Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology

Sabrina M. Neuman
sneuman@seas.harvard.edu
Harvard University
Cambridge, MA, USA

Brian Plancher
brian_plancher@g.harvard.edu
Harvard University
Cambridge, MA, USA

Thomas Bourgeat
bthom@csail.mit.edu
MIT
Cambridge, MA, USA

Thierry Tambe
ttambe@g.harvard.edu
Harvard University
Cambridge, MA, USA

Srinivas Devadas
devadas@mit.edu
MIT
Cambridge, MA, USA

Vijay Janapa Reddi
vj@eecs.harvard.edu
Harvard University
Cambridge, MA, USA

ABSTRACT

Robotics applications have hard time constraints and heavy computational burdens that can greatly benefit from domain-specific hardware accelerators. For the latency-critical problem of robot motion planning and control, there exists a performance gap of at least an order of magnitude between joint actuator response rates and state-of-the-art software solutions. Hardware acceleration can close this gap, but it is essential to define automated hardware design flows to keep the design process agile as applications and robot platforms evolve. To address this challenge, we introduce robomorphic computing: a methodology to transform robot morphology into a customized hardware accelerator morphology. We (i) present this design methodology, using robot topology and structure to exploit parallelism and matrix sparsity patterns in accelerator hardware; (ii) use the methodology to generate a parameterized accelerator design for the gradient of rigid body dynamics, a key kernel in motion planning; (iii) evaluate FPGA and synthesized ASIC implementations of this accelerator for an industrial manipulator robot; and (iv) describe how the design can be automatically customized for other robot models. Our FPGA accelerator achieves speedups of 8× and 86× over CPU and GPU when executing a single dynamics gradient computation. It maintains speedups of 1.9× to 2.9× over CPU and GPU, including computation and I/O round-trip latency, when deployed as a coprocessor to a host CPU for processing multiple dynamics gradient computations. ASIC synthesis indicates an additional 7.2× speedup for single computation latency. We describe how this principled approach generalizes to more complex robot platforms, such as quadrupeds and humanoids, as well as to other computational kernels in robotics, outlining a path forward for future robomorphic computing accelerators.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446746

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Robotics**.

KEYWORDS

robotics, hardware accelerators, dynamics, motion planning

ACM Reference Format:
Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. 2021. Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446746

1 INTRODUCTION

Complex robots such as manipulators, quadrupeds, and humanoids that can safely interact with people in dynamic, unstructured, and unpredictable environments are a promising solution to address critical societal challenges, from elder care [24, 53] to the health and safety of humans in hazardous environments [34, 60]. A major obstacle to the deployment of complex robots is the need for high-performance computing in a portable form factor. Robot perception, localization, and motion planning applications must be run online at real-time rates and under strict power budgets [12, 26, 47, 55].

Domain-specific hardware acceleration is a emerging solution to this problem, building on the success of accelerators for other domains such as neural networks [7, 23, 49]. However, while accelerators have improved the power and performance of robot perception and localization [7, 49, 56], relatively little work has been done for motion planning [33, 38].

Motion planning algorithms calculate a valid motion path from a robot's initial position to a goal state. Online motion planning approaches [41, 57] rely heavily on latency-critical calculation of functions describing the underlying physics of the robot, e.g., rigid body dynamics and its gradient [5, 14, 18]. There exist several software implementations that are sufficient for use in traditional control approaches [6, 16, 22, 27, 36, 39], but emerging techniques such as whole-body nonlinear model predictive control (MPC) [9, 26] reveal a performance gap of at least an order of magnitude:

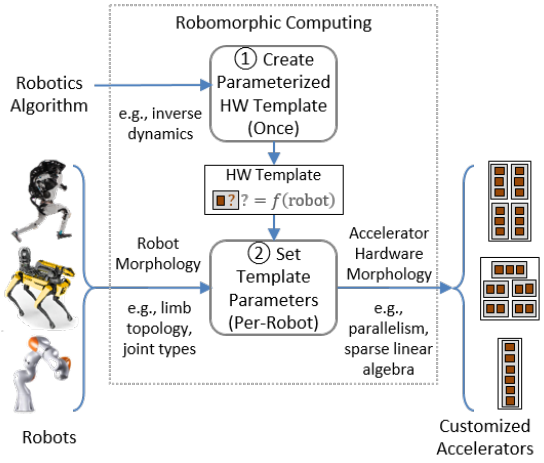

Figure 1: Overview of robomorphic computing, a design methodology to transform robot morphology into customized accelerator hardware morphology by exploiting robot features such as limb topology and joint type. This methodology can be applied to a wide variety of complex robots. Pictured are the Atlas [3], Spot [4], and LBR iiwa [31] robots, as examples.

robot joint actuators respond at kHz rates, but these promising approaches are limited to 100s of Hz by state-of-the-art software [12, 47]. This gap exists despite software code generation to optimize functions for a particular robot model [36].

Hardware acceleration can shrink the motion planning performance gap, but traditional accelerator design can be tedious, iterative, and costly. The paramount challenge is to provide formalized methodologies for systematic hardware synthesis that can generalize across different robots and algorithms, keeping the design process agile as applications evolve [21].

We address this challenge with *robomorphic computing*: a methodology to transform robot morphology into customized accelerator hardware morphology. Our systematic design methodology introduces robotics software optimizations to the hardware domain by *exploiting high-level information about a robot model to parameterize features of the accelerator*, e.g., using robot limb topology to determine processing element parallelism (Figure 1). In the robomorphic computing design flow: (1) a parameterized hardware template is created for a robotics algorithm once; then, (2) for each robot, the template parameters are set according to the robot morphology to *create an accelerator customized to that robot model.*

We demonstrate the use of our robomorphic computing methodology to design what we believe to be the first domain-specific hardware architecture for the gradient of rigid body dynamics. Previous workload analysis has shown that the dynamics gradient kernel, which is the target of the accelerator artifact presented in this work, can take 30% to 90% of the overall runtime of state-of-the-art nonlinear MPC motion planning and control algorithms [5, 41, 46, 47]. Using robomorphic computing, we identify opportunities in the dynamics gradient algorithm to parameterize parallelism and matrix sparsity patterns based on robot topology and joint types. Then, for a specific robot model we systematically set these parameters to exploit parallelism in hardware datapaths and sparsity in linear algebra functional units, resulting in a robot-customized accelerator.

We implement our dynamics gradient accelerator design on an FPGA for an industrial manipulator. Our FPGA accelerator, which implements the dynamics gradient kernel in hardware, achieves speedups of 8× and 86× over the computation latency of state-of-the-art CPU and GPU implementations for the task of computing a single dynamics gradient calculation. We also integrate the FPGA accelerator as a coprocessor system with I/O connections to a host CPU, as it would be deployed for an off-the-shelf solution today. Our FPGA accelerator demonstrates speedups of 1.9× to 2.9× over state-of-the-art CPU and GPU implementations when deployed as a coprocessor to a host CPU (measurements including computation and I/O round-trip latency) for the task of computing multiple dynamics gradients calculations, which is the typical use case in real motion planning and control applications. We also synthesize an ASIC implementation to evaluate the performance and power opportunities of a system on chip. ASIC synthesis using a 12 nm technology node indicates an additional 7.2× speedup for a single dynamics gradient calculation over our FPGA implementation.

In summary, the key contributions of this work include:

- Robomorphic computing, a new general methodology for the co-design of hardware accelerator architectures based on the high-level physical topology of a robot;
- Design of the first domain-specific accelerator for the gradient of rigid body dynamics, implemented for an industrial manipulator on an FPGA and a synthesized ASIC; and
- Discussion of how our design methodology generalizes to more complex robot platforms, e.g., quadrupeds and humanoids, and other computational kernels in robotics.

Our accelerator for the gradient of rigid body dynamics is a critical step towards enabling real-time, online motion planning and control for complex robots. More importantly, the general methodology that guided its design provides a roadmap for future accelerators for robotics applications.

2 BACKGROUND

In robotics, the main processing pipeline can be broken down into three fundamental stages: (1) perception, (2) mapping and localization, and (3) motion planning and control (see Figure 2). These stages can be run sequentially as a pipeline or in parallel loops leveraging asynchronous data transfers during runtime. During perception, a robot gathers information from its sensors and processes that data into usable semantic information (e.g., depth, object classifications). Next, the robot uses that labeled data to construct a map of its surrounding environment and estimates its location within that map. Finally, the robot plans and executes a safe obstacle-free motion trajectory through the space. If this is done online in real time, it allows the robot to adapt to unpredictable environments.

2.1 Robot Morphology

Traditional robots, including most commercial manipulator arms, quadrupeds, and humanoids, can be modeled as a topology of rigid links connected by joints (see Figure 3). The morphology of the robot can be disassembled into three principal components: L limbs,

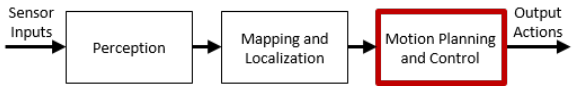

Figure 2: The processing pipeline for robotics. This work demonstrates applying robomorphic computing to a key kernel in the motion planning and control stage.

