European Congress on Computational Methods
in Applied Sciences and Engineering (ECCOMAS 2012)
J. Eberhardsteiner et.al. (eds.)
Vienna, Austria, September 10-14, 2012

COMPUTING REACHABLE SETS VIA BARRIER METHODS ON SIMD ARCHITECTURES

Lars Grüne[1], **Thomas U. Jahn**[2]

Chair of Applied Mathematics
Mathematical Institute
University of Bayreuth

e-mail: {[1]lars.gruene, [2]thomas.jahn}@uni-bayreuth.de

Keywords: reachable set, feasibility problem, sparse linear equation system, Runge-Kutta method, CUDA, parallelization, lower arrow form

Abstract. *We consider the problem of computing reachable sets of ODE-based control systems parallely on CUDA hardware. To this end, we modify an existing algorithm based on solving optimal control problems.*

The idea is to simplify the optimal control problems to pure feasibility problems instead of minimizing an objective function. We show that an interior point algorithm is well suited for solving the resulting feasibility problems and leads to a sequence of linear systems of equations with identical matrix layout. If the problem is defined properly, these matrices are sparse and can be transformed into a hierarchical lower arrow form which can be solved on CUDA hardware with sparse linear algebra and Cholesky's method.

We demonstrate the performance of our new algorithm by computing the reachable sets of two test problems on a CPU implementation using several explicit and implicit Runge-Kutta methods of different order. The experiments reveal a significant speedup compared to the original optimal control algorithm.

1 INTRODUCTION

For analyzing the behavior of control systems

$$\begin{array}{rcl} \dot{x}(t) & = & f(t, x(t), u(t)) \\ x(t) & \in & \mathcal{X} \\ u(t) & \in & \mathcal{U} \\ x(t_0) & = & x_0 \in \mathcal{X} \end{array} \tag{1}$$

with state and control constraint sets $\mathcal{X} \subset \mathbb{R}^n$ and $\mathcal{U} \subset \mathbb{R}^m$, the reachable sets

$$\mathcal{R}(T) = \{x(T) \in \mathcal{X} \mid x(\cdot) \text{ is solution of (1) with any valid } u(\cdot)\},\ T > t_0, \tag{2}$$

play an important role. Here, we call a control function $u \in L_\infty([t_0, T], \mathcal{U})$ valid if $x(t) \in \mathcal{X}$ holds for all $t \in [t_0, T]$. Provided an efficient algorithm for the approximate computation of $\mathcal{R}(T)$ is available, these sets can be used as the basis for solving problems like collision avoidance or forecasting the whole possible future behavior of the system.

Several different approaches to compute $\mathcal{R}(T)$ are proposed in the literature, like level–set methods [10], trajectory based optimal control [1], Hamilton–Jacobi–Bellman PDE approaches [2] or adaptive subdivision techniques [6] (the last algorithm considers $\bigcup_{T \geq 0} \mathcal{R}(T)$). All these different concepts have one thing in common: the computation takes a lot of time, especially for n greater than three or four.

In this paper, we want to develop a very fast numerical algorithm for approximating $\mathcal{R}(T)$. To achieve this goal, we propose a massively parallel algorithm on nVidia CUDA enabled hardware devices [3]. Since such hardware is very demanding, most algorithms will not be suited for this purpose. Among the methods cited above, the optimal control approach by Baier and Gerdts [1] is the most suitable and the algorithm we present in this paper will be obtained from a modification of [1]. In order to explain the design of our algorithm, in the next section we first look at the requirements of the CUDA devices in detail.

2 PRINCIPLES OF SIMD ARCHITECTURES

The CUDA GPU (graphics processing unit) is a SIMD (single instruction multiple data) streaming processor which requires totally different programming strategies compared to a serial processor. In this section we outline — in a simplified way — those characteristics of the hardware which motivate our decisions concerning the algorithm design. For details we refer to the CUDA documentation [3].

2.1 SIMD and thread enumeration

A GPU consists of several multiprocessors, each containing 32 cores[1]. Unlike a CPU, these multiprocessors cannot act independently. Instead, each of them has to process the same program stream, although not synchronously. The cores of a multiprocessor work even more restrictively: Each core runs one thread at the same time and has to process exactly the same instruction as the other cores of the multiprocessor (or do nothing), only the processed data may differ. Figure 2.1 illustrates this principle.

To assign a data element to the corresponding thread, an enumeration of the threads is mandatory. The GPU provides a hierarchical grid enumeration. The threads running on one

[1]Here, we consider the FERMI architecture that provides 32 cores and up to 48kB shared memory. These specifications differ on other architectures.

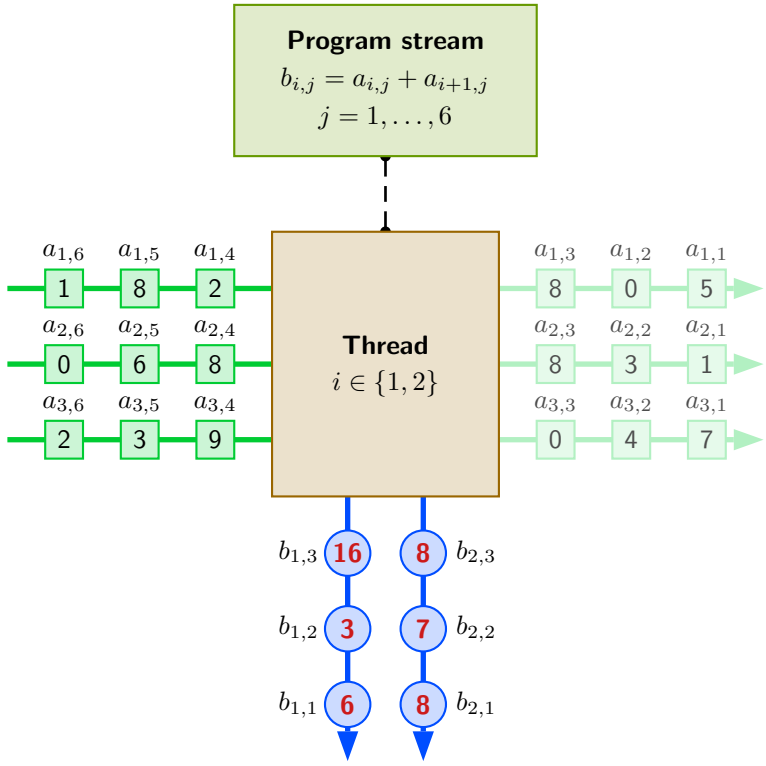

Figure 2.1: Illustration of a multiprocessor with two cores processing six addition instructions.

multiprocessor can be assigned to an element of a 3D-grid (up to 512 elements in total) called "Threadblock" (or simply "Block"). Several threadblocks can be arranged in a 2D-Grid (up to a maximum of 65535 $\times$ 65535 threadblocks), see Fig. 2.2.

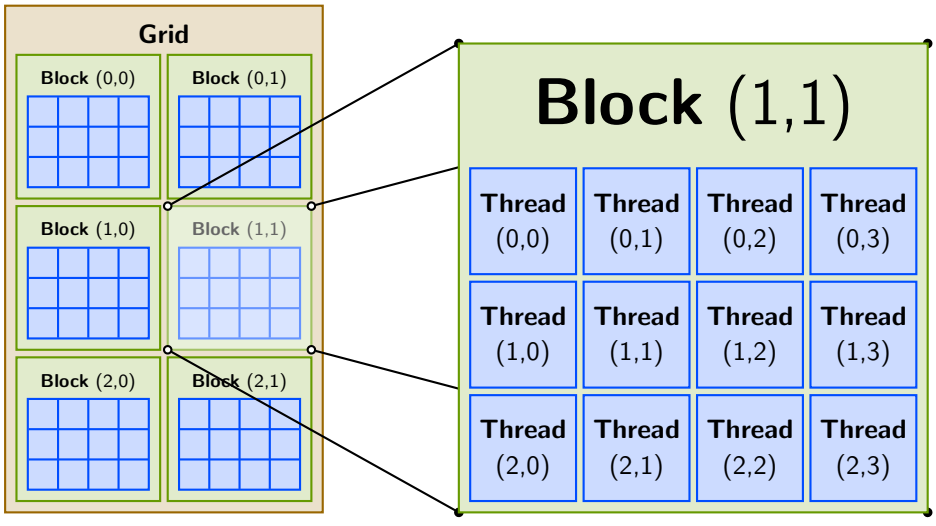

Figure 2.2: Example of a thread enumeration. In example the threadblocks just have 2D-indices.

