Technische Universität München

Department of Mathematics

Bachelor's Thesis

GAIO.jl: Set-oriented Methods for Approximating Invariant Objects, and their Implementations in julia

April Hannah-Lena Herwig

Supervisor: Prof Oliver Junge

Submission Date: ...

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Garching,

Zusammenfassung

Bei einer in englischer Sprache verfassten Arbeit muss eine Zusammenfassung in deutscher Sprache vorangestellt werden. Dafür ist hier Platz.

Contents

Remark. Credit for cited images goes entirely to image authors. Where a cited image is used, see its citation for more information.

1 Dynamical Systems

1.1 Motivation

Our goal is to investigate the qualitative, long-term behavior of systems in which a given function describes the trajectory of a point in an ambient space. Such dynamical systems are used in modelling physcial phenomena, economic forecasting, differential equations, etc. We wish to construct topological *covers* of sets which describe the infinite-time dynamics of some portions of the system, as well as statistical *invariant measures* which describe much larger sets in the space, but with less information.

The basic technique of all the topological algorithms is to split a compact set Q into a partition $\mathcal{P}$ of *boxes* - that is, generalized rectangles $[c_1 - r_1,\ c_1 + r_1) \times \ldots \times [c_d - r_d,\ c_d + r_d)$, with center vector $c \in \mathbb{R}^d$ and componentwise radii $r \in \mathbb{R}^d$. The algorithms will begin with a set of boxes $\mathcal{B}$, and then repeatedly subdivide each box in $\mathcal{B}$ into two (or more) smaller boxes, examine the dynamics of the subdivided boxes, and refine the box set to include only the boxes we are interested in.

The algorithms described in the present paper have been previously implemented in the statistical programming language `matlab` [21], but is now being fully refactored and reimplemented in the open-source, composable language `julia` [7]. The reason for this change is julia's high-level abstraction capabilities, just-in-time compilation, and in-built set-theoretical functions, which create short, elegant code which is nonetheless more performant. Source code for `GAIO` in `matlab` and `julia` and be found in [14] and [15], respectively.

1.2 Definitions

This section should be treated as an index of definitions, to be referred back to as necessary during reading. In the following, we assume M is a compact or a smooth manifold in $\mathbb{R}^d$, endowed with a metric d, and the map $f : M \to M$ is at least $\mathcal{C}^0$. We further assume that P is a partition of a compact set $Q \subset M$ of (up to a null set) disjoint boxes, and $\mathcal{B} \subset \mathcal{P}$ is a subset of boxes. Our setting is a *discrete, autonomous dynamical system*, that is, a system of the form:

$$x_{k+1} = f(x_k), \quad k = 0,\ 1,\ 2,\ \ldots \tag{1.1}$$

A continuous dynamical system $\dot{x} = F(x)$ can be *discretized* by, for example, considering the *Poincaré time-t map* over some $d-1$ dimensional hyperplane, or by setting one "step" of the system as integrating F for a set time t.

First, we set some notation for convenience.

Definition 1.1 (Image of a Box Set). We will call the *image of $\mathcal{B}$ under f* the set of boxes which intersect with the image $f(B)$, for at least one $B \in \mathcal{B}$. More precisely, it is

$$f(\mathcal{B}) = \left\{ R \in \mathcal{P} \quad | \quad f^{-1}(R) \cap \bigcup_{B \in \mathcal{B}} B \neq \emptyset \right\}. \tag{1.2}$$

Theorem 1.2 (Image of a Box Set)**.** $f(\mathcal{B})$ is the inclusion-minimal cover of $f(\bigcup_{B\in\mathcal{B}} B)$ with boxes from $\mathcal{P}$.

Proof. We have the equivalent characterisation

$$f(\mathcal{B}) = \{R \in \mathcal{P} \quad | \quad \exists\ B \in \mathcal{B} \text{ and } x \in B \ : \ f(x) \in R\}. \tag{1.3}$$

Hence if we remove one box R from $f(\mathcal{B})$, then there exists an $x \in \bigcup_{B\in\mathcal{B}} B$ which maps outside of the created box set.

□

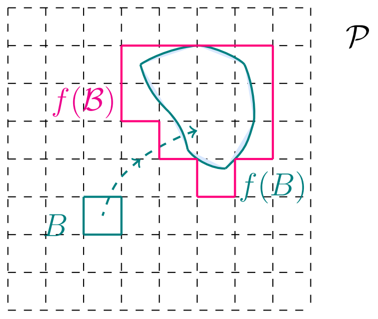

Figure 1.1: Image of the simple box set $\mathcal{B} = \{B\}$

We continue with a set of topological definitions of sets we wish to approximate.

Definition 1.3 ((Forward-, Backward-) Invariant)**.** [11] A set A is called *forward-invariant* if $f(A) \subset A$, *backward-invariant* if $f^{-1}(A) \subset A$, and *invariant* if it is both forward- and backward-invariant.

Definition 1.4 (Attracting Set)**.** [9] An invariant set A is called *attracting* with *fundamental neighborhood* U if for every open set $V \supset A$ there is an $N \in \mathbb{N}$ such that the tail $\bigcup_{k\geq N} f^k(U)$ lies entirely within A. The attracting set is also called *global* if the *basin of attraction*

$$B(A) = \bigcap_{k\geq 0} f^{-k}(U) \tag{1.4}$$

is the whole of $\mathbb{R}^n$.

The basin of attraction acts in some sense as the set for which all points eventually arrive in A. Since the map f is smooth, then the closure $\bar{A}$ is invariant too. With continuity it becomes clear that

$$A = \bigcap_{k \geq 0} f^k(U). \tag{1.5}$$

The global attractor is maximal in the sense that it contains all backward-invariant sets within the system. In particular, it contains local unstable manifolds.

Definition 1.5 (Stable and Unstable Manifolds)**.** [26] Let $\bar{x}$ be a fixed point of the diffeomorphism f, and U a neighborhood of x. Then the *local unstable manifold* is given by

$$W^u(\bar{x}, U) = \left\{ x \in U \mid \lim_{k \to \infty} d(f^{-k}(x),\ \bar{x}) = 0 \text{ and } f^{-k}(x) \in U\ \forall\, k \geq 0 \right\}. \tag{1.6}$$

The *global unstable manifold* is given by

$$W^u(\bar{x}) = \bigcup_{k \geq 0} f^k(\, W^u(\bar{x}, U)\,). \tag{1.7}$$

The dual definition of the *(local) stable manifold* is obtained by reversing the sign of k in the above equations.

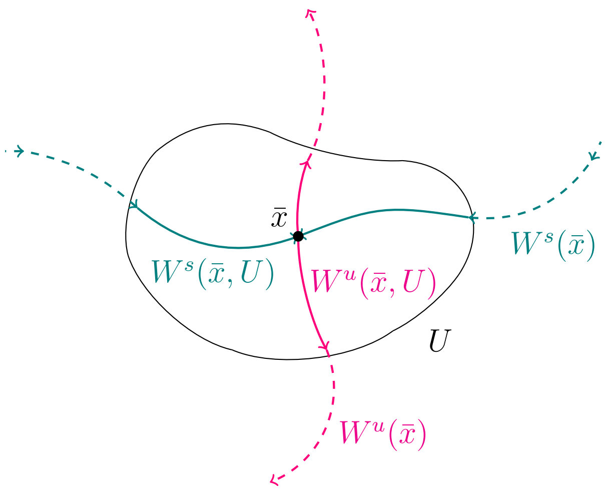

Figure 1.2: [13] Stable and unstable manifolds, local and global

Definition 1.6 (Pseudoperiodic)**.** [26] Let $n \in \mathbb{N}$. A set $\{x_k \mid k \in \{0, \ldots, n\}\}$ is called *ϵ-pseudoperiodic* if for any k, $\quad d(x_{k \bmod n},\ x_{k+1 \bmod n}) < \epsilon$.

As the name suggests, an ϵ-pseudoperiodic orbit is "almost" periodic inthe sense that it represents a "small" perturbation of a theoretically periodic orbit. In practice, such direct orbits may not be known, but it will preresent a naturally useful definition in our approximations.

