GPU Robot Motion Planning using Semi-Infinite Nonlinear Programming

Benjamin Chrétien, Adrien Escande, Abderrahmane Kheddar

GPU Robot Motion Planning using Semi-Infinite Nonlinear Programming

Benjamin Chrétien, Adrien Escande, and Abderrahmane Kheddar, *Senior Member, IEEE*

Abstract—We propose a many-core GPU implementation of robotic motion planning formulated as a semi-infinite optimization program. Our approach computes the constraints and their gradients in parallel, and feeds the result to a nonlinear optimization solver running on the CPU. To ensure the continuous satisfaction of our constraints, we use polynomial approximations over time intervals. Because each constraint and its gradient can be evaluated independently for each time interval, we end up with a highly parallelizable problem that can take advantage of many-core architectures. Classic robotic computations (geometry, kinematics, and dynamics) can also benefit from parallel processors, and we carefully study their implementation in our context. This results in having a full constraint evaluator running on the GPU. We present several optimization examples with a humanoid robot. They reveal substantial improvements in terms of computation performance compared to a parallel CPU version.

Index Terms—GPGPU, CUDA, nonlinear optimization, motion planning, robotics, parallel computing, HPC

1 Introduction

Robotic technology processes numerical data to transform them into physical actions: the essence of any robotic system is *motion*. The science and engineering of computing robot motions to achieve desired goals or tasks is *motion planning*. Since the models used for motion computations and the environment are prone to uncertainties, planned motion are achieved in a closed-loop control. In industrial robotics, since the tasks are predefined, the requirements are to compute motions that maximize performances in terms of speed, robustness, quality... under constraints of motor performances, limitations in terms of robot joint ranges, speed, collision avoidance, etc. Therefore motion planning can be formulated as a solution of an optimization program, e.g. [1]. Motion planning and control formalized as optimization programs can be solved using general-purpose optimization solvers. Yet, even for simple robotic arms, optimizing a whole trajectory is time consuming. Fortunately, for the well-structured environment of an industrial automation, the computation time is not an issue since motions are designed off-line.

The evolution of robotic technology is taking a path reminiscent of computers history. Bill Gates[1] and many other renowned fellows and companies (e.g. Google, Intel), foresee robotics to be the next technology revolution. Indeed, recent research and developments tend to bring robotic technology out of the classical large manufacturing and production lines. Robots are being democratized to invade much more application areas, including production in small-scale flexible enterprises, homes, and a plethora of other services where space is shared with humans. For such perspectives, robot motions are not necessarily driven by classical industrial needs and requirements in terms of power and performance. Indeed, motions are rather human-centric, stylized, smooth, "natural looking", emotional, "artistic", safe, acceptable, etc. The constraints under which the latter behavioral motions occur include similar intrinsic limitations with additional ones related to active perception. Robots will also grow in complexity and shape (e.g. humanoid, multi-arm or soft robots) and may offer redundant structures that allow having several possible motions to fulfill a given task or achieving multiple tasks at a time. In these applications –see e.g. humanoid robots in daily home services[2], for frail person home assistance[3], or in large-scale manufacturing[4]–, motion planning will result from embedded computation including real-time cognitive reasoning, environment sensing, and more complex constraints with robustness handling. Hence, service robotic systems must process on-line larger amounts of information, may embed the computational power, and, more importantly, quickly plan their motion in a safe and reactive way in order to adapt to fast-changing and varying environments.

As a direct consequence, most algorithms that are developed in terms of perception, knowledge processing, artificial intelligence, motion planning and control must be thought in terms of speed, exploiting parallel processing.

Our work focuses on motion planning. The ultimate way

- *B. Chrétien and A. Kheddar are with the CNRS-UM LIRMM, Interactive Digital Human group, Montpellier, France.*
- *B. Chrétien, A. Escande and A. Kheddar are with the CNRS-AIST JRL, UMI3218/RL, Tsukuba, Japan.*

- *Part of this work was presented (oral presentation only) at the International Conference on High Performance Scientific Computing (HPSC), 16-20 March 2015, Hanoi, Vietnam.*

Digital Object Identifier no. 10.1109/TPDS.2016.2521373

1. A robot in every home, Scientific American, January 2007
2. www.robohow.eu
3. www.projetromeo.com
4. www.comanoid.eu

in planning and control of behavioral dynamic motions is to have a look-ahead horizon of time and a set of upcoming tasks, and compute at each control loop the whole-body motion on this time horizon. If the computation power allows computing such trajectories at high frequency, planning and control can be merged. This is what the robotics community is aiming for. In this work, we exploit the GPU architecture in order to achieve dynamic motion planning with an application to humanoid robots (as they are one of the most complex robotic structures). We focus in this paper on the implementation of the motion planning formulated as a nonlinear semi-infinite program [2]. We report our results and discuss some problems we faced when using GPU for this purpose. We exploit several sources of parallelism that are not classical in GPU computation. By doing so, we are able to decrease computation times to a few seconds or less for problems that may require several minutes on multi-core CPUs, and expect a similar speedup for more complex scenarios with computation taking hours [2]. To be self-content and to address non-robotic-specialist readers, we introduce the necessary robotic knowledge in a comprehensive way to explain the implementation choices we made.

2 Problem Formulation

2.1 Robot definition

A robot is composed of a set of rigid objects, called links or bodies, assembled by joints. We consider robots with tree-like structures, i.e. the graph where the bodies are the nodes and the joints are the edges, is a tree. This is called a *kinematic tree*. For a robot with a fixed base, the base is chosen as the root of the tree. For robots without a fixed base, we consider only movements where at any time, at least one body is fixed in the environment (no flight phase). This body (which can be changed along the movement) is taken as a root. For example, for a humanoid robot, when a foot is on the floor, this foot is the root.
We note n the number of joints (so that there are $n+1$ bodies).

2.2 Equation of Motion

The motion of a robot can be described by a function $\boldsymbol{q}$ of the time, where $\boldsymbol{q}(t)$ is the vector of the joint parameters (here angular values) at time t, which is called *configuration*.

Not all functions $\boldsymbol{q}$ are admissible though. They must obey physics laws which, in robotics, are expressed by the Equation of Motion (EoM) and contact with friction modeling. For a robot without a fixed base yet with a fixed body, the EoM writes (see also [3]):

$$\begin{bmatrix}\mathbf{M}_r(\boldsymbol{q})\\ \mathbf{M}_j(\boldsymbol{q})\end{bmatrix}\ddot{\boldsymbol{q}} + \begin{bmatrix}\mathbf{B}_r(\boldsymbol{q},\dot{\boldsymbol{q}})\\ \mathbf{B}_j(\boldsymbol{q},\dot{\boldsymbol{q}})\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \mathbf{I}\end{bmatrix}\boldsymbol{\tau} + \begin{bmatrix}\mathbf{J}_r^{\intercal}(\boldsymbol{q})\\ \mathbf{J}_j^{\intercal}(\boldsymbol{q})\end{bmatrix}\boldsymbol{f} \quad (1)$$

where $\mathbf{M}$ accounts for the inertia of the robot, $\mathbf{B}$ represents the contribution of the gravity and the effect of speed (Coriolis forces, etc.), $\boldsymbol{\tau}$ is the vector of joint torques, $\boldsymbol{f}$ is the vector obtained by stacking all forces $\boldsymbol{f}_k$ ($k \in [1, n_f]$) applied on the robot at points p_k (including contact forces) and $\mathbf{J}$ is the Jacobian matrix of all points p_k, obtained by stacking the matrices $\frac{\partial p_k}{\partial \boldsymbol{q}}$. The upper part of this equation (subscript r, for robot) is directly the Euler-Newton laws expressing that the acceleration and change of angular rate of the robot, seen as a single rigid object, are function of the external forces. It does not appear for fixed-base robots. The lower part (subscript j, for joint) relates the inertia and external forces to the joint torques.
Some forces $\boldsymbol{f}_k$ are further restricted by friction laws (we use Coulomb's laws) to live in a subset $\mathcal{F}$, i.e. $\boldsymbol{f} \in \mathcal{F}$.

2.3 Cartesian quantities

At configuration $\boldsymbol{q}(t)$, the position and orientation of the i-th body w.r.t. a global world frame are given by vector $\boldsymbol{x}_i(\boldsymbol{q}(t))$ and an orientation matrix $\boldsymbol{\Theta}_i(\boldsymbol{q}(t))$, so that a point with coordinates p in the body's frame has coordinates $\boldsymbol{\Theta}_i(\boldsymbol{q}(t))p + \boldsymbol{x}_i(\boldsymbol{q}(t))$ in the world frame. The spatial and angular velocities (resp. accelerations) of the body are denoted by the vectors $\dot{\boldsymbol{x}}(\boldsymbol{q}(t))$ and $\boldsymbol{\omega}_i(\boldsymbol{q}(t))$ (resp. $\ddot{\boldsymbol{x}}_i(\boldsymbol{q}(t))$ and $\dot{\boldsymbol{\omega}}_i(\boldsymbol{q}(t))$), so that the speed of the same point is $\boldsymbol{\omega}_i(\boldsymbol{q}(t)) \times p + \dot{\boldsymbol{x}}_i(\boldsymbol{q}(t))$ (and the acceleration $\dot{\boldsymbol{\omega}}_i(\boldsymbol{q}(t)) \times p + \ddot{\boldsymbol{x}}_i(\boldsymbol{q}(t))$), where $\times$ denotes the cross product. For the sake of clarity, we drop the dependency in $\boldsymbol{q}$ and simply write $\boldsymbol{x}_i(t)$, $\boldsymbol{\Theta}_i(t)$, etc.

We denote by $G(t)$ (geometry) the set of all $\boldsymbol{x}_i(t)$ and $\boldsymbol{\Theta}_i(t)$, and by $K(t)$ (kinematics) the set of all their first and second derivatives.

2.4 General formulation

The problem we are interested in can be written as

$$\min_{\boldsymbol{q},f,\boldsymbol{\tau}} h(\boldsymbol{q}(t), \boldsymbol{f}(t), \boldsymbol{\tau}(t), G(t), K(t)) \quad (2)$$
$$\text{s.t. eq. (1)}$$
$$c_i(\boldsymbol{q}(t), \boldsymbol{f}(t), \boldsymbol{\tau}(t), G(t), K(t)) \geq 0 \; \forall t \in I_i, \forall i = 1 \cdots m$$

with h and c_i real-valued functions, m the number of constraints, $I_i \subseteq [0, T]$ the time interval (possibly a single instant $\{t_i\}$) on which the i-th constraint must be verified and T the total duration of the movement. Whenever we want to express an equality constraint $c = 0$, we replace it by $c + \epsilon \geq 0$ and $\epsilon - c \geq 0$ with ϵ small enough to have a good approximation but not too much to avoid over-constraining the resolution of the problem (ϵ is typically a fraction of the precision we want to achieve, *e.g.* 0.1mm on a position constraint). This is for instance used to define geometric contact conditions. Note that the unknowns of this problem are functions of time.

