Massively Parallel Analysis of Similarity Matrices on Heterogeneous Hardware

Tobias Rawald
Humboldt-Universität zu Berlin
and
Helmholtz Centre Potsdam - GFZ German Research Centre for Geosciences
trawald@gfz-potsdam.de

Mike Sips
Helmholtz Centre Potsdam - GFZ German Research Centre for Geosciences
sips@gfz-potsdam.de

Norbert Marwan
Potsdam Institute for Climate Impact Research
marwan@pik-potsdam.de

Ulf Leser
Humboldt-Universität zu Berlin
leser@informatik.hu-berlin.de

ABSTRACT

We conduct a study that investigates the performance characteristics of a set of parallel implementations of the recurrence quantification analysis (RQA) using OpenCL. Being an important tool in climate impact and medical research, a central aspect of RQA is the construction of a binary matrix that captures the similarities of multi-dimensional vectors. Based on this matrix, quantitative measures are derived. Starting with a baseline implementation, we diversify its properties along four dimensions: the representation of input data, the materialisation of the similarity matrix, the representation of similarity values and the recycling of intermediate results. We evaluate the performance of five implementations by varying the input parameter assignments, the hardware platform employed for execution and the default OpenCL compiler optimisations status. We come to the conclusion that the performance of conducting RQA highly depends on the selected implementation as well as the combination of these variables under investigation. Differences in runtime of up to one order of magnitude are observed, emphasising the importance of performance studies as presented here.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; G.1.0 [**Numerical Analysis**]: General—*Parallel Algorithms*

Keywords

Similarity Matrix, Parallel Algorithm, Heterogeneous Hardware, Recurrence Quantification Analysis

1. INTRODUCTION

Recurrence quantification analysis (RQA) is a statistical method to quantify the recurrent behaviour of dynamic systems, captured in one or more time series [11]. It has proven its potential in a variety of applications, such as the investigation of the climate system [12] and the early detection of epileptic states [3].

RQA is based on extracting multi-dimensional vectors from time series; each vector corresponds to a reconstructed state of the system at a point in time. To identify recurrences, these vectors are compared regarding their mutual similarities. The results of the comparisons are stored within a binary similarity matrix.

Matrix elements referring to pairs of vectors considered to be similar form vertically and diagonally connected sequences. Using frequency distributions of those lines, RQA derives quantitative measures. They allow to draw conclusions concerning the dynamics of the system under investigation [11].

Focussing on very long time series, in [13] we introduced coarse-grained parallelisation strategies to the problem of RQA. We presented an approach that divides the similarity matrix into multiple sub matrices, computing intermediate results for each sub matrix. This allows to process several sub matrices concurrently. Within a final step, the intermediate results are recombined into a global RQA result.

Even though our approach is independent of the concrete implementation, in [13] we compare a non-parallel version of RQA to a prototype of our approach based on OpenCL, which performs parts of the computation in a massively parallel manner. Exploiting the parallel computing capabilities of modern GPU processors, we achieved drastic performance improvements.

However, executing the prototype on different hardware platforms, we discovered that the relative performance improvements vary. Hence, in this publication we conduct a study that exemplarily examines a selection of factors influencing the overall performance characteristics of RQA.

We provide five implementations, which differ concerning input data representation, similarity matrix materialisation, similarity value representation and intermediate results recycling. Given a specific implementation, we investigate

Time Series:

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
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

$m = 2$ (Embedding Dimension)
$t = 2$ (Time Delay)

Extracted Vectors:

s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}
0.0	0.7	1.0	0.7	0.0	-0.7	-1.0	-0.7	0.0	0.7	1.0
1.0	0.7	0.0	-0.7	-1.0	-0.7	0.0	0.7	1.0	0.7	0.0

Figure 1: Vector Extraction. Given a time series capturing the sine function at multiples of $\pi/4$ starting at 0, consisting of thirteen data points. Applying the parameter values $m = 2$ and $t = 2$, eleven vectors are extracted.

the influence of the RQA input parameter assignments, the hardware platform used for execution and whether default OpenCL compiler optimisations are enabled.

