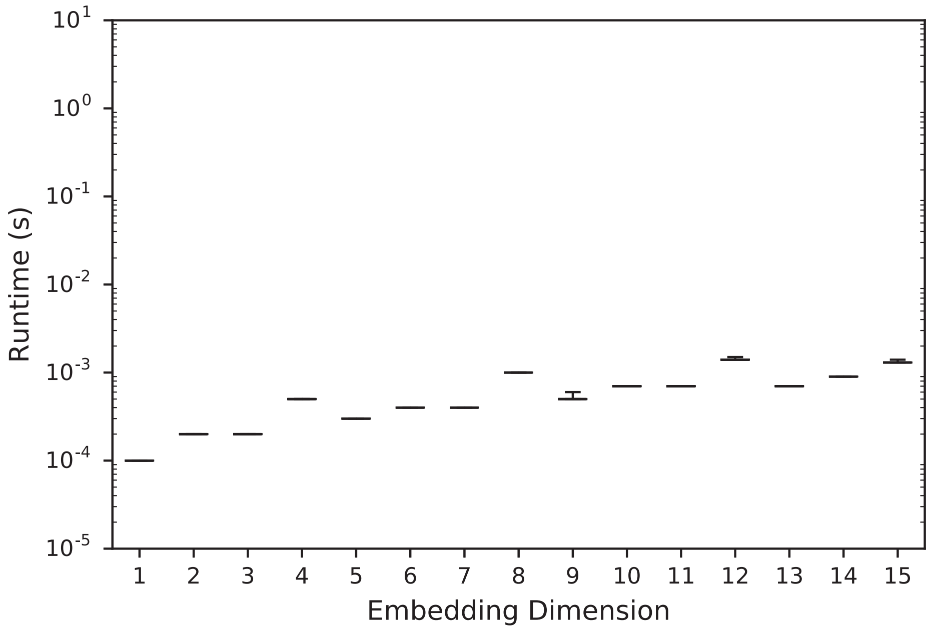


Figure E.43: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.45.

Table E.45: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.43.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0001	0.0001	0.0001	0.0001	0.0001
2	0.0002	0.0002	0.0002	0.0002	0.0002
3	0.0002	0.0002	0.0002	0.0002	0.0002
4	0.0005	0.0005	0.0005	0.0005	0.0005
5	0.0003	0.0003	0.0003	0.0003	0.0003
6	0.0004	0.0004	0.0004	0.0004	0.0004
7	0.0004	0.0004	0.0004	0.0004	0.0004
8	0.0010	0.0010	0.0010	0.0010	0.0010
9	0.0005	0.0005	0.0005	0.0005	0.0006
10	0.0007	0.0007	0.0007	0.0007	0.0007
11	0.0007	0.0007	0.0007	0.0007	0.0007
12	0.0014	0.0014	0.0014	0.0014	0.0015
13	0.0007	0.0007	0.0007	0.0007	0.0007
14	0.0009	0.0009	0.0009	0.0009	0.0009
15	0.0013	0.0013	0.0013	0.0013	0.0014

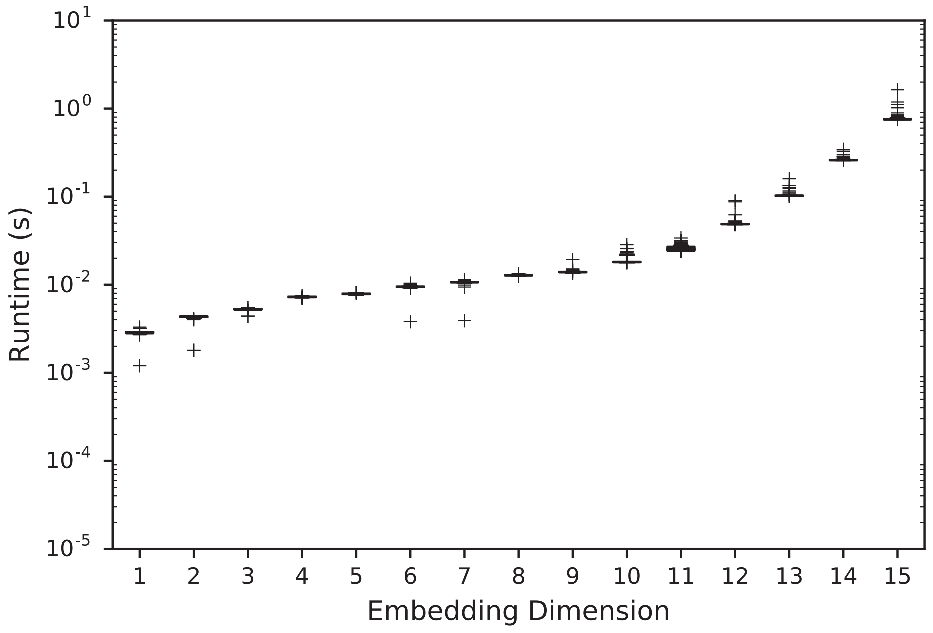


Figure E.44: Runtimes - Grid directory (atomic operations). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The runtimes are captured in tabular fashion in Tab. E.46.

Table E.46: Runtimes - Grid directory (atomic operations). The tabular runtime results referring to Fig. E.44.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0012	0.0028	0.0029	0.0029	0.0033
2	0.0018	0.0043	0.0043	0.0044	0.0044
3	0.0044	0.0052	0.0053	0.0053	0.0055
4	0.0071	0.0072	0.0073	0.0073	0.0075
5	0.0077	0.0078	0.0079	0.0079	0.0081
6	0.0038	0.0094	0.0095	0.0095	0.0104
7	0.0039	0.0106	0.0107	0.0107	0.0114
8	0.0125	0.0127	0.0128	0.0129	0.0134
9	0.0136	0.0138	0.0139	0.0140	0.0193
10	0.0178	0.0180	0.0181	0.0182	0.0284
11	0.0238	0.0243	0.0253	0.0270	0.0339
12	0.0481	0.0485	0.0487	0.0489	0.0901
13	0.1019	0.1022	0.1025	0.1028	0.1594
14	0.2585	0.2589	0.2592	0.2594	0.3445
15	0.7529	0.7534	0.7538	0.7550	1.3492

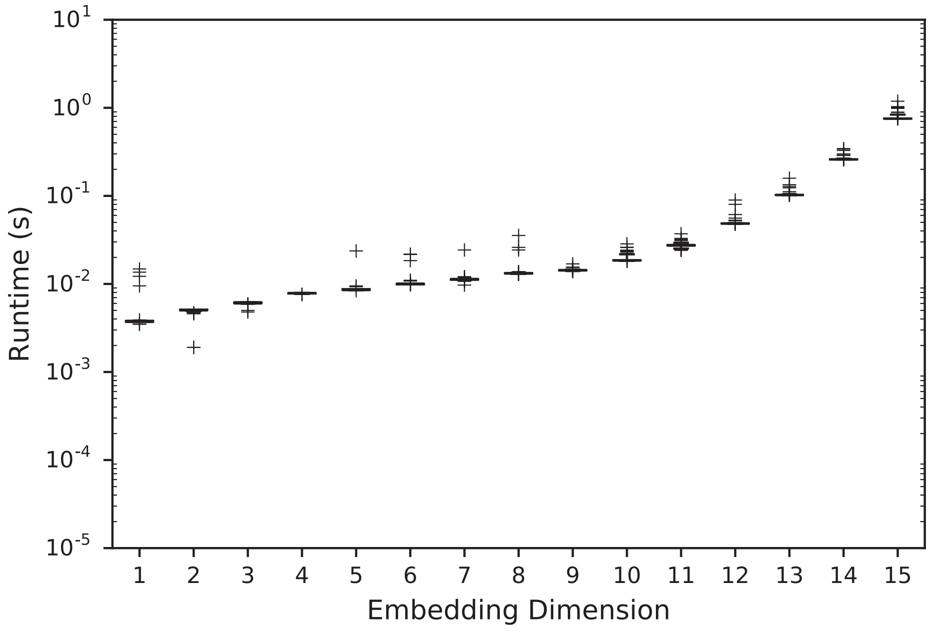


Figure E.45: Runtimes - Grid directory (sorting). The OpenCL kernels are executed on the AMD Radeon RX 470 compute device. The Python code regarding the sorting of the grid cell indices is executed on the Intel Core i5-3570 compute device. The runtimes are captured in tabular fashion in Tab. E.47.

Table E.47: Runtimes - Grid directory (sorting). The tabular runtime results referring to Fig. E.45.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0035	0.0037	0.0038	0.0038	0.0148
2	0.0019	0.0050	0.0051	0.0051	0.0051
3	0.0048	0.0060	0.0061	0.0062	0.0062
4	0.0076	0.0078	0.0078	0.0079	0.0079
5	0.0084	0.0085	0.0087	0.0087	0.0237
6	0.0098	0.0099	0.0099	0.0101	0.0218
7	0.0097	0.0112	0.0112	0.0114	0.0243
8	0.0129	0.0131	0.0132	0.0133	0.0355
9	0.0138	0.0142	0.0143	0.0144	0.0169
10	0.0182	0.0184	0.0185	0.0186	0.0285
11	0.0241	0.0272	0.0275	0.0277	0.0371
12	0.0479	0.0483	0.0486	0.0487	0.0897
13	0.1017	0.1021	0.1023	0.1026	0.1586
14	0.2584	0.2589	0.2592	0.2594	0.3444
15	0.7530	0.7544	0.7548	0.7553	1.3501

E.2.2 Multi-Dimensional Search Trees

Uniform Distribution

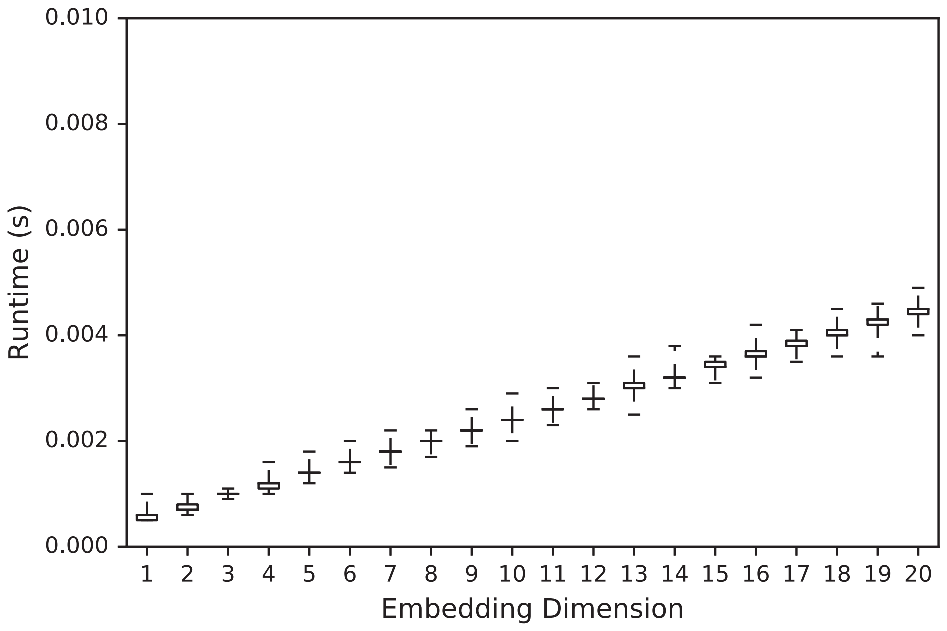


Figure E.46: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.48.