When a program stream is launched on the GPU, each multiprocessor loads up to 8 threadblocks (depending on the required amount of registers and shared memory of a threadblock,

see Section 2.2) and executes these threads in parallel, divided into so called warps[2]. When a multiprocessor finished executing a threadblock, the next unexecuted threadblock is processed.

2.2 Memory considerations

Each threadblock can make use of up to 48kB memory of a multiprocessor, shared between all threads of the block[1]. Referencing variables stored in this shared memory is generally as fast as addition or subtraction instructions. This memory is accessed by up to 8 concurrently processed threadblocks. Hence, high memory usage of a threadblock leads to a lower GPU load, which may reduce execution speed since intelligent scheduling of warp execution can be affected[3]. Ideally, a threadblock should not assign more than 6kB of shared memory.

The larger memory (e.g., 3GB on Tesla C2050 devices) on GPUs is called device memory. This memory is shared between all multiprocessors of a card, but the access is very slow compared to shared memory, often up to 100 times slower. Algorithms that need to share data via device memory after every few operations will not achieve good performance and are not suitable for execution on a GPU. Unfortunately, using the device memory cannot entirely be avoided, because at least input- and output-data of an algorithm has to be stored on it.

2.3 Suitable algorithms

First of all, an algorithm must be able to run a "long" time compared to the amount of data that has to be accessed on the device memory. As a rule of thumb, at least a thousand instructions (additions, multiplications, ...) per device memory access should be processed, the more the better (see [8] for some benchmark tests).

To provide a full GPU load of 48 warps per multiprocessor[4] with 14 multiprocessors (as, e.g., on a Tesla C2050), the algorithm should be able to be executed on at least 21504 parallel threads. Each block must be able to run independently of other blocks, because blocks cannot be synchronized.

Of course, these numbers are just examples, but they underline the main requirements on the algorithm: huge parallelization bandwidth, a lot of independent parts, very low memory consumption and many calculation instructions per memory access. This explains why the design of such algorithms is a challenging task.

3 ALGORITHM SPECIFICATION

3.1 The approach of Baier and Gerdts

The starting point of our algorithm is the optimal control approach by Baier and Gerdts [1]. In this approach, the (rectangular) domain $\mathcal{X} = [x_1^l, x_1^u] \times \cdots \times [x_n^l, x_n^u]$ is approximated by a discrete grid

$$\mathcal{G} = \left\{ (x_1, \ldots, x_n) \mid x_i = x_i^l + \frac{x_i^u - x_i^l}{G_i - 1} \cdot k, \quad k = 0, \ldots, G_i - 1 \right\} \tag{3}$$

with a number of $\prod_{i=1}^{n} G_i$ gridpoints. In contrast to other methods, the approach in [1] is very well suited for parallelization because for each point in the grid an independent computation

[2] In more detail: Each warp contains 32 threads which are executed parallely by the 32 cores of a multiprocessor. The different warps are executed serially

[3] For example, the GPU will try to execute some warps during waiting for memory accesses being finished.

[4] 8 blocks, each with 192 threads is the optimal GPU load on FERMI cards [3][8].

is performed and no iteration on the grid involving data exchanges between the grid points is needed. More precisely, for every $\tilde{x} \in \mathcal{G}$ the following optimal control problem is solved.

$$\begin{aligned} \hat{u}_N &= \operatorname*{argmin}_{u_N} J(u_N) \\ J(u_N) &:= \|x_N(N, u_N) - \tilde{x}\|^2 \end{aligned} \tag{4}$$

subject to

$$x_N(j, u_N) \in \mathcal{X}, \quad u_{N,j} \in \mathcal{U}, j = 1, \ldots, N. \tag{5}$$

Here u_N is a sequence of N control values $u_{N,j} \in \mathbb{R}^m$, $j = 1, \ldots, N$ and $x_N(j, u_N)$ is the numerical approximation of $x(t_0 + hj)$ with initial value $x_0 = \tilde{x}$, solved via j Runge–Kutta–steps with the piecewise constant control function

$$u(t) \equiv u_{N,k} \quad \text{for} \quad t \in [t_0 + h(k-1), t_0 + hk[, \quad k = 1, \ldots, j \tag{6}$$

and time step size $h = (T - t_0)/N$. By this discretization, the optimal control problem is converted into a static nonlinear program (NLP). The reachable set of the Runge–Kutta discretized problem is denoted by $\mathcal{R}_h(T)$ and forms an approximation of $\mathcal{R}(T)$, for details see [1].

The approximation $\tilde{\mathcal{R}}$ of $\mathcal{R}_h(T)$ computed by this algorithm is now obtained by collecting all endpoints $x_N(N, \hat{u}_N)$ of the resulting optimal trajectories. In case $\tilde{x} \in \mathcal{R}_h(T)$, this endpoint coincides with $\tilde{x}$. Otherwise, $x_N(N, \hat{u}_N)$ is the point on the boundary of $\mathcal{R}_h(T)$ closest to $\tilde{x}$. Hence, in any case $x_N(N, \hat{u}_N) \in \mathcal{R}_h(T)$ holds and the resulting approximation is a set of points in $\mathcal{R}_h(T)$ with a very dense approximation of its boundary. Moreover, even in case the minimizer was not able to find a global minimum, $x_N(N, \hat{u}_N) \in \mathcal{R}_h(T)$ still holds and we obtain at least an inner approximation of $\mathcal{R}_h(T)$. Figure 3.1 shows the result of this algorithm for the Rayleigh problem from Section 5.1.

A closer look at Figure 3.1 reveals that for many points in the interior or $\mathcal{R}_h(T)$ the endpoints visualized in the figure do not coincide with the grid points. This is due to the fact that the optimization algorithm is often not able to deliver an exact optimal solution. Hence, although this algorithm complies with the basic requirements of parallelization, since all the minimization problems can in principle be solved independently, we propose a modified version resolving this problem before we turn to the parallel version.

3.2 An algorithm for computing reachable sets

The modified version of the algorithm from [1] is obtained by converting the optimization problem into a pure feasibility problem. This is based on the observation that, actually, we are not really interested in an optimal solution, but simply in *any* solution that reaches a neighborhood of the gridpoint $\tilde{x} \in \mathcal{G}$. We thus introduce an inequality constraint

$$\|x_N(N, u_N) - \tilde{x}\| \leq \varepsilon_G \tag{7}$$

where ε_G denotes the distance of two neighboring points in the grid, optionally enlarged[5] by a scaling factor $\eta \geq 1$. This condition ensures that the trajectory targets at a ε_G–region around $\tilde{x}$. The objective function of the optimization problem is then set to

$$J(\cdot) \equiv 0. \tag{8}$$

[5] Allowing the endpoint–constraints of neighboring grid points to overlap turned out to yield better results when computing reachable sets which are hypersurfaces.