Functions h and c_i are written in their most generic way here. Even though G and K are functions of $\boldsymbol{q}$, we make them appear explicitly to emphasize their role as intermediate computation quantities. Each function does not need to depend on all $\boldsymbol{q}$, $\boldsymbol{f}$ $\boldsymbol{\tau}$, G and K.

2.5 Cost and constraints

The cost functions that are usually considered are:

- the jerk, for smoothness: $\int_0^T \dddot{\boldsymbol{q}}^2$,
- the energy, for efficiency: $\int_0^T \boldsymbol{\tau}^2$.

Minimizing the total motion time T is considered in [2], and other costs are possible.

As for constraints, we can distinguish several types. First, we use constraints on intrinsic robot's limitations:

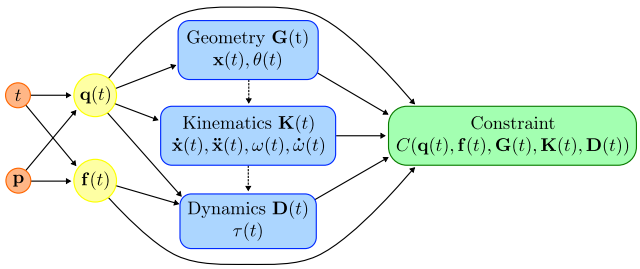


Fig. 1. Variable dependency graph. While the geometry, kinematics and dynamics depend on $\boldsymbol{q}(t)$ (plus $\boldsymbol{f}(t)$ for the dynamics), the kinematics (resp. the dynamics) may reuse data from the geometry (resp. the geometry and the kinematics).

- joint positions $\boldsymbol{q}^- \leq \boldsymbol{q}(t) \leq \boldsymbol{q}^+$,
- joint speeds $\dot{\boldsymbol{q}}^- \leq \dot{\boldsymbol{q}}(t) \leq \dot{\boldsymbol{q}}^+$,
- joint torques $\boldsymbol{\tau}^- \leq \boldsymbol{\tau}(t) \leq \boldsymbol{\tau}^+$.

as well as constraints $\mu_k^2 \boldsymbol{f}_{k,n}^2 - \left\|\boldsymbol{f}_{k,t}\right\|^2 \geq 0$ (where $\boldsymbol{f}_{k,n}$ and $\boldsymbol{f}_{k,t}$ are the normal and tangential part of the force $\boldsymbol{f}_k$ and μ_k the friction coefficient) to translate that $\boldsymbol{f} \in \mathcal{F}$. These constraints write as $c(\boldsymbol{q}) \geq 0$, $c(\boldsymbol{\tau}) \geq 0$, and $c(\boldsymbol{f}) \geq 0$.

Then we have constraints on the bodies positions $c(G) \geq 0$, typically requiring a point with coordinates p in the i-th body to be at a position p^{des} in the world, i.e. $\boldsymbol{\Theta}_i(\boldsymbol{q}(t))p + \boldsymbol{x}_i(\boldsymbol{q}(t)) = p^{\text{des}}$, or that the distance $\delta_{(b_i,b_j)}(t)$ between bodies i and j is greater than a safety margin, to avoid collision.
Similar constraints $c(K) \geq 0$ can be devised. Constraints on body position and velocity are combined to translate high-level tasks.

Finally, we can consider global constraints on the robot:

- position of the center of mass (CoM) $\boldsymbol{x_g}(t)$,
- velocity of the CoM $\dot{\boldsymbol{x}}_{\boldsymbol{g}}(t)$,
- acceleration of the CoM $\ddot{\boldsymbol{x}}_{\boldsymbol{g}}(t)$,
- Center of Pressure (CoP) at some contacts.

Some of these constraints are used to ensure the robot satisfies a stability criterion, e.g. for walking, the CoP should remain within the robot's support polygon.

2.6 Parametrization of variables

The above optimization problem is not solvable because its variables live in a function space of infinite dimension. To make its resolution tractable, we need to restrain the search space by approximating it. We do so by using parametrized functions. The choice of parametrization must be made carefully or there will not be any solution verifying eq. (1) at all time. First, we can see from the lower part of eq. (1) that $\boldsymbol{\tau}$ can be directly obtained from $\boldsymbol{q}$ and $\boldsymbol{f}$. This is called *inverse dynamics*. We can therefore remove it from the variables together with this part of the equation. Second, we show in [3] how to parametrize $\boldsymbol{q}$ and $\boldsymbol{f}$ to satisfy the upper part of eq. (1): any parametrization can be chosen for $\boldsymbol{q}$ and a sub-part $\tilde{\boldsymbol{f}}$ of $\boldsymbol{f}$, the rest of $\boldsymbol{f}$ is deduced from $\boldsymbol{q}$, $\tilde{\boldsymbol{f}}$ and the upper part of eq. (1).

We choose uniform B-Splines as parametrization for $\boldsymbol{q}$ and $\tilde{\boldsymbol{f}}$: for each element $y_i = q_i$ or $y_i = \tilde{\boldsymbol{f}}_i$, N control points are chosen and

$$y_i(t) = \sum_{j=0}^{N-1} \mathbf{p}_{i,j} B_{j,K}(t) \tag{3}$$

where $B_{j,K}$ is a basis function of degree K and $\mathbf{p}_{i,j}$ the associated control point.

All quantities $\boldsymbol{q}$, $\boldsymbol{f}$ $\boldsymbol{\tau}$, G and K depend on t and these parameters $\mathbf{p}_{i,j}$ (see also Fig. 1), thus the problem writes

$$\begin{aligned} &\min_{\mathbf{p}}\ h(\mathbf{p}, t) \\ &\text{s.t. } c_i(\mathbf{p}, t) \geq 0\ \forall t \in I_i, \quad \forall i = 1 \cdots m \end{aligned} \tag{4}$$

This problem is called a *Semi-Infinite Program* (SIP), because while the search space is of finite dimension, the number of constraints is infinite (each c_i for a given t is a constraint). Furthermore, since we are dealing with nonlinear constraints and cost functions, this problem is also a *Nonlinear Program* (NLP).

2.7 Constraint approximation

Various possibilities exist to tackle SIP, the most classical one being to enforce the constraints only at discrete instants t_i [4]. Between two such instants, constraints may be violated [5]. A trade-off must be made between the number of instants (the closer the instants are, the smaller the violation is) and the computation time due to the evaluation of constraints at each instant.

The choice we make is to reformulate the constraint $c_i(\mathbf{p}, t) \geq 0\ \forall t \in I_i$ into $\min_{t \in I_i} c_i(\mathbf{p}, t) \geq 0$. We thus end up with a finite number of constraints, while still ensuring that the original constraints are verified at all time. But evaluating the global minimum of a function over an interval can be far too costly. We therefore make two transformations. First, we split $[0, T]$ into N_{int} intervals T_k, and reformulate the constraints as $\min_{t \in I_i \cap T_k} c_i(\mathbf{p}, t) \geq 0\ k = 1 \cdots N_{\text{int}}$, keeping a finite number of constraints. Second, we make a polynomial approximation of c_i over each $I_i \cap T_k$. By choosing a polynomial of at most degree 5, we are able to compute analytically its global minimum $c_{i,k}^{\min}$ over the interval ($c_{i,k}^{\min}$ is 0 if $I_i \cap T_k = \emptyset$). We end up with an approximation of our problem:

$$\begin{aligned} &\min_{\mathbf{p}}\ h(\mathbf{p}, t) \\ &\text{s.t. } c_{i,k}^{\min}(\mathbf{p}, t) \geq 0\ \forall k \in [1, N_{\text{int}}], \quad \forall i = 1 \cdots m \end{aligned} \tag{5}$$

The accuracy of the polynomial approximation depends on the duration of the time intervals and the polynomial degree chosen. A bound of the error is easier to evaluate than with the time-grid method, although computing it is costly. Thus, we rely on the results presented in [2], and also omit the error bound.

2.8 Optimization process

The desired behavior of the robot is defined by a set of tasks. Each task can be written as a set of equality and/or inequality constraints involving appropriate robot model equations, and so does the cost function. These constraints and the cost function depend on $\boldsymbol{q}(t)$ and its time derivatives, which are replaced by their B-Spline expressions and their derivatives.

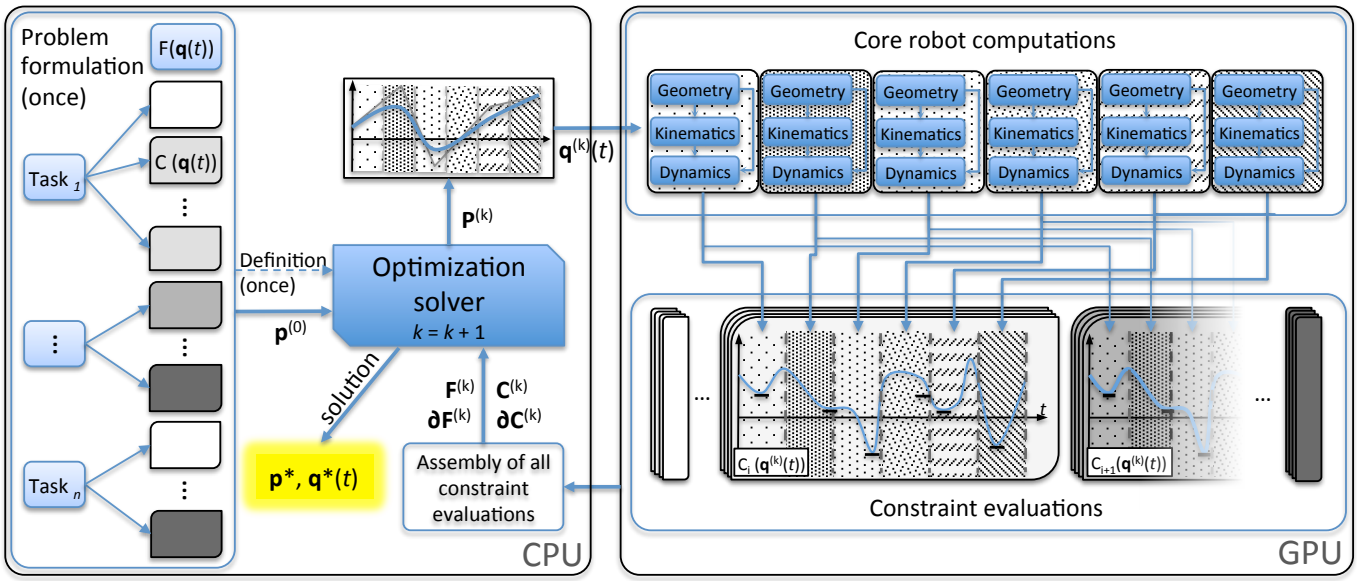


Fig. 2. Optimization process.
The computation is split between the CPU (building the problem, optimization solver, matrices assembly) and the GPU (core computation, constraint evaluations, gradient computation). The background patterns indicate different time intervals T_k, and the background colors refer to different constraint types.

