Solving Ordinary Differential Equations on GPUs

Karsten Ahnert, Denis Demidov and Mario Mulansky

Abstract Ordinary Differential Equations (ODEs) are a fundamental mathematical tool to model physical, biological or chemical systems, and they are widely used in engineering, economics and social sciences. Given their vast appearance, it is of crucial importance to develop efficient numerical routines for solving ODEs that employ the computational power of modern GPUs. Here, we present a high-level approach to compute numerical solutions of ODEs by developing a generic implementation of the most common algorithms and combining this with modern C++ libraries like VexCL and Thrust. Our approach is based on generic programming and results in highly scalable and easy-to-use source code.

1 Introduction

One of the most common problems encountered in Physics, Chemistry, Biology, but also Engineering or Social Sciences, is to find the solution of an initial value problem (IVP) of an ordinary differential equation (ODE). In fact, many physical laws are written in terms of ODEs, for example the whole classical mechanics, but ODEs also emerge from discretization of partial differential equations (PDEs) or in models

Karsten Ahnert
Ambrosys GmbH, Albert-Einstein-Str. 1-5, 14469 Potsdam, Germany,
e-mail: karsten.ahnert@gmx.de

Denis Demidov
Kazan Branch of Joint Supercomputer Center, Russian Academy of Sciences, Lobachevsky st. 2/31, 420011 Kazan, Russia,
e-mail: dennis.demidov@gmail.com

Mario Mulansky
Max-Planck Institute for the Physics of Complex Systems, Nöthnitzer Str. 38, 01187 Dresden,
TU-Dresden, Institute for Theoretical Physics, Zellescher Weg 17, 01069 Dresden
e-mail: mulansky@pks.mpg.de

of granular systems or when studying networks of interacting neurons. In the most cases one faces ODEs that are too complicated to be solved with analytic methods and one has to rely on numerical techniques to find at least an approximate solution. Of course, there exists a wide range of numerical algorithms to find such solutions of IVPs of ODEs. An introduction to both the mathematical background and the numerical implementation can be found in the textbooks from Hairer, Nørsett and Wanner [**?**, **?**]. The standard work for numerical programming, the "Numerical Recipes" [**?**] also contains detailed sections on solving ODEs. There are also several special classes of ODEs that require specific numerical methods, e.g. the Hamiltonian systems in physics which are typically solved using symplectic routines [**?**].

Obviously, there is a variety of numerical tools and libraries dedicated to solving ODEs. All mathematical software packages, like Matlab, Maple, Mathematica, or even R [**?**, **?**] contain routines for integrating ODEs. However, the focus here lies on the direct implementation of ODE simulations. For this task, one also finds a vast selection of numerical libraries, typically with Fortran or C/C++ bindings. Most prominent are probably the codes shipped with the "Numerical Recipes" book [**?**] containing several sophisticated explicit and implicit routines. The GNU scientific library (GSL) also provides ODE functionality [**?**], and finally the SUNDIALS suite [**?**] offers a modern implementation of all important algorithms. Unfortunately, none of those libraries supports GPU devices. However, there exists a highly flexible C++ library dedicated to ODEs: Boost.odeint[1], which is designed in such a generic way that the algorithms are implemented completely independent from the computational backend. Thus, by providing a computational backend that employs GPUs one immediately gets a GPU implementation of the ODE solver. Boost.odeint already includes several backends for GPU computations: for the NVIDIA CUDA-framework based on the Thrust[2] library or the CUDA MTL4 [3] [**?**] and for the OpenCL-framework based on VexCL[4], ViennaCL [5], or Boost.Compute[6]. In this text we will show how to implement ODE algorithms in such a generic way that separates the computational backend and thus greatly simplifies the portability to GPUs. Furthermore, we present two such backends, based on CUDA and OpenCL and develop several example simulations using these ODE codes. However, the most difficult part when writing an ODE simulation is the implementation of the right-hand-side (RHS) of the ODE, as it will be explained later. Hence, although Boost.odeint provides all the functionality to find a numerical solution of a given ODE, implementing the RHS of an ODE remains a non-trivial task.

The examples presented later will use modern C++ techniques and thus require the reader to be familiar with several advanced C++ concepts, e.g. we will make heavy use of templates to write generic code. Moreover, knowledge of the C++-

[1] http://www.odeint.com
[2] http://thrust.github.com
[3] http://www.simunova.com/gpu_mtl4
[4] https://github.com/ddemidov/vexcl
[5] http://viennacl.sourceforge.net/
[6] https://github.com/kylelutz/compute

Standard Library is also useful, specifically containers, iterators and algorithms. For the ODE algorithms implementation we make use of the C++03 standard only, but in some of the examples we employ the new C++11 and even C++14 abilities.

In the following sections we will give a short introduction to ODEs and the basic numerical schemes for finding approximate solutions (section 2), followed by a description of the generic implementation of those algorithms in section 3. Then in section 4 we will specifically describe how to use the various GPU backends and how they are implemented. The Boost.odeint library is introduced in section 5 and section 6 contains several examples on how to efficiently implement the RHS of different ODE problems together with discussion of performance implications of possible implementations. Finally, section 7 contains a short summary and conclusions.

2 Numerical Schemes

Before describing the generic implementation of ODE solvers and how to adapt them for GPU usage we will give a short introduction to ODEs and some mathematical background about the numerical schemes. This is mainly to familiarize the reader with our notation; for a more detailed description of the mathematics behind ODE integration we refer to standard textbooks, e.g. [**?**, **?**].

2.1 Ordinary Differential Equations

Generally, an ODE is an equation containing a function $x(t)$ of an independent variable t and its derivatives x', x'', ...:

$$F(x,x',x'',\ldots,x^{(n)},t) = 0. \tag{1}$$

This is the most general form, including implicit ODEs. However, we will here only consider *explicit* ODEs, which are of the form $x^{(n)} = f(x,x',x'',\ldots,x^{(n-1)})$ and are much simpler to be addressed numerically. The highest derivative n that appears in the ODE is called the *order* of the ODE. But any ODE of order n can be easily transformed into an n-dimensional ODE of first order. Therefore, it is sufficient to consider only first order differential equations where $n = 1$. The numerical routines presented later will all deal with initial value problems (IVP) where additionally to the ODE one has also given the value for x at a starting point $x(t = t_0) = x_0$. Thus, the mathematical formulation of the problem that will be numerically addressed throughout the following pages is:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t),t), \qquad \mathbf{x}(t=t_0) = \mathbf{x}_0. \tag{2}$$

Here, we use bold face $\mathbf{x}$ to indicate a possible vector character. Typically, the ODE is defined for real-valued variables, i.e. $\mathbf{x} \in \mathbb{R}^N$, but it is also possible to consider complex valued ODEs where $\mathbf{x} \in \mathbb{C}^N$. The function $\mathbf{f}(\mathbf{x},t)$ is called the right-hand-side (RHS) of the ODE. The most simple physical example for an ODE is probably the *harmonic oscillator*, e.g. a point mass connected to a spring. Newton's equation of motion for such a system is:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}q(t) = -\omega_0^2 q(t), \tag{3}$$

where $q(t)$ denotes the position of the mass and ω_0 is the oscillation frequency, a function of the mass m and the stiffness of the spring k: $\omega_0 = \sqrt{k/m}$. This can be brought into form (2) by introducing $p = \mathrm{d}q/\mathrm{d}t$, using $\mathbf{x} = (q,p)^T$ and defining some initial conditions, e.g. $q(0) = q_0$, $p(0) = 0$. Using the short-hand $\dot{\mathbf{x}} = \mathrm{d}\mathbf{x}/\mathrm{d}t$ and omitting explicit time dependencies we get:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \begin{pmatrix} p \\ -\omega_0^2 q \end{pmatrix}, \qquad \mathbf{x}(0) = \begin{pmatrix} q_0 \\ 0 \end{pmatrix}. \tag{4}$$

Note, that $\mathbf{f}$ in Eq. (4) does not depend on the variable t, which makes Eq. (4) an *autonomous* ODE. Also note that in this example the independent variable t denotes the time and $\mathbf{x}$ a point in phase spaces, hence the solution $\mathbf{x}(t)$ is the *trajectory* of the harmonic oscillator. This is a typical situation in physical ODEs and the reason behind our choice of variables t and $\mathbf{x}$.[7]

For the harmonic oscillator in Eq. (4), one can easily find an analytic solution of the IVP: $q(t) = q_0 \cos \omega_0 t$ and $p(t) = -q_0 \omega_0 \sin(\omega_0 t)$. For more complicated, non-linear ODEs it is often impossible to find an analytic solution and one has to employ numerical methods to at least find an approximate solution. One specific example are systems exhibiting *chaotic dynamics* [**?**], where the trajectories can not be described in terms of analytic functions. One of the first models where this has been explored is the so-called Lorenz-system [**?**], a three-dimensional ODE given by the following equations for $\mathbf{x} = (x_1, x_2, x_3)^T \in \mathbb{R}^3$:

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= R x_1 - x_2 - x_1 x_3 \\ \dot{x}_3 &= x_1 x_2 - b x_3, \end{aligned} \tag{5}$$

where $\sigma, R, b \in \mathbb{R}$ are parameters of the system. Although the solution might be impossible to find analytically, there are mathematical proofs about its *existence*

[7] In Mathematics, the independent variable is often called x and the function is $y(x)$.

and *uniqueness* under some conditions on the RHS $\mathbf{f}$, e.g. the Picard-Lindelöf theorem which requires $\mathbf{f}$ to be Lipschitz continuous [**?**]. Provided that this condition is fulfilled and a unique solution does exist, as it is the case for almost all practical problems, one can apply a numerical algorithm to find an approximate solution.