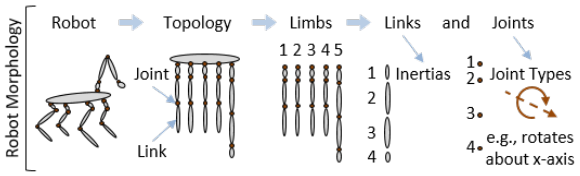

Figure 3: Robot morphology can be modeled as a topology of limbs, rigid links, and joints. Robomorphic computing exploits parallelism and matrix sparsity patterns determined by this structure.

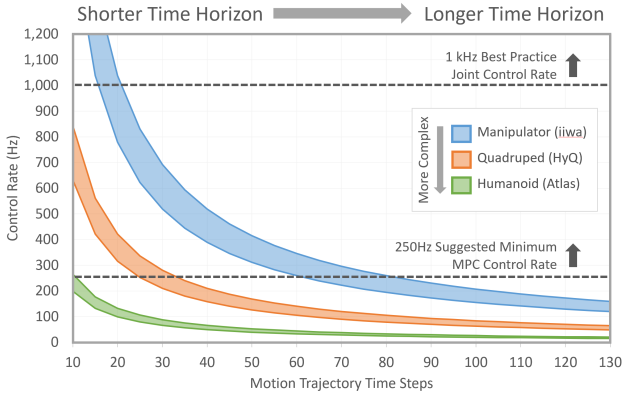

Figure 4: Estimated control rates for three robots using different trajectory lengths (based on state-of-the-art rigid body dynamics software implementations [5]), compared to ideal control rates required for online use [12]. We assume 10 iterations of the optimization loop. Current control rates fall short of the desired 250 Hz and 1 kHz targets for most trajectory lengths. This performance gap is worse for more complex robots, and grows with the number of optimization loop iterations.

a chain of N rigid links in a single limb, and the joints that connect those links.

The rigid links of the robot's limbs have inertial properties determined by their shape and distribution of mass. The joint type describes the movement constraints imposed upon the links connected by the joint. For example, a "revolute" joint about the x-axis means that the links connected by that joint can only rotate about the x-axis with respect to one another, and all other degrees of freedom are constrained.

A quadruped robot might have, e.g., $L = 4$ limbs, each with $N = 3$ links: a thigh, shin, and foot. The joint types might all be revolute about the x-axis. If the quadruped has an arm mounted on its torso, then it would have $L = 5$ limbs, where e.g., the arm has $N = 4$ links with z-axis revolute joints.

2.2 Algorithms Using Robot Morphology

Many critical robotics applications use information about robot morphology, including collision detection, localization, kinematics, and dynamics for soft and rigid robots [2, 11, 32, 45]. The design methodology we develop in this work can extend to all of these applications (see Section 7). In this paper, we focus on applying the methodology to develop an accelerator for one key kernel, the gradient of rigid body dynamics (see Section 3).

The dynamics of a robot with rigid limbs are described by its equations of motion, which relate the accelerations and forces on the joints of the robot. Common rigid body dynamics functions using the equations of motion include computing *forward dynamics*: joint accelerations out, given joint positions, velocities, and forces in; and *inverse dynamics*: joint forces out, given joint positions, velocities, and accelerations in. Several state-of-the-art software libraries [6, 16, 27, 36, 39], implement these functions using standard algorithms [14].

3 MOTIVATION

In this work, we use our design methodology to accelerate a computational bottleneck in motion planning and control, the dynamics gradient, which can take, e.g., 30% to 90% of the total runtime for emerging techniques such as whole-body nonlinear model predictive control (MPC) [5, 41, 46, 47].

Key Kernel: Forward Dynamics Gradient. A key kernel in many motion planning techniques is the first-order gradient of forward dynamics, which can be calculated in several ways [5, 17, 18, 26, 57]. The fastest of these methods uses analytical derivatives of the recursive Newton-Euler algorithm (RNEA) for inverse dynamics [5, 14, 35]. The result is then multiplied by a matrix of inertial quantities, to recover the gradient of forward dynamics (details in Section 5.1).

The fastest state-of-the-art software implementations of rigid body dynamics and its gradient [40] use templating and code generation [6, 36] to optimize functions for a particular robot model, incorporating robot morphology features into the code. In this work, we extend this software approach to the hardware domain with robomorphic computing. Using this methodology, we exploit computational opportunities in the dynamics gradient kernel to design a hardware accelerator.

Control Rate Performance. To motivate our work, we focus on a promising motion planning and control approach, whole-body nonlinear MPC [12, 26, 47]. Nonlinear MPC involves iteratively optimizing a candidate trajectory describing a robot's motion through space. This trajectory is made up of the robot's state at discrete time steps, looking some time horizon into the future. Longer time horizons increase resilience to disturbances. This online approach allows a robot to adapt to unpredictable environments by quickly recomputing safe trajectories in response to changes in the world.

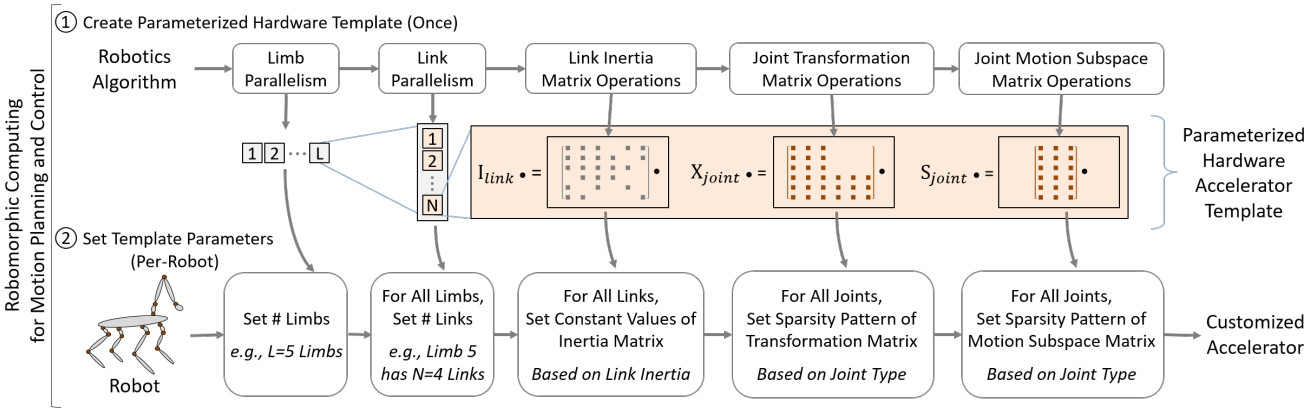

Figure 5: Using robomorphic computing for motion planning and control. First, a parameterized hardware template of the algorithm is created, exploiting high-level information about the robot model to identify opportunities for parallelism and sparse linear algebra. Then, for every target robot, the parameters of the hardware template are set based on the morphology of the robot to create a customized accelerator.

A significant performance gap exists for current state-of-the-art implementations of nonlinear MPC. Figure 4 shows estimated motion planning control rates for three robot models of increasing complexity (a manipulator, quadruped, and humanoid), for trajectories of different lengths. The three bands represent control rates for the three robots, extrapolated analytically from state-of-the-art compute times for the gradient of robot dynamics [5]. Control rates are calculated for 10 iterations of the optimization loop, allowing the trajectory to begin to converge towards an optimal solution.

Two horizontal thresholds are shown in Figure 4. The upper threshold is the 1 kHz control rate at which robot joint actuators are capable of responding. The lower 250 Hz threshold is a minimum suggested rate for nonlinear MPC to be run online [12]. The 250 Hz MPC planner would have to be part of a hierarchical system with faster-running low-level controllers interacting with the joint actuators. A performance gap of at least an order of magnitude has emerged: if nonlinear MPC could be run at kHz rates instead of 100s of Hz, the joint actuators could be controlled directly, maximizing robot reflexes and responsiveness.

With current software solutions, nonlinear MPC is unable to meet the desired 1 kHz or 250 Hz control rates in most cases. For example, the manipulator can only achieve the 1 kHz rate for short time horizons (under 25 time steps) and cannot achieve the minimum 250 Hz rate for more than about 80 time steps. The performance gap is worse for more complex robots such as the quadruped and humanoid. Additionally, for some applications, more than 10 iterations of the optimization loop may be required to achieve convergence [37], stretching the performance gap even further.

4 ROBOMORPHIC COMPUTING

Robomorphic computing is a hardware design methodology for robotics accelerators in which we *exploit high-level information about a robot model to parameterize features of the accelerator.* We introduce robomorphic computing to address performance challenges in motion planning and other robotics applications. There are two steps in the methodology:

(1) Create a hardware template for an algorithm once, parameterized by key components of robot morphology, e.g., limbs, links, and joints;
(2) For each robot, set the template parameters to customize the processors and functional units to produce a hardware accelerator tailored to that particular robot model.