We now define how the constraints and the cost function –most of which, if not all, are nonlinear–, are evaluated.

The Fig. 2 illustrates the process of the formulation we adopted. Given a problem made of high-level tasks, we build an optimization problem that consists in a cost function and a set of (nonlinear) constraints. If a constraint is to be enforced over a given time range I_i, we split it in multiple constraints each spanning a different time interval $I_i \cap T_k$.

Given the optimization problem and an initial guess $\mathbf{p}^{(0)}$, the optimization solver can start its iterative process to find the optimal solution $\mathbf{p}^*$ and its associated optimal joint trajectories $\boldsymbol{q}^*(t)$. At each iteration, the solver provides a new vector of optimization parameters $\mathbf{p}^{(k)}$, which is used to update the joint B-splines $\boldsymbol{q}^{(k)}(t)$. Then, the resulting trajectories are split over the independent time intervals and sent to the GPU-based evaluator.

We start by running the GPU-based dynamics evaluator that comprises the core algorithms: for each time interval, each geometry, kinematics and dynamics computation is done if required by constraints on said interval. The computation on a given interval is completely independent from the other intervals, and the same can be said for the gradient w.r.t. a given control point.

Next, having the factorized data, we evaluate the cost function and constraints (as well as their gradients), still on the GPU. Constraints of a similar type are processed simultaneously and independently, based on the problem definition. We may not have that many constraints of each type to process in parallel, but the computation is usually fast enough and unlikely to become a bottleneck.

Once constraints are evaluated, relevant data (cost function evaluation, constraint evaluations, and gradients) is copied back to the CPU for a final processing step. Since we compute data in a compact form on the GPU, we copy data from our compact data structures to the adequate rows and columns of the full Jacobian matrix. This is what we refer to as *assembly*.

Finally, we feed the optimization solver with the cost function's evaluation $\mathbf{F}^{(k)}$, its gradient $\partial\mathbf{F}^{(k)}$, the constraints $\mathbf{C}^{(k)}$ and the full constraint Jacobian matrix $\partial\mathbf{C}^{(k)}$. The solver will either end the optimization process or iterate with a new $\mathbf{p} = \mathbf{p}^{(k+1)}$.

3 Computation

3.1 Geometry and kinematics

Given the joint trajectories $\boldsymbol{q}$, we can compute the geometry of the robot, that is the orientations (e.g. represented by rotation matrices) and spatial positions $\boldsymbol{x}$ of its bodies. In practice, we note $\boldsymbol{\Theta}_i$ the matrix describing the rotation from the i-th body frame to the global frame.

Let $\lambda(i)$ be the parent index of the i-th body. The preceding joint index is $i-1$, and we can write a recursive formulation:

$$\boldsymbol{\Theta}_0 = R_0 \tag{6}$$

$$\boldsymbol{\Theta}_i(t) = \boldsymbol{\Theta}_{\lambda(i)}(t) S_{i-1} R(\boldsymbol{q}_{i-1}(t)) \tag{7}$$

where R_0 is the orientation of the root body, $R(\boldsymbol{q}_i(t))$ is the rotation matrix associated with the i-th joint, S_i is a static orientation shift associated with the i-th joint. Similarly:

$$\boldsymbol{x}_i(t) = \boldsymbol{x}_{\lambda(i)}(t) + L_i(\boldsymbol{\Theta}_{\lambda(i)}(t), \boldsymbol{\Theta}_i(t)) \tag{8}$$

where L_i is the vector from $\boldsymbol{x}_{\lambda(i)}$ to $\boldsymbol{x}_i$ in the world frame.

The kinematics involves computing $\dot{\boldsymbol{x}}$, $\ddot{\boldsymbol{x}}$, $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$. Since we deal with time polynomials as we will see in Sec. 3.3, a simple polynomial derivation can be done.

3.2 Gradient

For gradient-based optimization solvers, the computational bottleneck is likely to be the evaluation of the Jacobian matrix, that is the matrix that contains all the $\frac{\partial c_i}{\partial p_j}$, where c_i is a

constraint of the problem. For large problems with hundreds or thousands of parameters/constraints, this leads to a high computational load.

The analytical expression of the gradients is also a source of complexity. It is thus common to rely on either finite differences (FD) or automatic differentiation (AD). However, FD suffers from a lack of precision, which can be critical to convergence. Additionally, both FD or AD can be quite expensive computationally [2], even though GPU-based AD is being actively investigated [6].

Also, let us consider the time derivatives $\dot{X}$ and $\ddot{X}$ of a variable $X(\boldsymbol{q}(\boldsymbol{p},t))$, whose gradients are to be computed w.r.t. the parameters p_k of $\boldsymbol{p}$. Let us suppose we want to have a gradient pipeline relying on gradients w.r.t. q_j, that we convert to gradients w.r.t. p_k at the very end. By Schwarz's theorem, we have:

$$\frac{\partial X}{\partial p_k}(t) = \sum_{j=0}^{n_j-1} \frac{\partial X}{\partial q_j}\frac{\partial q_j}{\partial p_k} = \frac{\partial X}{\partial \boldsymbol{q}}\frac{\partial \boldsymbol{q}}{\partial p_k}$$
$$\frac{\partial \dot{X}}{\partial p_k}(t) = \frac{\partial}{\partial t}\left(\frac{\partial X}{\partial p_k}\right) = \frac{\partial X}{\partial \boldsymbol{q}}\frac{\partial^2 \boldsymbol{q}}{\partial t \partial p_k}$$
$$\frac{\partial \ddot{X}}{\partial p_k}(t) = \frac{\partial^2}{\partial t^2}\left(\frac{\partial X}{\partial p_k}\right) = \frac{\partial X}{\partial \boldsymbol{q}}\frac{\partial^3 \boldsymbol{q}}{\partial t^2 \partial p_k}$$

Thus, we are left with the alternative to either compute the pipeline w.r.t. p_k, which implies a larger number of threads used during the whole gradient pipeline computation, or compute it w.r.t. q_j, and add a composition step at the end that takes care of the transformation to gradients w.r.t. p_k, with sums over all the robot joints and multiplications with $\frac{\partial q_j}{\partial p_k}$ and its time derivatives. We currently process the geometry w.r.t. q_j, and the composition is made before computing the kinematics.

3.3 Polynomial approximation

The input of our geometry computation is joint splines, i.e. piecewise polynomial functions. Thus, the computation itself is done on polynomials. Furthermore, the entry point of our computation is the transformation of joint trajectories to rotation matrices, which involves $\cos(\boldsymbol{q}(t))$ and $\sin(\boldsymbol{q}(t))$. Thus, we use a Taylor approximation at the mid-point of each time interval, and continue the computation with these polynomial approximations over time intervals. This approximation is only valid on small time intervals, since we discard polynomial coefficients of higher degree which is only valid if $|\alpha_K t^K| \ll 1 \ \forall t \in T_k$. Anytime an operator such as $\cos$, $\sin$, square root, or even polynomial division is required, a similar approximation is made. For a given polynomial degree, the precision of such approximations can be increased by taking shorter time intervals, thus increasing the number of intervals to process.

For the choice of the polynomial degree K, two things need to be considered. In Sec. 2.7 we choose K such that an analytical expression of polynomial bounds can be computed. However, since we want to be able to use simple polynomial derivations for the kinematics and the dynamics (e.g. $\dot{\boldsymbol{x}}(t)$, $\ddot{\boldsymbol{x}}(t)$), we approximate the rotations with polynomials of degree $K+2$, such that the values at the end of our pipeline have the desired precision with a degree K.

4 Parallelization

4.1 Main ideas

The CPU implementation of the problem in [2] used automatic differentiation to compute the gradients, and OpenMP-based multi-threading to process each interval of the motion independently. The computation took from minutes for simple scenarios to hours for complex multi-contact joint trajectories generation of the HRP-2 humanoid robot; which is clearly not satisfactory. Instead, we investigate a GPGPU approach to see if the computing power of GPUs is beneficial to solve our motion planning problem.

Since most of the computation time is spent in the evaluation of the constraints and their gradients, we dedicated our efforts in implementing this part of the computation. In addition, keeping the separation between the solver and the problem formulation allows to assess the latter using state-of-the-art off-the-shelf nonlinear solvers. Moreover, there are current studies on porting optimization solvers to GPUs [7], and unless they can deal specifically with our problem, we can still consider this part as a black box.

To speedup the computation needed to evaluate the constraints, we consider the following:

Data-independent parallelism: Parts of the computation can be evaluated in parallel and fully independently, i.e. two distinct parallel evaluations will not share any data at any point. This includes:
- Evaluations over time intervals,
- Evaluations of gradients w.r.t. different p_i,
- Evaluations of constraints of the same type.

Data-dependent parallelism: Parts of the computation may rely on parallel algorithms using shared data:
- Model-based parallelism that exploits the properties of the kinematic tree describing the robot.

These different levels of parallelism can be studied separately. Moreover, though the scope or the individual speedup of each parallelization approach may be limited, the cumulative speedup is what actually matters.

4.2 Data-independent parallelism

Each interval (as seen on Fig. 2) can be treated individually. This was already accounted for in [2] where each CPU thread handles a different interval. This is a direct result of Sec. 3.3.

The same observation applies to the gradients w.r.t. the optimization parameters. Let N_{int} be the number of time intervals, K the order of the B-Splines, n the number of joints; we need to evaluate $N_{\text{int}}(K+1)n$ gradients. Hence, the number of gradients that need to be evaluated grows linearly with N_{int}, and they can be computed separately as well.

