Accelerating Robot Dynamics Gradients on a CPU, GPU, and FPGA

Brian Plancher[1], Sabrina M. Neuman[1], Thomas Bourgeat[2], Scott Kuindersma[1,3], Srinivas Devadas[2], Vijay Janapa Reddi[1]

***Abstract*—Computing the gradient of rigid body dynamics is a central operation in many state-of-the-art planning and control algorithms in robotics. Parallel computing platforms such as GPUs and FPGAs can offer performance gains for algorithms with hardware-compatible computational structures. In this paper, we detail the designs of three faster than state-of-the-art implementations of the gradient of rigid body dynamics on a CPU, GPU, and FPGA. Our optimized FPGA and GPU implementations provide as much as a 3.0x end-to-end speedup over our optimized CPU implementation by refactoring the algorithm to exploit its computational features, e.g., parallelism at different granularities. We also find that the relative performance across hardware platforms depends on the number of parallel gradient evaluations required.**

***Index Terms*—Computer Architecture for Robotics and Automation, Hardware-Software Integration in Robotics, Dynamics**

I. Introduction

SPATIAL algebra-based approaches to rigid body dynamics [1] have become a central tool for simulating and controlling robots due their ability to efficiently and exactly compute dynamic quantities and their gradients [2], [3], [4], [5]. Despite being highly optimized, existing implementations of these approaches do not take advantage of opportunities for parallelism present in the algorithm, limiting their performance [6], [7], [8]. This is critical because computing the gradient of rigid body dynamics accounts for 30% to 90% of the total computational time of typical nonlinear model-predictive control (MPC) implementations [9], [10], [7], [11].

Furthermore, these algorithms are typically run on off-the-shelf CPUs, but the performance of CPU hardware has been limited by thermal dissipation, enforcing a utilization wall that restricts the performance a single chip can deliver [12], [13]. Alternative computing platforms such as GPUs and FPGAs provide opportunities for higher performance and throughput by better exploiting parallelism.

In this work, we describe and compare the design of three accelerated implementations of the gradient of rigid body dynamics targeting three different computing platforms: a CPU, a GPU, and an FPGA. Each platform requires different design trade-offs (e.g., how to exploit parallelism), as specific algorithmic features map better or worse to each platform.

Our contribution is an optimized implementation of the gradient of rigid body dynamics for the GPU with a 6.5x speedup over an existing state-of-the-art GPU implementation, and end-to-end speedups of up to 3.0x over an optimized CPU implementation. We show that changing the problem partitioning to target the specific hardware platform can improve performance, enabling us to achieve a 2.8x speedup between our own initial and final GPU designs. We compare our final GPU design to an optimized FPGA implementation and find that while both outperform the CPU, the application context in which the kernel is run impacts which hardware platform performs best. In our experiments, at low numbers of parallel computations the FPGA outperforms the GPU, while the GPU scales better to higher numbers of parallel computations. We provide source code accompanying this work at http://bit.ly/fast-rbd-grad.

II. Related Work

Prior work on GPUs implemented sample-based motion planning largely through Monte Carlo roll-outs [14], [15], [16], [17], [18], [19], [20], while FPGAs have predominantly been used for fast mesh and voxel-based collision detection [21], [22], [23], [24], [25], [26], [27]. For dynamic trajectory optimization, several recent efforts leveraging multi-core CPUs and GPUs indicate that computational benefits from parallelism are possible [28], [2], [3], [29], [11], [30].

However, current state-of-the-art rigid body dynamics packages are not optimized for GPUs [11], and no packages exist for FPGAs. Prior examples of FPGA and accelerator implementations of MPC are either limited to linear dynamics or target cars, drones, and other lower degrees of freedom systems with only one or two rigid bodies [31], [32], [33].

GPUs and other parallel architectures have historically been used to accelerate gradient computations through numerical differentiation [34], [35]. However, these methods have been shown to have less favorable numerical properties when used for nonlinear MPC applications. By contrast, automatic

Manuscript received: October 15, 2020; Revised January, 5, 2021; Accepted January 19, 2021.

This paper was recommended for publication by Editor Tamim Asfour upon evaluation of the Associate Editor and Reviewers' comments.

This material is based upon work supported by the National Science Foundation (under Grant DGE1745303 and Grant 2030859 to the Computing Research Association for the CIFellows Project), and the Defense Advanced Research Projects Agency (under Grant HR001118C0018). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding organizations.

[1]Brian Plancher, Sabrina M. Neuman, Scott Kuindersma, and Vijay Janapa Reddi are with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA. {brian_plancher@g, sneuman@seas, scottk@seas, vj@eecs}.harvard.edu

[2]Thomas Bourgeat and Srinivas Devadas are with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA. bthom@csail.mit.edu, devadas@mit.edu

[3]Scott Kuindersma is also with Boston Dynamics, Waltham, MA.

differentiation of differentiable rigid body physics engines has shown promise for real-time nonlinear MPC use on the CPU [36]. Existing differentiable physics implementations for the GPU, however, were designed for machine learning, computer graphics, or soft robotics [37], [38], [39], [40], [41], [42]. As such, they are optimized for simulating large numbers of interacting bodies through contact using maximal coordinate and particle-based approaches. These approaches have been shown to produce less accurate solutions when used for rigid body robotics applications over longer time step durations [43].

III. Rigid Body Dynamics and its Gradients

In this section we provide background on rigid body dynamics and its gradients, describing the algorithm and its key features.