The results of our experiments show, that the performance of each implementation highly depends on the combination of hardware platform, default OpenCL compiler optimisations status as well as RQA input parameter assignments. Providing general guidelines, we support the selection of an implementation given a specific RQA scenario as well as computing environment (see Sect. 5.2). Recognising the fact that the exploration space covered is limited, we see this study as a first effort to address the performance comparison of parallel RQA implementations.

We believe that our work, apart from providing highly interest into the nature of RQA, is also relevant for other application areas that face similar problems, including nearest neighbour search.

2. OVERVIEW OF RECURRENCE QUANTIFICATION ANALYSIS

Recurrence quantification analysis is a method in the context of time series analysis [11]. It is based on:

1. extracting multi-dimensional vectors from a set of time series,
2. creating a similarity matrix by calculating pairwise vector similarities, and
3. quantifying small-scale structures within the similarity matrix.

There are several approaches for conducting each of these steps. For the sake of clarity, in this paper we consider performing RQA with the following properties: We are given a single time series consisting of floating point numbers; each value refers to a measurement of an output variable, e.g., the air temperature, of a dynamic system, e.g., the Earth's climate, at a specific point in time. To extract the multi-dimensional vectors, the so called *time delay* method is applied, building on the two parameters *embedding dimension* (m) and *time delay* (t). Starting at the first element of the time series, vectors of size m with the temporal offset t are extracted (see Fig. 1).

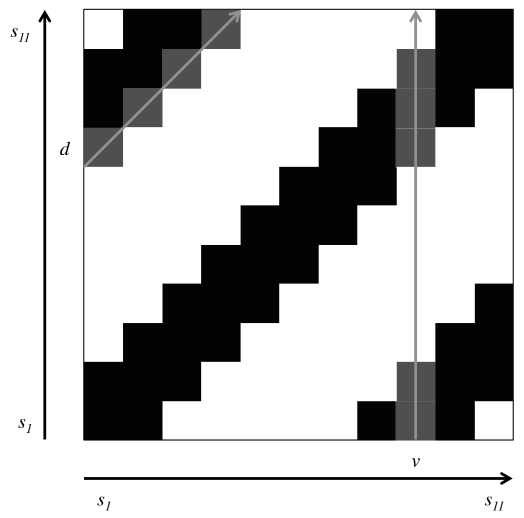


Figure 2: Thresholded Recurrence Plot. Referring to the example from the previous figure, the eleven vectors extracted are compared regarding their mutual similarities. Concerning the vector comparisons, the Euclidean norm is applied, using a similarity threshold of 1.0. The column v contains two lines; one of length 2 and one of length 3. The diagonal d comprises a line of length 4.

To compare those vectors concerning their mutual similarity, a metric such as the Euclidean norm is applied. By introducing a threshold condition regarding the vector similarities, all matrix elements fulfilling the condition are assigned the value 1 (recurrence point), whereas pairs of non-similar vectors are assigned the value 0. A visual representation of this matrix is referred to as thresholded recurrence plot (see Fig. 2). Recurrence points, encoded using the colour black, form vertical and diagonal lines, which are captured in corresponding histograms of line lengths. Based on these histograms, quantitive measures are calculated, including for example the average vertical line length.

3. PARALLEL RQA ALGORITHM

To enable a systematic analysis, we divide the problem of conducting RQA into three operators:

I The creation of the binary similarity matrix. (*create_matrix*)

II The detection of vertical lines within the similarity matrix. (*detect_vertical_lines*)

III The detection of diagonal lines within the similarity matrix. (*detect_diagonal_lines*)

We refine these operators into atomic units of computation:

I The computation of the similarity of a *single pair of multi-dimensional vectors.*

II The inspection of a *single column* of the similarity matrix concerning vertical lines.