2.2 Runge-Kutta Schemes

The most common general-purpose schemes for solving initial value problems of ordinary differential equations are the so-called *Runge-Kutta* (RK) methods [**?**]. We will focus on the *explicit* RK-schemes as those are easier to implement and well-suited for GPUs. They are a family of iterative one-step methods that rely on a temporal discretization to compute an approximate solution of the IVP. Temporal discretization means that the approximate solution is computed at time points t_n. So we use $\mathbf{x}_n$ for the numerical approximation of the solution $x(t_n)$ at time t_n. In the simplest, but most frequently used case of an equidistant discretization with a constant step size Δt, one writes for the numerical solution:

$$\mathbf{x}_n \approx \mathbf{x}(t_n), \quad \text{with} \quad t_n = t_0 + n \cdot \Delta t. \tag{6}$$

The approximate points $\mathbf{x}_n$ are obtained sequentially using a numerical algorithm that can in the most general form be written as:

$$\mathbf{x}_{n+1} = \mathbf{F}_{\Delta t}(\mathbf{x}_n). \tag{7}$$

The mapping $\mathbf{F}_{\Delta t}$ here represents the numerical algorithm, i.e. the Runge-Kutta scheme, that performs one iteration from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$ with the time step Δt. The numerical scheme is said to have the order m if the solution it generates is exact up to some error of order $m+1$:

$$\mathbf{x}_1 = \mathbf{x}(t_1) + O(\Delta t^{m+1}), \tag{8}$$

where $\mathbf{x}(t_1)$ here is the exact solution of the ODE at t_1 starting from the initial condition $\mathbf{x}(t_0) = \mathbf{x}_0$. Hence, m denotes the order of accuracy of a *single step* of the scheme.

The most basic numerical algorithm to compute such a discrete trajectory $x_1, x_2, \ldots$ is the *Euler scheme*, where $F_{\Delta t}(\mathbf{x}_n) = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(\mathbf{x}_n, t_n)$, which means the next approximation is obtained from the current one by:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(\mathbf{x}_n, t_n). \tag{9}$$

This scheme has no practical relevance because it only offers accuracy of order $m = 1$. A higher order can be reached by introducing intermediate points and thus dividing one step into several stages. For example, the famous "RK4" scheme, sometimes also called *the* Runge-Kutta method, has $s = 4$ stages and also order $m = 4$. It

$$\begin{array}{c|cccc} c_1 & & & & \\ c_2 & a_{2,1} & & & \\ c_3 & a_{3,1} & a_{3,2} & & \\ \vdots & \vdots & & \ddots & \\ c_s & a_{s,1} & a_{s,2} & \dots & c_{s,s-1} \\ \hline & b_1 & b_2 & \dots & b_{s-1} \quad b_s \end{array}$$

(a) Generic Butcher Tableau with s stages.

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

(b) Coefficients for the RK4 method.

Table 1: Butcher Tableaus.

is defined as follows:

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{1}{6}\Delta t(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \quad \text{with} \\ \mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_n, t_n), \\ \mathbf{k}_2 &= \mathbf{f}\left(\mathbf{x}_n + \frac{\Delta t}{2}\mathbf{k}_1, t_n + \frac{\Delta t}{2}\right), \\ \mathbf{k}_3 &= \mathbf{f}\left(\mathbf{x}_n + \frac{\Delta t}{2}\mathbf{k}_2, t_n + \frac{\Delta t}{2}\right), \\ \mathbf{k}_4 &= \mathbf{f}\left(\mathbf{x}_n + \Delta t\,\mathbf{k}_3, t_n + \Delta t\right). \end{aligned} \tag{10}$$

Note, how the subsequent computations of the intermediate results $\mathbf{k}_i$ depend on the result of the previous stage $\mathbf{k}_{j<i}$.

More generally, a Runge-Kutta scheme is defined by its number of stages s and a set of parameters $c_1 \dots c_s$, $a_{21}, a_{31}, a_{32}, \dots, a_{ss-1}$ and $b_1 \dots b_s$. The algorithm to calculate the next approximation x_{n+1} is then given by:

$$x_{n+1} = x_n + \Delta t \sum_{i=1}^{s} b_i k_i, \qquad \text{where} \qquad k_i = f(x_n + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j, t_n + c_i \Delta t). \tag{11}$$

The parameter sets $a_{i,j}$, b_i and c_i define the so-called Butcher tableau (see Table 1) and fully describe the specific Runge-Kutta scheme. The Butcher tableau for the RK4 scheme above is given in Table 1b. Note, that the above schemes have a lower triangular structure. For tableaus with entries in the upper right region the method becomes an implicit RK-scheme and can not easily be solved.

3 Generic Runge-Kutta Implementation

In this section, we will develop an implementation of the Runge-Kutta schemes described above. The code will be designed in such a way that it separates the algorithm from the underlying computations and thus can be easily ported to GPUs. We will therefore analyze the computational requirements of the Runge-Kutta algo-

Requirement	Representation in C++	Example
Represent mathematical entities	Template parameter	`vector<double>, double`
Memory management	Function specialization	`resize<state_type>`
Vector iteration	Template parameter	`container_algebra`
Elementary operations	Template parameter	`default_operations`

Table 2: Computational requirements of the Runge-Kutta algorithms.

rithms and produce a modularized implementation. In this way, we will be able to replace, for example, the memory management and the computational backend with GPU variants and thus obtain a GPU implementation without re-implementing the algorithm itself. This will allow us to easily use the same code with different GPU technologies, i.e. CUDA and OpenCL.

3.1 Computational Requirements

To analyze the algorithmic parts involved in a Runge-Kutta scheme, we will start with a straight-forward implementation that does not yet provide any modularization. Listing 1 shows such an implementation for the RK4 algorithm as given by Eq. (10). It defines a class `runge_kutta4` that provides a member function `do_step` which performs a single RK4 step given a system function `system`, the current state `x`, the current time `t` and the time step `dt`. Note how we use a template parameter `System` to specify the system function. This gives us already some flexibility as `do_step` immediately works with function pointers and functor object, but also in more complicated cases like generalized functions objects from `std::function` or `boost::function` [**?**, **?**] or even C++11 lambdas. Basically anything that defines a function call operator with the signature `operator()(state_type &x, state_type &k, double t)` can be supplied as `system` in `do_step`.

