



Universität Stuttgart

Institut für Parallele und Verteilte Systeme

Universitätsstraße 32
70569 Stuttgart

Masterarbeit
**Design and implementation of a
GPU-based simulation- and
analysis-environment for
dynamical systems**

Sebastian Künzel

Studiengang: Informatik
1. Prüfer: Herr Prof. Dr. rer. nat. habil. Holger Schwarz
2. Prüfer: Herr Prof. Dr. rer. nat. habil. Viktor Avrutin
Betreuer: Herr Prof. Dr. rer. nat. habil. Viktor Avrutin

begonnen am: 02.01.2022

beendet am: 02.07.2022

June 29, 2022

Abstract

In this thesis, a high-performance GPU-based simulation- and analysis-environment for dynamical systems is designed, implemented, and tested. Computation is optimized on the basis of the GPU hardware architecture. Minimal task-specific simulation programs are created using runtime compilation. Scan execution and analysis methods are designed and optimized for massive parallelization and integrated into a modular, efficient, scalable, and streamlined architecture. The new program is tested for performance and result accuracy. The impact of floating-point inaccuracy is discussed and shows how the current implementation of AnTGPU can mitigate this problem.

Contents

1. Introduction	9
2. Dynamical systems	11
2.1. Analysis of dynamical systems	12
2.2. Numerical analysis	12
3. The AnT Project	15
3.1. AnT scan architecture	16
3.2. AnT dynamical system analysis	17
3.2.1. AnT analysis methods	18
3.2.2. Performing analysis in AnT	20
4. GPU Computing	23
4.1. GPU hardware architecture	23
4.2. Strengths and weaknesses of GPUs computation	24
4.3. GPU computing frameworks	26
5. Goal and tasks of this thesis	29
6. Assessing GPU acceleration of AnT	31
7. AnTGPU: Architecture development	35
7.1. Computation flow	36
7.2. Abstract architecture	39
7.3. Dynamic kernel generation	39
7.4. Method execution and cross-thread computation	41
7.4.1. Trajectory iteration architecture	41
7.4.2. Cross-thread computation	41
7.5. CPU postprocessing	42
8. AntGPU: Method development	47
8.1. General trajectory evaluation	47

8.2. Period analysis	50
8.3. Lyapunov exponents analysis	52
8.4. Density analysis	54
8.5. Bandcounting	56
8.6. Symbolic sequence analysis	58
9. Testing and evaluation of AnTGPU	65
9.1. Performance comparison against AnT	66
9.2. Quality evaluation	69
9.2.1. Comparison with AnT	69
9.2.2. Mitigating floating-point number inaccuracy	72
9.3. AnTGPU on nested closed invariant curves	73
10. Conclusion and Outlook	81
Bibliography	83
A. AnTGPU User documentation	89
A.1. Config file documentation	89
A.1.1. Features	89
A.1.2. Syntax example	90
A.1.3. Option keys	90
A.2. System function file documentation	92
A.2.1. Syntax	93
A.2.2. Predefined variables	94
A.2.3. Forbidden variable names	94
A.2.4. Example function	95
A.3. Output file documentation	95
A.4. Program execution documentation	96
B. Example scan configurations	97
B.1. Modified logistic function & General evaluation	97
B.2. Gingerbreadman map & Period analysis	98
B.3. Hénon map & Lyapunov exponents	99
B.4. Rössler system & Lyapunov exponents	99
B.5. Tent map & Density analysis	100
B.6. PWS map & Bandcounting	101
B.7. PWS map & Period analysis	102
B.8. PWS map & Symbolic analysis	102

C. Example system functions	105
C.1. PWS map & Symbolic analysis	105
C.2. Rössler system & Lyapunov exponents	105

1. Introduction

In a dynamical system, a function describes the time-dependent evolution of a point, also referred to as a state, in space. In mathematics, physics, biology, and economics dynamical systems are used to model evolving behavior. Examples include the movement of celestial bodies or oscillating waveforms in a circuit [BS02] [CIKE93]. Typical questions revolve around the long-term evolution of a state or a set of states. This allows giving predictions on the behavior of a given system. Does a state converge to a fixed point or a periodic sequence of points? This question is investigated by period analysis. Do two initially close states diverge over time? The rate of divergence can be quantified by calculating Lyapunov exponents [WSSV85].

The above-mentioned and many more questions are answered by analyzing the system function. This includes the investigation of fixed points and attractors. In many systems, equations can be solved analytically and derivatives are easily calculated. In other cases, system functions are too complex to be analyzed analytically. In those cases, the analysis of the system is performed numerically. Many results in past and present work are based on numerical analysis of the respective system [SH98]. Often, programs are designed to perform a specific analysis on a predefined system. Apart from system-specific analysis programs there exist general analysis packages. One example is AnT, developed at the University of Stuttgart in 2001. AnT is still used today and has an active international userbase. In AnT the user can input a system function and select from a wide variety of analysis methods to be performed on the given system [Scho4].

In the last 20 years, there has been enormous progress in hardware performance. Parallel processing raised to industry standard. Gigabytes of memory and teraflops of processing power became available to home users in the form of high-performance GPUs [Nvib]. In recent years, researchers started to harness GPU processing power to numerically analyze dynamical systems [PMS21].

This work reports the results of developing a tool for the analysis of dynamic systems on the GPU that builds on the flexible, user-friendly, and function-rich AnT package. The user does not need to know the inner workings of analysis methods and does not need to implement the methods with respect to the specific hardware limitations of the GPU. At the same time, the analysis is performed with specifically designed and optimized analysis methods in a hardware-oriented architecture. This process is split into two phases. In the first phase, the existing AnT project is assessed for GPU acceleration. It has been found that the scan execution architecture of the AnT project can be modified to work efficiently on the GPU hardware. However, the implementation of the architecture and methods as well as the method execution architecture and the system function integration is not compatible with efficient GPU computing. In the second phase, a new program, scan architecture, method execution architecture, and system function integration is developed and implemented with a focus on efficient execution on GPU hardware.

First, an introduction to dynamical systems in the context of this work is given in chapter 2. Additionally, the AnT project and fundamentals of GPU computing are introduced in chapter 3 and chapter 4. A selection of the most important system classes and analysis methods in AnT is made. The selected system classes and analysis methods set the scope of this work. Other features of the AnT project will not be assessed, modified, optimized, or transferred to the GPU.

After that, the results of the assessment of GPU acceleration ability of the existing AnT project are discussed in chapter 6. Next, the process of creating a modified and GPU-oriented architecture is presented in detail in chapter 7. Following, in chapter 8 the selected analysis methods are modified to integrate with the GPU architecture, optimized with regards to processing and memory efficiency, and enhanced by new features. In some cases, the way in which a method performs analysis is changed entirely to allow for a more efficient implementation, while obtaining the same analysis result. Finally, in chapter 9 the new program AnTGPU is compared in speed and accuracy against the existing AnT project. Additionally, scans of nested closed invariant curves are computed with AnTGPU.

2. Dynamical systems

A discrete-time dynamical System consists of a non-empty set X and a map

$$f : X \rightarrow X$$

f is also referred to as system function. Given an initial state $x_0 \in X$, any higher order state can be calculated by iterating $x_{n+1} = f(x_n)$. This defines the n -th iterate of f as $f^n(x) = f \circ \dots \circ f(x_0)$ [BS02]. In practice X is often a subset of \mathbb{R}^m .

A continuous-time dynamical system consists of a space X and a parameterized family of maps

$$f^t : X \rightarrow X, t \in \mathbb{R}$$

with $f^0 = Id$ and $f^a \circ f^b = f^{a+b}, a \rightarrow t_1, b \rightarrow t_2$. Many dynamical systems encountered in practice have a discrete-time and a continuous-time version. [BS02]. A system of ordinary differential equations that are only dependant on the current state is called an autonomous system.

Often, a system has one or several control parameters. Parameters are independent of the state space and remain constant over state transition. State transition induces an order of states often referred to as time. The sequence of states obtained by iterating or integrating an initial state x_0 for a given set of parameters is called orbit or trajectory. A numerical integration step, performed by an arbitrary integration method is also named iteration in this work. In the limit, an orbit may diverge or converge into a limit set. Limit sets may take the form of fixed points, a period of points, a quasi-periodic set, or a chaotic set [SH98]. Given a limit set and a set of system parameters, the basin of attraction is the set of initial states that converge to the limit set. Given a basin of attraction that includes one or multiple states of the state-space, the corresponding limit set is often referred to as the attractor.

2.1. Analysis of dynamical systems

A dynamical system is analyzed by mapping its behavior for a large number of initial states and parameter values. A good initial strategy is given by Stuart [SH98]:

A fairly complete picture of a dynamical system may be obtained by determining all possible limit sets, determining how these limit sets change with respect to control parameters in the system, and then determining the basins of attraction of individual limit sets.

In practice, many aspects of the above strategy are analyzed in greater detail. There are multiple ways in which limit sets change with respect to the system parameters. Investigating these changes is subject to bifurcation theory. Common bifurcations include fold and flip bifurcations [Ste10]. Given a chaotic or quasi-periodic limit set, the limit distribution of states in the set may be of interest. Determining the limit distribution is subject to density analysis. A chaotic limit set can be composed of connected components, typically referred to as bands. It may be of interest to determine the number of disconnected subsets in the chaotic limit set. This is referred to as Bandcounting [Eck06].

Not mentioned in the quoted analysis strategy is quantifying the rate of divergence or convergence of two initially close states on an attractor over time. The rate of exponential divergence or convergence can be measured by calculating Lyapunov exponents. If one Lyapunov exponent in a dynamical system is positive, the system is often chaotic [WSSV85]. This is intuitive. Given a positive Lyapunov exponent, two initially close states diverge at an exponential rate. This makes long-term predictions very difficult. Additionally, the analysis of a dynamical system may involve basic statistics of system orbits. This includes mean states, maximum and minimum state values, and wave numbers. Wave numbers measure the relative amount of local minima of an orbit in a given time duration [Scho4].

2.2. Numerical analysis

Many dynamical systems are too complex to perform the above-mentioned analysis methods analytically. In this case, numerical analysis is performed.

Numerical analysis in the context of this work can be summarized as a sequence of scans. In a scan, for each point in a set of scan points, an orbit of a given length is calculated and analysis is performed on the orbit's states. A scan-point is a tuple containing an initial state and system parameters that remain fixed while calculating the corresponding orbit. Often the limit or asymptotic behavior of a dynamical system is to be analyzed. In numerical analysis, the limit behavior of an orbit is approximated by discarding several transient iterations. Transient iterations are the first n system function iterations of the initial state.

If the scan-points of a scan cover the combined state and parameter space of a dynamical system densely and the number of transient iterations is sufficient, the limit sets of a dynamical system and their basins of attractions can be approximated. Creating such a map of a dynamical system is usually computationally expensive. Let the number of dimensions of the combined state and parameter space be d . If every dimension of the state and parameter space is sampled by n orbits with m iterations, the total number of calculations is lower bounded by $n^d \cdot m$. This illustrates the need for high-performance computing and parallelization in the numerical analysis of dynamical systems.

3. The AnT Project

AnT is a simulation and analysis package for dynamical systems [Scho4]. It has been developed at the IPVS to support researchers and teachers. The current version, AnT 4.669, has been developed based on a set of requirements listed below:

- Modern software concepts
- Reuse-ability
- Support of rapid development, maintenance, and advancement
- Open source
- Scientific computing
- Distributed computing

The requirements have been implemented using object-orientated C++, a modular code structure connected by interfaces, separation of data recording and processing, and a client-server architecture to enable distributed computing. Additionally, CASE-Tools have been used. This includes a computer-based build process with GNU Autotools [Cal19], a computer-based documentation with Doxygen [Lar11] and computer-based versioning.

AnT supports a wide range of dynamical systems including discrete-time and continuous-time systems. Many system types supported by AnT extend the definition of dynamical systems that is used in the context of this work. Discrete-time systems supported by AnT include:

- Ordinary maps
- Coupled map lattices
- Recurrent maps

Continuous-time systems supported by AnT include:

- Ordinary Differential Equations
- Coupled ordinary differential equation lattices
- Partial differential equations
- Delay differential equations
- Multi-delay differential equations
- Functional differential equations
- Stochastic dynamical systems

3.1. AnT scan architecture

AnT supports four types of scans: one-dimensional scans, two-dimensional scans, user defined scans, and scan sequences. In AnT, a one-dimensional scan samples a line in the state-parameter space. Given a start and step size equation, the line is either sampled with uniform step size or logarithmically increasing or decreasing step size. There is an option to only use integer-valued step sizes. A real two-dimensional scan samples an axis-aligned plane in the state parameter space. Each dimension is sampled with an individual fixed step size. This is equivalent to a rectangular grid. Alternatively, an elliptical scan can be performed. A two-dimensional elliptical scan samples points on an ellipse in the plane. In a user-defined scan, a sequence of scan points is provided by the user. This allows sampling arbitrary shapes in the state parameter space at the expense of additional work for the user [Scho4]. Additionally, scan sequences can be defined. For example, a one-dimensional scan iterates over a sequence of nested one-dimensional scans. Both scan sequences and user-defined scans allow for scans of arbitrary dimensions.

Every scan results in a set of scan points, for which orbits have to be calculated and evaluated. The basic execution pattern used in AnT is called the machine concept. A machine works in three phases: Initialization of the task, processing the task as long as necessary in a cyclic manner, and finishing the task. The three phases are called pre, during, and post. In AnT the scan of a dynamical system is performed by two instances of the machine concept, a ScanMachine and an IterMachine. In the pre-phase, the ScanMachine initializes the scan and loads data specific to analysis methods.

In the during-phase, an IterMachine is launched. Additionally, analysis methods that operate on an inter-trajectory basis are executed. The during-phase is repeated for every scan point. In the post-phase analysis methods are processed, which aggregate data over the entire scan. In the pre-phase of the IterMachine, the trajectory and analysis methods that operate independently per trajectory are initialized. In the during-phase, the trajectory is calculated and the selected analysis methods are computed. This boils down to iterating the map or integrating the differential equation step by step using numerical integration. The during phase is repeated until the specified number of iterations has been reached [Scho4]. In the post-phase of the IterMachine, the analysis results are saved and processed. The scan architecture of AnT is illustrated in Figure 3.1.

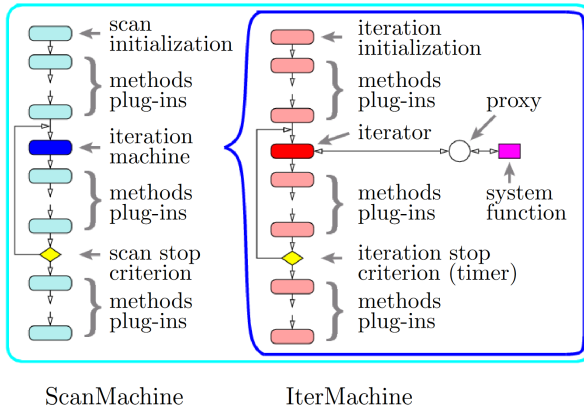


Figure 3.1.: AnT scan architecture with a schematic of the computation flow in the ScanMachine and the IterMachine.

3.2. AnT dynamical system analysis

Many important analysis methods and analysis concepts used in the analysis of dynamical systems have been introduced in chapter 2. First, an overview of the analysis methods implemented in AnT is given. Every method is

briefly described. Second, it is explained how analysis concepts such as bifurcation analysis are realized in AnT using the scan architecture and the implemented analysis methods.

3.2.1. AnT analysis methods

The following analysis methods are implemented in AnT:

- General trajectory evaluations
- Period analysis
- Region analysis
- Lyapunov exponents analysis
- Dimension analysis
- Frequency analysis
- Singular value analysis
- Check for conditions
- Symbolic sequence analysis
- Symbolic image analysis
- Generalized Poincaré sections
- Bandcounting

General trajectory evaluations Generalized trajectory evaluations include the saving of orbits or parts of orbits and calculating basic statistics of orbits. In basic statistics mean, minimum states, maximum states, and wave numbers are calculated.

Period analysis Period analysis investigates the asymptotical behavior of an orbit. Given a predefined compare precision, it is determined if the limit set of an orbit is a fixed point, a periodic set of points, or non-periodic. Additionally, an orbit is tested for divergence to infinity or negative infinity.

Region analysis Region analysis is not an independent analysis method. Before region analysis can be applied, period analysis has to be performed. A region of the same limit set is detected by period analysis marked by region analysis. Every region has a unique asymptotic behavior.

Lyapunov exponents analysis Lyapunov exponent analysis calculates one or multiple Lyapunov exponents of a given system. This helps to determine whether a system has chaotic behavior.

Dimension analysis Dimension analysis calculates characteristic quantities of an attractor. The calculated quantities are Kolmogorov–Sinai metric, capacity dimension, information dimension, and correlation dimension. Each quantity can reveal information about the formation of an attractor. The dimension analysis is based on a box-counting approach that calculates the spatial density of an attractor. Therefore density analysis as described in chapter 2 is a part of region analysis in AnT.

Frequency analysis Frequency analysis uses the external library FFTW to calculate the fast Fourier transform of an orbit. This allows computing the power spectrum and autocorrelation function of a given orbit.

Singular value analysis Singular value analysis computes a singular value decomposition of an orbit and is used to perform a principal component analysis. This allows the identification of the major spatial axis of a given orbit. The calculation is performed using the external library CLAPACK.

Check for conditions Check for conditions is hardly an analysis method. As soon as one of the implemented conditions is fulfilled the orbit length of the respective orbit is saved. The implemented conditions are: The orbit reaches a fixed point, the orbit reaches a given point, the orbit diverges from a given area, and the orbit does not diverge from a given area.

Symbolic sequence analysis In symbolic sequence analysis, the user defines a partition of the state space. A simple partition is a hyperplane, dividing the state space into two subspaces. Each partition is associated with a symbol from an alphabet. Given a simulated orbit, instead of saving individual states as vectors, the symbol of the partition in which the state is contained

is saved. This analysis represents an orbit as a symbolic string which is often more accessible to a scientist interpreting the simulation data.

