Parallel implicit ordinary differential equation solver for cuda

Tomasz M. Kardaś

August 11, 2014

Chapter 1

Parallel Implicit Ordinary Differential Equations Solver

A simplest definition of stiffness, sufficient for purposes of the present thesis is as follows [1]:

A set of ordinary differential equations is said to be stiff when certain implicit methods perform their solution "better" than explicit ones.

The common symptom of equations stiffening is the fact that, to maintain the accuracy of the solution the explicit solver's marching step has to be reduced to unreasonably small size. On the other hand, in such cases, implicit methods often can be used for solution of such problems with high accuracy and a reasonably large step size.

For the ordinary differential equation defined as:

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}, t) \tag{1.1}$$

the explicit solver at each step calculates the function $\vec{f}$ with use of solution from the previous steps and uses it to compute the new solution. For example, if the explicit Euler method is used:

$$y(t_{i+1}) = y(t_i) + \Delta t f(y(t_i), t_i) \tag{1.2}$$

here $y(t_{i+1})$ is the new solution at time $t_{i+1} = t_i + \Delta t$ and $y(t_i)$ is the calculated earlier solution at t_i, Δt is the step size. On the other hand, the implicit solver uses the value of $\vec{f}$ calculated for the next step. For example, if the implicit Euler method is used:

$$y(t_{i+1}) = y(t_i) + \Delta t f(y(t_{i+1}), t_{i+1}). \tag{1.3}$$

Unlike in the first case, this second equation is a potentially nonlinear equation for $y(t_{i+1})$ which has to be solved (for example with compute-intensive Newton

c_k	a_{kj}
	b_j

Table 1.1: Butcher tableau for Runge-Kutta method, c_k are the nodes, b_j are the weights and a_kj are the elements of Runge-Kutta matrix.

1	1
	1

Table 1.2: Butcher tableau for implicite Euler method.

iterative method). Therefore, for non-stiff cases explicit methods are usually faster and thus preferable.

Modern graphical cards are specialized in parallel computation. By use of these graphical processing units (GPU) action can be perform simultaneously on the set of data as long as this action is the same for each data entry [2]. This is not a limitation in the present case – the procedure of ODE solving is identical for each value of the parameter. Thus, the parallel ODE solver was implemented for GPU[1].

The scheme proposed by E. Hairer and G. Wanner [1] for implicit Runge-Kutta method implementation was applied. The new solution $\vec{y}_{i+1}$ is calculated from previous $\vec{y}_i$ by:

$$\vec{y}_{i+1} = \vec{y}_i + \sum_{j=1}^{s} d_j \vec{z}_j, \tag{1.4}$$

where d_j are constants, s is the number of method stages and $\vec{z}_j$ are calculated as solution of algebraic equation set:

$$\vec{z}_j = \Delta t \sum_{k=1}^{s} a_{jk} f(x_j + c_k \Delta t, \vec{y}_i + \vec{z}_k), \tag{1.5}$$

where a_{jk} are the elements of Runge-Kutta matrix, c_k are Runge-Kutta method nodes. The three stage Radau IIA fifth order implicit Runge-Kutta method and implicit Euler method of second order were implemented. The values of elements of Runge-Kutta matrix, nodes and weights for these methods are presented in Butcher [6] tables 1.3. and 1.2, the construction of Butcher table is presented in table 1.1. The Eq. 1.5. is solved with a simplified Newton iteration, which uses the Jacobian matrix of the system [1]. A constant of 10 Newtnonian iteration are performed per step. There is no step-size control implemented and the positions (x_i) at which the solution is given are proposed by the user.

The user has to supply the system function `SystemFunction` and Jacobian `SystemJacobian` of the system in the C++ language:

```
#include "arch_settings.h"

template<typename T> DEVICE void SystemFunction(
 T *pDerivatives,
 T *pPreviousSolution,
 T fCoordinate,
```

[1] It is possible to use explicit ODE solving methods in parallel on GPU with use of odeint [3] library and one of a few GPU interface library [4, 5].