A. Algorithm Overview

State-of-the-art recursive formulations of rigid body dynamics are described using spatial algebra [1]. These formulations operate in minimal coordinates and compute the joint acceleration $\ddot{q} \in \mathbb{R}^n$ as a function of the joint position $q \in \mathbb{R}^n$, velocity $\dot{q} \in \mathbb{R}^n$, and input torque $\tau \in \mathbb{R}^m$. These formulations leverage either the Articulated Body Algorithm (ABA) or a combination of the Composite Rigid Body Algorithm (CRBA) and the Recursive Newton-Euler Algorithm (RNEA), as follows:

$$\begin{aligned} &\ddot{q} = \text{ABA}(q, \dot{q}, \tau, f^{ext}) \\ &\ddot{q} = M^{-1}(\tau - c) \ \text{ where } \ M = \text{CRBA}(q) \\ &\qquad\qquad\qquad\qquad\quad c = \text{RNEA}(q, \dot{q}, 0, f^{ext}). \end{aligned} \tag{1}$$

The RNEA is shown in Algorithm 1.

Spatial algebra represents quantities as operations over vectors in $\mathbb{R}^6$ and matrices in $\mathbb{R}^{6\times 6}$, defined in the frame of each rigid body. These frames are numbered $i = 1$ to n such that each body's parent λ_i is a lower number. Transformations of quantities from frame λ_i to i are denoted as ${}^iX_{\lambda_i}$ and can be constructed from the rotation and transformation between the two coordinate frames, which themselves are functions of the joint position q_i between those frames and constants derived from the robot's topology. The mass distribution of each link is denoted by its spatial inertia I_i. S_i is a joint-dependent term denoting in which directions a joint can move (and is often a constant). v_i, a_i, f_i represent the spatial velocity, acceleration, and force of each rigid body.

Finally, spatial algebra uses spatial cross product operators $\times$ and $\times^*$, in which a vector is re-ordered into a matrix, and then a standard matrix multiplication is performed. The reordering is shown in Equation 2 for a vector $v \in \mathbb{R}^6$:

$$\begin{aligned} v\times &= \begin{bmatrix} 0 & -v[2] & v[1] & 0 & 0 & 0 \\ v[2] & 0 & -v[0] & 0 & 0 & 0 \\ -v[1] & v[0] & 0 & 0 & 0 & 0 \\ 0 & -v[5] & v[4] & 0 & -v[2] & v[1] \\ v[5] & 0 & -v[3] & v[2] & 0 & -v[0] \\ -v[4] & v[3] & 0 & -v[1] & v[0] & 0 \end{bmatrix} \\ v\times^* &= -v \times^T . \end{aligned} \tag{2}$$

Algorithm 1 RNEA$(q, \dot{q}, \ddot{q}, f^{ext}) \to c$

```
1: $v_0 = 0,\ a_0 = \text{gravity}$
2: for link $i = 1 : n$ do
3:     Compute ${}^iX_{\lambda_i}, S_i, I_i$
4:     $v_i = {}^iX_{\lambda_i} v_{\lambda_i} + S_i \dot{q}_i$
5:     $a_i = {}^iX_{\lambda_i} a_{\lambda_i} + S_i \ddot{q}_i + v_i \times S_i \dot{q}_i$
6:     $f_i = I_i a_i + v_i \times^* I_i v_i - f_i^{\text{ext}}$
7: for link $i = n : 1$ do
8:     $c_i = S_i^T f_i$
9:     $f_{\lambda_i} \mathrel{+}= {}^iX_{\lambda_i}^T f_i$
```

Algorithm 2 ∇RNEA$(\dot{q}, v, a, f, X, S, I) \to \partial c/\partial u$

```
1: for link $i = 1 : N$ do
2:     $\frac{\partial v_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} \left({}^iX_{\lambda_i} v_{\lambda_i}\right) \times S_i & u \equiv q \\ S_i & u \equiv \dot{q} \end{cases}$
3:     $\frac{\partial a_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} \left({}^iX_{\lambda_i} a_{\lambda_i}\right) \times S_i \\ v_i \times S_i \end{cases}$
4:     $\frac{\partial f_i}{\partial u} = I_i \frac{\partial a_i}{\partial u} + \frac{\partial v_i}{\partial u} \times^* I_i v_i + v_i \times^* I_i \frac{\partial v_i}{\partial u}$
5: for link $i = N : 1$ do
6:     $\frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u}$
7:     $\frac{\partial f_{\lambda_i}}{\partial u} \mathrel{+}= {}^iX_{\lambda_i}^T \frac{\partial f_i}{\partial u} + {}^iX_{\lambda_i}^T \left(S_i \times^* f_i\right)$
```

Algorithm 3 ∇Dynamics$(q, \dot{q}, \ddot{q}, f^{ext}) \to \partial \ddot{q}/\partial u$

```
1: $v', a', f', X, S, I \leftarrow \text{RNEA}(q, \dot{q}, \ddot{q}, f_{ext})$
2: $\partial c'/\partial u = \nabla\text{RNEA}(\dot{q}, v', a', f', X, S, I)$
3: $\partial \ddot{q}/\partial u = -M^{-1} \partial c'/\partial u$
```

Previous work showed that by using the $\ddot{q} = M^{-1}(\tau - c)$ approach and applying insightful linear algebra simplifications, the computation of the gradient of robot dynamics can be greatly accelerated on a CPU [7]. The results of that work include simplified analytical derivatives of the RNEA shown in Algorithm 2 (where u represents either q or $\dot{q}$), and a simplification of the overall gradient computation to a three step process shown in Algorithm 3. This process first computes the values v', a', f', X, S, I by running the RNEA (Algorithm 1) with $\ddot{q}$ used as an input (instead of 0 as shown in Equation 1). It then uses those values as inputs to the simplified analytical derivatives of the RNEA (Algorithm 2), and then transforms the resulting gradient, $\partial c'/\partial u$, into the final output $\partial \ddot{q}/\partial u$ by multiplying it by $-M^{-1}$. Our implementations are based on this approach.[1]

B. Key Algorithmic Features

In order to design accelerated implementations of the gradient of rigid body dynamics for use with nonlinear MPC, it is important to identify its key algorithmic features. These structural properties of the algorithm interact and impact the

[1]In this work we do not focus on the computation of M^{-1} as this can be cached from a previous step in many nonlinear MPC implementations.

computation differently on each hardware platform, as will be discussed in Section IV.