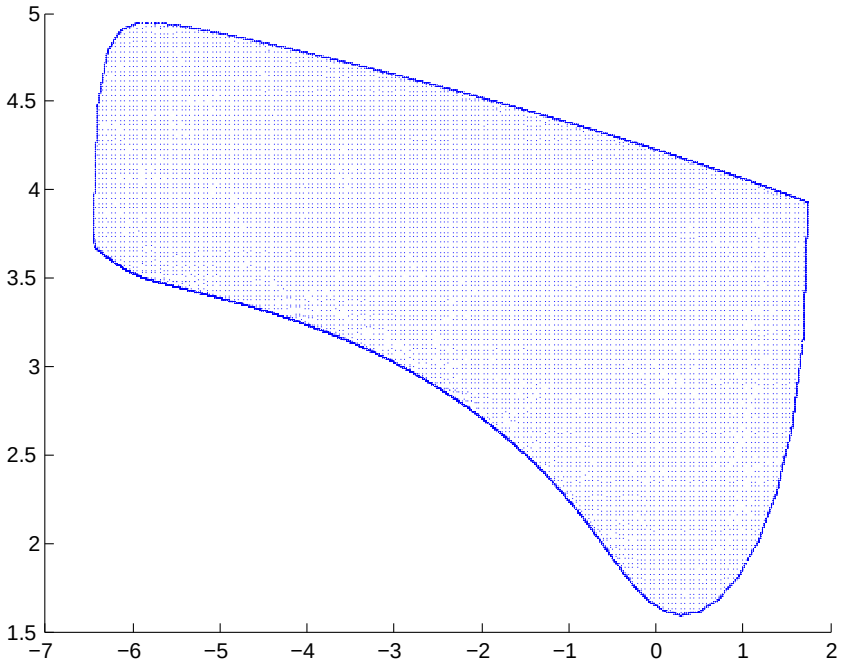

Figure 3.1: Reachable set of the Rayleigh problem (see Section 5.1) calculated via the optimal control method (grid: 200×200, $N = 20$).

The iterative optimization algorithm for solving the related NLP now computes control sequences $u_N^{(k)}$, $k = 1, 2, \ldots$ starting from an initial guess $u_N^{(0)}$. However, instead of iterating the algorithm until convergence, as termination condition for the optimizer loop we choose a penalty criterion of the form

$$P(u_N^{(k)}) := \sum_{\substack{i \\ g_i(u_N^{(k)})<0}} |g_i(u_N^{(k)})| < \nu\varepsilon_G, \tag{9}$$

where g_i are nonlinear inequality constraints including (7) and $\nu > 0$ is an accuracy parameter. In case $\tilde{x}$ is not reachable, inequality (9) will never be satisfied. However, then

$$\frac{P(u_N^{(k+1)})}{P(u_N^{(k)})} \to 1, \tag{10}$$

holds, which can be detected after few steps and serves as a termination criterion in this case. In order to reduce the number of iterations, the initial guess $u_N^{(0)}$ is chosen as the control sequence computed by the optimizer for an adjacent point, a method called warmstarting.

Assuming the reachable set $\mathcal{R}_h(t)$ is connected, the approximation $\tilde{\mathcal{R}}$ can now be computed by the following algorithm.

Algorithm Initialization: choose *any* valid u_N, set $\tilde{\mathcal{R}} := \emptyset$ and FIFO–Buffer $\mathcal{F} := \emptyset$.

(A1) Determine the grid point $\tilde{x}$ next to $x_N(N, u_N)$, set $\tilde{\mathcal{R}} \leftarrow \tilde{\mathcal{R}} \cup \{\tilde{x}\}$

(A2) For all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ with $\tilde{x}_i \notin \tilde{\mathcal{R}}$ do $\mathcal{F} \leftarrow (\tilde{x}_i, u_N)$.

(A3) If $\mathcal{F} = \emptyset$, exit

(A4) Pop $(\tilde{x}, \tilde{u}_N) \leftarrow \mathcal{F}$ from buffer.

(A5) **Solve the feasibility problem:** search u_N that solves $P(u_N) < \nu\varepsilon_G$, use $u_N^{(0)} := \tilde{u}_N$ for warmstarting.

(A6) If (A5) fails, go to (A3), otherwise go to (A1)

The advantages of this algorithm over the original problem (4) are as follows.

- The algorithm only processes gridpoints which are part of $\tilde{\mathcal{R}}$ (except for some points near the boundary of $\tilde{\mathcal{R}}$). Particularly on small sets or hypersurfaces this fact will lead to a significant speedup.
- Step (A5) will terminate immediately after a feasible control sequence has been found. No final iterations to achieve a good accuracy in minimizing an objective function are required.
- Using $\tilde{u}_N$ for warmstarting in step (A5) is essential. For most of the grid points, the control sequences that lead to two neighbored grid points are almost identical. Therefore step (A5) will usually require only two or three iterations (depending on the fineness of the grid).

At a first glance, the warmstarting seems to contradict the requirement that the problems in step (A5) can be solved independently which is crucial for an efficient parallelization. However, after very few executions of the loop (A1)–(A6) the number of elements in the buffer $\mathcal{F}$, i.e., the amount of data for which the Steps (A4)–(A5) can be executed in parallel, will be so large that this is not a serious limitation. Numerical experiments show that the loss in performance during the startup phase of the algorithm is by far compensated by the benefits of the warmstarting.

3.3 Distributing the algorithm to the CUDA hardware

While programming the GPU, the probably most important problem is the partitioning of the algorithm and the distribution of the parts to threads and threadblocks. Unfortunately, there is no golden rule or recipe for doing that since it highly depends on the structure of the algorithm.

We aim for solving (A5) on one single threadblock. This requires an efficient feeding of at least 32 (better 128) threads with similar instructions. After implementing this step, the compiler will issue the required ressources of the GPU program, particularly the needed amount of shared memory. This information is used to compute the total amount of threadblocks that can be processed by one multiprocessor at the same time (see [3]). Multiplied with the number of available multiprocessors, this gives us the total amount M of threadblocks that can be executed simultanously.

This leads to the following algorithm which is processed by M threadblocks in parallel[6]. On each threadblock, step (G7) is then distributed onto at least 32 parallel threads. Details of this distribution will be explained in the subsequent section.

[6]As already mentioned, the threadblocks do not process the steps simultanously. In order to avoid conflicts in the buffer access, mutex locks are used which force all threadblocks to pause until the locking threadblock issues the unlock command.

GPU–Algorithm Initialization before starting the GPU program: Set $\tilde{\mathcal{R}} := \emptyset$ and $\mathcal{F} := \emptyset$. Choose *any* valid u_N, do (G10) and (G11) once. Set $d = 0$. Run the GPU program with M threadblocks.

(G1) Lock mutex

(G2) If $\mathcal{F} \neq \emptyset$, goto (G6)

(G3) Set $d \leftarrow d + 1$

(G4) If $d = M$, exit

(G5) Unlock mutex, wait some time, lock mutex, set $d \leftarrow d - 1$ and goto (G2)

(G6) Pop $(\tilde{x}, \tilde{u}_N) \leftarrow \mathcal{F}$ from buffer, unlock mutex.

(G7) **Solve the feasibility problem:** search u_N that solves $P(u_N) < \nu\varepsilon_G$, use $u_N^{(0)} = \tilde{u}_N$ for warmstarting.

(G8) If (G7) fails, go to (G1)

(G9) Lock mutex

(G10) Determine grid point $\tilde{x}$, that is next to $x_N(N, u_N)$, set $\tilde{\mathcal{R}} \leftarrow \tilde{\mathcal{R}} \cup \{\tilde{x}\}$

(G11) For all gridpoints $\tilde{x}_i$ adjacent to $\tilde{x}$ with $\tilde{x}_i \notin \tilde{\mathcal{R}}$ do $\mathcal{F} \leftarrow (\tilde{x}_i, u_N)$.

(G12) Go to (G2)

4 SOLVING THE FEASIBILITY PROBLEM

The NLP corresponding to the feasibility problem in (G7) can be formulated in various ways. In order to solve (G7) for each $\tilde{x}$ with at least 32 threads and using at most 48kb of memory (less than 6kb would be best), the NLP has to be designed in a way that leads to a sparse NLP and this must be exploited in the minimization algorithm. To this end, we use a full discretization approach for the control problem which we explain in Section 4.2, below.

In order to ensure a high efficiency of the distribution of step (G7) onto the parallel threads, the structure and size of the numerical subproblems of the iterative minimization algorithm must always be identical. Barrier methods satisfy this requirement since the structure of these methods does not depend on activity of constraints and thus the linear equation system that has to be solved during the optimization steps always has the same layout. Hence, we will choose this method.