This methodology enables the systematic identification of fundamental architectural design paradigms, such as *parallelism* and *sparse data structures*, in robotics algorithms. After these structures are identified once, a portable accelerator template can be used indefinitely to programmatically exploit these computational opportunities for each new robot platform. In this section, we briefly describe the robomorphic design flow in the context of motion planning applications (see Figure 5).

Step 1: Create Hardware Template. A parameterized hardware template needs to be created once per robotics algorithm, e.g., inverse dynamics (Section 2.2). We break down the algorithm to identify architectural structures that are parameterizable by robot morphology features.

We take parallelism from loops iterating over the robot's limbs and links, and map that to parallel processing elements in the hardware accelerator template. We also identify linear algebra operations on key sparse robot matrices, and map them to functional units whose constant values and sparsity patterns are parameterized by the robot links and joint types.

For example, using joint transformation matrix sparsity patterns, we can prune operations from multiplication-addition trees implementing matrix-vector multiplications. This optimization affects the number of multipliers and adders, the circuit routing connecting them, and the decision of which values are constants and which are updated by joint position inputs during runtime. We can also identify constant values in the link inertia matrices, allowing us to

Algorithm 1 ∇ Forward Dynamics *w.r.t.* $u = \{q, \dot{q}\}$ [5].

1: v, a, f = Inverse Dynamics$(q, \dot{q}, \ddot{q})$ ▷ Step 1
2: $\partial\tau/\partial u$ = ∇ Inverse Dynamics$(\dot{q}, v, a, f)$ ▷ Step 2
3: $\partial\ddot{q}/\partial u = -M^{-1}\partial\tau/\partial u$ ▷ Step 3

Algorithm 2 Inverse Dynamics (ID) [14].
Sparse matrices ${}^iX_{\lambda_i}, I_i, S_i$ derived from robot morphology.

1: $v_0 = 0$; a_0 = gravity; Define λ_i = Parent of Link i
2: **for** Link $i = 1 : N$ **do** ▷ Forward Pass
3: Update ${}^iX_{\lambda_i}(q_i)$
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: $f_i = I_i a_i + v_i \times^* I_i v_i - f_i^{\text{external}}$
7: **for** Link $i = N : 1$ **do** ▷ Backward Pass
8: f_{λ_i} += ${}^iX_{\lambda_i}^T f_i$
9: $\tau_i = S_i^T f_i$

implement some operations as multiplication by a constant, which is a simpler operation than multiplication between two variables.

Again, the benefit of this method is that algorithm features that can be exploited for accelerator design (e.g., parallelism, sparse matrix operations) only need to be identified once, after which it is trivial to tune their parameters for each robot model.

Libraries of hardware templates can be distributed like software libraries or parameterized hardware intellectual property (IP) cores, e.g., RISC-V "soft" processors [20].

Step 2: Set Template Parameters. Once a hardware template has been created, it can be used to create customized accelerators for many different robot models by setting the parameters to match the robot morphology.

We use the numbers of limbs and links in the robot to set the numbers of parallel processing elements in the accelerator template. Additionally, we use link inertia values and joint types to set the constant values and sparsity patterns in key link and joint matrices. This tunes the sparsity and complexity of hardware functional units that perform linear algebra manipulations of these matrices, streamlining the type and number of mathematical operations.

For example, the first two links in the LBR iiwa manipulator [31] are connected by a joint whose transformation matrix has only 13 of 36 elements populated. As a result, in the corresponding functional unit for matrix-vector multiplication implemented using a tree of multipliers and adders, operations on zeroed elements can be pruned, reducing multipliers by 64% and adders by 77%.

5 DESIGN

To demonstrate robomorphic computing, we design a hardware accelerator for the dynamics gradient algorithm. We:

(1) Create a hardware template for the dynamics gradient; and
(2) Set the template parameters to customize the accelerator for an example target robot model, an industrial manipulator.

We chose a target robot with only a single limb as a proof of concept, however the techniques demonstrated here readily generalize to robots with multiple limbs (see Section 7). We evaluate this novel accelerator implemented in an FPGA coprocessor and a synthesized ASIC in Section 6.

5.1 Algorithm Details

In many state-of-the-art motion planning and control applications, quantities computed earlier in the optimization process (the joint acceleration $\ddot{q}$ and the inverse of the mass matrix M^{-1}) can be used to compute the forward dynamics gradient [5] using Algorithm 1: (Step 1) Compute inverse dynamics using $\ddot{q}$; (Step 2) Compute the inverse dynamics gradient with respect to inputs from all links; and (Step 3) Recover the forward dynamics gradient by multiplication with the inverse of the mass matrix M^{-1}.

Inverse Dynamics. The standard implementation of inverse dynamics (ID) is the Recursive Newton-Euler Algorithm [14] (Algorithm 2). First, there is a sequential forward pass from the base link of the robot out to its furthest link N, propagating per-link spatial velocities, accelerations, and forces (v_i, a_i, f_i) outward. Then, there is a sequential backward pass, updating the force values f_i and generating the output joint torques τ_i.

The inputs to the algorithm are the link position, velocity, and acceleration $(q_i, \dot{q}_i, \ddot{q}_i)$ expressed in their local coordinate frame, and three key matrices: the link inertia matrix, I_i; the joint transformation matrix, ${}^iX_{\lambda_i}$ (where λ_i is defined as the parent of link i); and the joint motion subspace matrix, S_i. The $I_i, {}^iX_{\lambda_i}, S_i$ matrices all have deterministic sparsity patterns derived from the morphology of the robot model (details in Section 5.2). The sin and cos of the link position q, used to construct the transformation matrices ${}^iX_{\lambda_i}$, can also be cached from an earlier stage of the optimization algorithm.

∇ Inverse Dynamics. The gradient of inverse dynamics (∇ID) is derived from line-by-line analytical derivatives of the ID (Algorithm 2) with respect to position q_j and velocity $\dot{q}_j$ for all links j. For details, see previous work [5]. Again, there is a sequential forward pass and backward pass. Operations on the sparse $I_i, {}^iX_{\lambda_i}, S_i$ matrices are similarly fundamental.

M^{-1} Multiplication. After calculating ∇ID with respect to inputs from all links, multiplication with M^{-1} recovers the forward dynamics gradient.

Workload Characteristics. Prior work has performed substantial workload analysis of these algorithms (see Section 8 for details). The dynamics gradient kernel spends most of its runtime on computation, instead of on memory accesses, making it a "compute-bound" application. Most of the workload is matrix-vector multiplication using matrices that are small (6×6 elements) and middlingly sparse (around 30% to 60% sparse), compared to the large and very sparse matrices in applications such as neural networks (around 50% to 99.9% sparse) [19] which have been the subject of much recent work on hardware acceleration [7, 23, 49]. Compression approaches that offer high performance for large sparse matrices, e.g., compressed sparse row (CSR) encoding [19], are not suitable for exploiting the sparsity in this application because they incur large overheads for encoding and decoding.

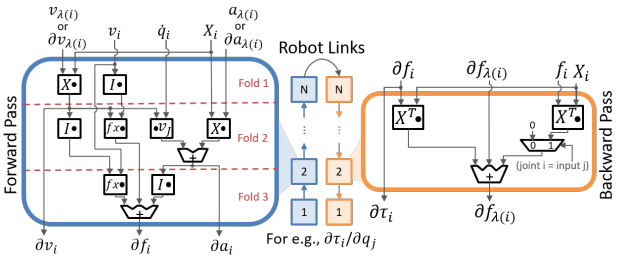

Figure 6: Datapath of the forward and backward pass units for a single link in Step 2 of Algorithm 1. The length of a single datapath and the total number of parallel datapaths are parameterized by the number of links in a robot's limb. The forward pass unit is folded into three sequential stages for efficient resource utilization.

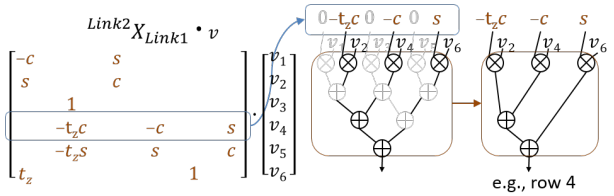

Figure 7: Example of one row of a transformation matrix dot product functional unit $X\cdot$ (as used, e.g., in the forward pass unit in Figure 6) for the joint between the first and second links of a manipulator. The sparsity of the tree of multipliers and adders is determined by the robot joint morphology.

5.2 Step 1: Create Hardware Template

Link-Based Parallelism and Data Flow. The first step of the forward dynamics gradient (Algorithm 1) is computing ID (Algorithm 2). This is a sequential operation with forward and backward passes whose lengths are parameterized by the number of links in a robot, N.