Time intervals are currently defined by the properties of the B-splines used. If we consider a B-spline of order K with N_{cp} control points, we can split the curve into $N_{\text{int}} = N_{\text{cp}} - K$ intervals (a.k.a. segments or bays) and each control point only influences $K+1$ intervals. Note that in order to refine the polynomial approximation, our implementation can either increase the number of control points or (equivalently) split these time intervals further, which

would, in both cases, increase the number of constraints applied to the problem.

4.3 Data-dependent parallelism

The development of efficient parallel dynamics algorithms for robotics applications has been extensively considered during the last decades [8], [9], [10], [11]. Current trends exploit multi-core and many-core architectures to reduce the computation time [12], [13]. The parallel methods usually provide good speedup for long kinematic chains, but in the case of actual robots, even complex ones, the number of joints rarely exceeds 50 (e.g. humanoid HRP-2: 32 or up to 43 joints with the hands), split over multiple chains, which makes the speedup with parallel methods not substantial.

Thus, we chose to use a parallel method that would reduce synchronization steps as much as possible while allowing the GPU to compute kernels in parallel. As a consequence, we currently use a method inspired from [14] where most of the calculation is done in parallel with fully independent kernels, and the rest is made of prefix operations that can be computed in a logarithmic parallel approach. Such operations have been used for polynomial interpolation [15] or even stream compaction [16], and are good candidates for parallelization. If we consider a binary operator $*$, then the (inclusive) prefix operation of $*$ on a vector $\boldsymbol{x} = [x_0, \dots, x_{n-1}]$ gives:

$$\text{PrefixOp}(*, \boldsymbol{x}) = [x_0, x_0 * x_1, x_0 * x_1 * x_2, \dots] \quad (9)$$

Indeed, efficient parallel prefix operation algorithms exist and are used on the GPU [16], [17], although these algorithms only tackle the case of chain dependencies, while we deal with a tree structure. In order to apply similar algorithms, the values need to be represented in the same frame (e.g. global frame), so that simple additions/multiplications are possible without any expensive frame transformation. Note that we only deal with 30 or 40 bodies, and the longest chain of the kinematic tree is not likely to contain more than half of the bodies, so the parallel speedup is not as important as for prefix sums over large vectors.

5 GPU IMPLEMENTATION

5.1 Main ideas and choices

Using GPUs to accelerate scientific computation has been a very active field of research during the past decade. Both the introduction of NVIDIA's GPGPU computing platform, CUDA, in 2007, and the 2009 release of the OpenCL standard played a major role since it made general purpose computation on GPUs more accessible than it used to be with programmable shaders.

Several fields greatly benefited from GPU parallelization, such as fluid dynamics [18] or machine learning, and the advent of widely-accepted GPGPU standards made it more accessible to researchers. Still, increased computational performances thanks to new highly-parallel processors nearly always involves rethinking both algorithms and data structures in play, in order to achieve important speedups.

In robotics, a GPU implementation of probabilistic motion planning is proposed in [19], [20]. Probabilistic planning consists in sampling the configuration space and choosing the best path through the samples that satisfy the tasks. The number of samples scales with the number of threads, so increasing the computing power of the GPU increases the quality of the solution as well as the chances of finding one. Sample-based methods usually generate jerky motions and are thus coupled with optimization-based post-processing [21].

Instead, our approach plans smooth motions with guaranteed constraints satisfaction using a gradient-based nonlinear solver. This is known to be computationally expensive. Although the problem's complexity makes it an unusual candidate for such parallelization, it also exhibits valuable properties such as strong data independence (e.g. gradient w.r.t. p_i is independent from gradient w.r.t. p_j), high-dimensionality (number of bodies in the robot, number of time intervals, number of control points to parametrize the problem). Solving the problem involves fully separate computation mixed with graph traversal steps. We clearly do not expect to reach the GPU's peak performance with our approach. Yet, our target is to bring the computations much closer to online planning of whole-body dynamic trajectories, which implies computation that takes few seconds, or ideally of a millisecond order to merge planning and control.

In order to ease our implementation, we use the RobOptim library [22], which provides a unified computational model for solving optimization problems. It also makes it easy to switch between different off-the-shelf solvers such as IPOPT [23], CFSQP or NAG, while providing useful features, e.g. sparse matrix support, B-splines, finite-difference checking, etc.

We can split our full computational pipeline into the following distinct steps:

1) Initialize the problem with the robot model, and the goal specification (cost function, constraints, parametrization).
2) Run the iterative solver computation. For each iteration:
   a) Update the computation pipeline with the current iteration's joint trajectories $\boldsymbol{q}(t)$.
   b) Direct computation:
      - Compute the geometry $G(t)$.
      - Compute the forward kinematics $K(t)$.
      - Compute the inverse dynamics $D(t)$.
   c) Gradient computation:
      - Compute the gradient of the geometry.
      - Compute the gradient of the forward kinematics.
      - Compute the gradient of the inverse dynamics.
   d) Evaluate the constraints and their gradients.
   e) Copy constraints data back to the host.
   f) Assemble and fill the solver's data (cost, gradient of cost, constraints, Jacobian matrix).
3) Return the optimal joint trajectories found.

All the memory allocation (on the CPU or the GPU) is done once in 1): the structure of our problem is fixed and does not change during the iterative optimization process.

In 2a) (resp. 2f)), we simply copy data from the CPU to the GPU (resp. from the GPU to the CPU). It is common to do one's best to overlap data transfers with computation on both the CPU and the GPU to maximize performance by alleviating the PCIe bottleneck [24]. However, for the update pipeline, the data to transfer at each iteration is

limited to the polynomial expression of the joint trajectories, which is extremely fast compared to the overall computation time ($\sim 40\mu$s on a GeForce GT 650M for typical scenarios). Hence, asynchronous copy while running another part of the computation is not required.

Similarly, data copied back from the GPU is limited to the constraints output (e.g. bounds of polynomials on time intervals) and their gradients w.r.t. their relevant control parameters. The cost of this copy is negligible.

2b) and 2c) is where the rigid body dynamics equations and their gradient counterparts are computed. Here, we depend on the fact that all the computation relies on previous parameters and fills our data structures (described in Sec. 5.5) while traversing our computation graph. We try to compute as much of it "fully" in parallel (i.e. without any synchronization involved) to maximize throughput.

The evaluation of our constraints happens in 2d): it consists in computing the bounds of polynomials over each time intervals.

5.2 CUDA memory and threading model

Understanding the underlying processing model and memory architecture of any high-performance computing hardware is essential to best take advantage of its computing power [25]. This includes storage limits of the different memory layers, their respective latencies, memory coalescing, caching, etc.

We use NVIDIA's CUDA to parallelize our code on the GPU, so we are using the associated nomenclature for GPGPU programming. On the GPU, the code is executed in groups of 32 threads called warps. These warps are parts of thread blocks, and these blocks form a grid (see Fig. 3).

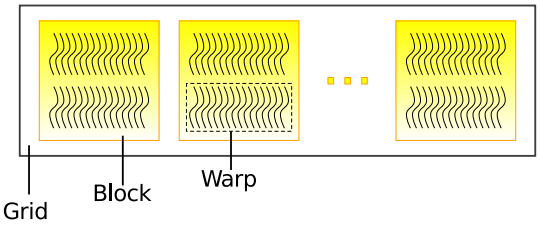


Fig. 3. CUDA threading model. Threads are grouped into warps (32 threads), warps are parts of thread blocks, and blocks are organized into a block grid.

Although the GPU is often considered as a SIMD (Single Instruction Multiple Data) architecture, describing it as a SIMT (Single Instruction Multiple Threads) architecture is more accurate since while threads of a warp do behave like "true" SIMD processors, different GPU multiprocessors with their distinct cores can process different instructions simultaneously.

Loads from global memory are done with 32-, 64- or 128-byte memory transactions. As a result, in order to achieve the best performance possible, data locality and proper alignment need to be guaranteed to prevent uncoalesced memory accesses. For instance, this kind of memory access, shown in Fig. 4, happens if:

- Memory access is misaligned, i.e. data does not start at the beginning of a memory segment,

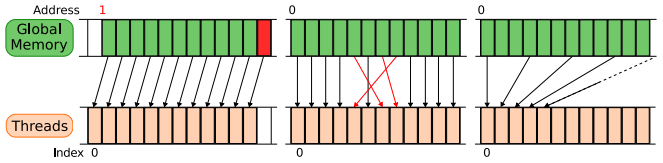


Fig. 4. Possible sources of uncoalesced memory accesses (from left to right): misaligned memory, non-sequential access, strided access.

- Memory access is not sequential, i.e. sequential threads of the same warp access data from the same page not sequentially,
- Memory access is strided, i.e. thread i accesses data$[n \times i]$ with $n > 1$.

Although more recent hardware is able to cope with non-sequential accesses within the same segment, misaligned data will likely lead to an extra load from global memory. As for strided data access, it can severely lower bandwidth, and this can be avoided by either (i) reorganizing data in global memory to get optimal access patterns, or (ii) by loading data from global memory to shared memory with coalesced memory accesses, and then reading from that shared memory since shared memory does not suffer from strided load operations (but it suffers from bank conflicts that need to be addressed instead). These problems need to be considered when designing the data structures used throughout our evaluation pipeline, otherwise we may not be able to achieve our performance goals.

Since threads from the same warp must run the same instructions at the same time, we also want to avoid thread divergence caused by branching. For instance, if threads from the same warp diverge to two branches with a similar complexity, the overall computation will be twice as slow. As a result, algorithms relying extensively on conditional branching may suffer from important performance drops. A notable example is linear algebra, more specifically performance of dense versus sparse matrix computation on the GPU, the latter being more challenging because of its very limited regularity [26]. The nature of our problem is such that only very specific parts are likely to suffer from branching (e.g. solving polynomial equations analytically), but this is still a strong constraint on what can be efficiently added when extending our work.

5.3 Kinematics and dynamics evaluators

Our constraints can involve the geometry, the kinematics and the dynamics of the robot, see Sec. 3.1. Thus, before evaluating these constraints, we need to compute any data required by the set of constraints defining our problem, see dependencies in Fig. 1. The algorithms used to compute data for geometry, kinematics and dynamics constraints are detailed in Alg. 1, Alg. 2 and Alg. 3 respectively.