III The inspection of a *single diagonal* of the similarity matrix concerning diagonal lines.

Having extracted N multi-dimensional vectors, the maximum degree of parallelism varies between N^2 (I), N (II) and $2N-1$ (III).

Performing similar operations on different data objects, each atomic unit is fully independent of any other unit regarding the execution of a single operator. However, there exist interdependencies between atomic units belonging to different operators: Prior to the detection of lines within a single column or diagonal, the corresponding vector similarities have to be computed.

The structure presented above allows us to perform RQA in a parallel manner. Although subdividing the problem into multiple operators, we mainly focus on the cumulative performance of all operators, regarding the evaluation.

4. EXPERIMENTAL SETUP

4.1 Implementation Strategies

Building on the OpenCL framework, we consider a computing environment that consists of a *host device* and a single *computing device*. The code executed on the host device is written in *Python* 2.7, utilising the package *PyOpenCL*. The atomic units of computation described in Sect. 3 are mapped to OpenCL kernels, implemented in *OpenCL C*.

We provide five RQA implementations, which differ along the following dimensions:

- Input Data Representation,
- Similarity Matrix Materialisation,
- Similarity Value Representation, and
- Intermediate Results Recycling.

In the following, we introduce each dimension and motivate the corresponding values. Regarding the evaluation, we include only a subset of possible value combinations. Nevertheless, we ensure that each value is featured within at least one implementation. Tab. 1 gives an overview of the individual properties of each implementation considered (see Impl. *A*–*E*).

Input Data Representation

Conducting RQA, multi-dimensional vectors are extracted from a time series. Regarding their representation within the memory of the computing device, the set of vectors may either be stored row-wise or column-wise. Choosing a *Row-Store* layout, all components of a single vector are stored consecutively. This requires to reorganise the data given by the input time series.

However, having to perform read-only operations on the vector data, a *Column-Store* layout [16] may be advantageous. Applying this approach, all values belonging to the same vector component are stored contiguously. Since segments of the input time series represent those columns, it can be transferred to the memory of the computing device without having to perform reorganisations.

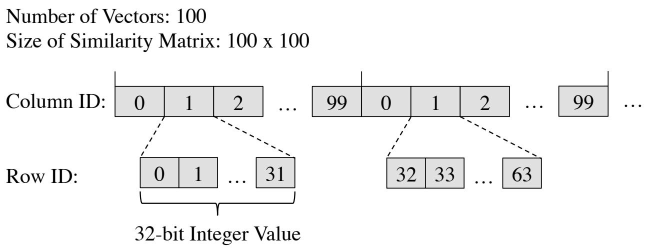


Figure 3: Bitwise Similarity Value Representation. The 32 bits of an integer value refer to a single column. Integer values stored contiguously refer to different columns. Each bit within an integer value refers to a different row of the similarity matrix.

Similarity Matrix Materialisation

The vectors extracted from the time series are compared regarding to their mutual similarities. The resulting binary similarity values are used as input for the detection of vertical and diagonal lines. The corresponding similarity matrix may be stored within the memory of the computing device (*Yes*). This requires that the size of this memory is sufficiently large enough.

Avoiding this restriction, the similarity values may be computed on-the-fly by transferring the computations to the operators for detecting vertical and diagonal lines (*No*). Previous work has shown that the computation of the pairwise similarities requires extensive computing [4]. We are interested, if there are conditions where computing similarity values outperforms writing them to and reading them from the memory.

Similarity Value Representation

Since device memory is a limited resource, the similarity matrix shall be represented in the most efficient manner. Using the bit-compression approach [14], a single bit is used to encode the binary result of a similarity comparison (*Bit*). A schematic illustration of the underlying memory layout is depicted in Fig. 3.