For the second step, ∇ID, we can design the datapaths of our accelerator template to exploit fine-grained link-based parallelism between partial derivatives. We refer to this parallelism as "fine-grained" because of the short duration of the independent threads of execution. They are joined in the third step of Algorithm 1, multiplication of the full gradient matrix with M^{-1}.

To exploit this parallelism, we create separate datapaths per link, made up of sequential chains of forward and backward pass processing units to compute each partial derivative. A single datapath of forward and backward pass units is illustrated in Figure 6. The latency of the computation in each datapath grows linearly with the number of links, $O(N)$. Because there are also as many datapaths as links, the total amount of work in the ∇ID step grows with $O(N^2)$, but when parallelized, its latency grows with $O(N)$.

Note that we compute ∇ID with respect to two inputs, position q and velocity $\dot{q}$. The gradients with respect to q and $\dot{q}$ are completely independent, but share common inputs, so datapaths for both can run in parallel to take advantage of data locality by processing common inputs at the same time.

Every step in the ∇ID forward and backward passes requires inputs v_i, a_i, f_i, produced by the ID for link i. To satisfy this data dependency, the steps of the computation of ID must execute one link ahead of the computation of the ∇ID datapaths. As a result, we are able to exploit parallelism between the first two steps of Algorithm 1, running the datapath that computes ID almost entirely in parallel to the ∇ID datapaths (offset by one link). With this design, computing both ID and ∇ID with respect to q and $\dot{q}$ can all be done with $O(N)$ total latency.

Link and Joint-Based Sparse Functional Units. The datapaths of the accelerator are built from chains of forward and backward pass processing units (see Figure 6). Within these units are circuits of sparse matrix-vector multiplication functional units, e.g., the $I\cdot$, $X\cdot$, and $\cdot v_j$ blocks in the forward pass. To minimize latency, dot products in these functional units are implemented as trees of multipliers and adders. Robot link and joint information can be used to parameterize these operations.

The link inertia matrix I has a fixed sparsity pattern for all robots. Its elements, however, are constant values that are determined by the distribution of mass in a robot's links. If these values are set per-robot, the multipliers in the $I\cdot$ unit can all be implemented as multiplications by a constant value, which are smaller and simpler circuits than full multipliers.

The joint transformation matrix ${}^{i}X_{\lambda_i}$ has a variable sparsity pattern that is determined by robot joint type. When this sparsity is set per-robot, the tree of multipliers and adders in the $X\cdot$ can be pruned to remove operations on zeroed matrix elements, streamlining the functional unit (see Figure 7).

The joint motion subspace matrix S_i has a sparsity pattern determined by joint type. For many common joint types, the columns of S_i are vectors of all zeroes with a single 1 that filter out individual columns of matrices multiplied by S_i. Per robot, this can be encoded within functional units such as $X\cdot$ and $\cdot v_j$ by pruning or muxing operations and outputs from matrix columns that are not selected by the S_i sparsity.

Robot-agnostic sparsity is also exploited in the hardware template. Cross product operations are encoded either as muxes that re-order outputs, or implemented as sparse matrix-vector multiplications, e.g., in the $fx\cdot$ units.

Architectural Optimizations. The final design is shown in Figure 8. We pipeline the forward and backward passes to hide latency and increase throughput for multiple computations.

We perform folding at two different levels of the design, to compress total accelerator area and conserve resources for FPGA implementation. Without aggressive folding, the number of multipliers needed for the template design would be enormous for almost any robot model, making it impossible to implement using the limited number of digital signal processing multiplier units on an FPGA, and consuming a large amount of area in an ASIC. We fold the forward passes of all parallel datapaths into a processor that executes for a single link at a time. We then feed back the results to iterate over all links in the sequential chain. This significantly reduces the number of functional units (a reduction of approximately $O(N)$ in area) in exchange for a small latency penalty (the cost of loading and storing intermediate results to registers). We fold the backward passes in the same manner.

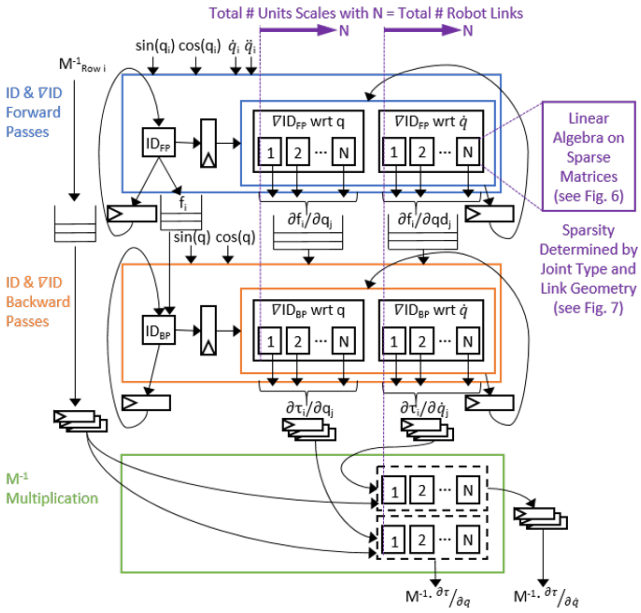

Figure 8: Parameterized hardware template of the dynamics gradient accelerator. The number of links N, link inertia and geometry, and joint types are all parameters that can be set to customize the microarchitecture for a target robot model.

For additional area and resource savings, we also fold the forward pass link units along three divisions, indicated in Figure 6. This allows us to re-use the sparse matrix-vector joint functional units, conserving the number of multipliers and adders needed in the design. Finally, we also performed folding to incorporate the third step of Algorithm 1 into the existing logic for earlier steps. We supplement the multipliers of the backward pass units of ∇ID with respect to velocity $\dot{q}$ to perform the $-M^{-1}$ multiplications in two clock cycles.

5.3 Step 2: Set Template Parameters

In Section 6 we evaluate our accelerator design implemented for the LBR iiwa industrial manipulator (pictured in Figure 1). This target robot model has $N = 7$ links and "revolute"-type joints about the z-axis. We set these parameters to customize the template in Figure 8, instantiating 7 parallel datapaths in the forward and backward passes of the gradients with respect to q and $\dot{q}$, and fixing the constants and sparsity patterns of the functional units (see the example illustrated in Figure 7).

6 EVALUATION

We evaluate the performance of the FPGA implementation of our accelerator, comparing it to off-the-shelf CPU and GPU baselines. We also evaluate a synthesized ASIC version of the accelerator pipeline. In these experiments, our accelerator is implemented for an industrial manipulator. However, in Section 7 we describe how robomorphic computing can be used to adapt the design for different robot models.

Table 1: Hardware System Configurations

Platform	CPU	GPU	FPGA
Processor	i7-7700	RTX 2080	XCVU9P
# of Cores	4	2944 CUDA (46 SM)	N/A
Max Frequency	3.6 GHz	1.7 GHz	55.6 MHz

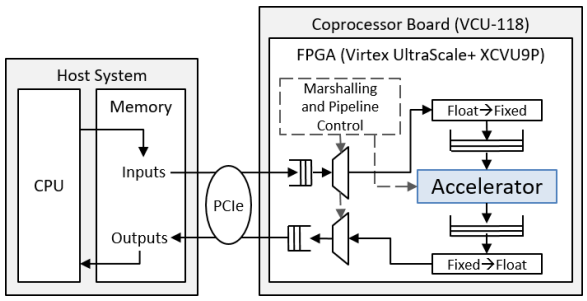

Figure 9: FPGA coprocessor system implementation.

6.1 Methodology

Our accelerator and the software baseline implementations all target the Kuka LBR iiwa-14 manipulator [31] (see Figure 1) using the robot model description file from the rigid body dynamics benchmark suite, RBD-Benchmarks [40].

Baselines. As baselines for comparison, we used state-of-the-art CPU and GPU software implementations of the rigid body dynamics gradient from previous work. The CPU baseline is from the dynamics library Pinocchio [6]. The application was parallelized across the trajectory time steps using a thread pool so that the overheads of creating and joining threads did not impact the timing of the region of interest. The GPU baseline is taken from previous work on implementing nonlinear MPC on a GPU [47], and is also parallelized across trajectory time steps.

Hardware Platforms. The platforms used in our evaluation are summarized in Table 1. Our CPU is a quad-core Intel i7-7700 running Ubuntu 16.04.6. Quad-core Intel i7 processors have been a common choice for the on-board computation for complex robot platforms, including many humanoids featured in the DARPA Robotics Challenge [29, 30, 48, 54]. Similarly, the Spot quadruped from Boston Dynamics [4] offers a quad-core Intel i5 processor on board. The GPU is an NVIDIA GeForce RTX 2080 with 2944 CUDA cores. This platform offers comparable compute resources to the GPU available as an add-on for the Spot quadruped, the NVIDIA P5000 with 2560 CUDA cores. We implemented our accelerator in Verilog on a Xilinx Virtex UltraScale+ VCU-118 board with a XCVU9P FPGA. This platform was selected because it offers a high number of digital signal processing units, which we use to perform the multiplications in our linear algebra-heavy workload. The FPGA design was synthesized at 55.6 MHz.