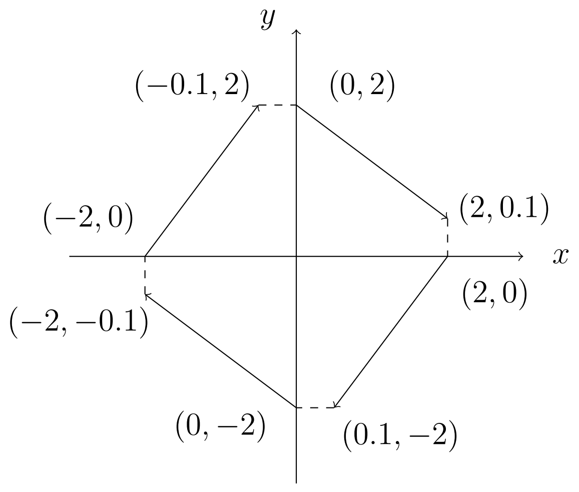

Figure 1.3: [26] A 0.1-pseudoperiodic orbit of the map $f(x, y) = (y,\ 0.05\,(1 - x^2)\,y - x)$

Definition 1.7 (Chain Recurrent)**.** [26] The point $\bar{x} \in M$ is called *chain recurrent* if for any $\epsilon > 0$ there exists an ϵ-pseudoperiodic orbit. The *chain recurrent set* $R_M(f)$ is the set of all chain recurrent points in M.

As shown in [11] we have the inclusion $R_M(f) \subset \bigcap_{k \geq 0} f^k(M)$, which also shows that $R_M(f)$ is an invariant set.

Finally, we give a set of measure-theoretical definitions for types of measures we wish to approximate.

Since our goal is to partition the manifold into a finite set of boxes, we must accept some amount of ”uncertainty” in how our sets look, and how *exactly* f maps such a set. We describe this noise using a stochastic transition function.

Definition 1.8 (Transition Function)**.** [10] Let $\mathcal{A}$ be a σ-agebra on M. A function $p : M \times \mathcal{A} \to [0, 1]$ is called *transition function* if

1. $p(\cdot, A) : M \to [0, 1]$ is measurable for all $A \in \mathcal{A}$,
2. $p(x, \cdot) : \mathcal{A} \to [0, 1]$ is a probability measure for all $x \in M$.

Example 1.9.

- [10] We can model the deterministic system using the dirac measure $p(x, A) = \delta_{f(x)}(A)$.
- The approximate box version of the system can be modelled as using a uniform probability density: for a point x let $\mathcal{B}(x) = \{B \in \mathcal{B} \mid x \in B\}$ be the (singleton) box set containing x. The define p as

$$p(x, A) = \frac{\mathcal{L}(\ A \cap f(\mathcal{B}(x))\ )}{\mathcal{L}(\ f(\mathcal{B}(x))\ )} \tag{1.8}$$

where $\mathcal{L}$ represents the d-dimensional Lebesque measure.

Definition 1.10 (Perron-Frobenius Operator, Invariant Measure)**.** [10] Let p be a transition function, and μ a measure on M. We define the *Perron-Frobenius operator* as

$$(P\mu)(A) = \int p(x,\ A)\ d\mu(x) \tag{1.9}$$

A measure μ is called *invariant* if it is a fixed point of P.

Remark. The Perron-Frobenius operator is often also called *transfer operator.*

Example 1.11. [10] We calculate

$$(P\mu)(A) = \int \delta_{f(x)}(A)\ d\mu(x) = \int \chi_A(f(x))\ d\mu(x) = \mu \circ f^{-1}(A). \tag{1.10}$$

In this case P simply becomes the *pushforward operator.*

An invariant measure can be used to understand the global behavior of a dynamical system, with more μ-mass assigned to regions which are visited frequently over long trajectories, and less μ-mass to regions visited less frequently.

Our "noisy" approximated system poses the benefit that while deterministic dynamical systems generally support the existence of multiple invariant measures, the stochastic system will (if the transition function has a strictly positive density for each $x \in M$) have a unique invariant measure, as shown in [18].

Definition 1.12 (Almost Invariant)**.** [10] A set $A \subset M$ is called *δ-almost invariant* with respect to μ if $\mu(A) \neq 0$ and

$$\int_A p(x, A)\ d\mu(x) = \delta\mu(A) \tag{1.11}$$

Example 1.13. [10] Consider the deterministic case $p(x, \cdot) = \delta_{f(x)}$. Then using the same calcuation as Ex. 1.11 we get

$$\delta = \frac{1}{\mu(A)} \int_A p(x, A)\ d\mu(x) = \frac{\mu(A \cap f^{-1}(A))}{\mu(A)}. \tag{1.12}$$

Theorem 1.14 (Hahn-decomposition)**.** [2] Let $(M, \mathcal{A}, \mu)$ be a measure space on M, where μ is any signed measure. Then there exists $E^+ \in \mathcal{A}$ with

$$\mu(E \cap E^+) \geq 0, \quad \mu(E \setminus E^+) \leq 0 \quad \text{for all } E \in \mathcal{A}\,. \tag{1.13}$$

In particular $\mu(E^+) \geq 0$ and $\mu(M \setminus E^+) \leq 0$.

A fixed point - or eigenmeasure with eigenvalue 1 - is not the only object of interest when considering the operator P. Suppose instead we have a finite (complex valued) measure with $P\nu = \lambda\nu$. Then we must have

$$\lambda\nu(M) = P\nu(M) = \int p(x, M)\ d\nu(x) = \int 1\ d\nu(x) = \nu(M). \tag{1.14}$$

Hence either $\lambda = 1$ or $\nu(M) = 0$. Now assume $\lambda \neq 0$, and that ν is scaled such that $|\nu|$ is a probability measure. Then by the Hahn decomposition we have two sets E^+ and $E^- = M \setminus E^+$ which partition M into positive and negative "regions" of M. From σ-additivity we have

$$0 = \nu(M) = \nu(E^+ \cup E^-) = \nu(E^+) + \nu(E^-), \tag{1.15}$$

hence $\nu(E^+) = -\nu(E^-)$. This means $|\nu|(E^+) = |\nu|(E^-) = \frac{1}{2}$.

Theorem 1.15 (Almost Invariant Decomposition)**.** [10] Suppose ν is scaled such that $|\nu|$ is a probability measure, and $\nu(E^+) = \frac{1}{2}$. Then $\nu = |\nu|$ over E^+ and

$$\delta + \sigma = \lambda + 1 \tag{1.16}$$

if E^+ is δ-almost invariant and $E^- = M \setminus E^+$ is σ-almost invariant.

Example 1.16. Finally, we can consider the deterministic case with $\lambda \approx -1$. Then the sets E^+ and E^- satisfy

$$\delta = \nu(E^+ \cap f^{-1}(E^+)) \,/\, \nu(E^+) \ \geq 0, \tag{1.17}$$

$$\sigma = \nu(E^- \cap f^{-1}(E^-)) \,/\, \nu(E^-) \ \geq 0, \tag{1.18}$$

$$\delta + \sigma = \lambda + 1 \approx 0. \tag{1.19}$$

This implies that nearly all the ν-mass of E^+ gets transported to E^-, and vice versa. We call this an *almost invariant two-cycle.*

2 Algorithms

2.1 Relative Attractor

The construction of a fundamental neighborhood U for a global attractor A is relatively difficult, but the description Eq. 1.4 lends to a natural *ansatz* for its approximation using a compact subdomain $Q \subset M$.

Definition 2.1 (Relative Global Attractor)**.** Let Q be compact. Then we define the *attractor relative to A* as

$$A_Q = \bigcap_{k \geq 0} f^k(Q) \tag{2.1}$$

Remark. It follows from the definition that the relative global attractor is a subset of the global attractor.

The idea to approximate the relative gloabl attractor is in two steps: first, we subdivide each of the boxes and second, discard all those boxes which do not intersect with the previous box set. The algorithm requires a map f, a box set $\mathcal{B}$, and a predefined number of steps n.

Algorithm 1 Relative Attractor

```
1: $\mathcal{B}_0 \leftarrow \mathcal{B}$
2: for $i = \{1, \ldots, n\}$ do
3:     $\mathcal{B}_i \leftarrow$ SUBDIVIDE( $\mathcal{B}_{i-1}$ )
4:     $\mathcal{B}_i \leftarrow \mathcal{B}_i \cap f(\mathcal{B}_i)$
5: return $\mathcal{B}_n$
```

Remark.

- Optionally, the set $\{\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_n\}$ can be returned instead. This will be true for all algorithms of this type.
- The precise technique for subdivision can be tuned depending on the situation. In `GAIO.jl`, boxes are bisected evenly along one dimension $k \in \{1, \ldots, d\}$. The dimension k along which to bisect is cycled through during the steps.