In the following we will extract the computational requirements for the Runge-Kutta algorithms from the simple implementation in Listing 1. First, we need to define a representation of the dependent variable $\mathbf{x}$. In the `runge_kutta4` class a `vector<double>` from the Standard Template Library [**?**] is used for that purpose (Line 7). After that, we need to define the type of the independent variable t (called the `time_type` below). In Listing 1 (Line 13) we use `double` for this purpose. Then we need to introduce variables for temporary results (Line 38) and allocate enough memory for the temporaries, done in the constructor (Line 10). And finally we have to perform the summation and multiplication, in general operations of the form:

$$\mathbf{y} = a_1\mathbf{x}_1 + a_2\mathbf{x}_2 + \cdots + a_s\mathbf{x}_s, \tag{12}$$

where $\mathbf{y}$ and $\mathbf{x}_n$ are of `state_type` and a_s are of floating point type, typically `double`. Hence, from a mathematical view point, these operations are vector-vector

Listing 1: Simple Runge-Kutta4 implementation `simple_runge_kutta4.hpp`

```
class runge_kutta4 {
public:
  typedef std::vector<double> state_type;
  runge_kutta4(size_t N)
    : N(N), x_tmp(N), k1(N), k2(N), k3(N), k4(N) { }
  template<typename System>
  void do_step(System system, state_type &x, double t, double dt)
  {
    const double dt2 = dt / 2;
    const double dt3 = dt / 3;
    const double dt6 = dt / 6;
    system(x, k1, t);
    for(size_t i = 0; i < N; ++i)
      x_tmp[i] = x[i] + dt2 * k1[i];
    system(x_tmp, k2, t + dt2);
    for(size_t i = 0 ; i < N; ++i)
      x_tmp[i] = x[i] + dt2 * k2[i];
    system(x_tmp, k3, t + dt2);
    for(size_t i = 0; i < N; ++i)
      x_tmp[i] = x[i] + dt * k3[i];
    system(x_tmp, k4, t + dt);
    for(size_t i = 0; i < N; ++i)
      x[i] += dt6*k1[i] + dt3*k2[i] + dt3*k3[i] + dt6*k4[i];
  }
private:
  const size_t N;
  state_type x_tmp, k1, k2, k3, k4;
};
```

Listing 2: Runge-Kutta class with templated types

```
template<
   class state_type,
   class value_type = double,
   class time_type = value_type
   >
class runge_kutta4 {
   // ...
};
typedef runge_kutta4< std::vector<double> > rk_stepper;
```

addition and scalar-vector multiplication. In the `runge_kutta4` class above we specifically perform the iteration over the elements of the `state_type` and use the intrinsic operators `+` and `*` on those elements which are just **`double`** values here. All the requirements identified above are again listed in Table 2. Note how in the `runge_kutta4` class in Listing 1 the parts to satisfy these requirements are hard-

coded into the class. If we want to change, for example, the `state_type` to some construct that resides on the GPU, we have to completely rewrite the class for a new `state_type`, but also to change the memory allocation and the vector operations, thus rewriting the whole algorithm, e.g. in terms of a new class `runge_kutta4_gpu`. In the next section, however, we will present a modularized implementation based on the requirements identified here, which allows to exchange the fundamental types, memory allocation and vector computations so that the code can be ported to GPUs without changing the algorithm itself.

3.2 Modularized Design

In the following, we will generalize the basic implementation above by moving the parts addressing the several requirements out of the `runge_kutta4` class and keeping only the essential algorithm.

We start with the fundamental types used to represent the mathematical objects in the Runge-Kutta schemes Eq. (10). From a computational point of view we identify three different kinds of objects:

1. The state of the solution at some time $\mathbf{x}(t)$, typically more dimensional and represented by a `vector<double>`.
2. The independent variable t, typically the time and represented by a `double`.
3. Parameters of the Runge-Kutta scheme as given in the Butcher Tableau (Table 1), usually also represented by `double` values.

The standard way to generalize an algorithm for arbitrary types in C++ is to introduce template parameters. We will also follow this approach and define three class template parameters `state_type`, `value_type` and `time_type`. Listing 2 shows the skeleton of the new `runge_kutta4` class. Note how we use default template parameters to provide `value_type` and `time_type` as `double`, so for the most typical case the user only has to specify the `state_type`, as shown exemplarily in Line 9. It should be noted that the derivatives might require a representation different from the state, especially if arithmetic types with dimensions are used, for example the ones from Boost.Units [**?**].

Let us now consider the memory allocation. In the basic implementation in Listing 1 this is done in the constructor which therefore requires the system size. This implementation relies on the existence of a `resize` and `size` member functions of the `state_type`, which is not generic enough because the `state_type` does not need to be a vector anymore, or even a container at all. Therefore we will change the implementation and introduce a templated helper function `resize` that takes care of the resizing and can be specialized by the user for any given `state_type`. The result is outlined in Listing 3. The `resize` function here adjusts the allocated memory of some object `out` using the size of the given object `in`. This is the most flexible way. With this technique the `runge_kutta4` class takes care of the memory automatically, and it works out-of-the-box for all containers that provide a `size` and

Listing 3: Memory allocation

```
1  template<class state_type>
2  void resize(const state_type &in, state_type &out) {
3    // standard implementation works for containers
4    out.resize(in.size());
5  }
7  // specialization for boost::array
8  template<class T, size_t N>
9  void resize(const boost::array<T, N> &, boost::array<T,N>& ) {
10   /* arrays don't need resizing */
11 }
13 template< ... >
14 class runge_kutta4 {
15   // ...
16   template<class Sys>
17   void do_step(Sys sys, state_type &x, time_type t, time_type dt)
18   {
19     adjust_size(x);
20     // ...
21   }
23   void adjust_size(const state_type &x) {
24     resize(x, x_tmp);
25     resize(x, k1);
26     resize(x, k2);
27     resize(x, k3);
28     resize(x, k4);
29   }
30 }
```

`resize` member functions. If some other `state_type` is employed, the user can implement an overload of the `resize` function to tell the `runge_kutta4` how to allocate memory. One example could be fixed-size arrays `boost::array<double,N>`, which live on the stack and do not require manual memory allocation. Hence, the `resize` function would just be empty (and disappear during the optimization step of the compilation), shown in Lines 7–11 in Listing 3. Note that this implementation supports out of the box the case when the system size changes during the integration, i.e. if the size of `x` changes between `do_step` calls. However, checking the system size at each step of the algorithm is not necessary for almost all situations and thus it is a waste of performance. This can be solved by adding a trivial logic that only calls `resize` during the first call of `do_step` (not shown here for clarity).

Now we arrive at the final and most difficult point: the abstraction of the numerical computation. As seen from the mathematical definition of the Runge-Kutta scheme in Eq. (11), we need to calculate vector-vector sums and scalar-vector products to perform a Runge-Kutta step. In the simplistic implementation above (Listing 1), this is done by explicit **`for`** loops and arithmetic operators `+` and `*`. In our abstraction of this computation, we divide these computations into two distinct parts: *iteration* and *operation*. The first one will be responsible for iterating over the ele-

Listing 4: Example algebra for the RK4 container_algebra.hpp

```
6  struct container_algebra {
7     template<class S1, class S2, class S3, class Op>
8     static void for_each3(S1 &s1, S2 &s2, S3 &s3, Op op) {
9        const size_t dim = s1.size();
10       for(size_t n = 0; n < dim; ++n)
11          op(s1[n], s2[n], s3[n]);
12    }
20 };
```

Listing 5: Example operations for the RK4 default_operations.hpp

```
6  struct default_operations {
7     template<class Fac1 = double, class Fac2 = Fac1>
8     struct scale_sum2 {
9        typedef void result_type;
11       const Fac1 alpha1;
12       const Fac2 alpha2;
14       scale_sum2(Fac1 alpha1, Fac2 alpha2)
15          : alpha1(alpha1), alpha2(alpha2) { }
17       template<class T0, class T1, class T2>
18       void operator()(T0 &t0, const T1 &t1, const T2 &t2) const {
19          t0 = alpha1 * t1 + alpha2 * t2;
20       }
21    };
48 };
```

ments of the involved state types, i.e. it addresses the vector character of the computation. The code structure that performs these iterations will be called `Algebra`. The *operation* on the other hand represents the computation that is performed for each element, i.e. within the iteration. The respective code structure will be called `Operation`.

We start with the `Algebra`. For the RK4 algorithm we need to provide two functions that do iteration over three and six container instances. A possible `Algebra` is presented in Listing 4.

The iteration is performed in terms of `for_each` functions that are gathered in a **`struct`** called `container_algebra`. The `for_each` functions expect a number of containers and an operation object as parameters. They simply perform the iteration over the elements of the containers and execute the given operation on each of the container's elements. Here we use a raw hand written for-loop which requires a `size()` member function and the `[]`-operator for the given container types `S1,S2...`. This loop could easily be generalized to use iterators which is the preferred and recommended way in C++ to iterate over containers. Inside the loop the functors `op` are applied to the elements of the containers. Listing 5 shows an exemplary implementation of such operations designed to be used within the

Listing 6: Generic RK4 implementation runge_kutta4.hpp

```
template<class state_type, class value_type = double,
       class time_type = value_type,
       class algebra = container_algebra,
       class operations = default_operations>
class runge_kutta4 {
public:
   template<typename System>
   void do_step(System &system, state_type &x,
                time_type t, time_type dt)
   {
      adjust_size( x );
      const value_type one = 1;
      const time_type dt2 = dt/2, dt3 = dt/3, dt6 = dt/6;
      typedef typename operations::template scale_sum2<
               value_type, time_type> scale_sum2;
      typedef typename operations::template scale_sum5<
               value_type, time_type, time_type,
               time_type, time_type> scale_sum5;
      system(x, k1, t);
      algebra::for_each3(x_tmp, x, k1, scale_sum2(one, dt2));
      system(x_tmp, k2, t + dt2);
      algebra::for_each3(x_tmp, x, k2, scale_sum2(one, dt2));
      system(x_tmp, k3, t + dt2);
      algebra::for_each3(x_tmp, x, k3, scale_sum2(one, dt));
      system(x_tmp, k4, t + dt);
      algebra::for_each6(x, x, k1, k2, k3, k4,
                     scale_sum5(one, dt6, dt3, dt3, dt6));
   }
private:
   state_type x_tmp, k1, k2, k3, k4;
   void adjust_size(const state_type &x) {
      resize(x, x_tmp);
      resize(x, k1); resize(x, k2);
      resize(x, k3); resize(x, k4);
   }
};
```

`container_algebra` above. It consists of two functor types organized in a **`struct`** called `default_operations`. The `scale_sum2` works with the `for_each3` above, and `scale_sum5` interacts with `for_each6`. Those functors consist of a number of parameters `alpha1`,`alpha2`... and a function call operator that calculates a simple product-sum (Listing 5).