Considering the detection of lines, this approach allows to process up to 32 similarity values of a single column without having to read additional data from the memory. In addition, it ensures that similarity values belonging to different columns are read using a single read instruction.

Nevertheless, this compression approach may introduce a computing overhead, having negative effects on the overall performance. Hence, we compare it to representing a similarity value using the smallest data object addressable (*Byte*).

Intermediate Results Recycling

To avoid matrix materialisation, similarity values may be computed on-the-fly during the line detection process, as explained earlier. Assuming that the execution model adheres to operator-at-a-time, similarity values computed within one line detection operator may be reused later on. Applying this concept of recycling [7], performance improvements may be exposed.

Omitting the *create_matrix* operator, we integrate the materialisation of the similarity values in *detect_vertical_lines* and reuse the results during the detection of diagonal lines

Table 1: Implementation Comparison.

Dimension	Value	Impl. *A*	Impl. *B*	Impl. *C*	Impl. *D*	Impl. *E*
Input Data Representation	Row-Store	✓				
	Column-Store		✓	✓	✓	✓
Similarity Matrix Materisalisation	Yes	✓	✓		✓	✓
	No			✓		
Similarity Value Representation	Byte	✓	✓			✓
	Bit				✓	
Intermediate Results Recycling	Yes					✓
	No	✓	✓	✓	✓	

(*Yes*). Here, the challenge is that the maximum degree of parallelism for detecting vertical lines is significantly smaller than creating the similarity matrix individually (*No*). Thus, our goal is to reveal whether there are conditions under which the positive impact of eliminating one operator is large enough to overcome this limitation.

4.2 Hardware Platforms

We evaluate each implementation using three computing devices. Each device is part of a system that runs on a 64-bit version of *openSUSE*. This includes an *Intel Core i7-3820* CPU running at up to 3.8 GHz, which is supplied with 16 GB of random access memory.

In addition, we employ an *NVIDIA GeForce GTX 690* graphics card, equipped with two GPU processors running at up to 1.019 GHz; each processor is supplied with 2 GB of memory. In the context of our evaluation, only one of those processors is used. The underlying system has version 331.49 of the NVIDIA graphics driver installed.

Adding diversity regarding the GPU architectures, we employ an *AMD Radeon HD 7470* GPU, equipped with a single processor running at up to 0.775 GHz. It is supplied with 0.5 GB of memory. The underlying system has version 14.9 of the AMD Catalyst driver installed.

4.3 Parameter Space

Given the three hardware platforms, we identified the following factors additionally influencing the performance characteristics:

- the parameters steering the properties of the similarity matrix, including:
  - the time series,
  - the embedding dimension,
  - the time delay,
  - the similarity measure, and
  - the similarity threshold, as well as
- the default OpenCL compiler optimisations.

To restrict the exploration space, we reduce the number of degrees of freedom addressed within the evaluation to two. This includes varying the embedding dimension between 1 and 32. Moreover, we observe the impact of disabling the default OpenCL compiler optimisations using the compiler flag *-cl-opt-disable*. We consider evaluating the impact of those optimisations as highly relevant, since they are vendor specific and may affect the computing results, e.g., the default activation of relaxed math operations on the NVIDIA GPU.

We employ a time series capturing the sine function, similar to Fig. 1, consisting of 10,000 data points. We choose this rather short length since we have to ensure that the resulting similarity matrix fits into the memory of all computing devices applied.

Regarding the similarity comparisons, we select the Euclidean norm in combination with a threshold of 1.0. Initial experiments have shown that the time delay parameter does not have considerable influence on the performance. Hence, we set this parameter to 2.

5. EVALUATION

5.1 Procedure

Concerning the evaluation, we consider an experiment to be a combination of:

- hardware platform,
- implementation,
- embedding dimension, and
- default OpenCL compiler optimisations status.