Table E.48: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.46.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0005	0.0005	0.0006	0.0006	0.0010
2	0.0006	0.0007	0.0007	0.0008	0.0010
3	0.0009	0.0010	0.0010	0.0010	0.0011
4	0.0010	0.0011	0.0012	0.0012	0.0016
5	0.0012	0.0014	0.0014	0.0014	0.0018
6	0.0014	0.0016	0.0016	0.0016	0.0020
7	0.0015	0.0018	0.0018	0.0018	0.0022
8	0.0017	0.0020	0.0020	0.0020	0.0022
9	0.0019	0.0022	0.0022	0.0022	0.0026
10	0.0020	0.0024	0.0024	0.0024	0.0029
11	0.0023	0.0026	0.0026	0.0026	0.0030
12	0.0026	0.0028	0.0028	0.0028	0.0031
13	0.0025	0.0030	0.0030	0.0031	0.0036
14	0.0030	0.0032	0.0032	0.0032	0.0038
15	0.0031	0.0034	0.0034	0.0035	0.0036
16	0.0032	0.0036	0.0036	0.0037	0.0042
17	0.0035	0.0038	0.0038	0.0039	0.0041
18	0.0036	0.0040	0.0040	0.0041	0.0045
19	0.0036	0.0042	0.0043	0.0043	0.0046
20	0.0040	0.0044	0.0045	0.0045	0.0049

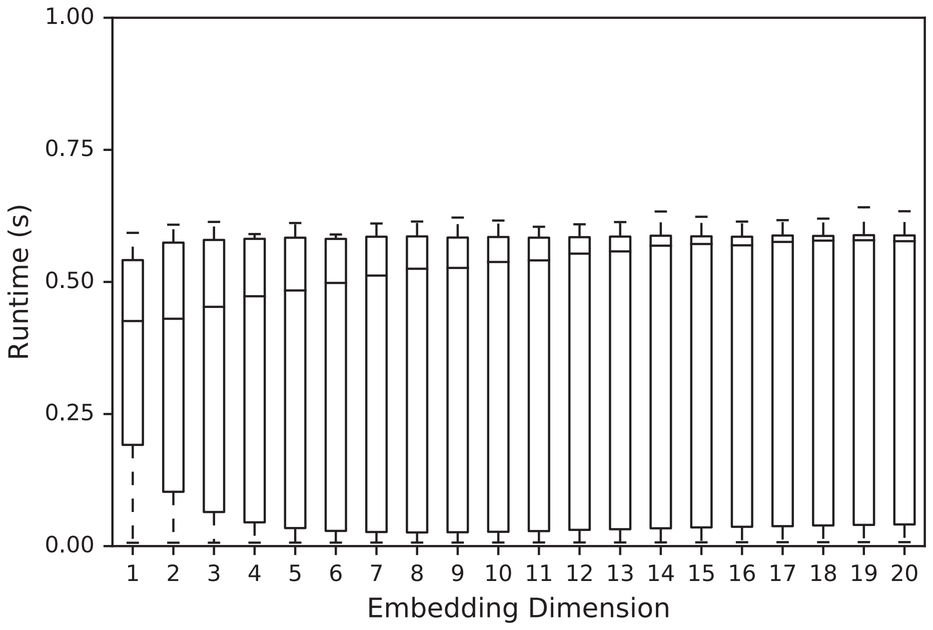


Figure E.47: Runtimes - `cKDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.49.

Table E.49: Runtimes - `cKDTree`. The tabular runtime results referring to Fig. E.47.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0063	0.1916	0.4260	0.5413	0.5929
2	0.0064	0.1028	0.4303	0.5744	0.6083
3	0.0064	0.0645	0.4529	0.5795	0.6136
4	0.0065	0.0450	0.4729	0.5816	0.5906
5	0.0066	0.0341	0.4839	0.5837	0.6115
6	0.0066	0.0288	0.4981	0.5815	0.5898
7	0.0067	0.0267	0.5121	0.5856	0.6106
8	0.0067	0.0259	0.5250	0.5862	0.6143
9	0.0067	0.0263	0.5265	0.5840	0.6218
10	0.0068	0.0271	0.5378	0.5850	0.6162
11	0.0069	0.0284	0.5408	0.5838	0.6044
12	0.0069	0.0308	0.5536	0.5847	0.6091
13	0.0069	0.0320	0.5578	0.5858	0.6133
14	0.0070	0.0337	0.5686	0.5875	0.6333
15	0.0071	0.0354	0.5718	0.5863	0.6233
16	0.0073	0.0365	0.5693	0.5856	0.6142
17	0.0074	0.0377	0.5758	0.5879	0.6169
18	0.0074	0.0391	0.5782	0.5871	0.6198
19	0.0075	0.0401	0.5790	0.5885	0.6413
20	0.0075	0.0410	0.5771	0.5880	0.6338

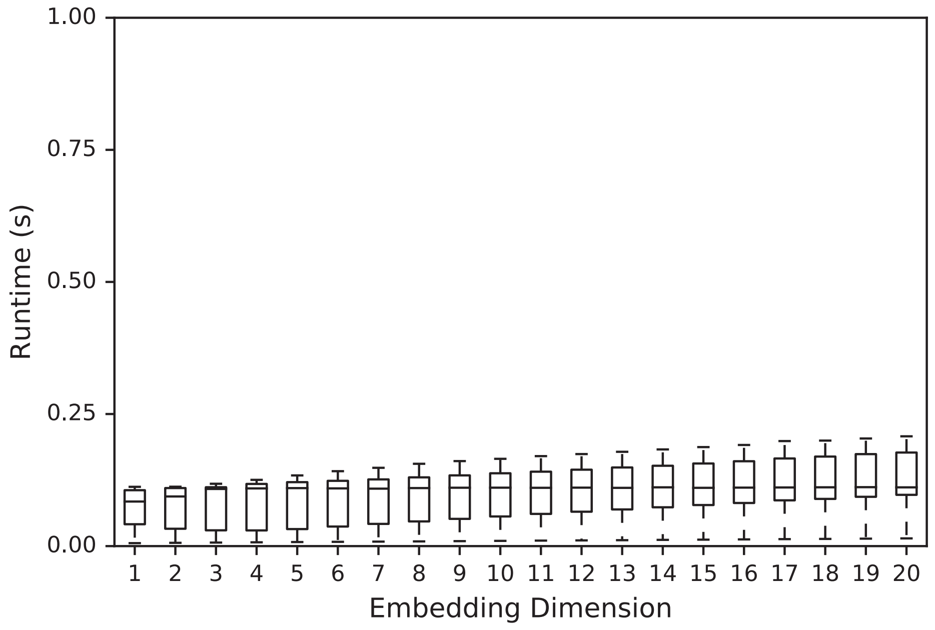


Figure E.48: Runtimes - `KDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.50.

Table E.50: Runtimes - `KDTree`. The tabular runtime results referring to Fig. E.48.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0056	0.0414	0.0844	0.1058	0.1122
2	0.0062	0.0329	0.0940	0.1099	0.1124
3	0.0067	0.0299	0.1082	0.1114	0.1180
4	0.0072	0.0297	0.1092	0.1177	0.1254
5	0.0077	0.0322	0.1098	0.1211	0.1338
6	0.0081	0.0370	0.1093	0.1236	0.1418
7	0.0085	0.0420	0.1088	0.1263	0.1482
8	0.0089	0.0467	0.1099	0.1301	0.1558
9	0.0094	0.0516	0.1105	0.1339	0.1609
10	0.0099	0.0561	0.1106	0.1379	0.1652
11	0.0104	0.0609	0.1104	0.1409	0.1703
12	0.0108	0.0651	0.1107	0.1447	0.1742
13	0.0112	0.0694	0.1103	0.1489	0.1785
14	0.0118	0.0735	0.1113	0.1521	0.1830
15	0.0122	0.0777	0.1103	0.1564	0.1874
16	0.0127	0.0816	0.1107	0.1606	0.1914
17	0.0132	0.0866	0.1110	0.1659	0.1988
18	0.0136	0.0893	0.1113	0.1695	0.1997
19	0.0141	0.0933	0.1116	0.1741	0.2037
20	0.0146	0.0970	0.1112	0.1771	0.2078

Normal Distribution

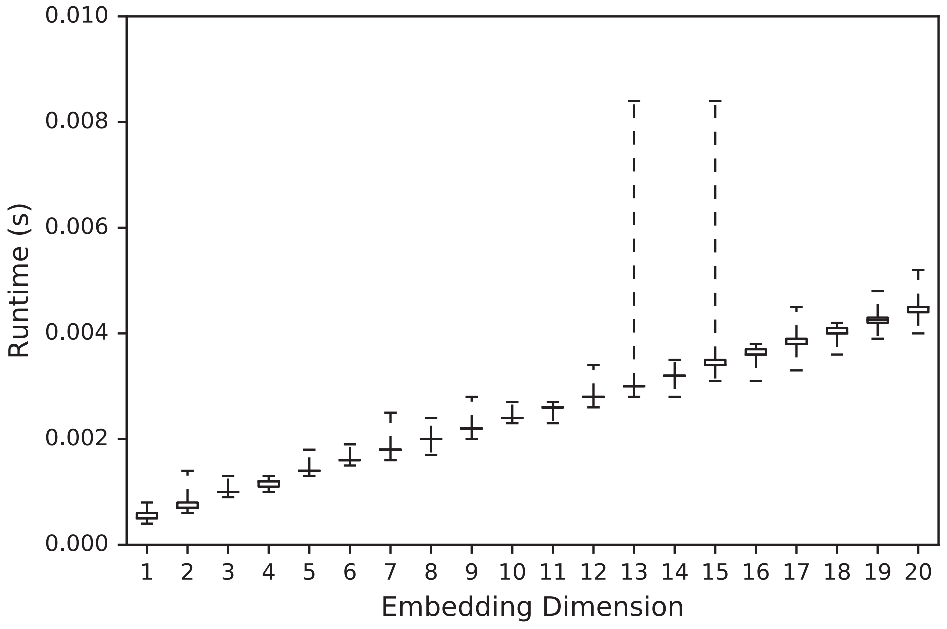


Figure E.49: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.51.