With these abstractions we have moved the computational details away from the algorithm into separate code structures and thus reached a generic implementation of the RK4 algorithm (shown in Listing 6). The `runge_kutta4` class got two more template parameters specifying the algebra and operations, i.e. the computational backend used for the calculation. We use the `container_algebra` and

`default_operations` from Listings 4 and 5 as the default values that will work for almost all cases. In the `do_step` method we now use the `for_each` functions from the given `Algebra` in combination with the `scale_sum` functors from the given `Operations` to perform the required computations. So the explicit **`for`**-loops, that were hard-coded into the algorithm in the first implementation (Listing 1), have been separated into two parts, an `algebra` and `operations`. Those parts are supplied to the algorithm in terms of template parameters and can thus be easily replaced without changing the algorithm itself. This flexibility now allows us to port the RK4 implementation to GPUs. The idea is to first provide a GPU data structure, e.g. a `gpu_vector` with the respective `resize` functions as required by the algorithm (Listing 3). Then we only need a `gpu_algebra` and `gpu_operations` to do the vector computations on the GPU in a parallelized way. Assuming we have implemented those pieces, the following code would give us a RK4 algorithm running on the GPU:

```
typedef runge_kutta4< gpu_vector<double>, double, double,
   gpu_algebra, gpu_operations > gpu_stepper;
```

So with the generalized implementation we have greatly simplified the problem of implementing a Runge-Kutta scheme on the GPU. Instead of having to start from scratch, we now only have to implement a basic data structure for the GPU (`gpu_vector`), provide low-level functions for memory allocation (`resize`), iteration (`algebra`) and fundamental calculations (`operations`). But the real strength of this approach is that these remaining problems are so fundamental that they are already solved for GPUs. Of course, there are libraries that provide data structures and memory management for the GPU, as well as parallelized iteration and element-wise computations. In the following sections we will introduce two such libraries and show how they are combined with the RK4 implementation from Listing 6 to produce a GPU-version.

It should be noted that this approach of separating the algorithm from the computations is not only valuable when aiming at GPU computations. With the implementation above we can, for example, also easily create a RK4 algorithm that work with arbitrary precision types instead of the usual **`double`**. Another example would be an ODE solver based on interval arithmetic [**?**], also easily implementable by providing some `interval_operations`.

3.3 Lorenz Attractor Example

Before considering the GPU backends we want to show how to use the codes above to compute a trajectory of the famous Lorenz system (5) introduced earlier in section 2.1. Listing 7 shows the implementation of a simulation of a trajectory for this system based on the `runge_kutta4` class developed above. As seen there, all that is left to do is to define the `state_type`, implement the RHS of the ODE, here done in terms of a functor `lorenz`, and define the initial conditions (Line 30). Now we

Listing 7: Computing a trajectory of the Lorenz system `lorenz_single.cpp`

```
#include <iostream>
#include <vector>
#include "runge_kutta4.hpp"
using namespace std;
typedef std::vector<double> state_type;
typedef ncwg::runge_kutta4< state_type > rk4_type;
struct lorenz {
   const double sigma, R, b;
   lorenz(const double sigma, const double R, const double b)
      : sigma(sigma), R(R), b(b) { }
   void operator()(const state_type &x,state_type &dxdt,double t)
   {
      dxdt[0] = sigma * ( x[1] - x[0] );
      dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
      dxdt[2] = -b * x[2] + x[0] * x[1];
   }
};
int main() {
   const int steps = 5000;
   const double dt = 0.01;
   rk4_type stepper;
   lorenz system(10.0, 28.0, 8.0/3.0);
   state_type x(3, 1.0);
   x[0] = 10.0;
   for( size_t n=0 ; n<steps ; ++n ) {
      stepper.do_step(system, x, n*dt, dt);
      cout << n*dt << ' ';
      cout << x[0] << ' ' << x[1] << ' ' << x[2] << endl;
   }
}
```

can use the Runge-Kutta algorithm implemented above (Listing 6) to iterate along the trajectory using a step size of $\Delta t = 0.1$.

4 GPU Backends

Having reduced the problem of running the ODE solver on GPUs to memory management and some basic algebra operations, we finally come to the point of implementing those necessities. Instead of relying on low-level GPU programming and thus essentially reinventing the wheel, we will use existing high-level libraries that offer GPU data structures as well as routines for algebraic operations. To cover all available GPU technologies we will develop two GPU backends, the first one based on the NVIDIA CUDA technology, the second one for the OpenCL framework. For the CUDA environments, we will employ the Thrust library [**?**], which is part of the

NVIDIA CUDA SDK [**?**]. In the case of OpenCL, we will rely on the VexCL library [**?**], an open source library developed at the Supercomputer Center of Russian Academy of Sciences.

4.1 Thrust Backend

The Thrust library is a C++ template library that provides containers and algorithms similar to the Standard Template Library (STL) [**?**], but capable of running parallel on a CUDA GPU. Besides the CUDA backend, Thrust also supports CPU parallelization via OpenMP [**?**] and Intel's Thread Building Block (TBB) [**?**], configurable at compile time by preprocessor variables. As said above, Thrust is part of the NVIDIA CUDA framework and thus requires the use of the `nvcc` compiler to generate code that can be executed on GPUs. For a thorough introduction into CUDA programming and Thrust in particular, we refer to the respective documents [**?**, **?**].

To handle the memory on the GPU, Thrust provides a `thrust::device_vector` template class similar to `std::vector` from the STL. This will be our basic `state_type` representing the state $\mathbf{x}$ of the dynamical system. As Thrust mimics the STL, the `thrust::device_vector` also has `size` and `resize` member functions, which means that the memory management for `std::vectors` given in Listing 3 also works nicely with `thrust::device_vectors` — no specialization is required. This is a nice example of how well-designed libraries, such as Thrust, decrease the required programming effort by increasing the re-usability of your code.

To ensure that the vector computations are executed in parallel on the GPU, we introduce a `thrust_algebra` as a replacement of the `container_algebra` (see Listing 4) above. To implement the `for_each3` and `for_each6` functions required in the algebra, we will employ Thrust's `thrust::for_each` routine. This routine has the following signature:

```
thust::for_each(Iterator begin, Iterator end, UnaryOperator op)
```

where the iterators `begin` and `end` define a range of data in a `device_vector` and `op` defines the operation performed for each element of the sequence. As seen from the signature above, `thrust::for_each` iterates only over a single range from `begin` to `end`, but for our `for_each3` and `for_each6` we need to iterate over several device vectors at once. Fortunately, this can be easily achieved by using `zip_iterators` that combine an arbitrary number of iterators into a single iterator and thus allows us to use `thrust::for_each` for iterating over several ranges at once. The implementation of the `thrust_algebra` based on `thrust::for_each` and `make_zip_iterator` in combination with `make_tuple` is shown in Listing 8. The usage of `make_zip_iterator` and `make_tuple` is almost self-explanatory: `make_tuple` combines the given parameters (iterators in this case) into a single tuple, and `make_zip_iterator` then converts this tuple of iterators into a single `zip_iterator` that can then be passed to the `for_each` algorithm. Note that the implementation of the `for_each6` algorithm is omitted here for clarity.

