GPU Robot Motion Planning using Semi-Infinite Nonlinear Programming
Benjamin Chrétien, Adrien Escande, Abderrahmane Kheddar

To cite this version:
Benjamin Chrétien, Adrien Escande, Abderrahmane Kheddar. GPU Robot Motion Planning using Semi-Infinite Nonlinear Programming. IEEE Transactions on Parallel and Distributed Systems, Institute of Electrical and Electronics Engineers, 2016, 27 (10), pp.2926-2939. 10.1109/TPDS.2016.2521373. hal-01266581

HAL Id: hal-01266581
https://hal.archives-ouvertes.fr/hal-01266581
Submitted on 4 Feb 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 Gates1 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, smart, “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 services2, for frail person home assistance3, or in large-scale manufacturing4, 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
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 q of the time, where q(t) is the vector of the joint parameters (here angular values) at time t, which is called configuration.

Not all functions 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]):

\[
\begin{bmatrix}
  M_r(q) & B_r(q, \dot{q}) \\
  B_r(q, \dot{q}) & B_j(q, \dot{q})
\end{bmatrix}
\begin{bmatrix}
  \ddot{q} \\
  \dot{\tau}
\end{bmatrix}
= \begin{bmatrix}
  0 \\
  I
\end{bmatrix}
+ \begin{bmatrix}
  J_r^T(q) \\
  J_j^T(q)
\end{bmatrix} f
\] (1)

where M accounts for the inertia of the robot, B represents the contribution of the gravity and the effect of speed (Coriolis forces, etc.), \( \tau \) is the vector of joint torques, f is the vector obtained by stacking all forces \( f_k \) (\( k \in [1, n_j] \)) applied on the robot at points \( p_k \) (including contact forces) and J is the Jacobian matrix of all points \( p_k \), obtained by stacking the matrices \( \frac{\partial q_i}{\partial q} \). 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 \( f_k \) are further restricted by friction laws (we use Coulomb’s laws) to live in a subset \( F \), i.e. \( f \in F \).

2.3 Cartesian quantities

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

We denote by \( G(t) \) (geometry) the set of all \( x_i(t) \) and \( \Theta_i(t) \), 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

\[
\begin{align*}
\min_{q, f, \tau, \Theta} & \quad h(q(t), f(t), \tau(t), G(t), K(t)) \\
\text{s.t.} & \quad \text{eq. (1)} \\
& \quad c_i(q(t), f(t), \tau(t), G(t), K(t)) \geq 0 \quad \forall t \in I_i, \forall i = 1 \cdots m
\end{align*}
\] (2)

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 \( \epsilon \) small enough to have a good approximation but not too much to avoid over-constraining the resolution of the problem (\( \epsilon \) 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 \( q \), we make them appear explicitly to emphasize their role as intermediate computation quantities. Each function does not need to depend on all \( q, f, \tau, G \) and \( K \).

2.5 Cost and constraints

The cost functions that are usually considered are:

\begin{itemize}
  \item the jerk, for smoothness: \( \int_0^T \dddot{q}^2 \)
  \item the energy, for efficiency: \( \int_0^T \ddot{\tau}^2 \)
\end{itemize}

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:
are chosen and
\[ y_i(t) = \sum_{j=0}^{N-1} p_{i,j} B_{j,K}(t) \]  
where \( B_{j,K} \) is a basis function of degree \( K \) and \( p_{i,j} \) the associated control point.

All quantities \( q, f, \tau, G \) and \( K \) depend on \( t \) and these parameters \( p_{i,j} \) (see also Fig. 1), thus the problem writes
\[
\min_{p} h(p, t) \\
\text{s.t. } c_i(p, t) \geq 0 \forall t \in I_i, \quad \forall i = 1 \cdots m
\]
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 \( c_i(p, t) \geq 0 \forall t \in I_i \) into \( \min_{t \in I_i} c_i(p, t) \geq 0 \). 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 \( N_{\text{int}} \) intervals \( T_k \), and reformulate the constraints as \( \min_{t \in I_i \cap T_k} c_i(p, t) \geq 0 \) for a given \( k \) and \( N_{\text{int}} \), 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 \( c_i^{\text{min}}(t) \) over the interval \( (c_i^{\text{min}}(t) = 0 \) if \( I_i \cap T_k = \emptyset \)). We end up with an approximation of our problem:
\[
\min_{p} h(p, t) \\
\text{s.t. } c_i^{\text{min}}(p, t) \geq 0 \forall k \in [1, N_{\text{int}}], \quad \forall i = 1 \cdots m
\]

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 \( q(t) \) and its time derivatives, which are replaced by their B-Spline expressions and their derivatives.