Table E.51: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.49.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0004	0.0005	0.0005	0.0006	0.0008
2	0.0006	0.0007	0.0007	0.0008	0.0014
3	0.0009	0.0010	0.0010	0.0010	0.0013
4	0.0010	0.0011	0.0012	0.0012	0.0013
5	0.0013	0.0014	0.0014	0.0014	0.0018
6	0.0015	0.0016	0.0016	0.0016	0.0019
7	0.0016	0.0018	0.0018	0.0018	0.0025
8	0.0017	0.0020	0.0020	0.0020	0.0024
9	0.0020	0.0022	0.0022	0.0022	0.0028
10	0.0023	0.0024	0.0024	0.0024	0.0027
11	0.0023	0.0026	0.0026	0.0026	0.0027
12	0.0026	0.0028	0.0028	0.0028	0.0034
13	0.0028	0.0030	0.0030	0.0030	0.0084
14	0.0028	0.0032	0.0032	0.0032	0.0035
15	0.0031	0.0034	0.0034	0.0035	0.0084
16	0.0031	0.0036	0.0036	0.0037	0.0038
17	0.0033	0.0038	0.0038	0.0039	0.0045
18	0.0036	0.0040	0.0040	0.0041	0.0042
19	0.0039	0.0042	0.0043	0.0043	0.0048
20	0.0040	0.0044	0.0045	0.0045	0.0052

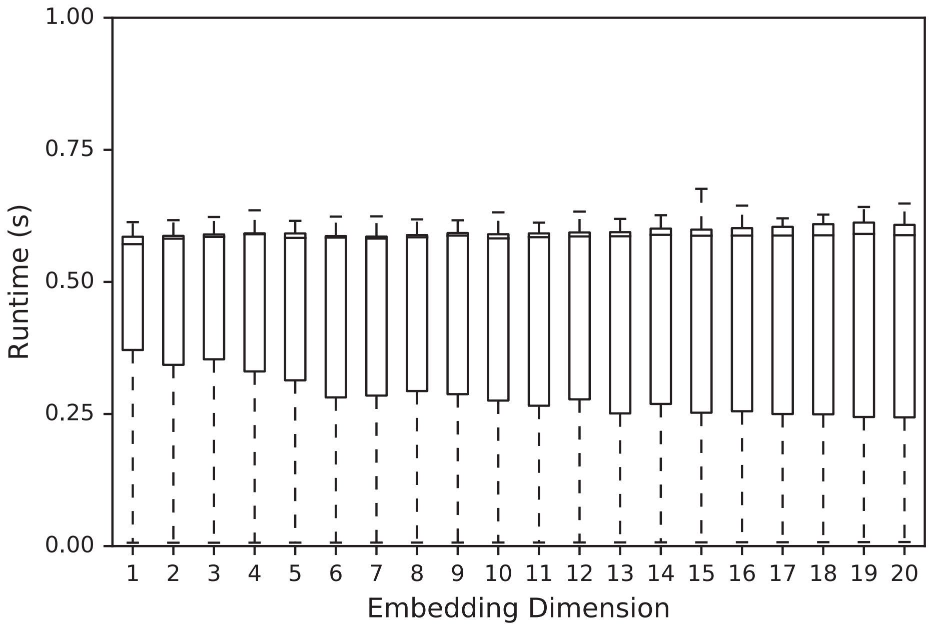


Figure E.50: Runtimes - `cKDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.52.

Table E.52: Runtimes - `cKDTree`. The tabular runtime results referring to Fig. E.50.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0064	0.3712	0.5715	0.5855	0.6132
2	0.0064	0.3430	0.5819	0.5872	0.6169
3	0.0064	0.3536	0.5853	0.5899	0.6229
4	0.0065	0.3307	0.5903	0.5920	0.6356
5	0.0066	0.3138	0.5834	0.5918	0.6156
6	0.0066	0.2815	0.5839	0.5868	0.6236
7	0.0067	0.2849	0.5821	0.5859	0.6241
8	0.0067	0.2935	0.5845	0.5886	0.6184
9	0.0067	0.2876	0.5877	0.5926	0.6167
10	0.0068	0.2755	0.5824	0.5903	0.6317
11	0.0068	0.2657	0.5847	0.5919	0.6122
12	0.0069	0.2778	0.5861	0.5937	0.6331
13	0.0070	0.2512	0.5864	0.5942	0.6193
14	0.0071	0.2690	0.5892	0.6009	0.6263
15	0.0071	0.2525	0.5875	0.5990	0.6762
16	0.0073	0.2553	0.5877	0.6018	0.6444
17	0.0073	0.2500	0.5878	0.6043	0.6204
18	0.0074	0.2495	0.5883	0.6093	0.6274
19	0.0075	0.2443	0.5909	0.6123	0.6419
20	0.0077	0.2437	0.5885	0.6080	0.6484

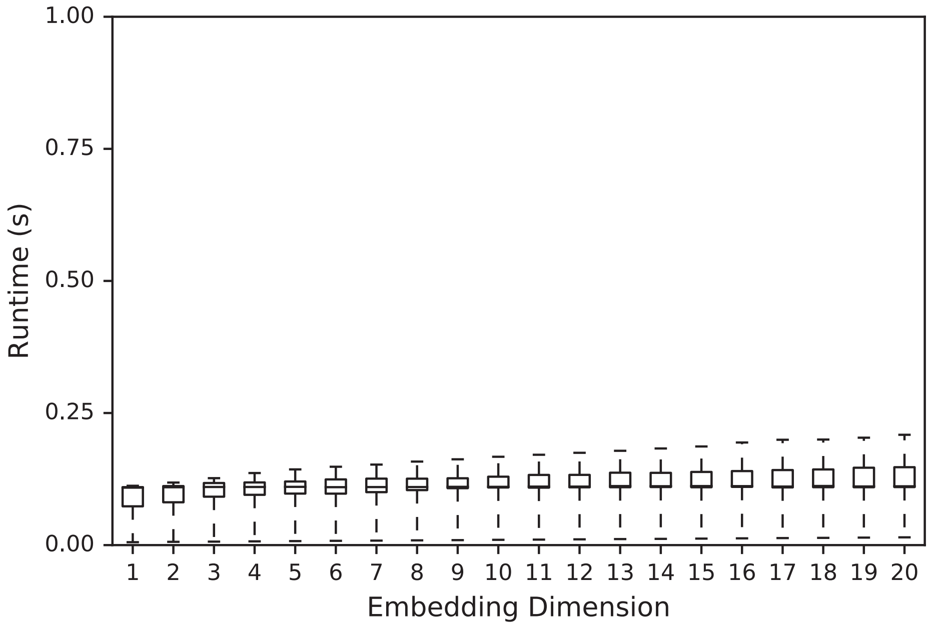


Figure E.51: Runtimes - `KDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.53.

Table E.53: Runtimes - `KDTree`. The tabular runtime results referring to Fig. E.51.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0055	0.0734	0.1088	0.1097	0.1123
2	0.0062	0.0811	0.1095	0.1117	0.1184
3	0.0066	0.0916	0.1100	0.1175	0.1267
4	0.0071	0.0952	0.1100	0.1186	0.1364
5	0.0076	0.0975	0.1103	0.1204	0.1433
6	0.0081	0.0974	0.1097	0.1244	0.1483
7	0.0085	0.1002	0.1099	0.1259	0.1524
8	0.0090	0.1041	0.1099	0.1258	0.1581
9	0.0094	0.1077	0.1103	0.1265	0.1624
10	0.0100	0.1091	0.1103	0.1295	0.1672
11	0.0104	0.1088	0.1108	0.1328	0.1710
12	0.0109	0.1094	0.1108	0.1329	0.1747
13	0.0114	0.1096	0.1118	0.1371	0.1785
14	0.0118	0.1099	0.1115	0.1368	0.1829
15	0.0123	0.1095	0.1118	0.1385	0.1867
16	0.0128	0.1102	0.1116	0.1401	0.1942
17	0.0133	0.1092	0.1109	0.1421	0.1994
18	0.0137	0.1096	0.1119	0.1432	0.1998
19	0.0142	0.1095	0.1110	0.1465	0.2033
20	0.0147	0.1098	0.1113	0.1475	0.2088

Exponential Distribution

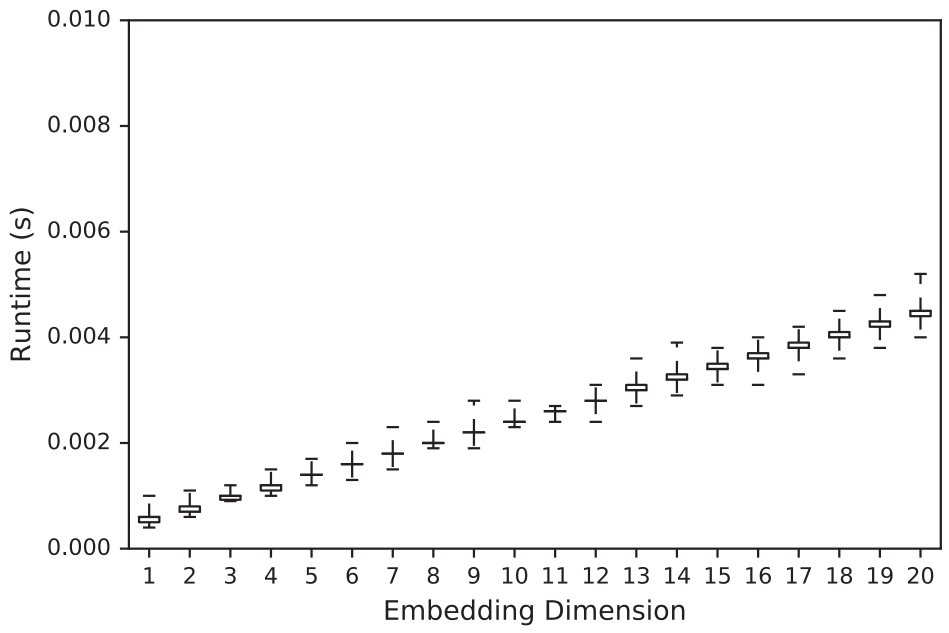


Figure E.52: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.54.

Table E.54: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.52.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0004	0.0005	0.0006	0.0006	0.0010
2	0.0006	0.0007	0.0007	0.0008	0.0011
3	0.0009	0.0009	0.0010	0.0010	0.0012
4	0.0010	0.0011	0.0012	0.0012	0.0015
5	0.0012	0.0014	0.0014	0.0014	0.0017
6	0.0013	0.0016	0.0016	0.0016	0.0020
7	0.0015	0.0018	0.0018	0.0018	0.0023
8	0.0019	0.0020	0.0020	0.0020	0.0024
9	0.0019	0.0022	0.0022	0.0022	0.0028
10	0.0023	0.0024	0.0024	0.0024	0.0028
11	0.0024	0.0026	0.0026	0.0026	0.0027
12	0.0024	0.0028	0.0028	0.0028	0.0031
13	0.0027	0.0030	0.0030	0.0031	0.0036
14	0.0029	0.0032	0.0032	0.0033	0.0039
15	0.0031	0.0034	0.0034	0.0035	0.0038
16	0.0031	0.0036	0.0036	0.0037	0.0040
17	0.0033	0.0038	0.0038	0.0039	0.0042
18	0.0036	0.0040	0.0040	0.0041	0.0045
19	0.0038	0.0042	0.0043	0.0043	0.0048
20	0.0040	0.0044	0.0045	0.0045	0.0052

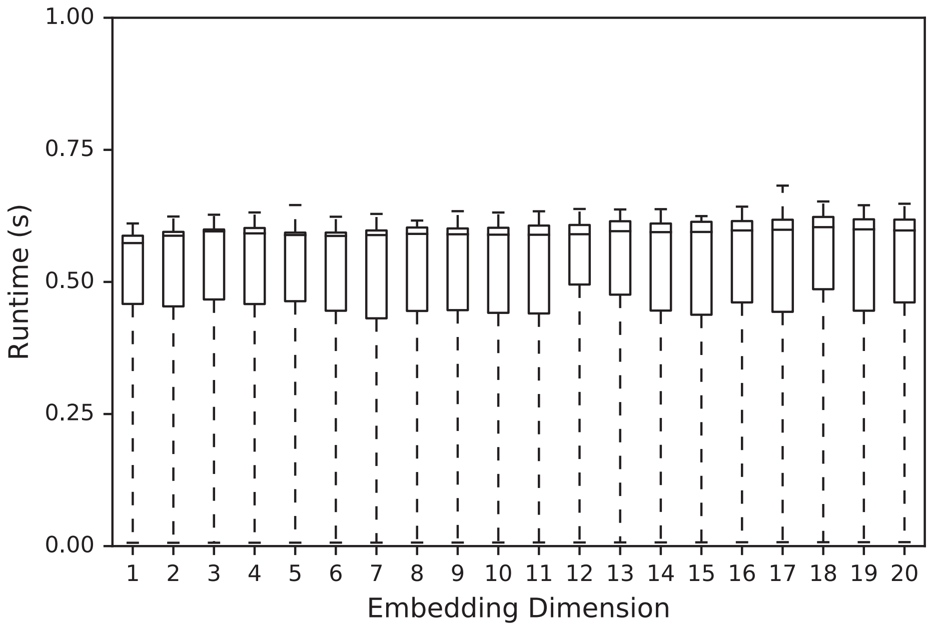


Figure E.53: Runtimes - `cKDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.55.