Listing 8: The Thrust Algebra `thrust_algebra.hpp`

```
6  struct thrust_algebra {
7     template<class S1, class S2, class S3, class Op>
8     static void for_each3(S1 &s1, S2 &s2, S3 &s3, Op op) {
9        thrust::for_each(
10             thrust::make_zip_iterator( thrust::make_tuple(
11                   s1.begin(), s2.begin(), s3.begin() ) ),
12             thrust::make_zip_iterator( thrust::make_tuple(
13                   s1.end(), s2.end(), s3.end() ) ),
14             op);
15    }
39 };
```

Listing 9: The Thrust Operations `thrust_operations.hpp`

```
9  struct thrust_operations {
10    template<class Fac1 = double, class Fac2 = Fac1>
11    struct scale_sum2 {
12       const Fac1 m_alpha1;
13       const Fac2 m_alpha2;
15       scale_sum2(const Fac1 alpha1, const Fac2 alpha2)
16          : m_alpha1(alpha1), m_alpha2(alpha2) { }
18       template< class Tuple >
19       __host__ __device__ void operator()(Tuple t) const {
20          thrust::get<0>(t) = m_alpha1 * thrust::get<1>(t) +
21                              m_alpha2 * thrust::get<2>(t);
22       }
23    };
48 };
```

Of course, we also need to replace the `default_operations`, containing the `scale_sum` functors (see Listing 5), by a CUDA-compatible implementation. These functions contain the code that in the end will run in parallel on the GPU, which means that they will be compiled into so-called *kernels*. Therefore, they need to be decorated by specific compiler instruction to make the `nvcc` compiler generate specific GPU code for those functions. For this purpose, CUDA provides the keywords `__device__` and `__host__`. The former indicates that a function will run on a GPU, and the latter assures that the compiler will also generate a CPU version. Listing 9 shows the implementation of the `thrust_operations`. The keywords are used before the function definition in Line 19.

Furthermore, we have to bear in mind that since we used `zip_iterators` in the `for_each`, the `scale_sum` functors also get the elements from several ranges packed in a single tuple. To access the individual elements, we have to unpack the tuple, which can be done by the Thrust's `get<N>(tuple)` function that simply returns the N-th entry of the given `tuple`. Together with the `thrust_algebra` (see Listing 8)

this completes the CUDA backend for the RK4 scheme. The following code defines a `gpu_stepper` class that computes an approximate trajectory using the GPU:

```
typedef thrust::device_vector<double> state_type;
typedef runge_kutta4< state_type, double, double,
   thrust_algebra, thrust_operations > gpu_stepper_type;
```

With this, we have successfully ported the RK4 scheme to GPUs using functionality from the Thrust library. However, for a complete simulation we also have to implement the RHS function such that it is also computed on the GPU. This is highly non-trivial and will be discussed in detail for several examples in Section 6.

4.2 VexCL Backend

The Thrust backend above allows to run ODE integration on NVIDIA GPUs only as it is based on the CUDA technology. To address a wider range of hardware, we will now present a computational backend based on OpenCL (Open Computing Language) [**?**]. OpenCL supports NVIDIA as well as AMD/ATI GPUs, but can also be used for parallel runs on multi-core CPUs.

As above, we will not start from scratch but rather employ the modern, well-designed GPGPU library VexCL [**?**]. The library does not only provide the required data structures, but also covers the vector operations which makes our work even simpler than with Thrust. As the data structure for representing a `state_type` we will use a `vex::vector`, which is again similar to a `std::vector`. Listing 10 shows the `resize` function specialized for the `vex::vector<T>`. Note how we have to pass on the list of OpenCL command queues that contains crucial information about where the data will reside (i.e. which compute device) to the vector's `resize` function. Just like the required size, we extract this information from the given `vex::vector` instance `in`. Additionally to the usual vectors, VexCL also provides a `vex::multivector<T,N>`, which is basically a group of `N` instances of `vex::vector<T>` and can be quite handy for some problems. Hence, we also provide the resize functionality for `vex::multivector<T,N>` in Listing 10.

We are left with the vector operations, but as mentioned above this is very simple with VexCL. Being a library designed specifically for linear algebra, VexCL natively supports vector-vector addition and scalar-vector multiplication. Assuming `x`, `y` and `z` are of type `vex::vector<double>` and `a` and `b` are **`double`** values, the following code performs the element-wise summation and scalar multiplication of the vectors:

```
z = a * x + b * y;
```

That means that the VexCL library intrinsically performs the iteration over the elements of the vector in parallel on an OpenCL compute device (i.e. a GPU). Mathematically, one can say that the `vex::vector` together with the standard `+` and `*` operators form a *vector space*. Hence, it is not required for us to implement a parallelized iteration ourselves and the existence of an `algebra` is not necessarily required, in contrast to the Thrust backend above (c.f. Listing 8). But as the `algebra`

Listing 10: Memory allocation for VexCL. vexcl_resize.hpp

```
11 template<class T>
12 void resize(const vex::vector<T> &in, vex::vector<T> &out) {
13    out.resize(in.queue_list(), in.size());
14 }
16 template<class T, size_t N>
17 void resize(const vex::multivector<T,N> &in,
18       vex::multivector<T,N> &out)
19 {
20    out.resize(in.queue_list(), in.size());
21 }
```

Listing 11: Vector space algebra. vector_space_algebra.hpp

```
6  struct vector_space_algebra {
7     template<class S1, class S2, class S3, class Op>
8     static void for_each3(S1 &s1, S2 &s2, S3 &s3, Op op) {
9        op(s1, s2, s3);
10    }
15 };
```

is part of the structure of our ODE solver and can not be neglected, we provide a trivial `vector_space_algebra` that simply forwards the operation directly to the vectors without performing an iteration. This is shown in Listing 11.

This implementation is not only useful for VexCL and its `vex::vector`, but also for any other vector library that provides vector operations in terms of `+` and `*` operators, e.g. MTL4 [**?**] or Boost.uBLAS [**?**]. To account for this generality we call this trivial algebra a `vector_space_algebra`, as it works with any type that forms a vector space. From the above it is also clear that for VexCL we do not need to take special care of the operations. As VexCL redefines the operators `+` and `*` itself, we can simply plug in the `default_operations` from the beginning (Listing 5). Therefore, the computational backend for OpenCL based on VexCL is finished and we can construct an algorithm that is capable of running on a GPU device with the following code:

```
typedef vex::vector<double> state_type;
typedef runge_kutta4< state_type, double, double,
   vector_space_algebra, default_operations > gpu_stepper_type;
```

5 The Boost.odeint Library

Above, we have shown how to implement the RK4 scheme in a generic way such that it can be easily ported to GPUs. We have demonstrated the strengths of this approach by providing two backends that address CUDA and OpenCL devices respectively. However, there is a vast potential for improvement and extension of this code. Although this goes well beyond the scope of the present text, we want to mention that a highly sophisticated implementation of the ideas and techniques above exists in the Boost.odeint library [**?**, **?**]. Boost.odeint also separates memory allocation, iteration and fundamental operations from the actual algorithm in the same way as described above in Section 3.2. But in contrast to the ad hoc implementation presented here, Boost.odeint is a fully grown library consisting of about 25,000 lines of C++ code. It includes a vastly larger functionality and we shortly list the most important points below:

- Arbitrary explicit Runge-Kutta schemes, predefined schemes: Dormand-Prince 5, Cash-Karp, Runge-Kutta78.
- Symplectic Runge-Kutta-Nyström schemes.
- Variable order method: Bulirsch-Stoer.
- Multistep methods: Adams-Bashforth, Adams-Bashforth-Moulton.
- Implicit routines: Rosenbrock method, implicit Euler.
- Step-size control and dense output.
- Integrate routines with observer support.
- Iterator and range interfaces.
- Support of arbitrary precision arithmetic with Boost.Multiprecision.
- Support of additional backends: eigen [**?**], GSL vectors [**?**], Math Kernel Library [**?**], Matrix Template Library [**?**], ViennaCL [**?**].

If Boost.odeint provides the necessary algorithms and functionality to solve a problem, we strongly advise to use this library. However, some problems require specialized schemes or additional computations. In this case the code developed in the previous pages should represent a good starting point to develop a specific algorithm in a generalized way that is easily portable to GPUs.