Symbolic image analysis In symbolic image analysis, a directed graph is constructed which represents the structure of the state space. The main idea is to divide the state space iteratively into subspaces that form the nodes of the graph. This allows locating basins of attraction.

Generalized Poincaré sections Poincaré sections reduce the complexity of the state space. In its simplest form, the intersections of continuous orbits with a lower-dimensional hyperplane in the state space are computed. This results in a lower-dimensional sectional image of the orbits in a dynamical system [Scho4]. Often these are more accessible to humans.

Bandcounting Bandcounting has been added to AnT in later versions [Ecko6]. Bandcounting counts the clusters in the density distribution of an orbit. A cluster is a connected region of non-zero density in the state space that is enclosed by a region with zero density.

3.2.2. Performing analysis in AnT

A typical problem in the analysis of dynamical systems is bifurcation analysis. A common first approach to this problem is drawing a bifurcation diagram. In a bifurcation diagram, the limit sets of orbits with a fixed initial state are plotted in dependence on the system parameters. Given a one-dimensional system with one parameter, the bifurcation diagram is two-dimensional. The system parameter is the independent variable and increased over the a -axis. To every system parameter value, the limit set of the corresponding orbit is associated. The logistical map is defined as:

$$x_{n+1} = a(1 - x_n)x_n$$

where $a \in [0,4]$ is the system parameter, $x \in [0,1]$ is the state in space. Exemplary bifurcation analysis on the logistical map is performed using a one-dimensional linear scan over $a \in [1.5, 4]$ with 1000 scan-points and $x_0 = 0.5$. For each parameter value, the limit set of the corresponding orbit is plotted in the state space. If no finite limit set is detected, the last 64 states are plotted. This is illustrated in 3.2. Chaotic limit sets can be observed for $a > 3.6$. From $a = 3$ onward, a cascade of period-doubling bifurcations is

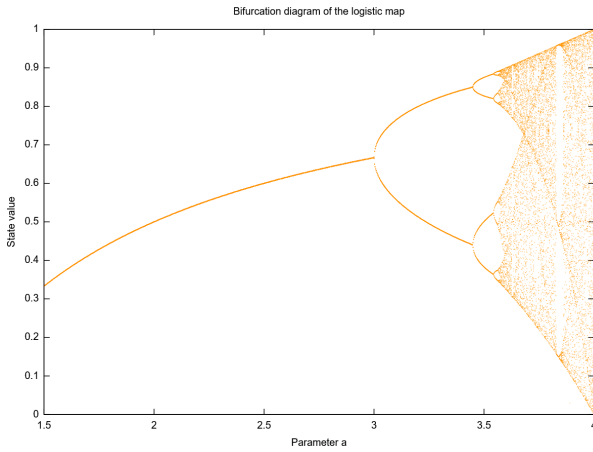


Figure 3.2.: Bifurcation diagram of the logistic map. At $a = 3$ the first period doubling bifurcation is located.

observed.

4. GPU Computing

In the last 10 years, general-purpose GPU computing became widely spread and used in the scientific community. Starting with parallel matrix multiplication and solvers for differential equations, scientific GPU computing evolved into a rich domain including physics and weather simulations, training for artificial intelligence and scientific visualization [Buc10] [Nvia]. Today, graphics cards have unprecedented parallel computing capabilities. Modern high-end devices like the NVIDIA RTX 3090 Ti card feature over 10000 computation cores with a combined performance of 40 TFLOPs [tec]. Given recent high-end desktop CPUs like the Intel i9-9900K, a speedup of up to 400x in floating-point operations per second can be achieved using the GPU for computing [set]. Nowadays, there are two major vendors of GPU hardware, NVIDIA, and AMD. After years of fierce competition, graphics cards from both vendors are mostly identical in terms of hardware architecture and design specification [ZPL⁺11]. In this chapter, the GPU hardware architecture is explained with the state-of-the-art NVIDIA GA102 architecture [GA1] on which the NVIDIA RTX 3090 Ti card is based.

4.1. GPU hardware architecture

A good way to quickly understand the GPU architecture is to follow the thread execution. If a GPU program is started, a user-defined number of threads is launched. Each thread computes the same function and is identified by an id. Using the id, global memory can be accessed and the function execution can be locally manipulated. Threads are grouped in blocks. Each block is executed by a single stream multiprocessor (SM) on the GPU. An SM can execute multiple blocks in parallel or series. The GA102 architecture has 84 SMs. Every SM is subdivided into partitions. On each SM partition, a warp is executed. A warp is the smallest computational unit that is executed. A warp always consists of 32 threads. Threads in a warp share an instruction counter. To ensure that all cores of an SM compute at any given time, the block size should be equal to the number of SM partitions times 32. The

whole picture of an SM is given in Figure 4.1.

The presented SM of the GA102 architecture has four partitions, 128KB shared memory, Texture memory (TEX), and a ray-tracing core. The ray-tracing core is a relatively new addition and not included in older cards. Texture memory is read-only memory. Shared memory is a fast read/write memory that is shared between all threads of the block executed on the SM. Each SM partition has a small shared L₀ instruction cache, a 64KB register file, 16 FP₃₂ cores, 16 FP₃₂/INT₃₂ cores, a tensor core, and four load-store units (LD/ST) and four special function units (SFU). The register file can be seen as the local memory of each warp. Data is stored in registers and passed to the computation units. Given a warp size of 32 threads, each thread has access to an average of 2KB of local memory. The arithmetic capabilities of the FP₃₂ and INT₃₂ cores are discussed in depth in section 4.2. Tensor cores are relatively new and not included in older cards. Load store units are used to access global memory from a thread. Global memory is much larger than local or shared memory and is usually sized at 24 GB per card. Each load store unit has a bandwidth of typically 384bit in the GA102 architecture. Therefore one load store unit can serve multiple threads in parallel. Special function units compute functions like $\frac{1}{x}$, sin, cos, exp and log.

4.2. Strengths and weaknesses of GPUs computation

Given the GA102 architecture, bottlenecks can be identified by analyzing a warp execution. The following potential bottlenecks are investigated:

- Special functions
- Looping
- Branching
- Global memory access

In general, a GPU operates parallel on a single set of instructions with independent data. A CPU operates on independent instructions on a stream of data. Pipelining, caching, branch prediction, and out-of-order execution has very limited support on a GPU. In the GA102 architecture, one SM partition can perform 32 FP₃₂ operations per clock cycle, 16 INT₃₂ operations

per clock cycle, and four special function operations per clock cycle. Therefore extensive use of special functions can slow down program execution significantly. Fixed length looping is generally not a problem. All threads in a warp share an instruction counter and take the same data path through the compute and storage units of an SM partition. If the loop length is thread-dependant and varies in a warp, all different data paths have to be processed in series by all threads, because the instruction counter is shared. Threads that are not supposed to take a particular data path are masked during the path execution. The same applies to branching. If a branch affects all threads of a warp in the same way, no additional computation except for the condition check is to be expected. If branches are thread-dependent, divergent data paths cause additional computation effort. The efficiency of global memory access depends mainly on the bus width of the load storage units and the global memory response time. Memory access of multiple threads in parallel can be combined in one load storage unit to maximize bus width usage. This is called memory coalescing. It works best if the access indices of each thread form a single connected memory block that can be loaded in at once [mem].

A GPU program with minimal special function usage, fixed loop lengths, and thread-independent branching can perform thousands of floating-point operations in parallel and massively outperform any CPU. Finally, arithmetic operation throughput is investigated in detail. The GA102 architecture equals a CUDA compute capability of 8.6 in the NVIDIA RTX 3090 Ti card [com]. Per clock cycle and SM (each with four partitions), the number of operations that can be performed are given in table 4.1 [pro].

Brief Summary of good GPU computing Finally, a small guide on good GPU programming can be given. Local memory should be used as often as possible, thread dependent branching and thread dependant loop lengths should be avoided. Global memory access should be used only when necessary and coalesced to maximize bus width usage. Type conversions, special functions, and especially FP64 operations should be avoided.

Functions in a GPU kernel code are generally inlined by the compiler. Therefore, recursions and dynamical binding are not possible [oclc]. Current frameworks do not support object-oriented programming. Even if supported,

Operation	Operations per clock cycle
FP32 mul, add	128
FP64 mul, add	2
FP32 reciprocal, special functions	16
Int32 add, sub, mul, shift	64
Compare, min, max	64
32-bit and, or, xor	64
32-bit Type conversions	16

Table 4.1: Arithmetic throughput of a GPU SM for different datatypes and operations.

object-oriented programming creates a huge control flow overhead which massively increases both memory usage and execution time. Given small instruction caches, shared instruction pointers, and small local memory, even modern GPU hardware is likely unable to support object-oriented programming effectively.

4.3. GPU computing frameworks

Currently, there are three GPU computing frameworks with a significant spread in scientific application and industry, OpenACC, CUDA, and OpenCL. CUDA and OpenCL are explicit frameworks. The developer creates a GPU program that is compiled and then executed on the GPU. Memory usage, thread management, looping and branching, datatypes, and register usage are mostly managed by the developer [ocl]. OpenACC is an implicit framework. Regions in the source code are annotated with compiler directives. Acceleration is mostly limited to loops. The compiler then tries to automatically generate a separate GPU program that accelerates the annotated region using the GPU. The compiler directives are closely related to OpenMP which parallelizes program code on multiple CPU cores. The acceleration capabilities are limited by data locality and race conditions. These can be countered by annotating atomic operations and optimizing data flow in the host program [acc].

In both types of frameworks, two programs are created. A CPU program

that exchanges data with the GPU, starts GPU thread execution, and optionally performs additional pre or post-processing on the data that is sent to the GPU [ocla]. During the execution of the CPU program, multiple GPU programs can be executed on the GPU in series and many data transfers can take place. CUDA is only supported by NVIDIA cards. OpenACC has limited support for AMD cards and full support for NVIDIA cards. OpenCL is compatible with most graphics cards and has good compatibility with old cards [oclb].

OpenCL was chosen as the GPU computing framework for this project. OpenCL has the highest compatibility with GPU hardware as it is not vendor-specific and allows low-level programming of the GPU multiprocessors. This enables higher performance and problem-specific optimization can be performed by the developer.



Figure 4.1: A single SM multiprocessor of the GA102 architecture. The SM is a composition of four partitions and shared memory. Every partition has floating-point, integer, load-store, and special function units. Every SM partition has a shared instruction counter, a local register file, and a small shared instruction cache.

5. Goal and tasks of this thesis

The goal of this thesis is to accelerate and parallelize scans using the GPU. Acceleration should at least cover multi-dimensional linear and logarithmic scans of ordinary maps and ordinary differential equations. At least the following important analysis methods are to be included in AnTGPU:

- General trajectory evaluations
- Period analysis
- Lyapunov exponents analysis
- Density analysis (part of Dimension analysis in AnT)
- Bandcounting
- Symbolic sequence analysis

The required analysis methods vaguely outline the core functionality of AnTGPU. In preparation for a more detailed description of how the analysis methods are realized and how the GPU is integrated, more requirements are formulated. The following requirements are listed in descending order of importance:

- Fast execution, high performance
- Compatible with common GPU hardware
- Long life cycle
- Modular, scalable, and easy to extend
- User friendly
- Platform independent

The goal of this thesis is satisfied in a three-step process:

Step 1 First, the existing AnT project is surveyed. Scan execution, system function integration, and method execution are investigated on both the architecture and implementation levels. This includes the analysis of computation flow and data structures. The analysis results are used to determine if the existing AnT source code can be reused or modified to efficiently support GPU acceleration in the context of the above-stated requirements.

Step 2 Second, a new scan architecture, method execution architecture, and system function integration are developed. The architecture is built with hardware orientation and efficiency in mind. All analysis methods that are to be included are modified, optimized, or developed newly from scratch. Some methods are extended by new features, not previously present in AnT.

Step 3 Finally, the newly developed architecture and methods are implemented and tested. For every analysis method, at least one example scan is computed and the execution time is compared to AnT. The results are checked for correctness and the quality of the results is determined with data-specific error measures. The impact of 32Bit floating-point number accuracy on the scan results is investigated in depth.

Additionally, AnTGPU is used to compute scans on a system that is subject to recent research. *Nested closed invariant curves* [AZ19] are scanned with AnTGPU and the results are briefly discussed.

6. Assessing GPU acceleration of AnT

The computation of trajectories from different scan points is fully parallelizable as every trajectory is only dependent on the initial state and system parameters. In theory, all required analysis methods can be implemented to operate locally on a trajectory. Given hardware characteristics, this might not be the optimal choice for density analysis and bandcounting and the reasons will be discussed in chapter 7.

A scan is GPU accelerated by computing and analyzing multiple trajectories in parallel. First, a batch of scan points is distributed to the multiprocessors of the GPU. Second, for every scan-point, a given number of iterations is performed together with analysis methods. Additionally, the limitations and capabilities of the GPU hardware have to be addressed. The most important limitations are a small instruction cache, shared instruction counters, and small local memory for both program code and runtime variables. The most important capabilities are a large number of parallel threads and a high throughput of FP32 and INT32 operations.

To parallelize scanning in AnT, the scan execution of AnT is investigated. AnT can execute scans on a single computer in standalone mode or distributed using a client-server architecture [Scho4]. It was chosen to start the investigation with an analysis of the scan execution in the client-server mode because it is remotely similar to thread execution on the GPU. In GPU computing, computational tasks and data are distributed to the GPU processing cores by the CPU. In this scenario, the CPU takes the role of the Server, and the threads executed on the GPU serve as clients. At this point, it is already clear, that the object-based source code of AnT is incompatible with the OpenCL C specification [ocl]. Using OpenCL, the GPU has to be programmed using a C-subset language. Additionally, object orientation introduces an overhead that conflicts with the primary requirement of this project: performance. Additionally, valuable local memory is consumed by stacks. The main purpose of this investigation is to determine the effort of modifying the source code. This analysis serves as a basis to decide whether

a new project is set up or AnT is modified to allow for GPU accelerated scans.

The GPU compatibility of the AnT architecture is first analyzed based on an example. A small example from the client-side scan execution is presented below. The functions included in the example are from "ScanData.cpp" of the AnT project.

```
typedef list<AbstractScanItem*> seq_t;
seq_t sequence;

void
ScanItemSequence::netClientScanNext ()
{
    ioStreamFactory->commit();

    // fetch next scanpoint from server
    string* scanPoint = anpClient->getScanPoint ();

    set (*scanPoint);
}

ScanItemSequence::set (string& scanpoint)
{
    std::istringstream is (scanpoint.c_str ());

    for (seq_t::iterator i = sequence.begin ();
        i != sequence.end (); ++i)
    {
        (*i)->set (is);
    }
}

template<typename ITEM_TYPE>
void
BasicScanItem<ITEM_TYPE>::set (void)
{
    *objPtr = currentValue;
}

template<typename ITEM_TYPE>
void
TwoDimensionalScanItem<ITEM_TYPE>::set ()
{
    *objPtr1 = currentValue1;
    *objPtr2 = currentValue2;
}
```

}

The execution in this example starts with the client fetching a scan-point from the server. After that the `ScanItemSequence::set` method is called. In this method the double linked list sequence is traversed by the iterator `i`. For each `AbstractScanItem` in sequence, the `set` method is called. Depending on the instance `i` of `AbstractScanItem`, this results in a call of `BasicScanItem<ITEM_TYPE>::set` or `TwoDimensionalScanItem<ITEM_TYPE>::set`. Both `BasicScanItem` and `TwoDimensionalScanItem` are descendants of `AbstractScanItem`.

This small example includes dynamic binding, a multi-layer class hierarchy, dynamic memory, and templates. None of the above-stated features are supported by OpenCL C [ocl]. Even if dynamic memory in form of a linked list is replicated on the GPU, the memory overhead for pointers and the lack of random access render it highly inefficient in parallel processing on the GPU. The extensive class hierarchy causes a snowball effect if modifications are made in a node class. If a for example a function has to be removed from the class hierarchy, the function has to be removed from the parent class as well, if present. Removal of a function in the parent class affects other child classes, quickly propagating modifications. Additionally, functionalities from the parent classes have to be stripped from the class hierarchy to ensure the functionality of the node class function outside the class hierarchy. The execution of scans, iteration of orbits, and execution of analysis methods are all performed in classes of one big class tree. This class tree is the biggest in AnT with more than 100 classes. It implements the concept of machines and transitions which is fundamental to AnT. The root class is `AbstractTransition`. Due to the high level of connectedness in the `AbstractTransition` class tree of AnT, stripping a subset of methods from the class hierarchy comes with an effort comparable to the creation of an entirely new program.

Another problem is the integration of the system function into the scan. In AnT the system function is programmed by the user in an external file and compiled into a dynamically linked library. The library contains the system function and is loaded before the scan starts. This grants AnT access to a user-defined system function. Using dynamic linking to integrate the system function into the scan and analysis process is incompatible with GPU computing as there is no support for dynamic linking. Additionally,

in AnT methods write data to a file during scan execution [Scho4]. This helps to limit memory usage but is not applicable to the GPU. File access is only possible on the CPU and covers data transferred from the GPU. In theory, it is possible to transfer data for every scan. However, this creates a huge overhead and does not allow for any parallelization. It is much more reasonable to transfer data in larger batches, given the fact that modern GPUs have memory comparable to the RAM available to the CPU.

7. AnTGPU: Architecture development

Based on the previous analysis it was found that creating a new project and architecture with limited scope as defined in chapter 5 is less effort than modifying the entire architecture of AnT including the class and method hierarchy, the memory management, the system function integration, and the file management.