Table E.55: Runtimes - `cKDTree`. The tabular runtime results referring to Fig. E.53.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0063	0.4584	0.5736	0.5875	0.6107
2	0.0063	0.4537	0.5875	0.5948	0.6238
3	0.0064	0.4668	0.5957	0.5993	0.6272
4	0.0064	0.4582	0.5919	0.6021	0.6315
5	0.0065	0.4635	0.5888	0.5934	0.6455
6	0.0066	0.4455	0.5871	0.5935	0.6234
7	0.0066	0.4311	0.5887	0.5974	0.6287
8	0.0067	0.4451	0.5910	0.6030	0.6162
9	0.0067	0.4466	0.5902	0.6012	0.6339
10	0.0068	0.4415	0.5895	0.6025	0.6314
11	0.0069	0.4403	0.5893	0.6067	0.6337
12	0.0069	0.4951	0.5903	0.6078	0.6381
13	0.0070	0.4759	0.5961	0.6149	0.6372
14	0.0070	0.4458	0.5943	0.6106	0.6377
15	0.0071	0.4379	0.5946	0.6138	0.6246
16	0.0073	0.4612	0.5975	0.6152	0.6425
17	0.0074	0.4434	0.5988	0.6178	0.6823
18	0.0074	0.4862	0.6037	0.6230	0.6523
19	0.0074	0.4455	0.5995	0.6185	0.6452
20	0.0075	0.4612	0.5975	0.6180	0.6480

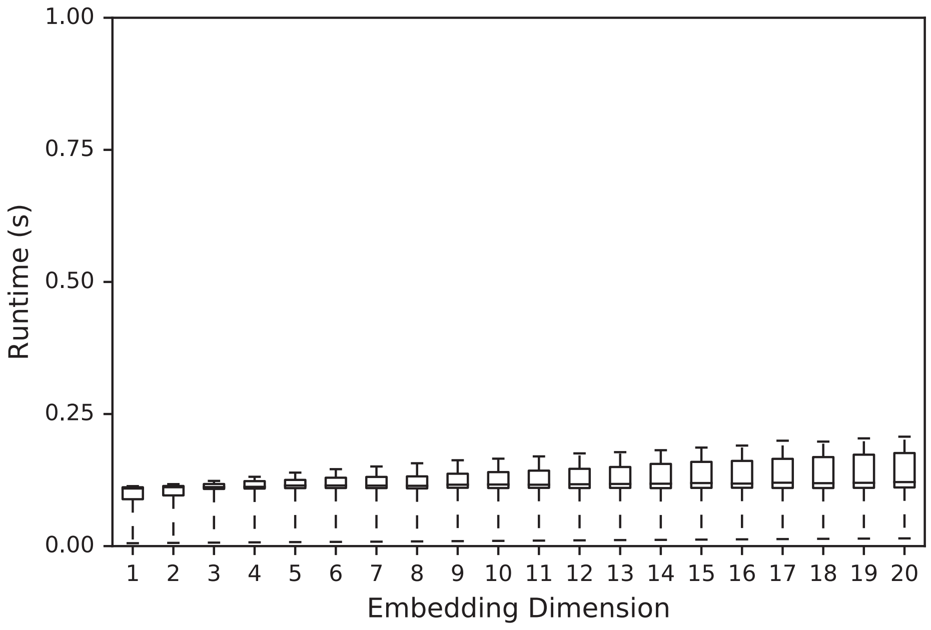


Figure E.54: Runtimes - `KDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.56.

Table E.56: Runtimes - `KDTree`. The tabular runtime results referring to Fig. E.54.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0055	0.0888	0.1091	0.1114	0.1135
2	0.0061	0.0960	0.1115	0.1138	0.1173
3	0.0066	0.1082	0.1116	0.1175	0.1234
4	0.0071	0.1089	0.1122	0.1230	0.1312
5	0.0076	0.1097	0.1147	0.1253	0.1391
6	0.0080	0.1099	0.1147	0.1295	0.1456
7	0.0084	0.1098	0.1145	0.1310	0.1507
8	0.0089	0.1092	0.1142	0.1320	0.1568
9	0.0094	0.1104	0.1163	0.1370	0.1625
10	0.0099	0.1099	0.1166	0.1401	0.1655
11	0.0104	0.1103	0.1162	0.1429	0.1698
12	0.0109	0.1099	0.1171	0.1463	0.1755
13	0.0114	0.1101	0.1177	0.1497	0.1778
14	0.0118	0.1097	0.1181	0.1556	0.1815
15	0.0123	0.1102	0.1194	0.1594	0.1865
16	0.0128	0.1104	0.1184	0.1613	0.1903
17	0.0133	0.1101	0.1200	0.1652	0.1995
18	0.0138	0.1099	0.1191	0.1685	0.1978
19	0.0142	0.1103	0.1198	0.1731	0.2039
20	0.0146	0.1110	0.1212	0.1761	0.2071

Cauchy Distribution

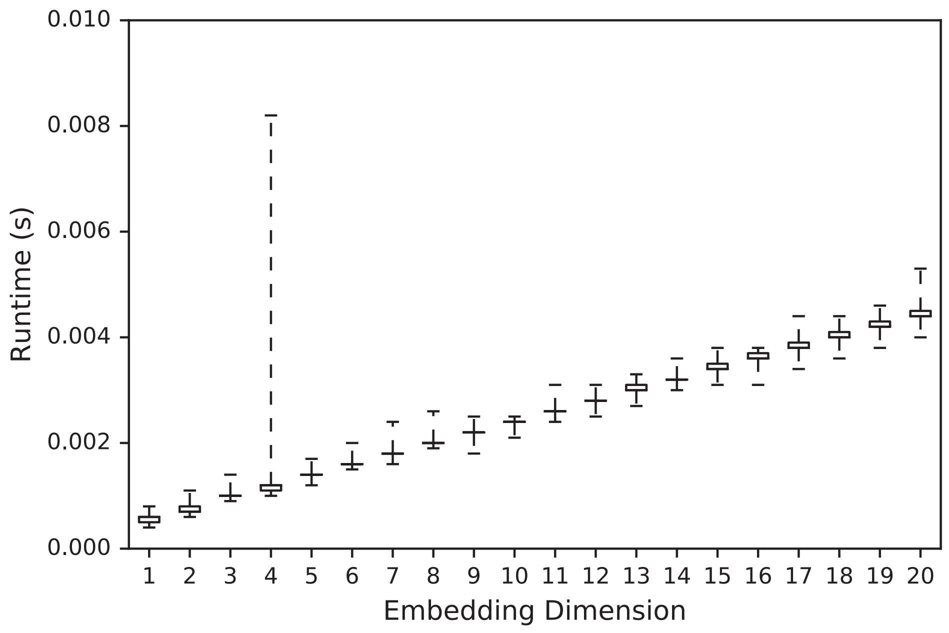


Figure E.55: Runtimes - Parallel brute-force processing. The OpenCL kernels are executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.57.

Table E.57: Runtimes - Parallel brute-force processing. The tabular runtime results referring to Fig. E.55.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0004	0.0005	0.0006	0.0006	0.0008
2	0.0006	0.0007	0.0007	0.0008	0.0011
3	0.0009	0.0010	0.0010	0.0010	0.0014
4	0.0010	0.0011	0.0012	0.0012	0.0082
5	0.0012	0.0014	0.0014	0.0014	0.0017
6	0.0015	0.0016	0.0016	0.0016	0.0020
7	0.0016	0.0018	0.0018	0.0018	0.0024
8	0.0019	0.0020	0.0020	0.0020	0.0026
9	0.0018	0.0022	0.0022	0.0022	0.0025
10	0.0021	0.0024	0.0024	0.0024	0.0025
11	0.0024	0.0026	0.0026	0.0026	0.0031
12	0.0025	0.0028	0.0028	0.0028	0.0031
13	0.0027	0.0030	0.0030	0.0031	0.0033
14	0.0030	0.0032	0.0032	0.0032	0.0036
15	0.0031	0.0034	0.0034	0.0035	0.0038
16	0.0031	0.0036	0.0036	0.0037	0.0038
17	0.0034	0.0038	0.0038	0.0039	0.0044
18	0.0036	0.0040	0.0040	0.0041	0.0044
19	0.0038	0.0042	0.0042	0.0043	0.0046
20	0.0040	0.0044	0.0044	0.0045	0.0053

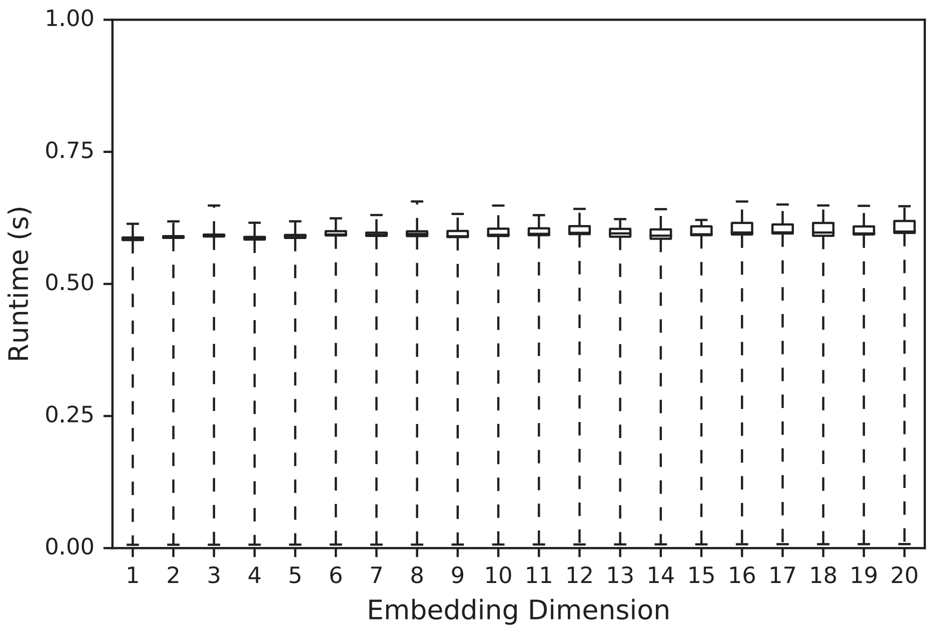


Figure E.56: Runtimes - `cKDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.58.