Coarse-Grained Parallelism: Many modern nonlinear MPC implementations have a step that requires tens to hundreds of independent computations of the gradient of rigid body dynamics [44], [45], [46], [11], [47], offering parallelism across these long-running, independent computations.

Fine-Grained Parallelism: The dynamics gradient algorithm contains opportunities for additional shorter-duration parallelism between each column j of the computation of $\partial c'/\partial u_j$, and also in low-level mathematical operations, e.g., between the many independent dot-product operations within a single matrix-vector multiplication.

Structured Sparsity: The underlying matrices used throughout the algorithm exhibit sparsity patterns that can be derived from the robot's topology. For example, robots with only revolute joints can be described such that all $S_i = [0, 0, 1, 0, 0, 0]^T$. In this way, all computations that are right-multiplied by S_i can be reduced to only computing and extracting the third column (or value). There are also opportunities to exploit structured sparsity in the transformation matrices and cross product matrices as shown in Equation 2.

Data Access Patterns: The gradient of rigid body dynamics exhibits regular access patterns through its ordered loops (assuming the local variables for each link j are stored regularly in memory). This enables quick and easy computations of memory address locations, batching of loads and stores, and even "pre-fetching" of anticipated memory loads in advance. However, the cross product operations are reorderings, leading to irregular memory access patterns.

Sequential Dependencies: Throughout the dynamics gradient algorithm, local variables have references to parent links whose values are computed in previous loop iterations.

Working Set Size: The dynamics gradient algorithm has a relatively small working set. It most frequently accesses only a few local variables between loop iterations, and a set of small matrices, in particular, I_i and ${}^iX_{\lambda_i} \in \mathbb{R}^{6\times6}$.

I/O Overhead: Depending on the hardware partitioning of the algorithm, the gradient of rigid body dynamics can require that a substantial amount of input and output (I/O) data is sent to and received from the GPU or FPGA coprocessor.

TABLE I
ALGORITHMIC FEATURES OF THE GRADIENT OF RIGID BODY DYNAMICS AND QUALITATIVE ASSESSMENTS OF THEIR SUITABILITY FOR DIFFERENT TARGET HARDWARE PLATFORMS.

Algorithmic Features	CPU	GPU	FPGA
Coarse-Grained Parallelism	moderate	excellent	moderate
Fine-Grained Parallelism	poor	moderate	excellent
Structured Sparsity	good	moderate	excellent
Irregular Data Patterns	moderate	poor	excellent
Sequential Dependencies	good	poor	good
Small Working Set Size	good	moderate	excellent
I/O Overhead	excellent	poor	poor

IV. CPU, GPU, AND FPGA DESIGNS

This section describes the designs of our optimized CPU, GPU, and FPGA implementations of the gradient of rigid body dynamics for use with MPC. In these designs we advantageously mapped the algorithmic features of the dynamics gradient algorithm, presented in Section III-B, to each hardware platform. Table I offers qualitative assessments of how well these features can be exploited by the different hardware platforms, which informed which advantageous features we leveraged, or disadvantageous bottlenecks we mitigated, in each implementation.

A. CPU Design

Our CPU implementation is based on Algorithm 2. In order to efficiently balance the coarse-grained parallelism exposed by tens to hundreds of computations of the dynamics gradient across a handful of processor cores, each core must work through several computations sequentially.

Efficient threading implementations can have a large impact on the final design. We designed a custom threading wrapper to enable the reuse of persistent threads leading to as much as a 1.9x reduction in end-to-end computation time as compared to continuously launching and joining threads.

We used the Eigen library [48] to vectorize many linear algebra operations, taking advantage of some limited fine-grained parallelism by leveraging the CPU's modest-width vector operations.

The current fastest CPU forward dynamics package, RobCoGen, exploits structured sparsity to increase performance [6], [8]. Building on this approach, we wrote custom functions to exploit the structured sparsity in the dynamics gradient using explicit loop unrolling and zero-skipping.

While this approach creates irregular data access patterns, this is not a problem for the CPU, as the values are small enough to fit in the CPU's cache hierarchy. The sequential dependencies and small working sets are also handled well by the CPU's fast clock rate, large pipeline, and sophisticated memory hierarchy. With all operations occurring on the CPU, there is no I/O overhead or partitioning of the algorithm across different hardware platforms.

B. GPU Design

As compared to a CPU, a GPU is a much larger set of very simple processors, optimized specifically for parallel computations with identical instructions over large working sets of data (e.g., large matrix-matrix multiplication). For maximal performance, the GPU requires groups of threads within each thread block to compute the same operation on memory accessed via regular patterns. As such, it is highly optimized for some types of native parallelism present in our application, but is inefficient on others.

