Int. J. Appl. Math. Comput. Sci., 2014, Vol. 24, No. 3, 551–566
DOI: 10.2478/amcs-2014-0040

amcs

HIGH–PERFORMANCE SIMULATION–BASED ALGORITHMS FOR AN ALPINE SKI RACER'S TRAJECTORY OPTIMIZATION IN HETEROGENEOUS COMPUTER SYSTEMS

ROMAN DĘBSKI

Department of Computer Science
AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: rdebski@agh.edu.pl

Effective, simulation-based trajectory optimization algorithms adapted to heterogeneous computers are studied with reference to the problem taken from alpine ski racing (the presented solution is probably the most general one published so far). The key idea behind these algorithms is to use a grid-based discretization scheme to transform the continuous optimization problem into a search problem over a specially constructed finite graph, and then to apply dynamic programming to find an approximation of the global solution. In the analyzed example it is the minimum-time ski line, represented as a piecewise-linear function (a method of elimination of unfeasible solutions is proposed). Serial and parallel versions of the basic optimization algorithm are presented in detail (pseudo-code, time and memory complexity). Possible extensions of the basic algorithm are also described. The implementation of these algorithms is based on OpenCL. The included experimental results show that contemporary heterogeneous computers can be treated as μ-HPC platforms—they offer high performance (the best speedup was equal to 128) while remaining energy and cost efficient (which is crucial in embedded systems, e.g., trajectory planners of autonomous robots). The presented algorithms can be applied to many trajectory optimization problems, including those having a black-box represented performance measure.

Keywords: trajectory optimization, heterogeneous computing, GPGPU, high-performance computing, alpine ski racing.

1. Introduction

Many problems related to trajectory optimization (e.g., in robotics, aerospace engineering or optimal control) cannot be solved analytically. This is the case, for instance, when the explicit formula of the cost functional (i.e., performance measure) is unknown[1] and its values can be obtained only from simulation. In such problems, derivative-related information is not available and, as a consequence, common practice[2] is to base the optimization process on one of many existing (meta-)heuristics[3] or on dynamic programming. Irrespective of the selected solution method, simulation-based optimization is practically always computationally complex mainly because of the cost of a single simulation, and for that reason it is usually solved in an HPC infrastructure. Nowadays this is usually a computer-cluster or cloud computing platform. However, in some situations this typical approach cannot be applied. This is the case, for instance, in many embedded systems[4] like trajectory planners of autonomous robots or, from a different category, tools supporting alpine ski racing course setters.

The aim of this paper is to present an effective approach to simulation-based continuous trajectory optimization using the example of a minimum-time problem taken from alpine ski racing. The presented algorithms are adapted to heterogeneous computer systems (i.e., composed of multiple processor types like CPU and GPU) and, as a consequence, can be the base for optimization tasks executed on both large-scale HPC-platforms and modern embedded multi-core/many-core systems[5].

One of the most important aspects of the proposed approach is the assumption that the algorithms are implemented in OpenCL (Open Computing

[1] That is, it is *opaque* or *black-boxed* to the optimization routine.
[2] Especially if the global optimum is to be found.
[3] For example, evolutionary algorithms, simulated annealing, particle swarm optimization, tabu search.
[4] Also real-time embedded systems.
[5] That is to say, which are composed of CPUs and GPUs, but possibly also of FPGAs and/or DSPs.

Language)—a parallel programming framework which, unlike CUDA (Compute Unified Device Architecture), is defined by an open standard and, by definition, is cross-platform. OpenCL implementations run now on widely used computing units, including CPUs and GPUs from Intel, AMD and NVIDIA, but also on DSPs and FPGAs. The OpenCL standard also defines the *OpenCL Embedded Profile*—a special version of the platform for embedded systems and mobile devices.

At the conceptual level, the key idea behind the presented algorithms is to use a grid-based discretization scheme to transform the continuous optimization problem into a search problem over a specially constructed finite graph, and then to apply dynamic programming to find the approximation of the global solution which, in the analyzed example, is the minimum-time ski line, represented as a piecewise-linear function. The cost of each edge in the graph is obtained from simulation, taking into account the corresponding boundary conditions (related to the values of linear velocity). When designing the algorithm, the main difficulty which had to be overcome was related to the kinematic constraints of the simulation model (the ski line has to be smooth).

The black-box represented performance measure and the use of OpenCL make the presented approach very general, both from the possible deployment point of view (typical HPC-platforms, modern embedded systems, mobile devices) and because of the scope of the optimization problems it covers.

This paper is organized as follows. The next section contains a review of related work. Following that, the optimization problem is defined (including a description of the simulation model). Next, the proposed solution methods are described. After that, experimental results are presented and discussed. The last section contains a conclusion of the study.

2. Related work

Probably the first person who formulated (scientifically) the trajectory optimization problem was Johan Bernoulli[6]. In June 1696 in *Acta Eruditorum* he challenged the “acutest mathematicians of the world” to solve *the brachistochrone problem*[7]. Among the first who found the solution was Johan’s brother Jakob. His solution is now considered (Stillwell, 2010) to be a major step in the development of the calculus of variations[8]—the field of mathematics which has played a crucial role in trajectory optimization (see, e.g., Stechert, 1963; Wuerl *et al.*, 2003).

Significant progress in the field of trajectory optimization was made in the 1950s thanks to the development of the digital computer and introduction of dynamic programming (Bellman, 1954), effective shortest path algorithms (Dijkstra, 1959; Bellman, 1958) and the Pontryagin maximum principle (Pontryagin *et al.*, 1962). These contributions, combined with Non-Linear Programming (NLP), have been the basis for many effective trajectory optimization methods which are nowadays often classified as either *direct* or *indirect* (see, e.g., von Stryk and Bulirsch, 1992; Betts, 1998; Lewis *et al.*, 2000; Szynkiewicz and Błaszczyk, 2011). The indirect methods, which are based on the calculus of variations, find solutions that satisfy the necessary conditions of optimality. The direct ones seek solutions having (locally) the best value of the performance measure.

A special group of trajectory optimization problems consists of those with *black-box represented* objective functions. In this case, most of the classic optimization methods cannot be used (at least not directly) and the optimization process is often based on one of the soft-computing methods (see, e.g., Pošík and Huyer, 2012; Pošík *et al.*, 2012; Szłapczyński and Szłapczyńska, 2012; Vasile and Locatelli, 2009; Ceriotti and Vasile, 2010).

Another important class of trajectory optimization methods comprises those based on graph shortest path algorithms (see, e.g., Crauser *et al.*, 1998; Rippel *et al.*, 2005; Dramski, 2012). Some research has been done on parallelization of these algorithms (see, e.g., Crauser *et al.*, 1998; Jasika *et al.*, 2012) including the possibility of their GPU-acceleration (see, e.g., Harish and Narayanan, 2007; Singla *et al.*, 2013).

As for trajectory optimization case studies, many of the existing ones are related to aerospace engineering (see, e.g., Yokoyama, 2002; Rippel *et al.*, 2005; Ceriotti and Vasile, 2010) but there are also a few others (see, e.g., Vanderbei, 2001) and among them some related to alpine ski racing (see, e.g., Kaps *et al.*, 1996; Hirano, 2006; Brodie, 2009).

3. Problem formulation

The trajectory optimization task is *to find, among all admissible trajectories, the one with the best value of the performance measure*. A system performance measure can be written in the following (general) form[9]:

$$J = h\big(\boldsymbol{x}(t_f), t_f\big) + \int_{t_0}^{t_f} g\big(\boldsymbol{x}(t),\ \boldsymbol{u}(t),\ t\big)\,\mathrm{d}t, \quad (1)$$

where t_0 and t_f are the initial and final times, respectively, respectively, h and g are scalar functions.

[6]Galileo may have been the first to consider the problem of finding the path of quickest descent (Babb and Currie, 2008).

[7]This problem is discussed in a broad sense by Sussmann and Willems (1997; 2002).

[8]The others with great contributions in the calculus of variations are Euler, Lagrange, Legendre, Hamilton and Weierstrass.

[9]This form is commonly used in control theory.

Sometimes a closed-form formula of the performance measure (i.e., the objective functional) is unknown, i.e., it is "opaque" (or black-boxed) to the optimization routine as shown in Fig.1. In this figure the inputs $x_1, x_2, \ldots, x_n$ represent a trajectory (encoded in some way) and J is a performance measure. A typical example of this situation is when objective functional values are taken from a computer simulation.

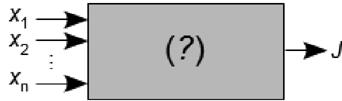

Fig. 1. Black-box functional (the value of J that corresponds to the input can be found only through simulation).

The trajectory optimization problem analyzed in this paper is taken from alpine ski racing (see Fig. 2). This problem can be treated as an example of a two-dimensional, constrained, continuous trajectory optimization task having a black-box (i.e., arbitrary complex) represented objective functional. The constraints in the optimization task are related to the setting of the course, which geometrically defines all admissible trajectories, and to the dynamics of the skier which is described in the next two subsections.

3.1. Alpine skier's model. Consider a skier (modeled as a material point of mass m) going down a slope with an angle α (see Fig. 3(b)). We assume that all turns are purely carved (i.e., with no skidding). The instantaneous center of the skiing line curvature is at point o_c and the corresponding curvature is equal to κ, while $\vec{v}$ and $\dot{\varphi}$ are instantaneous velocities: linear and angular, respectively. All the forces drawn in Fig. 3(b) are described below.