### 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 \( \tau \) can be directly obtained from \( q \) and \( f \). 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 \( q \) and \( f \) to satisfy the upper part of eq. (1); any parametrization can be chosen for \( q \) and a sub-part of \( f \), the rest of \( f \) is deduced from \( q, \dot{f} \) and the upper part of eq. (1).
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 $p^{(0)}$, the optimization solver can start its iterative process to find the optimal solution $p^*$ and its associated optimal joint trajectories $q^*(t)$. At each iteration, the solver provides a new vector of optimization parameters $p^{(k)}$, which is used to update the joint B-splines $q^{(k)}(t)$. 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 $F^{(k)}$, its gradient $\partial F^{(k)}$, the constraints $C^{(k)}$ and the full constraint Jacobian matrix $\partial C^{(k)}$. The solver will either end the optimization process or iterate with a new $p = p^{(k+1)}$.

3 Computation

3.1 Geometry and kinematics

Given the joint trajectories $q$, we can compute the geometry of the robot, that is the orientations (e.g. represented by rotation matrices) and spatial positions $x$ of its bodies. In practice, we note $\Theta_i$ 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:

$$
\Theta_0 = R_0
$$

$$
\Theta_i(t) = \Theta_{\lambda(i)}(t) S_{i-1} R(q_{i-1}(t))
$$

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

$$
x_i(t) = x_{\lambda(i)}(t) + L_i(\Theta_{\lambda(i)}(t), \Theta_i(t))
$$

where $L_i$ is the vector from $x_{\lambda(i)}$ to $x_i$ in the world frame.

The kinematics involves computing $\dot{x}, \ddot{x}, \omega$ and $\ddot{\omega}$. 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 $\partial p_i / \partial p_j$, 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 $X(q(p,t))$, whose gradients are to be computed w.r.t. the parameters $p_k$ of $p$. 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:

\[
\frac{\partial X}{\partial p_k}(t) = \sum_{j=0}^{n-1} \frac{\partial X}{\partial q_j} \frac{\partial q_j}{\partial p_k} = \frac{\partial X}{\partial q} \frac{\partial q}{\partial p_k} \\
\frac{\partial X}{\partial p_k}(t) = \frac{\partial}{\partial t} \left( \frac{\partial X}{\partial p_k} \right) = \frac{\partial X}{\partial q} \frac{\partial q}{\partial p_k} \\
\frac{\partial X}{\partial p_k}(t) = \frac{\partial^2}{\partial t^2} \left( \frac{\partial X}{\partial p_k} \right) = \frac{\partial X}{\partial q} \frac{\partial^2 q}{\partial t \partial p_k}
\]

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 $\frac{\partial q}{\partial p_k}$ 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 $\cos(q(t))$ and $\sin(q(t))$. 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. $\dot{\mathbf{x}}(t)$, $\ddot{\mathbf{x}}(t)$), 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 $N_{\text{int}}$ 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 $N_{\text{int}}$, 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 $N_{\text{cp}}$ 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 $x = [x_0, \ldots, x_{n-1}]$ gives:

$$\text{PrefixOp}(*, x) = [x_0, x_0 \ast x_1, x_0 \ast x_1 \ast x_2, \ldots]$$ (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 $q(t)$.
   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 GPU or the CPU) 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 (∼40µs 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).

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,
- 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 $q(t)$, we compute the positions $x(t)$ and orientations $\Theta(t)$ of the robot’s bodies, as in Sec. 3.1. \texttt{parfor} is a parallel for loop, \texttt{ParPrefixSum} is a parallel prefix sum, and \texttt{ParPrefixMult} is a parallel prefix multiplication.