With OpenCL a low-level GPU computing framework has been chosen that is compatible with most graphics cards produced in the last ten years [ocl`b`]. Using OpenCL, the GPU is programmed in a C-style language [ocl`c`]. To ensure consistency between GPU and CPU code it has been decided to code the CPU part of the project according to the C99 standard. This comes with various advantages. First, C99 is an almost universally supported standard that can be expected to be supported for many years to come. Additionally, C code can be compiled on Windows and Linux with minimal effort. This addresses the platform independence requirement of the project. The main data types in scanning and analysis are FP32 for floating-point operations and INT32 for integer operations. The precision is reduced compared to AnT which used FP64 but has a much higher throughput on the GPU. It is argued that a 64 times increase in arithmetic throughput outweighs double precision. Precision is indeed important in the analysis of a dynamic system. Therefore the effect of FP32 precision on analysis results is evaluated in a comparison against double precision in chapter 9.2.1. One major feature that makes OpenCL highly practical in the context of this project is the runtime compiler. In OpenCL, GPU code can be compiled at runtime with code stored in RAM. This turned out to be a nucleus around which AnTGPU is developed. One major advantage is that the user no longer has to compile the system function. If small changes to the function are made, with a runtime compiler it is sufficient so simply save the text file rather than run a compilation each time. Additionally dynamically linked libraries compiled on different operating systems are not interchangeable. This problem disappears when using a runtime compiler. Only the source file is required which is identical on all platforms. Additionally, a runtime

compiler allows for an enormous performance increase. AnT uses very general analysis methods and interfaces that can be used for a wide variety of scans and system functions. AnTGPU creates a specific and unique analysis program for every scan based on the configuration. This allows for scan-specific optimization which is impossible in a general analysis program. The whole process of generating the entire analysis program at runtime is described in the section 7.3.

7.1. Computation flow

The execution of AnTGPU is centered around the execution of a single scan of a dynamical system. The system is given by a formatted source file and the scan is customized by a configuration file. The source file allows the user to define custom variables, a system function, and a symbolic function used in the symbolic analysis. The configuration file specifies the states or parameters over which a scan is performed and configures the analysis methods. The configuration file is based on key-value pairs. This allows to only include options that are needed in a particular scan, independent of the order of options. A more detailed description of the system function file and the configuration file is given in the user documentation of AnTGPU in chapter A. Scans of arbitrary dimensions covering any axis-aligned sub-space of the state-parameter space are possible. The abstract architecture of AnTGPU is linear and performs the following tasks sequentially:

1. Scan for available GPU hardware
2. Check the user-selected device
3. Load and parse the configuration file
4. Calculate configuration data
5. Initialize memory
6. Load and parse system function file
7. Create and compile the GPU program
8. Execute the scan on the GPU
9. Transfer data from the GPU

10. Perform CPU postprocessing

11. Generate output files

Each of the above-listed tasks is associated with a method in the AnTGPU source. Every method performs error detection on the computation performed by it and returns a unique error code. Every error that is detected comes with a printed error message. After all listed tasks are completed, CPU and GPU memory is cleared.

Scan for available GPU hardware The system is scanned for available GPU hardware and presents the user with a list of found hardware and hardware specification.

Check the user selected device After the user chose the device on which the scan is executed, the chosen device is checked for compatibility, availability, and sufficient hardware resources.

Load and parse the configuration file The configuration file is loaded, parsed, and checked. Some options have dependencies on other options. It is checked that the dependencies are satisfied. The parsed configuration data is stored in a global data structure of the program.

Calculate configuration data Additional data is computed from the configuration data. This includes the size of the scan space, state space, and parameter space, the indices of constant parameters which are to be replaced in the system function, and the indices of constant initial values. The replacement of constant values and its effect is discussed in detail in section 7.3. Additionally, the order of analysis methods is computed. Different analysis methods start their analysis at different times during orbit iteration. Therefore not all methods have to be active at a given time.

Initialize memory Memory is initialized both on the GPU and the CPU. This includes memory allocation and transfer of initial values. GPU memory is allocated and filled in thread order. This allows threads with sequential indices executed in a warp to coalesce memory operations. The strategy of coalescing is explained in chapter 4.2. Initial data is computed on the CPU in this method.

Load and parse system function file The system function is loaded and parsed. This includes resolving to define directives, extracting the system and symbolic function, and creating separate system functions used to compute side trajectories in the Lyapunov exponent analysis.

Create and compile the GPU program A GPU program is created that executes the scan specified in the configuration file. Only methods that are activated in the scan configuration are included in the GPU program. This is described in detail in section 7.3.

Execute the scan on the GPU The GPU program is executed. Execution is performed in batches of scan points. This serves two purposes. First, on systems with a single GPU, the GPU is already used to render the graphical user interface. If long computations are executed on the GPU the display driver may interrupt the execution to ensure the functionality of the user interface of the operating system. Driver configuration can resolve this issue but on a single GPU system, this will result in a frozen screen during the execution of the scan. Using batched execution, parallel execution is performed in chunks with time in between to process the display rendering. There are more sophisticated solutions to the display driver problem including a second GPU for display purposes [Nvic]. Second, batched execution allows for progress measurements. This helps to estimate the execution time of very large scans. The progress is logged after each batch.

Transfer data from the GPU The data computed in batches on the GPU is transferred from the GPU global memory to the CPU memory.

Perform CPU postprocessing CPU post-processing is performed. Currently, only bandcounting and symbolic analysis require CPU postprocessing. Some problems like clustering or sorting require extensive computation on data that is distributed among many threads. Instead of creating a massive synchronization bottleneck to compute those problems on the GPU in parallel, the respective analysis method is split. The easy parallelizable and mostly on local data operating part is computed on the GPU, intermediate data is transferred to the CPU and post-processed in series. This hybrid approach uses the strengths of GPU and CPU optimally in the context of a given system analysis method.

Generate output files Output files are generated to save the analysis results to permanent storage. Currently, two formats are supported. The first format is string-based and accessible to humans including metadata of the scan and it is compatible with Gnuplot for visualization. The second format is optimized for minimal storage usage and requires a special visualization tool. This is useful on large multidimensional scans, where raw binary data fills the memory close to its maximum capacity. If this data is written to a file in string format, the resulting file is often useless, as it is too large to be loaded into memory for visualization or manual analysis.

7.2. Abstract architecture

One of the main features of AnTGPU is parallel scan execution. Each trajectory is calculated and analyzed by a single GPU thread running in parallel on an SM of the GPU. The computation is performed in batches of threads. Every batch ensures that the number of threads in the batch is a multiple of the number of threads that can be executed in parallel on the GPU. This ensures that no SMs are left idling at any given time. The batch execution architecture is sequential and shares the pre, during, and post-phase with AnT. In the pre-phase, GPU memory is allocated and initialized with the trajectory initial states and system parameters. In the during-phase, the next batch of threads is configured. This includes passing memory addresses to write data on the GPU. Additionally, it is possible to transfer analysis results after one or several batches. This feature is not used yet and will be discussed in the outlook. Second, batches of trajectories are scanned on the GPU, while simultaneously writing analysis results to the GPU global memory. During a scan, as much computation as possible is performed in local memory to increase performance. After the last batch has been processed, the system enters the post-phase. In the post-phase, the GPU analysis data is transferred to the CPU and post-processed. The scan execution architecture is illustrated in figure 7.1.

7.3. Dynamic kernel generation

To increase performance, cross-platform capabilities, and user-friendliness, AnTGPU creates and compiles the GPU scan and analysis program, called GPU kernel during program execution. The GPU kernel is identical for every thread started on the GPU. In the case of AnTGPU, every trajectory

is computed and analyzed by the same GPU kernel. The benefits of runtime compilation to cross-platform capabilities and user-friendliness have already been addressed at the beginning of chapter 7. In this section, the great benefits of dynamic kernel generation using runtime compilation to computational performance are presented. A dynamical system with a two-dimensional state-space and ten parameters is given. A 100000 iterations long scan is performed over two system parameters, and the mean state and period of each scan point are computed. This means that during the scan the initial state and eight parameters remain constant. The initial state remains constant for each scan point and the parameters remain constant in every iteration. Using dynamic kernel generation, the constant values are inserted into the source code as constants. No local memory has to be used to hold constant variables during scan execution and no global memory has to be allocated to provide constant values to the GPU threads at trajectory execution startup. In the above examples, two analysis methods are performed during the trajectory iteration. In this example, the computation of the mean state starts at iteration 10000 and the computation of the period at iteration 99950. Using dynamic kernel generation the iteration of the trajectory is split into three parts in the GPU program. The first part iterates from 0 to 9999 with no analysis methods active. The second part iterates from 10000 to 99949 with only the mean state analysis active. The third part iterates from 99950 to 99999 with both the mean state and period analysis active. This removes the need to check at every iteration whether any implemented analysis method is active. Additionally, the source code is usually smaller, as only a subset of analysis methods is included in the program source. In the unlikely case that all analysis methods are activated for a scan and all analysis methods start at different iterations, the source code is longer than without dynamic program generation but still removes the need for condition checking.

Dependent on the system state space dimension, certain analysis methods work differently. One example is the calculation of all Lyapunov exponents of a system that performs regular Gram-Schmidt orthonormalizations on a set of vectors with dimensions equal to the state space of the analyzed system [WSSV85]. In a general analysis program, multiple loops are necessary to iterate over the dimensions of the state space vector. Using dynamical kernel generation, the loops are unrolled and unnecessary calculations skipped. This removes the need for additional loop variables and loop condition checking. The entire GPU kernel generated by AnTGPU includes only one loop variable which stores the index or time of the current iteration. It is used to iterate the trajectory and start analysis methods.

7.4. Method execution and cross-thread computation

During the iteration of a trajectory on a GPU, thread analysis methods are computed. In AnTGPU, every analysis method that is activated by the user starts computation in a user-defined iteration and runs until the iteration of the trajectory has reached the user-defined end. An exception is mean state, min state, max state, and wavenumber computation which share a common start iteration. A method execution start point $s[i]$ can be shared by multiple analysis methods.

7.4.1. Trajectory iteration architecture

Similar to the IterMachine of AnT, AnTGPU has an architecture that structures the computation performed on a single trajectory of a scan. The evaluation of a trajectory starts with a new thread that is executed on the GPU. In the beginning, initial data is transferred from the global memory of the GPU to the local memory of the thread. This limits the number of global memory accesses during the thread lifetime which increases performance. After that, for every method execution start point $s[i]$ an iteration block is inserted into the kernel. An iteration block iterates the system function and computes all analysis methods that start at $s[i]$ and before. After the last iteration block is executed, the analysis results are transferred from the thread-local memory to the global GPU memory. The architecture is given in figure 7.2.

7.4.2. Cross-thread computation

Both density analysis and Bandcounting operate on multiple trajectories in parallel. Given a one-dimensional example system, the state space density is a one-dimensional function that is approximated using a box count method on a one-dimensional array. All trajectories that share the same system parameters and differ only in their initial states write to the same density array in parallel. Without any additional measures, this causes race conditions and leads to incorrect results. The problem is addressed in AnTGPU using atomic operations. They provide a basic level of synchronization that is sufficient in this application. In the case of density analysis, the density counts are increased using the atomic increment function provided by OpenCL [ocl]. In the case of atomic increment, one memory value is incremented by a single thread. Until the increment is finished, other threads

must wait and cannot access the memory value being incremented. Due to the shared instruction counter in SM partitions a single waiting thread causes a slowdown. How serious is this bottleneck? Not as serious as it seems. The larger the array used to store the density counts and the higher the dimension of the state-space, the less likely it is for two threads running in parallel to both increment the same density count in the same iteration. It is assumed that in the above presented one-dimensional example the state-space is scanned with 500 scan-points and the density is estimated using 500 boxes. In the case of all 500 scan-points of the state space being iterated in parallel, the expected number of threads accessing a particular box is $\frac{1}{500} * 500 = 1$, assuming that the distribution of states is uniform. In reality, this is most likely not the case, especially in the case of periodic orbits. Despite this, the expected number of threads accessing a particular box is usually small enough to not impact the performance more than computation in other computationally expensive analysis methods like Lyapunov exponents. Asymptotically, for a periodic attractor with a period length of n the expected number of threads accessing a single bucket is $\frac{t}{n}$, with t being the number of threads. For a chaotic attractor, the expected value is usually lower as infinitely many states are part of the attractor set which leads to a more uniform distribution. All claims are made under the assumption that the box array is properly fitted to the attractor. This means that the box size and offset are chosen appropriately. As both density analysis and bandcounting use a box-counting grid to derive information about attractors, this evaluation applies to both methods equally. A detailed look into the workings of both methods is given in chapter 8.4. In summary, simple cross-thread computation and synchronization in the context of density analysis and Bandcounting is not a major bottleneck. In the case of parallel computation of the density distribution of an attractor, the performance benefits outweigh the costs significantly even though atomic operations are used. This theoretical result is strengthened by the performance comparisons in chapter 9.1.

7.5. CPU postprocessing

Most analysis methods in AnTGPU run independently of other threads on local data of a single GPU thread. In density analysis, a state distribution is approximated from multiple trajectories in parallel. This can cause data conflicts and the problem is solved using atomic operations which guarantee

the correct execution of small arithmetic operations. In Bandcounting and symbolic analysis, clustering of the density grid and sorting of symbolic strings turned out to be too expensive to parallelize on the GPU. In the case of clustering, the next cluster is dependent on the previous which makes this problem much more suitable for serial computing. The sorting of symbolic strings is parallelizable but requires many memory operations and unpredictable branching which is caused by the varying number of comparisons to complete the sorting. Different initial permutations take different data paths to complete the sorting. If two threads of the same SM partition branch at different times, clock cycles are wasted due to the shared instruction counter. This problem has been described in detail in chapter 4.2.

An easy solution to the above-mentioned problems is hybrid computing. Parallel computations are performed on the GPU, intermediate data is transferred to the CPU and post-processed. In this approach, the strengths of both systems are combined to solve a difficult problem efficiently. Splitting a method into a CPU and a GPU part requires serious reasoning. In this work, hybrid computation is only used if necessary. During the development of AnTGPU, it was planned to compute period analysis using hybrid computation. Later it turned out to be less efficient than a pure GPU computation. To compute the period in AnTGPU, it is checked if a reference state repeats later in the iteration of the trajectory. Following a repeating state, the number of iterations since the reference state is computed and written to global memory. Different trajectories can have different periods. It was assumed that asynchronous branching in the threads of one warp would cause serious delay. To solve this problem, hybrid computation is proposed. The trajectory is iterated on the GPU until the reference state and the reference state of each thread is transferred to the CPU. Then, for each thread, a few hundred iterations are computed on the CPU and the period length is determined. It turned out that the initial assumption of slow branching was false. Given a trajectory of 100000 states, period analysis is usually active in the last 500 states. Branching delays in this small fraction of the total computation are insignificant. Instead, the massive parallel computing power of the GPU allows for faster period length calculation even with branching delays. Additionally, performing iteration on the CPU requires knowledge of the system function not only to the GPU but also to the CPU. This requires a second runtime compilation for the CPU which causes additional overhead and increases the program complexity.

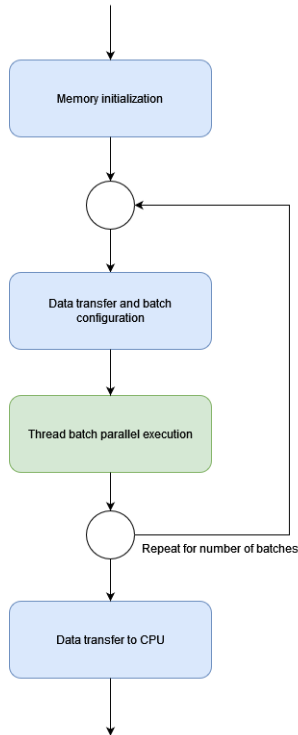


Figure 7.1.: Parallel scan execution architecture of AnTGPU. CPU operations are colored blue, and GPU operations are colored green. After an initialization phase, multiple batches of threads are started in series. Before a new batch is launched, data is transferred from and to the GPU and the batch is configured. As soon as all batches are computed, the results are transferred to the CPU.

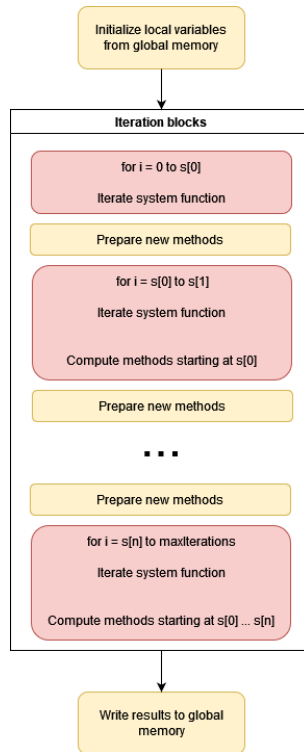


Figure 7.2.: Trajectory analysis architecture. Every trajectory is analyzed on a GPU thread. At startup, local memory is initialized. The thread initialization is followed by iteration blocks. Every iteration block iterates the system function for a predefined number of iterations and performs a predefined sequence of analysis methods. After the execution of the last iteration block, the analysis results are copied to global GPU memory.

8. AntGPU: Method development

The analysis methods implemented in AnTGPU are based on a selection of important methods from the original AnT project. Every method from AnT had to be adapted to the GPU hardware. This includes the usage of local memory and the removal of loops and branches. Many methods are optimized for higher performance or are extended by new useful features which are not present in AnT. Some AnTGPU methods share only the specification with their AnT counterpart. The same results are computed in a new way that is adapted to the hardware of the GPU and more efficient in terms of memory usage and instruction throughput. In the following sections, the functionality of each method is described briefly. After the short description, the implementation in AnT is presented and compared to the implementation in AnTGPU. Improvements and changes are explained and reasons for the changes are given. Key performance characteristics are data locality to ensure optimal parallel performance, low global memory usage, and instruction throughput. For many methods, the functionality has been reduced to the core functionality. Additional options present in AnT may be added later if they turn out to be important to the user. At this moment this is not the case. Finally, every method is demonstrated in at least one example. This includes introducing a dynamic system, formulating a scan, and discussing the scan results. Usually, system behavior in dependence on parameters or the structure of an attractor is investigated. Scans are described in brief textual form in this chapter. The complete AnTGPU configuration for each scan is added to the appendix of this work in chapter B.