3.1.1. Equations of motion. After applying Newton's second law for directions ξ_1 and ξ_2 (see Fig. 3(b)), we get

$$\begin{cases} m\ddot{\xi}_1 = mg\sin\alpha - F_{fd}\frac{\dot{\xi}_1}{\sqrt{\dot{\xi}_1^{\,2}+\dot{\xi}_2^{\,2}}} \\ \qquad - F_r\frac{\dot{\xi}_2}{\sqrt{\dot{\xi}_1^{\,2}+\dot{\xi}_2^{\,2}}}\operatorname{sgn}\dot{\varphi}, \\ m\ddot{\xi}_2 = -F_{fd}\frac{\dot{\xi}_2}{\sqrt{\dot{\xi}_1^{\,2}+\dot{\xi}_2^{\,2}}} + F_r\frac{\dot{\xi}_1}{\sqrt{\dot{\xi}_1^{\,2}+\dot{\xi}_2^{\,2}}}\operatorname{sgn}\dot{\varphi}, \end{cases} \tag{2}$$

where

$$F_{fd} = F_f + F_d \tag{3}$$

is the sum of snow resistance (friction) F_f and air resistance (drag) F_d, which are expressed in the following way[10]:

$$F_f = \mu mg\cos\alpha, \tag{4}$$

$$F_d = k_1 v^2 = k_1\left(\dot{\xi}_1^{\,2} + \dot{\xi}_2^{\,2}\right) = mk\left(\dot{\xi}_1^{\,2} + \dot{\xi}_2^{\,2}\right), \tag{5}$$

[10] See Kaps *et al.* (1996).

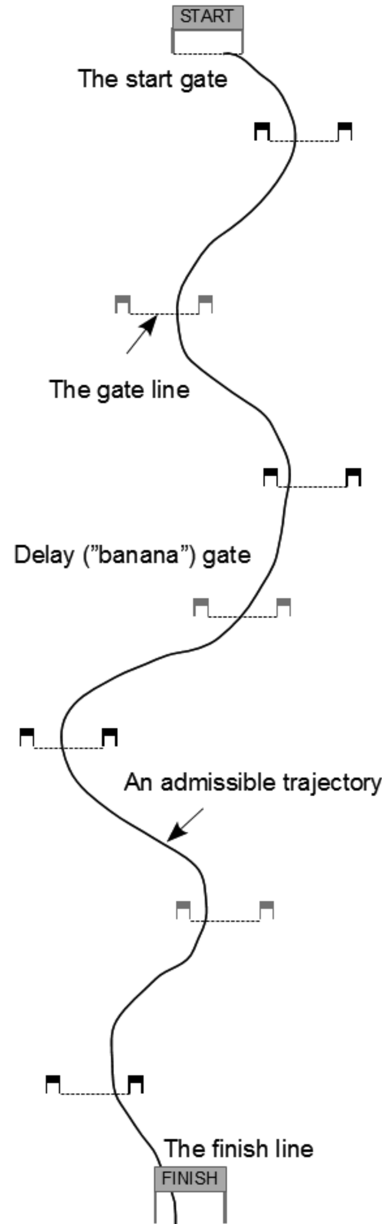

Fig. 2. Alpine ski racer's trajectory optimization (analyzed in this paper) as an example of a constrained, continuous trajectory optimization problem having a black-box (i.e., arbitrary complex) represented objective functional. (Note that admissible trajectories are those in which the skier passes correctly all the gate lines.)

and

$$F_r = m\left(\dot{\xi}_1^{\,2} + \dot{\xi}_2^{\,2}\right)|\kappa| + mg\sin\alpha\frac{\dot{\xi}_2}{\sqrt{\dot{\xi}_1^{\,2} + \dot{\xi}_2^{\,2}}}\operatorname{sgn}\dot{\varphi} \tag{6}$$

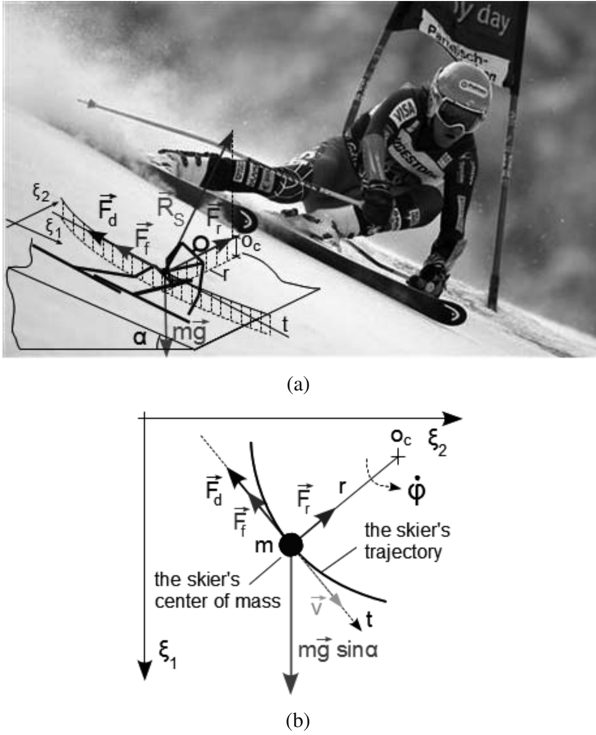

Fig. 3. Forces acting on a skier that is going around a purely carved turn (the forces are reduced to the skier's center of mass and the surface of the ski slope): Ted Ligety (one of the best contemporary alpine ski racers) passes a gate during the first run of an alpine ski men's World Cup Giant Slalom in Garmisch-Partenkirchen, Germany, Feb. 24, 2013 (photo by Shinichiro Tanaka; forces added by the author) (a), model (forces reduced to the surface of the ski slope, i.e., $\xi_1 - \xi_2$) (b).

is the $\xi_1 - \xi_2$ plane component of the perpendicular force exerted on the skis by the snow (see Fig. 3(b)). After dividing both sides by m, we get

$$\begin{cases} \ddot{\xi_1} = g \sin\alpha - F_{fd1} \frac{\dot{\xi_1}}{\sqrt{\dot{\xi_1}^2 + \dot{\xi_2}^2}} \\ \qquad - F_{r1} \frac{\dot{\xi_2}}{\sqrt{\dot{\xi_1}^2 + \dot{\xi_2}^2}} \operatorname{sgn} \dot{\varphi}, \\ \ddot{\xi_2} = -F_{fd1} \frac{\dot{\xi_2}}{\sqrt{\dot{\xi_1}^2 + \dot{\xi_2}^2}} + F_{r1} \frac{\dot{\xi_1}}{\sqrt{\dot{\xi_1}^2 + \dot{\xi_2}^2}} \operatorname{sgn} \dot{\varphi}, \end{cases} \tag{7}$$

where

$$F_{fd1} = \frac{F_{fd}}{m}, \qquad F_{f1} = \frac{F_f}{m}, \qquad F_{d1} = \frac{F_d}{m}, \qquad F_{r1} = \frac{F_r}{m}. \tag{8}$$

If we introduce

$$\boldsymbol{x} = (x_1, x_2, x_3, x_4)^\top = \left(\xi_1, \xi_2, \dot{\xi_1}, \dot{\xi_2}\right)^\top \tag{9}$$

as the vector of the system state variables (or simply the states) at time t, and

$$\boldsymbol{u} = (u_1) = (|\kappa|) \tag{10}$$

as the vector of the system control inputs at time t, then the system may be described by the following four first-order differential equations:[11]

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{a}\big(\boldsymbol{x}(t),\, \boldsymbol{u}(t)\big), \tag{11}$$

where[12]

$$\boldsymbol{a} = \begin{pmatrix} x_3 \\ x_4 \\ g \sin\alpha - x_3 f_{fd}(x_3, x_4) - x_4 f_r(x_3, x_4, u_1) \\ -x_4 f_{fd}(x_3, x_4) + x_3 f_r(x_3, x_4, u_1) \end{pmatrix} \tag{12}$$

and

$$f_{fd} = \frac{\mu g \cos\alpha + k\left(x_3^2 + x_4^2\right)}{\sqrt{x_3^2 + x_4^2}}, \tag{13}$$

$$f_r = \sqrt{x_3^2 + x_4^2}\, u_1 + g \sin\alpha \frac{x_4}{x_3^2 + x_4^2} \operatorname{sgn} \dot{\varphi}. \tag{14}$$

The above set of equations describes a nonlinear and time-invariant system.

3.1.2. Boundary conditions. Let t_0 be the time of leaving the start gate and t_f be the time of passing the finish line. The skier starts at the start gate at point $S(x_{1S}, x_{2S}),\ S \in l_{SG}$, so, clearly, when using the new state variables

$$x_1(t_0) = x_{1S}, \quad x_2(t_0) = x_{2S}. \tag{15}$$

We assume that the initial speed is zero, so

$$x_3(t_0) = v_{01S} = 0, \quad x_4(t_0) = v_{02S} = 0. \tag{16}$$

The only boundary condition at the finish point $F(x_{1F}, x_{2F}),\ S \in l_{FG}$, is related to the skier's position,

$$x_1(t_f) = x_{1F}, \quad x_2(t_f) = x_{2F}. \tag{17}$$

Note that l_{SG} and l_{FG} are the gate lines—the start and finish ones, respectively.