\textbf{Input:} $q(t)$, robot data  
\textbf{Output:} $x(t), \Theta(t)$

1: \texttt{parfor} $j_i \in \{\text{joints}\}$ do
2: \hspace{1em} # Rotation from body $\lambda(i)$ to body $i$:
3: \hspace{2em} Compute $R_{\lambda}(t) = \text{Rot}(q_i(t))$
4: \endparfor
5: # Apply parallel prefix multiplication over kinematic tree
6: \texttt{parfor} $b_i \in \{\text{bodies}\}$ do
7: \hspace{1em} # $L_i(t)$ is the vector $\vec{x}_i - \vec{x}_{i+1}(t)$ expressed in the
8: \hspace{2em} # world frame:
9: \hspace{2em} Compute $L_i(t)$
10: \endparfor
11: # Apply parallel prefix sum over kinematic tree
12: $x(t) = \text{ParPrefixSum} \left( L(t), \text{robot data} \right)$
13: return $x(t), \Theta(t)$

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

\textbf{Input:} $x(t), \Theta(t)$  
\textbf{Output:} $\dot{x}(t), \ddot{x}(t), \omega(t), \dot{\omega}(t)$

1: \texttt{parfor} $b_i \in \{\text{bodies}\}$ do
2: \hspace{1em} # Simple polynomial derivations:
3: \hspace{2em} $\dot{x}_i(t) = \frac{dx_i}{dt}(t)$  \hspace{1em} $\omega_i(t) = \left( \Theta_i \frac{d\Theta_i}{dt}(t) \right)$
4: \hspace{2em} $\ddot{x}_i(t) = \frac{d\dot{x}_i}{dt}(t)$  \hspace{1em} $\dot{\omega}_i(t) = \frac{d\omega_i}{dt}(t)$
5: \endparfor
6: return $\dot{x}(t), \ddot{x}(t), \omega(t), \dot{\omega}(t)$

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 $\Theta$ 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 $R(q_i)$. If we try to get the analytical expression architecture, especially if all the bodies fit inside a single warp.

In Alg. 3, the expression of contact forces over time $f(t)$ 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].

Algorithm 3 Inverse dynamics: with robot’s accelerations $\ddot{x}(t)$ and the contact forces acting upon the robot $f(t)$, we compute the joint torques $\tau(t)$. $F(t)$ 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 $\{\mu(i)\}$, $Z_i$ describes the $i$-th joint’s axis, $\Theta_i(t)$ transforms a vector from the body frame to the global world frame, and $N_i(t)$ 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.

\textbf{Input:} $\ddot{x}(t), f(t)$, inertial parameters  
\textbf{Output:} $\tau(t)$

1: \texttt{parfor} $b_i \in \{\text{bodies}\}$ do
2: \hspace{1em} $IF_i(t) = m_i(\ddot{x}_i(t) - g) - f_i(t)$
3: \endparfor
# Apply parallel prefix sum over kinematic tree starting # from the leaves
4: $F(t) = \text{ParPrefixSum} \left( IF(t) \right)$
5: \texttt{parfor} $b_i \in \{\text{bodies}\}$ do
6: \hspace{1em} Compute $N_i(t)$
7: \hspace{2em} $K_i(t) = \Theta_i(t) k_i \Theta_i^T(t) F_i(t)$
8: \hspace{2em} $L_{\mu(i)}(t) = \Theta_i(t) l_{\mu(i)} \Theta_i^T(t) F_i(t)$
9: \hspace{2em} $IT_i(t) = N_i(t) - K_i(t) + \sum_{j \in \{\mu(i)\}} L_j(t)$
10: \endparfor
# Apply parallel prefix sum over kinematic tree starting # from the leaves
11: $T(t) = \text{ParPrefixSum} \left( IT(t) \right)$
12: \texttt{parfor} $j_i \in \{\text{joints}\}$ do
13: \hspace{1em} $\tau_i(t) = \Theta_i^T(t) T_i(t) \cdot Z_i$
14: \endparfor
15: return $\tau(t)$

of the gradient of $\Theta_i$ w.r.t. a control point $p_j$, we get:

$$\frac{\partial \Theta_i}{\partial p_j} q)) \frac{\partial R(q_{i-1})}{\partial p_j} \frac{\partial R(q_{i-1})}{\partial p_j} (10)$$

We can rewrite this expression as follows:

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

with:

$$\alpha_i = R(q_i), \beta_i = \Theta_{\lambda(i)}(q) \frac{\partial R(q_{i-1})}{\partial p_j}, \text{and } \gamma_i = \frac{\partial \Theta_i}{\partial p_j}(q)$$