As noted in Table I, the GPU can use blocks of threads to efficiently take advantage of large-scale, coarse-grained parallelism across many independent dynamics gradient computations. However, the fine-grained parallelism within each computation can be harder to exploit effectively on a GPU as there are many different low-level operations that can occur

Algorithm 4 ∇RNEA-GPU$(\dot{q}, v, a, f, X, S, I) \rightarrow \partial c/\partial u$

```
1:  for link $i = 1 : n$ in parallel do
2:      $\alpha_i = {}^iX_{\lambda_i} v_{\lambda_i} \quad \beta_i = {}^iX_{\lambda_i} a_{\lambda_i} \quad \gamma_i = I_i v_i$
3:      $\alpha_i = \alpha_i \times S_i \quad \beta_i = \beta_i \times S_i \quad \delta_i = v_i \times S_i$
        $\zeta_i = f_i \times S_i \quad \eta_i = v_i \times^*$
4:      $\zeta_i = -{}^iX_{\lambda_i}^T \zeta_i \quad \eta_i = \eta_i I_i$
5:  for link $i = 1 : n$ do
6:      $\frac{\partial v_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} \alpha_i & u \equiv q \\ S_i & u \equiv \dot{q} \end{cases}$
7:  for link $i = 1 : n$ in parallel do
8:      $\mu_i = \frac{\partial v_i}{\partial u} \times^* \quad \rho_i = \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} \beta_i \\ \delta_i \end{cases}$
9:  for link $i = 1 : n$ do
10:     $\frac{\partial a_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \rho_i$
11: for link $i = 1 : n$ in parallel do
12:     $\frac{\partial f_i}{\partial u} = I_i \frac{\partial a_i}{\partial u} + \mu_i \gamma_i + \eta_i \frac{\partial v_i}{\partial u}$
13: for link $i = n : 1$ do
14:     $\frac{\partial f_{\lambda_i}}{\partial u} \mathrel{+}= {}^iX_{\lambda_i}^T \frac{\partial f_i}{\partial u} + \zeta_i$
15: for link $i = n : 1$ in parallel do
16:     $\frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u}$
```

simultaneously. Without careful refactoring, the algorithm requires frequent synchronization points and exhibits low overall thread occupancy. This is the major performance limitation of current state-of-the-art GPU implementations [11].

To address this bottleneck, the core of our optimized GPU implementation is a refactored version of Algorithm 2, shown in Algorithm 4. In this refactoring, we moved computations with (nearly) identical operations into the same parallel step to better fit the hardware's computational model. For example, Line 3 computes a series of cross products. While some of these operations could be executed in parallel with the matrix-vector multiplications in Line 2, this would lead to thread divergence and was thus avoided.

Out-of-order computations and irregular data access patterns are also inefficient to compute on a GPU. It is important to avoid these, even at the expense of creating large numbers of temporary variables. Therefore, in our GPU implementation, unlike on the CPU and FPGA, we did not exploit the sparsity in the X, $\times$, and $\times^*$ matrices. Instead, we used standard threaded matrix multiplication for the X matrices. For the $\times$ and $\times^*$ matrices we used a two-step process to first create temporary matrices to capture the re-ordering, and then used standard threaded matrix multiplication to compute the final values. For example, the temporary value $\mu_i = \partial v_i/\partial u \times^*$, computed on Line 8, takes as an input n values of $\partial v_i/\partial u$, each having $2i$ columns $\forall i \in (1, n)$, and produces a matrix $\in \mathbb{R}^{6\times 6}$ for each column. The most efficient GPU solution is to compute each matrix in parallel with a loop over the entries to avoid thread divergence, even though each entry in each matrix is naturally parallel.

The sequential dependencies within each dynamics gradient computation are challenging for the GPU, as it must introduce synchronization points at every data dependency. Additionally, GPUs typically run at about half the clock speed of CPUs (e.g., 1.7GHz versus 3.6GHz for the GPU and CPU evaluated in Section V), further hindering their performance on sequential code. Therefore, in our implementation we re-ordered the computations to minimize the amount of work done in serial loops, the main driver of sequential dependencies, at the expense of generating even more temporary values. For example, we precomputed α in Line 2 of the initial parallel computation on Lines 1-4 to reduce the amount of work done in the later serial loop in Lines 5-6. Due to the small working set size, even after generating these additional temporary values we were able to fit everything in the GPU's shared memory (cache), minimizing latency penalties. Even so, due to the highly serial nature of the algorithm, many serial operations and synchronization points still existed in our final implementation.

Finally, because data needs to be transferred between a coprocessor board and a host CPU, I/O is a serious constraint for GPUs and FPGAs. We implemented both *split* and *fused* GPU kernels to analyze the I/O and compute trade-offs induced by different problem partitionings of Algorithm 3 between the CPU and the coprocessor. In the *split* kernel we only computed the most parallel and compute-intensive section of the algorithm on the accelerator, the $\partial c'/\partial u$ computation. In the initial *fused* kernel we minimized I/O by computing both the v', a', f' and $\partial c'/\partial u$ computations on the accelerator. We also designed a *completely-fused* kernel computing all of Algorithm 3. This lead to an increase in I/O as compared to the fused kernel, but reduced both the number of synchronization points with the host CPU, as well as the total amount of computation done on the CPU. To further reduce the total I/O in our GPU and FPGA implementations, we computed the transformation matrices X on-board, even in the *split* kernel where it was already computed on the CPU.

On the GPU, we used the NVIDIA CUDA [49] library's built-in functions for transferring data to and from the GPU over PCIe Gen3. As suggested by GPU manufacturers, we copied all of the needed data over to the GPU memory once, let it run, and then copied all of the results back. Finally, to better leverage the PCIe data bus, we ensured that all values were stored as a single contiguous block of memory.

C. FPGA Design

FPGAs have reconfigurable logic blocks and programmable interconnects that allow them to implement custom hardware functional units, data flows, and processing pipelines. FPGA designs often also use *fixed-point* arithmetic to perform math faster while using less energy and area per computation as compared to *floating-point* arithmetic. The trade-off is that the dynamic range and precision of fixed-point numbers are reduced. However, for many applications, this reduction still produces high quality end-to-end solutions (e.g., quantization for neural networks) [50], [51], [52].