3.1.3. Performance measure. The aim of the optimization task is to find a trajectory having the minimum value of t_f (i.e., the time of passing the finish line), so this is an example of a minimum-time problem. For such problems,

$$h\big(\boldsymbol{x}(t_f), t_f\big) = 0, \quad g\big(\boldsymbol{x}(t),\, \boldsymbol{u}(t),\, t\big) = 1, \tag{18}$$

[11] In a more general case, when the problem is time-varying, the expression (11) takes the form $\dot{\boldsymbol{x}}(t) = \boldsymbol{a}\big(\boldsymbol{x}(t),\, \boldsymbol{u}(t),\, t\big)$.

[12] See Eqns. (3)–(5).

so, if we additionally assume that $t_0 = 0$, the performance measure simplifies to

$$J = \int_0^{t_f} \mathrm{d}t = t_f - 0 = t_f. \tag{19}$$

We can see from Eqn. (19) that simulation is the only way of evaluating the performance measure.

3.2. One-dimensional approximation model. The skier's trajectory can be approximated by a piecewise-linear function[13]. This modification simplifies the problem significantly—instead of a one (complex) two-dimensional problem, we have a series of (simple) one-dimensional ones. Each of the sub-problems is related to one segment only (see Fig. 4, note a local coordinate system $\zeta\eta$, set for each segment).

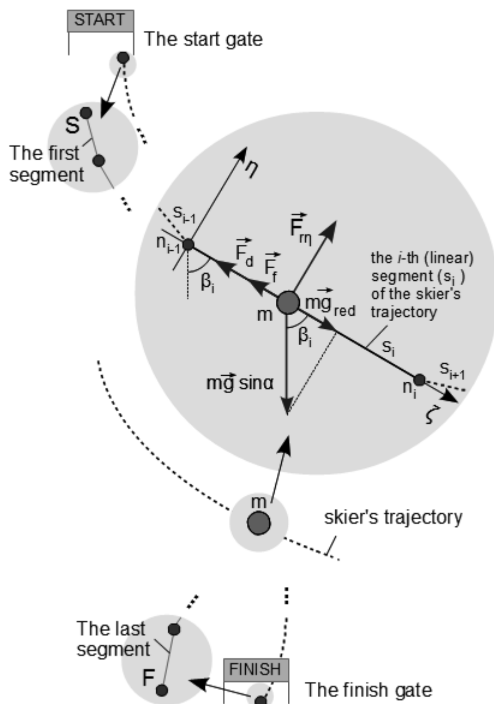

Fig. 4. One-dimensional approximation model.

3.2.1. Equations of motion. Equations (2) simplify to the following form:

$$\begin{cases} m\ddot{\zeta} = mg_{red} - (F_f + F_d), \\ 0 = -F_{r\eta} - mg\sin\alpha\sin\beta, \end{cases} \tag{20}$$

where

$$g_{red} = g\sin\alpha\cos\beta \tag{21}$$

can be considered "*reduced gravitational acceleration*"[14] and F_f and F_d are defined by Eqns. (4) and (5), respectively. Only the first equation is important in the simulation. The second one expresses the condition of equilibrium in the normal (to the trajectory) direction.

After dividing both sides of the first of Eqns. (20) by m and simplifying the result, for the i-th segment we get

$$\ddot{\zeta}_i + k\dot{\zeta}_i^{\,2} = g\cos\alpha\left(\tan\alpha\;\cos\beta_i - \mu\right). \tag{22}$$

If, similarly as before, we introduce

$$\boldsymbol{x}^{(i)} = \left(x_1^{(i)},\, x_2^{(i)}\right)^\top = \left(\zeta_i,\, \dot{\zeta}_i\right)^\top \tag{23}$$

as the vector of the system state variables at time t, and

$$\boldsymbol{u}^{(i)} = \left(u_1^{(i)}\right) = (\beta_i) \tag{24}$$

as the vector of the system control inputs at time t, then the system can be described by the following two first-order differential equations:

$$\dot{\boldsymbol{x}}^{(i)}(t) = \boldsymbol{a}^{(i)}\big(\boldsymbol{x}^{(i)}(t),\, \boldsymbol{u}^{(i)}(t)\big), \tag{25}$$

where[15]

$$\boldsymbol{a}^{(i)} = \begin{pmatrix} x_2^{(i)} \\ g\cos\alpha\left(\tan\alpha\cos u_1^{(i)} - \mu\right) - k(x_2^{(i)})^2 \end{pmatrix}. \tag{26}$$

3.2.2. Boundary conditions. The skier's trajectory is now approximated by a piecewise-linear function. We assume that its first (linear) segment starts at point $S(\xi_{1S}, \xi_{2S})$ and the last one ends at $F(\xi_{1F}, \xi_{2F})$ (see Fig. 4). The boundary conditions have to be written now for each segment. An additional assumption, related to the speed values in two subsequent segments, has to be introduced into the model

$$|\boldsymbol{v}_s^{(i)}| = c|\boldsymbol{v}_f^{(i-1)}|, \tag{27}$$

where $\boldsymbol{v}_s^{(i)} = \boldsymbol{x}_{2s}^{(i)}$ is the initial speed for the (i)-th segment, and $\boldsymbol{v}_f^{(i-1)} = \boldsymbol{x}_{2e}^{(i-1)}$ is the final speed for the $(i-1)$-th segment.

3.2.3. Performance measure. The trajectory segmentation changes Eqn. (19) in the following way:

$$J = t_f = \sum_s t_f^{(s)}, \tag{28}$$

i.e., in order to find the total time of motion, a series of simulations (one for each segment—s) have to be performed.

[13] The more segments, the lower the approximation error.

[14] To the slope plane ($g\sin\alpha$) and to the current linear segment direction ($g\sin\alpha\cos\beta$).

[15] Compare Eqns. (3), (4) and (5)

4. Solution methods

The key idea behind the algorithms presented in this section is to use a grid-based discretization scheme to transform the continuous optimization problem into a search problem over a specially constructed finite graph, and then to apply dynamic programming to find an approximation of the global solution which, in the analyzed example, is the minimum-time ski line represented as a piecewise-linear function (see Fig. 5). Of course, the finer this grid, the better the final result (i.e., closer to the exact solution). The grid (mesh) is based on equidistant nodes (Fig. 5 presents eight nodes in a row, but there can be any number of them). The graph representing this grid is directed, acyclic and topologically sorted (the nodes in row r are followed by the nodes in row $r+1$ as are the edges of the graph which correspond to the linear segments of the trajectory). These features allow the search process to be much more effective when compared with the standard Dijkstra (shortest path) algorithm. The cost of each edge in the graph is obtained from simulation, taking into account the corresponding boundary conditions (related to the values of linear velocity). It is important to note that at the beginning of the optimization process there is no cost matrix. The cost of each edge (linear segment) can be calculated only when the corresponding initial velocity is known, and this value depends on the results received from the simulation for preceding segments. This is a sequential component of the optimization process.

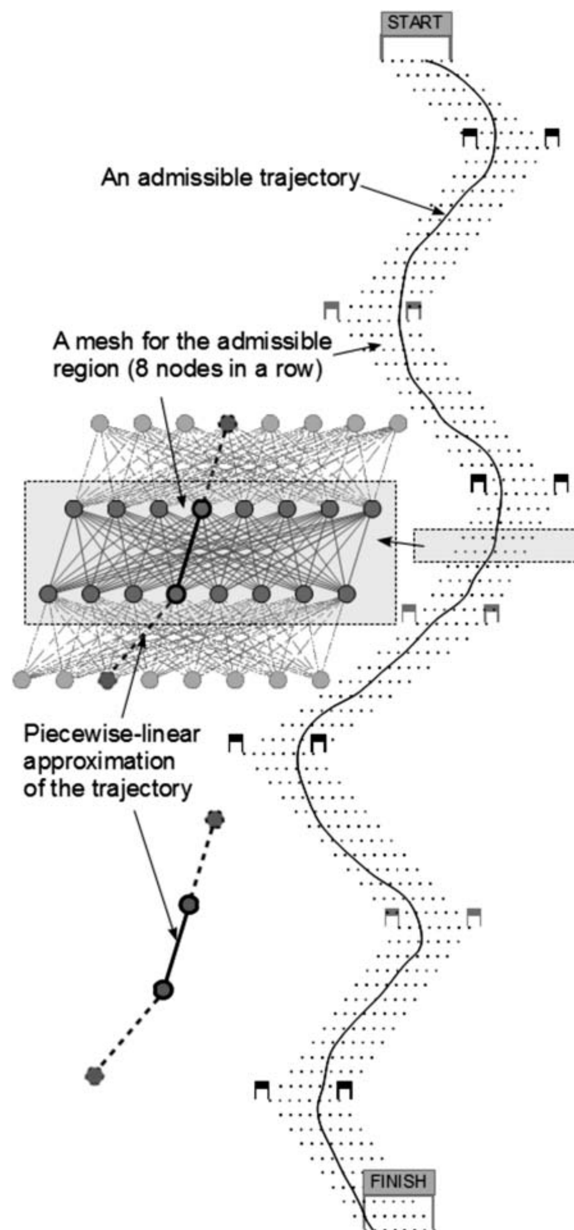

Fig. 5. Grid-based discretization scheme transforming the continuous optimization problem into a search problem. The grid nodes are grouped in rows and columns (the grid shown in the figure has eight nodes in a row—it is an eight-column structure).