Proposition 2.2. [11, 9] Set $Q_i = \bigcup_{B \in \mathcal{B}_i} B, \quad Q_\infty = \bigcap_{n \geq 1} Q_n$. For all i we have

1. $Q_{i+1} \subset Q_i$
2. $A_Q \subset Q_i$
3. $A_Q = Q_\infty$

In particular, this shows that Q_∞ is backward-invariant.

2.2 Unstable Manifold

From the definition of the local unstable manifold $W^u(\bar{x}, U)$ we see that the relative gloabl attractor $R_Q(f)$ contains the local unstable manifold, and, provided the set Q is sufficiently small, $W^u(\bar{x}, U)$ coincides with $R_Q(f)$. For further details, see [8, 27]. Using this knowledge, we can approximate the global unstable manifold $W^u(\bar{x})$:

1. first, we perform an *initialiation step*: replace the calculation of the local unstable manifold with the calculation of the relative attractor for a small set Q' surrounding the fixed point $\bar{x}$, using Algorithm 1.

Algorithm 2 Continuation Step

```
1: $\mathcal{B}_0 \leftarrow \mathcal{B}$
2: $\mathcal{B}_1 \leftarrow \mathcal{B}$
3: while $\mathcal{B}_1 \neq \emptyset$ do
4:     $\mathcal{B}_1 \leftarrow f(\mathcal{B}_1)$
5:     $\mathcal{B}_1 \leftarrow \mathcal{B}_1 \setminus \mathcal{B}_0$
6:     $\mathcal{B}_0 \leftarrow \mathcal{B}_1 \cup \mathcal{B}_0$
7: return $\mathcal{B}_0$
```

2. Second, we repeat the following *continuation step*: map the current box set forward one iteration, and note any new boxes which are hit. These new boxes get added to the box set. Repeat until there are no new boxes added to the set.

Proposition 2.3. [8] The algorithm in general cannot guarantee covering of the entire unstable manifold, nor can it guarantee covering of the entirety of $W^u(\bar{x}) \cap Q$. This is because $W^u(\bar{x})$ could in theory exit Q, but return at another point. The algorithm can however guarantee covering of the connected component of $W^u(\bar{x}) \cap Q$ which contains $\bar{x}$.

2.3 Chain Recurrent Set

The algorithm to compute the chain recurrent set is first due to [25]. The idea is to construct a directed graph whose vertices are the box set $\mathcal{B}$, and for which edges are drawn from B_1 to B_2 if f maps any part of B_1 into B_2. Call this graph $\textsc{graph}(f,\ \mathcal{B})$, and call $\bar{d}$ the maximum *diameter* of a box in our partition, ie $\bar{d} = \max_{\substack{B \in \mathcal{P},\\ x,y \in B}} d(x, y)$.

We can now ask for a subset of the vertices, for which each vertex is part of a directed cycle. We can equivalently characterize this set as follows:

Definition 2.4 (Strongly Connected)**.** For a directed graph $G = (V, E)$, a subset $H \subset V$ of vertices is called *strongly connected* if for all $u,\ v\ \in\ H$ there exist paths in both directions between v and u. Denote by $\textsc{scc}(G, v)$ the strongly connected subgraph which includes v, and by $\textsc{sccs}(G) = \bigcup_{v \in V} \textsc{scc}(G, v)$ the subgraph induced by union of strongly connected components.

Algorithm 3 Chain Recurrrent Set

```
1: $\mathcal{B}_0 \leftarrow \mathcal{B}$
2: for $i = \{1,\ \dots,\ n\}$ do
3:     $\mathcal{B}_i \leftarrow \textsc{subdivide}(\mathcal{B}_{i-1})$
4:     $G \leftarrow \textsc{graph}(f,\ \mathcal{B}_{i-1})$
5:     $\mathcal{B}_i \leftarrow \textsc{sccs}(G)$
6: return $\mathcal{B}_n$
```

Proposition 2.5. A cycle $\{B_0,\ B_1,\ \dots,\ B_{n-1}\}$ exists in G of and only if there exists an ϵ-pseudoperiodic orbit $\{x_0,\ x_1,\ \dots,\ x_{n-1}\}$ with $x_i \in B_i$. In particular, the vertices of G form a covering of the chain recurrent set with Hausdorff distance at most $\bar{d}$.

Proof. Consider a cycle in G. Then by construction, for each edge (B_i, B_{i+1}) in the cycle we can find a point $x_i \in B_i$ which gets mapped to B_{i+1}. Doing this for all edges we get a set $\{x_0,\ x_1,\ \dots,\ x_{n-1}\}$ which satisfies $d(x_{i \bmod n},\ x_{i+1 \bmod n}) \leq \bar{d}$, ie a $\bar{d}$-pseudoperiodic orbit.

Conversely, consider an ϵ-pseudoperiodic orbit $\{x_0,\ x_1,\ \dots,\ x_{n-1}\}$, $\epsilon \leq \bar{d}$. Then each x_i is in a box B_i, and since $d(x_{i \bmod n},\ x_{i+1 \bmod n}) \leq \bar{d}$, there is an edge (B_i, B_{i+1}) in G. Hence $\{B_0,\ B_1,\ \dots,\ B_{n-1}\}$ is a cycle in G.

□

Corollary 2.6. Proposition 2.2 holds for Algorithm 3. For further details, see [25].

2.4 Invariant Measure

We shift focus to approximating invariant measures for the Perron-Frobenius operator P. For simplicity we will work only with measures absolutely continuous with respect to the Lebesque measure on M, ie measures for which there exists a *kernel* k with

$$p(x, A) = \int_A k(x, y)\, dy. \tag{2.2}$$

In this case we can define $P : L^1 \to L^1$ as

$$(P\phi)(y) = \int \phi(x)\, k(x, y)\, dx. \tag{2.3}$$

Note that ϕ is the density of an invariant measure $\mu_\phi(A) = \int_A \phi(x)\, dx$.

We will use a Galerkin approximation for P which maintains the eigenvalues and cyclic behavior of P. Let $\mathcal{P}$ be a partition of the compact set Q into equally sized, (up to Lebesque null sets) disjoint closed sets. Then we project to a subspace $\chi_\mathcal{P}$ generated by the basis $\{\chi_B \mid B \in \mathcal{P}\}$ of indicator functions on the boxes of our partition. For $\mathcal{P} = \{B_1,\ B_2,\ \dots,\ B_n\}$ we define the matrix

$$P_{ij} = \frac{\mathcal{L}\left(B_j \cap f(B_i)\right)}{\mathcal{L}(B_j)}, \quad i,\, j = 1, \dots, n, \tag{2.4}$$

as well as a linear operator $Q_n P : \chi_\mathcal{P} \to \chi_\mathcal{P}$ as

$$(Q_n P)\, \chi_{B_i} = \sum_{j=1}^{n} P_{ij}\, \chi_{B_j}. \tag{2.5}$$

To realize this approximation, we need to calculate P_{ij}. For this there are two techniques discussed in [11]. Currently the only technique built into `GAIO.jl` is a Monte Carlo approach. Namely, we choose a fixed number r of test points in B_i, and set P_{ij} as the fraction of test points which land in B_j:

Algorithm 4 Invariant Measure

```
1: for $i, j \in \{1, \ldots, n\}$ do
2:     $p \leftarrow$ Choose $\{p_1, \ldots, p_r\}$ randomly from a uniform distribution on $B_i$
3:     $P_{ij} \leftarrow |\ B_j \cap f(p)\ |\ /\ |\ p\ |$
4: $v \leftarrow$ Find a fixed point of $P$
5: $\phi_n \leftarrow \sum_{i=1}^{n} v_i\, \chi_{B_i}$
6: return $\phi_n$
```

Remark.

- In practice the test points are only generated once from the unit cube $[-1, 1]^d$, and then scaled to fit inside the box B_i.
- Line 4 is acheived using the Fortran library ARPACK [19], which has been wrapped in julia by the creators of Arpack.jl [23]. Recently, the underlying algorithm (known as implicitly restarted Arnoldi method) has been implemented in pure julia under the package name ArnoldiMethod.jl [31].

Proposition 2.7. [18] Let $p(x, A)$ be a transition function with globally lipschitz continuous kernel k. Then the operator P is compact. Further, if k is strictly positive, P has a unique fixed point $\phi \in L^1$.