Parts of the computation where we need to visit the kinematic tree to compute some values are represented as parallel prefix operations as seen in Sec. 4.3. We consider both 3D vector addition and 3D matrix multiplications as operators. For now, prefix operations are done solely with shared memory, but this could be improved using the shuffle instruction over warps available since the Kepler

Algorithm 1 Geometry: with $\boldsymbol{q}(t)$, we compute the positions $\boldsymbol{x}(t)$ and orientations $\boldsymbol{\Theta}(t)$ of the robot's bodies, as in Sec. 3.1. **parfor** is a parallel for loop, *ParPrefixSum* is a parallel prefix sum, and *ParPrefixMult* is a parallel prefix multiplication.

Input: $\boldsymbol{q}(t)$, robot data
Output: $\boldsymbol{x}(t), \boldsymbol{\Theta}(t)$
1: **parfor** $j_i \in \{joints\}$ **do**
    # *Rotation from body* $\lambda(i)$ *to body* i:
2: Compute $R_i(t) = Rot(\boldsymbol{q}_i(t))$
3: **end parfor**
Apply parallel prefix multiplication over kinematic tree
4: $\boldsymbol{\Theta}(t) =$ *ParPrefixMult* $(R(t),$ robot data$)$
5: **parfor** $b_i \in \{bodies\}$ **do**
    # $L_i(t)$ *is the vector* $\overrightarrow{\boldsymbol{x}_i\boldsymbol{x}_{i+1}}(t)$ *expressed in the*
    # *world frame*:
6: *Compute* $L_i(t)$
7: **end parfor**
Apply parallel prefix sum over kinematic tree
8: $\boldsymbol{x}(t) =$ *ParPrefixSum* $(L(t),$ robot data$)$
9: **return** $\boldsymbol{x}(t), \boldsymbol{\Theta}(t)$

Algorithm 2 Forward kinematics: with the geometry, we compute the velocities $\dot{\boldsymbol{x}}(t)$ and accelerations $\ddot{\boldsymbol{x}}(t)$ of the bodies. The $\check{}$ operator returns the vector x associated with a skew-symmetric matrix $\hat{x}$.

Input: $\boldsymbol{x}(t), \boldsymbol{\theta}(t)$,
Output: $\dot{\boldsymbol{x}}(t), \ddot{\boldsymbol{x}}(t), \boldsymbol{\omega}(t), \dot{\boldsymbol{\omega}}(t)$
1: **parfor** $b_i \in \{bodies\}$ **do**
    # *Simple polynomial derivations*:
2: $\dot{\boldsymbol{x}}_i(t) = \dfrac{d\boldsymbol{x}_i}{dt}(t)$ $\qquad$ $\boldsymbol{\omega}_i(t) = \left(\widehat{\boldsymbol{\Theta}_i^\intercal \dfrac{d\boldsymbol{\Theta}}{dt}}\right)(t)$
3: $\ddot{\boldsymbol{x}}_i(t) = \dfrac{d\dot{\boldsymbol{x}}_i}{dt}(t)$ $\qquad$ $\dot{\boldsymbol{\omega}}_i(t) = \dfrac{d\boldsymbol{\omega}_i}{dt}(t)$
4: **end parfor**
5: **return** $\dot{\boldsymbol{x}}(t), \ddot{\boldsymbol{x}}(t), \boldsymbol{\omega}(t), \dot{\boldsymbol{\omega}}(t)$

architecture, especially if all the bodies fit inside a single warp.

In Alg. 3, the expression of contact forces over time $\boldsymbol{f}(t)$ is required, and this value will be provided by an extra block that depends on the results of the forward kinematics and an extra set of optimization parameters. The details on how we will parametrize these forces and the algorithm that will be used are available in [3].

5.4 Gradient computation

The analytical computation of our dynamics pipeline brought some challenging bottlenecks. For instance, let us consider the gradient of the absolute link orientations $\boldsymbol{\Theta}$ w.r.t. the control points of the joint splines. The main difficulty here lies in the kinematic tree structure: the orientation of the link located at a node of the tree depends on all the relative rotations of its predecessors in the tree (cf. eq. (7)).

For the sake of conciseness, let us merge the static orientations S_i to $R(\boldsymbol{q}_i)$. If we try to get the analytical expression

Algorithm 3 Inverse dynamics: with robot's accelerations $\ddot{\boldsymbol{x}}(t)$ and the contact forces acting upon the robot $\boldsymbol{f}(t)$, we compute the joint torques $\boldsymbol{\tau}(t)$. $F_i(t)$ is the general joint force exerted on body i by children bodies $\{\mu(i)\}$, $T_i(t)$ is the general joint torque exerted on body i by children bodies $\{\mu(i)\}$, ${}^iZ_i$ describes the i-th joint's axis, $\boldsymbol{\Theta}_i(t)$ transforms a vector from the body frame to the global world frame, $N_i(t)$ is the resultant torque applied to a rigid body presented in [27], k_i (resp. l_i) is the distance from the body's CoM to the previous (resp. next) link, m_i is the mass of the i-th body.

Input: $\ddot{\boldsymbol{x}}(t), \boldsymbol{f}(t)$, inertial parameters
Output: $\boldsymbol{\tau}(t)$
1: **parfor** $b_i \in \{bodies\}$ **do**
2: $IF_i(t) = m_i(\ddot{\boldsymbol{x}}_i(t) - \boldsymbol{g}) - \boldsymbol{f}_i(t)$
3: **end parfor**
Apply parallel prefix sum over kinematic tree starting
from the leaves
4: $F(t) =$ *ParPrefixSum* $(IF(t))$
5: **parfor** $b_i \in \{bodies\}$ **do**
6: Compute $N_i(t)$
7: $K_i(t) = \boldsymbol{\Theta}_i(t) k_i \boldsymbol{\Theta}_i^\intercal(t) F_i(t)$
8: $L_{\mu(i)}(t) = \boldsymbol{\Theta}_i(t) l_{\mu(i)} \boldsymbol{\Theta}_i^\intercal(t) F_{\mu(i)}(t)$
9: $IT_i(t) = N_i(t) - K_i(t) + \sum_{j \in \{\mu(i)\}} L_j(t)$
10: **end parfor**
Apply parallel prefix sum over kinematic tree starting
from the leaves
11: $T(t) =$ *ParPrefixSum* $(IT(t))$
12: **parfor** $j_i \in \{joints\}$ **do**
13: $\boldsymbol{\tau}_i(t) = \boldsymbol{\Theta}_i^\intercal(t)\ T_i(t) \cdot {}^iZ_i$
14: **end parfor**
15: **return** $\boldsymbol{\tau}(t)$

of the gradient of $\boldsymbol{\Theta}_i$ w.r.t. a control point p_j, we get:

$$\frac{\partial \boldsymbol{\Theta}_i}{\partial p_j}(\boldsymbol{q}) = \frac{\partial \boldsymbol{\Theta}_{\lambda(i)}}{\partial p_j}(\boldsymbol{q})\ R(\boldsymbol{q}_{i-1}) + \boldsymbol{\Theta}_{\lambda(i)}(\boldsymbol{q})\ \frac{\partial R(\boldsymbol{q}_{i-1})}{\partial p_j} \quad (10)$$

We can rewrite this expression as follows:

$$\gamma_i = \gamma_{\lambda(i)}\ \alpha_{i-1} + \beta_i \quad (11)$$

with:
$\alpha_i = R(\boldsymbol{q}_i)$, $\beta_i = \boldsymbol{\Theta}_{\lambda(i)}(\boldsymbol{q})\ \dfrac{\partial R(\boldsymbol{q}_{i-1})}{\partial p_j}$, and $\gamma_i = \dfrac{\partial \boldsymbol{\Theta}_i}{\partial p_j}(\boldsymbol{q})$

By unrolling the recursion of eq. (12) for a simple chain ($\boldsymbol{\Theta}_0$ describes the constant orientation of the root link, and the links are part of the same chain starting from the root), we get:

$$\begin{aligned}
\gamma_0 &= 0 \\
\gamma_1 &= \beta_1 \\
\gamma_2 &= \beta_1\alpha_1 + \beta_2 \\
\gamma_3 &= \beta_1\alpha_1\alpha_2 + \beta_2\alpha_2 + \beta_3 \\
\gamma_4 &= \beta_1\alpha_1\alpha_2\alpha_3 + \beta_2\alpha_2\alpha_3 + \beta_3\alpha_3 + \beta_4 \\
&\vdots
\end{aligned}$$

In the case of a tree, this generalizes into:

$$\gamma_i = \sum_{j=1}^{\text{len}(b_i)-2} \left(\beta_{b_i[j]} \prod_{k=j}^{\text{len}(b_i)-2} \alpha_{b_i[k]} \right) + \beta_i, \quad \forall i \in [1, n] \tag{12}$$

where b_i is the kinematic chain from the root to the i-th link, and $\text{len}(b_i)$ its length.

Let $B = [\beta_i]$ and $\Pi_\alpha = [\pi_\alpha[j,i]]$ where $\pi_\alpha[i,j] = \prod_{k=j}^{\text{len}(b_i)-2} \alpha_{b_i[k]}$. Both of these matrices can be computed in parallel: B in $\mathcal{O}(1)$, and Π_α in $\mathcal{O}(\log_2 n)$.

For a simple kinematic chain, Π_α can be represented as a dense upper-triangular matrix. In the general case, the columns of Π_α do not reach the diagonal when branching is involved (cf. Fig. 5). Moreover, Π_α is to be evaluated and stored only once per time interval, while B is to be computed for each time interval as well as each control point active on the time interval. We do not need to copy these matrices back to the host, but this increases the global memory requirements of our method.

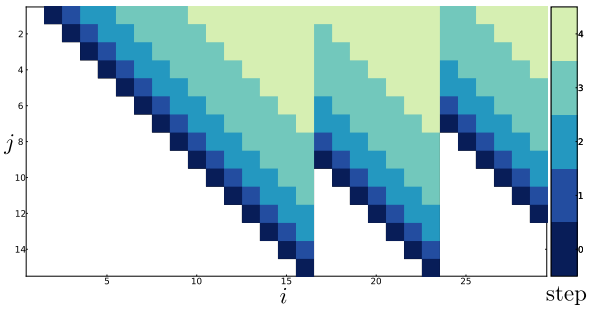


Fig. 5. Parallel filling of Π_α for the simplified HRP-2 model (cf. Fig. 7). Each color represents a different computation step in the logarithmic algorithm.

The computation of Π_α relies on an iterative process described in Alg. 5. The computation plan $P(b)$ (what to compute, with which data, and when) can be precomputed once and for all, and this process is described in Alg. 4.