To reduce the impact of outliers, we conduct each experiment five times. For the purpose of measuring the runtime behaviour of the implementations, we rely on *profiling events* as part of the OpenCL API, collecting information about the average runtime of the three operators. Furthermore, we use the *sprofile* [1] command line tool to retrieve extended performance information provided by the AMD GPU.

5.2 General Guidelines

The cumulative runtime results are depicted in Fig. 4, having the default OpenCL compiler optimisations disabled, and Fig. 5, having them enabled.

As expected, increasing the dimensionality of the vectors, the runtime increases as well. Enabling the default compiler optimisations has a positive impact on the cumulative runtime, independent of the implementation as well as the hardware platform employed. Considering the GPU devices, implementation *A*, using a row-wise layout for storing the multi-dimensional vectors, benefits the least. Whereas the relative difference in runtime between *A* and the other implementations is narrow considering the CPU, it widens more drastically regarding the GPU devices. Hence, considering GPU devices, a row-wise layout should be avoided.

Compared to the other implementations, B shows well-balanced performance characteristics, relying on the column-wise memory layout. Applying an embedding dimension of 32, it is among the two fastest implementations independent of the hardware platform applied. Eliminating the similarity matrix materialisation, implementation C delivers performance improvements considering small embedding dimensions, as expected.

The usage of the bit-representation in implementation D proves to be reasonable for larger embedding dimensions. The corresponding runtime curves start at a higher plateau, but have the smallest slope, independent of hardware platform and default compiler optimisations status. Diminishing the compression overhead with increasing dimensionality, the curves of D converge towards the corresponding curves of B.

Recycling intermediate results, as employed in implementation E, does not present runtime benefits across all hardware platforms. Considering the CPU, it is the fastest implementation, for nearly all embedding dimension values. Regarding the GPU devices, E delivers runtime improvements for vectors having small dimensionality, but is eventually outperformed by implementation B and D.

Considering a given hardware platform, time series as well as RQA input parameter assignments, we propose employing an implementation that comprises the following features:

- column-wise input data representation,
- materialisation of the similarity matrix,
- byte representation of the similarity values, and
- usage of a separate *create_matrix* operator.

Although this combination does not deliver the best performance under all circumstances, it appears to be a reasonable choice based on the evaluation results.

5.3 Detailed Performance Analysis

We present selected details on the impact of using different implementation strategies. The runtime results as well as the performance counter values listed below refer to an embedding dimension of 32.

Input Data Representation

Comparing the hardware platforms applied, the row-store layout for representing the vectors has the least worst impact considering the CPU. Having the default compiler optimisations disabled, the *create_matrix* operator of implementation A (0.79s) is as nearly as fast as the same operator of B (0.75s). Enabling the optimisations, creating the matrix in A (0.44s) consumes twice as much runtime as in B (0.22s).

Additionally, the impact of changing the access pattern to the device memory is illustrated by the cache hit rate produced on the AMD GPU. Disabling the compiler optimisations, the *create_matrix* operator of A has a rate of 23.39%, whereas executing the same operator of B results in a rate of 91.36%.

Similarity Matrix Materialisation

Not materialising the similarity matrix presents advantages concerning the cumulative runtime using small embedding dimensions. Regarding the NVIDIA GPU, the break-even point of implementation B and C is a dimensionality of 3.

Experiencing a drastic increase in fetch operations, the ratio between the amount of *arithmetical logical unit* (ALU) instructions performed by the AMD GPU in comparison to the number of *fetch unit* instructions decreases; from 17.97 (B) to 2.25 (C) regarding the detection of vertical lines, having the default optimisations enabled.

Similarity Value Representation

Encoding similarity values using a single bit leads to an increase in ALU instructions for all three operators, reflecting the corresponding computing overhead. However, the custom layout presented in Sect. 4 improves the memory access. Considering the AMD GPU, this results in an increased cache hit rate for detecting diagonal lines; from 3.26% (B) to 21.52% (D), having the default compiler optimisations enabled.