8.1. General trajectory evaluation

General evaluation of a trajectory consists of saving states, calculating basic statistics such as the mean state, the component-wise minimum state, maximum state, and calculating wave numbers. Wave numbers count the number of local minima across each dimension of the state space along a trajectory.

Each count is then normalized with the trajectory length [Scho4].

The computation of the general evaluation is almost identical in AnT and AnTGPU, except for the actual implementation. In AnTGPU the last n states are saved into global GPU memory to be transferred to the CPU and saved. Independently, at the beginning of every trajectory evaluation, a minimum and maximum state variable are initialized with a very large and a very small value in the local memory of the thread. As soon as general evaluations are active, in every iteration the component-wise minima and maxima are updated. After the trajectory iteration is finished, both local minimum and maximum states are transferred to global GPU memory. The mean state is calculated identically. A local mean state is initialized with zero. As long as general evaluations are active, the current state vector is added to the mean state. After the trajectory iteration is finished, the mean state is divided by the number of iterations the method was active and the result is transferred to global memory.

The computational performance of all general evaluations is very good. Local operations, fully parallelizable computations, only additions and comparisons are computed. This allows for maximum throughput.

General trajectory evaluation is demonstrated on a modified logistic map, defined by the following system function:

$$x_{n+1} = a \sin(x_n)(1 - x_n^2)$$

A two-dimensional scan is computed in AnTGPU with $x \in [-3, 3]$ and $a \in [-2.7, 1.52]$. The resolution of the scan is 1000×1000 scan points. For every scan-point, a trajectory of 100000 states is computed, the first 10000 states of each orbit are transient. For every trajectory, the last state, maximum state, and minimum state are saved. The complete scan configuration is given in chapter B.1. In the visualization, the value range is set to $[-4, 3]$. The results are shown in figure 8.1.

For $a = -0.55$ it can be seen that the last state value of the trajectory is dependent on the initial state value. In pink regions, the last state value is -1 , in blue regions, it is smaller than -2 . It is therefore assumed, that at least two attractors are coexisting in the state space. This is confirmed by looking at the minimum and maximum values of the trajectories for $a = 0.55$. In the pink regions, the maximum and minimum values are equal to the value -1 . In the blue regions, the maximum value is approximately -2.3 .

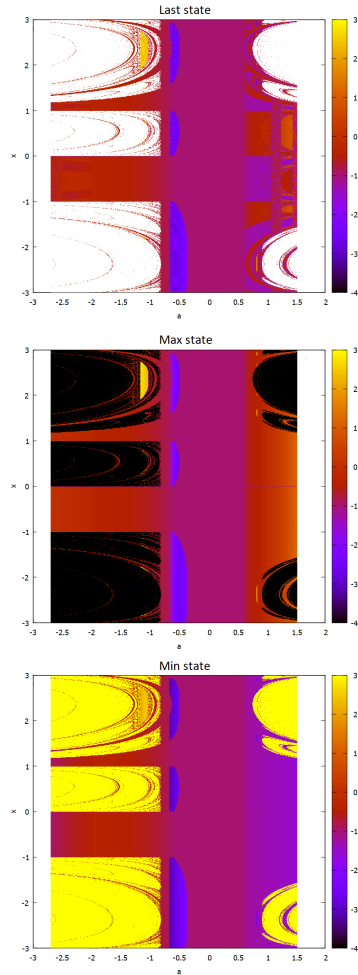


Figure 8.1.: In the top figure the last state of the scan is visualized. For $a = -0.55$ coexisting attractors are assumed. This is confirmed by the minimum and maximum states in the middle and bottom figures.

This result confirms that trajectories from the blue region do not intersect trajectories in the pink regions. Therefore at least two attractors coexist in the state space. Additionally, the minimum and maximum values in the pink regions are equal. This means that for all pink regions the attractor is the same fixpoint. In further work, the basin of attraction for the fixpoint attractor can be computed from the scan results and the nature of the other attractors can be investigated.

8.2. Period analysis

In period analysis it is determined if a trajectory converges to a periodic limit set. Asymptotically, a trajectory can converge to a periodic set, an infinite quasi-periodic set, or any other infinite set, often referred to as a chaotic attractor. A trajectory can also diverge to positive or negative infinity or an iteration can result in a non-resolvable calculation like a division by zero.

In AnT it is first checked if the last state of a trajectory is divergent. The check is performed by comparing the maximum norm of the last state with a threshold and additionally checking for NaN. Second, the last n states of the trajectory are compared to the last state of the trajectory. If the maximum norm of the difference between the last state and a previous state is smaller than a threshold, a period is found and the length of the period is stored. Although it is possible to store the last states in global memory in AnTGPU and implement the method accordingly, for large scans and long periods a lot of memory is occupied by the last states. Fortunately, this method can be improved to be more efficient in terms of computational performance and memory usage. If period checking up to a length n is performed in AnTGPU, the n -last state is saved as a reference in local thread memory. The reference state is then checked for divergence using the same approach used in AnT. In every following iteration, the reference state is compared to the current state using a threshold on the maximum norm of the difference between the current state and the reference state. If a period is found, the length is transferred to global memory and further comparison is stopped. In AnTGPU, the period length can be used to split the last state save file that is generated in general trajectory evaluations into two files. One file contains only states of periodic trajectories while the other contains only states of divergent or aperiodic trajectories.

This approach has two major advantages over the approach used in AnT. Instead of n last states, only two states have to be stored to check for a period length up to n . These two states are the current state of the trajectory which is already used in the system iteration and the reference state. Additionally, checks are performed alongside the trajectory iteration. No calculation is performed after the iteration of the trajectory stops.

Period analysis is demonstrated on the Gingerbreadman map. The Gingerbreadman map is a two-dimensional map, defined by the following system function [Dev88]:

$$\begin{cases} x_{n+1} = 1 - y_n + |x_n| \\ y_{n+1} = x_n \end{cases}$$

A two-dimensional scan is computed in AnTGPU with $x \in [-10, 10)$ and $y \in [-10, 10)$. The resolution of the scan is 1000×1000 scan points. For every scan-point, a trajectory of 100000 states is computed and period analysis is performed on the last 512 states. This allows for detecting periods with lengths up to 512 states. The complete scan configuration is given in chapter B.2. The scan result is illustrated in figure 8.2.

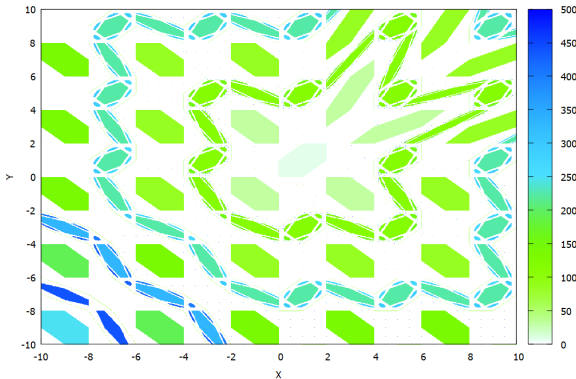


Figure 8.2.: Period length plot of the Gingerbreadman map. Multiple periodic attractors with different period lengths coexist in state space.

In the scan result, multiple polygons of different period lengths are visible.

They are mostly hexagonal and most likely are basins of attraction of periodic attractors. This result is a basis for further analysis. Good follow-up questions are: Are all periods in an area of equal period length equal? What happens in areas where no period was detected? These questions can be answered by computing more scans and using other analysis methods, like the minimum and maximum values, mean states, or saving the last states and comparing them. A clockwise rotation of 135 degrees of the image shows the gingerbread man upright.

8.3. Lyapunov exponents analysis

Lyapunov exponents quantify the convergence or divergence of two initially close trajectories. A system with at least one positive Lyapunov exponent is often a chaotic system. In a multi-dimensional system, there exists a Lyapunov exponent for every dimension.

In AnT the Lyapunov exponents of a system are computed according to Wolf [WSSV85]. Given a main trajectory t , an orthogonal base is spanned in the state space. Each vector of the orthogonal base has a small euclidean length, initially set to a constant epsilon. This represents the initial small deviations in state space. The sum of the current state t_n and a vector from the orthogonal base equals the initial state of a side trajectory s . There are as many side trajectories as the state space has dimensions. For a given number of iterations, the main trajectory and all side trajectories are iterated using the system function. After some iterations, the side trajectories diverge from or converge to the main trajectory which distorts the initially orthogonal base. After the required number of iterations is performed, the vector base spanned from the main trajectory to the side trajectories is orthogonalized, normalized, and rescaled with epsilon. Sequentially, starting with the first side trajectory, a Gram-Schmidt orthonormalization is performed. For each side trajectory, the distance d_i to the main trajectory is computed and the i -th Lyapunov exponent L_i is updated [WSSV85].

$$L_i += \log_2 \left(\frac{d_i}{\epsilon} \right)$$

At the end of the trajectory iteration, the Lyapunov exponents are divided by the number of iterations the Lyapunov calculation was active.

The above-explained approach was transferred mostly unchanged to AnTGPU. Equally to the main trajectory, the side trajectories are computed in local memory. Additionally, the Lyapunov exponents and some helper variables are stored in local memory. After the trajectory iteration is completed, the Lyapunov exponents are scaled and transferred to global GPU memory. Using dynamic kernel generation, the Gram-Schmidt reorthonormalization is implemented specifically for a given system. No loops iterating over the system dimensions are included. This slightly increases the computational performance. Gram-Schmidt reorthonormalization is computationally expensive as it has quadratic complexity in the state dimensionality and is susceptible to numerical errors [For15]. Many special function operations are needed to process multiple divisions and logarithms. Fortunately, the calculation of Lyapunov exponents is fully parallelizable and all computations can be performed in local memory. This gives a huge performance boost to this expensive analysis method. If problems related to numerical stability turn out to be significant during the program life-cycle, the modified Gram-Schmidt process can be implemented, replacing the classical Gram-Schmidt process used in the computation of Lyapunov exponents.

The calculation of the Lyapunov spectrum is demonstrated on the Hénon map. The Hénon map is a two-dimensional map with two parameters, $a > 0$ and $b \in [0,1]$ [Hén76]. The system function is defined as:

$$\begin{cases} x_{n+1} = 1 + y_n - ax_n^2 \\ y_{n+1} = bx_n \end{cases}$$

A two-dimensional scan is computed in AnTGPU with $a \in [0,2)$ and $b \in [0,1)$. The resolution of the scan is 1000×1000 scan points. The initial state is fixed to $(0.25, 0.25)$. For every scan-point, a trajectory of 10000 states is computed and both Lyapunov exponents are calculated after 10000 transient iterations. The complete scan configuration is given in chapter B.3. The Lyapunov spectrum of the Hénon map is illustrated in figure 8.3.

In white areas the map is divergent and no Lyapunov exponents can be calculated. Positive Lyapunov exponents are plotted with a yellow palette, negative exponents with a green palette. Areas with a positive Lyapunov exponent are easily spotted. Positive Lyapunov exponents are necessary for chaotic behavior. The transition from non-chaotic to chaotic behavior is visible. In regions of positive Lyapunov exponent values, areas of stability with negative Lyapunov exponent values exist. These areas have the shape of a swallow and are called Milnor's swallows [Ber18].

8.4. Density analysis

Density analysis is used to map an attractor in state space. Once a trajectory converged to an attractor, every iteration moves the current state through the attractor. A hypergrid is spanned in the state space, covering the estimated area of the attractor. For every grid cell, the number of states that are in the grid cell is counted during the iteration of the trajectory. This results in an unnormalized density distribution which approximates not only the shape of the attractor but also gives insight into the most frequently visited areas of the attractor.

In AnT, density analysis is performed with an individual density grid for every scan point. To compute a good asymptotic approximation of the attractor density, usual tenths to hundreds of billions of iterations are necessary. This is extremely time-consuming. Therefore, in AnT usually, only one scan point is evaluated in density analysis which converges to an attractor and subsequently maps out the attractor structure in the state space.

In AnTGPU computing a single trajectory on its own is pointless as it opposes the very idea of parallelization. Even if multiple trajectories are computed in parallel, having one density grid per trajectory is not feasible. A two-dimensional Grid can easily reach the size of four million grid cells. The density is stored in a UIN32 array to maximize GPU instruction throughput. Assuming that one thousand trajectories are computed in parallel, already sixteen billion bytes of memory are needed. Average modern GPUs have around six billion bytes of global memory, which means that this relatively small two-dimensional example is already out of reach for common GPU and CPU memory sizes. One- and two-dimensional density grids are very common, as they are relatively easy to visualize. Additionally, iterating a particular trajectory hundreds of billion times on a GPU core is as fast if not slower than iterating the same trajectory on the CPU. In many cases, the density of an attractor is mapped only for a small number of system parameters which means that only a few trajectories have to be computed. It is therefore essential to parallelize the density estimation of a single scan point. Given fixed system parameters, multiple hundred or thousand trajectories are started in the state space close to the initial scan point. This creates a bunch of trajectories in which each trajectory reaches the attractor of interest in a slightly different path. Once the trajectory bunch reached the attractor and individual trajectories start to diverge, they rapidly map out

the attractor. In this case, every trajectory of a bunch writes to a common density hypergrid. Only a fraction of the iterations per trajectory that is necessary when mapping an attractor with a single trajectory is needed if a trajectory bunch is used. Each trajectory of the bunch is iterated in parallel and maps different parts of the attractor in the shared density grid. This approach is much faster than density estimation with a single trajectory and fully uses the parallelization of the GPU. To synchronize the threads writing to a shared density grid atomic operations are used. Each grid cell counts states using a `UINT32`. Overflow protection is added to prevent a full cell from overflowing and invalidating the result. The above presented new parallel approach assumes that trajectories of a bunch diverge sufficiently in an attractor. Given chaotic attractors which are mostly analyzed this is usually the case, especially if positive Lyapunov exponents are present which are one of the characteristics of a chaotic system.

Given different system parameters, an attractor might have different shapes and asymptotical state distributions. In AnT, multiple bunches of trajectories can be calculated in parallel and in series to map out attractors at different locations in the state and parameter space. For high dimensional state spaces the density distribution is no longer visualizable. Additionally, high-dimensional and high-resolution grids require enormous amounts of memory. To address this problem, AnTGPU allows projecting the states of high dimensional state spaces to an arbitrary axis-aligned hyperplane in the state space. This is equivalent to an aggregation of the full-dimensional density distribution across one or several dimensions. The density distribution in this hyperplane is called a slice. A slice is much easier to visualize and requires considerably less memory while still capturing important structures. Multiple slices from different angles can help reconstruct and visualize the structure of high-dimensional attractors in lower dimensions.

Density analysis is demonstrated on the tent map. The tent map is a one-dimensional map with one parameter $a \in [0,2]$ [HbL]. The system function is defined as:

$$x_{n+1} = \begin{cases} ax_n & x_n < \frac{1}{2} \\ a(1 - x_n) & \textit{otherwise} \end{cases}$$

The tent map is bounded in the interval $x \in [0,1]$ for all values of a in the bounds given above. For every value of a there exists a global

attractor that is attractive to almost all initial conditions. Exceptions are the fixpoints of the map and their preimages. To map out the density of the attractor for a given parameter, a trajectory bundle of 350000 uniformly spaced initial values was iterated 2000 times per trajectory. The first 1000 values were discarded. This procedure was repeated 1000 times to plot the density development of the global attractor for $a \in [1.1, 2]$. In total $1,000 \cdot 350,000 \cdot 2,000 = 700,000,000,000$ iterations are computed. The complete scan configuration is given in chapter B.5. The resulting density distribution is illustrated in figure 8.4.

The density distribution is scaled logarithmically to the basis e . The merging of the two main bands of the global attractor at $a = \sqrt{2}$ is visible. After the merging, a higher density is visibly in areas where the previously separated bands overlap.

8.5. Bandcounting

The state set of a chaotic attractor can be a single connected region of the state space or the union of multiple disjoint regions in state space. The number of connected state-space regions that are part of an attractor is called Bandcount. For arbitrary system functions, numerical Bandcounting is very similar to density analysis. Given a density distribution, the number of connected regions of grid cells with cell value greater than zero approximates the Bandcount of the attractor. The Bandcount of an attractor is computed using the density grid. Therefore, numerical Bandcounting is an aggregation of the spatial density distribution. Finding and counting clusters of grid cells with a value greater than zero is difficult to parallelize as for every given grid cell information about its neighbors is necessary to determine connectedness. Additionally, previously visited clusters have to be marked as visited to avoid double counting. Therefore it was chosen to compute the Bandcount on the CPU based on the density grid data computed on the GPU. The approach chosen in AntGPU is based on a flooding algorithm. First, an empty floodmap is initialized which is used to mark visited cells. After that, the following procedure repeats until no more unvisited cells with a value greater than zero exist.

```
while unvisited cells with value > 0 exist:
    //search cluster
    Find unvisited cell with value > 0 and mark as visited
```

```

//flood cluster, cells to be visited are stored in a queue
Given above cell with value > 0:
    visit all currently unvisited neighbors with value > 0
    mark those neighbors as visited

//increase clustercount
bandcount++

```

While the above-presented approach works well, computing the Bancount for millions of system parameter values requires millions of density matrices to be stored and later processed which uses a gigantic amount of memory. This problem is unique to Bandcounting, where the density matrix of a trajectory bunch is aggregated into a single number. This allows a serial processor to use a single density matrix and aggregate the data after every trajectory evaluation. In theory, it is possible to compute the density matrices for multiple parameter sets in batches on the GPU and perform Bandcounting on the batch data on the CPU to aggregate the data as soon as the GPU and CPU memory is exhausted. This approach requires much tighter control of thread execution order and batch sizes and is discussed in the outlook of this work. To this point, a different solution to this problem is implemented. Distinguishing between cells with a value greater than zero and cells with the value zero is a binary decision. The actual value of a cell is not important to Bandcounting. Therefore Bandcounting uses a different grid in which each grid cell holds a binary value that indicates if this cell has been visited at least once by a trajectory. This approach reduces the memory usage by a factor of 32 compared to a regular grid cell with a `UINT32` value storing the count of states. In the AnTGPU implementation, `INT32` datatypes are still used to maximize instruction throughput. Binary operations such as `bit shift` and `or` are used store 32 grid cells in a single unsigned integer. This combines great performance with much lower memory consumption.