4.1. Elimination of unfeasible solutions.

The skier's model assumes that all turns are purely carved (or at least close). This means that the angle between any two segments (see Fig. 6) in the final trajectory should be as small as possible (this constraint is related to the curvature of the trajectory). To eliminate the solution candidates which do not meet the assumption and, at the same time, keep the computational process stable, we introduce a penalty function defined in the following way:

$$v_0^{(s)} = \begin{cases} v_1^{(s-1)} & \text{if } \cos\delta >= U_m, \\ v_1^{(s-1)} \cos^4\delta & \text{if } L_m < \cos\delta < U_m, \\ 0 & \text{otherwise.} \end{cases} \quad (29)$$

In Eqn. (29) U_m represents the upper margin (assumed in the experiments to be 0.98) and L_m stands for the lower margin (assumed in the experiments to be 0.5).

Note that Eqn. (29) is just an example of a penalty function[16] though its effectiveness has been verified in experiments.

[16]It reduces the skier's speed in the case of overly tight turns (so with skidding, which always causes braking).

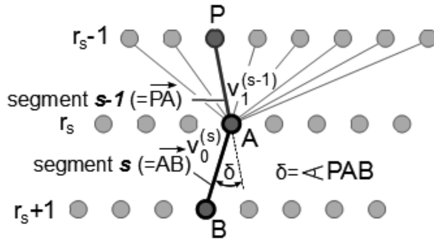

Fig. 6. Angle δ representing a local value of the trajectory curvature.

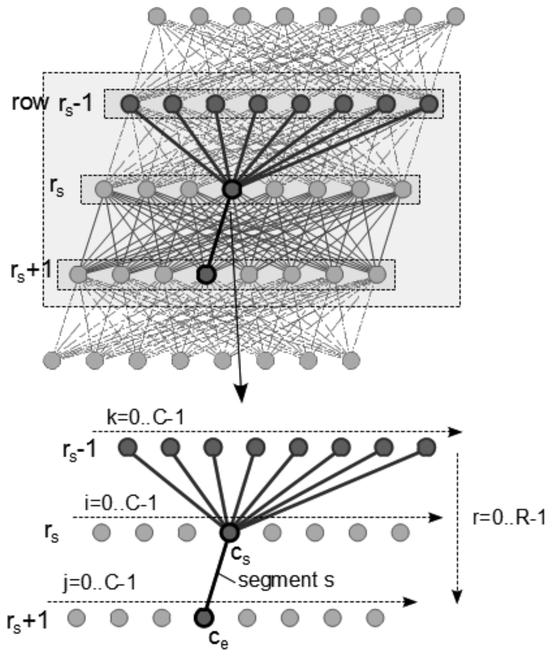

Fig. 7. Dependence of the segments in the mesh: segment $s(r_s, c_s, c_e)$ depends on all segments $s^{(k)}(r_s - 1, k, c_s)$. Arrows with ranges $r = 0, \dots, R-1; i = 0, \dots, C-1; j = 0, \dots, C-1; k = 0, \dots, C-1$ describe the loops of the algorithm pseudo-codes presented in this section.

4.2. Basic optimization algorithm. In the analysis that has been carried out so far it is assumed that nodes from any two subsequent rows (layers) are fully connected (i.e., each node in row r is connected to all nodes from rows $r-1$ and $r+1$). This means that, in order to evaluate a single linear segment we have to perform C simulations (see Fig. 7). This is a consequence of the definition of the penalty function (see Eqn. (29) and Fig. 6). The number of simulations can be usually reduced if we make use of some features of the model. For instance, in the case of the skier model, the set of segments which end in a node, say node c, $\{s_c^{(i)} : i = 0, \dots, C-1\}$, is partially ordered

$$s_c^{(l)} \preceq s_c^{(m)} \iff t_{f(c)}^{(l)} \le t_{f(c)}^{(m)} \wedge v_{f(c)}^{(l)} \ge v_{f(c)}^{(m)}. \quad (30)$$

So if $s_c^{(l)} \preceq s_c^{(m)}$, there is no need to perform simulation for $s_c^{(m)}$. Note that the basic version of the algorithm assumes performing simulation for all segments, which is more general and keeps parallel computation more regular.

The above analysis is not related to the simulation for the segments from the first row, because they do not have any predecessors. The pseudo-code of simulation for these segments is shown as Algorithm 1. In the first step

---

Algorithm 1. sim_0: Perform simulation for a segment in mesh row 0.

Require: c_s, c_e
{@param c_s: the segment start node column index}
{@param c_e: the segment end node column index}
1: $(l^{(s)}, g_{red}^{(s)}) := get_segm_stat_data(0, c_s, c_e)$
2: $(t_1^{(s)}, v_1^{(s)}) := simulation_for(l^{(s)}, g_{red}^{(s)}, 0, 0)$
{set $t_1^{(s)}$ and $v_1^{(s)}$ as s's optimal values}
3: $update_segment(0, c_s, c_e, t_1^{(s)}, v_1^{(s)}, NULL)$

---

we calculate the segment length $l^{(s)}$ and the corresponding reduced gravitational acceleration $g_{\mathrm{red}}^{(s)}$ (see Eqn. (21)).

Note that this pair of values is referenced in this section as *static data* of the segment. Next, the final values for time $t_1^{(s)}$ and speed $v_1^{(s)}$ are obtained through simulation. In the last step these values are stored because they will be needed in simulations for segments from the first layer (see Eqn. (29) and Fig. 6).

Algorithm 2 shows the main steps of the simulation for the segments from rows $1, \dots, R-2$. The differences (when compared with Algorithm 1) are a consequence of using the penalty function (see Eqn. (29)). To find the shortest time of travel to the segment end point (e.g., point (r_s+1, c_e) in Fig. 7), it is necessary to perform C simulations updating, if necessary, the best solution found so far (compare Dijkstra, 1959). The variable $pidx_{\mathrm{opt}}$ stores the index of the segment from row $r_s - 1$ which corresponds to the locally optimal piece of trajectory (with the shortest time of travel to the segment end point; indexes $pidx_{\mathrm{opt}}$ are used in the final part of the algorithm to find the optimal solution).

Note that sim_0 and sim_{1R} presented as Algorithms 1 and 2 are *OpenCL kernels*.

4.2.1. Serial (reference) version. Algorithm 3 presents the serial version of the trajectory optimization procedure (compare loop control variables with Fig. 7). The time complexity of the algorithm is equal to $\Theta\left(RC^3 t_{\mathrm{sim}}\right)$, where R and C are the numbers of rows[17] and columns (in the mesh), respectively, and t_{sim} is

[17]The dependence on the number of rows is caused by the constant component present in the simulation time, needed for solving the non-linear equation $t_f^{(s)} = f(l^{(s)}, g_{\mathrm{red}}^{(s)})$.

Algorithm 2. sim_{1R}: Perform simulation for a segment in mesh rows $[1, \ldots, R-1)$.

Require: r_s, c_s, c_e
{@param r_s: the segment start node row index}
{@param c_s: the segment start node column index}
{@param c_e: the segment end node column index}
1: $t_{\text{opt}} := T_{\text{MAX}}$
2: $(l^{(s)}, g_{\text{red}}^{(s)}) :=$ $get_segm_stat_data(r_s, c_s, c_e)$
{For all mesh nodes in the $(r_s - 1)$-th row}
3: **for all** k in [0..C) **do**
{Initialize $t_0^{(s)}, v_0^{(s)}$ with the corresponding final values for segment $(r_s - 1, k, c_s)$}
4: $(t_0^{(s)}, v_{0i}^{(s)}) :=$ $get_segm_dyn_data(r_s - 1, k, c_s)$
{If necessary, correct v_0, see Eqn.29}
5: $v_0^{(s)} = correct_v_0(r_s - 1, k, c_s, c_e, v_{0i}^{(s)})$
6: $(t_1^{(s)}, v_1^{(s)}) :=$ $simulation_for(l^{(s)}, g_{red}^{(s)}, t_0^{(s)}, v_0^{(s)})$
{If necessary, update the best values so far}
7: **if** $(t_0^{(s)} + t_1^{(s)} < t_{\text{opt}})$ **then**
8: $t_{\text{opt}} := t_0^{(s)} + t_1^{(s)}$
9: $v_{\text{opt}} := v_1^{(s)}$
10: $pidx_{\text{opt}} := k$
11: **end if**
12: **end for**
{Update the segment data}
13: $update_segment(r_s, c_s, c_e, t_{\text{opt}}, v_{\text{opt}}, pidx_{\text{opt}})$

the time needed for a single simulation. The algorithm needs $\Theta(RC)$ memory (i.e., the memory consumption is proportional to the size of the mesh).

Note that the time complexity of this serial algorithm can be reduced, for instance, by modifications described in section 4.3.

4.2.2. Parallel version. Algorithm 3 can be parallelized by making use of the fact that simulations for all segments in the same row are independent[18] of each other. This idea is shown in Fig. 8. The simulation tasks can be performed in OpenCL-capable embedded systems (they can be considered μ-HPC systems), in cluster-based HPC infrastructures or in hybrid, hierarchical systems.

Algorithm 4 presents pseudo-code of the parallel version of the trajectory optimization procedure. Functions sim_0_kernel and sim_{1R}_kernel are OpenCL kernels, i.e., pieces of code prepared for parallel execution on OpenCL-capable devices. They are almost the same as sim_0 and sim_{1R}, with two OpenCL “decorators” (*kernel* and *global*) added into the code.