Proposition 2.8. [10, 20, 11, 17, 16] Suppose the transition function $p = p_\epsilon$ converges in the weak* sense to the dirac measure, ie

$$p_\epsilon(x, \cdot) \overset{*}{\rightharpoonup} \delta_{f(x)} \quad \text{as} \quad \epsilon \to 0. \tag{2.6}$$

(An example of this would be Ex. 1.9 for a sequence of partitions $\mathcal{P}_\epsilon$ with diameters $d_\epsilon = \max_{\substack{B \in \mathcal{P}_\epsilon, \\ x, y \in B}} d(x, y) \to 0$). Suppose further that the diffeomorphism f has a hyperbolic attracting set A with and an open set $U \supset A$ such that for the kernels we have

$$k_\epsilon(f(x), y) = 0 \quad \text{if} \quad x \in \overline{U},\ y \notin U. \tag{2.7}$$

Let P_ϵ be the Perron-Frobenius operator for the transition function p_ϵ. Then there exist unique fixed points ϕ_ϵ^n of $Q_n P_\epsilon$, and the sequence of fixed points converge to the fixed point ϕ of the true operator P, ie

$$\phi_\epsilon^n \to \phi \quad \text{as} \quad n \to \infty,\ \epsilon \to 0. \tag{2.8}$$

2.5 Almost Invariant Sets (and Cycles)

Our final algorithm is quite analogous to the algorithm for invariant measures of P. We begin with a theorem about the convergence our discretized operator.

Theorem 2.9. [10] Suppose our transition function p has a kernel k (see Eq. 2.2) which satisfies

$$\iint |k(x,y)|^2 \, dx \, dy < \infty \tag{2.9}$$

Then the Frobenius Perron operator $P : L^2 \to L^2$ is compact. Further $Q_n P$ converges strongly to P in the operator norm on L^2, ie

$$\| Q_n P - P \| \to 0 \quad \text{as} \quad n \to \infty. \tag{2.10}$$

Assume now that $Q_n P$ has an eigenvalue 1 with multiplicity 2. Then by the above theorem, the eigenvalue will split into two simple eigenvalues, one of which will move away from 1 (the other will stay at 1 since $Q_n P$ is column stochastic). If we now find this eigenvector ϕ_n, then (again by the above theorem and the continuity of P) the positive and negative regions of ϕ_n will converge to the sets E^+ and E^- from Theorem 1.15. That is, they are almost invariant.

Algorithm 5 Almost Invariant Sets

```
1: for $i, j \in \{1, \ldots, n\}$ do
2:     $p \leftarrow$ Choose $\{p_1, \ldots, p_r\}$ randomly from a uniform distribution on $B_i$
3:     $P_{ij} \leftarrow |\, B_j \cap f(p) \,| \,/\, |\, p \,|$
4: $v \leftarrow$ Find an eigenvector to the second largest eigenvalue of $P$
5: $E^+ \leftarrow \cup\{B_i \mid v_i \geq 0\}$
6: $E^- \leftarrow \cup\{B_i \mid v_i < 0\}$
7: return $E^+$, $E^-$
```

Remark. The algorithm can also be used to find almost invariant cycles. In this case line 4 simply needs to be changed to find an eigenvector near -1.

3 Julia

3.1 A (Very) Brief Introduction to the Julia Language

The julia language GitHub page decribes julia as ”a high-level, high-performance dynamic language for technical computing” [5]. Its creators come from Matlab, Python, Lisp among others, and they desired a langauge as syntactically simple as Python, as powerful for linear algebra as Matlab, and as fast as C [7]. To achieve this, julia is just-in-time compiled using LLVM to generate native machine code. Further, ”julia uses multiple dispatch as a paradigm, which makes it easy to express many object-oriented and functional programming patterns" [6]. As a brief example of multiple dispatch, consider a simple user defined type `2dBox` (similar to the implementation in `GAIO.jl`):

```julia
struct 2dBox
    center::Tuple{Float64,Float64}
    radius::Tuple{Float64,Float64}
end
```

Then we can create a `2dBox` by calling `b = 2dBox((0.0, 0.0), (1.0, 1.0))`. A natural question to ask is whether a vector, say `x = [0.5, 0.6]`, lies within `b`. To do this, we can *overload* the function `in` from julia base:

```julia
function Base.in(x::Vector, b::2dBox)
    all( b.center .- b.radius .≤ x .≤ b.center .+ b.radius )
end
```

This function uses julia's *dot-syntax* to vectorize operations, eg. `+` by writing `.+`. We can now call the base function `in` or its unicode alias `∈`:

```julia
x = [0.5, 0.6]
b = 2dBox((0.0, 0.0), (1.0, 1.0))

x ∈ b           # returns true
x .- 2 ∈ b      # returns false
x .- 2 ∉ b      # returns true
```

The above example demonstrates the syntactical simplicity of julia. It should be noted that despite this simplicity, julia can generate highly performant machine code. For further illustration, consider the algorithm for the global unstable manifold (see Algorithm 2) implemented in matlab (Listing 1) and in julia (Listing 2). The julia code is one third as long, mirrors the pseudocode much more closely, and still runs faster than the matlab code.

3.2 GAIO.jl

This section is meant as an introduction to the most important concepts in the implementation of `GAIO.jl`. For more information see the respective docstrings, eg. by typing `julia> ? Box` into the julia REPL.

The package is built off of three base structures `Box`, `BoxPartition`, `BoxMap`:

```julia
struct Box{N,T <: AbstractFloat}
    center::SVector{N,T}
```

Listing 1: Unstable manifold algorithm in matlab

```
function gum(t, f, X, depth)
dim = t.dim;
none = 0; ins = 2; expd = 4;              % defining flags
nb0 = 0;  nb1 = t.count(depth);           % bookkeeping the no. of boxes
tic; j = 1;
t.set_flags('all', ins, depth);
while nb1 > nb0                           % while new boxes nonempty
t.change_flags('all', ins, expd);         % mark inserted boxes
b = t.boxes(depth); M = size(b,2);        % get the geometry of the boxes
flags = b(2*dim+1, :);
I = find(bitand(flags,expd));             % find boxes to expand
b = b(:,I); N = size(b,2);
S = whos('X'); l = floor(5e7/S.bytes);
for k = 0:floor(N/l),                     % split in chunks of size l
    K = k*l+1:min((k+1)*l,N);
    c = b(1:dim,K);                       % center ...
    r = b(dim+1:2*dim,1);                 % ... and radii of the boxes
    n = size(c,2); E = ones(n,1);
    P = kron(E,X)*diag(r) + ...           % sample points in all boxes
        kron(c',ones(size(X,1),1));
    t.insert(f(P)', depth, ins, none);    % map sample points, insert boxes
end
t.unset_flags('all', expd);               % unflag recently expanded boxes
nb0 = nb1; nb1 = t.count(depth);
j = j+1;
end
```

Listing 2: Unstable manifold algorithm in julia

```
function unstable_set(F::BoxMap, B::BoxSet)
    B₀ = B
    B₁ = B
    while B₁ ≠ ∅
        B₁ = F(B₁)
        B₁ = B₁ \ B₀
        B₀ = B₁ ∪ B₀
    end
    return B₀
end
```

```julia
    radius::SVector{N,T}
end
```

A Box is a half-open generalized rectangle, that is, a product of half-open intervals $[a_1, b_1) \times [a_2, b_2) \times \ldots \times [a_d, b_d)$. We now need to represent a partition $\mathcal{P}$ of boxes:

```julia
struct BoxPartition{N,T,I<:Integer} <: AbstractBoxPartition{Box{N,T}}
    domain::Box{N,T}
    left::SVector{N,T}
    scale::SVector{N,T}
    dims::SVector{N,I}
    dimsprod::SVector{N,I}
end
```

Instead of keeping track of positions for each individual box, we use integer indices for an equidistant *box grid* and only store the dimensions of the grid using the attribute `dims`. The attributes `left`, `scale`, `dimsprod` are used to quickly calculate box indices from a given point (see `point_to_key`, `key_to_box` in `GAIO.jl`). We further use the index structure to our advantage when storing box sets $\mathcal{B}$, only storing sets of indices corresponding to a partition:

```julia
struct BoxSet{B,P<:AbstractBoxPartition{B},S<:AbstractSet} <: AbstractSet{B}
    partition::P
    set::S
end
```

Finally, we need a way to convert a map f defined on Q to a map `F` defined on boxes of a `BoxPartition`:

```julia
struct SampledBoxMap{A,N,T,F,D,I} <: BoxMap
    map::F
    domain::Box{N,T}
    domain_points::D
    image_points::I
    acceleration::A
end
```

When a `SampledBoxMap` is initialized, we require a function to calculate test points that are mapped by the given function f. This is `domain_points`. These test points are then mapped forward by f, and the boxes which are hit become the image set. More precisely, mapping a box set is done in two main steps within GAIO (see Listing 5):

1. Test points within the box are generated (or retrieved) for the specified box using `domain_points(box_center, box_radius)`. These test points are mapped forward by the given function f.