By unrolling the recursion of eq. (12) for a simple chain ($\Theta_0$ describes the constant orientation of the root link, and the links are part of the same chain starting from the root), we get:

$$\gamma_0 = 0$$
$$\gamma_1 = \beta_1$$
$$\gamma_2 = \beta_1 \alpha_1 + \beta_2$$
$$\gamma_3 = \beta_1 \alpha_2 + \beta_2 \alpha_2 + \beta_3$$
$$\gamma_4 = \beta_1 \alpha_3 + \beta_2 \alpha_2 \alpha_3 + \beta_3 \alpha_3 + \beta_4$$

...
In the case of a tree, this generalizes into:

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

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 \( \Pi_a = [\pi_a[j,i]] \) where \( \pi_a[i,j] = \prod_{k=j}^{\text{len}(b_i)-2} \alpha_{b_i[k]} \). Both of these matrices can be computed in parallel: \( B \) in \( O(1) \), and \( \Pi_a \) in \( O(\log_2 n) \).

For a simple kinematic chain, \( \Pi_a \) can be represented as a dense upper-triangular matrix. In the general case, the columns of \( \Pi_a \) do not reach the diagonal when branching is involved (cf. Fig. 5). Moreover, \( \Pi_a \) 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.

The computation of \( \Pi_a \) 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 \( \Pi_a \) also provides \( \Theta_i(q) \) 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 \( \Pi_a \) 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 \( 3 \times 3 \) 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:** \( P(b) \)

1. \# Maximum length of a chain:
   
   \[ \text{len}_{\text{max}} = \max(\text{len}(b_i) \text{ for } b_i \in b) \]

2. \# Total number of steps for the logarithmic loop:
   
   \[ n_{\text{steps}} = \lceil \log_2(\text{len}_{\text{max}}) \rceil \]

3. \# Body index:
   
   \[ P(b)[i,j] = \text{len}(b_i) - 2 - i ) \]

4. \# We take a predecessor in the same row:
   
   \[ v1 = \{ i, \text{pred}(idx, \text{row} = i) \} \]

5. \# We take a predecessor in the same column:
   
   \[ v2 = \{ \text{pred}(idx, \text{col} = j), j \} \]

6. \# Main (iterative) loop:
   
   \[ \text{for } k = 0 \text{ to } n_{\text{steps}} - 1 \text{ do} \]

7. \[ P(b)[p,row,p.col] = p.v1 + p.v2 \]

8. \[ \text{end for} \]

**Algorithm 5** Parallel computation of \( \Pi_a \)

**Input:** \( A = [\alpha_i], \) list of kinematic chains \( b = \{b_i\} \), computation plans for each step \( P(b) \)

**Output:** \( \Pi_a \)

1. \# Initialization (parallel loop):
   
   \[ \text{parfor } b_i \in b \text{ do} \]

2. \[ \Pi_a[\text{len}(b_i) - 1, i] = \alpha_i \]

3. \[ \text{end parfor} \]

4. \# Maximum length of a chain:
   
   \[ \text{len}_{\text{max}} = \max(\text{len}(b_i) \text{ for } b_i \in b) \]

5. \# Total number of steps for the logarithmic loop:
   
   \[ n_{\text{steps}} = \lceil \log_2(\text{len}_{\text{max}}) \rceil \]

6. \# Main (iterative) loop:
   
   \[ \text{for } k = 0 \text{ to } n_{\text{steps}} - 1 \text{ do} \]

7. \[ \text{parfor } p \in P[k] \text{ do} \]

8. \[ \Pi_a[p.row,p.col] = p.v1 + p.v2 \]

9. \[ \text{end parfor} \]

10. \[ \text{end for} \]

11. \[ \text{return } \Pi_a \]
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 (≈30 or 40 bodies in our humanoid robots, ~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 $\dot{q}$) from those that concern control points on different joint trajectories (e.g. constraints on $x$). 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.
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
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.

---

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

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

**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.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>n (input)</td>
<td>196</td>
<td>196</td>
<td>336</td>
</tr>
<tr>
<td>m (output)</td>
<td>1224</td>
<td>1524</td>
<td>1960</td>
</tr>
<tr>
<td>Nonzeros in the Jacobian</td>
<td>11424</td>
<td>34944</td>
<td>30688</td>
</tr>
<tr>
<td>Motion time (s)</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
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 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


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 Ecole 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), UM3218/CR, 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 BSc 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), UM3218/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.