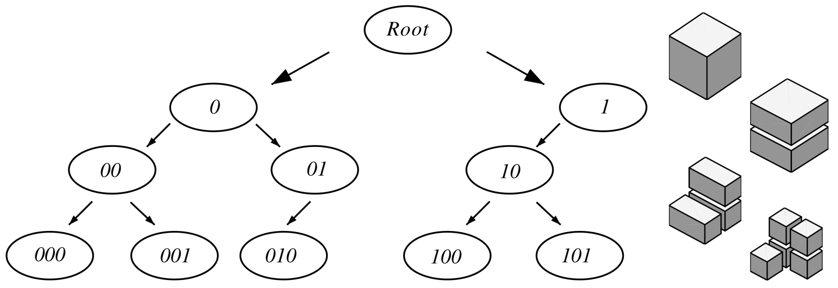

Figure 1: Hierarchical storage of the box collections in the software package GAIO. Illustration taken from [2].

1 Motivation

This software is provided to analyse the dynamics of maps of the form

$$x_{n+1} = f\left(x_n\right) \tag{1}$$

or flows given by an ODE

$$\dot{x}\left(t\right) = f\left(t, x\left(t\right)\right), \quad t \in \left[t_0, t_{end}\right] \tag{2}$$

using a set-oriented ansatz. It relies on the idea of the software package GAIO, see [1], but utilises massive parallelisation on both CPU and graphics cards (GPU).
Based on a surrounding hyperrectangle/box, one can gradually bisect boxes in arbitrary coordinate direction, discard them or insert new boxes. For a three-dimensional illustrative example, see figure 1. In addition, one can set flags in each box, see also section 3.1. While in GAIO the whole tree is stored, here only the leaves are explicitly stored. It means in effect that by setting a flag of a box in smaller depth, all its descendants inherit this flag automatically. Vice versa, a flag in such a box is seen as set if and only if all of its descendant leaves have the flag set. Apart from that, all "tree algorithms" are supposed to yield the same results in both software packages.
Here, so called point iterators are additionally provided. As example, see figure 2. There, four two-dimensional boxes are displayed. The point iterator traverses each box and therein an inner grid with 4^2 points. Each grid point can be mapped with an arbitrary map; in figure 2, the identity map is used. The numbers indicate the order of traversing. Yet, those points are not stored explicitly. That means, they are only computed when needed. This saves memory and increases memory bandwidth.

2 Installation

2.1 Version Management

Different versions of $\mathcal{B}_{12}$ are treated with lower case letters starting from `a`, i.e. $\mathcal{B}_{12a}$, $\mathcal{B}_{12b}$, and so on. In the following, the versions are denoted by $\mathcal{B}_{12x}$ or `B12x`.

2.2 Requirements

Using this software requires the following packages:

- CUDA, at least version 7.0, and its requirements.

If one additionally wants to use the Matlab interface, one needs

- Matlab, at least version R2013b (8.2), and
- Boost, at least version 1.31.

It is also recommended to install

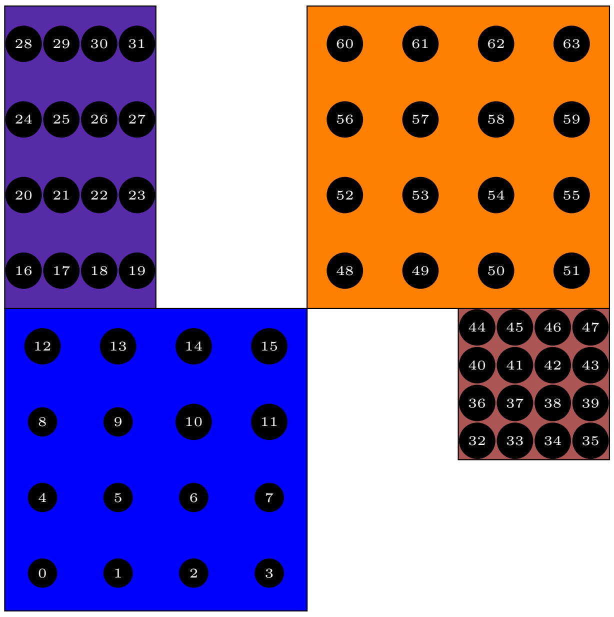

Figure 2: Illustration of point iterators. Each black circle indicates a grid point, which is mapped with the identity map. The numbers inside indicate the ordering, i.e. the "begin" iterator points to 0 and the "end" iterator points to (imaginary) 64.

- OpenMP,
- a CUDA-capable GPU with compute capability ≥ 2.0, and
- CMake, at least version 3.5.