2. For each mapped test point `fp`, an optional set of "perturbations" are generated using `image_points(fp, box_radius)`. For each of the perturbed points, the index of the box within the `BoxPartition` containing this point is calculated. This index gets added to the image set.

Tha `acceleration` attribute is what we will concern ourselves with for the remainder of the present paper. Naturally, if an increase in accuracy is required, a larger set of test points may be chosen. This leads to a dilemma: the more accurate we wish our approximation `F` to be, the more we need to map very similar test points forward, causing a considerable slow down for complicated dynamical systems. However, the process of mapping each test point forward is completely independent on other test points. This means we do not need to perform each calculation sequentially; we can *parallelize.*

4 Parallelization using the CPU

4.1 CPU Architechture

When referring to CPU architecture one typically either means *instruction set architecture (ISA)* when referring to a range of CPUs, or the *microarchitechture* of a specific CPU model. The ISA is the blueprint for a set of abstract characteristics such as supported instructions, data types, register count, etc. The microarchitechture is the *implementation* of this blueprint for a CPU. We restrict to the discussion of the microarchitecture for the Intel 3rd gen (Ivy Lake) line of processors, though the level of detail is low enough that the statements will apply to most modern CPUs built on the x86 64-bit ISA.

A processor microarchitecture can be further split to 3 components:

- The front end, which consists of instruction cache and instruction fetch / decode units. This is responsible for fetching batches of instructions from memory, storing the batches in the instruction cache, and decoding the instructions into a set of *micro-operations, or μOps.*

- The back end (or execution engine), which includes the reorder buffer, unified scheduler (also called reservation station), and various execution ports. Since not all instructions are necessarily dependent on one-another, they often be reordered, or multiple instructions can be executed simultaneously. This is known as *out-of-order execution.* The reorder buffer stores the the *order* of μOps until they are retired. The scheduler takes μOps and dispatches them to the various ports, each of which is specialized for a subset of instructions.

- The memory system. This includes (typically) three cache levels: L1, L2, and L3. The L1 cache is faster than L2, but has a lower storage capacity. The same holds for L2 vs L3 cache.

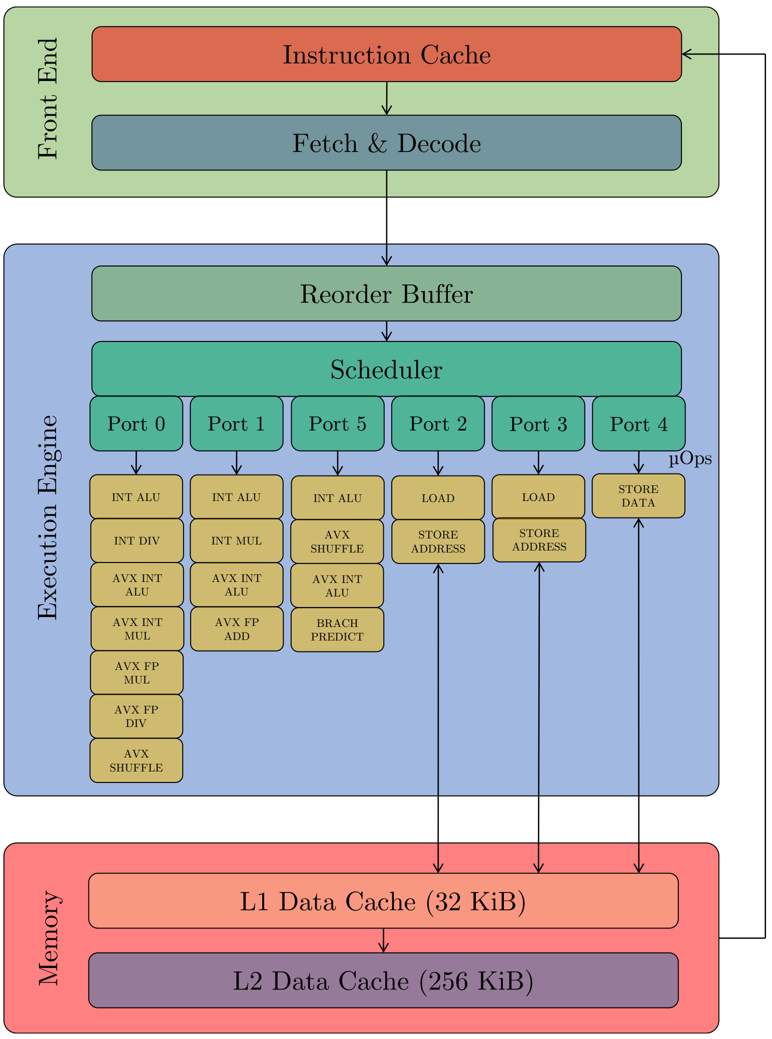

Figure 4.1: CPU microarchitecture (figure inspiration [22, 1])

All three can be limiting factors in a computation, though we will mainly consider optimizations for the back end. Additionally, a modern processor will copy this basic structure multiple times, each copy is referred to as a *thread*.

The naivest but most practically difficult technique to increase performance is to utilize the whole instruction set available in the acrhitecture. Obviously it can't be expected that the programmer optimizes for every microarchitecture and every instruction; if that were true we would all be writing pure assembly language. But there are some easy changes one can make, primarily using *fused-multiply-add (FMA)* instructions. An example from [22]: consider the simple dynamical system over $\mathbb{R}$:

$$x_{k+1} = x_k^2 + p \tag{4.1}$$

for some p. To perform one iteration, the CPU needs to call `MUL` once and then `ADD` once, both of which have latency of roughly 5 clock cycles. Using `FMA`, this can be performed in one operation, nearly doubling performance. A consequence of this is that code which does not harness FMA instructions can harness at most half of the peak theoretical power of the CPU.

In julia, this can be performed with the `@muladd` macro from `MuladdMacro.jl` [28], which automatically converts all combinations of multiply and add into calls to the inbuilt `muladd` function.

Conversely, a conceptually difficult but practically easy technique to implement is multithreading. Multithreading is more conceptually difficult because the process of creating, scheduling and synchronizing processes over multiple cores is relatively complicated. However, due to the obvious benefit offered by multithreaded programming, mature libraries have developed to hide these complications under a layer of abstraction. In julia, examples include the `Threads` module from julia base, as well as the package `Transducers.jl` [4] which offers a backend for packages such as `FLoops.jl` [3]. Both options offer macros - `@threads` and `@floop` respectively - which can simply be appended to `for` loops to automatically execute them over multiple threads.

Our main technique for parallelization on the CPU is the *Advanced Vector eXtension (AVX)*. The philosophy of AVX was initially characterized in [12] as *Single Instruction Multiple Data (SIMD)*. As the name would suggest: when a single instruction, eg. `ADD`, needs to be applied to multiple floating point numbers, then one could pack 4 double precision or 8 single precision floating point numbers into a single *vector* stored into the YMM (vector) register. This ic alled a *gather* operation. Two of these vectors can then be passed into port 1 to the AVX floating point arithmetic-logic-unit (`AVX FP ADD` in Fig. 4.1), and added all at once. This reduces the execution time from 20 clock cycles (4 double precision add operations at 5 clock cycles each) to just 5 clock cycles (1 packed add operation at 5 clock cycles).

The majority of modern compilers can automatically convert simple loops and vectorized functions into SIMD instructions. This however is not gauranteed, as the compiler neeeds to first prove that there are no data dependencies. Hence to reach peak CPU performance, it is often required to manually vectorize.