Table E.58: Runtimes - `cKDTree`. The tabular runtime results referring to Fig. E.56.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0063	0.5831	0.5866	0.5879	0.6138
2	0.0064	0.5872	0.5884	0.5904	0.6184
3	0.0064	0.5898	0.5915	0.5934	0.6484
4	0.0065	0.5840	0.5876	0.5890	0.6158
5	0.0066	0.5871	0.5912	0.5927	0.6185
6	0.0067	0.5918	0.5935	0.5998	0.6242
7	0.0066	0.5910	0.5930	0.5971	0.6304
8	0.0066	0.5906	0.5947	0.5996	0.6561
9	0.0067	0.5887	0.5902	0.6002	0.6325
10	0.0068	0.5908	0.5931	0.6046	0.6483
11	0.0069	0.5923	0.5948	0.6053	0.6302
12	0.0069	0.5944	0.5967	0.6094	0.6420
13	0.0070	0.5894	0.5957	0.6042	0.6227
14	0.0071	0.5856	0.5914	0.6032	0.6415
15	0.0071	0.5923	0.5939	0.6090	0.6211
16	0.0073	0.5936	0.5973	0.6155	0.6560
17	0.0074	0.5955	0.5976	0.6126	0.6503
18	0.0074	0.5911	0.5973	0.6155	0.6485
19	0.0075	0.5936	0.5955	0.6088	0.6478
20	0.0076	0.5966	0.5991	0.6191	0.6471

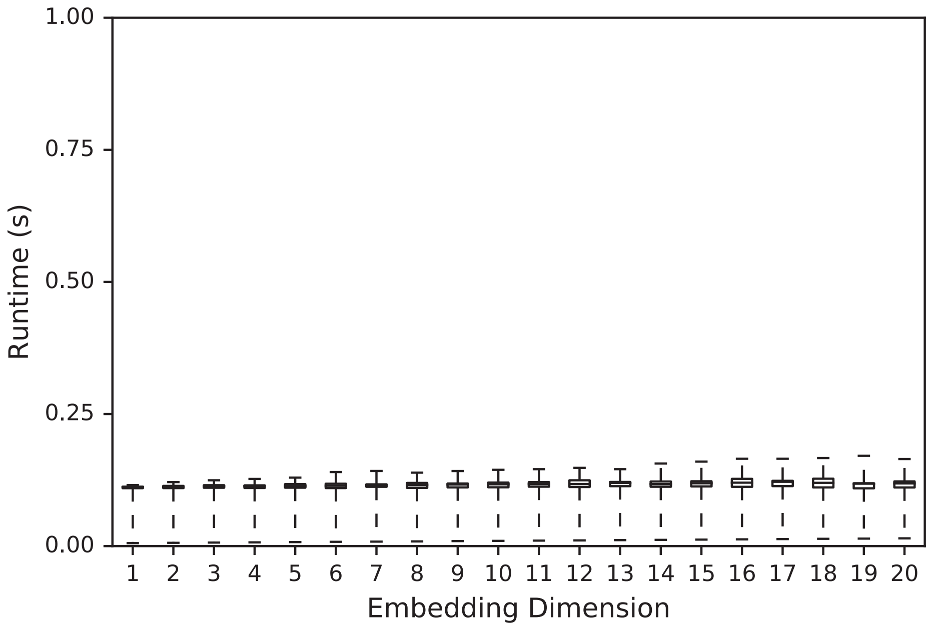


Figure E.57: Runtimes - `KDTree`. The Python code is executed on the Intel Xeon E5620 compute device. The runtimes are captured in tabular fashion in Tab. E.59.

Table E.59: Runtimes - `KDTree`. The tabular runtime results referring to Fig. E.57.

Embedding Dimension	*Min*	25%	50%	75%	*Max*
1	0.0056	0.1096	0.1118	0.1125	0.1155
2	0.0062	0.1098	0.1121	0.1139	0.1212
3	0.0067	0.1104	0.1132	0.1153	0.1246
4	0.0071	0.1100	0.1134	0.1150	0.1271
5	0.0076	0.1105	0.1135	0.1173	0.1296
6	0.0081	0.1098	0.1142	0.1181	0.1402
7	0.0085	0.1122	0.1150	0.1169	0.1422
8	0.0089	0.1102	0.1155	0.1196	0.1390
9	0.0095	0.1111	0.1166	0.1185	0.1421
10	0.0099	0.1113	0.1170	0.1203	0.1444
11	0.0104	0.1124	0.1174	0.1211	0.1456
12	0.0108	0.1118	0.1175	0.1246	0.1481
13	0.0113	0.1134	0.1195	0.1214	0.1456
14	0.0118	0.1123	0.1173	0.1223	0.1563
15	0.0123	0.1130	0.1189	0.1227	0.1599
16	0.0128	0.1122	0.1201	0.1273	0.1653
17	0.0133	0.1135	0.1217	0.1236	0.1653
18	0.0138	0.1111	0.1198	0.1276	0.1667
19	0.0142	0.1094	0.1181	0.1190	0.1709
20	0.0147	0.1110	0.1187	0.1225	0.1648

Runtime Ratios

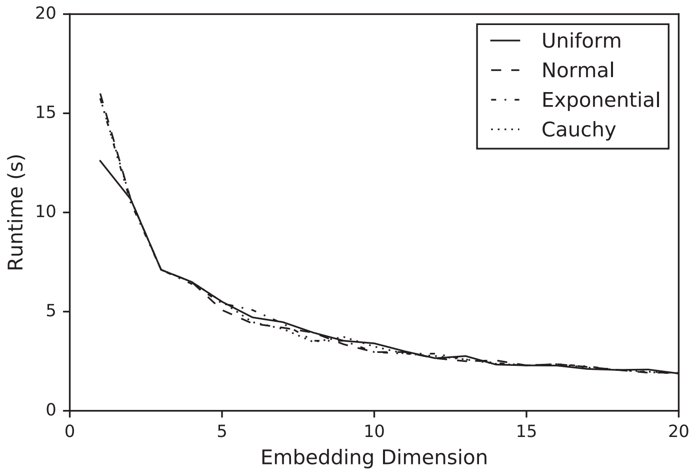


Figure E.58: Runtime ratios. The minimum runtimes of the `cKDTree` implementation and the parallel brute-force implementation are compared regarding each of the four distribution types applied.

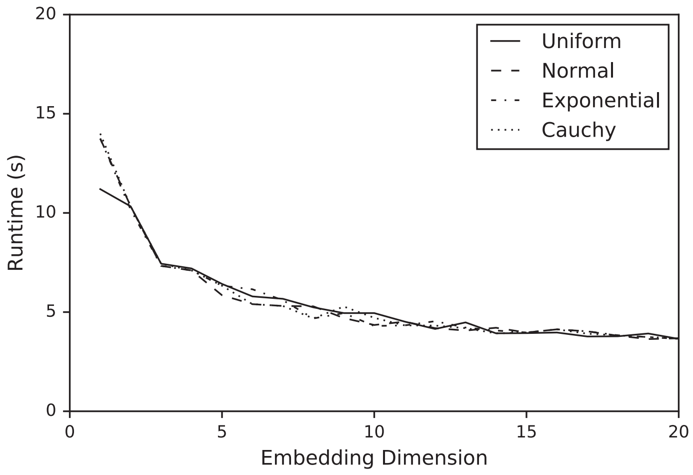


Figure E.59: Runtime ratios. The minimum runtimes of the `KDTree` implementation and the parallel brute-force implementation are compared regarding each of the four distribution types applied.

Table E.60: Runtime ratios. The tabular ratios referring to Fig. E.58.

Embedding Dimension	*Uniform*	*Normal*	*Exponential*	*Cauchy*
1	12.60	16.00	15.75	15.75
2	10.67	10.67	10.50	10.67
3	7.11	7.11	7.11	7.11
4	6.50	6.50	6.40	6.50
5	5.50	5.08	5.42	5.50
6	4.71	4.40	5.08	4.47
7	4.47	4.19	4.40	4.13
8	3.94	3.94	3.53	3.47
9	3.53	3.35	3.53	3.72
10	3.40	2.96	2.96	3.24
11	3.00	2.96	2.88	2.88
12	2.65	2.65	2.88	2.76
13	2.76	2.50	2.59	2.59
14	2.33	2.54	2.41	2.37
15	2.29	2.29	2.29	2.29
16	2.28	2.35	2.35	2.35
17	2.11	2.21	2.24	2.18
18	2.06	2.06	2.06	2.06
19	2.08	1.92	1.95	1.97
20	1.88	1.93	1.88	1.90

Table E.61: Runtime ratios. The tabular ratios referring to Fig. E.59.

Embedding Dimension	*Uniform*	*Normal*	*Exponential*	*Cauchy*
1	11.20	13.75	13.75	14.00
2	10.33	10.33	10.17	10.33
3	7.44	7.33	7.33	7.44
4	7.20	7.10	7.10	7.10
5	6.42	5.85	6.33	6.33
6	5.79	5.40	6.15	5.40
7	5.67	5.31	5.60	5.31
8	5.24	5.29	4.68	4.68
9	4.95	4.70	4.95	5.28
10	4.95	4.35	4.30	4.71
11	4.52	4.52	4.33	4.33
12	4.15	4.19	4.54	4.32
13	4.48	4.07	4.22	4.19
14	3.93	4.21	4.07	3.93
15	3.94	3.97	3.97	3.97
16	3.97	4.13	4.13	4.13
17	3.77	4.03	4.03	3.91
18	3.78	3.81	3.83	3.83
19	3.92	3.64	3.74	3.74
20	3.65	3.68	3.65	3.68

E.3 Automatic Performance Tuning for Implementation Selection

E.3.1 Selection Strategies

AMD Radeon RX 470

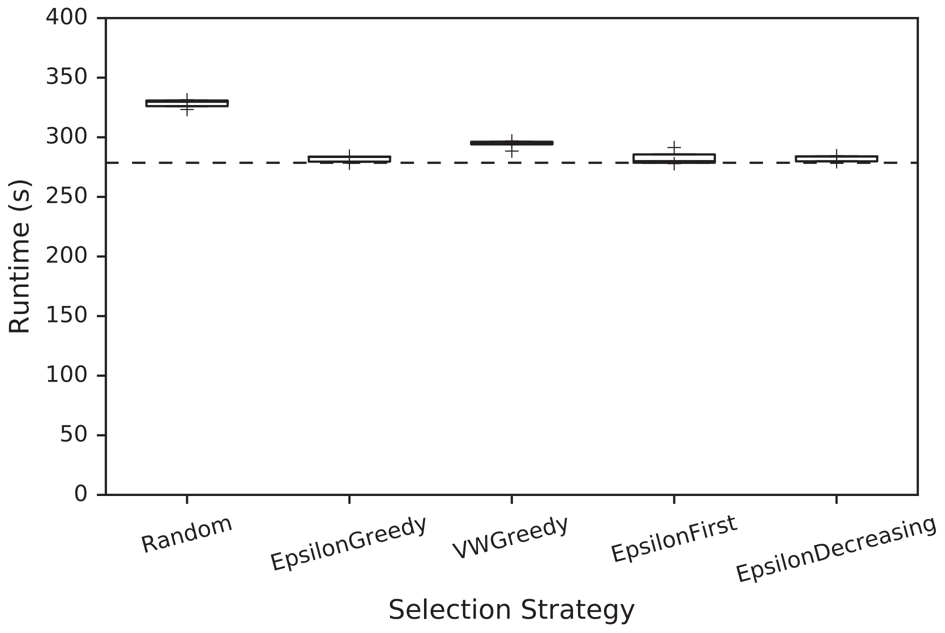


Figure E.60: Runtimes. The OpenCL code is executed on the AMD Radeon RX 470 compute device contained by computing system *(A)*. The runtimes are captured in tabular fashion in Tab. E.62. The dotted line represents the runtime when always selecting the flavour with the highest average reward for processing each sub matrix. Its properties are depicted in Tab. E.63.