Bandcounting is demonstrated on the following one-dimensional piecewise-linear discontinuous map with two system parameters [ASo8]:

$$x_{n+1} = \begin{cases} ax_n + u + 1 & x_n < 0 \\ ax_n + u - 1 & \text{otherwise} \end{cases}$$

A two-dimensional scan is computed in AnTGPU with $a \in [1, 2)$ and $u \in [-1, 1)$. The resolution of the scan is 1000×1000 scan points. The initial state is fixed to $x_0 = 0.25$. For every scan-point, a trajectory of 1000000 states is computed and Bandcounting is calculated after 20000 transient iterations. The complete scan configuration is given in chapter B.6. For $a \in [1, 2)$ the

map has chaotic attractors. The Bandcount of each attractor is illustrated in figure 8.5.

Divergent trajectories are plotted in white. A fractal pattern of increasing Bandcounts is visible. The close a is to one, the more attractors with a high number of bands occur. For $a \in [0,1)$ the map converges to periodic orbits. Therefore a transition between periodic and chaotic behavior is observable at $a = 1$. The periodic interval $a \in [0,1)$ is illustrated in figure 8.6. The period was computed with the same scan setting as above, only with a varied range for a and using period analysis instead of Bandcounting. The number of system function iterations has been reduced to 100000. The complete scan configuration is given in chapter B.7. It can be observed that a low period attractor transforms into a low bandcount chaotic attractor at $a = 1$.

8.6. Symbolic sequence analysis

Symbolic sequence analysis is well described in the words of Hao [Hao91].

One divided the phase space into a finite number of partitions and labels each partition with a symbol (a letter from some alphabet). Instead of representing the trajectories by infinite sequences of numbers - iterates from a discrete map or sampled points along the trajectories of a continuous flow - one watches the alternation of symbols. In so doing, one loses a great amount of detailed information, but some of the invariants, robust properties of the dynamics may be kept...

The phase space is called state space in this work. In AnT, the user can define an arbitrary symbolic function that converts a state to a symbol. In AntGPU this is equally possible. It is possible to store the n last symbols of a trajectory in a character array. Symbols are restricted to the values of the datatype `char`. Additionally, a new feature has been added to AntGPU. Multiple periodic symbolic sequences generated from the same periodic attractor can differ in shift. A simple example is the sequence AB and BA. Both symbolic sequences can arise from the same two-state periodic attractor, depending on the state at the beginning of recording the symbolic sequence. The shift is very confusing to the user, as equal periods are not visible at the first glance. In AntGPU, periods can be sorted by shift. This ensures, that equal periods are represented by the same symbolic string with the same shift. Shift sorting is performed on the CPU. The symbolic strings

calculated on the GPU serve as intermediate data and are shift-sorted in the CPU post-processing function if requested by the user. Shift-sorting shifts a symbolic string to the right until the most sorted shift in ascending order is found. A simple example is the string “291”. The three possible shifts are “129”, “291” and “912”. “129” is the most sorted shift in ascending order starting from the left. This method always computes a specific shift amongst all possible shifts of a symbolic string. This allows the user to spot equal periods at the first glance.

Symbolic sequence analysis is demonstrated on the piecewise-linear system introduced in section 8.5. A simple L/R symbolic separation of the state space is chosen. In a one-dimensional L/R separation with a threshold, all states below the threshold are mapped to the symbol L and all states above the threshold are mapped to the symbol R. For the given piecewise-linear system the following symbolic function is chosen:

$$\text{sym}(x_n) = \begin{cases} L & x_n < 0 \\ R & \text{otherwise} \end{cases}$$

A system function file of the piecewise-linear map with the above-stated symbolic function is given in chapter C.1. Symbolic strings are hard to visualize in an image. Therefore rotation numbers are computed and visualized. Given a trajectory with a periodic limit set and an L/R symbol separation, the rotation number is the relative amount of the symbol L in one period of the symbolic sequence. The piecewise-linear system is scanned in the parameter space with $a \in [0,1)$ and $u \in [-1,1)$ and 1000x1000 scan-points using symbolic analysis and period analysis and 256 last states. The initial state is set to $x_0 = 0.25$. The complete scan configuration is given in chapter B.8. The rotation numbers of the piecewise-linear map are illustrated in figure 8.7.

When varying u for a fixed a , a devil’s staircase, also known as the Cantor function can be observed [Kee80].

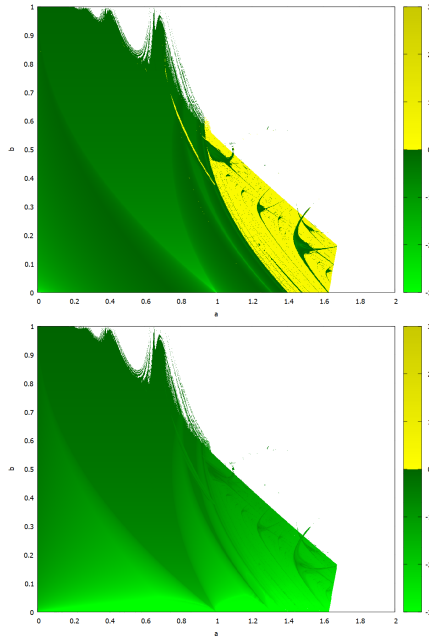


Figure 8.3.: The Lyapunov spectrum of the Hénon map. The first Lyapunov exponent is illustrated in the top figure, and the second Lyapunov exponent is illustrated in the bottom figures. At $a = 1.5$, $b = 0.2$ in the top figure, a non-chaotic region embedded in a chaotic region called Milnor's swallow is observed. Chaotic regions are colored yellow, divergent regions are colored white.

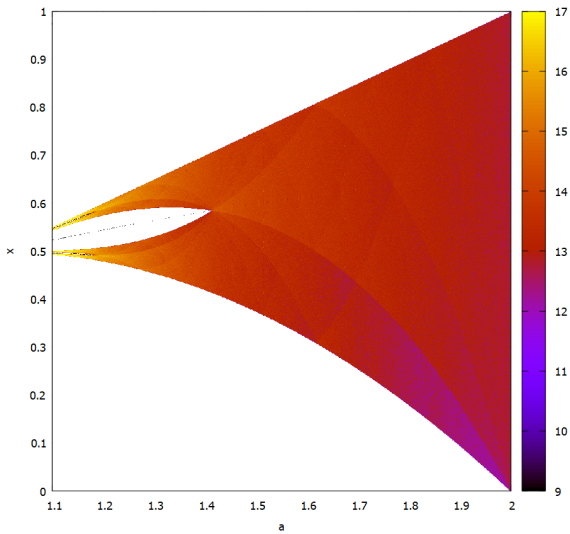


Figure 8.4.: Attractor density estimation of the chaotic attractor in the tent system. At $a = \sqrt{2}$ the two main bands of the attractor merge and overlap. The density is plotted logarithmically.

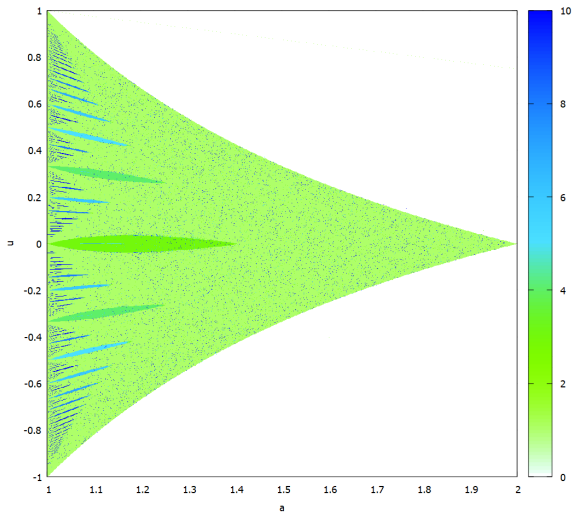


Figure 8.5.: Chaotic attractor band counts of a piecewise-linear system in parameter space. For a close to 1, a fractal pattern of increasing band counts is observed. Divergent trajectories are plotted in white.

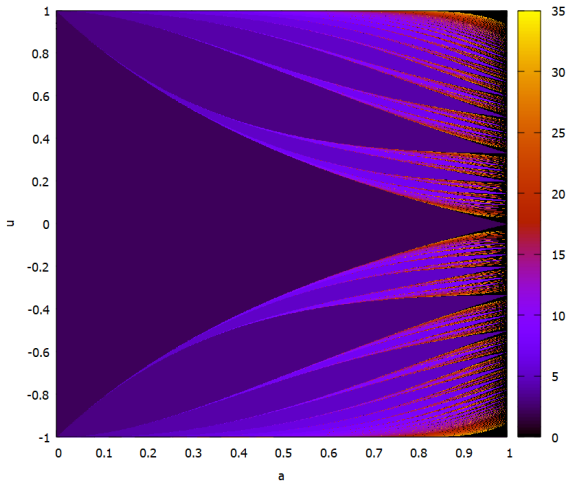


Figure 8.6.: Periodic attractor period lengths of a piecewise-linear system in parameter space. For a close to 1, a fractal pattern of increasing period lengths is observed.

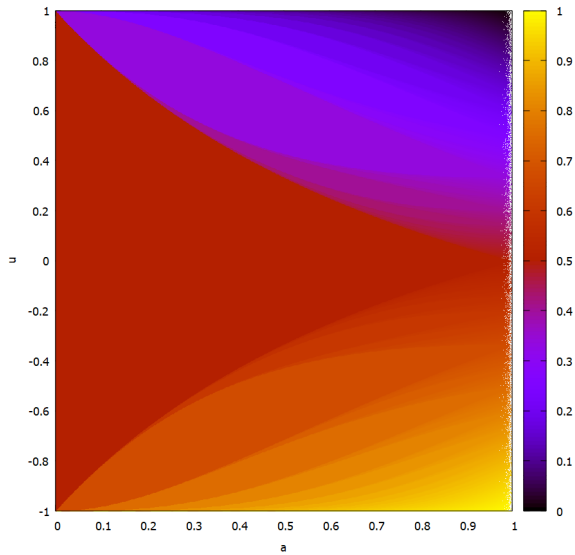


Figure 8.7.: Rotation numbers of the periods in a piecewise-linear system. If u is varied for a fixed a , a Cantor function is observed.

9. Testing and evaluation of AnTGPU

Archiving a significant performance increase over AnT was one of the main motivations to create AnTGPU. In this chapter, the execution time of every analysis method is measured. For every method, a sample system is analyzed with practical scan settings. An example of a practical scan setting for a two-dimensional scan is 1000x1000 scan points as the resulting data matrix can easily be visualized with good resolution. The example scans of chapter 8 will be reused in this chapter. Additionally, the Rössler system is introduced and scanned. First, for each scan, the execution time is measured using both programs. Second, selected methods are tested for accuracy using an appropriate metric. This quantifies the impact of the reduced floating-point accuracy on the analysis result.

No continuous system has been introduced yet. The Rössler system is a three-dimensional system of autonomous ordinary differential equations with three parameters. Currently, only discrete systems in the form of ordinary maps have been presented. AnTGPU is designed to support both ordinary maps and ordinary differential equations. The Rössler system is defined by the following set of equations:

$$\begin{cases} \frac{dx}{dt} = -y - z \\ \frac{dy}{dt} = -x + ay \\ \frac{dz}{dt} = b + z(x - c) \end{cases}$$

(x,y,z) form the three dimensional state-space, (a,b,c) form the three-dimensional parameter space. At the parameter setting $(a = 0.2, b = 0.2, c = 5.7)$ a chaotic attractor exists [Rös76]. The Rössler system is a continuous dynamical system. To compute a trajectory from a set of differential equations, numerical integration is used. Numerical integration is performed with Runge-Kutta 4 and an appropriate stepsize [WH96].

A scan is computed with AnTGPU to calculate the Lyapunov spectrum of the Rössler system. The initial state is fixed to $(0,0,0)$ and the parameters

are chosen $a = 0.15$, $b = 0.2$, $c \in [3,12]$. c is sampled with 1000 scan points. For each scan point, a trajectory of 300000 states is computed using Runge-Kutta 4 numerical integration with a stepsize of $h = 0.01$. All three Lyapunov exponents are computed. The complete scan configuration is given in chapter B.4. The system function is given in chapter C.2. The first two Lyapunov exponents are illustrated in figure 9.1.

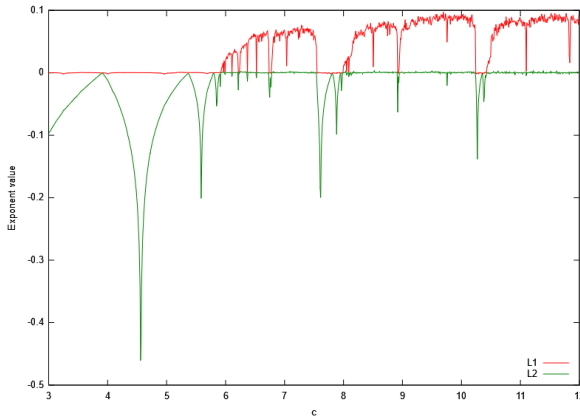


Figure 9.1.: The first two Lyapunov exponents of the Rössler system. The first Lyapunov exponent is plotted in red, the second Lyapunov exponent is plotted in green. At $a \approx 5.7$ the first Lyapunov exponent becomes positive and chaotic behavior appears.

The third Lyapunov exponent is plotted with a different value range in a different figure. This preserves the details of the first two exponents. The third Lyapunov exponent of the Rössler system is illustrated in figure 9.2.

9.1. Performance comparison against AnT

Performance testing is conducted on a Debian 10 system with an Intel(R) Core(TM) i7-4770K CPU with 24GB RAM and an NVIDIA GTX 980 Ti GPU. Additional testing on an NVIDIA RTX 3060 mobile GPU showed similar results. For every combination of system and scan configuration,

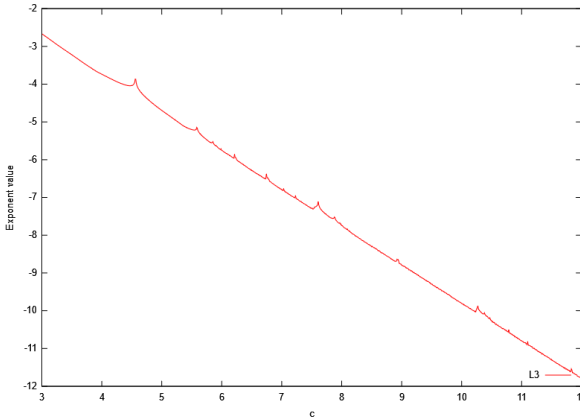


Figure 9.2.: Third Lyapunov exponent of the Rössler system.

the execution time in seconds of the scan is measured for both AnT and AnTGPU. In the following test, only one analysis method is computed in each scan. This allows measuring the performance impact of each method. The performance measurements are given in table 9.1.

System name	Analysis method	Scan-points	Iterations	AnT	AnTGPU	Speedup
Rössler RK4	Lyapunov exponents 1,2,3	10^3	$1.2 * 10^9$	439.6s	2.8s	157x
Gingerbread man	Period analysis	10^6	10^{11}	10192.1s*	3.9s	2613x
Hénon	Lyapunov exponents 1,2	10^6	$3 * 10^{11}$	51631.8s*	29.3s	1762x
Tent	Density analysis	$3.5 * 10^8$	$7 * 10^{11}$	68158.4s*	115.5s	590x
Piecewise-linear	Bandcounting	10^6	10^{12}	190110.5s*	25.1s	7574x
Piecewise-linear	Period analysis	10^6	10^{11}	9689.1s*	3.9s	2484x
Modified logistic	Min/Max	10^6	10^{11}	22424.3*	16.1s	1392x
Piecewise-linear	Symbolic analysis	10^6	10^{11}	26161.3*	23.1	1132x

Table 9.1.: Performance measures for the example scans presented in chapter 8 and chapter 9. Execution time is measure for both AnT and AnTGPU and the speedup factor is calculated.

In table 9.1, scans have been configured equally for both AnTGPU and AnT. A detailed configuration of the AnTGPU scans including precision and epsilon values is given in chapter B. Many AnT scan execution times are estimates. Estimates are annotated with an asterisk. In this case, the scan

was stopped after one percent of the scan points were processed. Completing large scans in AnT can take from multiple hours to several days. This is not feasible on a home computer. In AnT, the number of operations per trajectory is almost identical. No multi-threading is used in AnT. Therefore no thread synchronization affects the computation time of a trajectory. Summarized, the execution time estimates of the AnT scans are relatively accurate. This is easily confirmed empirically on smaller scans like the scan of the Rössler system with Runge-Kutta 4 in AnT. One percent of the computation was completed after approximately 6.5s. This results in an estimated execution time of 650s which is very close to the actual execution time of 633s.

Two major conclusions can be drawn from the comparison of the scan execution time. First, AnTGPU is most efficient when large numbers of scan points are analyzed. The closer the number of scan points comes to just a single trajectory, the smaller the advantage of AnTGPU over AnT becomes. Usually one-dimensional or two-dimensional scans are computed. Often 10^3 scan-points are analyzed in one-dimensional scans, resulting in a 1000 pixel wide plot. In a two-dimensional scan usually 10^6 scan-points are analyzed, resulting in a 1000x1000 pixel image. In the one-dimensional example scan of the Rössler system, the scan execution time of AnTGPU was “only” 157 times faster than AnT, while in the two-dimensional Bandcount scan of the piecewise-linear map the scan execution time was approximately 7574 times faster.