4.2 Implementation in GAIO.jl

We wish to map an array of test points $x = (x_1,\ x_2,\ \dots,\ x_n),\ \ x_i \in \mathbb{R}^d$ forward, with as much parallelism as possible. For example, consider for $d = 3$ and a box $[0,1]^3$ the test points:

```julia
x = [   # each tuple is seen as a point in 3d space
    (1., 0., 0.),
    (0., 1., 0.),
    (0., 0., 1.),
        (0., 0., 0.)
        # etc ...
    ]
```

We could equivalently characterize this array as an array of packed floats in the ”packed” space $(\mathbb{R}^4)^3$:

```
    x = [   # three 4xFloat64<...> are seen as a point in our "packed" 3d space
        (
            4xFloat64<1., 0., 0., 0.>,
            4xFloat64<0., 1., 0., 0.>,
            4xFloat64<0., 0., 1., 0.>
        )
        # etc ...
    ]
```

The packed SIMD vectors can be constructed using the `Vec` type from `SIMD.jl` [29], a convenience wrapper around julia's base SIMD vector type (which itself is just special tuple). So our goal becomes managing memory careully to (efficiently) convert from one vector of vectors to a vector of packed vectors, and vice versa.

A convenient fact about the `Tuple` type in julia is that if its elements are ”bits” types (that is, they can be represented by a string of bits that can be *stack-allocated*), then the tuple is stored *contiguously* in memory. For eg. numeric types like `Float32`, this means that the exact positions in memory of each element can be deduced from the memory positions of the tuple. Julia provides the convenience function `reinterpret` which changes the type interpretation of a block of memory.

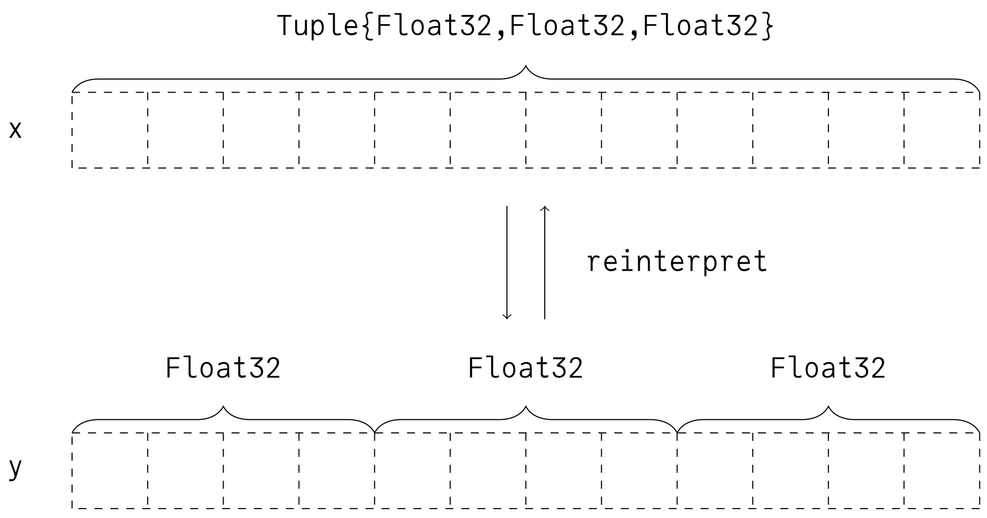

Figure 4.2: Illustration of julia's `reinterpret` function, where each square is 1 byte

Using `reinterpret`, we can change the problem to reordering the indices of x. Hence consider the vector i of indices of x:

$$i = (\ \underbrace{1,\ 2,\ 3,}_{\text{first point}}\quad \underbrace{4,\ 5,\ 6,}_{\text{second point}}\quad \ldots,\ \underbrace{3n-2,\ 3n-1,\ 3n}_{\text{nth point}}\ ) \tag{4.2}$$

We wish to permute i such that each group of four points has its respective elements stored contiguously. Once this is done, we can call `reinterpret` again to convert the type representation to a vector of packed tuples. In particular we therefore require that the number of points n is divisible by 4.

$$S(i) = (\ \underbrace{\underbrace{1,\ 4,\ 7,\ 10,}_{\text{first elements}}\quad \underbrace{2,\ 5,\ 8,\ 11,}_{\text{second elements}}\quad \underbrace{3,\ 6,\ 9,\ 12,}_{\text{third elements}}}_{\text{first packed point}}\ \ldots,\ 3n-6,\ 3n-3,\ 3n\ ) \tag{4.3}$$

We generalize this permutation S to arbitrary dimension d, SIMD vector length s and vector of points x with length n in Listing 3.

This function provides the fundamental technique for parallelization on the CPU. We first gather the points, then pass the gathered points to the function f. However, we still need to convert the packed mapped points back to single points for use with other functions afterward. This is the inverse of a gather operation, and is called *scatter* (see Listing 4).

We now have the tool set to accelerate the function `map_boxes` using SIMD. For the implementation, see Listing 6.

It is important to note that not all elementary instructions have SIMD equivalents. Hence some more complicated functions cannot be parallelized in this way. Since GAIO.jl only controls test points, box partitions, etc., *the efficient implementation of the function f is solely up to the user* .

4.3 Results

5 Parallelization using the GPU

5.1 GPU Architechture

While parallelizing code specifically for a CPU, one wishes to increase the amount of instruction-level-parallelism *per thread.* The CPU is built to perform tasks which are as large as possible, and with a little latency as possible, in a serial fashion. Latency is hidden on the CPU via small instruction registers but large low-latency on-chip caches and by out-of-order execution. However, one could alternatively approach the problem of increasing parallelism and hiding latency by increasing the number of threads entirely. This is the core core philosophy behind *massively parallel computing* - the philosophy

Listing 3: Conversion function to packed tuples

```julia
function tuple_vgather(x::Vector{NTuple{d,T}}, s) where {d,T}
    # x is a vector of tuples, each of length d and datatype T

    n = length(x)
    m = n ÷ s
    if n != m * s
        throw(DimensionMismatch("length of input not divisible by simd"))
    end

    # Change type interpretation of x's memory
    vr = reinterpret(T, x)

    # Initialize the vector of packed tuples
    vo = Vector{NTuple{d,Vec{s,T}}}(undef, m)

    # The indices that form the first element of the first packed vector
    idx = Vec(ntuple( i -> (i-1) * d, s ))

    # Grab the indices of the first element of the i-th packed vector,
    # then jump by d*s to grab the indices of the second element, and so on
    for i in 1:m
        vo[i] = ntuple( j -> vr[idx + (i-1) * d * s + j], d )
    end

    return vo
end
```

Listing 4: Conversion back to single tuples

```julia
function tuple_vscatter(y::Vector{NTuple{d,Vec{s,T}}}) where {s,d,T}
    # y is a vector of packed tuples, tuples of SIMD Vecs

    # Initialize the unpacked vector
    vo = Vector{NTuple{d,T}}(undef, s * length(y))

    # Create a view of vo which is made of individual numbers of type T
    vr = reinterpret(T, vo)

    # The indices that form the first element of the first packed vector
    idx = Vec(ntuple( i -> d * (i-1), s ))

    # set the values of vr as the permuted values of y
    for i in 1:d, j in 1:length(vi)
        vr[idx + (j-1) * d * s + i] = y[i + (j-1) * d]
    end

    return vo
end
```

adopted in the GPU. The GPU follows a throughput-oriented design in which as many tasks as possible are assigned to different threads concurrently. Less focus is given to low-latency memory access and operations, and more focus is given to launching many separate threads.

To manage this amount of parallel work, the components of a GPU are split into a hierarchy:

- A single processing unit is known as a thread. A single consumer-level GPU can have on the order of tens of thousands of threads (Compare this with the threadcount of a typical consumer-level CPU of 16 or 32 threads).
- Threads are grouped into warps. Each warp consists of 32 threads and a *warp scheduler*. This is the most granular level of scheduling that a GPU has. In particular this means that every thread in a warp performs a single instruction on different data. If a SIMD set (a set of data for which one operation should occur) is larger than the size of a warp, then the whole warp is launched multiple times.
- Threads are further grouped into (variably sized) blocks. Blocks are used by the next level in the hierarchy (SM) to manage execution over the GPU.
- Finally, blocks are grouped into streaming multiprocessors or SMs. Each SM contains a set of blocks as well as an instruction register and a scheduler. Instructions and data which get sent to the GPU are split and stored in the SMs, and each SM can then schedule tasks to the blocks that it controls.