Table E.62: Runtimes. The tabular runtime results referring to Fig. E.60.

Selection Strategy	*Min*	25%	50%	75%	*Max*
baseline	278.09	278.36	278.85	278.86	278.87
random	323.31	326.12	329.81	330.87	331.46
ϵ-greedy	278.23	279.55	283.57	283.75	284.13
vw-greedy	288.43	294.05	294.91	296.26	296.97
ϵ-first	277.87	278.65	279.97	285.61	291.41
ϵ-decreasing	279.49	279.82	283.86	283.96	284.53

Table E.63: Properties of best-performing flavour.

Property	*Value*
Input Data Representation	Column-wise
Recurrence Matrix Representation	Uncompressed
Similarity Value Representation	Byte
Intermediate Results Recycling	No
Recurrence Matrix Materialisation	Yes
Loop Unrolling Factor	2^1

Nvidia GeForce GTX 690

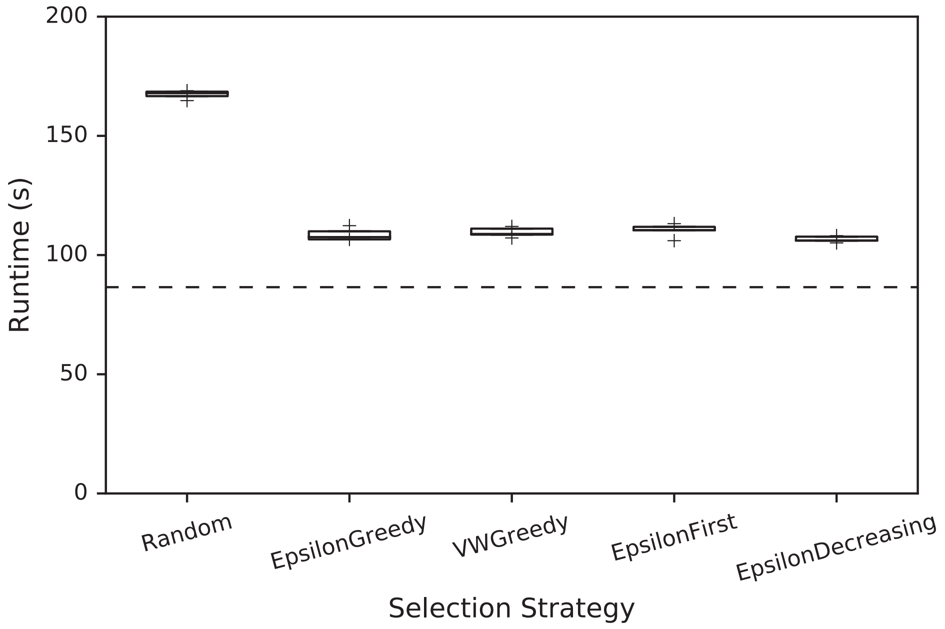


Figure E.61: Runtimes. The OpenCL code is executed on the four Nvidia GeForce GTX 690 compute devices contained by computing system *(B)*. The runtimes are captured in tabular fashion in Tab. E.64. The dotted line represents the runtime when always selecting the flavour with the highest average reward for processing each sub matrix. Its properties are depicted in Tab. E.65.

Table E.64: Runtimes. The tabular runtime results referring to Fig. E.61.

Selection Strategy	*Min*	25%	50%	75%	*Max*
baseline	86.28	86.36	86.45	86.70	86.84
random	164.79	166.66	167.9	168.53	168.94
ϵ-greedy	106.57	106.58	107.53	109.96	112.33
vw-greedy	107.17	108.57	108.85	111.09	112.01
ϵ-first	106.03	110.42	110.46	111.84	113.17
ϵ-decreasing	105.1	106.06	106.12	107.72	108.14

Table E.65: Properties of best-performing flavour.

Property	*Value*
Input Data Representation	Column-wise
Recurrence Matrix Representation	Uncompressed
Similarity Value Representation	Byte
Intermediate Results Recycling	Yes
Recurrence Matrix Materialisation	Yes
Loop Unrolling Factor	2^3

Intel Xeon E5620

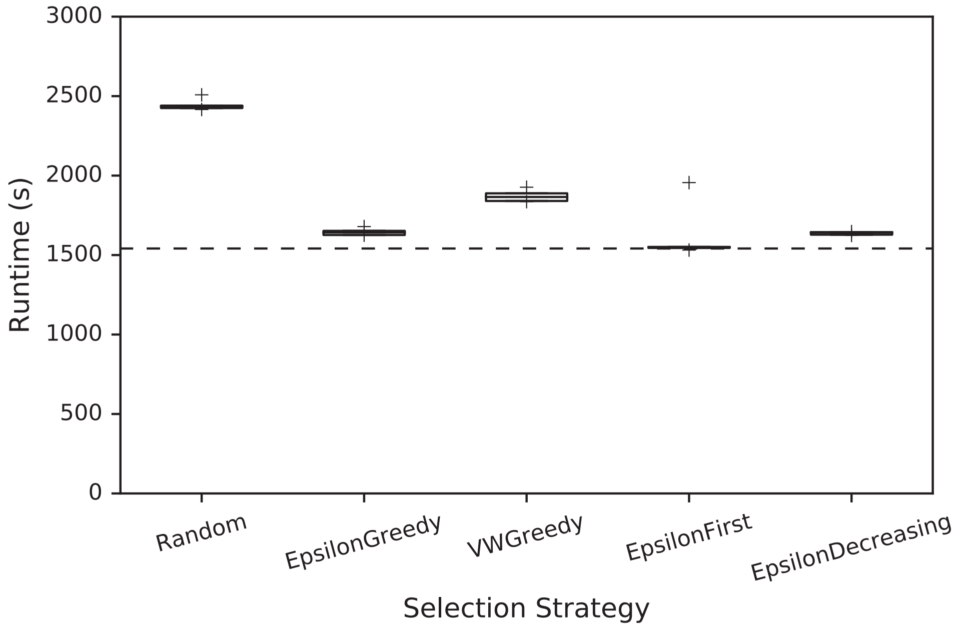


Figure E.62: Runtimes. The OpenCL code is executed on the CPU compute devices contained by computing system *(C)*, consisting of two Intel Xeon E5620 processors. The runtimes are captured in tabular fashion in Tab. E.66. The dotted line represents the runtime when always selecting the flavour with the highest average reward for processing each sub matrix. Its properties are depicted in Tab. E.67.

Table E.66: Runtimes. The tabular runtime results referring to Fig. E.62.

Selection Strategy	*Min*	25%	50%	75%	*Max*
baseline	1537.62	1540.24	1542.47	1542.72	1543.20
random	2415.78	2424.95	2435.59	2440.10	2508.24
ϵ-greedy	1624.70	1626.34	1642.39	1652.19	1679.29
vw-greedy	1835.90	1840.29	1865.43	1888.30	1926.92
ϵ-first	1531.44	1544.83	1549.47	1551.06	1955.99
ϵ-decreasing	1623.38	1628.30	1642.78	1643.85	1647.38

Table E.67: Properties of best-performing flavour.

Property	*Value*
Input Data Representation	Column-wise
Recurrence Matrix Representation	Uncompressed
Similarity Value Representation	Bit
Intermediate Results Recycling	Yes
Recurrence Matrix Materialisation	Yes
Loop Unrolling Factor	2^0

E.3.2 Efficiency

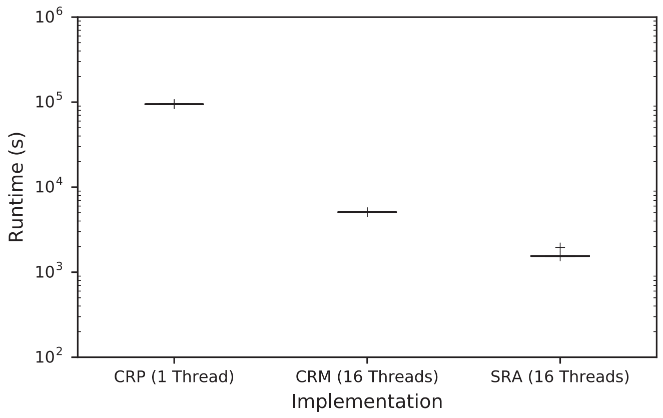


Figure E.63: Runtimes. The runtimes of the following implementations are compared: Commandline Recurrence Plots (CRP), Commandline RQA Multithreaded (CRM) and Scalable Recurrence Analysis (SRA). Note that the implementations are only executed on the Intel Xeon E5620 CPUs of computing system *(C)*. The Commandline Recurrence Plots implementation uses only a single CPU thread, while the remaining implementations use all 16 threads provided by the two CPUs. Note that the runtimes of the *SRA* implementation refers to the ϵ-first strategy. The numerical results are presented in Tab. E.68.

Table E.68: Runtimes. The tabular runtime results referring to Fig. E.63.