2.3 Definition of Custom Maps

To use f of (1) or (2) a C++ functor needs to be implemented, i.e. a class/struct with member function `operator()`. In the following, we want to implement f of (1) or (2) calling the corresponding functor `MyMap`.
First, create the header file `B12x/src/include/Map_MyMap.h`. This header file must

- be guarded against redefinition via preprocessor directives,
- include `MapTools.h`,
- define the functor `MyMap`
  - that is included in the `b12` namespace,
  - that inherits from `public thrust::binary_function<REAL*,REAL*,bool>` where `REAL` stands for the floating point type,
  - whose member functions are all inlined and declared as both host and device function,
  - whose constant `operator()` member function additionally takes a constant real pointer as input and a non-constant real pointer as output and returns a boolean, and
  - that is (only) templated by Dimension and floating point type if you want to use CMake compilation or the Matlab interface.

That is, the minimal source code defining the functor `MyMap` might look like this:

```
#pragma once

#include "MapTools.h"
```

```
namespace b12 {

template<Dimension DIM, typename REAL>
struct MyMap : public thrust::binary_function<REAL*, REAL*, bool>
{
  __host__ __device__
  inline bool operator()(const REAL *, REAL *) const;
};

} // namespace b12
```

For a more general approach, consider implementing a help functor defining the right-hand side of (1) or (2). This functor can then be wrapped into the actual functor `MyMap` by using `IteratingMap` for (1) to allow the use of function composition, i.e. f^k for any $k = 0, 1, 2, \dots, 2^{32} - 1$, or `RK4` for (2) to solve the non-autonomous ODE (or `RK4Auto` when autonomous) by means of the classical Runge-Kutta method where the step size and step number can be set. For this, also see `Map_Henon.h` and `Map_Lorenz.h`.
To obtain maximal efficiency, the functor should

- use static, i.e. known at compile time, indices for the pointers given as arguments in `operator()`, which can also be realised by unrolled loops, see e.g. `Map_Identity.h`,
- not allocate dynamic memory,
- be implemented separately for single and double precision if the right-hand side of (1) or (2) contains functions for which the CUDA compiler provides different floating-point versions, e.g. for trigonometric, logarithmic, exponential or probabilistic functions, also see `Map_Standard.h`.

2.4 Compilation

Copy the $\mathcal{B}_{12x}$ folder into a location with enough available memory space, usually of the order of double-digit gigabytes depending on the amount of desired combinations of architectures, compute capabilities, phase space dimensions, floating-point types, and so on, see section 2.4.2 for more details.
Note that compilations may occupy extremely much time, especially for GPU code!

2.4.1 Plain Compilation

When neither CMake nor the Matlab interface is required, one may compile its `.cu` files with the nvcc compiler using the following flags:

`-arch=sm_**` to generate code for the correspondig GPU / compute capability,

`-std=c++11` to enable C++11 support,

`-O3` to let the compiler optimise the code, and

`-Xcompiler -fopenmp -lgomp` to allow OpenMP parallelization on CPU.

2.4.2 CMake Compilation

This is the recommended way for compiling. When a source code modification is implemented, e.g. the creation of a new map functor, only the dependent files are automatically (re)compiled. To use this method, one should modify `B12x/UserSettings.txt` containing the following variables:

`computeCapability` ... the compute capability for which the code is compiled for (when no GPU is available, use a value like 3.5), `installPath` . . . the installation directory, i.e. `installPath/B12/B12x` will contain the compiled library,

`ThrustArchitectures` . . . list of memory/parallelisation schemes (`CPP`: sequential on CPU, `OMP`: parallel on CPU using OpenMP, `CUDA`: parallel on GPU),

`Dimensions` . . . list of phase space dimensions,

`Reals` . . . list of C++ floating-point types (only `float` and `double` supported),

`MaxDepths` . . . list of maximal depths for the `ImplicitBoxTree`, also see section 3,

`Grids` . . . list of grid functors (currently only `InnerGrid` and `FullGrid` are tested),

`Maps` . . . list of map functors, which also implicitly contains the identity map for all dimensions defined in `Dimensions`.