The logarithmic algorithm used for the computation of Π_α also provides $\boldsymbol{\Theta}_i(\boldsymbol{q})$ for the direct computation of the pipeline. Although nonlinear optimization solvers might have specific queries (e.g. solvers like CFSQP may require the evaluation of individual constraints), we usually end up computing both the direct computation and the gradients during a solver iteration. Thus, the computation of Π_α could also benefit to the direct computation pipeline.

Note that our dynamics simulator can also be used for gradient-free methods: in this case, we would need a solver able to generate multiple queries simultaneously to use the GPU as effectively as possible. Thanks to RobOptim, this can be achieved with any plugin supporting a gradient-free algorithm (e.g. PaGMO's generalized island model [28]).

5.5 Data structures and kernel launch strategy

During the optimization process, we need to store a wide range of data depending on the kind of constraints considered. This can be the robot's geometric data (links spatial positions and orientations) or kinematics data (links velocities). Contact forces and weight, or more generally external forces, also need to be stored for the inverse dynamics. As all of these data describe trajectories on time intervals, each of them is stored as time polynomials.

Since we are dealing mostly with 3×3 polynomial matrices related to each body of the kinematic tree, memory requirements can be quite high. When multiplying such matrices, we can either i) copy the result directly to global memory, ii) use shared memory as a temporary buffer, or iii) try to use only registers to buffer intermediate data before copying the results back to global memory. Since the quantity of shared memory used by each multiprocessor is limited (48kB for our GPUs), the more shared memory each block requires, the fewer blocks can be processed concurrently by the multiprocessor. Thus, the high-level strategy that will minimize computation time depends greatly on

Algorithm 4 Initialization of $P(b)$

Input: list of kinematic chains $b = \{b_i\}$
Output: $P(b)$
 # Maximum length of a chain:
$\text{len}_{\max} = \max(\text{len}(b_i) \text{ for } b_i \in b)$
 # Total number of steps for the logarithmic loop:
$n_{\text{steps}} = \text{ceil}(\log_2(\text{len}_{\max}))$
$P(b) = [\{\} \text{ for } k = 0 \text{ to } n_{\text{steps}} - 1]$
for $b_j \in b$ **do**
  for $i = 0$ **to** $\text{len}(b_j) - 2$ **do**
    $step = \text{ceil}(\log_2(\text{len}(b_j) - 2 - i))$
    # Body index:
    $idx = b_j[i]$
    # We take a predecessor in the same row:
    $v1 = (i, \text{pred}(idx, \text{row} = i))$
    # We take a predecessor in the same column:
    $v2 = (\text{pred}(idx, \text{col} = j), j)$
    # $\Pi_\alpha[i,j] = v1 * v2$ *at given step*:
    $p = \{i, j, v1, v2\}$
    $P(b)[step].add(p)$
  end for
end for
return $P(b)$

Algorithm 5 Parallel computation of Π_α

Input: $A = [\alpha_i]$, list of kinematic chains $b = \{b_i\}$, computation plans for each step $P(b)$
Output: Π_α
 # Initialization (parallel loop):
parfor $b_i \in b$ **do**
  $\Pi_\alpha[\text{len}(b_i) - 1, i] = \alpha_i$
end parfor
 # Maximum length of a chain:
$\text{len}_{\max} = \max(\text{len}(b_i) \text{ for } b_i \in b)$
 # Total number of steps for the logarithmic loop:
$n_{\text{steps}} = \text{ceil}(\log_2(\text{len}_{\max}))$
 # Main (iterative) loop:
for $k = 0$ **to** $n_{\text{steps}} - 1$ **do**
  parfor $p \in P[k]$ **do**
    $\Pi_\alpha[p.row, p.col] = p.v1 * p.v2$
  end parfor
end for
return Π_α

the problem's parameters (number of links, number of time intervals etc.), and choosing an optimal strategy can be difficult, especially since the CUDA compiler may rearrange instructions and memory transactions extensively to reduce memory latency and maximize throughput.

Based on the typical dimensions of our problem ($\sim$30 or 40 bodies in our humanoid robots, $\sim$10 or 20 time intervals if we want to generate a motion in an online planning scenario), and the fact that steps where data are to be shared across threads only involve different bodies; we decided to map computation per time intervals and per control points (for gradient computation) to individual thread blocks, and each robot body is assigned to a different thread index. Although this implies that we currently rely on a low number of threads per block, a higher number of threads can be used with more complex kinematic trees or if multiple robots are processed simultaneously. The latter could help support *multistart* efficiently, that is solving the same problem simultaneously with multiple different initial conditions to reduce the impact of local minima. Besides, several kernels merely involve registers and global memory, without any shared work between threads. For these kernels, a simple heuristic that sets the number of threads per blocks and the total number of blocks to increase occupancy could be used.

In the meantime, going over 32 bodies leads to –at least– twice the number of warps executed, but for scenarios without any manipulation tasks, models with fixed hand joints are below the threshold. In order to avoid uncoalesced memory accesses, data for different robot bodies is thus stored contiguously in memory. Moreover, since our robot structures are constant, access patterns are known at compilation time, which may help the compiler optimize kernels even more.

Note that storage requirements could be critical for large problems running on older GPUs without enough RAM, but all hardware we have on hand satisfy our current needs. Still, for a multistart scenario, the memory limits could then become another bottleneck to take into account, since computation would need to be split in multiple batches that are processed serially.

Constant data, e.g. the robot's inertial parameters or the graph parameters used for the forward kinematics and the inverse dynamics, are stored in texture memory. A similar choice can be made for bounding volume geometries for collision detection when it is implemented.

Each thread deals with polynomials, which leads to an inner loop over the polynomial coefficients when performing operations on them. Since the polynomial size is fixed during the compilation, we can unroll this inner loop, and get improved performance thanks to *instruction-level parallelism* (ILP) that helps hide some of the memory latency [29]. Granting the impact is limited since we are not dealing with high-degree polynomials, it still provides good performance at lower occupancy since all the computation is done with such polynomials.

5.6 Sparsity structure

Exploiting the sparsity of the problem has several advantages:

- it reduces the computation involved when querying the constraint Jacobian,
- it reduces the memory footprint (less allocations/copies),
- it helps the solver optimize its computation.

If we look at the structures of the Jacobian matrices from our example scenarios in Fig. 6, we can clearly distinguish constraints that concern individual joints (e.g. limits on $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$) from those that concern control points on different joint trajectories (e.g. constraints on $\boldsymbol{x}$). For the latter, we currently consider a dense block over the corresponding time interval, but this could be improved by applying a filter based on the robot's kinematic tree structure, e.g. the position of a link only depends on the joints that link it to the free-flying body. This would not change anything w.r.t. the simulator, but would make filling the Jacobian faster and help the solver to converge.

6 Results

We assess our implementation using motion planning with a humanoid robot. In the following examples, our optimization parameters are the control points of the uniform cubic B-splines used to parametrize the joint trajectories.

Tests are conducted on different NVIDIA GPUs whose proprieties are summarized in Table 1. We compare the performance of the optimization process when dealing with single- or double-precision floating-point numbers, and with the previous parallel CPU version.

6.1 Scenarios

Our test scenarios are made with the HRP-2 humanoid robot with motions lasting several seconds. We use constraints described in Sec. 2.5 without a cost function to evaluate the performance of our library. We also froze some joints (e.g. hands, neck) since there is no manipulation done. We end up with a total number of 28 joints (29 bodies). The position of the main contact link is given (right foot).

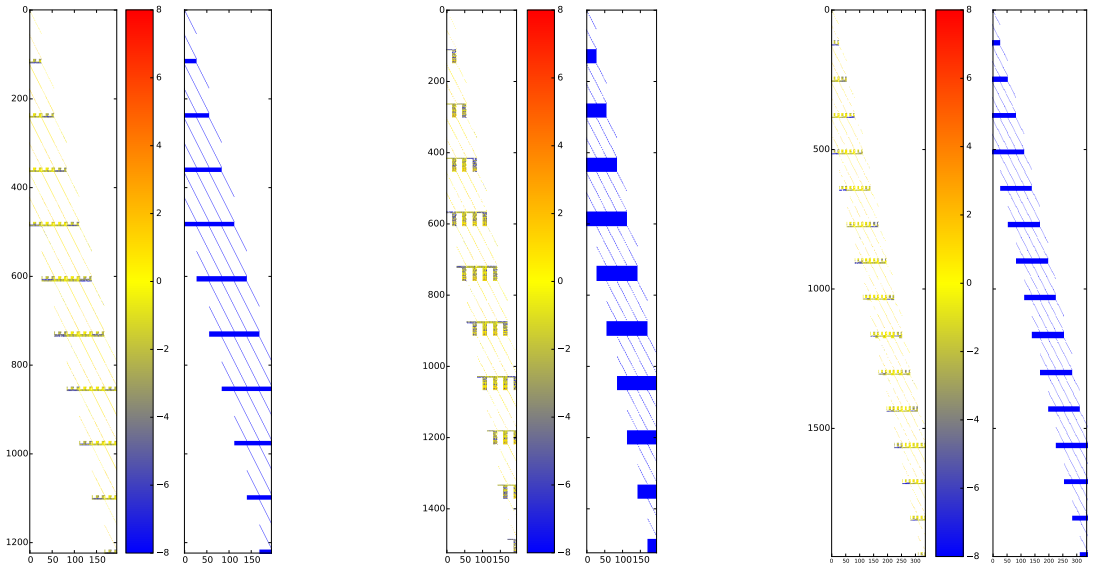
Scenario 1 The robot starts in the half-sitting configuration with its right hand horizontal (must be kept so all the time); then lowers its CoM below a given threshold (while shifting it above the support polygon of its other foot); then gets back to the original configuration. All of it is done under joint position and speed limits.

Scenario 2 Same as the previous one, except that instead of constraining the right hand, a constraint to keep its two feet on the ground is used (i.e. additional equality constraints on the second foot for position and orientation).

Scenario 3 The robot achieves a kick motion with a minimum velocity of 1m/s, while keeping its CoM above its right foot, keeping its right hand fixed in position and orientation (with a small margin of error) under joint position and speed limits.

6.2 Timings

The reported timings are the time spent in the optimization process, i.e. we clock the call to RobOptim's `solve()` method. The initialization time is not included, since the allocations and preprocessing are done once and for all.