Implementation	*Runtime (s)*				
	Min	25%	50%	75%	*Max*
Commandline Recurrence Plots (Single Thread)	94594.06	94596.68	94599.97	94600.50	95038.45
Commandline RQA Multithread (OpenMP)	5067.44	5078.48	5080.10	5083.21	5089.57
Scalable Recurrence Analysis (OpenCL)	1531.44	1544.83	1549.47	1551.06	1955.99

E.3.3 Scalability

Nvidia GeForce GTX 690

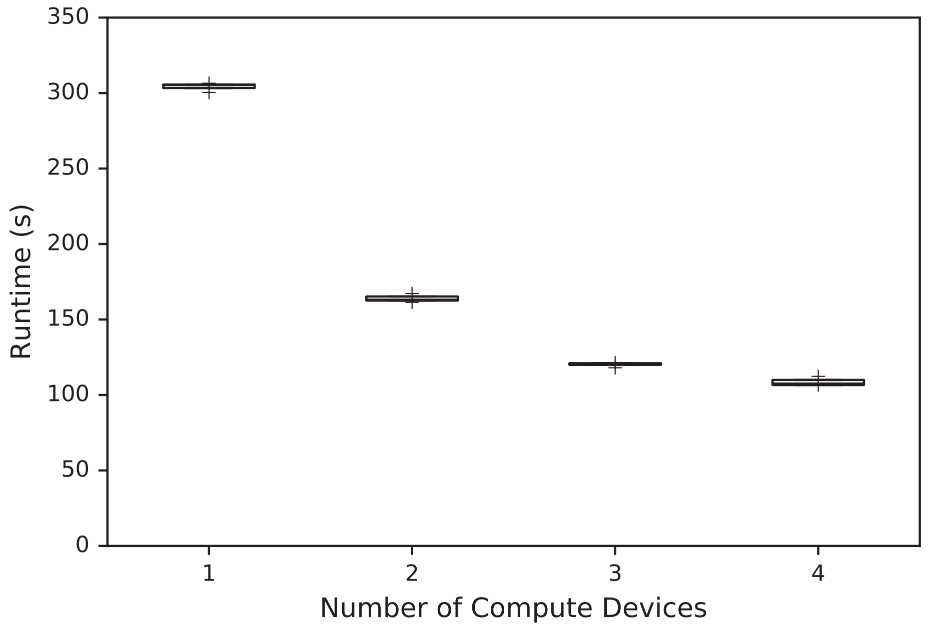


Figure E.64: Runtimes. The OpenCL code is executed on subsets of the four Nvidia GeForce GTX 690 compute devices contained by computing system *(B)*. Note, the ϵ-greedy strategy is applied for flavour selection. The runtimes are captured in tabular fashion in Tab. E.64.

Table E.69: Runtimes - Selection Strategies. The tabular runtime results referring to Fig. E.64.

Number of Compute Devices	*Min*	25%	50%	75%	*Max*
1	300.38	303.34	305.39	305.66	306.52
2	161.36	162.52	163.11	165.25	167.21
3	118.03	119.92	120.45	121.05	121.44
4	106.57	106.58	107.53	109.96	112.33

Bibliography

Advanced Micro Devices, Inc. *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*, June 2012. URL `https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`.

Advanced Micro Devices, Inc. *AMD GPU Performance API - User Guide, Version 2.17*, Dec 2013a. URL `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/GPUPerfAPI-UserGuide-2-17.pdf`.

Advanced Micro Devices, Inc. *AMD Accelerated Parallel Processing - OpenCL Programming Guide - rev2.7*, Nov 2013b.

Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 15h Processors*, Jan 2014. URL `http://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf`.

Advanced Micro Devices, Inc. AMD Joins OpenACC, Jan. 2015. URL `http://developer.amd.com/community/blog/2015/01/07/amd-joins-openacc/`.

Advanced Micro Devices, Inc. Vega: AMD's New Graphics Architecture for Virtually Unlimited Workloads, Jan 2017. URL `http://www.amd.com/en-us/press-releases/Pages/vega-amds-new-2017jan05.aspx`.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, pages 483–485, 1967.

APPLIED PARALLEL COMPUTING LLC. OpenMP 4.0 on NVIDIA CUDA GPUs, Oct. 2015. URL `https://parallel-computing.pro/index.php/9-cuda/43-openmp-4-0-on-nvidia-cuda-gpus`.

A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 918–929, 2006.

ARM. *ARM Cortex-A57 MPCore Processor - Revision: r1p3 - Technical Reference Manual*, Feb 2016. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/DDI0488H_cortex_a57_mpcore_trm.pdf`.

ARM Limited. *ARM NEON Intrinsics Reference*, May 2014.

D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, pages 54–55, Aug 1991.

B. Barney. OpenMP Programming Model, Feb. 2016. URL `https://computing.llnl.gov/tutorials/openMP/#ProgrammingModel`.

H. Bassily and J. Wagner. Application of nonlinear time series principles to assess power generation gas turbine health. volume 10, pages 629–635, 2008.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

R. E. Bellman. *Adaptive control processes - A guided tour.* Princeton University Press, 2015.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244 – 251, 1979.

J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13(2):155–168, 1980.

J. Beyer and J. Larkin. Targeting GPUs with OpenMP 4.5 Device Directives, Apr 2016. URL `http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf`.

P. E. Black. Greedy algorithm. *Dictionary of Algorithms and Data Structures*, 2, 2005. URL `https://xlinux.nist.gov/dads//HTML/greedyalgo.html`.

C. Böhm, B. Braunmüller, F. Krebs, and H. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 379–388, 2001.

S. Burke. Sapphire RX 470 Platinum Review & Benchmark vs. GTX 1060, RX 480 4GB, Aug 2016. URL `http://www.gamersnexus.net/hwreviews/2544-sapphire-rx-470-platinum-review-benchmark-vs-480-gtx-1060`.

S. Carrubba, C. F. II, A. L. Chesson Jr., and A. A. Marino. Numerical analysis of recurrence plots to detect effect of environmental-strength magnetic fields on human brain electrical activity. *Medicinal Engineering & Physics*, 32(8):898–907, 2010.

S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5, 2006.

T. Chelidze and T. Matcharashvili. *Dynamical Patterns in Seismology*, pages 291–334. Springer, Cham, 2015.

Y. Chen and H. Yang. Multiscale recurrence analysis of long-term nonlinear and nonstationary time series. *Chaos, Solitons and Fractals: the interdisciplinary journal of Nonlinear Science, and Nonequilibrium and Complex Phenomena*, 45(7):978–987, 2012.

K. C. Chua, V. Chandran, U. R. Acharya, and C. M. Lim. Computer-based analysis of cardiac state using entropies, recurrence plots and Poincare geometry. *Journal of Medical Engineering & Technology*, 32(4):263–272, 2008.

T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 189–200, 2000.

A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, 49(1):67–104, 2004.

cplusplus.com. stdout, Mar 2016. URL `http://www.cplusplus.com/reference/cstdio/stdout/`.

T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, 2012.

T. de Lima Prado. Recurrence quantification analysis in images with CUDA. Jun 2012.

L. Di Angelo and L. Giaccari. An efficient algorithm for the nearest neighbourhood search for point clouds. *IJCSI International Journal of Computer Science Issues*, 8(5), 2011.

J. F. Donges, R. V. Donner, M. H. Trauth, N. Marwan, H. J. Schellnhuber, and J. Kurths. Nonlinear detection of paleoclimate-variability transitions possibly related to human evolution. *Proceedings of the National Academy of Sciences*, 108(51):20422–20427, 2011.

L. Durant, O. Giroux, M. Harris, and N. Stam. Inside Volta: The World's Most Advanced Data Center GPU, May 2017. URL `https://devblogs.nvidia.com/parallelforall/inside-volta/`.

J.-P. Eckmann, S. Oliffson Kamphorst, and D. Ruelle. Recurrence Plots of Dynamical Systems. *Europhysics Letters*, 5:973–977, 1987.

F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21 (9):948–960, 1972.

W. R. Franklin. Nearest point query on 184m points in e3 with a uniform grid. In *CCCG*, pages 239–242, 2005.

V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2008, Anchorage, AK, USA, 23-28 June, 2008*, pages 1–6, 2008.

J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 331–342, 2001.

A. G. Gray and A. W. Moore. N-body'problems in statistical learning. In *NIPS*, volume 4, pages 521–527. Citeseer, 2000.

S. Green. Particle Simulation using CUDA, July 2012. URL `http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/particles/doc/particles.pdf`.

S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, and W. S. Cleveland. Large complex data: divide and recombine (d&r) with rhipe. *Stat*, 1(1):53–67, 2012.

A. Guillon. An Introduction to OpenCL C++. 2015.

A. Gupta and I. S. Mumick. Materialized views. chapter Maintenance of Materialized Views: Problems, Techniques, and Applications, pages 145–157. 1999.

A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, 1993.

R. Hegger, H. Kantz, and T. Schreiber. TISEAN 3.0.1 - Nonlinear Time Series Analysis, Feb. 2016. URL `http://www.mpipks-dresden.mpg.de/~tisean/Tisean_3.0.1/index.html`.

P. Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, 2nd edition, April 2004.

Intel Corporation. *Writing Optimal OpenCL Code with Intel OpenCL SDK - Revision: 1.3*, 2011.

Intel Corporation. *Intel Xeon Processor E3-1200 v5 Product Family Datasheet - Volume 1 of 2*, Oct 2015. URL `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e3-1200v5-vol-1-datasheet.pdf`.

Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, Feb 2016a.

Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, Feb. 2016b.

Intel Corporation. Intel Many Integrated Core Architecture - Advanced, April 2016c. URL http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *ACM Trans. Database Syst.*, 35(4):24:1–24:43, Oct. 2010.

J. Iwaniec, T. Uhl, W. J. Staszewski, and A. Klepka. Detection of changes in cracked aluminium plate determinism by recurrence analysis. *Nonlinear Dynamics*, 70(1):125–140, 2012.

J. S. Iwanski and E. Bradley. Recurrence plots of experimental data: To embed or not to embed? *Chaos*, 8(4):861–871, 1998.

E. Jones, T. Oliphant, P. Peterson, et al. Open source scientific tools for python, Feb. 2001. URL http://archive.osc.edu/supercomputing/training/python/python_pt2_0702.pdf.

N. Jouppi. Google supercharges machine learning tasks with TPU custom chip , May 2016. URL https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, second edition, 2003. ISBN 9780511755798. Cambridge Books Online.

K. Kato and T. Hosino. Solving k-nearest vector problem on multiple graphics processors. *CoRR*, abs/0906.0231, 2009.

A. Keller. RQA X - Source Code, Feb. 2016. URL http://www.recurrence-plot.tk/rqax.zip.

J. Kessenich. SPIR-V - A Khronos-Defined Intermediate LanguageforNative Representation of Graphical Shaders and Compute Kernels. 2015.

J. Kestelyn. Introducing parquet: Efficient columnar storage for apache hadoop, Mar. 2013. URL https://blog.cloudera.com/blog/2013/03/introducing-parquet-columnar-storage-for-apache-hadoop/.

Khronos Group. Khronos Conformance, April 2016a. URL https://www.khronos.org/conformance/.

Khronos Group. Conformant Products - OpenCL, Mar 2016b. URL https://www.khronos.org/conformance/adopters/conformant-products#opencl.

Khronos Group. OpenCL Resources, April 2016c. URL https://www.khronos.org/opencl/resources.

Khronos OpenCL Working Group. *The OpenCL Specification - Version: 1.1, Document Revision: 44*, Jun 2011.

Khronos OpenCL Working Group. *The OpenCL Specification - Version: 1.2, Document Revision: 19*, Nov 2012.

Khronos OpenCL Working Group. *The OpenCL Specification - Version: 2.1, Document Revision: 23*, Nov 2015.

P. Konsor. Performance Benefits of Half Precision Floats, Aug 2012. URL `https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats`.

D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1111–1120, 2008.

G. Litak, A. K. Sen, and A. Syta. Intermittent and chaotic vibrations in a regenerative cutting process. *Chaos, Solitons & Fractals*, 41(4):2115–2122, 2009.

L. M. Little, P. McSharry, S. J. Roberts, D. A. E. Costello, and I. M. Moroz. Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection. *BioMedical Engineering OnLine*, 6(23):1–19, 2007.