$\frac{4-\sqrt{6}}{10}$	$\frac{88-7\sqrt{6}}{360}$	$\frac{296-169\sqrt{6}}{1800}$	$\frac{-2+3\sqrt{6}}{225}$
$\frac{4+\sqrt{6}}{10}$	$\frac{296+169\sqrt{6}}{1800}$	$\frac{88+7\sqrt{6}}{360}$	$\frac{-2-3\sqrt{6}}{225}$
1	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{1800}$	$\frac{1}{9}$
	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{1800}$	$\frac{1}{9}$

Table 1.3: Butcher tableau for three stage Radau IIA implicit method of the fifth order [1].

```
 const T *pParams,
 const T *pConstParams)
{
        ...
}

template<typename T> DEVICE void SystemJacobian(
 T *pJacobian,
 T *pPreviousSolution,
 T fCoordinate,
 const T *pParams,
 const T *pConstParams)
{
        ...
}
```

here `T` is the floating point number type (`float` or `double`), `pDerivatives` and `pJacobian` points to memory area where calculated derivatives and elements of Jacobian should be placed, `pPreviousSolution` points to the solution from previous iteration, `fCoordinate` is the current value of time coordinate, `pParams` points at the memory containing parameter values specific for current instance, `pConstParams` points at the memory containing parameter values common for all instance, the `"arch_settings.h"` file contains required definitions and CUDA inclusions (`DEVICE` key word). The file containing `SystemFunction` and `SystemJacobian` definitions has to be included in CUDA C file of the following form:

```
#define FLOAT double

#include "harmonic_oscillator.hpp"
#include "RadauIIA.hpp"
#include "parallel_solver.hpp"
```

here, example `"harmonic_oscillator.hpp"` file is used, it contains definitions of `SystemFunction` and `SystemJacobian` for simple harmonic oscillator, by setting the `FLOAT` to `float` or `double` the precision of calculations can be chosen, the method of solution can be chosen by inclusion of `"RadauIIA.hpp"` for Radau IIA method or `"implicite_euler.hpp"` for implicit Euler method.

The CUDA C file can be either linked together wit other code, or can be compiled into a CUDA kernel [2] (Parallel Thread Execution "ptx" file) and used from other software, like MATLAB. In both cases the point of entry of the external code is through the following function:

```
GLOBAL void solveAllOde(
 FLOAT *pSolution,
 const FLOAT *pTimes,
 int iTSize,
 const FLOAT *pInitialCondition,
 int iProblemDimension,
```

```
const FLOAT *pParams,
int iParamsNumber,
int iParamsVariantsNumber,
const FLOAT *pConstParams,
FLOAT *pBuffer,
int iBufferUnitSize)
```

where `iProblemDimension` is the number of differential equations in the ODE set, `iParamsVariantsNumber` is the number of variants of parameter or initial condition values, `iParamsNumber` is the number of variable parameters per variant, `iTSize` is the number of time points at which the solution should be computed, `pSolution` points to the memory area where the solution of all the ODE sets for all parameter values and initial conditions will be placed (it should have a size of `iProblemDimension` × `iParamsVariantsNumber` × `iTSize`), `pTimes` points to values of time at which the solution will be computed, `pInitialCondition` points at values of initial conditions (size is `iProblemDimension` × `iTSize`), `pParams` points at values of parameters for all variants (size is `iParamsNumber` × `iParamsVariantsNumber`), `pConstParams` points to the memory containing parameters common for all variants, `pBuffor` is the method specific calculation buffer of size `iBufferUnitSize`. The size of the buffer depend on `iProblemDimension` and can be obtaine by use of macrodefinitions `RADAUIIA_BUFFOR_SIZE(iProblemDimension)` and `IMPLICITE_EULER_BUFFOR_SIZE(iProblemDimension)` for Radau and Euler methods respectively. The `solveAllOde` function distributes the parameter values, initial conditions and memory buffers to particular instances of ODE solvers. The code of the library has been published on the internet [7].

Bibliography

[1] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II Stiff and Differential-Algebraic Problems* (Springer-Verlag, 1980).

[2] NVIDIA, http://www.nvidia.pl/, *NVIDIA CUDA getting started guide for Microsoft Windows*.

[3] K. Ahnert and M. Mulansky, *Boost Numeric Odeint library documentation*, http://www.boost.org/.

[4] K. Ahnert, M. Mulansky, D. Demidov, K. Rupp, and P. Gottschling, "Solving odes with cuda and opencl using boost.odeint," in "FOSDEM 2013," (2013).

[5] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling, "Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries," SIAM Journal of Scientific Computing **35** (2013).

[6] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations I Nonstiff Problems* (Springer, 1993), 2nd ed.

[7] T. M. Kardaś, "Parallel implicit ordinary differential equation solver for cuda." .