CPU code was compiled using Clang 10. Code for the GPU was compiled with `nvcc` 11 using `g++5.4`. The GPU and FPGA were connected to a host CPU via PCIe. The GPU used PCIe Gen 3, however the FPGA was restricted to PCIe Gen 1 due to software limitations in the Connectal [25] framework used to implement communication between the host CPU and FPGA.

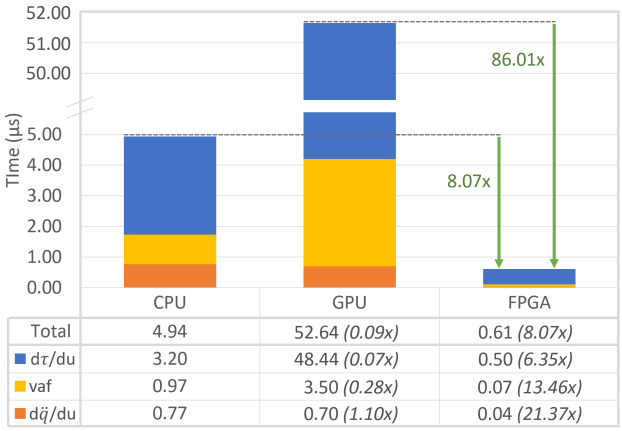

Figure 10: Computation latency of a single calculation of the dynamics gradient. Our FPGA accelerator gives 8× and 86× speedups over the CPU and GPU. The GPU suffers here from high synchronization and focus on throughput, not latency.

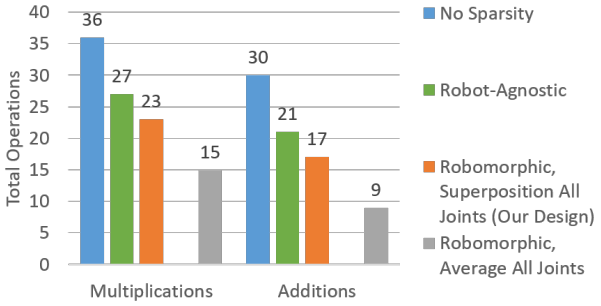

Figure 11: To reduce FPGA resource utilization, we implement a single transformation matrix-vector multiplication unit for all joints in the target robot. Using a superposition of sparsity patterns, we recover 33.3% of average sparsity while conserving area.

Measurements. We disabled TurboBoost on the CPU and fixed all cores at maximum frequency. HyperThreading was enabled for threadpool use. Time was measured with `clock_gettime()`, using `CLOCK_MONOTONIC`. For Section 6.2, we take the mean of one million trials for CPU and GPU. For FPGA, we extract computation latency from cycle counts and clock frequency. For Section 6.3, we measure round-trip wall clock time (including computation and I/O latency) and take the mean of one hundred thousand trials.[1]

Coprocessor System Integration. We integrated our design into a system with the accelerator as an FPGA coprocessor connected to a host CPU (see Figure 9). The CPU generates inputs, then calls the accelerator on the FPGA to compute the dynamics gradient.

Communication between the CPU and FPGA is implemented using Connectal [25]. This framework sets up links that act as direct memory accesses in C++ on the CPU-side software, and FIFO buffers in Bluespec System Verilog on the FPGA-side hardware. Connectal's support of our FPGA board is currently limited to PCIe Gen 1, but the bandwidth is sufficient for our accelerator to outperform CPU and GPU baselines (Section 6.3). Conversion between CPU floating-point data types and accelerator fixed-point data types is implemented on the FPGA using Xilinx IP Cores.

6.2 FPGA Accelerator Evaluation

Latency Results. We compare the latency of a single execution of forward dynamics gradient on the CPU, GPU, and our FPGA implementation (Figure 10). The result is broken down into the three steps of Algorithm 1: inverse dynamics (ID); the gradient of inverse dynamics (∇ID); and M^{-1} multiplication.

In the FPGA results in Figure 10, the pipelining (which would overlap the latency of the forward and backward pass pipelines) was ignored in order to present the latency of a single time step computation being passed through the entire accelerator, from start to finish. Additionally, in our design the datapaths of ID and ∇ID are largely overlapped for the forward and backward passes, but slightly offset. There is a 2-iteration delay that represents the latency overhead of this offset. The ID algorithm needs to be run one robot link "iteration" ahead of each link in the ∇ID algorithm, to produce the "v,a,f" inputs it requires (see Algorithm 1). Because this happens in both the forward pass and the backward pass, it translates into a total overhead of 2 iterations: one extra iteration of the forward pass, plus one extra iteration of the backward pass.

Figure 10 shows the FPGA accelerator demonstrating a significant speedup over the CPU and GPU, despite a much slower clock speed (see Table 1). The accelerator latency is 8× faster than the CPU and 86× faster than the GPU.

The FPGA outperforms the CPU and GPU because it has minimal control flow overhead and can fully exploit parallelism from small matrix operations throughout the workload and parallelism from partial derivatives in ∇ID before they join for the M^{-1} multiplications. This is because the structure of the algorithm is explicitly implemented in the datapaths of the accelerator (Figure 8), including parallelism directly determined by the target robot.

The GPU fares poorly in this experiment because it is a platform optimized for parallel throughput, not the latency of a single calculation. It experiences an especially long latency for ∇ID, the step of Algorithm 1 with the largest computational workload. GPU processors are designed for large vector operations and have difficulty exploiting parallelism from the small sparse matrices in ∇ID. The algorithm is also very serial because of inter-loop dependencies in the forward and backward passes, and joining of partial derivatives in ∇ID for M^{-1} multiplications, forcing many synchronization points and causing overall poor thread occupancy.

The CPU can also only exploit limited parallelism in the linear algebra through vector operations, but its pipeline is optimized for single-thread latency, so it outperforms the GPU.

Joint Transformation Matrix Sparsity. Our FPGA platform offered a limited number of digital signal processing multipliers. We found that our accelerator design used 77.5% of 6840 digital signal processing (DSP) blocks available on our FPGA platform because of the high number of matrix-vector multiplications in the workload.

[1] For the GPU and FPGA 98% of trials were within 2% of the mean. For the CPU 90% of trials were within 2% of the mean for longer trajectories (≥ 32 time steps) but only within 10% of the mean for shorter trajectories.

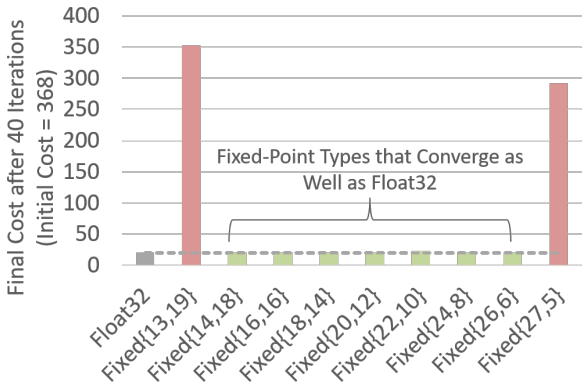

Figure 12: We used 32-bit fixed-point with 16 decimal bits in our design due to FPGA multiplier constraints. However, a range of fixed-point numerical types deliver comparable optimization cost convergence to baseline 32-bit floating-point. Fixed-point labeled as "Fixed{integer bits, decimal bits}".

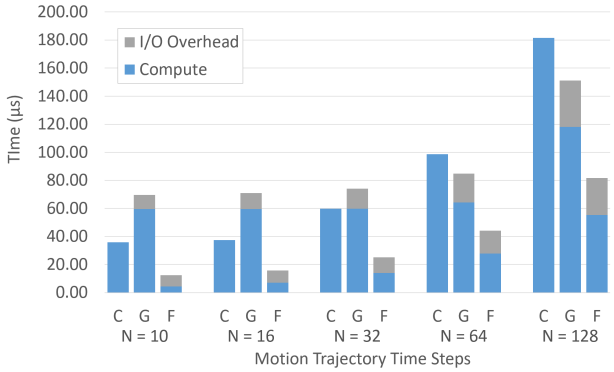

Figure 13: Coprocessor round-trip latency (including I/O) for a range of trajectory time steps. Our FPGA accelerator (F) gives speedups of 2.2× to 2.9× over CPU (C) and 1.9× to 5.5× over GPU (G) for the task of computing multiple dynamics gradients calculations (one per time step).

To conserve multiplier resources, we implemented a single transformation matrix-vector multiplication unit for all seven joints in our target robot model (see Section 5.3). This unit covers a superposition of the matrix sparsity patterns in all individual joints. This design choice uses sparsity to reduce the total operations, while avoiding the waste of area and resources from creating seven separate units.