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Fig. 6. Jacobian matrices for each scenario presented in Sec. 6.1. The left matrix describes the $\log_{10}$ of the Jacobian matrix's amplitudes, and the right one shows the actual sparsity structure taken into account. Dense blocks represent unfiltered constraints w.r.t. the interval's control points.

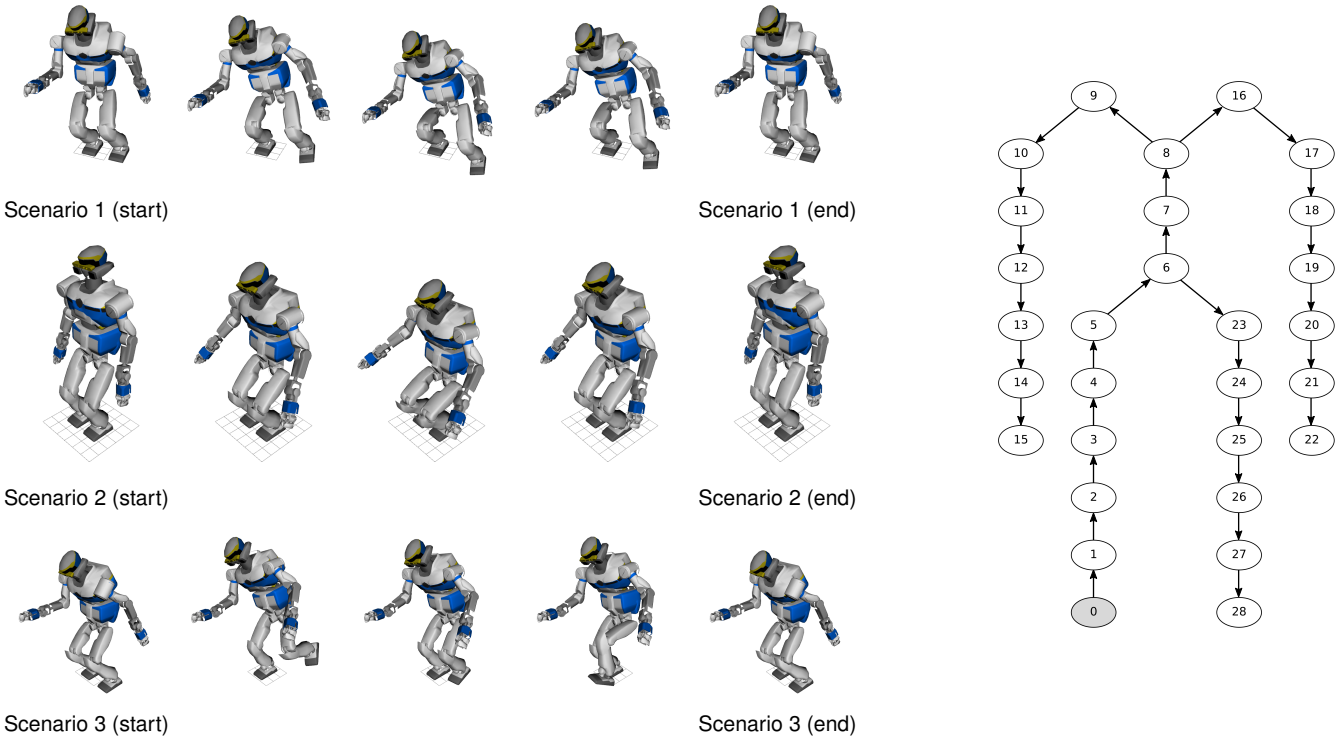


Fig. 7. Snapshots from whole-body motion planning with the HRP-2 humanoid robot, and the associated simplified kinematic tree (root = right foot).

Thus, timings include GPU computation, memory transfers between host and device, solver data filling and computation done in the NLP solver (IPOPT). Even though computing the inverse dynamics is not mandatory (since there is no contact forces), we still included its computation to give a fair order of the computation time we expect, even though adding contact forces will increase the number of optimization parameters resulting in a more expensive gradient computation for the inverse dynamics.

Three different setups are considered:

- Desktop with a Tesla C2070 GPU and an Intel Xeon W5590 CPU @ 3.33GHz,
- Laptop with a GeForce GT 650M GPU and an Intel Core i7-3740QM CPU @ 2.70GHz,
- Server with a GeForce GTX Titan Black GPU and an Intel Core i7-4930K CPU @ 3.40GHz.

On the laptop running the optimization with the GeForce GT 650M GPU, computation time (excluding the solver) for

Model	Tesla C2070	GeForce GT 650M	GeForce GTX Titan Black
Microarchitecture	Fermi	Kepler	
Compute Capability (CC)	2.0	3.0	3.5
# Shared Multiprocessors (SMs)	14	2	15
# cores/SM	32	192	
Memory (MiB)	5375	2048	6144
GPU max clock rate (MHz)	1147	835	980
32-bit registers per thread	63		255
Maximum amount of shared memory per MP (KB)	48		

TABLE 1
NVIDIA GPUs considered for these tests: older Tesla card, mobile GPU, and more recent high-end hardware.

Scenarios	Scenario 1	Scenario 2	Scenario 3
n **(input)**	196	196	336
m **(output)**	1224	1524	1960
Nonzeros in the Jacobian	11424	34944	30688
Motion time (s)	4	4	5

TABLE 2
Scenarios with the HRP-2 humanoid model (29 bodies). We use cubic B-splines for the parametrization of joint trajectories. 10 intervals were chosen for the first and second scenarios, and 15 for the third one.

a typical iteration of the second scenario is classified as follows:

- Direct computation: 9.27%
- Gradient computation: 75.27%
- Filling RobOptim's sparse Jacobian matrix: 14.71%
- Others: 0.75%

We reached the point where simply fetching data copied back from the GPU in compact form and filling the global sparse Jacobian with it is no longer negligible w.r.t. the total computation time. If we also take into account the average time spent in the solver at each iteration, we get:

- Time spent in the parallel simulator: 91.69%
- Time spent in the solver (IPOPT): 8.31%

Although we can reduce the number of nonzeros in the Jacobian matrix by taking the kinematic tree into account, or optimize the sparse Jacobian filling process, time spent inside the NLP solver is out of our control for we are using off-the-shelf solvers.

Since historically, GPUs have been used for computer graphics were high precision is rarely required, most GPUs will have a much higher throughput in single precision than in double precision. The theoretical ratio for peak performance FP64/FP32 may vary from $1/2$ to $1/32$ depending on the hardware. This difference is explained by a varying number of cores dedicated to double-precision computation, which is often lower for cards designed for the gaming industry. Thus, we made the choice of supporting both, since a higher precision may lead to a faster convergence as seen with the third scenario in Table 3 (NLP solvers deal with double-precision floats), but a lower precision allows for faster computation on the GPU.

We disabled ECC (error-correcting code) support on the Tesla card, since this feature is not available on the other cards. It is not that relevant in this context: computation lasts a few seconds, and the solver may be able to recover from most single-bit errors should they happen. Still, if we manage to reduce computation time even more and start merging our planning with low-level control, it would be worth reconsidering ECC errors that could potentially lead to critical failures on the robot.

The previous parallel CPU version presented in [2] supported more features (e.g. contact forces, collision detection), so we are currently unable to make a direct comparison for the resolution of full problems. For similar scenarios however (no constraint on forces or torques, no cost function, no collision detection, same number of intervals), a timing comparison between the CPU and GPU versions is available in Table 4. We summarize average timings per iteration to compare the computational load independently from the optimization process itself. For all the CPUs considered, the maximum number of processors available was lower than the number of time intervals.

As for more complex scenarios involving $n \approx 2000$ optimization parameters and $m \approx 30000$ constraints, scaling was an issue for the CPU version, since it took several hours to compute. Even though this represents highly challenging problems, we still expect to run the computation with our GPU approach in (at most) a matter of minutes for such extreme scenarios. A lot of effort was also spent over the years to optimize the performance of the CPU library (e.g. avoiding costly allocations), which can still be done for our GPU implementation.

6.3 Limitations

GPUs provide competitive accelerators that can speed up computation time substantially for highly-parallelizable problems. Yet, GPU parallelization remains a challenging task for a wide range of applications. Making GPGPU computation more accessible without sacrificing performance is being actively researched [30]. GPGPU frameworks evolve to improve usability: the introduction of Unified Virtual Memory in CUDA reduces the quantity implementation that had to be managed manually by the developers, for GPUs with Compute Capability 3.0 or higher.

Most high-level GPGPU libraries such as Thrust or ArrayFire rely on high-level abstractions to perform classical operations (sum, scan, etc.) on simple data types (integer, floats). Here, we are far from the traditional use case, i.e. polynomial computation with multiple logical dimensions. Moreover, we may be sacrificing performance for ease of use by using generic implementations not adapted for our use case. As a result, we developed our own low-level polynomial GPU library customized for our problem.

Running our code on different architectures revealed how difficult choosing the best implementation strategy for a given kernel is. For instance, some accumulations provide acceptable performance on the GT 650M card with atomic operations on global memory, but the Tesla C2070 had a better throughput when relying on temporary shared memory.

Scenarios		Scenario 1		Scenario 2		Scenario 3	
Floating-point precision		single	double	single	double	single	double
IPOPT iterations		13	14	12	12	19	16
Computation time (s): total (average per iter.)	**Tesla C2070**	0.422 (0.032)	0.833 (0.060)	0.541 (0.045)	0.872 (0.073)	1.027 (0.054)	1.508 (0.094)
	GeForce GT 650M	0.787 (0.061)	1.535 (0.110)	0.865 (0.072)	1.485 (0.124)	1.833 (0.096)	2.755 (0.172)
	GeForce GTX Titan Black	0.312 (0.024)	0.426 (0.030)	0.392 (0.033)	0.496 (0.041)	0.680 (0.036)	0.766 (0.048)

TABLE 3
Timings (total and average per iteration) for the HRP-2 scenarios with both single- and double-precision floats, for each GPU, and without any cost function. This includes geometry, kinematics, dynamics (albeit not required), constraint evaluations and their gradient counterpart, and time spent in the solver. Note that an IPOPT iteration may not always involve a constraint Jacobian query, so for two different scenarios, even if the number of iterations is similar, the actual number of Jacobian evaluations may be different.