6 Example Problems

6.1 Lorenz Attractor Ensemble

In the first example we consider the Lorenz system (5). Solutions of the Lorenz system usually furnish very interesting behavior in dependence on one of its parameters. For example, one might want to study the chaoticity in dependence on the parameter R. Therefore, one would create a large set of Lorenz systems (each

Listing 12: lorenz_thrust_v1.hpp

```
30 typedef thrust::device_vector<double> state_type;
31 struct lorenz_system {
40    struct lorenz_functor {
41       double sigma, b;
42       lorenz_functor(double sigma, double b)
43          : sigma(sigma), b(b) {}
45       template<class T>
46       __host__ __device__ void operator()(T t) const {
47          double x = thrust::get<0>( t );
48          double y = thrust::get<1>( t );
49          double z = thrust::get<2>( t );
50          double R = thrust::get<3>( t );
52          thrust::get<4>( t ) = sigma * ( y - x );
53          thrust::get<5>( t ) = R * x - y - x * z;
54          thrust::get<6>( t ) = -b * z + x * y;
55       }
56    };
58    template<class State, class Deriv>
59    void operator()(const State &x, Deriv &dxdt, double t) const {
60       BOOST_AUTO(start,
61             thrust::make_zip_iterator( thrust::make_tuple(
62                   x.begin(),
63                   x.begin() + n,
64                   x.begin() + 2 * n,
65                   R.begin(),
66                   dxdt.begin(),
67                   dxdt.begin() + n,
68                   dxdt.begin() + 2 * n
69                   ) )
70             );
72       thrust::for_each(start, start+n, lorenz_functor(sigma, b));
73    }
74 };
```

with a different parameter R), pack them all into one system and solve them simultaneously. In a real study of chaoticity one may also calculate the Lyapunov exponents [**?**], which requires to solve the Lorenz system and their linear perturbations.

In the Thrust version of the example we define the state type as `device_vector` of size $3n$, where n is the system size. The X, Y, and Z components of the state are held in the continuous partitions of the vector. The system functor holds the model parameters and provides a function call operator with the necessary signature. Here we use the standard Thrust technique of packing the state components into a zip iterator which is then passed to a `thrust::for_each` algorithm (Listing 12).

The system function object for the VexCL version of the Lorenz attractor example is more compact than the Thrust variant because VexCL supports a rich set of vector expressions. We represent the three components of attractor trajectory as a

Listing 13: lorenz_vexcl_v1.cpp

```
28 typedef vex::multivector<double, 3> state_type;
29 struct lorenz_system {
36    void operator()(const state_type &x, state_type &dxdt,
37                    double t) const
38    {
39       dxdt = std::tie(
40             sigma * (x(1) - x(0)),
41             R * x(0) - x(1) - x(0) * x(2),
42             x(0) * x(1) - b * x(2) );
43    }
44 };
```

`multivector<double,3>`. Since VexCL provides all necessary overloads for the `multivector` type, we are able to use the `vector_space_algebra` in this case (Listing 13).

Figure 1 shows performance results for the Thrust, VexCL, and CPU versions of the Lorenz attractor example. Time in seconds required to make a 1000 of RK4 iterations is plotted against the ensemble size N. Lines denoted "Thrust v1" and "VexCL v1" correspond to the versions presented above. "CPU v1" is the Thrust version compiled for the OpenMP backend. Times for the Thrust and the VexCL versions of the code are given for the NVIDIA Tesla K20c GPU. Times for the CPU runs are given for the Intel Core i7 920 CPU (all four cores of which were used through OpenMP technology). It is clear from the figure that the initial implementations for the Thrust and the VexCL libraries perform equally well for large problem sizes and are about 14 times faster than the CPU version. VexCL has higher initialization costs and hence is a bit slower than Thrust for smaller problems. However, the distinction seems not as important once we note that both the Thrust and the VexCL versions loose to the CPU version for $N \lesssim 10^4$.

Note that both the Thrust and the VexCL versions above have the same drawback. Namely, both of them use device vectors as state type. Hence, intermediate state variables used in the steppers are stored in the global GPU memory. Moreover, each operation results in a launch of a separate compute kernel. A kernel launch has nonzero overhead both in CUDA and in OpenCL, but more importantly, each kernel needs to both read and write intermediate states from/to the global GPU memory. Since the problem is memory bound, this leads to a severe drop in performance.

We could overcome the above problem by providing a monolithic kernel which would encode the stepper logic and provide the complete solution in a single launch. However, the use of such kernel would also mean the loss of the flexibility we achieved so far by separation of algorithm and the underlying computations: one would have to completely re-implement the kernel for each new stepper. Luckily, VexCL library allows us to generate such a fused kernel automatically by providing the `vex::symbolic<T>` class template. Instances of the type dump to the specified output stream any arithmetic operations they are being subjected to. For example,

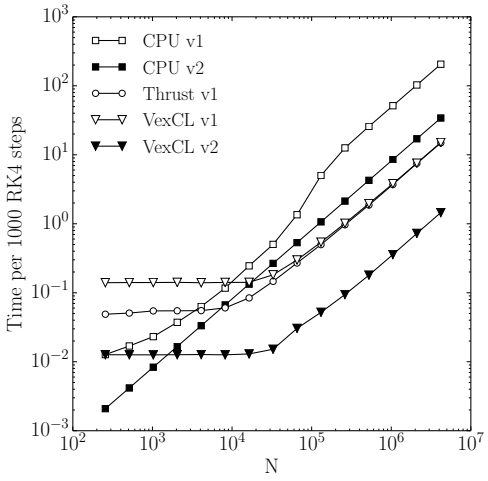

Fig. 1: Performance results for the Lorenz attractor example.

in the following code snippet two symbolic variables are declared and participate in an arithmetic expression:

```
vex::generator::set_recorder(std::cout);
vex::symbolic<double> x = 6, y = 7;
x = sin( x * y );
```

This generates the following output:

```
double var1 = 6;
double var2 = 7;
var1 = sin( (var1 * var2) );
```

This is implemented by overloading arithmetic operators and mathematical functions for the symbolic classes. So when two symbolic variables are being added, the overloaded addition operator just outputs names of the variables divided by symbol "+" to the specified output stream. By defining the state type to be `boost::array< vex::symbolic<double>, 3>`, and using the same algebra and the system function as in Listing 7, we are able to record the sequence of arithmetic operations made by a Runge-Kutta stepper. This gives us a fused kernel which is as effective as a manually written one (Listing 14).

This approach has some obvious restrictions: namely, it only supports embarrassingly parallel problems (no data dependencies between threads of execution), and it does not allow conditional statements or loops with non-constant number of iterations. But when the method works, it works very well. This version of the code is denoted "VexCL v2" on Figure 1 and is about 10 times faster than the initial VexCL implementation.

Listing 14: lorenz_vexcl_v2.cpp

```
34 typedef vex::symbolic<double> sym_vector;
35 typedef boost::array<sym_vector, 3> sym_state;
64 // Custom kernel body will be recorded here
65 std::ostringstream body;
66 vex::generator::set_recorder(body);
68 // State types that would become kernel parameters
69 sym_state sym_S = {{
70    sym_vector(sym_vector::VectorParameter),
71    sym_vector(sym_vector::VectorParameter),
72    sym_vector(sym_vector::VectorParameter)
73 }};
75 sym_vector sym_R(sym_vector::VectorParameter, sym_vector::Const);
77 // Stepper type
78 odeint::runge_kutta4_classic<
79    sym_state, double, sym_state, double,
80    odeint::container_algebra, odeint::default_operations
81    > stepper;
83 // Record single RK4 step
84 lorenz_system sys(sym_R);
85 stepper.do_step(sys, sym_S, 0, dt);
87 // Generate the kernel from the recorded sequence
88 auto kernel = vex::generator::build_kernel(ctx, "lorenz",
89       body.str(), sym_S[0], sym_S[1], sym_S[2], sym_R);
91 // Real state initialization
92 vex::vector<double> X(ctx, n), Y(ctx, n), Z(ctx, n), R(ctx, n);
93 X = Y = Z = 10.0;
94 R = Rmin + dR * vex::element_index();
99 // Integration loop
100 for(double t = 0; t < t_max; t += dt)
101    kernel(X, Y, Z, R);
```

We use a similar approach in order to accelerate the CPU version of the example. Namely, we create a Boost.odeint stepper for a single Lorenz attractor (state type is `boost::array<double,3>`), and then we use an outer loop which iterates over the complete ensemble (Listing 15). This version of the code ("CPU v2") uses less memory and is more cache-friendly. As a result, it is about 6 times faster than the Thrust example with the OpenMP backend. Unfortunately, the Thrust library does not allow the same type of optimization. We could in principle create a device function that would operate on a single attractor (by calling `runge_kutta4<...>::do_step` from inside the function), and apply the function to the complete ensemble with the help of the `thrust::for_each` algorithm. But CUDA requires all device functions to be decorated with `__device__` keyword, and the Boost.odeint functions are not marked as such.