To leverage this feature, we first validated that a fixed-point implementation of the dynamics gradient kernel would

Algorithm 5 ∇RNEA-FPGA$(\dot{q}, v, a, f, X, S, I) \to \partial c/\partial u$

```
1: for link $i = 1 : N$ do
2:     $\alpha_i = I_i v_i$
       $\frac{\partial v_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} {}^iX_{\lambda_i} v_{\lambda_i} \times S_i & u \equiv q \\ S_i & u \equiv \dot{q} \end{cases}$
3:     $\beta_i = \frac{\partial v_i}{\partial u} \times^* \alpha_i \qquad \gamma_i = I_i \frac{\partial v_i}{\partial u}$
       $\frac{\partial a_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} {}^iX_{\lambda_i} a_{\lambda_i} \times S_i \\ v_i \times S_i \end{cases}$
4:     $\frac{\partial f_i}{\partial u} = I_i \frac{\partial a_i}{\partial u} + \beta_i + v_i \times^* \gamma_i$
5: for link $i = N : 1$ do
6:     $\frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u} \quad \delta_i = {}^iX_{\lambda_i}^T \frac{\partial f_i}{\partial u} \quad \zeta_i = {}^iX_{\lambda_i}^T (S_i \times^* f_i)$
7:     $\frac{\partial f_{\lambda_i}}{\partial u}$ += $\delta_i + \zeta_i$
```

provide sufficient end-to-end accuracy. We solved a trajectory optimization problem from previous work [11] that involved moving a Kuka LBR IIWA-14 manipulator [53] from a start to goal state, sweeping the numerical data types used in the gradient computation. A wide range of fixed-point data types were able to solve the problem successfully.[2] Fixed-point conversion on the CPU incurred considerable overhead, so we instead used dedicated Xilinx IP cores on the FPGA, reducing overhead latency by as much as 3.6x.

We designed custom functional units and datapaths to exploit fine-grained parallelism. As with our GPU design, we refactored the algorithm to enable the FPGA to better exploit these structures, as shown in Algorithm 5. For example, we computed each column j of $\partial c'/\partial u_j$ in parallel datapaths routed through the hardware. Within each $\partial c'/\partial u_j$ column we exploited parallelism between brief linear algebra operations, as well as within those operations, by instantiating many multiplier and adder units in parallel. We were also able to almost entirely overlap the computation of v', a', f' with the computation of $\partial c'/\partial u$ by using additional parallel datapaths.

As Table I indicates, unlike the GPU, the FPGA can fully exploit the fine-grained parallelism between different linear algebra operations as operations are not performed by threads, but instead are performed natively in independent, heterogeneous, parallel hardware circuits. For example, in Lines 3 and 6, cross products and matrix-vector products are executed in parallel using different custom hardware circuits.

Coarse-grained parallelism, however, was limited in our particular FPGA design because we made a design choice to prioritize reducing latency. This resulted in heavy utilization of limited digital signal processing (DSP) resources, even on a large FPGA platform: 77.5% of the 6840 DSP blocks for our final design used in Section V. This is despite careful re-use or *folding* of multiplier resources, which, e.g., reduced the resource requirements of the forward passes of Algorithms 1 and 2 by about 7x in our design. Because of this high utilization we were only able to compute one dynamics gradient calculation at a time on the FPGA, and thus were unable to exploit coarse-grained parallelism between gradient computations. Leveraging this paralleism would have required either substantially more DSP resources (perhaps offered on future FPGA platforms or though an ASIC design), or further aggressive pipelining and folding of the circuits in our current design, at the expense of latency.

Using the reconfigurable connections of the FPGA, we were also able to exploit the sparse structure of the X, $\times$, and $\times^*$ matrices by pruning operations from trees of multipliers and adders, further decreasing latency. For example, when multiplying variables by the X matrices, we were able to reduce some of the $\mathbb{R}^6$ dot product operations from a 4-level tree with 6 multiplications and 5 additions to a 3-level tree with 3 multiplications and 2 additions. Implementing this sparsity in hardware datapaths not only decreases latency, but also helps our implementation avoid irregular data access patterns by encoding them directly in the circuit routing.

By creating processing units tailored to the dataflow of the algorithm, our implementation streamlines sequential chains of dependencies between links by creating dedicated hardware to iterate over those loops with minimal overhead.

All of these reductions in overhead are crucial to obtaining a performance advantage on an FPGA, as designs on the reconfigurable circuit fabric typically have substantially slower clock speeds than the highly-optimized circuit routing within CPUs and GPUs (e.g., 55.6MHz for our design versus 1.7GHz and 3.6GHz for the GPU and CPU in Section V).

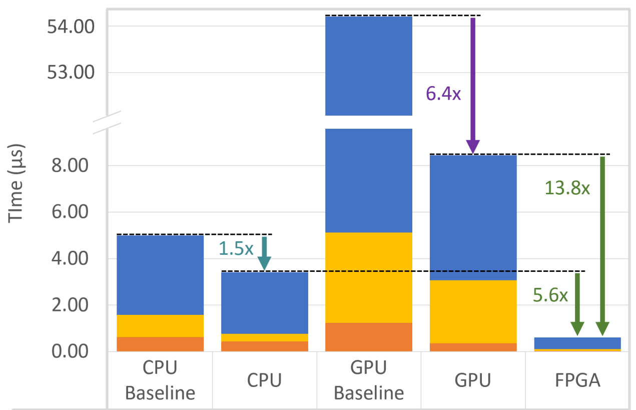
We used a framework called Connectal [54] to implement the I/O transfer between the FPGA and a host CPU. Connectal's support for the FPGA platform we used is currently restricted to PCIe Gen1, limiting our I/O bandwidth. Therefore, a major choice in our design was how to partition the algorithm between hardware and software to balance the total amount of I/O overhead with our other hardware constraints (e.g., DSPs). Based on the results of our problem partitioning experiments done on the GPU in Section V-C, we chose to use a *completely-fused* kernel, implementing all of Algorithm 3 on the FPGA. By using this partitioning and by pipelining the I/O with the dynamics gradient computations, we were able to achieve I/O overhead comparable to that of the GPU.