4.1 The interior–point algorithm

We use the interior–point algorithm describted at [9]. In our setting without objective function, the barrier problem of a basic interior point method is defined as

$$\begin{aligned} \min_{x,s} \quad & -\mu \sum_i \log s_i \\ \text{subject to} \quad & h(x) = 0 \\ & g(x) - s = 0 \end{aligned} \tag{11}$$

where x is the optimization variable, s is a vector of slack variables, g are inequality and h equality restrictions and $\mu \to 0$. The additional requirement $s \geq 0$ will be automatically satisfied due to the terms $-\log s_i$ in the optimization objective.

The solution of (11) will be obtained by solving the KKT–conditions

$$\begin{aligned} -\frac{\partial}{\partial x}h(x)y - \frac{\partial}{\partial x}g(x)z &= 0 \\ -\mu S^{-1}e + z &= 0 \\ h(x) &= 0 \\ g(x) - s &= 0 \end{aligned} \tag{12}$$

where S is a diagonal matrix containing the entries of s, $e = (1, \ldots, 1)^T$ and y, z are Lagrange multipliers. Using Newton's method to compute x, s, y, z requires to solve

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} & 0 & \frac{\partial}{\partial x}h(x)^T & \frac{\partial}{\partial x}g(x)^T \\ 0 & \Sigma & 0 & -I \\ \frac{\partial}{\partial x}h(x)^T & 0 & 0 & 0 \\ \frac{\partial}{\partial x}g(x)^T & -I & 0 & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_s \\ -p_y \\ -p_z \end{pmatrix} = - \begin{pmatrix} -\frac{\partial}{\partial x}h(x)^T y - \frac{\partial}{\partial x}g(x)^T z \\ z - \mu S^{-1}e \\ h(x) \\ g(x) - s \end{pmatrix} \tag{13}$$

iteratively, where

$$\mathcal{L}(x, s, y, z) = -h(x)^T y - (g(x) - s)^T z \tag{14}$$

and Σ is a diagonal matrix with the entries z_i/s_i. Solving (13) with $x = x^{(i)}$, $s = s^{(i)}$, $y = y^{(i)}$ and $z = z^{(i)}$, the next iterate of Newton's method is given by

$$\begin{aligned} x^{(i+1)} &= x^{(i)} + \alpha_s p_x \\ s^{(i+1)} &= s^{(i)} + \alpha_s p_s \\ y^{(i+1)} &= y^{(i)} + \alpha_z p_y \\ z^{(i+1)} &= z^{(i)} + \alpha_z p_z \end{aligned} \tag{15}$$

with suitably chosen step lengths α_s, α_z (see [9] for details). To sparsify the Hessian $\nabla^2_{xx}\mathcal{L}$, the matrix will be approximated by a limited memory BFGS–approach

$$\underbrace{\begin{pmatrix} \xi I & 0 & \frac{\partial}{\partial x}h(x)^T & \frac{\partial}{\partial x}g(x)^T \\ 0 & \Sigma & 0 & -I \\ \frac{\partial}{\partial x}h(x)^T & 0 & 0 & 0 \\ \frac{\partial}{\partial x}g(x)^T & -I & 0 & 0 \end{pmatrix}}_{=:H} + \underbrace{\begin{pmatrix} L \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{=:U} \underbrace{\begin{pmatrix} KL^T & 0 & 0 & 0 \end{pmatrix}}_{=:V^T}. \tag{16}$$

Using the Sherman–Morrison–Woodbury (SMW) formula for computing $(H + UV^T)^{-1}$ results in the inversion of H instead of solving (13), see [9]. This advantage comes at the expense of a further matrix inversion when applying the SMW formula. However, choosing a very short BFGS history compared to the large optimization vector makes the computational effort of this additional inversion negligible.

4.2 Defining the restrictions

We have to define g and h such that $\frac{\partial}{\partial x}g(x)$ and $\frac{\partial}{\partial x}h(x)$ are sparse matrices. Using the Runge–Kutta–Method [4] with r stages

$$\begin{array}{c|c} c & A \\ \hline & b \end{array} \tag{17}$$

with $A \in \mathbb{R}^{r\times r}$ and $c, b \in \mathbb{R}^r$, $x_N(N, u_N)$ can be calculated via

$$\begin{array}{rcl} k_{i,j} & = & f(t_{i-1} + c_j h, x_{i-1} + h\sum_{l=1}^r a_{j,l}k_{i,l},),\ j = 1,\dots,r \\ x_i & = & x_{i-1} + h\sum_{l=1}^r b_l k_{i,l} \end{array} \tag{18}$$

with $i = 1,\dots,N$ and $x_i := x_N(i, u_N)$. The full discretization approach now consists of introducing all x_i and $k_{i,j}$ as additional optimization variables and including the equalities in (18) as additional equality constraints. We thus concatenate $u_{N,i} \in \mathbb{R}^m$, $x_i \in \mathbb{R}^n$ and $k_{i,j} \in \mathbb{R}^n$ into one vector of optimization variables

$$x := (u_{N,1},\dots,u_{N,N}, x_1,\dots,x_n, k_{1,1},\dots,k_{1,r},\dots,k_{N,1},\dots,k_{N,r})^T \in \mathbb{R}^{N(m+n(r+1))} \tag{19}$$

The equality constraints can now be defined as

$$h(x) = \begin{pmatrix} f_{1,1} - k_{1,1} \\ \vdots \\ f_{1,s} - k_{1,s} \\ x_0 + h\sum_{l=1}^s b_l k_{1,l} - x_1 \\ \vdots \\ f_{N,1} - k_{N,1} \\ \vdots \\ f_{N,s} - k_{N,s} \\ x_{N-1} + h\sum_{l=1}^s b_l k_{N,l} - x_N \end{pmatrix} = 0 \tag{20}$$

with

$$f_{i,j} := f(t_{i-1} + c_j h,\ x_0 + h\sum_{l=1}^s a_{j,l}k_{i,l}\ ,\ u_{N,i}). \tag{21}$$

In our implementation, we will only consider "reduced" boxed constraints of the form

$$\begin{array}{c} \tilde{x}_L \le P_L x_i \quad \text{and} \quad P_U x_i \le \tilde{x}_U \\ \tilde{u}_L \le u_{N,i} \le \tilde{u}_U \\ i = 1,\dots,N \end{array} \tag{22}$$

where P_L and P_U are (not necessarily square) permutation matrices, that allow to pick those components of x_i which have to be restricted[7]. Further, x_N has to comply with the endpoint condition (7). To allow different grain sizes for the different dimensions of the grid, we model this condition as

$$\tilde{x} - \varepsilon_G \le x_N \le \tilde{x} + \varepsilon_G, \quad \varepsilon_G \in \mathbb{R}^n \tag{23}$$

[7]Vector inequalities $a \le b$ for $a, b \in \mathbb{R}^n$ are meant componentwise, i.e., $a_i \le b_i$ for all $i = 1,\dots,n$.

We define the inequality constraints combining (22) and (23) as

$$g(x) = \begin{pmatrix} \tilde{u}_U - u_1 \\ u_1 - \tilde{u}_L \\ \vdots \\ \tilde{u}_U - u_N \\ u_N - \tilde{u}_L \\ \tilde{x}_U - P_U x_1 \\ P_L x_1 - \tilde{x}_L \\ \vdots \\ \tilde{x}_U - P_U x_{N-1} \\ P_L x_{N-1} - \tilde{x}_L \\ {P_U}^T \min^* \{\tilde{x}_U, P_U(\tilde{x} + \varepsilon_G)\} + \left(I - {P_U}^T P_U\right)(\tilde{x} + \varepsilon_G) - x_N \\ x_N - {P_L}^T \max^* \{\tilde{x}_L, P_L(\tilde{x} - \varepsilon_G)\} - \left(I - {P_L}^T P_L\right)(\tilde{x} - \varepsilon_G) \end{pmatrix} \geq 0 \tag{24}$$

with x defined as in (19),

$$\min^*\{a, b\} := \begin{pmatrix} \min\{a_1, b_1\} \\ \vdots \\ \min\{a_n, b_n\} \end{pmatrix}, \qquad a, b \in \mathbb{R}^n \tag{25}$$

and $\max^*\{a, b\}$ defined analogously. Obviously, $\frac{\partial}{\partial x} h(x)$ and $\frac{\partial}{\partial x} g(x)$ are sparse matrices with