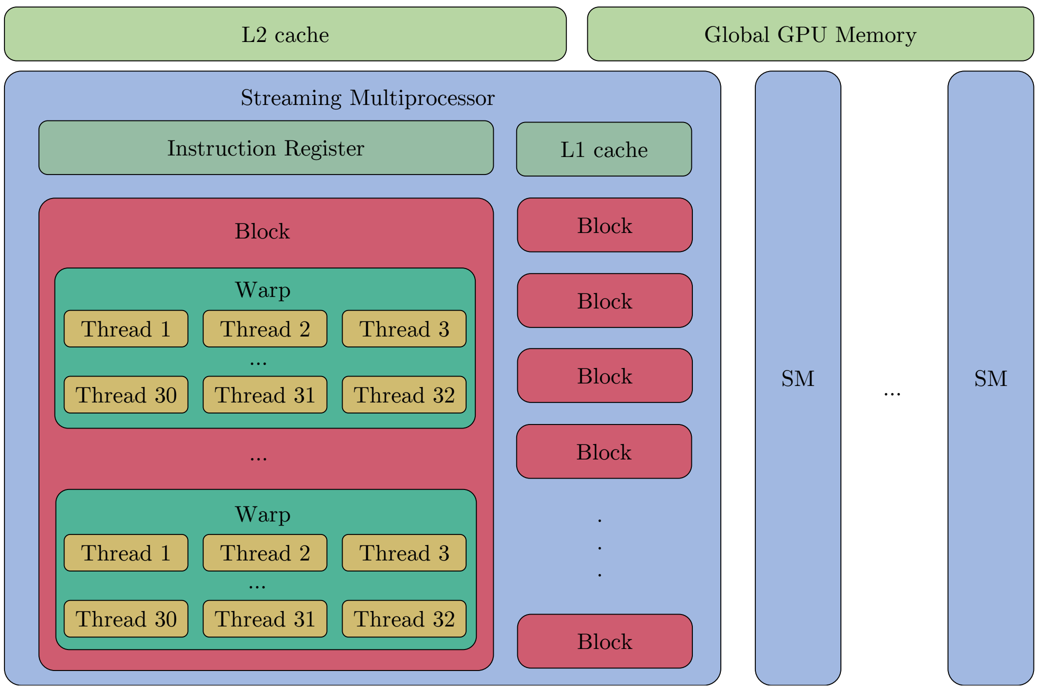

Figure 5.1: GPU microarchitecture (figure inspiration [22])

This hierarchical structure simplifies the task of handling such a massive number of threads since only one control unit is required per warp, which leaves more space for packing additional threads onto the chip. Further, the large instruction register means that each SM can hold many instructions and dispatch them to eligible warps before stalling due to a data request [22]. A warp may of course need to access the slower L1 or L2 caches for data, but while that is happening other warps are already performing different operations, so that - in effect - this memory access latency is hidden.

An important consideration when launching computations on the GPU (known as *kernels*) is the block size. Since all threads in a SM must share the instruction register, increasing the block size (and therefore the number of threads controlled by the SM) means that there will necessarily be less register slots per thread. Conversely, decreasing block size will increase available register slots per thread, but will mean less threads total can be launched by the SM, since each SM has a maximal number of residing threads determined by the hardware.

Definition 5.1. We call the ratio of the number of residing threads n in a SM vs the theoretical maximum number of threads per SM allowed by the hardware N the *occupancy* O, that is, $O = n/N$.

Example 5.2. Nvidia's Ampere architecture supports a maximum of $N = 2048$ threads per SM, each with 65536 register slots and maximal block count of 32. Supported block sizes range from 1 to 1024, and a maximum of 255 register slots per thread [24].

- If we choose a block size of 32 to match the size of a warp, we get the maximum number of residing threads $n = 32{\cdot}32 = 1024$ and our occupancy is $O = n/N = 50\%$. Each thread has access to $65536/1024 = 64$ register slots.
- If we instead choose a block size of 64, then we reach full occupancy, but each thread now only has access to 32 register slots.

The precise ”best” balance between occupancy and registers is highly problem dependent, and there are many other hardware-based factors which affect the true achievable occupancy. This makes optimal GPU performance not always feasible when writing generic code. However, Nvidia provides an official occupancy calculator API which suggests appropriate thread and block counts to approximately maximize occupancy under register size constraints.

5.2 Implementaion in GAIO.jl

The technique for parallelization via the GPU is quite natural: one thread is responsible for one test point. Initially, a more involved approach was tested, though this was replaced by a ”naive” approach, as described in [30, 22]. The choice was made under the motto ”keep it simple, stupid”.

With the exception of Apple’s ”unified” desktop system-on-chip design, memory for the CPU and GPU are kept separate. Hence if a computation is to be launched on the GPU, the respective instructions and data must first be transferred from the system RAM via a PCIe bus to the GPU’s internal memory (VRAM). Transferring data via a PCIe bus is much slower than transferring data within a chip (eg. from the CPU’s L1/L2 cache to instruction registers). Hence it is ideal to reduce necessary transfers and perform as much of a computation solely on the GPU or CPU.

In order to reduce transfers, a global set of test points is calculated when the `SampledBoxMap` is initialized. These points are normalized to the unit cirle w.r.t. $||\cdot||_\infty$, that is, the unit cube $[-1, 1]^d$. The global test points are transferred only once to the GPU’s VRAM at initialization, and then each point p is appropriately rescaled using `@muladd p .* r .+ c` where r and c are the box’s center and radius, respectively. This way, only the information about the map f, the partition $\mathcal{P}$, and the indices of the boxes to be mapped must be transferred to the GPU at runtime. Each thread calculates one index for the box containing the thread’s mapped point, and writes this index to an array. Finally, the array is transferred to the CPU and reduced to become the image set. The implementation of the kernel is shown in Listing 7, and the GPU accelerated `map_boxes` function in Listing 8.

As with the CPU kernel, there are some factors only in control fo the user. Modern GPUs are primarily designed for *single-precision* operations, and are correspondingly much less efficient using double precision arithmetic. GAIO.jl does its best to convert all objects such as partitions and box sets to single precision, but again *the efficient implementation of the function f is solely up to the user* .

5.3 Results

6 Appendix

Listing 5: Function to calculate $f(\mathcal{B})$

```julia
function map_boxes(g::SampledBoxMap, source::BoxSet)
    P = source.partition

    # Use multithreaded iteration over the box set
    @floop for box in source
        c, r = box.center, box.radius
        domain_points = g.domain_points(c, r)

        # Map each test point
        for p in domain_points
            fp = g.map(p)
            hitbox = point_to_box(P, fp)

            # Skip to next iteration if `fp` lands outside `P`
            isnothing(hitbox) && continue

            # Add perturbations to fp
            r = hitbox.radius
            image_points = g.image_points(c, r)
            for ip in image_points
                hit = point_to_key(P, ip)
                isnothing(hit) && continue

                # Use @reduce syntax to initialize an empty
                # key set and add hits during iteration
                @reduce(image = union!(Set{keytype(P)}(), hit))
            end
        end
    end
    return BoxSet(P, image)
end
```

Listing 6: Function to calculate $f(\mathcal{B})$ with CPU acceleration

```julia
function map_boxes(
        g::SampledBoxMap{C,N}, source::BoxSet
    ) where {simd,C<:BoxMapCPUCache{simd},N}

    # Preallocated temporary storage objects.
    # `temp_vec` and `temp_points` are reinterprets of
    # the same array, `temp_points` is an array of tuples
    idx_base, temp_vec, temp_points = g.acceleration

    P = source.partition
    @floop for box in source

```

```
        # Grab a separate section of the
        # temporary storage for each thread
        tid  = (threadid() - 1) * simd
        idx  = idx_base + tid * N
        mapped_points = @view temp_points[tid+1:tid+simd]

        domain_points = g.domain_points(box.center, box.radius)
        for p in domain_points
            fp = g.map(p)

            # Scatter packed `fp` into temporary storage
            tuple_vscatter!(temp_vec, fp, idx)

            # continue as with normal `map_boxes`
            for q in mapped_points
                hitbox = point_to_box(P, q)
                isnothing(hitbox) && continue
                image_points = g.image_points(q,  hitbox.radius)
                for ip in image_points
                    hit = point_to_key(P, ip)
                    isnothing(hit) && continue
                    @reduce(image = union!(Set{keytype(P)}(), hit))
                end
            end
        end
    end
    return BoxSet(P, image)
end
```