To include `MyMap`, one should define a variable with this name as list of two strings consisting of integers, e.g. `set(MyMap "2 3" "3 5")`. In this example, `MyMap` would be compiled for the dimensions 2 and 3 where the constructors take 3 and 5 arguments, respectively. Subsequently, add `"MyMap"` to the list `Maps`.
Afterwards the libraries can be compiled and built with CMake. For Linux systems e.g., open a terminal and type in the following commands:

```
cd [path to the B12x folder]/build
cmake ..
make -j4
make -j4
make -j4
sudo make install
sudo make install
```

The number behind the `j` indicates the number of cores used to parallelise compilation.
Afterwards, make sure that the software package can be found by using the command:

```
echo 'export PATH=/opt/B12/B12x/:${PATH}' >> ~/.bashrc
```

2.5 Uninstallation

To uninstall $\mathcal{B}_{12x}$, only the folders `B12x` containing the software need to be deleted.

3 Usage

3.1 Integer Data Types

Even though in Matlab 64 bit integers are used to represent count numbers and indices, always pay attention to the underlying data types, especially to their ranges.

Dimension $(0, \ldots, 255)$

stands for possible phase space dimensions for (1) and (2).

Depth $(0, \ldots, 255)$

indicates the actual depth in ImplicitBoxTrees. In particular, 0 stands for the tree's root, `getLeafDepth()` = 255 = -1 for its leaves, and `getUnsubdividableDepth()` = 254 = -2 for nodes that are not leaves and whose only child is a leaf or both children are leaves. The maximal possible depth is determined by the template parameter, see section 3.2, but not greater than 253.

NrBoxes $(0, \ldots, 2^{32} - 1 = 4,294,967,295)$

is basically used indicating search indices or box counts. The largest value, also representable as `NrBoxes(-1)`, is reserved for the search index "not found".

NrPoints $(0, \ldots, 2^{64} - 1)$

is internally used to index points. Therefore, using at least 2^{64} points at once leads to undefined behaviour.

Flags

ranges actually form 0 to 255. But it should actually be seen as a type storing eight different on/off states, representable as powers of two. That is, 4 e.g. stands for the third state, while $73 = 64 + 8 + 1$ stands for the seventh, fourth, and first state. The meaning of each state is up to the user's decision. Besides, there is the enumeration declaration `Flag` containing `NONE` = 0, `HIT` = 1, `INS` = 2, `EXPD` = 4, `SD` = 8, `EQU` = 16, `ENTR` = 32, `EXTR` = 64, and `ALL` = 255. By design, checking whether the Flag `NONE` = 0 is set always yields true.

3.2 C++

When implementing plain CUDA C++ code, include `Grids.h`, the relevant `Map_MyMap.h`, and `ImplicitBoxTree.h`. Unless the corresponding libraries are already precompiled and installed, define the preprocessor variable `INCLUDE_IMPLICIT_BOX_TREE_MEMBER_DEFINITIONS` before including. The header files can be found in `B12x/src/include`.
The class `ImplicitBoxTree` is templated by the enumeration declaration `Architecture`, see section 2.4.2, `Dimension` for the desired phase space dimension of (1) and (2), a floating-point precision (default = double) for the coordinate computations, and an unsigned 8-bit integer (default = 64) representing the maximal possible depth, which can take on the values 32, 64, 72, 80, 96, and 128. For example, `ImplicitBoxTree<CUDA, 3, float, 96>` is a valid type. All the member methods are described in `ImplicitBoxTree.h`. The same applies for `Grids.h`. As simplification to distinguish between CPU and GPU, see the implementations in `ThrustSystem.h`. In particular, `ThrustSystem<A>::Vector<T>` can be used as vector storage type, where `A` stands for an `Architecture` and `T` for an arbitrary type, `typename ThrustSystem<A>::execution_policy()` as parallelisation scheme for Thrust algorithms, and `ThrustSystem<A>::Memory::isOnHost()` as boolean indicating which memory is used.