$$\frac{\partial}{\partial x} h(x) = \left(\begin{array}{cc|cc|cc} \partial_u f_1 & & & & \partial_x f_1^A & \\ 0 & & -I & & B & \\ \hline & \ddots & \partial_x f_1 & & & \ddots \\ & & I & -I & & \\ \hline & & & \ddots & & \\ \hline & \partial_u f_N & & \partial_x f_{N-1} & & \partial_x f_N^A \\ & 0 & & I \quad -I & & B \end{array}\right) \tag{26}$$

$$\frac{\partial}{\partial x} g(x) = \left(\begin{array}{cc|ccccc|c} -I & & & & & & & \\ I & & & & & & & \\ & \ddots & & & & & & \\ & -I & & & & & & \\ & I & & & & & & \\ \hline & & -P_U & & & & & 0 \\ & & P_L & & & & & \\ & & & \ddots & & & & \\ & & & & -P_U & & & \\ & & & & P_L & & & \\ & & & & & & -I & \\ & & & & & & I & \end{array}\right) \tag{27}$$

$$\partial_x f_i := \begin{pmatrix} \partial_x f_{i,1} \\ \vdots \\ \partial_x f_{i,s} \end{pmatrix} \qquad \text{(abbreviating } \partial_x f_{i,j} := \partial f_{i,j}/\partial x\text{)} \tag{28}$$

$$\partial_u f_i := \begin{pmatrix} \partial_u f_{i,1} \\ \vdots \\ \partial_u f_{i,s} \end{pmatrix} \tag{29}$$

$$\partial_x f_i^A := \begin{pmatrix} a_{1,1}h\partial_x f_{i,1} - I & \cdots & a_{1,s}h\partial_x f_{i,1} \\ \vdots & \ddots & \vdots \\ a_{s,1}h\partial_x f_{i,s} & \cdots & a_{s,s}h\partial_x f_{i,s} - I \end{pmatrix} \tag{30}$$

$$B := \begin{pmatrix} hb_1 I & \cdots & hb_s I \end{pmatrix} \tag{31}$$

4.3 Exploiting sparsity

After applying the L–BFGS approach and using the SMW formula, (13) is replaced by a similar system with ξI in place of $\nabla^2_{xx}\mathcal{L}$. This system can be rearranged to

$$\overbrace{\begin{pmatrix} \xi I + \dfrac{\partial}{\partial x}g(x)^T \Sigma \dfrac{\partial}{\partial x}g(x) & \dfrac{\partial}{\partial x}h(x)^T \\ \dfrac{\partial}{\partial x}h(x) & 0 \end{pmatrix}}^{=:\hat{H}} \begin{pmatrix} p_x \\ -p_y \end{pmatrix} = \begin{pmatrix} \dfrac{\partial}{\partial x}h(x)^T y + \dfrac{\partial}{\partial x}g(x)^T \left(z - \Sigma g(x) + \mu S^{-1}e\right) \\ -h(x) \end{pmatrix} \tag{32}$$

$$\begin{aligned} p_s &= \frac{\partial}{\partial x}g(x)p_x + g(x) - s \\ p_z &= -\Sigma p_s - z + \mu S^{-1}e \end{aligned} \tag{33}$$

We identify the entries of Σ as

$$\Sigma := \begin{pmatrix} \sigma^u_{U,1} & & & & & & & & & & \\ & \sigma^u_{L,1} & & & & & & & & & \\ & & \ddots & & & & & & & & \\ & & & \sigma^u_{U,N} & & & & & & & \\ & & & & \sigma^u_{L,N} & & & & & & \\ & & & & & \sigma^x_{U,1} & & & & & \\ & & & & & & \sigma^x_{L,1} & & & & \\ & & & & & & & \ddots & & & \\ & & & & & & & & \sigma^x_{U,N} & & \\ & & & & & & & & & \sigma^x_{L,N} \end{pmatrix}, \tag{34}$$

where $\sigma^u_{U,i}$ is a diagonal matrix containing the z_j/s_j–elements, that correlate with the upper constraints of the i–th m–dimensional control vector ($\sigma^u_{L,i}$, $\sigma^x_{U,i}$ and $\sigma^x_{L,i}$ respectively), and define

$$\begin{aligned} \sigma^u_i &:= \xi I + \sigma^u_{U,i} + \sigma^u_{L,i}, \quad i = 1, \ldots, N \\ \sigma^x_i &:= \xi I + P_U^T \sigma^x_{U,i} P_U + P_L^T \sigma^x_{L,i} P_L, \quad i = 1, \ldots, N-1 \\ \sigma^x_N &:= \xi I + \sigma^x_{U,N} + \sigma^x_{L,N} \end{aligned} \tag{35}$$

which represent the evaluated entries of $\xi I + \frac{\partial}{\partial x}g(x)^T \Sigma \frac{\partial}{\partial x}g(x)$. The structure of the resulting symmetric sparse matrix $\hat{H}$ of (32) is illustrated in Fig. 4.1.

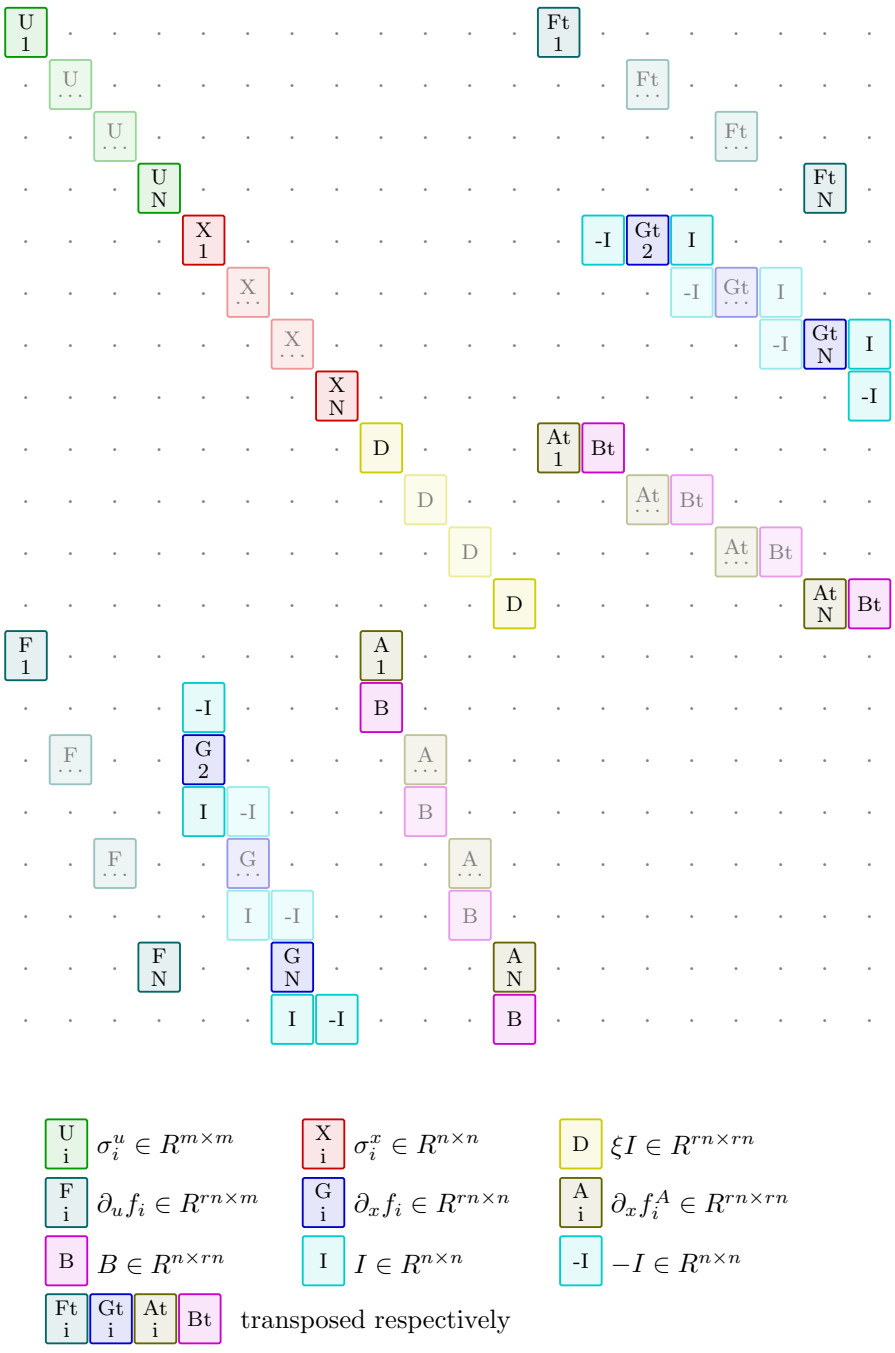

Figure 4.1: Sparse structure of the matrix $\hat{H}$ in (32).