Note that in general, the FPGA development process is considered more challenging than the process for CPUs and GPUs. Since it is difficult, time-intensive, and error-prone, it is advisable to perform thorough design analysis in advance of implementation, to minimize the number of design revisions when possible.

V. Evaluation

We evaluated two timing metrics to understand the performance of our designs: the latency of a single computation of the algorithm, and the end-to-end latency (including I/O and other overhead) to execute a set of N computations. We compared our optimized implementations on a CPU, GPU, and FPGA against baseline implementations, against each other, and against different problem partitionings of our own implementations. Source code accompanying this evaluation can be found at http://bit.ly/fast-rbd-grad.

[2]We used a 32-bit integer with 16 bits before and after the decimal point. This is safely in the valid datatype range (as low as 12 bits before and 5 bits after), and integrates easily with the CPU's 32-bit floating-point math.



	CPU Baseline	CPU	GPU Baseline	GPU	FPGA
Total	4.99	3.42	54.41	8.44	0.61
■ dc'/du	3.41	2.65	49.29	5.37	0.50
■ v', a', f'	0.96	0.34	3.89	2.71	0.07
■ $d\ddot{q}/du$	0.62	0.44	1.23	0.36	0.04

Fig. 1. Latency of one computation of the gradient of rigid body dynamics for the Kuka manipulator across hardware platforms.

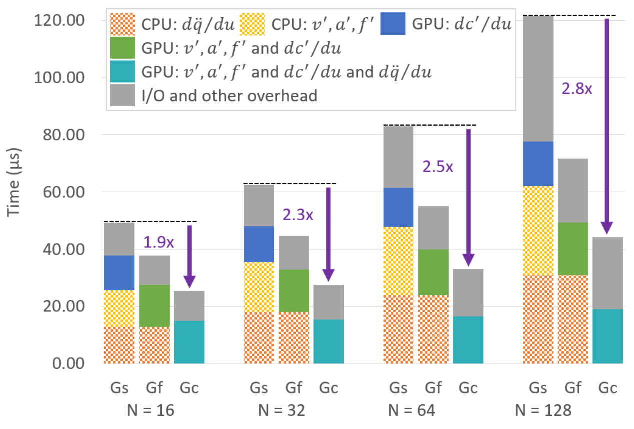


Fig. 2. Runtime of $N = 16$, 32, 64, and 128 computations of our accelerated implementations of the dynamics gradient kernel for the Kuka manipulator using different problem partitionings between the CPU and [G]PU coprocessor: the [s]plit, [f]used, and [c]ompletely-fused kernels as described in Section IV-B.

A. Methodology

Our designs were implemented using a Kuka LBR IIWA-14 manipulator as our robot model. All CPU results were collected on a 3.6GHz quad-core Intel Core i7-7700 CPU running Ubuntu 18.04 and CUDA 11. Our GPU and FPGA results used a 1.7GHz NVIDIA GeForce GTX 2080 GPU and a Virtex UltraScale+ VCU-118 FPGA. The FPGA design was synthesized at a clock speed of 55.6MHz. For clean timing measurements on the CPU, we disabled TurboBoost and fixed the clock frequency to the maximum. Code was compiled with `Clang 10` and `g++7.4`, and time was measured with the Linux system call `clock_gettime()`, using `CLOCK_MONOTONIC` as the source. For single computation measurements on the CPU and GPU, we took the average of one million trials. For the FPGA, we extracted the timing from cycle counts. For end-to-end results on all platforms, we took the median of one hundred thousand trials. These end-to-end latency results include I/O and any other overhead needed to execute $N = 16$, 32, 64, and 128 computations because, as mentioned earlier, many nonlinear MPC robotics applications use tens to hundreds of dynamics gradient computations.

B. Single Computation Latency

The latency for a single computation is shown in Figure 1. These times represent the computation performed in a single run of the baseline or optimized implementation on its target hardware platform, excluding overheads. The three colors represent the three steps of Algorithm 3.

By leveraging the optimizations in Section IV-A, our CPU implementation is 1.5x faster than the state of the art [7].

The GPU implementations struggled in this single computation test as GPUs derive their benefit from throughput offered by extreme parallelism. However, for our optimized implementation, while the GPU v', a', f' latency is 8.0x slower than the CPU, the $\partial c'/\partial u$ latency is only 2.0x slower. This improved scaling is the result of the re-factoring done in Section IV-B to expose more parallelism, reducing the computations done in serial loops. Leveraging these optimizations, our GPU implementation is 6.4x faster than the existing state-of-the-art [11], driven by a 9.2x speedup in the $\partial c'/\partial u$ step.

The FPGA implementation is significantly faster than either the CPU or GPU, with speedups of 5.6x and 13.8x over our optimized CPU and GPU implementations, respectively.[3] This is because the dedicated datapaths in the FPGA implementation handle sequential operations well, and the fine-grained parallelism allows for the execution of many densely-packed parallel operations (Section IV-C). Parallelism exploited in the design process also overlapped most of the computation of v', a', f' with the computation of $\partial c'/\partial u$. Therefore, the v', a', f' times shown represent only the small additional latency incurred by this design choice.

All three of our designs, as described in Section IV, leveraged the interactions between the algorithmic features described in Section III-B to achieve faster than state-of-the-art performance.

C. Problem Partitioning

Figure 2 explores the impact of changing the problem partitioning between the CPU and the coprocesser on the end-to-end latency of our accelerated implementations. We compare (from left to right within each group) the "[s]plit", "[f]used", and "[c]ompletely fused" accelerated [G]PU kernels as described in Section IV-B.