Intermediate Results Recycling

Focussing on the execution on the CPU, the reuse of similarity values in E is advantageous compared to any other implementation. Enabling the default OpenCL compiler optimisations, implementation E (0.31s) outperforms its direct successor B (0.37s), regarding the cumulative runtime.

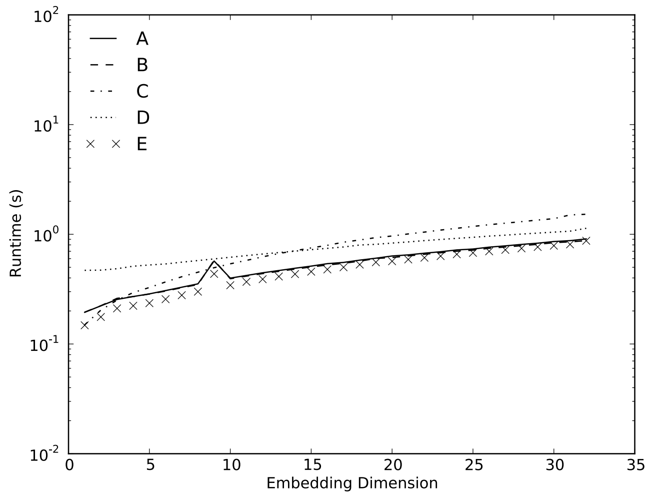
6. RELATED WORK

A number of RQA implementations are available, posing restrictions concerning the size of the similarity matrices that can be processed [10, 17]. The *Commandline Recurrence Plots* (CRP) software allows to analyse time series of arbitrary size [9]. However, it relies on computing the RQA measures using a single CPU thread. For an overview of free RQA software, we refer to [2].

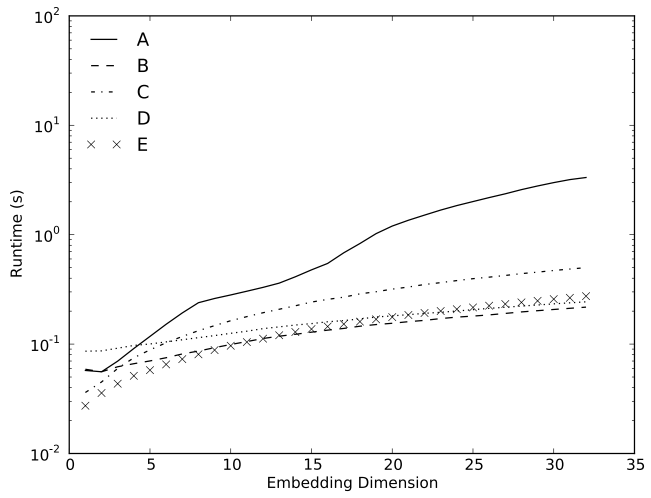
In [15] prior efforts to bring RQA to the GPU are described, comprising several limitations that hamper the analysis of long time series. This includes being restricted to similarity matrices that fit into the memory of the GPU device. Relying on the concepts of *Divide & Recombine* [6], our approach presented in [13] allows to process similarity matrices of arbitrary size. We demonstrated the capabilities of our approach for a specific RQA scenario from climate impact research. Examining a time series consisting of over one million data points, we were able to reduce the runtime from over six hours, using the CRP software, to almost five minutes, using an OpenCL implementation of our approach running on two GPUs.

Considerable efforts have been made to accelerate database operations. Exploiting the computing capabilities of general-purpose graphics cards, in [5] several parallel implementations for database operations, such as semi-linear query, are presented. The conclusion is that depending on the operation investigated, GPUs enable drastic performance improvements.