Figure 11 compares the reduction in operations from this design choice to several baselines. "No Sparsity" is total operations for a dense 6×6 matrix-vector multiplication. "Robot-Agnostic" assumes the upper-right quadrant of the transformation matrix is empty, which is true regardless of robot model. "Robomorphic, Superposition All Joints" is our design choice. Finally, "Robomorphic, Average All Joints" shows the average sparsity of all seven joint matrices, to indicate how well superposition approaches individual sparsity. To achieve or pass this bound would require seven separate units.

Our design choice recovered 33.3% of the average robomorphic sparsity of the individual joint matrices in a single matrix-vector multiplication unit, avoiding the expense of area and resources it would take to instantiate seven units instead. This was a worthwhile tradeoff on our FPGA platform, where area and resources were highly constrained.

Accuracy and Numerical Precision. In our FPGA design, we used a 32-bit fixed-point numerical type with 16 decimal bits. Fixed-point reduces the area and complexity of arithmetic operations compared to floating-point. To validate this choice, we experimented with different data types for the dynamics gradient function within a nonlinear MPC implementation [47]. Figure 12 shows optimization cost convergence results. We used a type-generic Julia software implementation to compare 32-bit fixed-point to a 32-bit floating-point baseline. We explored different numbers of bits for the integer versus decimal, labeled as "Fixed{integer bits, decimal bits}".

A range of fixed-point values worked as well as floating-point, validating our design choice. Results indicate it is possible to use 20 bits (14 integer, 6 decimal) in future work, reducing bit width throughout the computation by 37.5%.

However, we used 32 bits in our current design both because it was convenient for data I/O with a CPU, and our FPGA's digital signal processing multipliers are 27×18 bits, so all operands between 19 and 36 bits require two multipliers.

6.3 Coprocessor System Evaluation

Coprocessor Round-Trip Timing Results. While the results in Figure 10 show computation-only latency for a single calculation of the dynamics gradient, Figure 13 shows coprocessor round-trip latency for a coprocessor system implementation of the accelerator (Figure 9) performing multiple gradient calculations. We define the "coprocessor round-trip" latency as the time between the host CPU making the function call to perform the gradient calculation, and the time when the final gradient calculation data is available in the host CPU memory. This time includes the latency of sending inputs to the coprocessor over communication channels, the latency of all of the computation run on the coprocessor, and the latency of sending and writing outputs back from the coprocessor to the host CPU memory.

The inputs and outputs of the gradient calculations were sent and received by the host CPU, but in the GPU and FPGA experiments, all of the gradient calculations were performed on the GPU or FPGA. FPGA and GPU results include I/O overheads. We evaluate a representative range of trajectories, from 10 to 128 time steps [8, 10, 42, 47, 57]. Each time step requires one dynamics gradient calculation.

The FPGA accelerator gives speedups of 2.2× to 2.9× over CPU and 1.9× to 5.5× over GPU because of its very low latency (see Figure 10). However, the scaling of FPGA performance in this experiment is ultimately limited by throughput at higher numbers of time steps. On our current FPGA platform we heavily utilized the limited digital signal processor resources on the FPGA for linear algebra operations (see Section 6.2). As a result, we could only instantiate the complete accelerator pipeline for a single gradient computation. By contrast, the CPU has 4 cores and the GPU has 46 SMs, so they can process multiple gradient computations in parallel.

Table 2: Synthesized ASIC (12nm Global Foundries) and baseline FPGA results for accelerator computational pipeline.

Platform	FPGA	Synthesized ASIC	
Process Corner	Typical	Slow	Typical
Technology Node [nm]	14	12	12
Max Clock [MHz]	55.6	250	400
Area [mm^2]	N/A	1.627	1.885
Power [W]	9.572	0.921	1.095

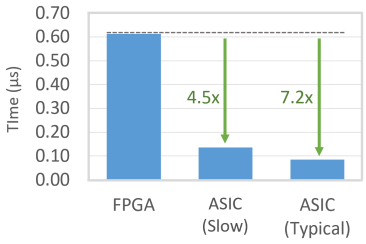

Figure 14: ASIC synthesis indicates a 4.5× to 7.2× speedup in single computation latency over the FPGA results.

The FPGA and GPU have comparable I/O overhead from round-trip memory transfers to the host CPU, despite the FPGA having a lower bandwidth I/O connection than the GPU (PCI Gen 1 vs. Gen 3). We achieve this by pipelining the I/O data marshalling with the execution of each computation.

For small numbers of time steps (32 time steps and below in Figure 10), the CPU benefits from low latency and outperforms the GPU. Beginning at 64 time steps, however, the GPU benefits from high throughput on the large number of parallel computations, and surpasses the CPU. Thread and kernel launch overheads flatten the scaling of both the CPU and GPU at low numbers of time steps (10, 16, and 32).

6.4 Synthesized ASIC Results

While we already see meaningful speedups from implementing our accelerator on an FPGA (see Figures 10 and 13), an ASIC platform offers additional performance and power benefits. We ran ASIC synthesis on the core computational pipeline of our design: the forward and backward pass processors, plus the intermediate SRAM between them (see Figure 8), using the Global Foundries 12nm technology node at both slow and typical process corners. Table 2 compares these ASIC results to our Xilinx XCVU9P FPGA synthesized using the Vivado Design Suite [15]. The FPGA results are the power of the accelerator design measured from Vivado simulation. We report the user design power of the FPGA for fair comparison with our ASIC values. The accelerator design power results include static power.

The maximum clock speed of the core computational pipeline on the ASIC (typical corner) is 7.2× faster than the clock speed of our FPGA implementation. Figure 14 compares the latency for a single computation on the ASIC versus the FPGA.

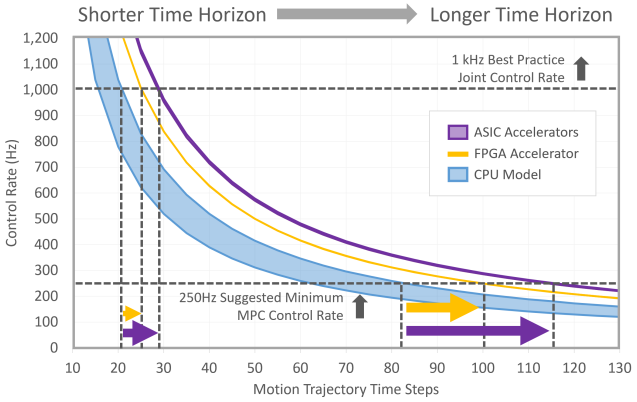

Figure 15: Projected control rate improvements from our dynamics gradient accelerator using the analytical model from Figure 4. We enable planning on longer time horizons for a given control rate, e.g., up to about 100 or 115 time steps instead of 80 at 250 Hz. ASIC results show a narrow range between process corners.

A system-on-chip will allow us to instantiate multiple parallel pipelines, improving the throughput of our accelerator. On the FPGA we can only fit a single pipeline due to limited multiplier resources (Section 6.3). A synthesized ASIC area of 1.9 mm^2 (typical corner), however, suggests many pipelines can fit on a chip. For example, Intel's 14 nm quad-core SkyLake processor [13] is around 122 mm^2, nearly 65× our pipeline area.

A major ASIC benefit is low power dissipation. Power budgets are an emerging constraint in robotics [55], especially for untethered robots carrying heavy batteries. For example, the Spot quadruped has a typical runtime of 90 minutes on a single charge of its battery [4], limiting its range and potential use cases. Power dissipation of our design on an ASIC (typical corner) is 8.7× lower than the calculated power on an FPGA.

6.5 Projected Control Rate Improvement

Finally, we revisit the analytical model from Figure 4 to project control rate improvements from using our dynamics gradient accelerator (Figure 15). We enable faster control rates, which robots can use to either perform more optimization loop iterations to compute better trajectories, or plan on longer time horizons, e.g., up to about 100 or 115 time steps instead of 80 at 250 Hz. Exploring longer planning horizons allows robots to increase their resilience to disturbances and unlock new behaviors.

7 DISCUSSION AND FUTURE WORK

Targeting Other Robotics Applications. The robomorphic computing design methodology can be applied to other critical robotics applications that draw on robot morphology information, including collision detection, localization, kinematics, and dynamics for flexible "soft" robots [2, 11, 32, 45]. For all of these additional robotics applications, a parameterized template only needs to be created once per algorithm.

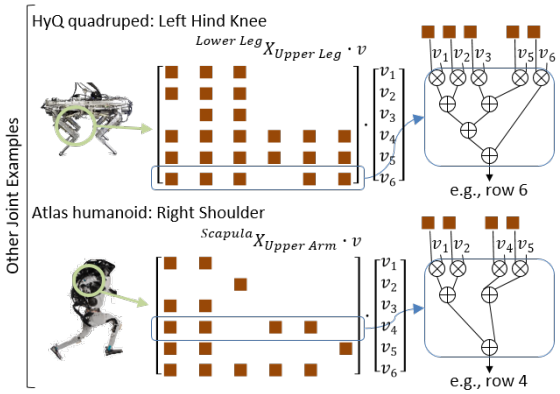

Figure 16: Other examples of joints on real robots [3, 52]. The transformation matrices of these joints exhibit different sparsity patterns, which robomorphic computing translates into sparse matrix-vector multiplication functional units.