Moving more of the computation onto the GPU increased the computational latency. However, since the $\partial c'/\partial u$ computation is by far the most expensive computation, and the GPU can easily compute many computations in parallel, the increase is quite small. This can be seen in the slight increase from the blue to green to teal bar in each group.

At the same time, the fused kernels have reduced I/O overhead, which is more important as N increases. Furthermore, removing the high-level algorithmic synchronization points

[3] No baseline implementation exists in the literature for the FPGA.

between the CPU and GPU, and the corresponding batched CPU computations, shown by the checkered orange and yellow bars, greatly reduced overall latency.

By changing our problem partitioning and moving more computation onto the GPU (moving from the "Gs" to "Gc" kernels), we improved the end-to-end latency of the optimized GPU designs substantially: by 1.9x for $N = 16$, up to 2.8x for $N = 128$.

D. End-to-End CPU, GPU, and FPGA Comparisons

Figure 3 compares our most-optimized implementations across all hardware platforms: the [C]PU implementation and the [G]PU and [F]PGA [c]ompletely-fused implementations.

Within each group, the first bar is the CPU design, where the entire algorithm is computed in 4 persistent threads to make full use of the 4 processor cores without overtaxing them with additional threads.

The second bar is the "Gc" kernel which was able to provide a 1.2x to 3.0x performance improvement over the CPU design for $N = 16, 128$ respectively. The GPU performed better as the number of computations increased and was the top performer at higher numbers of computations ($N = 64, 128$), where its support for massive amounts of both coarse-grained and fine-grained parallelism prevailed.

The third bar is the FPGA "Fc" kernel which was also able to outperform the CPU in all cases, ranging from a 1.9x to 1.6x improvement. The FPGA was the top performer at lower numbers of computations ($N = 16, 32$) due to our minimal-latency design, but could not exploit coarse-grained parallelism at higher numbers of computations as it could only process one dynamics gradient calculation at a time due to resource limitations (see Section IV-C).

Note that the best results were achieved by exploiting opposing design trade-offs to map opportunities for parallelism onto each hardware platform (e.g., exploiting high throughput on the GPU versus low latency on the FPGA). These results illustrate how successful acceleration on different hardware platforms depends on both the features of the algorithm, and the application context in which the kernel will be run (Table I). It is critical to design software with the strengths and weaknesses of the underlying hardware platform in mind.

VI. Conclusion and Future Work

In this work, we accelerated rigid body dynamics gradients through the development of hardware-optimized implementations targeting a CPU, GPU, and FPGA. We showed that using these hardware-accelerated implementations can lead to performance improvements both in the runtime of individual kernels and in overall end-to-end timing of multiple computations. Our best GPU and FPGA performance results showed end-to-end improvements of as much as 3.0x over our faster than state-of-the-art CPU design.

Promising directions for future work include the development of a full robot dynamics package with code generation for the CPU, GPU, and FPGA, to enable researchers to easily target and utilize these different hardware platforms with their own robot models. Alternate formulations of rigid body dynamics can also be explored, which may expose additional parallelism [55], [56], [57], [58]. Implementing a custom accelerator chip for our FPGA design would further reduce the latency of a single computation and allow multiple computations to occur in parallel, increasing throughput. Finally, integrating these accelerated dynamics gradient implementations into existing MPC software frameworks [59], [60], [11] would increase their ease-of-use and applicability to other robot platforms.

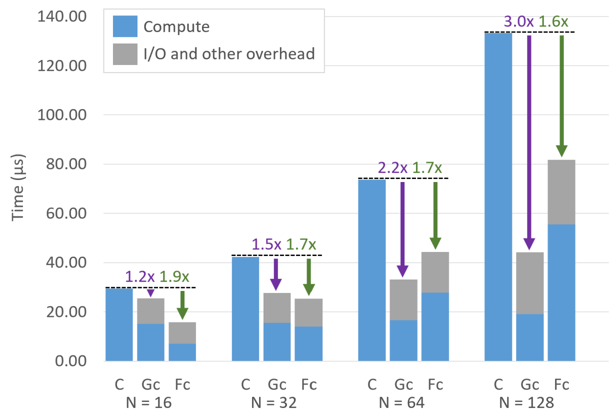


Fig. 3. Runtime of $N = 16$, 32, 64, and 128 computations of the accelerated dynamics gradient kernels for the Kuka manipulator for the [C]PU, [G]pu, and [F]PGA using the [c]ompletely-fused kernel.

References

[1] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer.

[2] F. Farshidian, *et al.*, "Real-time motion planning of legged robots: A model predictive control approach," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics*.

[3] M. Giftthaler, *et al.*, "A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control." http://arxiv.org/abs/1711.11006

[4] B. Plancher and S. Kuindersma, "Realtime model predictive control using parallel ddp on a gpu," in *Toward Online Optimal Control of Dynamic Robots Workshop at the 2019 International Conference on Robotics and Automation (ICRA)*, Montreal, Canada, May. 2019.

[5] S. Kleff, *et al.*, "High-frequency nonlinear model predictive control of a manipulator," *Preprint*, 2020.

[6] M. Frigerio, *et al.*, "RobCoGen: A code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages."

[7] J. Carpentier and N. Mansard, "Analytical derivatives of rigid body dynamics algorithms," in *Robotics: Science and Systems*, 2018.

[8] S. Neuman, *et al.*, "Benchmarking and workload analysis of robot dynamics algorithms," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[9] J. Koenemann, *et al.*, "Whole-body Model-Predictive Control applied to the HRP-2 Humanoid," in *Proceedings of the IEEERAS Conference on Intelligent Robots*.

[10] M. Neunert, *et al.*, "Fast nonlinear Model Predictive Control for unified trajectory optimization and tracking," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*.

[11] B. Plancher and S. Kuindersma, "A Performance Analysis of Parallel Differential Dynamic Programming on a GPU," in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*.