Listing 15: lorenz_cpu_v2.cpp

```
67 #pragma omp parallel for
68 for(size_t i = 0; i < n; ++i) {
69    odeint::runge_kutta4_classic<
70       state_type, double, state_type, double,
71       odeint::container_algebra, odeint::default_operations
72       > stepper;
74    lorenz_system sys(R[i]);
75    for(double t = 0; t < t_max; t += dt)
76       stepper.do_step(sys, x[i], t, dt);
77 }
```

6.2 Chain of Coupled Phase Oscillators

As a second example we consider a chain of coupled phase oscillators. A phase oscillator describes the dynamics of an autonomous oscillator [**?**]. Its evolution is governed by the phase φ, which is a 2π-periodic variable growing linearly in time, i.e. $\dot{\varphi} = \omega$, where ω is the phase velocity. The amplitude of the oscillator does not occur in this equation, so interesting behavior can only be observed if many of such oscillators are coupled. In fact, such a system can be used to study phenomena like synchronization, wave and pattern formation, phase chaos, or oscillation death [**?**, **?**]. It is a prominent example of an emergent system where the coupled system shows a more complex behavior than its constituents.

The concrete example we analyze here is a chain of nearest-neighbor coupled phase oscillators [**?**]:

$$\dot{\varphi}_i = \omega_i + \sin(\varphi_{i+1} - \varphi_i) + \sin(\varphi_i - \varphi_{i-1}). \tag{13}$$

The index i denotes here the i-th phase in the chain. Note, that the phase velocity is different for each oscillator.

From the implementation point of view, the main difference between the phase oscillator chain and the Lorenz attractor examples is that in the former example the values of neighboring vector elements are needed in order to compute the system function. In the Thrust version this is implemented with help of fancy iterators. First, we define device functors `left_nbr` and `right_nbr` returning left and right neighbor positions for the i-th element. Then we create a couple of permutation iterators from transformed counting iterators (with `left_nbr` and `right_nbr` used as transformation functors), pack the resulting iterators together with iterators `x`, `omega`, and `dxdt` into a `zip_iterator`. Finally we call the `thrust::for_each` algorithm with the accordingly defined system functor (Listing 16).

We use a similar technique for the VexCL version of the example. VexCL provides the `vex::permutation` function that allows to permute arbitrary expressions (Listing 17). Note how the use of C++11 **`auto`** keyword in Lines 38–40 allows us to conveniently capture intermediate expressions and thus simplify the code in Line 42.

Listing 16: po_thrust.cpp

```
25 typedef thrust::device_vector< double > state_type;
26 struct phase_oscillators {
33    struct left_nbr : thrust::unary_function<size_t, size_t> {
34       __host__ __device__ size_t operator()(size_t i) const {
35          return (i > 0) ? i - 1 : 0;
36       }
37    };
47    struct sys_functor {
48       template< class Tuple >
49       __host__ __device__ void operator()( Tuple t ) {
50          double phi_c = thrust::get<0>(t);
51          double phi_l = thrust::get<1>(t);
52          double phi_r = thrust::get<2>(t);
53          double omega = thrust::get<3>(t);
55          thrust::get<4>(t) = omega +
56             sin(phi_r - phi_c) + sin(phi_c - phi_l);
57       }
58    };
60    void operator() (const state_type &x, state_type &dxdt,
61          double dt)
62    {
63       BOOST_AUTO(start, thrust::make_zip_iterator(
64                thrust::make_tuple(
65                   x.begin(),
66                   thrust::make_permutation_iterator(
67                      x.begin(),
68                      thrust::make_transform_iterator(
69                         thrust::counting_iterator<size_t>(0),
70                         left_nbr()
71                         )
72                      ),
73                   thrust::make_permutation_iterator(
74                      x.begin(),
75                      thrust::make_transform_iterator(
76                         thrust::counting_iterator<size_t>(0),
77                         right_nbr(n - 1)
78                         )
79                      ),
80                   omega.begin(),
81                   dxdt.begin()
82                   )
83                )
84             );
86       thrust::for_each(start, start + n, sys_functor());
87    }
88 };
```

Listing 17: po_vexcl.cpp

```
25 typedef vex::vector<double> state_type;
26 struct phase_oscillators {
31    void operator()(const state_type &phi, state_type &dxdt,
32          double t) const
33    {
34       VEX_FUNCTION(left, size_t(size_t),
35             "return (prm1 > 0) ? prm1 - 1 : 0;");
36       VEX_FUNCTION(right, size_t(size_t, size_t),
37             "return (prm1 >= prm2) ? prm2 : prm1 + 1;");
39       auto idx = vex::element_index();
40       auto phi_l=vex::permutation(left(idx))(phi);
41       auto phi_r=vex::permutation(right(idx,phi.size()-1))(phi);
43       dxdt = omega + sin(phi_r - phi) + sin(phi - phi_l);
44    }
45 };
```

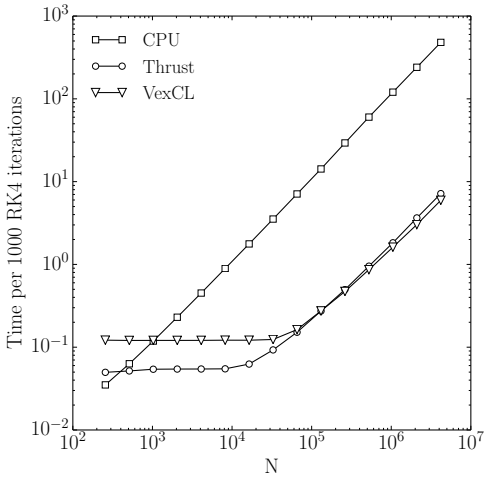

Fig. 2: Performance results for the chain of coupled phase oscillators example.

The performance results for the chain of coupled phase oscillators are presented in Figure 2. Again, the Thrust and the VexCL versions show similar results for large problems (with VexCL being faster by about 20%). The GPU versions are 70–80 times faster than the CPU version (which is the Thrust version compiled for the OpenMP backend). The higher acceleration w.r.t. the Lorenz attractor example is explained by the higher FLOP/byte ratio of the problem.

6.3 Molecular dynamics

Molecular dynamics (MD) are a simulation technique for a large number of small interacting particles, typically with local interaction forces. Examples are systems of molecules [**?**], granular systems [**?**], or coarse-grained models of fluid molecules.

Here, we study a two dimensional MD simulation described by the following equations of motion for particle i

$$m_i \ddot{x}_i = f_{loc}(x_i) + f_{fric}(\dot{x}_i) + \sum_{j \in S_i} f_{int}(x_i, x_j) \; . \tag{14}$$

m_i is the mass of the particle, f_{loc} is a local external force, for example the gravity. $f_{int}(x_i, x_j)$ is the (low-range) interaction between the particles i and j and the sum goes over all particles in an appropriate surrounding S_i of particle i. The second term is the friction which usually is only velocity dependent. Of course, other terms might also be included here, but for our purposes the above equation is generic enough to explain most details of implementing a molecular dynamics simulation. The restriction to two dimensions is easily generalizable to three dimensions. In fact, most of the following code is already independent of the concrete dimension.

For the interaction we use the Lennard-Jones potential [**?**]

$$f_{int}(x_i, x_j) = -\frac{r}{|r|} \frac{\mathrm{d}V}{\mathrm{d}r} \qquad \text{with} \quad r = x_i - x_j \tag{15}$$

with

$$V(r) = 4\varepsilon \left(\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right) . \tag{16}$$

It is used to describe the interaction of chemically unbounded atoms and molecules. Here ε is the strength of the interaction and σ denotes the interaction radius. The interaction decreases very fast with increasing distance of the particles $f \sim r^{-7}$. So, to speed up the simulations one usually restricts the interactions for particle i to particles withing its surrounding $S_i = \{ j : |x_i - x_j| < 4\sigma \}$. Of course, this means that mathematically the Lennard-Jones is not continuous anymore, but this is only of minor importance for our sample application. In practice several possibilities to overcome this discontinuity exist.

How can one implement such rather complicated systems of ODEs in a high-performance way on GPUs? The obvious idea would be to discard the locality of the potential and calculate all pairwise interaction for all particles. Unfortunately, this brute-force solution is far from being optimal. The computational complexity is $O(n^2)$ since all possible pairwise interactions are calculated. As explained above the interaction decreases very fast with increasing particle distance, so one should only take neighboring particles into account. In the following we present an algorithm for this problem and its GPU-implementation.