Example for 20 iterations of the subdivision algorithm with the Hénon map

```
#include "Grids.h"
#include "Map_Henon.h"
#define INCLUDE_IMPLICIT_BOX_TREE_MEMBER_DEFINITIONS
#include "ImplicitBoxTree.h"
#undef INCLUDE_IMPLICIT_BOX_TREE_MEMBER_DEFINITIONS

using namespace b12;

int main()
{
  const Architecture A = CUDA;
  const Dimension DIM = 2;

  ThrustSystem<A>::Vector<double> center(DIM, 0.0);
  ThrustSystem<A>::Vector<double> radius(DIM, 3.0);

  // create tree with surrounding box: [-3, 3]^2
  ImplicitBoxTree<A, DIM, double, 64> ibt(center, radius);
  // create an inner grid scheme with 10^DIM points for each box
  InnerGrid<DIM, double> grid(10);

  // create Henon map functor with parameters 1.4 and 0.3, once applied
  Henon<DIM, double> map(1.4, 0.3, 1);

  // set the Flag SD in the root
  // child nodes/boxes inherit the Flags while subdividing
  ibt.setFlags(Flag::SD, 0);

  for (int i = 1; i <= 20; ++i) {
    // subdivide all leaf boxes that have Flag SD set
    ibt.subdivide(Flag::SD);

    // define a iterator pair representing the range, i.e. [begin, end),
    // of all points created by grid for each leaf box and mapped by map;
    // NOTE: After the ImplicitBoxTree has changed, such an iterator has
    // undefined behaviour!!!
    auto itPair = ibt.makePointIteratorPairOverLeaves(grid, map);

    // set the Flag HIT in all leaf boxes that are 'hit' by the mapped points
    ibt.setFlagsByIterators(itPair, Flag::HIT, -1);

    // remove all leaf boxes (and corresponding parents) that have HIT not set
    ibt.remove(Flag::HIT);

    // unset HIT for next iteration
    ibt.unsetFlags(Flag::HIT, -1);

    // print iteration and number of leaves
    std::cout << i << ": " << ibt.count(-1) << std::endl;
  }

  return 0;
}
```

3.3 Matlab

To use $\mathcal{B}_{12x}$ in Matlab, one should add the appropriate folder to Matlab's search path by calling `addpath(genpath('/opt/B12/B12x/matlab'));`
After installation, the following classes are provided. Their instructions can be found in the corresponding m-files stored in `/opt/B12/B12x/matlab`.

ImplicitBoxTree, ompImplicitBoxTree, gpuImplicitBoxTree

All three classes represent the ImplicitBoxTree class, where ImplicitBoxTree works sequentially on CPU, ompImplicitBoxTree in parallel (OpenMP) on CPU, and gpuImplicitBoxTree on GPU. The "template arguments", see section 3.2, are passed via a cell array into the constructor. Note that the assignment operator only works as shallow copy. For a deep copy, use the (copy) constructor. It is also worth mentioning that many member functions have only one version, whose behaviour depends on the input, see e.g. `setFlags`, compared to their C++ code equivalents, e.g. `setFlags`, `setFlagsByIterators`, `setFlagsByPoints`, `setFlagsBySearch`, and `setFlagsByStencil`. That is why there are no methods like `makePointIteratorPair`.

GridFunctor, MapFunctor

Both classes have the same structure. The first component is the string indicating the C++ type, i.e. for GridFunctor "InnerGrid" or "FullGrid" and for MapFunctor "MyMap" are valid.

The second component stores the dimension while the third one stores the vector of constructor arguments.

Example programs

In B12x/matlab, you can find some example implementations to become more familiar with the usage in Matlab.

Equivalent example from above

```
addpath(genpath('/opt/B12/B12x/matlab'));

DIM = 2; HIT = 1; SD = 8;

% create tree on GPU with surrounding box: [-3, 3]^2
ibt = gpuImplicitBoxTree([0;0], [3;3], {DIM, 'double', 64});

% create an inner grid scheme with 10^DIM points for each box
G = GridFunctor('InnerGrid', DIM, 10);

% create Henon map functor with parameters 1.4 and 0.3, once applied
M = MapFunctor('Henon', DIM, [1.4, 0.3, 1]);

% set the Flag SD in the root
% child nodes/boxes inherit the Flags while subdividing
ibt.setFlags('all', SD);

for i = 1:20
    % subdivide all leaf boxes that have Flag SD set
    ibt.subdivide(SD);

    % a cell array containing a grid, a map, a depth, and a Flag;
    % in each leaf box (depth = -1) that have Flags 0 set, i.e. in effect
    % all boxes, 10^2 inner grid points are created and then mapped via
    % the Henon map
    scheme = {G, M, -1, 0};

    % set the Flag HIT in all leaf boxes that are 'hit' by the mapped points
    ibt.setFlags(scheme, HIT, -1);

    % remove all leaf boxes (and corresponding parents) that have HIT not set
    ibt.remove(HIT);

    % unset HIT for next iteration
    ibt.unsetFlags('all', HIT);

    % print iteration and number of leaves
    fprintf('%2d: %9d\n', ibt.depth(), ibt.count(-1));
end
```

References

[1] Michael Dellnitz, Gary Froyland, and Oliver Junge. The algorithms behind gaio - set oriented numerical methods for dynamical systems. In *Ergodic theory, analysis, and efficient simulation of dynamical systems*, pages 145–174. Springer, 2001.

[2] Michael Dellnitz and Oliver Junge. Set oriented numerical methods for dynamical systems. *Handbook of dynamical systems*, 2(1):900, 2002.