There are two main reasons for this difference in speedup factors. AnT saves data to the file system after every processed scan-point. Therefore scans with a low number of scan points and long trajectories dramatically increase the performance of AnT if the number of system iterations per second is measured. Using batched execution, very large numbers of iterations can be computed in AnTGPU without installing a second GPU or modifying the display driver configuration. In some scans, especially when calculating Lyapunov exponents in higher-dimensional systems like Rössler, the computation of a single trajectory becomes very expensive. This is a problem. Unlike large initial state spaces or large parameter spaces, long trajectories cannot be broken up into multiple batches. In the case of the computation of all Lyapunov exponents for the Rössler system, the maximum trajectory length with a single GPU system and default display driver configuration is about 30000 iterations. Simpler systems can exceed this limit by several orders of magnitude. To reach 30000 iterations per

trajectory in the Rössler scan without interruption by the display driver, the stream processors of the GPU were partially idling. In other words, one batch of scan-points used only a subset of the computing resources of the GPU. This ensures lower latency for special function computation which is necessary for the calculation of Lyapunov exponents. The GPU architecture has a lower throughput for special function operations. This is described in chapter 4.2. This creates additional overhead. If very long or very expensive trajectories are computed, it is recommended to install a second GPU and separate display rendering and scientific computing or configure the display driver to stop updating the user interface, effectively freezing the screen during the scan execution. Additionally, batch overhead can be reduced dramatically as the GPU used to compute the scan no longer updates the display graphics between batches. This can further increase scan execution time by a huge margin.

The second conclusion from the execution time comparison in table 9.1 is that given favorable conditions such as the absence of special functions combined with proper configuration, AnTGPU outperforms AnT by more than three orders of magnitude. This is an outstanding improvement and allows to reconsider the usage of double-precision floating-point numbers, as a performance reduction of 64 caused by double performance still results in a performance increase of 50 times over AnT under favorable conditions. Additionally, special double-precision GPUs exist that have a double-precision throughput comparable to the single-precision throughput of the card used in this test. This might be an option for researchers with a large budget [tes].

9.2. Quality evaluation

The example scans computed in chapter 8.2 show that the analysis methods implemented in AnTGPU produce correct results. All output plots are in accordance with literature and previous scans computed with AnT. In this section, the difference between results computed with AnTGPU and AnT is measured and investigated in detail.

9.2.1. Comparison with AnT

In the following test scenarios, the results computed by the well-tested AnT program are used as reference values. The outputs of AnTGPU are grouped

in two categories: unordered outputs and ordered outputs. Examples of unordered outputs are symbolic strings or period lengths. An unordered output value is either identical to the reference value or incorrect. No measure of proximity to the reference value can be defined. Ordered outputs can be floating-point numbers or integer numbers. For both categories of output values, an error measure is defined. The error measure maps the output values of a scan to a real number. The error measure for unordered values calculates the fraction of output values that are unequal to the reference value. The error measure for ordered values is the average absolute difference between the reference values and the AnTGPU output values. The reference values are denoted as \mathbf{r} , AnTGPU output values are denoted as \mathbf{v} . N is the number of values in a scan. For unordered output values the error measure $err(\mathbf{v}, \mathbf{r})$ is defined as following:

$$err(\mathbf{v}, \mathbf{r}) = \frac{1}{N} \sum_{i=1}^N \phi(v_i, r_i)$$

$$\phi(v_i, r_i) = \begin{cases} 0 & v_i = r_i \\ 1 & otherwise \end{cases}$$

For ordered integer and floating-point values, the accuracy measure is defined as follows:

$$err(\mathbf{v}, \mathbf{r}) = \frac{1}{N} \sum_{i=1}^N |v_i - r_i|$$

For each category of output values, an example scan is taken and the error measure is computed with an identically configured scan from AnT. First, an example for unordered outputs is presented. The period lengths of a piecewise-linear system have been computed in a two-dimensional scan in chapter 8.5. The previous scan is repeated two times with an identical configuration in a one-dimensional setting. a is fixed to the value of 0.5 for the first scan and 0.3 for the second scan. A visual comparison of the AnTGPU output and the reference values for the first scan is given in figure

System name	Analysis method	Error value
Piecewise-linear	Period analysis $a = 0.5$	0.001
Piecewise-linear	Period analysis $a = 0.3$	0

Table 9.2.: Error measures of the period analysis of a piecewise-linear system from chapter 8.5. 1000 scan points were analyzed.

9.3. The average absolute differences between all outputs and references are given in the table 9.2.

The outputs for $a = 0.3$ are identical. The outputs for $a = 0.5$ differ in one value for 1000 scan-points scanned. It turned out to be the first scan point in which the outputs of AnT and AnTGPU differ. Given $x_0 = 0.25, a = 0.5, \Phi - 1$ the output for AnTGPU is 1 while the output for AnT is 0. This means that AnTGPU detected a period of length one while AnT detected no period. For the given parameter setting and initial value, the system function reduces to

$$x_{n+1} = \begin{cases} 0.5x_n & x_n < 0 \\ 0.5x_n - 2 & \text{otherwise} \end{cases}$$

After one iteration the initial value $x_0 = 0.25$ is mapped to $x_1 = -1.875$. From this point on, the trajectory converges from the negative half of the real number line to zero exponentially. Period detection in this case depends on the floating-point standard and method implementation. No further actions are taken.

Second, an example for ordered outputs is presented. All three Lyapunov exponents of the Rössler system have been presented in section 9. A visual comparison of the AnTGPU output and the reference values for the first two Lyapunov exponents is given in figure 9.4. The average absolute differences between all outputs and references are given in the table 9.3.

The average absolute errors to the Lyapunov exponents are quite small. However, most of the difference is caused by single values that differ substantially. This can be observed in figure 9.4 at $c \approx 9.5$. Some spikes in the AnTGPU output are too short and one spike is missing entirely. Numerical estimation of Lyapunov exponents in parameter ranges with chaotic behavior such as $c = 9$ is extremely sensitive to state values. Even incredible small

System name	Analysis method	Error value
Rössler RK4	Lyapunov exponent 1	0.002888
Rössler RK4	Lyapunov exponent 2	0.000695
Rössler RK4	Lyapunov exponent 3	0.003597

Table 9.3.: Average absolute error measures of the Lyapunov exponent analysis of the Rössler system from chapter 9. 1000 scan points were analyzed. The errors are small but larger than rounding errors caused by floating point inaccuracy. This indicates that rounding errors of floating point inaccuracy were magnified during the trajectory iteration. Visible errors are only present in chaotic regions where error magnification is characteristic.

trajectory deviations caused by the lower floating-point accuracy used in AnT can snowball into larger errors. This is not problematic. The Lyapunov spectrum is, except for single values, approximated very accurately. The problem is not solved by using double precision. If the results of AnT using double precision are compared to quadruple precision results, the same problem will appear again in regions with chaotic behavior.

9.2.2. Mitigating floating-point number inaccuracy

The errors witnessed in figure 9.4 are a general problem of numerical simulation of dynamical systems and are not limited to Lyapunov exponents or single-precision floating-point numbers. In some cases, errors induced by floating-point inaccuracy can be mitigated by more sophisticated analysis methods. This is the case for density analysis. In chapter 8.4, density analysis was demonstrated on the tent map. In a large scan, the density of the chaotic attractor in dependency of the parameter a was computed using AnTGPU. In AnTGPU, density analysis is performed differently than in AnT. Instead of computing a single very long trajectory that maps out the spatial density of the attractor, a bundle of short trajectories is computed that maps out the spatial density in parallel. A more detailed description of parallel density estimation is given in chapter 8.4. This approach not only allows to parallelize density analysis but also reduces floating-point errors. Density analysis on the tent map is performed again with three different scans. The first scan computes a single trajectory with 100000 iterations for every parameter value a using 32Bit floating-point numbers. The second scan computes a single trajectory with 100000 iterations for every parameter

value a using 64Bit floating-point numbers. The second scan resembles a typical density analysis in AnT. The third scan computes 350000 trajectories with 2000 iterations for every parameter value a using 32Bit floating-point numbers. The results are illustrated in figure 9.5 from top to bottom.

In the top density diagram, the vertical lines correspond to unstable pseudo periods that formed due to floating-point inaccuracy. In the middle density diagram double precision is used. This either prevents pseudo periods from forming or forces them into larger period lengths that look much smoother and approximate the density of the chaotic attractor better. In the bottom density diagram, single-precision floating-point numbers are used but almost no pseudo periods are visible. This is a direct consequence of the analysis method using trajectory bundles. The probability of every trajectory from a bundle converging to the same pseudo period is small. Instead, many trajectories do not converge to pseudo periods, and those that do overlap each other and are averaged out by other trajectories. The final scan computed many more function iterations in total but is fully parallelizable and the trajectory bundle can be chosen smaller or with different spacing. Measuring the effects of trajectory spacing and the influence of trajectory size is left for future work.

9.3. AnTGPU on nested closed invariant curves

When creating a scientific analysis program evaluation mustn't be solely conducted on textbook examples. In the previous sections, AnTGPU showed great performance on dynamic systems such as Rössler or Hénon. Most of the previously covered systems are well known and investigated. In this section, AnTGPU is used to investigate nested closed invariant curves in a two-dimensional piecewise-linear system with five parameters. The system function is a 2D border collision normal form [AZ19] and given below:

$$x_{n+1} = \begin{cases} \tau_l x_n + y_n + \mu & x_n \leq 0 \\ \tau_r x_n + y_n + \mu & x_n > 0 \end{cases}$$

$$y_{n+1} = \begin{cases} -\delta_l x_n & x_n \leq 0 \\ -\delta_r x_n & x_n > 0 \end{cases}$$

The above-introduced system is a state-of-the-art subject of research and a good example of a system that is analyzed with AnTGPU. At the parameter point $\tau_l = -0.39024$, $\tau_r = 0.3$, $\delta_l = 0.5$, $\delta_r = 1.8094$ and $\mu = 0.05$ multiple

nested attractors coexist in the state space [AZ19]. The state space is scanned to show the attractors and their basins of attraction. A low and a high magnification scan is created to show the nesting of the basins of attractions. In the low magnification scan, the state space is scanned with $x \in [-8, -7]$ and $y \in [-6, -5]$. In the high magnification scan the state space is scanned with $x \in [-7.5, -7.3]$ and $y \in [-5.6, -5.4]$. The state space is sampled with 1000×1000 scan points in both scans. For every scan-point, 100000 function iterations are computed and period length analysis is performed on the last 512 states. The resulting basins of equal period length are illustrated in figure 9.6.

In figure 9.6 it appears that five attractors are nested in the state space. To show that the basins of equal period length equal basins of attraction for one periodic attractor each, the system is additionally scanned for minimum and maximum values. If in a region of equal period length different minimum or maximum values are found, there are either multiple periods with the same length or the period analysis precision is not sufficient. The minimum and maximum state values are illustrated in figure 9.7. Maximum values are shown in the left figures and minimum values in the right figures. Maximum values are shown in the top features, minimum values in the bottom features.

It is difficult to spot in figure 9.7 but in the region, $X \in [-5.1, 5)$ and $Y \in [-7.8, -7.7)$ different minimum and maximum values are in a region of equal period length. This region is scanned again with 1000×1000 scan-points, Only the minimum values of the first dimension and the period length are investigated. The minimum values are plotted in the top figure and the period lengths in the bottom figure. The result is illustrated in figure 9.8.

This clearly shows, that more attractors than previously assumed are present in the state space in the region $x \in [-5.1, 5)$ and $y \in [-7.8, -7.7)$. At $X = -5.03$ and $Y \in (-7.755, -7.756)$ the minimum values range from -8.74 to -8.84 . This value range is well within floating point accuracy and period analysis compare accuracy.

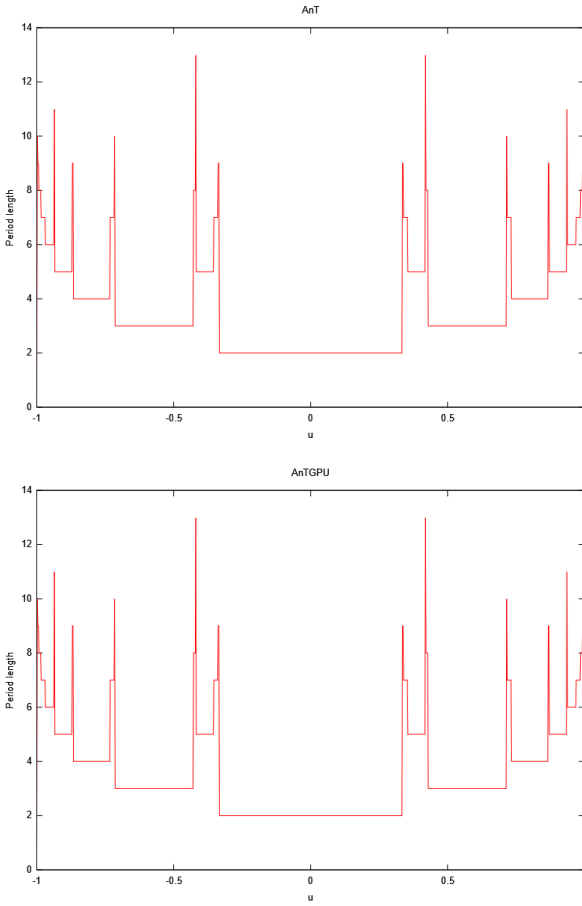


Figure 9.3.: Comparison of the period analysis results on a piecewise-linear map from chapter 8.5. Both plots are identical, $a = 0.5$.

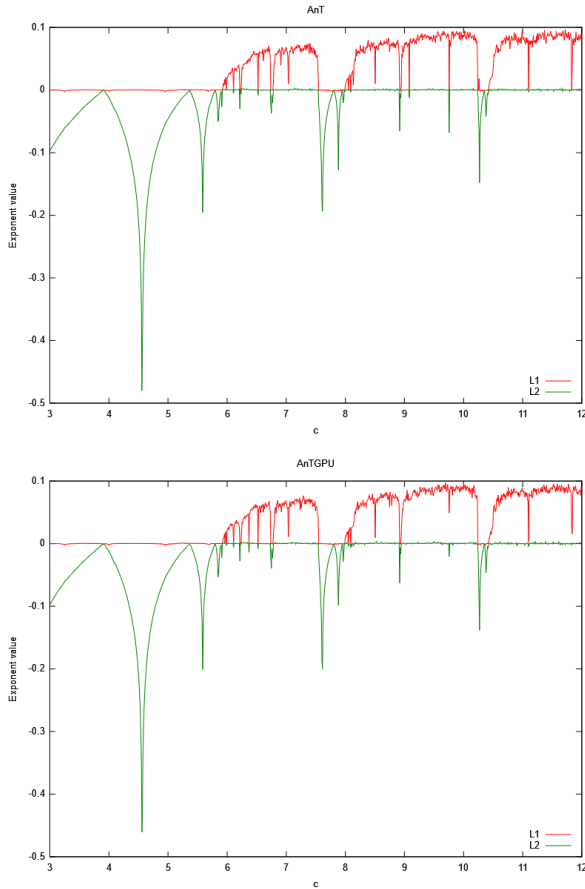


Figure 9.4.: Comparison of the Lyapunov exponent analysis results of the Rössler system from chapter 9. Both plots are very similar. A small difference is observed at $c \approx 9.5$ for both Lyapunov exponents. The first Lyapunov exponent is illustrated in red, and the second Lyapunov exponent is illustrated in green.

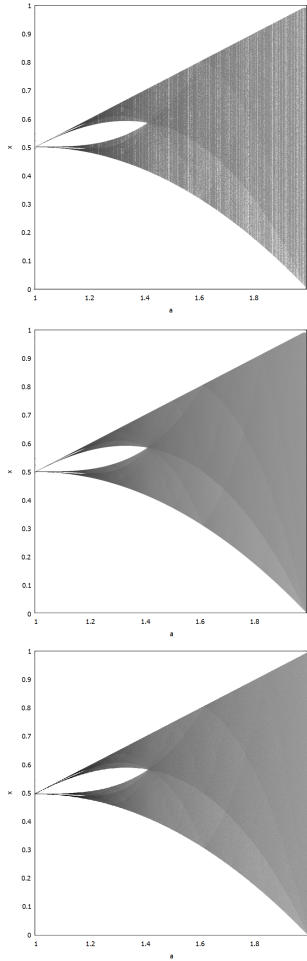


Figure 9.5.: In the top figure the attractor density of the tent system is computed with a scan over a single long trajectory using FP32 accuracy. In the middle figure, the attractor density of the tent system is computed with a scan over a single long trajectory using FP64 accuracy. In the bottom figure, the attractor density of the tent system is computed with a scan over a trajectory bundle with 35000 short trajectories in parallel using FP32 accuracy.

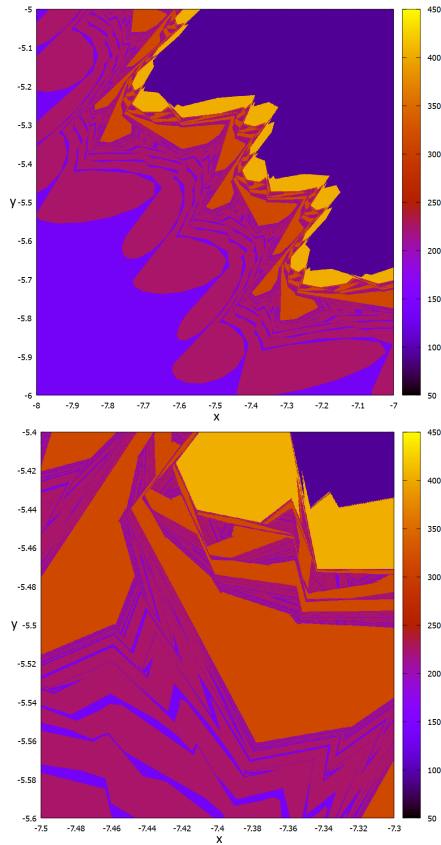


Figure 9.6.: In the top figure a low magnification period analysis scan of the system introduced in chapter 9.3. In the bottom figure, a higher magnification scan is computed. In the analysis five different period lengths are present. It is assumed that five periodic attractors coexist in state space. Later it is revealed that this is not the case.