Computer	Desktop (Tesla C2070)		Laptop (GT 650M)		Server (GTX Titan Black)	
	Avg. (ms/iter.)	Speedup	Avg. (ms/iter.)	Speedup	Avg. (ms/iter.)	Speedup
CPU (1 thread)	2715	1.0	2178	1.0	2235	1.0
CPU (2 threads)	1747	1.6	1144	1.9	1255	1.8
CPU (4 threads)	1011	2.7	712	3.1	697	3.2
CPU (6 threads)	-	-	654	3.3	521	4.3
CPU (8 threads)	-	-	602	3.6	515	4.4
CPU (10 threads)	-	-	-	-	485	4.6
CPU (12 threads)	-	-	-	-	479	4.7
GPU (double)	94	28.9	172	12.7	48	46.6
GPU (single)	54	50.3	96	22.7	36	62.1

TABLE 4
Average time per iteration in ms for the parallel CPU version (for multiple thread counts) and our GPU version (for single- and double-precision floats), with associated speedup relative to the single-threaded CPU version. The scenario considered has $n \approx 300$, with geometry and kinematics constraints ($m \approx 2000$), and 15 time intervals. Computation on the CPU is done with double-precision floating numbers. This table gives an idea of the speedup to expect for the core computation on different systems, and we expect an even better speedup once highly-parallelizable collision constraints are added.

Also, when venturing off the beaten tracks, it is not uncommon to find errors in the dependencies of one's project. In our case, since our method is not usually ported to GPUs, we encountered several bugs in the CUDA compiler, and the long release cycle of proprietary software can then become an extra problem that needs to be dealt with.

7 Conclusion

Our work lays outside the box of usual GPGPU problems. It paves the way for a full-fledged GPU multi-contact motion planning library dedicated to complex robotic systems. By leveraging the high dimensionality as well as its sparsity structure, and formulating the problem at best of what could be parallelized, we managed to compute in a matter of seconds or less continuous whole-body trajectories of our motion planning on the GPU. This result brings us closer to real-time and hence to a closed-loop whole-body model preview controller (the Holy Grail researched in robotics). There is still room for improvements regarding the computation times. Ideally, the solver would also be part of the GPU and even customized to robotic problems to gain higher factors of performance.

Our future work involves integrating contact forces and their parametrization to the optimization problem and the GPU pipeline [3]. Since the sequence of contacts is found by a contact planner [31] or given by the user, the numbers of contacts and related parameters are known a priori. Plus, we design the problem such that any change in the contact configuration happens on interval boundaries, allowing us to integrate them in the current framework. Contact forces may be discontinuous on such changes. If continuity is required by the motion design, it can be enforced with additional constraints. As for collision avoidance, some interesting works specifically target distance computation along a trajectory (*e.g.* [32]), but rely on iterative computations (GJK algorithm), while we would rather seek a closed-form expression of the distance (of a chosen norm) that could be easily integrated to our polynomial-based GPU pipeline. In the long term, we also plan to embed at best performances parts of the solver in the GPU architecture.

Acknowledgments

This work is supported by EU FP7 IP RoboHow.Cog (www.robohow.eu), and by the Japan Society for Promotion of Science (JSPS): Postdoctoral Fellowship P13786, and Grant-in-Aid for Scientific Research (B) 25280096.

References

[1] M. Guilbert, L. Joly, and P.-B. Wieber, "Optimization of complex robot applications under real physical limitations," *The International Journal of Robotics Research*, vol. 27, no. 5, pp. 629–644, 2008.

[2] S. Lengagne, J. Vaillant, E. Yoshida, and A. Kheddar, "Generation of Whole-body Optimal Dynamic Multi-Contact Motions," *International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1104–1119, Apr. 2013.

[3] B. Chrétien, A. Escande, and A. Kheddar, "Continuously satisfying constraints with contact forces in trajectory optimization for humanoid robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.

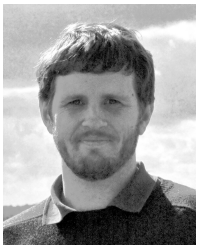
[4] S. Miossec, K. Yokoi, and A. Kheddar, "Development of a software for motion optimization of robots - Application to the kick motion of the HRP-2 robot," in *IEEE International Conference on Robotics and Biomimetics*, 2006, pp. 299–304.

[5] S. Lengagne, N. Ramdani, and P. Fraisse, "Guaranteed computation of constraints for safe path planning," *2007 7th IEEE-RAS International Conference on Humanoid Robots*, 2007.

[6] G. Kozikowski and B. Kubica, "Interval Arithmetic and Automatic Differentiation on GPU Using OpenCL," in *Applied Parallel and Scientific Computing*, ser. Lecture Notes in Computer Science, P. Manninen and P. Öster, Eds. Springer Berlin Heidelberg, 2013, vol. 7782, pp. 489–503.

[7] E. Smith, J. Gondzio, and J. Hall, "GPU Acceleration of the Matrix-Free Interior Point Method," in *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 681–689.
[8] R. Featherstone, "A divide-and-conquer articulated-body algorithm for parallel $\mathcal{O}(\log(n))$ calculation of rigid-body dynamics. part 1: Basic algorithm," *The International Journal of Robotics Research*, vol. 18, no. 9, pp. 867–875, Sep. 1999.
[9] ——, "A divide-and-conquer articulated-body algorithm for parallel $\mathcal{O}(\log(n))$ calculation of rigid-body dynamics. Part 2: Trees, loops, and accuracy," *The International Journal of Robotics Research*, vol. 18, no. 9, pp. 876–892, Sep. 1999.
[10] K. Yamane and Y. Nakamura, "Parallel $\mathcal{O}(\log n)$ algorithm for dynamics simulation of humanoid robots," in *IEEE-RAS International Conference on Humanoid Robots*, 2006, pp. 554–559.
[11] ——, "Comparative Study on Serial and Parallel Forward Dynamics Algorithms for Kinematic Chains," *International Journal of Robotics Research*, vol. 28, no. 5, pp. 622–629, May 2009.
[12] A. Tasora, D. Negrut, and M. Anitescu, "GPU-based parallel computing for the simulation of complex multibody systems with unilateral and bilateral constraints: an overview," *Multibody Dynamics*, vol. 23, 2011.
[13] K. Bhalerao, J. Critchley, and K. Anderson, "An efficient parallel dynamics algorithm for simulation of large articulated robotic systems," *Mechanism and Machine Theory*, 2012.
[14] J. J. Zhang, Y. F. Lu, and B. Wang, "A nonrecursive Newton-Euler formulation for the parallel computation of manipulator inverse dynamics," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 28, no. 3, pp. 467–471, 1998.
[15] O. Eğecioğlu, E. Gallopoulos, and C. K. Koç, "A parallel method for fast and practical high-order newton interpolation," *BIT*, vol. 30, no. 2, pp. 268–288, Jun. 1990.
[16] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.
[17] S. W. Ha and T. D. Han, "A scalable work-efficient and depth-optimal parallel scan for the GPGPU environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2324–2333, 2013.
[18] E. Rustico, G. Bilotta, A. Herault, C. Del Negro, and G. Gallo, "Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 43–52, Jan 2014.
[19] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 187–200, 2012.
[20] C. Park, J. Pan, and D. Manocha, "High-Dof Robots in Dynamic Environments Using Incremental Trajectory Optimization," *International Journal of Humanoid Robotics*, vol. 11, no. 02, 2014.
[21] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," *IEEE International Conference on Robotics and Automation*, 2009.
[22] T. Moulard, B. Chrétien, and E. Yoshida, "Software Tools for Nonlinear Optimization– Modern Solvers and Toolboxes for Robotics," *Journal of the Robotics Society of Japan*, vol. 32, no. 6, pp. 536–541, 2014.
[23] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," in *Mathematical Programming*. Springer-Verlag, 2006, vol. 106, no. 1, pp. 25–57.
[24] B. V. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for CPU-GPU data transfers," *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 11–20, 2014.
[25] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems & Software*, Mar. 2010, pp. 235–246.
[26] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," *Nvidia Technical Report*, pp. 1–32, 2008.
[27] H. Hemami, "A state space model for interconnected rigid bodies," *IEEE Transactions on Automatic Control*, 1982.
[28] D. Izzo, M. Ruciski, and F. Biscani, "The generalized island model," in *Parallel Architectures and Bioinspired Algorithms*, ser. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2012, vol. 415, pp. 151–169.
[29] V. Volkov, "Better performance at lower occupancy," *Proceedings of the GPU Technology Conference*, vol. 10, 2010.
[30] T. Han and T. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, Jan 2011.
[31] A. Escande, A. Kheddar, and S. Miossec, "Planning contact points for humanoid robots," *Robotics and Autonomous Systems*, vol. 61, no. 5, pp. 428–442, 2013.
[32] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.

Benjamin Chrétien is a Doctoral student at CNRS-UM LIRMM since October 2012. He received his MSc degree in Aerospace Engineering from ISAE SUPAERO, Toulouse, France, in 2012, and currently prepares his thesis under the supervision of Abderrahmane Kheddar. His research interests include humanoid robotics, high-performance computing, GPGPU algorithms, numerical optimization and motion planning.



Adrien Escande received the MSc degree in 2005 from École des Mines de Paris, France and the PhD degree in 2008 in robotics from Université d'Évry Val-d'Essonne, France after spending three years in the CNRS-AIST Joint Robotics Laboratory (JRL), UMI3218/CRT, Tsukuba, Japan. He then worked as a research scientist in CEA-LIST at Fontenay-aux-Roses, France, until the end of 2012 and is now back at JRL. His current research interests include whole-body planning and control for humanoid robots and mathematical optimization for robotics.

Abderrahmane Kheddar received the BSCS degree from the Institut National d'Informatique (ESI), Algiers, the MSc and PhD degrees in robotics, both from the University of Pierre and Marie Curie, Paris 6. He is presently Directeur de Recherche at CNRS. He is the Director of the CNRS-AIST Joint Robotic Laboratory (JRL), UMI3218/RL, Tsukuba, Japan; and the leader of the Interactive Digital Humans (IDH) team at CNRS-UM LIRMM, Montpellier, France. His research interests include humanoid robotics, haptics, and thought-based control using brain machine interfaces. He is a founding member of the IEEE/RAS chapter on haptics, the co-chair and co-founding member of the IEEE/RAS Technical committee on model-based optimization. He is presently Editor of the IEEE Transactions on Robotics, and the Journal of Intelligent and Robotic Systems; he is a founding member of the IEEE Transactions on Haptics and served in its editorial board in 2007-2010, he also served as associate editor in the MIT Press PRESENCE. He coordinated or acted as a PI for several EU projects. He is titular member of the National Academy of Technologies of France (NATF) and a Senior Member of the IEEE Society.