D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6 (1), 2002.

N. Marwan. How to avoid potential pitfalls in recurrence plot based data analysis. *International Journal of Bifurcation and Chaos*, 21(4):1003–1017, 2011.

N. Marwan. COMMANDLINE RECURRENCE PLOTS, Feb. 2016. URL `http://tocsy.pik-potsdam.de/commandline-rp.php`.

N. Marwan. A Comprehensive Bibliography About RPs, RQA And Their Applications, June 2017. URL `http://www.recurrence-plot.tk/bibliography.php`.

N. Marwan and C. L. Webber, Jr. Mathematical and computational foundations of recurrence quantifications. In C. L. Webber, Jr. and N. Marwan, editors, *Recurrence Quantification Analysis*, Understanding Complex Systems, pages 3–43. Springer International Publishing, 2015.

N. Marwan, M. C. Romano, M. Thiel, and J. Kurths. Recurrence Plots for the Analysis of Complex Systems. *Physics Reports*, 438(5–6):237–329, 2007.

N. Marwan, J. F. Donges, Y. Zou, R. V. Donner, and J. Kurths. Complex network approach for recurrence analysis of time series. *Physics Letters A*, 373(46):4246–4254, 2009.

M. M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8, 2007.

Microsoft Corporation. MS-DOS overview, Mar 2016. URL `https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/windows_dos_overview.mspx?mfr=true`.

G. M. Mindlin and R. Gilmore. Topological analysis and synthesis of chaotic time series. *Physica D*, 58(1–4):229–242, 1992.

H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 307–318, 2001.

National Oceanic and Atmospheric Administration. Quality Controlled Local Climatological Data (QCLCD) Publication, Jun 2017. URL `https://data.noaa.gov/dataset/quality-controlled-local-climatological-data-qclcd-publication`.

E. J. Ngamga, A. Nandi, R. Ramaswamy, M. C. Romano, M. Thiel, and J. Kurths. Recurrence analysis of strange nonchaotic dynamics. *Physical Review E*, 75(3):036222, 2007.

NumPy developers. NumPy 1.9.0 - core - fromnumeric.py - sort, May 2014a. URL `https://github.com/numpy/numpy/blob/v1.9.0/numpy/core/fromnumeric.py#L686-L792`.

NumPy developers. NumPy 1.9.0 - arraysetops.py - unique, Jul 2014b. URL `https://github.com/numpy/numpy/blob/v1.9.0/numpy/lib/arraysetops.py#L96-L212`.

NVIDIA Corporation. *NVIDIA's next generation CUDA compute architecture: Fermi - v1.1*, 2009a. URL `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

NVIDIA Corporation. *NVIDIA OpenCL Best Practices Guide - Version 1.0*, Aug 2009b. URL `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`.

NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE - v7.5*, Sep 2015a. URL `https://www3.nd.edu/~zxu2/acms60212-40212/CUDA_C_Programming_Guide_V7.5.pdf`.

NVIDIA Corporation. *Release 349 Graphics Drivers for Windows, Version 350.12*, Apr 2015b. URL `http://us.download.nvidia.com/Windows/350.12/350.12-win8-win7-winvista-desktop-release-notes.pdf`.

NVIDIA Corporation. NVIDIA DGX-1: Essential Instrument for AI Research, May 2017. URL `http://images.nvidia.com/content/technologies/deep-learning/pdf/dgx-1-ai-supercomputer-datasheet-v2.pdf`.

Object Management Group, Inc. *OMG Unified Modeling Language™(OMG UML) - Version 2.5*, Mar 2015.

OpenACC-standard.org. About OpenACC, July 2016. URL `http://www.openacc.org/About_OpenACC`.

OpenMP Architecture Review Board. OpenMP 4.0 API Released, Jul 2013. URL `http://openmp.org/wp/openmp-40-api-released/`.

OpenMP Architecture Review Board. The OpenMP API specification for parallel programming, Feb. 2016. URL `http://openmp.org/wp/`.

V. Pankratius, A.-R. Adl-Tabatabai, and W. Tichy. *Fundamentals of Multicore Software Development.* CRC Press, Inc., 1st edition, 2011.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

M. C. Peel, B. L. Finlayson, and T. A. McMahon. Updated world map of the köppen-geiger climate classification. *Hydrology and earth system sciences discussions*, 4(2):439–473, 2007.

S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An investigation of the performance portability of opencl. *Journal of Parallel and Distributed Computing*, 73(11):1439–1450, 2013.

L. A. Piegl and W. Tiller. Algorithm for finding all k nearest neighbors. *Computer-Aided Design*, 34(2):167–172, 2002.

H. Poincaré. Sur le problème des trois corps et les équations de la dynamique. *Acta mathematica*, 13:1–270, 1890.

D. I. Ponyavin and N. V. Zolotova. Cross Recurrence Plots Analysis of the North-South Sunspot Activities. volume 2004, pages 141–142, 2005.

Potsdam Institute for Climate Impact Research. Long-Term Meteorological Station Potsdam Telegrafenberg, April 2016. URL `https://www.pik-potsdam.de/services/climate-weather-potsdam`.

B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1231–1242, 2013.

T. Rawald. PyRQA - A tool to conduct recurrence quantification analysis (RQA) and to create recurrence plots in a massively parallel manner using the OpenCL framework, Jul 2015. URL `https://pypi.python.org/pypi/PyRQA`.

T. Rawald, M. Sips, N. Marwan, and D. Dransch. Fast Computation of Recurrences in Long Time Series. In *Translational Recurrences. From Mathematical Theory to Real-World Applications*, volume 103 of *Springer Proceedings in Mathematics & Statistics*, pages 17–29. Springer International Publishing, 2014a.

T. Rawald, M. Sips, N. Marwan, and D. Dransch. Fast Recurrence Quantification Analysis on GPUs. In *Proceedings of the 2014 International Symposium on Nonlinear Theory and its Applications (NOLTA2014)*, pages 325–329, 2014b. URL `http://www.epapers.org/nolta2014/ESR/paper_details.php?PHPSESSID=9k8143lu32hkt6ki33du80gqq6&paper_id=6033`.

T. Rawald, M. Sips, N. Marwan, and D. Dransch. Fast Recurrence Quantification Analysis on GPUs. In *EGU General Assembly 2014*, volume 16 of *Geophysical Research Abstracts*, 2014c. URL `http://meetingorganizer.copernicus.org/EGU2014/EGU2014-14824.pdf`.

T. Rawald, M. Sips, N. Marwan, and U. Leser. Massively parallel analysis of similarity matrices on heterogeneous hardware. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 56–62, 2015.

T. Rawald, M. Sips, and N. Marwan. PyRQA - Conducting recurrence quantification analysis on very long time series efficiently. *Computers & Geosciences*, 2016.

H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, pages 527–535, 1952.

V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl. The operator variant selection problem on heterogeneous hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, pages 1–12, 2015.

M. A. Roth and S. J. V. Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An experimental study on performance portability of opencl kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, page 3, 2010. URL `http://saahpc.ncsa.illinois.edu/papers/paper_2.pdf`.

T. Rybak. Using GPU to Improve Performance of Calculating Recurrence Plot. `http://www.wi.pb.edu.pl/pliki/nauka/zeszyty/z6/Rybak-full.pdf`, 2010.

Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition.* Society for Industrial and Applied Mathematics, 2 edition, Apr. 2003.

H. Samet. *Foundations of Multidimensional and Metric Data Structures.* The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123694469.

T. Sauer, J. A. Yorke, and M. Casdagli. Embedology. *Journal of statistical Physics*, 65(3-4): 579–616, 1991.

D. Schultz, S. Spiegel, N. Marwan, and S. Albayrak. Approximation of diagonal line based measures in recurrence quantification analysis. *Physics Letters A*, 379(14-15):997–1011, 2015.

scikit-learn developers. scikit-learn - 1.6. Nearest Neighbors, Oct. 2016. URL `http://scikit-learn.org/stable/modules/neighbors.html`.

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

F. Takens. Detecting strange attractors in turbulence. In Warwick, editor, *Dynamical systems and turbulence*, pages 366–381. Springer, 1980.

M. Thiel, M. C. Romano, P. L. Read, and J. Kurths. Estimation of dynamical invariants without embedding by recurrence plots. *Chaos*, 14(2):234–243, 2004.

D. P. Todey, D. Herzmann, and E. Takle. The Iowa Environmental Mesonet - combining observing systems into a single network. In *Sixth Symposium on Integrated Observing Systems*, 2002.

L. Torvalds et al. The Linux Kernel Archives, Jan 2017. URL `https://kernel.org/`.

N. Trevett. OpenCL: State of the Nation. In *International Workshop on OpenCL (IWOCL)*, May 2017. URL `https://www.khronos.org/assets/uploads/developers/library/2017-iwocl/IWOCL-Neil-Trevett-Keynote_May17.pdf`.

J. W. Tukey. *Exploratory Data Analysis.* Addison Wesley, 1977.

P. Van Leeuwen, D. Geue, M. Thiel, D. Cysarz, S. Lange, M. C. Romano, N. Wessel, J. Kurths, and D. H. W. Grönemeyer. Influence of paced maternal breathing on fetal – maternal heart rate coordination. *Proceedings of the National Academy of Sciences*, 106(33):13661–13666, 2009.

J. Vanderplas. Benchmarking nearest neighbor searches in Python, Apr. 2013. URL `https://jakevdp.github.io/blog/2013/04/29/benchmarking-nearest-neighbor-searches-in-python/`.

C. L. Webber Jr. Charles L. Webber, Jr., Ph.D. - RQA SOFTWARE, Feb. 2016. URL `http://homepages.luc.edu/~cwebber/`.

C. L. Webber Jr. and J. P. Zbilut. Dynamical assessment of physiological systems and states using recurrence plot strategies. *Journal of Applied Physiology*, 76(2):965–973, 1994.

C. L. Webber Jr. and J. P. Zbilut. *Recurrence quantification analysis of nonlinear dynamical systems*, pages 26–94. National Science Foundation (U.S.), 2005.

R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

C. Witt. *Clustering von Recurrence Plots*. Humboldt University of Berlin, 2015.

C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140, 2008.

C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 916–927, 2009.

J. P. Zbilut and C. L. Webber Jr. Embeddings and delays as derived from quantification of recurrence plots. *Physics Letters A*, 171(3–4):199–203, 1992.

J. P. Zbilut, M. Koebbe, H. Loeb, and G. Mayer-Kress. Use of Recurrence Plots in the Analysis of Heart Beat Intervals. pages 263–266, 1990.

J. P. Zbilut, J.-M. Zaldívar-Comenges, and F. Strozzi. Recurrence quantification based Liapunov exponents for monitoring divergence in experimental data. *Physics Letters A*, 297(3–4):173–181, 2002.

Y. Zhang, M. S. II, and A. A. Chien. Improving performance portability in opencl programs. In *Supercomputing - 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, pages 136–150, 2013.

Z. Q. Zhao and S. C. Li. Identifying spatial patterns and dynamic of climate change using recurrence quantification analysis – case study of qinghai-tibet plateau. *International Journal of Bifurcation and Chaos*, 21(4):1127–1139, 2011.

J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 526–535, 2007.

D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.