Listing 7: GPU kernel to calculate $f(\mathcal{B})$

```
function map_boxes_kernel(
        f,          # Function
        keys,       # Keys to be mapped
        points,     # "Global" test points
        out_keys,   # Array to hold mapped box indices ...
        P,          # ... wrt the partition P
    )

    # Calculate the index of the GPU thread
    ind = (blockIdx().x - 1) * blockDim().x + threadIdx().x - 1
    stride = gridDim().x * blockDim().x
    nk, np = length(keys), length(points)
    len = nk * np - 1

    # In case `len` is too large, launch the warp repeatedly
    for i in ind : stride : len

        # Loop over the test points, then the box index
        m, n = divrem(i, np) .+ 1
        p    = points[m]
```

```
            key  = keys[n]
            box  = key_to_box(P, key)
            c, r = box.center, box.radius

            # Map the point forward and record the result
            fp   = f(@muladd p .* r .+ c)
            hit  = point_to_key(P, fp)

            # If `fp` lands outside the partition, assign it the
            # out-of-bounds index 0. This will later be discarded
            out_keys[i+1] = isnothing(hit) ? hit : 0

        end
    end
```

Listing 8: Function to calculate $f(\mathcal{B})$ with GPU acceleration

```
function map_boxes(
        g::SampledBoxMap{C}, source::BoxSet
    ) where {SZ,C<:BoxMapGPUCache{SZ}}
    # BoxMapGPUCache holds a max size `SZ` of box indices
    # which can fit in GPU memory

    # We use `Stateful` to iterate through
    # test points `SZ` points at a time
    P, keys = source.partition, Stateful(source.set)

    # Normalized test points
    points = g.domain_points(P.domain.center, P.domain.radius)

    image = Set{keytype(P)}()
    while !isnothing(keys.nextvalstate)
        stride = min(SZ, length(keys))

        # Allocate input/output indices on GPU
        in_keys = CuArray{Int32,1}(collect(Int32, take(keys, stride)))
        out_keys = CuArray{Int32,1}(undef, length(in_keys) * length(points))

        # Launch GPU kernel and wait until task is comlete
        launch_kernel_then_sync!(
            map_boxes_kernel!,
            g.map, in_keys, points, out_keys, P
        )

        # Transfer output indices to CPU memory
        out_cpu = Array{Int32,1}(out_keys)
        union!(image, out_cpu)

        # Manually deallocate GPU arrays since julia has
        # more difficulty garbage collecting GPU memory
        CUDA.unsafe_free!(in_keys); CUDA.unsafe_free!(out_keys)
    end

    # remove out-of-bounds index
    delete!(image, 0i32)

    return BoxSet(P, image)
end
```

References

[1] Andreas Abel and Jan Reineke. “uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Apr. 2019. DOI: 10.1145/3297858.3304062. URL: https://doi.org/10.1145%2F3297858.3304062.

[2] Hans Wilhelm Alt. *Linear Functional Analysis*. Springer Berlin, 2016. ISBN: 978-1-4471-7279-6. DOI: 10.1007/978-1-4471-7280-2.

[3] Takafumi Arakaki et al. *FLoops,jl*. https://github.com/JuliaFolds/FLoops.jl.git. 2022.

[4] Takafumi Arakaki et al. *Transducers.jl*. https://github.com/JuliaFolds/Transducers.jl.git. 2022.

[5] Jeff Bezanson et al. *julialang*. https://github.com/JuliaLang/julia.git. 2022.

[6] Jeff Bezanson et al. *julialang.org*. https://julialang.org/. 2022.

[7] Jeff Bezanson et al. “Julia: A fast dynamic language for technical computing.” In: *arXiv* (2012). DOI: https://doi.org/10.48550/arXiv.1209.5145.

[8] Michael Dellnitz and Adreas Hohmann. “The Computation of Unstable Manifolds Using Subdivision”. In: *Nonlinear Systems and Chaos*. Ed. by Haim Brezis. Vol. 19. Progress in Nonlinear Differential Equations and their Applications. 1996, pp. 449–459. DOI: https://doi.org/10.1007/978-3-0348-7518-9.

[9] Michael Dellnitz and Andreas Hohmann. “A Subdivision Algorithm for the Computation of Unstable Manifolds and Global Attractors”. In: *Numerische Mathematik* 75 (1997). DOI: https://doi.org/10.1007/s002110050240.

[10] Michael Dellnitz and Oliver Junge. “On the Approximation of Complicated Dynamical Behavior”. In: *SIAM Journal on Numerical Analysis* 2.36 (1999).

[11] Michael Dellnitz, Oliver Junge, and Gary Froyland. “The Algorithms Behind GAIO - Set Oriented Numerical Methods for Dynamical Systems”. In: *Ergodic Theory, Analysis, and Efficient Simulations of Dynamical Systems*. Ed. by Bernold Fiedler. Springer Berlin, 2001, pp. 145–174. DOI: https://doi.org/10.1007/3-540-35593-6.

[12] Michael J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE.* Vol. 54. 1966, pp. 1901–1909. DOI: `https://doi.org/10.1109/PROC.1966.5273`.

[13] Andreas Johann. “Nichtlineare Dinamik”. lecture notes. 2021.

[14] Oliver Junge. *GAIO.* `https://github.com/gaioguy/GAIO`. 2020.

[15] Oliver Junge, April Herwig, Lukas Mayrhofer, et al. *GAIO.jl.* `https://github.com/gaioguys/GAIO.jl.git`. 2022.

[16] R. Z. Khas’minskii. “Principle of Averaging for Parabolic and Elliptic Differential Equations and for Markov Processes with Small Diffusion”. In: *Theory of Probability & Its Applications* 8.1 (1963), pp. 1–21. DOI: `10.1137/1108001`. eprint: `https://doi.org/10.1137/1108001`. URL: `https://doi.org/10.1137/1108001`.

[17] Yuri Kifer. *Random Perturbations of Dynamical Systems.* Progress in Probability. Birkhäuser Boston, MA, 1988. DOI: `https://doi.org/10.1007/978-1-4615-8181-9`.

[18] Andrzej Lasota and Michael C. Mackey. *Chaos, Fractals, and Noise. Stochastic Aspects of Dynamics.* Springer New York, NY, 1994. DOI: `https://doi.org/10.1007/978-1-4612-4286-4`.

[19] Richard B. Lehoucq, Danny C. Sorensen, and Chao Yang. *ARPACK Users’ Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, 1998.

[20] Tien-Yien Li. “Finite approximation for the Frobenius-Perron operator. A solution to Ulam’s conjecture”. In: *Journal of Approximation Theroy* 17.2 (1976), pp. 177–186. DOI: `https://doi.org/10.1016/0021-9045(76)90037-X`.

[21] *MATLAB version 9.3.0.713579 (R2017b).* The Mathworks, Inc. Natick, Massachusetts, 2017.

[22] Dániel Nagy, Lambert Plavecz, and Ferenc Hegedűs. *Solving large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs: performance comparisons of MPGOS, ODEINT and DifferentialEquations.jl.* 2020. DOI: `10.48550/ARXIV.2011.01740`. URL: `https://arxiv.org/abs/2011.01740`.

[23] Andreas Noack et al. *Arpack.jl.* `https://github.com/JuliaLinearAlgebra/Arpack.jl.git`. 2021.

[24] *NVIDIA A100 Tensor Core GPU Architecture.* Tech. rep. nvidia, 2020. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`.

[25] George Osipenko. “Construction of Attractors and Filtrations”. In: *Banach Center Publications* 47 (1999).

[26] George Osipenko. *Dynamical Systems, Graphs, and Algorithms.* Springer Berlin, 2007. DOI: `https://doi.org/10.1007/3-540-35593-6`.

[27] Jacob Palis and Wellington Melo. *Geometric Theory of Dynamical Systems*. Springer New York, 1982. DOI: `https://doi.org/10.1007/978-1-4612-5703-5`.

[28] Christopher Rackauckas, David Widmann, et al. *MuladdMacro.jl*. `https://github.com/SciML/MuladdMacro.jl.git`. 2022.

[29] Erik Schnetter, Kristoffer Carlsson, et al. *SIMD.jl*. `https://github.com/eschnett/SIMD.jl.git`. 2022.

[30] Christopher P. Stone, Andrew T. Alferman, and Kyle E. Niemeyer. “Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on GPUs, MICs, and CPUs”. In: *Computer Physics Communications* 226 (2018), pp. 18–29. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2018.01.015`.

[31] Harmen Stoppels et al. *ArnoldiMethod.jl*. `https://github.com/JuliaLinearAlgebra/ArnoldiMethod.jl.git`. 2021.