[18]The problem is embarrassingly parallel.

Algorithm 3. Serial trajectory optimization.

{For all mesh nodes in the 0-th row}
1: **for all** i in [0..C) **do**
{For all mesh nodes in the 1-st row}
2: **for all** j in [0..C) **do**
3: $sim_0(i, j)$
4: **end for**
5: **end for**
{For the rest rows in the mesh}
6: **for all** r in [1..R-1) **do**
{For all mesh nodes in the r-th row}
7: **for all** i in [0..C) **do**
{For all mesh nodes in the $(r+1)$-th row}
8: **for all** j in [0..C) **do**
9: $sim_{1R}(r, i, j)$
10: **end for**
11: **end for**
12: **end for**
13: **return** $opt_trajectory, fin_time, fin_velocity$

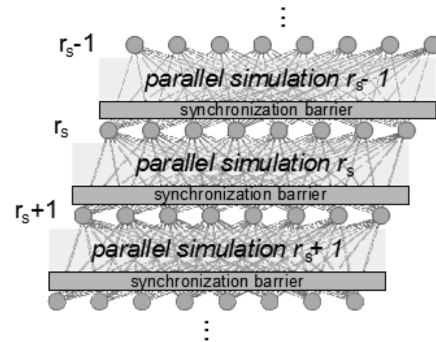

Fig. 8. Parallel simulation for all segments in the same mesh row. The synchronization barrier is needed because of the dependence between nodes from subsequent rows.

The parallel algorithm time complexity is equal to $\Theta(RCt_{\text{sim}})$. The annotation $@PARALLEL(f(\cdot))$ expresses[19] that function f CAN BE executed in parallel for different pieces of data referenced by the pointer—$data_p$.

4.3. Possible extensions of the basic algorithm. In some situations the algorithms described above have to be changed to meet certain constraints. It might be the case, for instance, when the algorithm time and/or memory complexity is too high. The memory constraint can be a problem in some embedded systems or when the available (OpenCL-capable) GPU has too little RAM. One way of addressing this issue is to divide the computation into stages which represent subsequent parts of the mesh. Each

[19]In pseudo-code only as there is no such annotation in OpenCL.

Algorithm 4. Parallel trajectory optimization.

1: @PARALLEL $(sim_0_kernel(data_p))$
2: **for all** r in [1..R-1] **do**
3: @PARALLEL $(sim_{1R}_kernel(r, data_p))$
4: **end for**
5: **return** $opt_trajectory, fin_time, fin_velocity$

stage can be handled separately, with partial results stored on hard drive. Unfortunately, this approach often is not acceptable. Sometimes there is just no hard drive (or any device of this type) in the system. Other times, the increased number of I/O operations adversely affects the computation time.

Another approach, i.e., successive mesh refinements, is shown in Fig. 9. The number of mesh nodes can be set so that all its data structures can be stored in RAM. The optimization task is divided into a couple of iterations in which subsequent meshes (of the same size or not) are generated and used to find better and better approximations of the exact solution. This

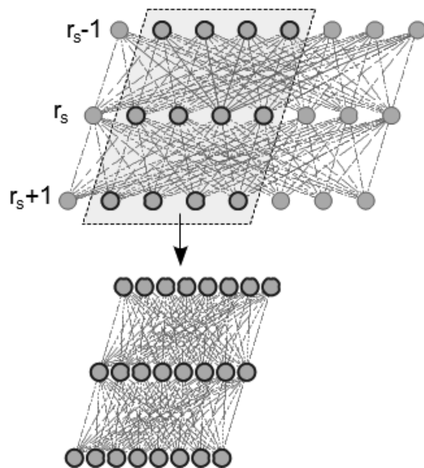

Fig. 9. Mesh refinement: it reduces the algorithm memory complexity and improves the accuracy of the final result.

approach reduces the memory size needed to perform the optimization process. It can sometimes also reduce the computation time, if the total number of performed simulations is smaller than in the basic algorithm.

Another modification, shown in Fig. 10, assumes reduction in the number of connections between nodes. The smaller number of connections decreases both the memory size needed to store the mesh data structures and the total number of simulations to be performed. As a consequence, both the time and memory complexity can be (significantly) reduced. Unfortunately, the effectivenesses of this approach is very dependent on the problem being solved. In many cases we cannot reduce the search space in this way without the danger of losing

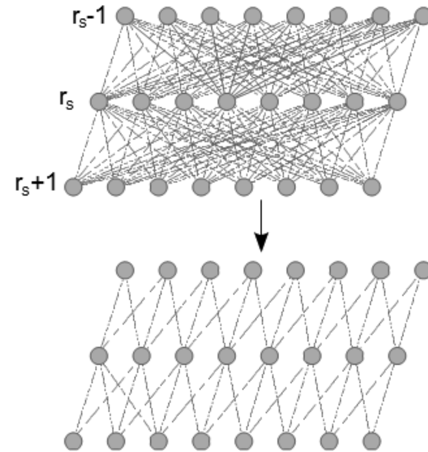

Fig. 10. Reduction of the number of connections between nodes (i.e., the degree of the nodes).

good solution candidates.

The last possibility (discussed in this section) of extending the basic algorithm is based on combining the global search with a local one. In the skier's trajectory optimization case the local algorithm can be executed for each turn separately. The local optimization can improve the final result—sometimes significantly. This is because the search space of the local algorithm can be continuous, which means that there is no accuracy limit related to the mesh granularity. One of the key aspects of successful application of local optimization is choosing, in the search space, a basis appropriate for the problem.

5. Experimental results

To demonstrate the effectiveness of the algorithms presented in Section 4, two series of experiments[20] were carried out to find the optimal trajectory (i.e., the ski line) for the unsymmetrical course setup shown in Fig. 2. The ski slope α was constant in all experiments and set to $\pi/12$. The first series was related to the skier's model with $\mu = 0.00, k = 0.00$ (i.e., with no friction and no drag force, see Eqns. (4), (5) and (25)), whilst the second one with $\mu = 0.12, k = 0.05$ (these values represent the upper limits of the coefficients).

In all experiments a MacBook Pro[21] with OS X 10.8.4 and OpenCL 1.1, having two OpenCL-capable devices:

- processor **Intel Core i7-3740QM** @ 2.7 GHz,
- graphic card **nVidia GeForce GT 650m**,[22]

[20]Each experiment was carried out 5 times, the values were then averaged.
[21]With 16 GB of DDR3L 1600 MHz RAM.
[22] Two compute units, each having 192 processing elements (CUDA

was used. This computer system can be visualized from the OpenCL perspective as shown in Fig. 11 (cf. Gaster *et al.*, 2013), if we assume that there are two OpenCL devices as specified above.

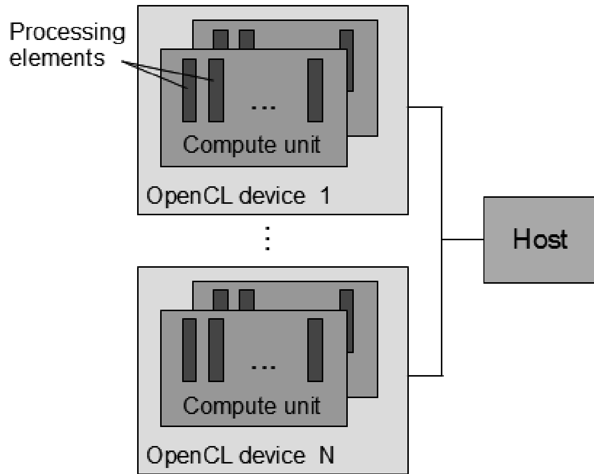

Fig. 11. OpenCL platform model with one host and N OpenCL devices. Each OpenCL device has one or more compute units, each of which has one or more processing elements.

5.1. Performance analysis: Impact of a global number of work-items.

The results for $\mu = 0.00$ and $k = 0.00$ are summarized by Table 1 and Fig. 12. In Table 1 the subsequent columns represent the following: m—the number of mesh nodes in one row, t_s—the execution time of the serial version of the optimization algorithm (see Algorithm 3), $t_{\text{ocl}-\text{CPU}}$ and $t_{\text{ocl}-\text{GPU}}$ are the execution times of (parallel) Algorithm 4 run on the CPU and the GPU, respectively, and finally, for reference, t_f—the final result of the optimization. It can be seen from the last column that the minimum (acceptable) number of mesh nodes (in one row) is 32. The optimal mesh should have between 32 and 64 nodes in one row, depending on the required accuracy.

The speedups[23] are presented in Fig. 12. Two facts are worth noting with regard to speedups: on the GPU there is a strong dependence on the mesh-size whereas on the CPU this dependence is almost negligible. The first of these is a typical feature of computations performed on a GPU. The more work-items (which can be seen as threads) the GPU has to handle, the better the speedup. Another important aspect of utilizing a GPU as a general-purpose computation device is the cost of data transfers between the host and the GPU. This overhead can be sometimes so high that using the GPU has no benefit. The figure also shows that there are two characteristic ranges of mesh node numbers—in the first one, with $m \leq 16$, the CPU was more effective, whilst in the second one, for bigger values of m, the GPU performed better (for $m = 128$ the speedup is equal to 128). This observation demonstrates an important advantage of using OpenCL for programming heterogeneous computer systems—the resources can be used more effectively with no additional programming cost (the code is exactly the same for all OpenCL-capable devices). Moreover, the selection of the computing device can be done at run-time.