To make use of the sparse structure we need an efficient way to solve the system of equations $\hat{H}x = c$ without eliminating zeros. Ideally, the amount of temporary memory while computing should only moderately exceed those needed to store the entries of $\hat{H}$. Additionally, we have to keep in mind, that the system has to be solved on at least 32 SIMD threads.

Our approach is to transform the system into lower arrow form

$$\begin{pmatrix} D_1 & & & N_1^T \\ & \ddots & & \vdots \\ & & D_N & N_N^T \\ N_1 & \cdots & N_N & M \end{pmatrix} \begin{pmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_N \\ X \end{pmatrix} = \begin{pmatrix} \hat{c}_1 \\ \vdots \\ \hat{c}_N \\ C \end{pmatrix} \tag{36}$$

by symmetric permutations. After that the system of equations can be solved via

$$\begin{aligned} W &:= M - \sum_{j=1}^{N} N_j D_j^{-1} N_j^T, \qquad w := C - \sum_{j=1}^{N} N_j D_j^{-1} \hat{c}_j \\ X &= W^{-1} w, \qquad \hat{x}_i = D_i^{-1}(\hat{c}_i - N_i^T X), \qquad i = 1, \ldots, N \end{aligned} \tag{37}$$

as described in [5]. The transformation into lower arrow form can be performed by a symmetrical permutation of $\hat{H}$, which can be chosen such that the submatrices D_i of the transformed matrix are lower arrow shaped, as well, see Fig. 4.2. As a result, the solution of (37) can be divided into numerous small and independent matrix inversions which can be efficiently distributed onto the parallel threads. Additionally, σ_i^u and σ_i^x are diagonal matrices when solely using boxed constraints. Needless to say that this fact turns their inversion into a trivial task.

Due to the simple sparse structure of N_j, operations like $N_j D_j^{-1} N_j^T$ can be implemented very efficiently. It can also be shown, that W (and the equivalent matrices of the subproblems produced by computing D_j^{-1}) are symmetric and positive definite, so that Cholesky's method can be used. Moreover, W has a tridiagonal form made of $n \times n$ matrices and Cholesky's method can be tuned to make use of that structure.

5 NUMERICAL EXAMPLES

We test the algorithm by computing the approximations of reachable sets of two problem settings called *Rayleigh's problem* and *Kenderov's problem* in [1]. While our algorithm is designed for GPU implementation, so far only a parallel CPU version is implemented which we use for our numerical tests. More precisely, for our runtime–benchmarks we will use a F77–implementation running parallelly on two Intel Quadcore–CPUs with 2.0 GHz. For all tests we used $\eta = 1.2$, $\nu = 10^{-2}$, $\xi = 0.1$ (see (7), (9) and (32)) and a L-BFGS history of size 1 as algorithm parameters. As the algorithm does not actually work on points that do not belong to the set approximation, there is no need to consider a total gridsize for comparison issues. Thus, we will just mention the grain size of the used grids.

5.1 Rayleigh

The control system of the Rayleigh problem is defined as

$$\begin{aligned} \dot{x}(t) &= y(t) \\ \dot{y}(t) &= -x(t) + y(t)\,(1.4 - 0.14 y(t)^2) + 4u(t) \\ x(0) &= -5 \\ y(0) &= -5 \\ u(t) &\in [-1, 1] \\ t &\in [0, T], \;\; T = 2.5 \end{aligned} \tag{38}$$

To compare the algorithm's runtime with the algorithm of Baier and Gerdts, we use the same gain sizes and step sizes $h = T/N$ and Euler's method as ODE solver. The tests show a significant speedup compared to distance function approach of Baier and Gerdts (see Table 1).

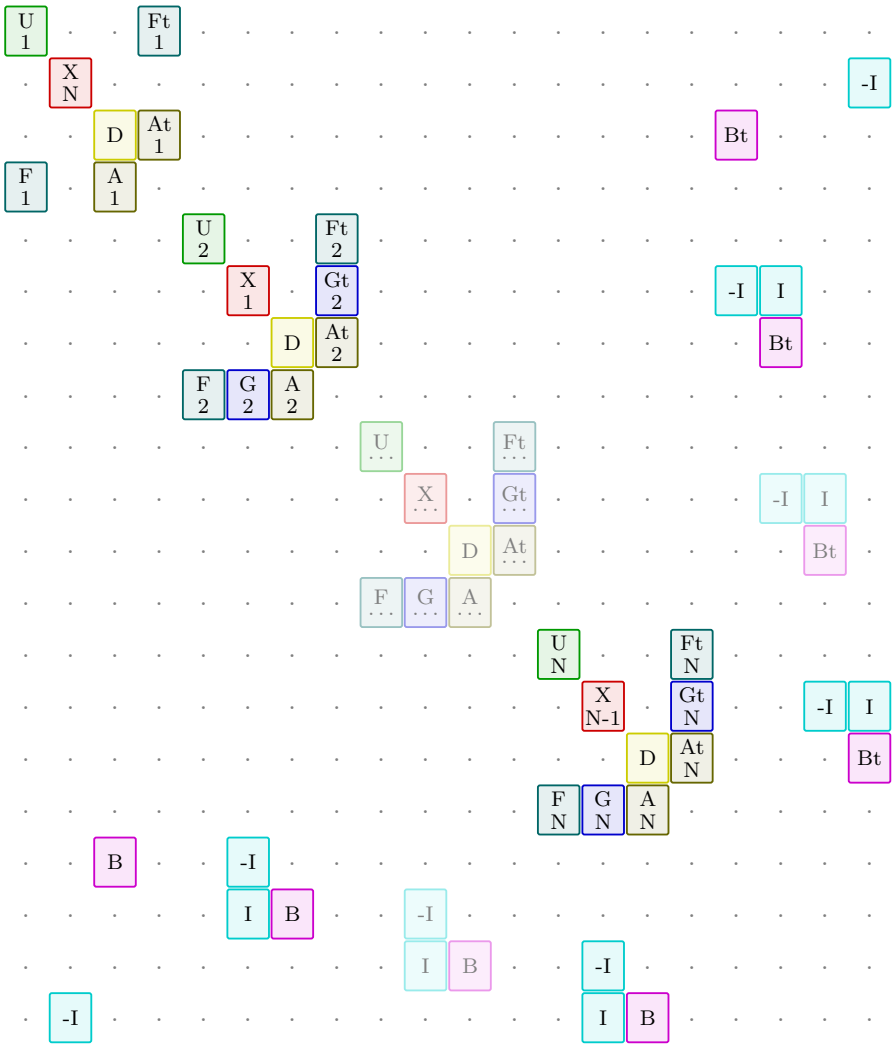

Figure 4.2: $\hat{H}$ transformed to lower arrow form by symmetrical permutations (see fig. 4.1 for symbol legend)

Figure 5.1 illustrates the plots of some sets for different step sizes and ODE solvers[8] with constant grain size. The thin black line denotes the border of the reference set[9]. The corresponding benchmarks are displayed in Table 2.

5.2 Kenderov

Kenderov's problem leads to a 1–dimensional reachable set shaped as a circle. Just working on lines will prevent the algorithm to "disseminate", so that could be a problem. The control

[8]Euler's method and Radau IA method with different orders, see [7].