These algorithms can be addressed with robomorphic computing because they are tightly coupled with the physical properties of the target robot. For example, standard implementations of rigid body dynamics and kinematics are built upon the same transformations and inertial properties represented in the sparse matrices (e.g., joint transformation matrices ${}^{i}X_{\lambda_i}$ and link inertia matrices I_i) that robomorphic computing maps into pruned sparse linear algebra functional units (Figure 5).

High-fidelity collision detection requires kinematics implicitly. While approximate approaches to collision detection might draw conservative ellipses around the robot and not require precise kinematics, high-fidelity approaches using full kinematics are required when the robot must operate in tight spaces or perform dexterous tasks.

The dynamics of "soft" robots with flexible limbs can be modeled as piecewise approximations using long chains of rigid bodies [44] or by using hybrid rigid-soft dynamics algorithms built on top of traditional rigid body dynamics algorithms [50]. Such approaches are suitable targets for acceleration using robomorphic computing by scaling and extending the methodology for rigid body robots.

Targeting Different Robot Models. In previous sections, we evaluated an accelerator that was customized to target the LBR iiwa industrial manipulator [31]. Here, we describe how the accelerator design would change if it were implemented for other robot models with different joint types, and topologies of limbs and links. Because we used robomorphic computing to design a parameterized accelerator template, it is systematic and simple to customize the template to different robots.

Like the iiwa joint in Figure 7, additional examples of joints on real robots are shown in Figure 16: the left hind knee of the HyQ quadruped [52], and the right shoulder of the Atlas humanoid [3]. Each joint's transformation matrix has a sparsity pattern. Using robomorphic computing, these sparsity patterns directly program the structure of sparse matrix-vector multiplication units by pruning a tree of multipliers and adders.

The topology of limbs and links in a robot model parameterizes the parallelism exposed in the hardware template. For example, if we target our dynamics gradient template to the HyQ quadruped, the customized accelerator will have 4 parallel limb processors, each with 3 parallel datapaths (one per link). The limb outputs will be periodically synchronized at a central torso processor to combine their overall impact.

Automating the Methodology. The most significant contribution of robomorphic computing is that while we performed it manually as a proof-of-concept in Section 5, it is a systematic design flow (Figure 5) that can be automated in future work.

The design of the parameterized hardware template can be automated using a domain-specific language and a high-level synthesis (HLS) flow [1]. These HLS flows use a combination of domain-specific libraries of hand-optimized RTL modules (written once by a hardware engineer) and high-level languages that parameterize and instantiate those modules.

Setting the parameters per-robot to create customized hardware accelerators is also simple to automate. The necessary parameters are already parsed and extracted from robot description files by existing robot dynamics software libraries [6, 36]. These parameters can be used in an HLS flow to automatically output customized hardware. Users can then create accelerators without intervention from roboticists or hardware engineers.

8 RELATED WORK

Workload Analysis. Previous work [40] performed a thorough architectural performance analysis of our key algorithm, RNEA (Algorithm 2), giving insights into its computational characteristics (e.g., compute-bound, spends less than around 10% of clock cycles on memory stalls, working set fits in a 32kB L1 cache with a $< 0.3\%$ miss rate). This analysis, combined with our insights into robot morphology (e.g., deterministic matrix sparsity patterns), motivated our design of a hardware accelerator where we could tailor the computational resources to exactly match the needs of this highly compute-bound application.

Previous work has also broken down how much time is spent on each major computational component of the high-level motion planning workload we examine in Section 3, nonlinear MPC. This work has shown that the dynamics gradient kernel is a major bottleneck, taking as much as 30% to 90% of the total runtime [5, 41, 46, 47], depending on implementation details of the higher-level algorithm and hardware platform. This analysis also demonstrates that the dynamics gradient kernel is an atomic step of the overall workload with a clearly-defined interface, making it an ideal target for hardware acceleration.

Software. Our design methodology builds on previous work in robotics software engineering [5, 36] by introducing per-robot optimization techniques to the hardware domain. Several state-of-the-art software libraries [6, 16, 27, 36, 39] implement robot dynamics functions for use in motion planning and control, but current solutions exhibit a significant performance gap for complex robots (see Section 3).

Hardware. Most work on robotics hardware acceleration to date has focused on perception, including numerous accelerators for neural networks and computer vision [7, 23, 49]. There has also

been recent work on hardware for mapping and localization, e.g., simultaneous localization and mapping (SLAM) [56]. For motion planning and control, prior hardware acceleration work has been limited to collision detection [33, 38] and planning for systems with simple dynamics, such as cars and drones [51]. The focus of our work is motion planning and control applications for robots with complex multi-body dynamics, e.g., manipulators, quadrupeds, and humanoids, which remain largely unexplored.

Design Methodologies. There is an expanding ecosystem of work in developing automatable architectural design tools [21]. A few examples include simulation tools that automatically model underlying computing hardware targets [58]; languages and compilers for performing high-level synthesis from software to RTL [43]; and domain-specific design methodologies to codesign algorithms, architecture, and circuits for, e.g., neural networks and digital signal processing [49, 59]. Our approach shares features with some of these related works, such as the use of parameterized hardware templates, but it also opens up new design possibilities by introducing a new input to the hardware-software codesign process: the physical morphology of a complex robot model.

There is emerging interest in hardware codesign for robotics applications, but work has been limited to neural network-based techniques and platforms with simple dynamics, like drones [28]. Our methodology, by contrast, can be applied to traditional, non-learning-based robotics algorithms and robots with complex dynamics, e.g., manipulators.

9 CONCLUSION

In this work, we introduce robomorphic computing: a design methodology to transform robot morphology into a customized hardware accelerator morphology. Using robomorphic computing, we develop the first hardware accelerator for the gradient of rigid body dynamics, a key bottleneck in emerging robot motion planning and control techniques. We implement the accelerator for an industrial manipulator on an FPGA deployed in a coprocessor system, and a synthesized ASIC. Our accelerator achieves significant speedups over state-of-the-art CPU and GPU solutions. Robomorphic computing generalizes to different robot models and robotics applications, and has a clear pathway to automation. This methodology provides a roadmap for future work in accelerators for robotics applications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant 2030859 to the Computing Research Association for the CIFellows Project, Grant DGE1745303, and Grant 1718160; 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 the views of the funding organizations.

REFERENCES

[1] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal, and Mikel Luján. 2014. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[2] Michael Bloesch, Marco Hutter, Mark A Hoepflinger, Stefan Leutenegger, Christian Gehring, C David Remy, and Roland Siegwart. 2013. State estimation for legged robots-consistent fusion of leg kinematics and IMU. *Robotics* 17 (2013), 17–24.

[3] Boston Dynamics. Accessed in 2020. Atlas® | Boston Dynamics. https://www.bostondynamics.com/atlas Available: bostondynamics.com/atlas.

[4] Boston Dynamics. Accessed in 2020. Spot® | Boston Dynamics. https://www.bostondynamics.com/spot Available: bostondynamics.com/spot.

[5] Justin Carpentier and Nicolas Mansard. 2018. Analytical Derivatives of Rigid Body Dynamics Algorithms. *Robotics: Science and Systems* (2018).

[6] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, and Nicolas Mansard. 2019. The Pinocchio C++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 614–619.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*. ACM/IEEE.

[8] Jared Di Carlo, Patrick M Wensing, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. 2018. 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, 1–9.

[9] Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. 2009. Efficient numerical methods for nonlinear MPC and moving horizon estimation. In *Nonlinear model predictive control*. Springer, 391–417.

[10] Tom Erez, Kendall Lowrey, Yuval Tassa, Vikash Kumar, Svetoslav Kolev, and Emanuel Todorov. 2013. An integrated system for real-time model predictive control of humanoid robots. In *2013 13th IEEE-RAS International conference on humanoid robots (Humanoids)*. IEEE, 292–299.

[11] Christer Ericson. 2004. *Real-time collision detection*. CRC Press.

[12] Farbod Farshidian, Edo Jelavic, Asutosh Satapathy, Markus Giftthaler, and Jonas Buchli. 2017. Real-time motion planning of legged robots: A model predictive control approach. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. IEEE, 577–584.

[13] Eyal Fayneh, Marcelo Yuffe, Ernest Knoll, Michael Zelikson, Muhammad Abozaed, Yair Talker, Ziv Shmuely, and Saher Abu Rahme. 2016. 14nm 6th-generation Core processor SoC with low power consumption and improved performance. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 72–73.

[14] Roy Featherstone. 2008. *Rigid body dynamics algorithms*. Springer.

[15] Tom Feist. 2012. Vivado design suite. *White Paper* 5 (2012), 30.