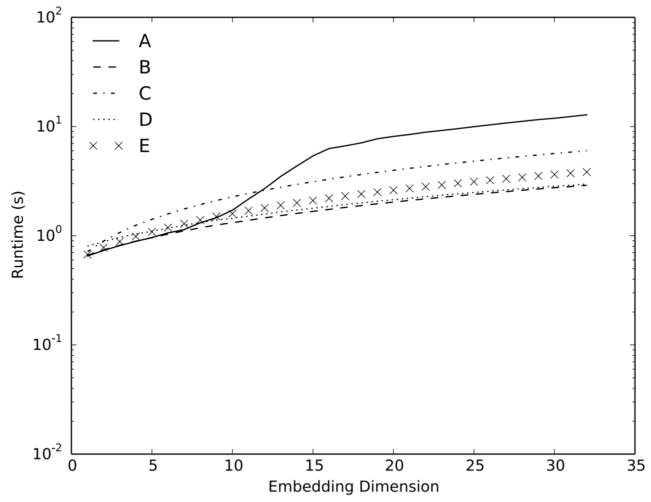
A prominent database operation similar to RQA is the k-nearest neighbour search (kNN). Within both techniques, comparing a set of objects regarding their mutual similarities is a key aspect. Adapting kNN processing to many-core systems, a large amount of similarity comparisons is performed concurrently. Experimental results illustrate that executing a parallelised version of the algorithm on the GPU is two orders of magnitudes faster than performing the search on the CPU [4].