[9]The reference set has been computed using a grid with grain size 10^{-3}, $N = 320$ and Radau (order 3) method and can be considered as almost exact.

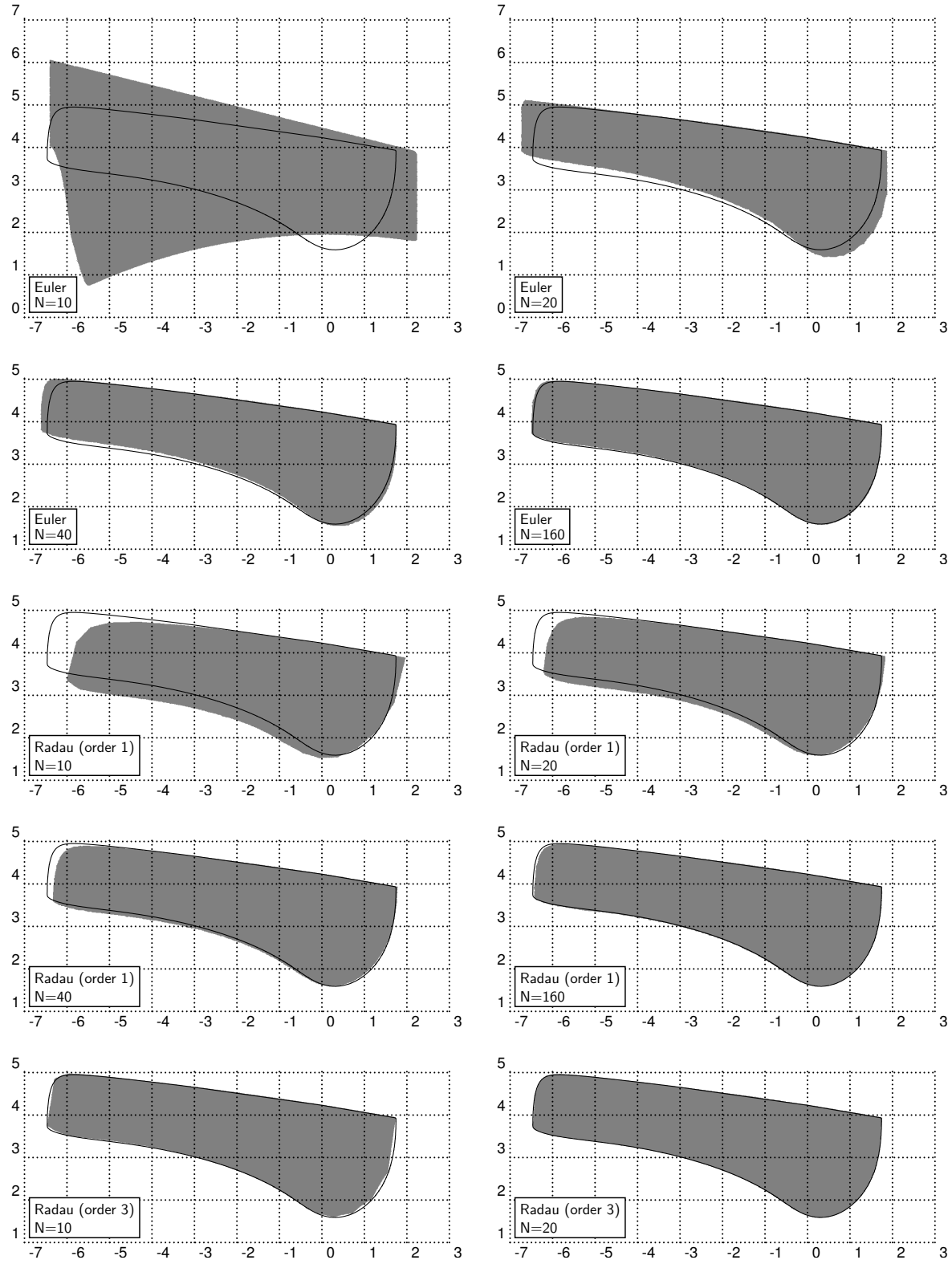

Figure 5.1: Approximation of Rayleigh's set with different ODE solvers, step sizes and constant grain size 10^{-2}.

N	Grain size	Runtime A1	Runtime A2	Runtime A3
10	$2.5 \cdot 10^{-1}$	2.076 s	0.148 s	0.053 s
20	$1.25 \cdot 10^{-1}$	24.302 s	1.988 s	0.157 s
40	$6.25 \cdot 10^{-2}$	424.194 s	33.334 s	0.901 s
80	$3.125 \cdot 10^{-2}$	—	684.691 s	6.010 s
160	$1.5625 \cdot 10^{-2}$	—	$\approx$ 19500 s	24.409 s

Table 1: Overview of the runtimes of Baier and Gerdts non-adaptive (A1) and adaptive algorithm (A2) compared to our algorithm using Euler's method (A3) while working on Rayleigh's problem.

N	Euler		Radau (order 1)		Radau (order 3)		Radau (order 5)	
	total [s]	point [10^{-5} s]	total [s]	point [10^{-5} s]	total [s]	point [10^{-5} s]	total [s]	point [10^{-5} s]
10	3.437	1.216	2.293	1.467	3.451	2.333	5.136	3.462
20	5.018	3.214	5.005	3.285	8.391	5.594	12.744	8.480
40	12.163	8.042	12.259	8.124	20.824	13.844	32.237	21.401
80	27.975	18.609	30.049	20.009	48.993	32.561	78.439	52.082
160	64.968	43.265	64.239	42.798	116.198	77.217	180.346	119.685

Table 2: Overview of the runtimes (total runtime and time per reachable gridpoint) of different step sizes and ODE solvers with constant grain size 10^{-2} while working on Rayleigh's problem.

system is defined as

$$\begin{aligned} \dot{x}(t) &= 8(a_{11}x(t) + a_{12}y(t) - 2a_{12}y(t)u(t)) \\ \dot{y}(t) &= 8(-a_{12}x(t) + a_{11}y(t) + 2a_{12}x(t)u(t)) \\ x(0) &= 2 \\ y(0) &= 2 \\ u(t) &\in [-1, 1] \\ t &\in [0, T], \;\; T = 1 \end{aligned} \tag{39}$$

with $a_{11} := \sigma^2 - 1$, $a_{12} := \sigma\sqrt{1-\sigma^2}$ and $\sigma := 0.9$.

Figure 5.2 shows the approximation of the reachable set with different step sizes and ODE solvers. Here, the thin black line denotes the whole exact reference set[10]. As one can see, the overlapping factor $\eta = 1.2$ ensures a good spreading of the algorithm, so that the whole set can be computed. Even when using higher order methods, that lead to a very thin approximation, the set is computed correctly. Surprisingly, the algorithm reveals some additional areas of the approximated reachable set, that have not been detected by the algorithm of Baier and Gerdts. The benchmarks for a constant grid, different step sizes and ODE solvers are shown in Table 3. The computation time per reachable point is considerably higher compared to the Rayleigh problem, which is due to the fact that most points lie close to the boundary of the reachable sets. Hence, the computation needs more time since many unreachable points have to be touched that are still "almost" reachable from the numerical point of view. Compared to the computation of

[10]The reference set has been computed on a grid with grain size 10^{-3}, $N = 500$ and Radau (order 3) method.

an inner point, for such points the optimizer loop needs much more iterations until it considers the point as unreachable.

Again, we observe a very good speedup compared to the distance function approach as shown in Table 4. Note that for small values of N the results of the two algorithms differ considerably and the algorithm from [1] computes much smaller sets, which is why the speedup only becomes visible for large N.

N	**Euler**		**Radau (order 1)**		**Radau (order 3)**		**Radau (order 5)**	
	total [s]	point [10^{-4} s]	total [s]	point [10^{-4} s]	total [s]	point [10^{-4} s]	total [s]	point [10^{-4} s]
20	9.684	0.370	1.500	1.522	2.526	17.701	3.500	24.423
40	10.315	2.216	3.939	4.586	4.527	30.776	6.496	42.278
80	19.196	13.154	11.146	17.275	8.251	56.279	12.399	83.834
160	33.774	52.120	26.763	59.499	16.212	109.247	24.624	164.490
320	34.000	129.671	32.915	136.463	33.917	225.062	53.118	350.384

Table 3: Overview of the runtimes (total runtime and time per reachable gridpoint) of different step sizes and ODE solvers with constant grain size 10^{-2} while working on Kenderov's problem.