[16] Martin L. Felis. 2016. RBDL: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots* (2016), 1–17. https://doi.org/10.1007/s10514-016-9574-0

[17] Gianluca Garofalo, Christian Ott, and Alin Albu-Schäffer. 2013. On the closed form computation of the dynamic matrices and their differentiations. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2364–2359.

[18] Markus Giftthaler, Michael Neunert, Markus Stäuble, Marco Frigerio, Claudio Semini, and Jonas Buchli. 2017. Automatic differentiation of rigid body dynamics for optimal control and estimation. *Advanced Robotics* 31, 22 (2017), 1225–1237.

[19] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[20] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 2019. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[21] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* (2019).

[22] Michael G Hollars, Dan E Rosenthal, and Michael A Sherman. 1991. SD/FAST User's Manual. *Symbolic Dynamics Inc* (1991).

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, iemthu DLe, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*. ACM/IEEE.

[24] Claudia Kalb. 2020. Could a robot care for grandma? *National Geographic* (Jan 2020).

[25] Myron King, Jamey Hicks, and John Ankcorn. 2015. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 13–22.

[26] Jonas Koenemann, Andrea Del Prete, Yuval Tassa, Emanuel Todorov, Olivier Stasse, Maren Bennewitz, and Nicolas Mansard. 2015. Whole-body model-predictive control applied to the HRP-2 humanoid. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3346–3351.

[27] Twan Koolen and Robin Deits. 2019. Julia for robotics: Simulation and real-time control in a high-level programming language. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 604–611.

[28] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2020. The Sky Is Not the Limit: A Visual Performance Model for Cyber-Physical Co-Design in Autonomous Machines. *IEEE Computer Architecture Letters* 19, 1 (2020), 38–42.

[29] Eric Krotkov, Douglas Hackett, Larry Jackel, Michael Perschbacher, James Pippine, Jesse Strauss, Gill Pratt, and Christopher Orlowski. 2017. The DARPA robotics challenge finals: results and perspectives. *Journal of Field Robotics* 34, 2 (2017), 229–240.

[30] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. 2016. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots* 40, 3 (2016), 429–455.

[31] KUKA AG. Accessed in 2020. LBR iiwa | KUKA AG. https://www.kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa Available: kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa.

[32] Steven M LaValle. 2006. *Planning algorithms*. Cambridge university press.

[33] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. 2018. Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[34] Courtney Linder. 2019. A Cave Is No Place for Humans, So DARPA Is Sending In the Robots. *Popular Mechanics* (Aug 2019).

[35] J Luh, M Walker, and R Paul. 1980. Resolved-acceleration control of mechanical manipulators. *IEEE Trans. on Automatic Control* 25, 3 (1980), 468–474.

[36] Frigerio Marco, Buchli Jonas, Darwin G Caldwell, and Semini Claudio. 2016. RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages. *Journal of Software Engineering in Robotics* 7, 1 (2016), 36–54.

[37] Carlos Mastalli, Rohan Budhiraja, Wolfgang Merkt, Guilhem Saurel, Bilal Hammoud, Maximilien Naveau, Justin Carpentier, Ludovic Righetti, Sethu Vijayakumar, and Nicolas Mansard. 2020. Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control. In *IEEE International Conference on Robotics and Automation (ICRA)*.

[38] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *MICRO*. IEEE/ACM.

[39] Maximilien Naveau, Justin Carpentier, Sébastien Barthelemy, Olivier Stasse, and Philippe Souères. 2014. METAPOD: Template META-PrOgramming applied to dynamics: CoP-CoM trajectories filtering. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. IEEE, 401–406.

[40] Sabrina M Neuman, Twan Koolen, Jules Drean, Jason E Miller, and Srinivas Devadas. 2019. Benchmarking and Workload Analysis of Robot Dynamics Algorithms. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE.

[41] Michael Neunert, Cédric De Crousaz, Fadri Furrer, Mina Kamel, Farbod Farshidian, Roland Siegwart, and Jonas Buchli. 2016. Fast nonlinear model predictive control for unified trajectory optimization and tracking. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 1398–1404.

[42] Michael Neunert, Farbod Farshidian, Alexander W Winkler, and Jonas Buchli. 2017. Trajectory optimization through contacts and automatic gait discovery for quadrupeds. *IEEE Robotics and Automation Letters* 2, 3 (2017), 1502–1509.

[43] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70.

[44] KeJun Ning and Florentin Wörgötter. 2009. A novel concept for building a hyper-redundant chain robot. *IEEE transactions on robotics* (2009).

[45] Cagdas D Onal and Daniela Rus. 2013. Autonomous undulatory serpentine locomotion utilizing body dynamics of a fluidic soft robot. *Bioinspiration & biomimetics* 8, 2 (2013), 026003.

[46] Zherong Pan, Bo Ren, and Dinesh Manocha. 2019. GPU-based contact-aware trajectory optimization using a smooth force model. In *Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 4.

[47] Brian Plancher and Scott Kuindersma. 2018. A Performance Analysis of Parallel Differential Dynamic Programming on a GPU. In *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*.

[48] Nicolaus A Radford, Philip Strawser, Kimberly Hambuchen, Joshua S Mehling, William K Verdeyen, A Stuart Donnan, James Holley, Jairo Sanchez, Vienny Nguyen, Lyndon Bridgwater, Reginald Berka, Robert Ambrose, Christopher McQuin, John D. Yamokoski, Stephen Hart, Raymond Guo, Adam Parsons, Brian Wightman, Paul Dinh, Barrett Ames, Charles Blakely, Courtney Edmonson, Brett Sommers, Rochelle Rea, Chad Tobler, Heather Bibby, Brice Howard, Lei Nui, Andrew Lee, Michael Conover, Lily Truong, David Chesney, Robert Platt Jr., Gwendolyn Johnson, Chien-Liang Fok, Nicholas Paine, Luis Sentis, Eric Cousineau, Ryan Sinnet, Jordan Lack, Matthew Powell, Benjamin Morris, and Aaron Ames. 2015. Valkyrie: NASA's first bipedal humanoid robot. *Journal of Field Robotics* 32, 3 (2015), 397–419.

[49] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*. ACM/IEEE.

[50] Federico Renda and Lakmal Seneviratne. 2018. A geometric and unified approach for modeling soft-rigid multi-body systems with lumped and distributed degrees of freedom. In *ICRA*. IEEE.

[51] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. 2018. RoboX: An end-to-end solution to accelerate autonomous control in robotics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 479–490.

[52] Claudio Semini, Nikos G Tsagarakis, Emanuele Guglielmino, Michele Focchi, Ferdinando Cannella, and Darwin G Caldwell. 2011. Design of HyQ–a hydraulically and electrically actuated quadruped robot. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 225, 6 (2011), 831–849.

[53] Jonathan Shaw. 2020. The Coming Eldercare Tsunami. *Harvard Magazine* (Jan 2020).

[54] Anthony Stentz, Herman Herman, Alonzo Kelly, Eric Meyhofer, G Clark Haynes, David Stager, Brian Zajac, J Andrew Bagnell, Jordan Brindza, Christopher Dellin, Michael George, Jose Gonzalez-Mora, Sean Hyde, Morgan Jones, Michel Laverne, Maxim Likhachev, Levi Lister, Matt Powers, Oscar Ramos, Justin Ray, David Rice, Justin Scheifflee, Raumi Sidki, Siddhartha Srinivasa, Kyle Strabala, Jean-Philippe Tardif, Jean-Sebastien Valois, J. Michael Vande Weghe, Michael Wagner, and Carl Wellington. 2015. CHIMP, the CMU highly intelligent mobile platform. *Journal of Field Robotics* 32, 2 (2015), 209–228.

[55] Soumya Sudhakar, Sertac Karaman, and Vivienne Sze. 2020. Balancing Actuation and Computing Energy in Motion Planning. In *ICRA*. IEEE.

[56] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. 2018. Navion: a fully integrated energy-efficient visual-inertial odometry accelerator for auto. nav. of nano drones. In *VLSI Circuits*. IEEE.

[57] Yuval Tassa, Tom Erez, and Emanuel Todorov. 2012. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 4906–4913.

[58] Manish Vachharajani, Neil Vachharajani, David A Penry, Jason A Blome, and David I August. 2002. Microarchitectural exploration with Liberty. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings*. IEEE, 271–282.

[59] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. 2020. A Synthesis-Aided Compiler for DSP Architectures (WiP Paper). In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 131–135.

[60] Guang-Zhong Yang, Bradley J. Nelson, Robin R. Murphy, Howie Choset, Henrik Christensen, Steven H. Collins, Paolo Dario, Ken Goldberg, Koji Ikuta, Neil Jacobstein, Danica Kragic, Russell H. Taylor, and Marcia McNutt. 2020. Combating COVID-19—The role of robotics in managing public health and infectious diseases. *Science Robotics* (Mar 2020).