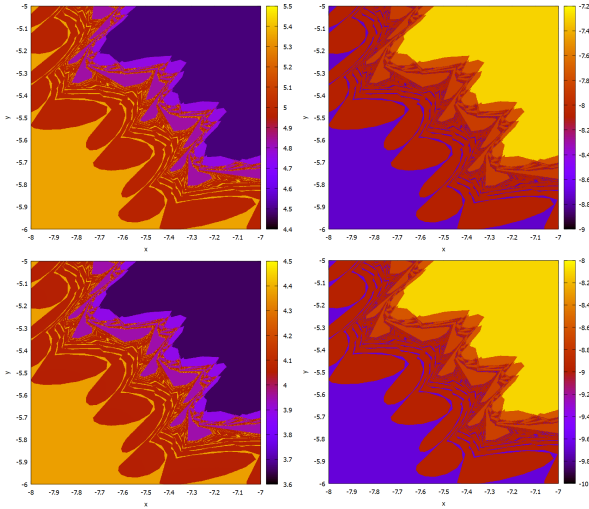


Figure 9.7.: Minimum and maximum values of the trajectories for the system introduced in chapter 9.3. Maximum values are plotted on the left, minimum values are plotted on the right. The X dimension is plotted on the top, the Y dimension on the bottom. The scan area is the same as in the low magnification period analysis scan. For $X = 7.75$ and $Y = 5.1$ different minimum values in a region of equal period length are observed. This confirms that at least multiple periodic attractors with equal period length coexist.

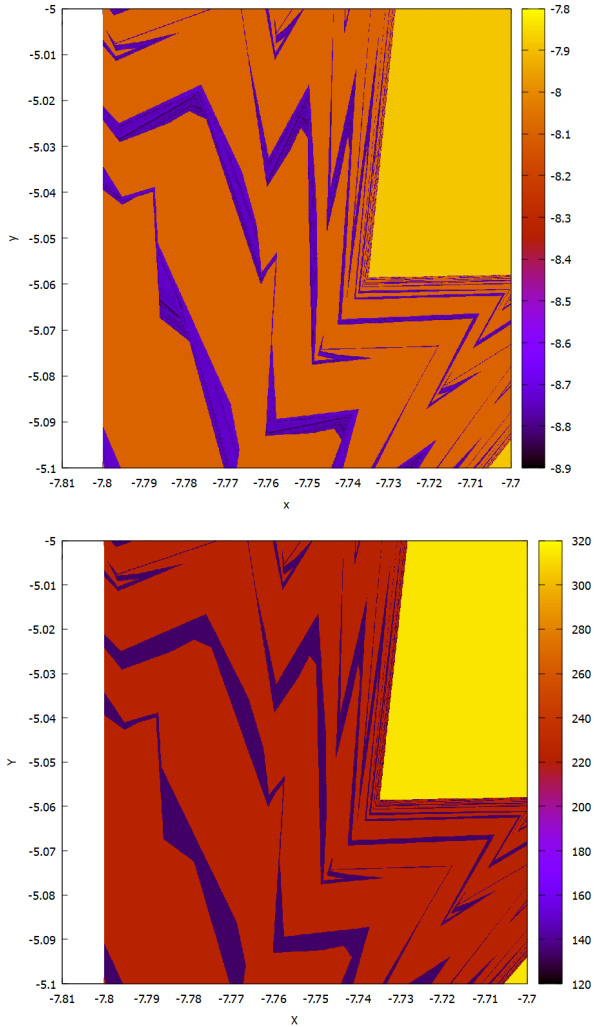


Figure 9.8.: Minimum values of the trajectories for the system introduced in chapter 9.3 in the top figure. This close-up scan is computed to visualize the different minimum values in an area of equal period length. A period length scan of the same area is given in the bottom figure.

10. Conclusion and Outlook

AnTGPU provides three orders of magnitude performance increase over AnT and can process trillions of system function iterations in under a minute. In mere seconds billions of bytes of simulation data are generated and aggregated in parallel in the system analysis methods implemented in AnTGPU. AnTGPU features period analysis, Lyapunov exponent analysis, attractor density estimation, attractor band counting, symbolic sequence analysis, and statistical analysis. These analysis methods are the most important methods from the existing AnT project. The AnT project was analyzed in-depth to develop a highly efficient, GPU-based scan execution and method execution architecture. This manifested in the abandonment of object orientation, dynamic binding, dynamic memory, dynamic linking, and iterator-based data structures. In AnTGPU memory locality, registers and caches, parallel memory access, high instruction throughput, minimal instruction count, efficient multithread looping and branching, and runtime compilation were used to archive maximum performance. This allows using the hardware of the GPU as efficiently as possible. Hardware analysis was conducted on the state-of-the-art NVIDIA GA102 architecture. Runtime compilation allows creating of a scan and problem-specific analysis program that outperforms any general analysis program. Additionally, no dynamically linked library has to be compiled by the user to integrate the system function. AnTGPU uses OpenCL as a computation framework that is compatible with hardware from both NVIDIA and AMD and can be used on Microsoft Windows and Linux. The source code adheres strictly to the C99 standard. This ensures a long life cycle and good cross-platform compatibility.

Testing was performed on all analysis methods and great results were achieved for both, performance and accuracy. Due to performance reasons, 32Bit floating-point numbers are used in AnTGPU, compared to 64Bit floating point numbers used in AnT. The impact of the reduced floating-point accuracy on the analysis result quality was investigated. It was found that result inaccuracies do not magnify overproportionally in most methods. Attractor density estimation is affected more severely by the reduced accuracy.

In AnTGPU, attractor density estimation uses a new approach to estimate the density which mitigates the impact of the lower floating point accuracy and allows for large-scale parallelization.

In the scope of this work, several key insights have been gathered regarding the development and usage of AnTGPU:

- OpenCL is used because it has the best hardware compatibility. Almost all GPUs of the last ten years are compatible.
- 32Bit floating-point numbers have a 64 times higher throughput on commercial GPUs.
- Object-orientation and non-contiguous memory cause a huge overhead. Using thread registers and thread-local memory in analysis methods boosts performance significantly. This attributes to a performance increase of at least factor ten.
- Methods that aggregate data of multiple trajectories use global memory with atomic operations.
- Every trajectory is simulated and analyzed on one GPU thread. A trajectory is atomic.
- On systems with a single GPU batch execution is eventually interrupted by the display driver. On the Rössler system with RK4 calculating Lyapunov exponents this happened after 400000 iterations. A second GPU with no display connected can compute uninterrupted.
- Computing the average of many short trajectories with varying initial values instead of one long trajectory mitigates floating point inaccuracy. 100000 trajectories of length 1000 estimated the density of the tent map attractor almost as accurate as one trajectory with 1000000 iterations and double-precision.
- In most scans AnTGPU is more than 1000 times faster than AnT.
- Special functions such as \log , \sin , $\frac{1}{x}$ slow down computation by a factor of four.
- Loop length in system function should not depend on the state of the trajectory. Fixed length loops are preferred. In the worst-case performance is reduced by a factor of 32.

AnTGPU was created to be a foundation. It was clear from the beginning, that the timeframe of this work requires limitations to the scale of AnTGPU. In future work, the batched thread execution of AnTGPU can be used to transfer, process, and aggregate analysis data from the GPU as soon as the GPU memory is exhausted. This removes the memory limitations of the GPU and allows to compute scans of unprecedented size. Over a trillion bytes of data and quadrillions of system function iterations become feasible. The large performance increase archived in AnTGPU over AnT allows introducing 64Bit floating point calculation to the GPU. Even with reduced performance due to the higher precision, a huge performance boost is achieved with the efficient architecture of AnTGPU. The visualization of large multidimensional analysis results with CPU-based plot programs is often slow and not interactive. AnTGPU was developed to be later integrated into an interactive GPU accelerated analysis studio. Many additions to the analysis methods in AnTGPU are planned. This includes new methods and additional data aggregation. A new method that maps an attractor based on hash functions is planned.

Bibliography

- [acc] The OpenACC Application Programming Interface. <https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf>. Accessed: 2022-05-26.
- [ASo8] Viktor Avrutin and Michael Schanz. On the fully developed bandcount adding scenario. *Nonlinearity*, 21(5):1077, 2008.
- [AZ19] Viktor Avrutin and Zhanybai T Zhusubaliyev. Nested closed invariant curves in piecewise smooth maps. *International Journal of Bifurcation and Chaos*, 29(07):1930017, 2019.
- [Ber18] Pierre Berger. Zoology in the h\`enon family: twin babies and milnor’s swallows. *arXiv preprint arXiv:1801.05628*, 2018.
- [BSo2] Michael Brin and Garrett Stuck. *Introduction to dynamical systems*. Cambridge university press, 2002.
- [Buc10] Ian Buck. The evolution of gpus for general purpose computing. In *Proceedings of the GPU Technology Conference 2010*, page 11, 2010.
- [Cal19] John Calcote. *Autotools: a practitioner’s guide to GNU autoconf, automake, and libtool*. No Starch Press, 2019.
- [CIKE93] Leon O Chua, Makoto Itoh, Ljupco Kocarev, and Kevin Eckert. Chaos synchronization in chua’s circuit. *Journal of Circuits, Systems, and Computers*, 3(01):93–108, 1993.
- [com] CUDA GPU Compute Capability. <https://developer.nvidia.com/cuda-gpus>. Accessed: 2022-05-26.
- [Dev88] Robert L Devaney. Fractal patterns arising in chaotic dynamical systems. In *The science of fractal images*, pages 137–168. Springer, 1988.
- [Ecko6] Bernd Eckstein. *Bandcounter: counting bands of multiband chaotic attractors*. Universitätsbibliothek der Universität Stuttgart, 2006.

- [For15] William Ford. Chapter 14 - gram-schmidt orthonormalization. In William Ford, editor, *Numerical Linear Algebra with Applications*, pages 281–297. Academic Press, Boston, 2015.
- [GA1] NVIDIA AMPERE GA102 GPU ARCHITECTURE. <https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf>. Accessed: 2022-05-26.
- [Hao91] Bai-lin Hao. Symbolic dynamics and characterization of complexity. *Physica D: Nonlinear Phenomena*, 51(1-3):161–176, 1991.
- [HbL] Roberto Hasfura-b and Phillip Lynch. Periodic points of the family of tent maps.
- [Hén76] Michel Hénon. A two-dimensional mapping with a strange attractor. In *The theory of chaotic attractors*, pages 94–102. Springer, 1976.
- [Kee80] J. P. Keener. Chaotic behavior in piecewise continuous difference equations. *Trans. Am. Math. Soc.*, 261(2):589–604, 1980.
- [Lar11] Robert S Laramee. Bob’s concise introduction to doxygen. Technical report, Technical report, The Visual and Interactive Computing Group, Computer . . . , 2011.
- [mem] Memory Coalescing. <https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/>. Accessed: 2022-05-26.
- [Nvia] CUDA Zone. <https://developer.nvidia.com/cuda-zone>. Accessed: 2022-05-26.
- [Nvib] NVIDIA TESLA V100-GPU. <https://www.nvidia.com/de-de/data-center/tesla-v100>. Accessed: 2022-05-26.
- [Nvic] Using CUDA and X. https://nvidia.custhelp.com/app/answers/detail/a_id/3029/~/using-cuda-and-x. Accessed: 2022-05-26.

- [ocla] OpenCL Overview. <https://www.khronos.org/opengl/>. Accessed: 2022-05-26.
- [oclb] OpenCL supported devices. <https://www.khronos.org/conformance/adopters/conformant-products/opengl>. Accessed: 2022-05-26.
- [oclc] The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html. Accessed: 2022-05-26.
- [PMS21] Krishna Pusuluri, Hil GE Meijer, and Andrey L Shilnikov. Homoclinic puzzles and chaos in a nonlinear laser model. *Communications in Nonlinear Science and Numerical Simulation*, 93:105503, 2021.
- [pro] CUDA Programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#maximize-instruction-throughput>. Accessed: 2022-05-26.
- [Rös76] Otto E Rössler. An equation for continuous chaos. *Physics Letters A*, 57(5):397–398, 1976.
- [Scho4] M. Schanz. *The AnT project: On the simulation and analysis of dynamical systems*. Habilitation. University of Stuttgart, 2004.
- [set] CPU performance. https://setiathome.berkeley.edu/cpu_list.php. Accessed: 2022-05-26.
- [SH98] Andrew Stuart and Anthony R Humphries. *Dynamical systems and numerical analysis*, volume 2. Cambridge University Press, 1998.
- [Ste10] Shlomo Sternberg. *Dynamical systems*. Courier Corporation, 2010.
- [tec] Nvidia RTX 3090 Ti. <https://www.techpowerup.com/gpu-specs/geforce-rtx-3090-ti.c3829>. Accessed: 2022-05-26.
- [tes] NVIDIA Tesla A100. <https://www.nvidia.com/de-de/data-center/a100/>. Accessed: 2022-05-26.

- [WH96] Gerhard Wanner and Ernst Hairer. *Solving ordinary differential equations II*, volume 375. Springer Berlin Heidelberg New York, 1996.
- [WSSV85] Alan Wolf, Jack B Swift, Harry L Swinney, and John A Vastano. Determining lyapunov exponents from a time series. *Physica D: nonlinear phenomena*, 16(3):285–317, 1985.
- [ZPL⁺11] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In *2011 IEEE international symposium on workload characterization (IISWC)*, pages 205–215. IEEE, 2011.

A. AnTGPU User documentation

In the following sections, the most recent version of the AnTGPU user documentation is included. The documentation was last edited on 19.05.2022 and describes the configuration file and system file syntax used in AnTGPU 1.7. Additionally, simple examples are given. All option keys are briefly described and correspond to the methods introduced in chapter 8.

A.1. Config file documentation

The system configuration is saved in a file “system”.cfg. If the config file is in a different folder than AnTGPU or named differently, a path with a file name has to be specified during the program execution or as a command-line option.

A.1.1. Features

- The config language is key-value pair based. Each option is specified by a key, a delimiter ‘=’, and a value
- Inline comments are denoted by “”. The remaining line after this character is ignored
- Empty lines don’t affect the data interpretation and can be used for structuring
- The order of the key-value pairs can be arbitrary. Correct interpretation does not rely on the order of the key-value pairs
- Invalid syntax of key-value pairs raises a warning. Invalid pairs are not interpreted
- Specifying only a subset of the supported key-value pairs is allowed. This can affect the program execution if missing pairs are dependent on each other

- Keys are unique. No key is allowed to be contained in another key
- The addition of unsupported key-value pairs does not affect the config interpretation
- Keys can be indexed. This allows for a key to be specified multiple times with different values of the same datatype
- Values can be tuples of simple datatypes. This allows a single key to reference multiple values of possibly different data types.