Table 1. Average execution times (in ms), from five runs, for different numbers of mesh nodes in a row (the model with $\mu = 0.00, k = 0.00$). The last column (t_f) contains the final results. Note that the code of the serial (reference) algorithm was compiled with -O0 compiler flag.

m	t_s	$t_{\text{ocl}-\text{CPU}}$	$t_{\text{ocl}-\text{GPU}}$	t_f[s]
8	533	104	368	26.50
16	3896	536	698	17.78
32	28329	3574	1296	11.60
64	220923	25517	2985	11.37
128	1739789	213490	13498	11.33

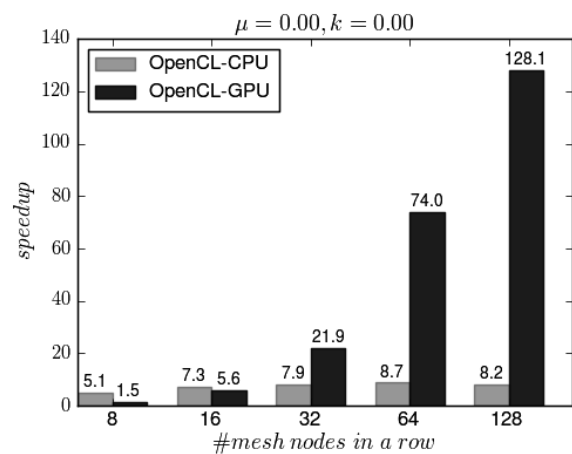

Fig. 12. Speedups for the skier's model with no friction and no drag (i.e., $\mu = 0.00, k = 0.00$). The number of mesh nodes is related to the total number of OpenCL work-items (threads) in the following way: $\#\textit{work-items} = m^2$.

The results for $\mu = 0.12$ and $k = 0.05$ are summarized by Table 2 and Fig. 13. Longer simulation times were the result of a very slow motion of the skier (which was caused by high forces of friction and drag). These parameters of the skier's model were selected to demonstrate worse (than before) results of utilizing the GPU as a general-purpose computing device. Figure 13

cores), warp size 32, 1 GB of GDDR5 memory, 48 KB of local memory, 64 KB of constant memory.

[23]Calculated in a classic way, i.e., as $s = t_s/t_p$.

Table 2. Average execution times (in ms), from five runs, for different numbers of mesh nodes in a row (the model with $\mu = 0.12, k = 0.05$). The last column (t_f) contains the final results. Note that the code of the serial (reference) algorithm was compiled with -O0 compiler flag.

m	t_s	$t_{\text{ocl-CPU}}$	$t_{\text{ocl-GPU}}$	t_f[s]
8	708	132	1455	49.60
16	4985	636	1911	40.64
32	42998	5115	6250	35.87
64	345426	38725	19251	35.33
96	1152450	104740	34918	34.87

presents the corresponding values of the speedup. The speedups related to the CPU were again similar in the whole range of the change of parameter m. It may be surprising that the best speedup observed on the CPU was equal to 11.0 because it was achieved on the device having only four cores, each one with two logical threads (which means eight threads executing simultaneously). This was the result of OpenCL compiler optimizations. Speedups

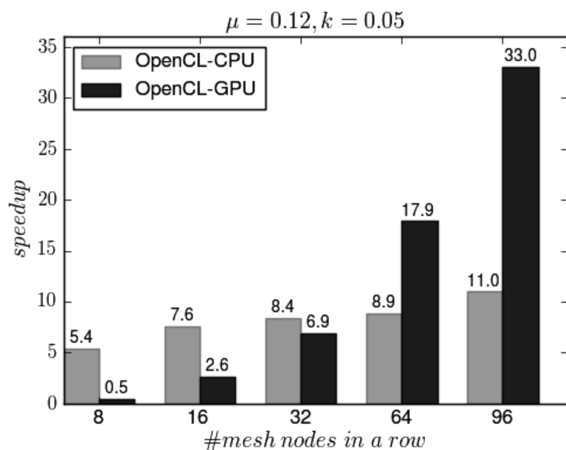

Fig. 13. Speedups for the skier's model with $\mu = 0.12, k = 0.05$. The number of mesh nodes is related to the total number of OpenCL work-items (threads) in the following way: $\#work\text{-}items = m^2$.

related to the GPU were significantly worse than before (i.e., for $\mu = 0.00$ and $k = 0.00$). Some points are worth noting here: for eight mesh-nodes in a row the GPU-accelerated computation took twice as much time as the serial algorithm, for 32 nodes the speedup was only 6.9 and was worse than the corresponding one for the CPU, and finally, the biggest speedup was equal to 33.0. This worse performance was a consequence of a very unbalanced (among work-items) computational load[24]. Because of the way the computation was synchronized (see Fig. 8), the computation for a single mesh row was as long as the longest simulation of a single segment.

5.2. Performance analysis: Impact of a local number of work-items. According to the OpenCL execution model (see Gaster *et al.*, 2013), a CPU host defines an N-dimensional computation domain over some region of an OpenCL device's DRAM memory. Every index of this N-dimensional domain at runtime corresponds to a work-item (which executes the same OpenCL kernel). These work-items are grouped (by the host) into work-groups. All work-items from the same work-group execute concurrently on the same compute unit. They can share local memory (this is a special memory space with a work-group scope), and can be synchronized using barriers. Each work-item has two indexes—a global (in the computational domain) and a local one (within the work-group it belongs to). The number of the work-items in one work-group is called work-group size.

To analyze the impact of a work-group size on the (computational) speedup, the same[25] experiments (as in the previous section) were carried out for different work-group sizes including a special "auto" mode (in which the OpenCL platform determines the best value of this parameter[26]). The results are summarized in Tables 3 and 4. These results are averages from five runs. The

Table 3. Average execution times (in ms), from five runs, on GPU for different numbers of mesh nodes (m) in a row and different number of work items (w_i) in one work group (the model with $\mu = 0.00, k = 0.00$).

	$m = 8$	$m = 16$	$m = 32$	$m = 64$
$w_i = 4$	389	917	3994	27063
$w_i = 16$	407	717	1614	7910
$w_i = 64$	407	758	1319	3132
$w_i = 256$	–	758	1359	3017
$w_i = 1024$	–	–	1394	4016
$w_i = auto$	404	753	1371	3043

first of the tables is related to the skier's model with $\mu = 0.00$ and $k = 0.00$, and the second one to the model with $\mu = 0.12$ and $k = 0.05$. The empty spaces are a consequence of the OpenCL platform constraints—for certain parameters the experiment could not be run. These constraints are related (among others) to the maximum number of work-items in a work-group (in the case of GT 650m it is equal to 1024), the size of local memory (48 kB) and the size of constant memory (64 kB). There is

[24]This issue can be addressed by equalizing (approximately) the simulation times for all segments in a row.

[25]But only on the GPU; a programmer cannot control the work-group size on the CPU for the current OpenCL platform.

[26]A programmer can chose this option by setting the work-group size to 0.

Table 4. Average execution times (in ms), from five runs, on GPU for different numbers of mesh nodes (m) in a row and different number of work items (w_i) in one work group (the model with $\mu = 0.12, k = 0.05$).

	$m = 8$	$m = 16$	$m = 32$	$m = 64$
$w_i = 4$	1332	1996	10318	-
$w_i = 16$	1424	1963	7026	28921
$w_i = 64$	1450	1993	6739	19739
$w_i = 256$	–	1910	6248	19223
$w_i = 1024$	–	–	6394	25936
$w_i = auto$	1440	1918	6237	19260

also an important platform parameter describing the *recommended* work-group sizes. They should be equal to even multipliers of a constant defined by NVIDIA as *a warp* and by AMD *a wave-front*. In the case of GT 650m it is equal to 32.

The values from Tables 3 and 4 are presented in Figs. 14–16 from the computational speedup perspective. Figure 14 shows that, for a small GPU computational load[27], the work-group size change impact on the speedup was negligible for both sets of the skier's model parameters.

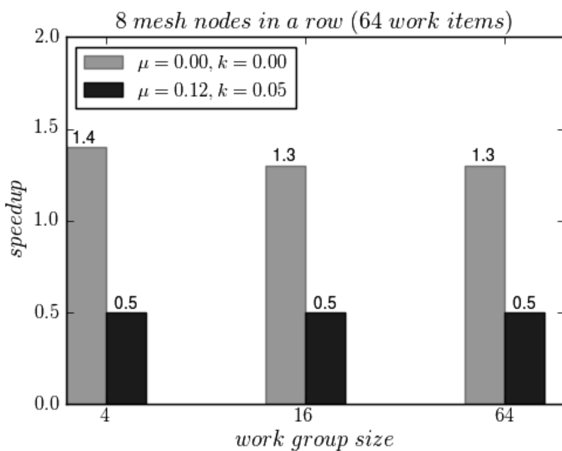

Fig. 14. Comparison of speedups in eight mesh nodes in a row based trajectory optimization for two sets of a skier's model parameters and different work-group sizes.

The same could be observed in the case of 16 mesh nodes in a row (which means $16 \times 16 = 256$ work-items), but only for the second model (for which the observable speedups were worse in general, compare Figs. 12 and 13). For the first one, there was a small (25%) speedup increase with the work-group size change from $2 \times 2 = 4$ to $4 \times 4 = 16$ (see Tables 1–4).