[12] H. Esmaeilzadeh, *et al.*, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. ACM.

[13] G. Venkatesh, *et al.*, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM.

[14] C. Park, J. Pan, and D. Manocha, "Real-time optimization-based planning in dynamic environments using GPUs," in *2013 IEEE International Conference on Robotics and Automation*.

[15] S. Heinrich, A. Zoufahl, and R. Rojas, "Real-time trajectory optimization under motion uncertainty using a GPU," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[16] A. Wittig, V. Wase, and D. Izzo, "On the Use of Gpus for Massively Parallel Optimization of Low-thrust Trajectories."

[17] B. Ichter, *et al.*, "Real-time stochastic kinodynamic motion planning via multiobjective search on GPUs," in *2017 IEEE International Conference on Robotics and Automation*.

[18] G. Williams, A. Aldrich, and E. A. Theodorou, "Model predictive path integral control: From theory to parallel computation," *Journal of Guidance, Control, and Dynamics*, 2017.

[19] S. Ohyama and H. Date, "Parallelized nonlinear model predictive control on gpu," in *2017 11th Asian Control Conference (ASCC)*. IEEE, 2017.

[20] P. Hyatt, C. S. Williams, and M. D. Killpack, "Parameterized and gpu-parallelized real-time model predictive control for high degree of freedom robots," *arXiv preprint arXiv:2001.04931*, 2020.

[21] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE, 2006.

[22] S. Murray, *et al.*, "Robot Motion Planning on a Chip," in *Robotics: Science and Systems*.

[23] ——, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] X. Shi, *et al.*, "Hero: Accelerating autonomous robotic tasks with fpga," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018.

[25] S. Murray, *et al.*, "A programmable architecture for robot motion planning acceleration," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2019.

[26] R. Li, *et al.*, "Fpga-based design and implementation of real-time robot motion planning," in *2019 9th International Conference on Information Science and Technology (ICIST)*. IEEE, 2019.

[27] Z. Wan, *et al.*, "A survey of fpga-based robotic computing," *arXiv preprint arXiv:2009.06034*, 2020.

[28] D. Kouzoupis, *et al.*, "A Block Based ALADIN Scheme for Highly Parallelizable Direct Optimal Control," in *Proceedings of the American Control Conference*.

[29] T. Antony and M. J. Grant, "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures."

[30] Z. Pan, B. Ren, and D. Manocha, "Gpu-based contact-aware trajectory optimization using a smooth force model," in *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '19. New York, NY, USA: ACM, 2019.

[31] F. Xu, *et al.*, "Fast nonlinear model predictive control on fpga using particle swarm optimization," *IEEE Transactions on Industrial Electronics*, 2015.

[32] J. Sacks, *et al.*, "Robox: an end-to-end solution to accelerate autonomous control in robotics," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018.

[33] I. McInerney, G. A. Constantinides, and E. C. Kerrigan, "A survey of the implementation of linear model predictive control on fpgas," *IFAC-PapersOnLine*, 2018.

[34] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, 2009.

[35] D. Michéa and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards," *Geophysical Journal International*, 2010.

[36] M. Neunert, *et al.*, "Fast Derivatives of Rigid Body Dynamics for Control, Optimization and Estimation," in *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*.

[37] J. Bender, M. Müller, and M. Macklin, "A survey on position based dynamics, 2017," in *Proceedings of the European Association for Computer Graphics: Tutorials*, 2017.

[38] F. de Avila Belbute-Peres, *et al.*, "End-to-end differentiable physics for learning and control," in *Advances in neural information processing systems*, 2018.

[39] J. Degrave, *et al.*, "A differentiable physics engine for deep learning in robotics," *Frontiers in neurorobotics*, 2019.

[40] Y. Hu, *et al.*, "Chainqueen: A real-time differentiable physical simulator for soft robotics," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019.

[41] J. Austin, *et al.*, "Titan: A parallel asynchronous library for multi-agent and soft-body robotics using nvidia cuda," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020.

[42] Y. Hu, *et al.*, "Difftaichi: Differentiable programming for physical simulation," in *International Conference on Learning Representations (ICLR)*, 2020.

[43] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015.

[44] J. Di Carlo, *et al.*, "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018.

[45] T. Erez, *et al.*, "An integrated system for real-time model predictive control of humanoid robots," in *2013 13th IEEE-RAS International Conference on Humanoid Robots*.

[46] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[47] M. Neunert, *et al.*, "Trajectory Optimization Through Contacts and Automatic Gait Discovery for Quadrupeds."

[48] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[49] NVIDIA, *NVIDIA CUDA C Programming Guide*, version 9.1 ed. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[50] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *The International Conference on Learning Representations (ICLR)*, 10 2016.

[51] B. Reagen, *et al.*, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[52] A. Finnerty and H. Ratigner, "Reduce power and cost by converting from floating point to fixed point," 2017.

[53] KUKA AG, "Lbr iiwa — kuka ag," Accessed in 2020, available: kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa. https://www.kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa

[54] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[55] R. Featherstone, "A divide-and-conquer articulated-body algorithm for parallel o (log (n)) calculation of rigid-body dynamics. part 1: Basic algorithm," *The International Journal of Robotics Research*, 1999.

[56] K. Yamane and Y. Nakamura, "Comparative Study on Serial and Parallel Forward Dynamics Algorithms for Kinematic Chains."

[57] Y. Yang, Y. Wu, and J. Pan, "Parallel Dynamics Computation Using Prefix Sum Operations."

[58] J. Brüdigam and Z. Manchester, "Linear-time variational integrators in maximal coordinates," 2020.

[59] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems." http://drake.mit.edu

[60] M. Giftthaler, *et al.*, "The control toolbox — an open-source c++ library for robotics, optimal and model predictive control," *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2018.