(a) Intel Core i7-3820

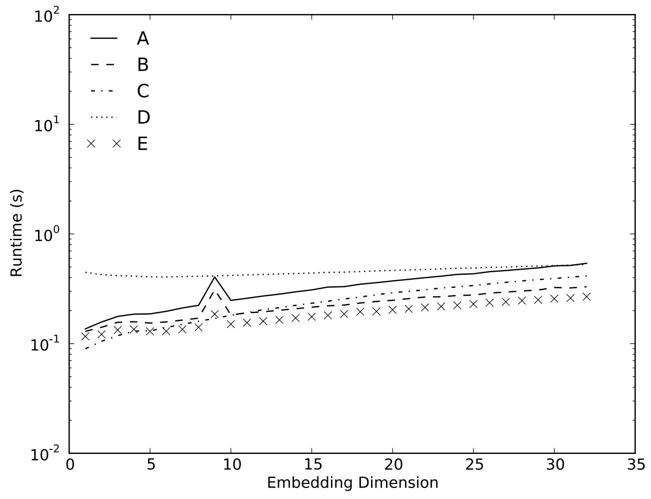


(b) NVIDIA GeForce GTX 690

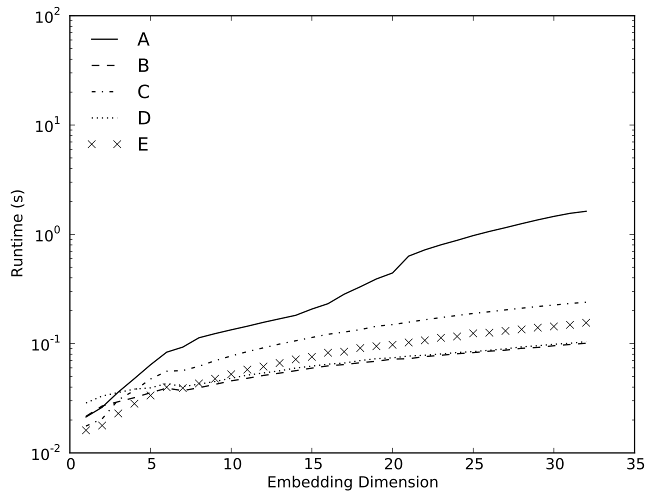


(c) AMD Radeon HD 7470

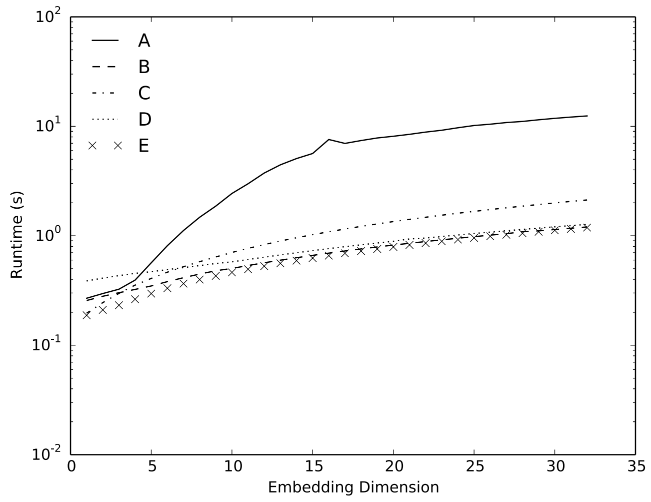
Figure 4: Disabling Default OpenCL Compiler Optimisations. Cumulative runtime of executing the operators *create_matrix*, *detect_vertical_lines* and *detect_diagonal_lines*.



(a) Intel Core i7-3820



(b) NVIDIA GeForce GTX 690



(c) AMD Radeon HD 7470

Figure 5: Enabling Default OpenCL Compiler Optimisations. Cumulative runtime of executing the operators *create_matrix*, *detect_vertical_lines* and *detect_diagonal_lines*.

Previous work focussed on employing a set of optimisations to gain runtime improvements on a specific device. To the best of our knowledge, we provide the first structured approach to analyse the performance characteristics of parallel RQA implementations. In this regard, we benefit from using the OpenCL framework for heterogeneous computing [8], which allows us to execute identical code on a variety of hardware platforms.

7. CONCLUSION

We present a structured approach to evaluate the performance of five parallel implementations analysing binary similarity matrices in the context of RQA. Assessing the performance of each implementation, we vary their characteristics along four dimensions, including the representation of input data, the materialisation of the similarity matrix, the representation of the similarity values as well as the recycling of intermediate results.

Building on the OpenCL framework, we investigate the influence of the hardware platform used for execution, input parameter assignments and default OpenCL compiler optimisations enabled on the performance. We examine the runtime behaviour as well as additional indicators, e.g., the cache hit rate. We come to the conclusion, that an implementation using column-wise input data representation in combination with similarity matrix materialisation provides reasonable performance, regarding a given RQA scenario. Subsuming, we see our study as a first effort towards a comprehensive analysis of parallel RQA implementations.

8. ACKNOWLEDGEMENTS

This work is supported by grants from the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

9. REFERENCES

[1] Advanced Micro Devices, Inc. APP Profiler Settings. `http://developer.amd.com/tools-and-sdks/archive/amd-app-profiler/user-guide/app-profiler-settings/`, 2014.

[2] J. Belaire-franch and D. Contreras. Recurrence plots in nonlinear time series analysis: Free software. *Journal of Statistical Software*, 2002.

[3] 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.

[4] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2008.

[5] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[6] 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.

[7] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM.

[8] Khronos Group. OpenCL 1.1 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, Sept. 2010.

[9] N. Marwan. Commandline Recurrence Plots, Version 1.13z. `http://tocsy.pik-potsdam.de/commandline-rp.php`, 2006.

[10] N. Marwan. CRP Toolbox, Version 5.17. `http://tocsy.pik-potsdam.de/CRPtoolbox`, 2013. platform independent (for Matlab).

[11] 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.

[12] D. I. Ponyavin and N. V. Zolotova. Cross Recurrence Plots Analysis of the North-South Sunspot Activities. volume 2004, pages 141–142, 2005.

[13] 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, 2014.

[14] M. A. Roth and S. J. Van Horn. Database Compression. *SIGMOD Rec.*, 22(3):31–39, Sept. 1993.

[15] 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.

[16] 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.

[17] C. L. Webber Jr. RQA Software, Version 14.1. `http://homepages.luc.edu/~cwebber`, 2013. only for DOS.