[27]That is, $8 \times 8 = 64$ work-items (globally); they are grouped in $8/2 \times 8/2 = 16$, $8/4 \times 8/4 = 4$, $8/8 \times 8/8 = 1$ work-groups, respectively (as shown in Fig. 14).

The corresponding dependence for 32 mesh nodes in a row (meaning 1024 work-items) is shown in Fig. 15. This computational load was big enough to start utilizing the GPU's parallel processing capabilities. In this case the work-group size change impact on the speedup was significant, especially for the skier's model with $\mu = 0.00$ and $k = 0.00$. The figure shows that the work-group sizes

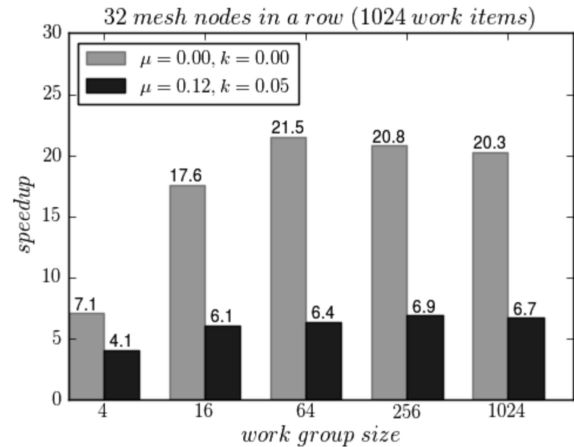

Fig. 15. Comparison of speedups in 32 mesh nodes in a row based trajectory optimization for two sets of a skier's model parameters and different work-group sizes.

should be bigger than $4 \times 4 = 16$, with the maximum speedup observed at work-group size equal to $8 \times 8 = 64$. Because the subsequent values were only slightly worse, we should consider an optimal range (e.g., $[8, 32]$), rather than the single value.

Finally, the dependence for 64 mesh nodes in a row ($64 \times 64 = 4096$ work-items) is shown in Fig.16. Similarly, the work-group size change impact on the speedup was significant. In this case the optimum was reached at the work-group size equal to $16 \times 16 = 256$, but again the value for $8 \times 8 = 64$ was only a bit worse.

After taking into account that the work-group size should be equal to even multipliers of a warp value (for the GT 650m it is equal to 32), one can draw the conclusion that, in the analyzed computational problem, they should be equal to 64 or 256. On the other hand, the results in Tables 3 and 4 show that the "auto mode" is usually good enough (at least as the first approach) for most cases. Note that the results presented in the previous section were obtained with the use of this mode, thus giving the OpenCL platform control over the optimal work-group size.

5.3. Final result analysis: Impact of unfeasible solution elimination.

The optimal trajectories presented in this paragraph were calculated in a mesh having 64

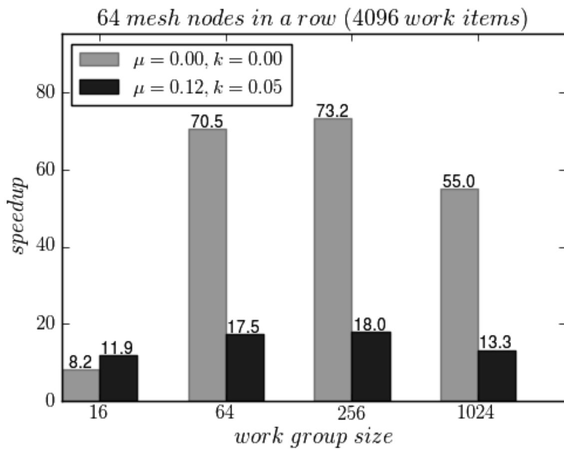

Fig. 16. Comparison of speedups in 64 mesh nodes in a row based trajectory optimization for two sets of a skier's model parameters and different work-group sizes.

nodes in a row (note that: these nodes are too close to each other to be seen as separate dots in Figs. 17 and 18) and with the skiers model having no friction and no drag force (i.e., with $\mu = 0.00, k = 0.00$). Parts (a) of both of these figures correspond to the results received from optimization without elimination of unfeasible solutions (see Fig. 6 and Eqn. 29). On the other hand, Parts (b) are related to the results obtained when the elimination of unfeasible solutions was activated.

Figure 17 shows the two optimal trajectories for a simple symmetrical course. The solution reflected in Fig. 17(a) is a sequence of seven brachistochrones (cycloides)—it is not a feasible solution because of these "zero-radius" turns at each gate[28]. The finish time for this solution was $t_{f_{\rm ne}} = 10.27$s[29]. Solution b is both feasible and optimal, with the corresponding $t_{\rm f_e} = 10.36$ s.

Figure 18 shows the two trajectories for an unsymmetrical course, which is much more challenging for the athletes. In this case, Part (a) is an unfeasible solution composed of eight brachistochrones with the corresponding $t_{f_{\rm ne}} = 11.21$ s. Part (b) of the figure shows the trajectory which is feasible and optimal. Its corresponding finish time $t_{\rm f_e} = 11.37$ s.

Note that the penalty function defined by Eqn. (29) proved effective not only in the examples presented above but also for problems in which both the skier's model parameters and/or course set-ups were different. However, it is just an example of such a function, which was needed to demonstrate the whole optimization algorithm.

[28]One of the key assumptions of the model was that all turns are purely carved, with no skidding (which causes braking).

[29]With "ne" standing for "no elimination".

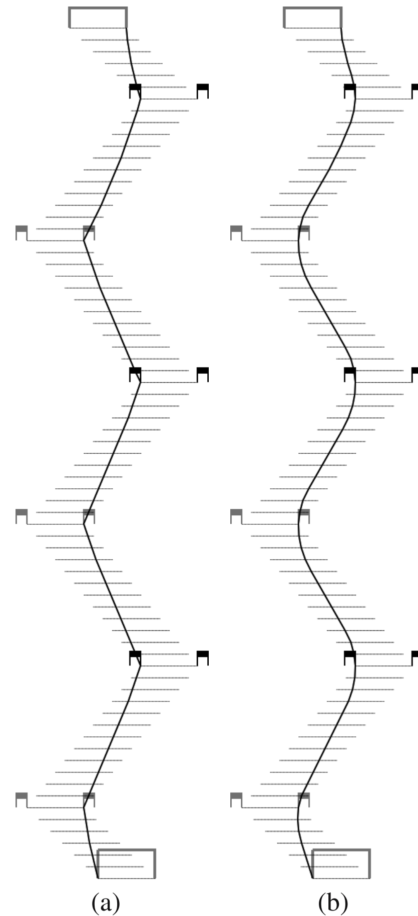

Fig. 17. Trajectory optimization result for a symmetrical course, the skier's model with $\mu = 0.00, k = 0.00$ and WITHOUT elimination of unfeasible solutions (a), WITH elimination of unfeasible solutions (b).

6. Conclusion

Effective, simulation-based trajectory optimization algorithms adapted to heterogeneous computer systems were studied using an example taken from alpine ski racing. Both the serial and parallel versions of the optimization algorithm were presented in detail (pseudo-code, time and memory complexity). Possible extensions of the basic algorithm were also described.

The experimental results showed that contemporary heterogeneous computers can be used effectively in simulation-based continuous trajectory optimization problems. Such computers can be treated as μ-HPC platforms—they offer high peak performance (the effective use of the OpenCL-capable GPU accelerated the optimization procedure even up to 128 times!) while remaining energy and cost efficient. This fact can be very important in the embedded systems domain.

The presented algorithms can be applied to many trajectory optimization problems, including those having a black-box represented performance measure and/or the

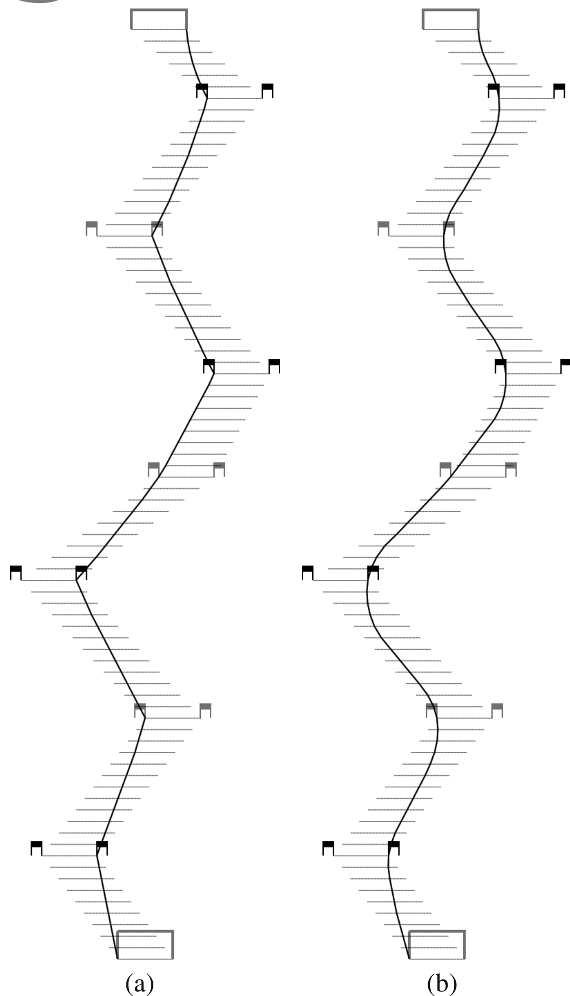

Fig. 18. Trajectory optimization result for a unsymmetrical course, the skier's model with $\mu = 0.00, k = 0.00$ and WITHOUT elimination of unfeasible solutions (a), WITH elimination of unfeasible solutions (b).

ones in which the trajectory cannot be approximated by a piecewise-linear function.