N	**Grain size**	**Runtime A1**	**Runtime A2**	**Runtime A3**
20	$5 \cdot 10^{-2}$	1.296 s	0.152 s	0.644 s
40	$2.5 \cdot 10^{-2}$	14.313 s	0.752 s	1.469 s
80	$1.25 \cdot 10^{-2}$	234.151 s	5.980 s	4.300 s
160	$6.25 \cdot 10^{-3}$	$\approx$ 5208 s	66.528 s	16.76 s
320	$3.125 \cdot 10^{-3}$	$\approx$ 168155 s	$\approx$ 1283 s	75.487 s

Table 4: Overview of the runtimes of Baier and Gerdts non-adaptive (A1) and adaptive algorithm (A2) compared to our algorithm using Euler's method (A3) while working on Kenderov's problem.

6 CONCLUSIONS

We presented an optimal control based algorithm for reachable sets which is designed for massively parallel implementation on a GPU. Even in its parallel CPU implementation tested in this paper the algorithm significantly outperforms the method from [1] which we used as starting point for the design of our algorithm.

The speedup is mainly due to exploiting the sparse structure of the feasibility problem which is at the core of our algorithm. Moreover, the computational cost per grid point seems to increase only linearly with the number of time steps. As such, our new approach reduces the time needed for approximating reachable sets to an order of seconds instead of minutes or hours. A further significant speedup is expected for the future GPU implementation, since our algorithm already exhibits the necessary structure for this implementation.

Besides this GPU implementation, further research will aim at various extensions of the algorithm, removing some of the limitations of the current version. For instance, our approach

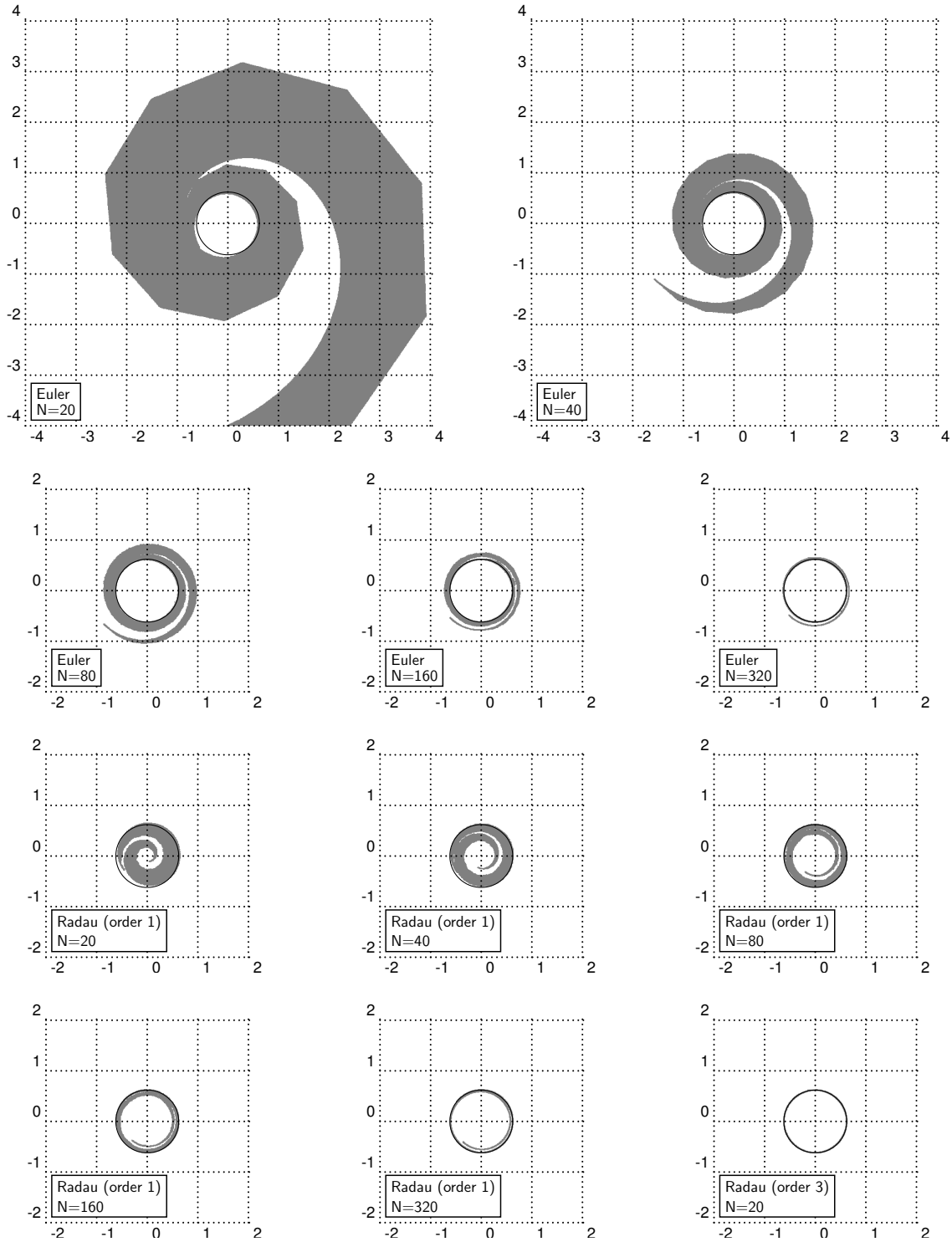

Figure 5.2: Approximation of Kenderov's set with different ODE solvers, step sizes and constant grain size 10^{-2}.

relies on the fact that the feasible control for two neighboring points in the reachable set is similar, for which there is, of course, no warranty. Violating this assumption will cause an incomplete approximation of the set. To handle this, tuning the optimizer for finding global minima will be necessary.

We note that the algorithm can be easily modified to compute lower dimensional projections of reachable sets for higher dimensional systems. First tests with the computation of a 2–dimensional projection of a reachable set for a 7–dimensional automotive model turned out to be very promising with an execution time in the order of seconds.

We finally remark that introducing an objective function to rate the control sequences on the right hand side of (13) does not affect the sparse structure. Hence, in principle, the structure can also be exploited for general nonlinear optimal control problems. However, in this case the optimizer has to be modified by using a filter and a merit function in order to balance the weight of objective function and constraint, see [11], [9].

REFERENCES

[1] R. Baier and M. Gerdts. A computational method for non-convex reachable sets using optimal control. In *Proceedings of the European Control Conference (ECC) 2009, Budapest, Hungary*, pages 97–102, 2009.

[2] O. Bokanowski, N. Forcadel, and H. Zidani. Reachability and minimal times for state constrained nonlinear problems without any controllability assumption. *SIAM J. Control Optim.*, 48(7):4292–4316, 2010.

[3] NVIDIA Corp. NVIDIA CUDA C programming guide, Version 3.2, 2010.

[4] P. Deuflhard and F. Bornemann. *Scientific computing with ordinary differential equations*, volume 42 of *Texts in Applied Mathematics*. Springer Verlag, New York, 2002.

[5] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[6] L. Grüne. Subdivision techniques for the computation of domains of attraction and reachable sets. In *Proceedings of NOLCOS 2001, St. Petersburg, Russia*, pages 762–767, 2001.

[7] M. Hermann. *Numerik gewöhnlicher Differentialgleichungen: Anfangs- und Randwertprobleme [Numerics of Ordinary Differential Equations: Initial and boundary value problems]*. Oldenbourg Verlag, Munich, 2004.

[8] T. Jahn. Implementierung numerischer Algorithmen auf CUDA–Systemen. Diploma thesis, University of Bayreuth, 2010.

[9] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Verlag, 2nd edition, 2006.

[10] J. A. Sethian. *Level set methods*, volume 3 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 1996.

[11] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006.