The basic idea is to assign particles to a regular grid of relatively large cells and calculate the interaction of particle i only with the particles located in neighboring

Listing 18: mdc_thrust_v2.cu

```
71  template< typename LocalForce , typename Interaction >
72  struct md_system_bs {
204   void operator()(point_vector const &x, point_vector const &v,
205         point_vector &a, double t) const
206   {
207     typedef thrust::counting_iterator< size_t > ci;
209     // Reset the ordering.
210     thrust::copy(ci(0), ci(prm.n), part_ord.begin());
212     // Assign each particle to a cell.
213     thrust::for_each(
214        thrust::make_zip_iterator( thrust::make_tuple(
215           x.begin(), cell_coo.begin(), cell_idx.begin()
216           ) ) ,
217        thrust::make_zip_iterator( thrust::make_tuple(
218           x.end(), cell_coo.end(), cell_idx.end()
219           ) ) ,
220        fill_index_n_hash( prm ));
222     // Sort particle numbers in part_ord by cell numbers.
223     thrust::sort_by_key(cell_idx.begin(), cell_idx.end(),
224           part_ord.begin());
226     // Find range of each cell in cell_idx array.
227     thrust::lower_bound(cell_idx.begin(), cell_idx.end(),
228           ci(0), ci(prm.n_cells), cells_begin.begin());
230     thrust::upper_bound(cell_idx.begin(), cell_idx.end(),
231           ci(0), ci(prm.n_cells), cells_end.begin());
233     // Handle boundary conditions
234     thrust::transform(x.begin(), x.end(), x_bc.begin(),
235           bc_functor(prm));
237     // Calculate the local and interacttion forces.
238     thrust::for_each(
239        thrust::make_zip_iterator( thrust::make_tuple(
240           x_bc.begin(), v.begin(), cell_coo.begin(),
241           ci(0), a.begin()
242           ) ),
243        thrust::make_zip_iterator( thrust::make_tuple(
244           x_bc.end(), v.end(), cell_coo.end(),
245           ci(prm.n), a.end()
246           ) ),
247        interaction_functor(cells_begin, cells_end, part_ord,
248           x, v, prm)
249        );
250   }
288 };
```

cells, see Listing 18. This method is also known as cell list algorithm. Another popular ansatz for the interaction computation, — the neighbor list — takes only the neighbors of particle i into account [**?**]. In the following we will only concentrate on the first method.

Listing 19: mdc_thrust_v2.cu

```
71  template< typename LocalForce , typename Interaction >
72  struct md_system_bs {
139   struct interaction_functor {
169     template< typename Tuple >
170     __host__ __device__ void operator()( Tuple const &t ) const {
171       point_type X = thrust::get<0>( t );
172       point_type V = thrust::get<1>( t );
173       index_type index = thrust::get<2>( t );
174       size_t cell_idx = thrust::get<3>( t );
176       point_type A = local_force(X, V);
178       for(int i = -1; i <= 1; ++i) {
179         for(int j = -1; j <= 1; ++j) {
180           index_type cell_index = index + index_type(i, j);
181           size_t cell_hash = get_cell_idx(cell_index, nx, ny);
182           for(size_t ii = cells_begin[cell_hash],
183                   ee = cells_end[cell_hash]; ii < ee; ++ii)
184           {
185             size_t jj = order[ii];
187             if( jj == cell_idx ) continue;
188             point_type Y = x[jj];
190             if( cell_index[0] >= nx ) Y[0] += xmax;
191             if( cell_index[0] < 0 ) Y[0] -= xmax;
192             if( cell_index[1] >= ny ) Y[1] += ymax;
193             if( cell_index[1] < 0 ) Y[1] -= ymax;
195             A += interaction(X, Y);
196           }
197         }
198       }
200       thrust::get<4>( t ) = A;
201     }
202   };
288 };
```

In the two-dimensional case each cell can be identified either by a two dimensional index (j_x, j_y) or by a one dimensional index $j = i_x + n_x j_y$ where n_x is the number of cells in x-direction. The ordering of the particles is done in two steps. First, the cell index j of each particle is calculated and stored in a vector `cell_idx`, lines 213–220. Secondly, the particles are sorted in ascending order according to the cell index. Of course, the vector of particles is not ordered itself. Instead, a vector with indices is created and sorted according to the cell indices. This is done by the `sort_by_key` algorithm from Thrust which sorts the first container and reorders the second container according to the order of the first one. The `part_ord` vector is then used as the index to refer to the original element in the particles vector. This kind of sort algorithm is also know as bucket sort [**?**].

The `cell_idx` vector now consists of a sorted array of the cell indices for each particles. Next we find the range (begin and end) for each cell in `cell_idx` which

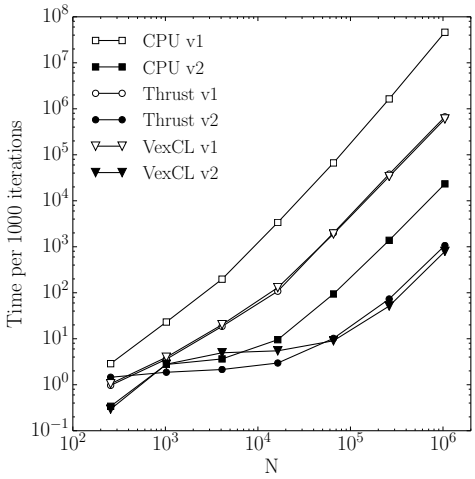

Fig. 3: Performance results for the molecular dynamics example.

corresponds to particles located in each of the cells (Lines 227–231). The range limits are stored in the `cells_begin` and `cells_end` arrays.

The final step is to compute the local forces and interactions for all particles, see Lines 238–249. Here we loop over all particles and velocities. The result is the acceleration which is stored in the vector `a`. The vector `cell_coo` contains the index of the cell in which the current particle is located. The interaction functor is shown in Listing 19. First, the local force is calculated in Line 176. Then two loops iterate over all neighboring cells of the current particle. Inside that loop the interaction between all particles in this cell and the particle is calculated. Lines 190–193 perform checks and corrections if particles are out of boundaries or are located on the opposite side of the considered domain.

At this point we only need to define the concrete solver type. A classical solver for molecular dynamic simulation is the Velocity-Verlet algorithm [**?**], which is used for second order ODEs and makes single RHS evaluation during one step. Here we use the implementation of the method from Boost.odeint.

The VexCL implementation follows the Thrust variant closely, so we omit the code for the sake of conciseness. VexCL provides `sort_by_key` primitive, and we had to implement `lower_bound` and `upper_bound` algorithms in form of custom VexCL functions. We also had to use custom kernel in order to compute the interaction force. The kernel source is very similar to the Thrust interaction functor (Listing 19). See `md_vexcl_v2.cpp` file for the complete VexCL solution.

Figure 3 shows performance results for the different versions of the molecular dynamics example. Versions denoted by "v1" implement the straight-forward algorithm with $O(n^2)$ complexity. "v2" versions employ the bucket sort optimization.

Both of the CPU versions use separate code which was again omitted from the text. The versions that use bucket sort optimization are expectedly faster than the "v1" algorithm. The Thrust and the VexCL versions show similar performance for large enough problems on the same hardware (with VexCL by 10–30% faster than Thrust). For both versions the GPU implementations are orders of magnitude faster than the CPU implementation (factor 75 for "v1" and 25 for "v2"). But the biggest performance boost comes from the algorithmic complexity reduction: e.g. the optimized VexCL version runs 300 times faster than the straight-forward one.

7 Summary and Conclusions

We have presented a high-level approach to compute numerical solutions of ODEs by developing a generic implementation of common ODE solvers. The proposed framework is very flexible and is able to adapt several CPU and GPU backends. The Thrust and the VexCL backends considered here are very different with respect to their interface design, but nevertheless are easily incorporated with our approach to generic algorithms. The proposed ideas and techniques are already implemented in the Boost.odeint library, which offers a vastly larger functionality, including more steppers and more backends.

Regarding the backend choice, it seems that the use of VexCL results in generally shorter and cleaner code for the kind of problems we considered here. Admittedly, for the more advanced molecular dynamics example we had to implement a custom OpenCL kernel, although the implementation was very similar to the corresponding Thrust functor. Performance-wise, VexCL showed slightly better results for the larger problems, but due to OpenCL initialization cost was slower for the smaller problem sizes. The main advantage of VexCL (and of OpenCL libraries in general) seems to be the larger set of supported hardware. It should be noted that Boost.odeint supports many other backends, which allows the user to choose the one best suited for the problem at hand, or the one they feel most comfortable with. This freedom is the great advantage of the modularized, generic design that we presented here for ODE solvers. It is clear that this technique can be applied to other numerical algorithms as well.

8 Acknowledgments

This work has been partially supported by the Russian Foundation for Basic Research (RFBR) grants No 12-07-0007 and 12-01-00333a. M. M. thankfully acknowledges financial support through the visitors program of the MPIPKS, Dresden (Germany).