Future research work will concentrate on experimenting with the proposed extensions of the basic (parallel) algorithm, on different ways of approximating the trajectory, on the verification of the proposed algorithm in an *augmented cloud*[30] environment, and on further analysis of the impact of the model parameters on optimization results.

Acknowledgment

The research presented in the paper was partially supported by the European Commission FP7 through the project *ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems*, under the contract no. 288570.

[30]See, e.g., Dębski *et al.*, (2012; 2013), Byrski *et al.* (2012)

References

Babb, J. and Currie, J. (2008). The brachistochrone problem: Mathematics for a broad audience via a large context problem, *The Montana Mathematics Enthusiast* **5**(2–3): 169–183.

Bellman, R. (1954). The theory of dynamic programming, *Bulletin of the American Mathematical Society* **60**: 503–515.

Bellman, R. (1958). On a routing problem, *Quarterly of Applied Mathematics* **16**: 87–90.

Betts, J.T. (1998). Survey of numerical methods for trajectory optimization, *Journal of Guidance Control and Dynamics* **21**(2): 193–207.

Brodie, M. (2009). *Development of Fusion Motion Capture for Optimisation of Performance in Alpine Ski Racing*, Ph.D. thesis, Massey University, Wellington.

Byrski, A., Dębski, R. and Kisiel-Dorohinicki, M. (2012). Agent-based computing in an augmented cloud environment, *Computer Systems Science and Engineering* **27**(1): 7–18.

Ceriotti, M. and Vasile, M. (2010). MGA trajectory planning with an ACO-inspired algorithm, *Acta Astronautica* **67**(9–10): 1202–1217.

Crauser, A., Mehlhorn, K., Meyer, U. and Sanders, P. (1998). A parallelization of Dijkstra's shortest path algorithm, *in* L. Brim, J. Gruska and J. Zlatuška (Eds.), *Mathematical Foundations of Computer Science*, Springer, London, pp. 722–731.

Dębski, R., Byrski, A. and Kisiel-Dorohinicki, M. (2012). Towards an agent-based augmented cloud, *Journal of Telecommunications and Information Technology* (1): 16–22.

Dębski, R., Krupa, T. and Majewski, P. (2013). ComcuteJS: A web browser based platform for large-scale computations, *Computer Science* **14**(1): 143–152.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik* **1**(1): 269–271.

Dramski, M. (2012). A comparison between Dijkstra algorithm and simplified ant colony optimization in navigation, *Zeszyty Naukowe, Maritime University of Szczecinie* **29**(101): 25–29.

Gaster, B.R., Howes, L.W., Kaeli, D.R., Mistry, P. and Schaa, D. (2013). *Heterogeneous Computing with OpenCL—Revised OpenCL 1.2 Edition*, Morgan Kaufmann, Waltham, MA.

Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA, *in* S. Aluru, M. Parashar, R. Badrinath and V. Prasanna (Eds.), *High Performance Computing—HiPC 2007*, Springer, Berlin, pp. 197–208.

Hirano, Y. (2006). Quickest descent line during alpine ski racing, *Sports Engineering* **9**(4): 221–228.

Jasika, N., Alispahic, N., Elma, A., Ilvana, K., Elma, L. and Nosovic, N. (2012). Dijkstra's shortest path algorithm serial and parallel execution performance analysis, *Proceedings of the 35th International Convention, MIPRO 2012, Opatija, Croatia*, pp. 1811–1815.

Kaps, P., Nachbauer, W. and Mossner, M. (1996). Determination of kinetic friction and drag area in alpine skiing, *in* C. Mote, R. Johnson and W. Hauser (Eds.), *Ski Trauma and Skiing Safety*, Vol. 10, American Society for Testing and Materials, Philadelphia, PA, pp. 165–177.

Lewis, R.M., Torczon, V. and Trosset, M.W. (2000). Direct search methods: Then and now, *Journal of Computational and Applied Mathematics* **124**(1–2): 191–207.

Pontryagin, L.S., Boltyanski, V.G., Gamkrelidze, R.V. and Mischenko, E.F. (1962). *The Mathematical Theory of Optimal Processes*, Interscience, New York, NY.

Pošík, P. and Huyer, W. (2012). Restarted local search algorithms for continuous black box optimization, *Evolutionary Computation* **20**(4): 575–607.

Pošík, P., Huyer, W. and Pál, L. (2012). A comparison of global search algorithms for continuous black box optimization, *Evolutionary Computation* **20**(4): 509–541.

Rippel, E., Bar-Gill, A. and Shimkin, N. (2005). Fast graph-search algorithms for general-aviation flight trajectory generation, *Journal of Guidance, Control, and Dynamics* **28**(4): 801–811.

Singla, G., Tiwari, A. and Singh, D.P. (2013). New approach for graph algorithms on GPU using CUDA, *International Journal of Computer Applications* **72**(18): 38–42.

Stechert, D.G. (1963). *On the Use of the Calculus of Variations in Trajectory Optimization Problems*, Rand Corporation, Santa Monica, CA.

Stillwell, J. (2010). *Mathematics and Its History*, 3rd edn., Springer, New York, NY.

Sussmann, H.J. and Willems, J.C. (1997). 300 years of optimal control: From the brachystochrone to the maximum principle, *IEEE Control Systems* **17**(3): 32–44.

Sussmann, H.J. and Willems, J.C. (2002). The brachistochrone problem and modern control theory, *Contemporary Trends in Nonlinear Geometric Control Theory and Its Applications, Mexico City, Mexico,* pp. 113–166.

Szłapczyński, R. and Szłapczyńska, J.(2012). Customized crossover in evolutionary sets of safe ship trajectories, *International Journal of Applied Mathematics and Computer Science* **22**(4): 999–1009, DOI: 10.2478/v10006-012-0074-x.

Szynkiewicz, W. and Błaszczyk, J. (2011). Optimization-based approach to path planning for closed chain robot systems, *International Journal of Applied Mathematics and Computer Science* **21**(4): 659–670, DOI: 10.2478/v10006-011-0052-8.

Vanderbei, R.J. (2001). Case studies in trajectory optimization: Trains, planes, and other pastimes, *Optimization and Engineering* **2**(2): 215–243.

Vasile, M. and Locatelli, M. (2009). A hybrid multiagent approach for global trajectory optimization, *Journal of Global Optimization* **44**(4): 461–479.

von Stryk, O. and Bulirsch, R. (1992). Direct and indirect methods for trajectory optimization, *Annals of Operations Research* **37**(1): 357–373.

Wuerl, A., Crain, T. and Braden, E. (2003). Genetic algorithm and calculus of variations-based trajectory optimization technique, *Journal of Spacecraft and Rockets* **40**(6): 882–888.

Yokoyama, N. (2002). Trajectory optimization of space plane using genetic algorithm combined with gradient method, *23rd International Congress of Aeronautical Sciences, Toronto, Canada*, pp. 513.1–513.10.

Roman Dębski works as an assistant professor at the Department of Computer Science at the AGH University of Science and Technology. He obtained his M.Sc. in mechanics (1997) and computer science (2002), and a Ph.D. specializing in computational mechanics (2002). Before joining the university he worked in the IT industry for over 15 years. His current interests include mathematical modeling and computer simulations, parallel processing, heterogeneous computing and trajectory optimization. Additionally, he is a ski instructor and a member of the Association of Instructors and Trainers of the Polish Ski Federation.

Appendix
Simulation helper functions API

{@param l: the segment length}
{@param g_{red}: the reduced gravit. acceleration}
{@param t_0: the initial time}
{@param v_0: the initial velocity}
{@returns t_1: the final time}
{@returns v_1: the final velocity}
1: $simulation_for(l, g_{red}, t_0, v_0) : (t_1, v_1)$

{@param r: the segment start node row index}
{@param c: the segment start node column index}
{@param n: the segment end node column index}
{@returns $l^{(s)}$: the segment length}
{@returns $g^{(s)}_{red}$: the segment reduced acceleration}
2: $get_segm_stat_data(r, c, n) : (l^{(s)}, g^{(s)}_{red})$

{@params r, c, n: see $get_segm_stat_data$}
{@returns: $t^{(s)}_1$: the segment final time}
{@returns: $v^{(s)}_1$: the segment final velocity}
3: $get_segm_dyn_data(r, c, n) : (t^{(s)}_1, v^{(s)}_1)$

{@params r, c, n: see $get_segm_stat_data$}
{@param t_{opt}: the best value of t_1 (found so far)}
{@param v_{opt}: the best value of v_1 (found so far)}
{@param $pidx_{opt}$: the previous segment index}
4: $update_segment(r, c, n, t_{opt}, v_{opt}, pidx_{opt})$

{@param r_p: the prev segment start node row idx}
{@param c_p: the prev segment start node col idx}
{@param c_s: the curr segment start node col idx}
{@param c_e: the curr segment end node col idx}
{@param v_{0i}: the final velocity before correction}

{@returns v_0: the final velocity after correction}
5: $correct_v_0(r_p, c_p, c_s, c_e, v_{0i})$

Received: 20 August 2013
Revised: 21 February 2014