A.1.2. Syntax example

An example of comments, a key-value pair, a key with tuple value, and an indexed key:

```
#I am a comment
key = value
tuplekey = (0, 1, hallo)

key2[0] = value0 #I am a comment
key2[1] = value1
```

A.1.3. Option keys

In the following table, all supported option keys are listed with information about the data type, the syntax of the value, and a description of the option. **Value syntax is sometimes given with quotation marks eg. "value". In the config file, these quotation marks have to be omitted.**

Basic system options

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
num_batches	int	no	> 0	Sets the number of batches in which the scan is executed on the GPU.	no
type	enum	no	"map" or "diff"	Sets the system type.	no
name	string	no		Sets the system file name. Used to find the system function file.	no
path	string	no		Path of the system config file.	yes
state_dim	int	no	1 to 30000	Sets system state dimension.	no
initial_state[n]	float (Tuple)	yes	"(float value)" or "(float range_min, float range_max, int num_points, enum spacing_method)"	Sets the n-th initial state dimension to a value or range.	no
parameter_dim	int	no	0 to 30000	Sets system parameter space dimension.	no
parameter_state[n]	float (Tuple)	yes	"(float value)" or "(float range_min, float range_max, int num_points, enum spacing_method)"	Sets the n-th parameter space dimension to a value or range.	no
scan_iterations	int	no	> 0	Sets the number of iterations per trajectory.	yes
output_format	enum	no	"gnu" or "antstudio"	Sets the format of output files. This can also affect the number of files generated.	yes
scan_points_file	string	no		File name of the scan point file. Used in "antstudio" output only.	yes

Basic saving

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_last_states	bool	no	"true" or "false"	Sets if the last states are calculated. They are required for cyclic and acyclic files.	yes
last_states_num	int	no	> -1	Sets how many of the last states are saved.	yes
save_last_states	bool	no	"true" or "false"	Sets if the last states are save to the disk.	yes
last_states_file	string	no		Filename of last states save.	yes

General evaluation

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_min_values	bool	no	"true" or "false"	Sets if min values are computed and saved to file.	yes
calc_max_values	bool	no	"true" or "false"	Sets if max values are computed and saved to file.	yes
calc_wave_numbers	bool	no	"true" or "false"	Sets if wavenumber values are computed and saved to file.	yes
calc_mean_values	bool	no	"true" or "false"	Sets if mean values are computed and added to the general evaluation file. If all options are false, this feature is deactivated.	yes
general_transient_iterations	int	no	> -1	Sets how many transient iterations occur before the methods "mean", "min_max" and "wavenumbers" are activated.	yes
min_value_file	string	no		Filename of min value save.	yes
max_value_file	string	no		Filename of max value save.	yes
mean_value_file	string	no		Filename of mean value save.	yes
wavenumber_file	string	no		Filename of wavenumber save.	yes

Period analysis

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_period	bool	no	"true" or "false"	Sets if period is computed and the period length is saved to file.	yes
max_period_length	int	no	> -1	Sets the largest period that is tested. Setting 0 deactivates this feature entirely.	yes
compare_precision	float	no	> 0	Sets the compare precision for period testing.	yes
divergence_bound	float	no	> 0	Sets the divergence bound. If the max-norm of a state exceeds this bound, the trajectory is marked as "divergent" or -1 in the period length file.	yes
period_length_file	string	no		Filename of period length save.	yes
save_periodic_trajectories	bool	no	"true" or "false"	Sets if the cyclic trajectories are saved to a separate file. Only available for "gnu" output format. Requires calc_last_states equals "true".	yes
save_acyclic_trajectories	bool	no	"true" or "false"	Sets if the acyclic trajectories are saved to a separate file. Only available for "gnu" output format. Requires calc_last_states equals "true".	yes
period_file	string	no		Filename of cyclic trajectory file.	yes
acyclic_file	string	no		Filename of acyclic trajectory file.	yes

Density analysis

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_density	bool	no	"true" or "false"	Sets if density is computed and saved to file.	yes
density_grid_dimension	int	no	> -1	Sets the dimension of the density grid.	yes
density_grid_slice_dimensions	int	yes	> 0	Sets the system state dimensions across which the grid is spanned. This allows to use only a subset of the system state dimensions in the density grid.	yes
density_grid_bounds	float (tuple)	yes	"(float min, float max)"	Sets the bounds of the density grid across the indexed grid dimension.	yes
density_grid_resolution	int	int	> 0	Sets the density grid resolution across the indexed grid dimension.	yes
density_file	string	no		Filename of density grid save.	yes
density_transient_iterations	int	no	> -1	Sets the number of transient iterations occurring before the density is sampled.	yes
density_overflow_protection	bool	no	"true" or "false"	Sets if overflow protection is used on the unsigned ints of the density grid. Uses more computation power.	yes

Bandcounting

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_bandcount	bool	no	"true" or "false"	Sets if bandcount is computed and saved to file.	yes
bandcount_grid_dimension	int	no	> -1	Sets the dimension of the bandcount grid.	yes
bandcount_grid_slice_dimensions	int	yes	> 0	Sets the system state dimensions across which the grid is spanned. This allows to use only a subset of the system state dimensions in the bandcount grid.	yes
bandcount_grid_bounds	float (tuple)	yes	"(float min, float max)"	Sets the bounds of the bandcount grid across the indexed grid dimension.	yes
bandcount_grid_resolution	int	int	> 0	Sets the bandcount grid resolution across the indexed grid dimension.	yes
bandcount_file	string	no		Filename of bandcount grid save.	yes
bandcount_transient_iterations	int	no	> -1	Sets the number of transient iterations occurring before the bandcount is sampled.	yes

Lyapunov exponents

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_lyapunov_exp	bool	no	"true" or "false"	Sets if lyapunov exponents are computed and saved to file.	yes
num_lyapunov_exp	int	no	state_dim >= value > 0	Sets how many lyapunov exponents are computed.	yes
lyapunov_steps	int	no	> 0	Sets how many iterations are computed until a gram-schmidt reorthogonalization is performed.	yes
lyapunov_eps	float	no	> 0	Sets the epsilon that displaces the side trajectories around the main trajectory.	yes
lyapunov_transient_iterations	int	no	> 0	Sets how many transient iterations are computed before the lyapunov exponents are computed.	yes
lyapunov_file	string	no		Filename of Lyapunov exponent save.	yes

Symbolic analysis

Key Specifier	Datatype	Indexed	Value Syntax	Description	Optional
calc_symbolics	bool	no	"true" or "false"	Sets if symbolics are computed and saved to file.	yes
num_last_symbol_states	int	no	> 0	Sets how many last states are appended to the symbol string.	yes
sort_symbolics	bool	no	"true" or "false"	Sets if symbolics are shifted such that equal periods have the same phase shift.	yes
symbolic_file	string	no		Filename of symbolic save.	yes

A.2. System function file documentation

A system function is a single function that specifies a state transition. System functions are saved as "systemfilename".c. The "systemfilename" is specified

in the config file. If the file is in a different folder than the program, a path to the directory has to be specified in the config file.

A.2.1. Syntax

The system function follows a simple syntax that is inspired by a C method declaration. There is only one function evaluated in the file. Additional functions are not supported.

C-style comments are fully supported. Four directives structure the system function file. `define`, `vardef`, `program` and `symbol`.

C-style defines can be used to simplify variable names. Only **define A B** is supported. This replaces A with B. A define statement may be followed by a comment or a line break but not source code. Defines can be used to abbreviate variable names. If parameters are constant it is recommended to specify them as variables and set the value in the config file. The parser will optimize constant values. Define dependencies are not supported. In the following example, X will be replaced by Y but not further by Z. Y will be replaced by Z.

```
#define X Y
#define Y Z

//example string
XXYY

//output of this define implementation
YYZZ
```

The directives `vardef`, `program`, and `symbol` indicate a region of code until the next directive is found. After a directive code can be added according to the [OpenCL C Specification](#). This includes math functions and control flow. No external libraries are allowed. `vardef` and `symbol` are optional. `vardef` allows the user to define and initialize additional variables, not given by the system. `symbol` allows specifying a symbol-function. The general structure of the system function file is given below.

```
#define X Y

#vardef
//my variables
#program
//my system function
#symbol
//my symbol function
```

A.2.2. Predefined variables

The user has to use predefined variables for states and parameters in his program or symbol section. n after a variable indicates, that there are multiple variables of the same name, starting at 0, specifying dimension. **E.g. `state[n]` can be `state0`, `state1`, ..., `stateN`. The state is always followed by a numeric value.** State and parameters are indexed as specified in the options file. Predefined variables are described below:

Name	Datatype	Description
<code>state[n]</code>	float	Individual dimensions of the current state vector.
<code>nstate[n]</code>	float	Individual dimensions of the next state vector. Write the new values here.
<code>param[n]</code>	float	Parameters of the system function.
<code>_symbol</code>	char	Current symbol, associated to the current state.

A.2.3. Forbidden variable names

The internal functions use variables, that are not allowed to be declared by the user. If the variables are not listed as predefined variables above, it is not recommended to use forbidden variables at all. The names of the forbidden variables are given below:

Name	Name	Name
<code>state[n]</code>	<code>_systemParameters</code>	<code>_periodLength</code>
<code>nstate[n]</code>	<code>_meanStates</code>	<code>_densityMatrixStack</code>
<code>param[n]</code>	<code>_minStates</code>	<code>_bandCountMatrixStack</code>
<code>_symbol</code>	<code>_maxStates</code>	<code>_lyapunovExponents</code>
<code>threadOffset</code>	<code>_waveNumbers</code>	<code>_symbolStrings</code>
<code>_initialStates</code>	<code>_lastStates</code>	<code>threadIdx</code>
<code>paramSpaceIdx</code>	<code>meanstate[n]</code>	<code>minstate[n]</code>
<code>maxstate[n]</code>	<code>wavenumber[n]</code>	<code>pstate[n]</code>
<code>refstate[n]</code>	<code>densityGridIdx[n]</code>	<code>densityStackIndex</code>
<code>bandCountGridIdx[n]</code>	<code>bandCountStackIndex</code>	<code>lyapunov[n]expstate[m]</code>
<code>nlyapunov[n]expstate[m]</code>	<code>lyapunovexp[n]</code>	<code>lyapunovlength[n]</code>
<code>lyapunovcycle</code>	<code>_i</code>	

Additionally, variables and macros defined in the OpenCL specification should be used appropriately.

A.2.4. Example function

A complete example function with all directives used is given below:

```
#define X state0
#define A param0

#vardef
float a;
a = 3.2;

int put;

#program
nstate0 = A*X*(1.0-X); //I am a comment
put = 4;
a = put * 2;

#symbol
if(X > X*X){
    _symbol = 'A';
} else{
    _symbol = 'B';
}
```

Not matching the above specification for correctly structuring the system function may lead to errors or unpredictable outcomes.

A.3. Output file documentation

AnTGPU outputs analysis results for each group of methods specified in the config file. Two formats affect all output files, "gnu" and "antstudio". "antstudio" output requires custom plotting tools and is therefore not introduced here. If AnTGPU is used independently from its plotting framework, it is recommended to use the "gnu" output format alongside Gnuplot.

In the Gnuplot format, each data point is stored in a row. Columns are separated by blanks. From left to right first initial states, then parameters, and finally data points are stored in a row.

A.4. Program execution documentation

Program execution is straightforward. The executable can be launched from the command line or a GUI without arguments. Necessary data is provided by the user during runtime.

If an automated execution is required, additional arguments can be passed. Currently, these arguments are the `platform_id`, `device_id`, and config file path. They are used to select the preferred OpenCL device and locate the configuration file. The first two arguments are supplied as Integers in string format greater than `-1`. The file path is supplied as a string with forward slashes.

A Linux and Windows example is given below:

```
./AntGPU 0 0 /a/b/mysystem.cfg  
AntGPU.exe 0 0 C:/mysystem.cfg
```

This example selects platform `0` and device `0` automatically and sets a path. Other values can be used for different devices, platforms, and paths. No further user input is required.

B. Example scan configurations

In this chapter the AnTGPU configuration files for most scans computed are appended. The following configuration files are to be used with AnTGPU 1.7.

B.1. Modified logistic function & General evaluation

Configuration file of the scan of the modified logistic function using general trajectory evaluation.

```
#dynamical system
num_batches = 200
type = map
name = demomap
state_dim = 1
initial_state[0] = (-3,3,1000, lin)
parameter_dim = 1
parameter_state[0] = (-2.7, 1.52, 1000, lin)
scan_iterations = 100000
output_format = gnu
scan_points_file = scanPoints.csv

#last states
calc_last_states = true
last_states_num = 1
save_last_states = true
last_states_file = lastStates.csv

#general evaluation
calc_min_values = true
calc_max_values = true
calc_wave_numbers = false
```

```
calc_mean_value = false
general_transient_iterations = 10000
min_value_file = minvalues.csv
max_value_file = maxvalues.csv
mean_value_file = meanvalues.csv
wavenumber_file = wavenumbers.csv
```

B.2. Gingerbreadman map & Period analysis

Configuration file of the scan of the Gingerbreadman map using Period analysis.

```
#dynamical system
num_batches = 2
type = map
name = demomap
state_dim = 2
initial_state[0] = (-10, 10, 1000, lin)
initial_state[1] = (-10, 10, 1000, lin)
parameter_dim = 0
#parameter_state[0] = (1.5, 4, 1000, lin)
scan_iterations = 100000
output_format = gnu
scan_points_file = scanPoints.csv

#period
calc_period = true
max_period_length = 512
compare_precision = 1e-03
divergence_bound = 1000
period_length_file = periodLength.csv
save_periodic_trajectories = false #gnuplotter only
save_acyclic_trajectories = false #gnuplotter only
period_file = periods.csv #gnuplotter only
acyclic_file = acyclic.csv #gnuplotter only
```

B.3. Hénon map & Lyapunov exponents

Configuration file of the scan of the Hénon map using Lyapunov exponents.

```
#dynamical system
num_batches = 50
type = map
name = demomap
state_dim = 2
initial_state[0] = (0.25)
initial_state[1] = (0.25)
parameter_dim = 2
parameter_state[0] = (0, 2, 1000, lin)
parameter_state[1] = (0, 1, 1000, lin)
scan_iterations = 100000
output_format = gnu #gnu = gnuplotter output
scan_points_file = scanPoints.csv

#lyapunov
calc_lyapunov_exp = true
num_lyapunov_exp = 2
lyapunov_steps = 3
lyapunov_eps = 1e-03
lyapunov_transient_iterations = 10000
lyapunov_file = lyapunovExpFile.csv
```

B.4. Rössler system & Lyapunov exponents

Configuration file of the scan of the Rössler system using Lyapunov exponents.

```
#dynamical system
num_batches = 50
type = map
name = demomap
state_dim = 3
initial_state[0] = (0)
```

B. Example scan configurations

```
initial_state[1] = (0)
initial_state[2] = (0)
parameter_dim = 3
parameter_state[0] = (0.15)
parameter_state[1] = (0.2)
parameter_state[2] = (3, 12, 1000, lin)
scan_iterations = 200000
output_format = gnu
scan_points_file = scanPoints.csv

#lyapunov
calc_lyapunov_exp = true
num_lyapunov_exp = 3
lyapunov_steps = 10
lyapunov_eps = 1e-03
lyapunov_transient_iterations = 100000
lyapunov_file = lyapunovExpFile.csv
```

B.5. Tent map & Density analysis

Configuration file of the scan of the Tent map using density analysis.

```
#dynamical system
num_batches = 1000
type = map
name = demomap
state_dim = 1
initial_state[0] = (0, 1, 350000, lin)
parameter_dim = 1
parameter_state[0] = (1.1, 2, 1000, lin)
scan_iterations = 2000
output_format = gnu
scan_points_file = scanPoints.csv

#density
calc_density = true
density_grid_dimension = 1
```

```
density_grid_slice_dimensions[0] = 0
density_grid_bounds[0] = (0, 1)
density_grid_resolution[0] = 1000
density_file = densityMatrixList.csv
density_transient_iterations = 1000
density_overflow_protection = false
```

B.6. PWS map & Bandcounting

Configuration file of the scan of the piecewise-linear map using Bandcounting.

```
#dynamical system
num_batches = 50
type = map
name = demomap
state_dim = 1
initial_state[0] = (0.25)
parameter_dim = 2
parameter_state[0] = (1, 2, 1000, lin)
parameter_state[1] = (-1, 1, 1000, lin)
scan_iterations = 1000000
output_format = gnu
scan_points_file = scanPoints.csv

#density
calc_bandcount = false
bandcount_grid_dimension = 1
bandcount_grid_slice_dimensions[0] = 0
bandcount_grid_bounds[0] = (-2.0, 2.0)
bandcount_grid_resolution[0] = 128
bandcount_transient_iterations = 10000
bandcount_file = bandCounts.csv
```

B.7. PWS map & Period analysis

Configuration file of the scan of the piecewise-linear map using Period analysis.

```
#dynamical system
num_batches = 50
type = map
name = demomap
state_dim = 1
initial_state[0] = (0.25)
parameter_dim = 2
parameter_state[0] = (0, 1, 1000, lin)
parameter_state[1] = (-1, 1, 1000, lin)
scan_iterations = 100000
output_format = gnu
scan_points_file = scanPoints.csv

#period
calc_period = true
max_period_length = 32
compare_precision = 1e-04
divergence_bound = 1000
period_length_file = periodLength.csv
save_periodic_trajectories = false
save_acyclic_trajectories = false
period_file = periods.csv
acyclic_file = acyclic.csv
```

B.8. PWS map & Symbolic analysis

Configuration file of the scan of the piecewise-linear map using Symbolic analysis.

```
#dynamical system
num_batches = 50
type = map
```

```
name = demomap
state_dim = 1
initial_state[0] = (0.25)
parameter_dim = 2
parameter_state[0] = (0, 1, 1000, lin)
parameter_state[1] = (-1, 1, 1000, lin)
scan_iterations = 100000
output_format = gnu
scan_points_file = scanPoints.csv

#period
calc_period = true
max_period_length = 256
compare_precision = 1e-04
divergence_bound = 1000
period_length_file = periodLength.csv
save_periodic_trajectories = false
save_acyclic_trajectories = false
period_file = periods.csv
acyclic_file = acyclic.csv

#symbols
calc_symbolics = true
num_last_symbol_states = 256
sort_symbols = true
symbolic_file = symbolFile.csv
```


B. Example scan configurations

C. Example system functions

In this chapter the AnTGPU system function files for more complex scans are appended. This includes a symbolic function and numerical integration method. The following system function files are to be used with AnTGPU 1.7.

C.1. PWS map & Symbolic analysis

```
#define X state0
#define A param0
#define U param1

#program
if(X < 0){
nstate0 = A*X+U+1;
}else{
nstate0 = A*X+U-1;
}

#symbol
if(X < 0){
_symbol = 'L';
}else{
_symbol = 'R';
}
```

C.2. Rössler system & Lyapunov exponents

```
#define X state0
#define Y state1
```

C. Example system functions

```
#define Z state2

#define A param0
#define B param1
#define C param2

#define h 0.01
#define h2 0.005

#vardef
float k1;
float k2;
float k3;
float k4;

float l1;
float l2;
float l3;
float l4;

float m1;
float m2;
float m3;
float m4;

#program
k1 = -(Y+Z);
l1 = X + A*Y;
m1 = B + (X-C)*Z;

k2 = -(Y + h2*l1 + Z + h2*m1);
l2 = (Y + h2*l1)*A + X + h2*k1;
m2 = B + (X +h2*k1 - C)*(Z+h2*m1);

k3 = -(Y + h2*l2 + Z + h2*m2);
l3 = (Y + h2*l2)*A + X + h2*k2;
m3 = B + (X +h2*k2 - C)*(Z+h2*m2);

k4 = -(Y + h*l3 + Z + h*m3);
```

```
l4 = (Y + h*l3)*A + X + h*k3;  
m4 = B + (X +h*k3 - C)*(Z+h*m3);
```

```
nstate0 = X + h*0.1666666*(k1 + 2*k2 + 2*k3 + k4);  
nstate1 = Y + h*0.1666666*(l1 + 2*l2 + 2*l3 + l4);  
nstate2 = Z + h*0.1666666*(m1 + 2*m2 + 2*m3 + m4);
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Druck-Exemplaren überein.

Datum und Unterschrift:

Declaration

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

Date and Signature: