Parallel and Constrained Differential Dynamic Programming for Fast Robotic Motion Planning

A thesis presented

by

Brian Kyle Plancher

to

the John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Masters of Engineering

in the subject of

Engineering Sciences

Harvard University

Cambridge, Massachusetts

May 2018

Dissertation Advisor:
Professor Scott Kuindersma

Author:
Brian Kyle Plancher

Parallel and Constrained Differential Dynamic Programming for Fast Robotic Motion Planning

Abstract

Differential Dynamic Programming (DDP) has become a popular approach to performing trajectory optimization for complex, underactuated robots. However, using DDP in dynamic environments presents three practical challenges. First, the evaluation of dynamics derivatives during optimization creates a computational bottleneck, particularly in implementations that capture second-order dynamic effects. Second, constraints on the states (e.g., boundary conditions, collision constraints) require additional care since the state trajectory is implicitly defined from the inputs and dynamics. Third, computing solutions fast enough for online robotic motion planning can be challenging. This thesis addresses these problems by first building on recent work on Unscented Dynamic Programming (UDP)—which eliminates dynamics derivative computations in DDP—to support general nonlinear state and input constraints to high precision using an augmented Lagrangian. We then leverage parallel computations for increased throughput and systematically analyze the insights, challenges, tradeoffs, and benefits of implementing a parallelized variant of DDP on both a multi-core CPU and a graphics processing unit (GPU). Finally, we present results demonstrating the performance of our constrained UDP (CUDP) and parallel DDP algorithms on several simulated robot systems including a quadrotor and a 7-DoF robotic arm.

Contents

Abstract . . . iii
Acknowledgments . . . viii

1 Introduction **1**

2 Algorithmic Background **4**
2.1 Differential Dynamic Programming (DDP) . . . 4
2.2 Unscented Dynamic Programming (UDP) . . . 9
2.3 Augmented Lagrangian Methods . . . 11

3 Constrained Unscented Dynamic Programming (CUDP) **13**
3.1 Constrained DDP Background . . . 13
3.2 The CUDP Algorithm . . . 15
3.3 Experiments . . . 19
3.3.1 Inverted Pendulum . . . 19
3.3.2 Quadrotor . . . 23
3.3.3 Robotic Arm . . . 25

4 Parallelizing DDP **28**
4.1 Instruction Level Parallelizations for DDP . . . 29
4.2 Algorithmic Level Parallelism for DDP . . . 32
4.2.1 Backward Pass . . . 32
4.2.2 Forward Pass . . . 35
4.3 Final Parallel Algorithm . . . 36

5 Using Parallel DDP for Fast Robotic Motion Planning **38**
5.1 CUDA Background . . . 39
5.2 C++ Threading Background . . . 41
5.3 Implementation Details . . . 42
5.4 Examples . . . 44
5.4.1 Simple Pendulum . . . 45

5.4.2 Inverted Pendulum 47
5.4.3 Quadrotor 49

6 Conclusion **51**

References **53**

List of Tables

3.1 CUDP inverted pendulum optimization results. 23
3.2 CUDP quadrotor optimization results. 25
3.3 CUDP Kuka arm optimization results. 27

List of Figures

2.1 Graphical representation of the Unscented Transform in UDP. 10

3.1 Inverted Pendulum [1] 20
3.2 Cost and constraint violation per iteration for CUDP succeeding on the inverted pendulum. 21
3.3 Cost and constraint violation per iteration for iLQR-A failing on the inverted pendulum. 22
3.4 Collision-free quadrotor trajectory computed by CUDP. 24
3.5 Collision-free Kuka arm trajectory computed by constrained UDP. 26

4.1 Graphical representation of a parallel reduction. 30
4.2 Graphical representation of the backward pass algorithmic level parallelizations. 33
4.3 Graphical representation of the linear coordinate transformation. 33
4.4 Graphical representation of the forward pass algorithmic level parallelizations. 35

5.1 High level architecture differences between a single CPU and GPU processor [2]. 39
5.2 Total cost per iteration for the simple pendulum swing up. 45
5.3 Median time per iteration time for the simple pendulum swing up. 46
5.4 Total cost per iteration for the inverted pendulum swing up. 48
5.5 Median time per iteration time for the inverted pendulum swing up. 48
5.6 Total cost per iteration for the quadrotor flight. 49
5.7 Median time per iteration time for the quadrotor flight. 50

Acknowledgments

This work would not have been possible without...

...my adviser, Professor Scott Kuindersma, for not only providing amazing guidance, support, and insights throughout this process, but more importantly, for taking a chance on a management consultant trying to get into robotics. The trust you have placed in me over the past couple years has not only been empowering, but has also allowed me to live a full life while completing this work. I doubt I could have been as successful with anyone else as an adviser, and am grateful for the opportunity to continue working with you through my doctoral work over the coming years.

...my co-author, Professor Zachary Manchester, without whose patience, expertise, thoughtfulness, and aid in those first couple of months when I was trying to get up to speed, I am not certain I would have been able to convince Scott that I was worth keeping around. Furthermore, without your insights, guidance, and original research, this work would not be possible. We are all sad that you are no longer a postdoc in our lab, but also excited to see what amazing work you will do as a Professor!

...my engineering community: my undergraduate thesis advisor, Professor Greg Morrisett, who also took a chance supporting a former economics major who switched late into computer science, Professor Gu-Yeon Wei, for all of his guidance and support and for teaching me that there is more to fast computing than good software, Professors David Parkes and James Mickens for their support and advice, and of course, the members of the Harvard Agile Robotics Lab, especially my fellow G1/2s for all of their useful feedback, insights, help, and friendship throughout this process.

...my K-12 teachers, especially Ms. Boyea, and Mr. Yoon for instilling in me both a passion for learning, and a respect for hard work.

...my family and friends, for their unwavering, friendship, love, and support.

This work was partially supported by an Internal Research and Development grant from Draper, Inc.

To my mom, dad, pop, ah-ah, grammy, gramps, and sisters who have never stopped cheering me on when I succeed, picking me up when I fall, keeping me company on the phone when I'm driving/biking/walking, and leading me on the right path in life.

To my wife and puppy whose unconditional love and support make every morning a great morning (as long as it is past 10am) and who constantly push me to be the best version of myself.

Chapter 1

Introduction

Trajectory optimization algorithms [3] have been particularly successful in solving the historically important problems of control and path planning for complex dynamic robots [4; 5; 6; 7; 8]. These algorithms pose these problems as a constrained non-convex nonlinear optimization problem—minimizing a cost function (e.g., the energetics of the system) subject to the system dynamics and other real world constraints (e.g., contact with obstacles in the environment). Two popular classes of these algorithms are direct transcription and shooting methods. Direct transcription methods explicitly represent the state, the controls, the dynamics, and any additional constraints. These methods have received a large amount of attention as the problem can be structured as a large (and sparse) nonlinear program that can be solved efficiently using off-the-shelf packages such as the Sequential Quadratic Programming (SQP) solver SNOPT [9], or the Interior Point solver IPOPT [10]. Shooting methods, on the other hand, never explicitly represent the full optimization space and instead optimize over the inputs through minimizing an approximate *cost-to-go* function.

One key subclass of shooting methods is Differential Dynamic Programming (DDP), which parameterizes only the input trajectory and uses Bellman's optimality principle to iteratively solve a sequence of much smaller optimization problems in order to compute the optimal input (and corresponding state) trajectory [11; 12]. Recently, DDP and its variants, particularly the iterative Linear Quadratic Regulator (iLQR) [13], have received increased attention due to growing evidence that online robotic motion planning using a model predictive control (MPC) approach is possible for high-dimensional robots [6; 14; 15; 16; 17]. This is particularly exciting because, as Koenemann et. al. [15] states, "[c]ontrolling the robot with a permanently-updated optimal trajectory, also known as model predictive control, is the Holy Grail of whole-body motion generation." However, while constraints are easily and directly addressed in direct transcription methods, in shooting methods, constraints are typically only approximately addressed by augmenting the native cost with constraint penalty functions. This is problematic as most robot tasks of interest include actuator limits and state constraints (e.g., obstacle avoidance, reaching a desired goal state, maintaining contact with the environment) that need to be strictly satisfied.

At the same time, the impending end of Moore's Law and the end of Dennard Scaling have led to a utilization wall that limits the that performance a single chip can deliver [18; 19]. Computer scientists searching for higher throughput have had to look beyond the CPU to exploit large scale parallelism. For example, in the field of machine learning, computer scientists and hardware engineers have almost exclusively used either high performance GPUs, or designed custom accelerator chips to provide increased performance [20]. Recently, these ideas have even been used in the field of robotics to enable real-time collision checking for a robotic arm in a constrained environment through the use of a custom voxel based collision checker on an FPGA [21; 22], and to compute trajectories on GPUs through sample based methods [23; 24; 25; 26]. Despite these impressive results, and historic interest in parallel trajectory optimization [27], there has been little analysis of the use of modern parallel hardware to accelerate trajectory optimization.

Our goal in this work is twofold. First, we introduce a variant of DDP, Constrained Unscented Dynamic Programming (CUDP), that captures nonlinear constraints on states and inputs with high accuracy while maintaining favorable convergence properties through the use of an augmented Lagrangian. We present results demonstrating its favorable performance on several simulated robot systems. Second, we build on the previous work to systematically analyze the performance benefits and importantly, the trade-offs, of both algorithmic and instruction level parallelism for DDP algorithms by describing, analyzing, and experimenting with a particular implementation of an iLQR algorithm targeting modern multi-core CPUs and GPUs.

In the remainder of this work we first review key concepts from DDP, augmented Lagrangian methods, and the unscented transform (Chapter 2). We then introduce the constrained unscented dynamic programming (CUDP) algorithm, and describe experimental results on an inverted pendulum, a quadrotor flying through a virtual forest, and a manipulator avoiding obstacles in simulation (Chapter 3). We then discuss the ways that DDP algorithms can be parallelized and present our parallel iLQR algorithm (Chapter 4). Finally, we review practical insights and trade-offs that were encountered in our implementation, both for multi-core CPUs and for GPUs, and describe experimental results on a simple pendulum, an inverted pendulum, and a Quadrotor in simulation (Chapter 5).

Chapter 2

Algorithmic Background

2.1 Differential Dynamic Programming (DDP)

The classical DDP algorithm begins by simulating (shooting) a discrete time approximation of the underlying continuous system dynamics forward from an initial state with an initial guess for the input trajectory. An affine control law is then computed in a backwards pass using local quadratic approximations of the cost-to-go. The forward pass is then performed again using this control law to update the input trajectory. This process is repeated until convergence.

Second derivatives of the dynamics (rank-three tensors) are required to compute the quadratic cost-to-go approximations used in DDP. This computational bottleneck led to the development of the iterative Linear Quadratic Regulator (iLQR) algorithm [13], which uses only first derivatives of the dynamics to reduce computation time at the expense of slower convergence. Recent work has proposed a completely derivative-free variant of DDP that uses a deterministic sampling scheme inspired by the unscented Kalman filter [28]. The resulting algorithm, Unscented Dynamic Programming (UDP) [29], has the same computational complexity per-iteration as iLQR with finite-difference derivatives, but provides empirical convergence approaching that

of the full second-order DDP algorithm. In the remainder of the paper, "DDP" is used to refer to vanilla DDP, iLQR, and UDP when the statement holds for all algorithms.

We now derive the classical DDP algorithm by assuming a discrete-time nonlinear dynamical system of the form:

$$x_{k+1} = f(x_k, u_k), \tag{2.1}$$

where $x \in \mathbb{R}^n$ is the system state and $u \in \mathbb{R}^m$ is a control input. The goal is to find an input trajectory, $U = \{u_0, \ldots, u_{N-1}\}$, that minimizes an additive cost function,

$$J(x_0, U) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k), \tag{2.2}$$

where x_0 is the initial state and $x_1, \ldots, x_N$ are computed by integrating the discrete time forward dynamics (2.1).

Using Bellman's principle of optimality [30], the *optimal cost-to-go*, $V_k(x)$, can be defined by the recurrence relation:

$$\begin{aligned} V_N(x) &= \ell_f(x_N) \\ V_k(x) &= \min_u \ \ell(x, u) + V_{k+1}(f(x, u)). \end{aligned} \tag{2.3}$$

When interpreted as an update procedure, this relationship leads to classical dynamic programming algorithms [30]. However, the curse of dimensionality prevents direct application of dynamic programming to most systems of interest to the robotics community. In addition, while $V_N(x) = \ell_f(x_N)$, and often has a simple analytical form, $V_k(x)$ will typically have complex geometry that is difficult to represent due to the nonlinearity of the dynamics (2.1). DDP avoids these difficulties by settling for *local* approximations to the cost-to-go along a trajectory.

$Q(\delta x, \delta u)$ is defined as the local change in the minimization argument in (2.3) under perturba-

tions, $\delta x, \delta u$:

$$Q(\delta x, \delta u) = \ell(x + \delta x, u + \delta u) + V(f(x + \delta x, u + \delta u)) - \ell(x, u) - V(f(x, u)). \tag{2.4}$$

Taking the second-order Taylor expansion of Q results in:

$$Q(\delta x, \delta u) \approx \frac{1}{2} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_{xx} & Q_{xu} \\ Q_u & Q_{xu}^T & Q_{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}, \tag{2.5}$$

where the block matrices are computed as:

$$\begin{aligned} Q_{xx} &= \ell_{xx} + f_x^T V'_{xx} f_x + V'_x \cdot f_{xx} \\ Q_{uu} &= \ell_{uu} + f_u^T V'_{xx} f_u + V'_x \cdot f_{uu} \\ Q_{xu} &= \ell_{xu} + f_x^T V'_{xx} f_u + V'_x \cdot f_{xu} \\ Q_x &= \ell_x + f_x^T V'_x \\ Q_u &= \ell_u + f_u^T V'_x. \end{aligned} \tag{2.6}$$

Following the notation used elsewhere [6], the explicit time indices are dropped and a prime is used to indicate the next timestep. Derivatives with respect to x and u are denoted with subscripts. The rightmost terms in the equations for Q_{xx}, Q_{uu}, and Q_{xu} involve second derivatives of the dynamics, which are rank-three tensors. As mentioned previously, these tensor calculations are relatively expensive and are often omitted, resulting in the iLQR algorithm [13].

Minimizing equation (2.5) with respect to δu results in the following correction to the control trajectory:

$$\delta u = -Q_{uu}^{-1} (Q_{ux} \delta x + Q_u) \equiv K \delta x + \kappa, \tag{2.7}$$

which consists of an affine term κ and a linear feedback term $K\delta x$. These terms can be

substituted back into equation (2.5) to obtain an updated quadratic model of V:

$$\begin{aligned} V_x &= Q_x - Q_{xu}\kappa \\ V_{xx} &= Q_{xx} - Q_{xu}K. \end{aligned} \tag{2.8}$$

Additionally the expected change in the optimal cost-to-go can be computed:

$$\delta V^* = -\frac{1}{2} Q_u \kappa. \tag{2.9}$$

Therefore, a backward update pass can be performed starting at the final state, x_N, by setting $V_N = \ell_f(x_N)$, and iteratively applying the above computations. A forward simulation pass is then performed to compute a new state trajectory using the updated controls. This forward-backward process is repeated until the algorithm converges within a specified tolerance.

DDP, like other variants of Newton's method, can achieve quadratic convergence near a local optimum [12; 31]. However, care must be taken to ensure good convergence behavior from arbitrary initialization: a line search parameter, α, must be added to the forward pass to ensure a satisfactory decrease in cost, and a regularization term, ρI_m, must be added to Q_{uu} in equation (2.7) to ensure positive-definitiveness [32]. Finally, a choice must be made to use a Euler, midpoint, Runge–Kutta [33], or other higher order integrator, to compute the discrete time approximation of the system dynamics, which trades off integration accuracy for speed.

This full procedure is summarized in Algorithm 1.

Algorithm 1: *DDP*

1: Initialize the algorithm and load in initial trajectories
2: **while** cost not converged **do**
3: Compute V_N and derivatives
4: **for** $k = N-1, \ldots, 0$ **do**
5: (2.6) $\rightarrow Q^k$
6: **if** Q^k_{uu} is invertible **then**
7: (2.7) $\rightarrow K_k, \kappa_k$
8: (2.8), (2.9) $\rightarrow V_k$ and derivatives
9: **else**
10: Increase ρ go to line 4
11: **end if**
12: **end for**
13: $\alpha = 1$
14: $\tilde{x}_0 = x_0$
15: **for** $k = 0, \ldots, N-1$ **do**
16: $\tilde{u}_k = u_k + \alpha\kappa_k + K_k(\tilde{x}_k - x_k)$
17: $\tilde{x}_{k+1} = f(\tilde{x}_k, \tilde{u}_k)$
18: **end for**
19: (2.2) and $\tilde{X}, \tilde{U} \rightarrow \tilde{J}$
20: **if** $\tilde{J}$ satisfies line search criteria **then**
21: Update $X \leftarrow \tilde{X}$, $U \leftarrow \tilde{U}$
22: **else**
23: Reduce α and go to line 15
24: **end if**
25: Quadratize cost at X, U
26: Quadratize dynamics at X, U
27: **end while**

Backward Pass

Forward Pass

Next Iteration Setup

2.2 Unscented Dynamic Programming (UDP)

UDP [29] replaces the gradient and Hessian calculations in equation (2.6) with approximations computed from a set of sample points. Inspired by the Unscented Kalman Filter [28], points are sampled from a level set of the cost-to go function and propagated backward in time through the nonlinear dynamics. They are then used to compute a new cost-to-go approximation at the earlier timestep.

To compute the derivatives of $V(f(x,u))$ appearing in the Hessian (2.6), a set of $2(n+m)$ sample points are generated from the columns of the following matrix:

$$L = chol\left(\begin{bmatrix} V'_{xx} & 0 \\ 0 & \ell_{uu} \end{bmatrix}^{-1}\right). \tag{2.10}$$

Each column, L_i, is scaled by a constant factor, β, and both added to and subtracted from the vector $[x';u]$ (again, using the shorthand $x = x_k$, $u = u_k$, $x' = x_{k+1}$):

$$\begin{bmatrix} \tilde{x}'_i \\ \tilde{u}_i \end{bmatrix} = \begin{bmatrix} x' \\ u \end{bmatrix} + \beta L_i \qquad \begin{bmatrix} \tilde{x}'_{i+m+n} \\ \tilde{u}_{i+m+n} \end{bmatrix} = \begin{bmatrix} x' \\ u \end{bmatrix} - \beta L_i. \tag{2.11}$$

The samples are then propagated backwards through the dynamics such that $\tilde{x}_i = f^{-1}(\tilde{x}'_i, \tilde{u}_i)$. A discrete time backwards dynamics function can always be defined for a continuous-time dynamical system by simply applying the integration rule backwards in time. Note that this problem is not well posed for dynamics that include rigid contact unless certain smoothing approximations are made [34].

Using these sample points, the Hessian in equation (2.6) becomes:

$$\begin{aligned} \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix} &= M^{-1} + \begin{bmatrix} \ell_{xx} & \ell_{xu} \\ \ell_{ux} & 0 \end{bmatrix} \\ M &= \frac{1}{2\beta^2} \sum_{i=1}^{2(n+m)} \left(\begin{bmatrix} \tilde{x}_i \\ \tilde{u}_i \end{bmatrix} - \begin{bmatrix} x \\ u \end{bmatrix}\right)\left(\begin{bmatrix} \tilde{x}_i \\ \tilde{u}_i \end{bmatrix} - \begin{bmatrix} x \\ u \end{bmatrix}\right)^T. \end{aligned} \tag{2.12}$$

The gradient terms in (2.6) can be computed from the same set of sample points by solving a linear system,

$$\begin{bmatrix} Q_x \\ Q_u \end{bmatrix} = D^{-1} \begin{bmatrix} V_x'\tilde{x}_i - V_x'\tilde{x}_{i+m+n} \\ \vdots \\ V_x'\tilde{x}_{n+m} - V_x'\tilde{x}_{2(m+n)} \end{bmatrix} + \begin{bmatrix} \ell_x \\ \ell_u \end{bmatrix}, \tag{2.13}$$

$$D = \begin{bmatrix} \tilde{x}_i - \tilde{x}_{i+m+n} & \dots & \tilde{x}_{m+n} - \tilde{x}_{2(m+n)} \\ \tilde{u}_i - \tilde{u}_{i+m+n} & \dots & \tilde{u}_{m+n} - \tilde{u}_{2(m+n)} \end{bmatrix},$$

which is equivalent to a centered finite difference. This procedure is shown graphically below in Figure 2.1:

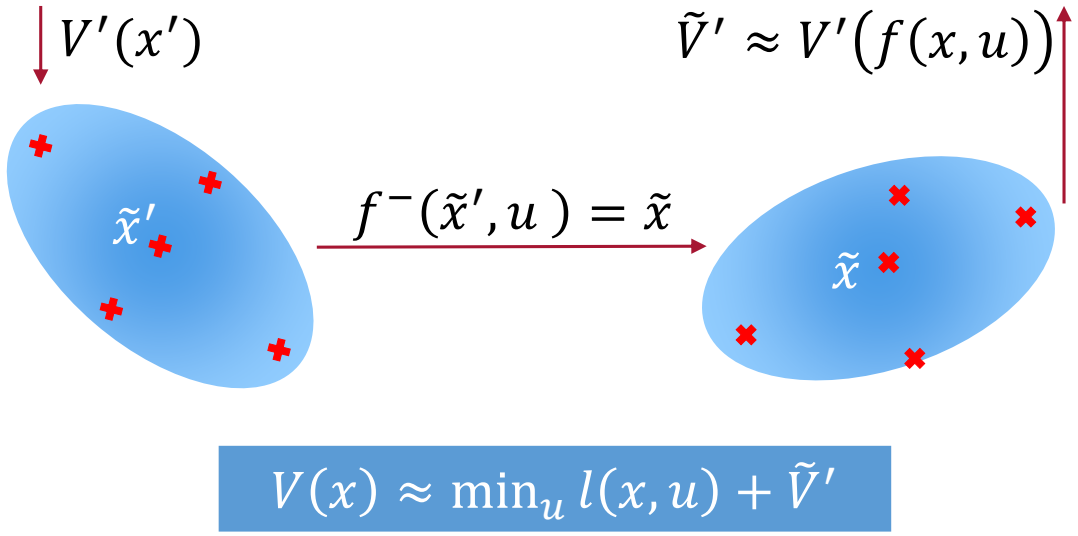

Figure 2.1: *Graphical representation of the Unscented Transform in UDP.*

2.3 Augmented Lagrangian Methods

Trajectory optimization aside, a natural approach to approximately enforcing constraints in optimization algorithms is to apply a quadratic penalty to constraint violations. Suppose that we wish to solve the generic minimization problem:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & g(z) \\ \text{subject to} \quad & c_i(z) = 0 \quad i \in \mathcal{E}, \end{aligned} \tag{2.14}$$

where $g(z)$ and $c_i(z)$ are smooth nonlinear functions. Penalty methods for solving constrained optimization problems start by defining a new cost function,

$$g_p(z;\mu) = g(z) + \frac{\mu}{2} \sum_{i \in \mathcal{E}} c_i(z)^T c_i(z), \tag{2.15}$$

where μ is a scalar weighting parameter. As $\mu \to \infty$, the minimizing value of $g_p(z;\mu)$ will converge toward satisfaction of the constraints [35]. While μ often does not have to grow unbounded for a solution to be found within a given tolerance, numerical issues are still prevalent since the condition number of the Hessian of g_p grows with μ (see [35] for additional details).

To overcome the numerical issues associated with penalty methods, augmented Lagrangian solvers add a linear term to g_p that estimates the Lagrange multipliers associated with the constraints:

$$\mathcal{L}_A(z;\mu,\lambda) = g(z) + \frac{\mu}{2} \sum_{i \in \mathcal{E}} c_i(z)^T c_i(z) + \sum_{i \in \mathcal{E}} \lambda_i c_i(z). \tag{2.16}$$

Given initial values for μ and λ, an unconstrained minimization is performed, after which μ and λ are updated. As in penalty methods, μ is systematically increased across these *major iterations* using a predefined schedule. However, the presence of the λ terms allows convergence with much smaller values of μ.

The update for λ at major iteration j can be derived by considering the first-order necessary

conditions evaluated at an approximate minimizer, z_j:

$$0 \approx \nabla_z \mathcal{L}_A(z_j; \mu_j, \lambda^j) = \nabla_z g(z_j) - \sum_{i \in \mathcal{E}} [\lambda_i^j - \mu_j c_i(z_j)] \nabla_z c_i(z_j). \tag{2.17}$$

Recall that the first-order necessary conditions for a local solution, z^*, λ^*, of the original constrained optimization problem is given by differentiating the (true) Lagrangian [35]:

$$0 = \nabla_z g(z^*) - \sum_{i \in \mathcal{E}} \lambda_i^* \nabla_z c_i(z^*). \tag{2.18}$$

Comparing (2.17) and (2.18), a natural update rule for λ arises:

$$\lambda_i^{j+1} \leftarrow \lambda_i^j - \mu_j c_i(z_j) \quad \forall i \in \mathcal{E}. \tag{2.19}$$

It can be shown that given λ^*, the solution, z^*, of (2.14) is a strict local minimizer of (2.16) for all μ above some minimum value [35]. Practically speaking, the aim is to quickly improve estimates of λ^* so that reasonable approximations of z^* can be computed by minimizing (2.16) without μ growing too large.

Chapter 3

Constrained Unscented Dynamic Programming (CUDP)[1]

3.1 Constrained DDP Background

As mentioned earlier, while DDP has been shown to be a very powerful algorithm, the framework does not naturally handle arbitrary state and input constraints despite the need for them for many robotic tasks including: respecting torque limits on actuators, avoiding obstacles, reaching a desired goal state, and maintaining or breaking contact with the environment.

Several authors have proposed methods for adding constraints to DDP methods. These approaches fall into two classes. The first approach was first described by Pantoja and Mayne in 1989 [37] and later by Tassa et. al. in 2014 [38]. In both cases *hard constraints* are put on the inputs by projecting the optimal feedback controller onto the input constraint manifold. When the constraints are limted to bounding box constraints, which is natural for torque limits, this is done by solving a Box QP in the backward pass instead of the usual Ricatti

[1] Co-authored with Zachary Manchester and Scott Kuindersma [36]

equations. This approach is able to guarantee that torque constraints are never violated at any iteration. Farshidian et. al. [39] extend this to equality constraints on both the state and input through a similar projection framework. However, this extension is still unable to handle pure state constraints which are handled via penalty functions. Finally, Xie et. al. [40] extend this approach to state and/or input constraints by solving a QP in both the forward and backward passes.

The second approach is to include all of the constraints as a penalty term in the cost function. These methods easily allow for arbitrary constraints of all types to be constructed. However, while these *soft constraints* are penalized for in the cost function their satisfaction is never explicitly required. As such, the final trajectory may not exactly satisfy the constraints and designing effective penalty functions that satisfy constraints to high precision can be difficult and time consuming. One solution is to use a continuation method and increase penalty weighting coefficients in the cost function until convergence to a result that satisfies the constraints [39]. However, it is well known that these methods often lead to numerical ill-conditioning before reaching the desired constraint tolerance [41; 35]. In practice, this leads to infeasible trajectories or collisions when run on hardware. Despite this, penalty methods have seen broad application in robotics.

It is important to note that while we focus on quadratic penalties in this work, there are a wide variety of other penalty functions that can be used. For example, L_1 loss functions have been used for collision-free path planning in robotic arms and humanoids [42] and van den Berg [43] uses exponential barrier cost terms in his LQR Smoothing algorithm to prevent a quadrotor from colliding with cylindrical obstacles. That said, many of these other loss functions still have other challenges. For example, barrier methods require particular care in cases where feasible initial guesses cannot be easily generated.

As pointed out by other researchers [44], augmented Lagrangian methods may be particularly well-suited to trajectory optimization problems as they not only allow the solver to temporarily traverse infeasible regions and aggressively move towards local optima, like other soft constraint

based continuation methods, but they also allow for constraints to be satisfied to high precision. A theoretical formulation of this method was proposed for use in the DDP context over two decades ago [45] and was later used to develop a hybrid-DDP algorithm [46], but this work appears to have received little attention in the robotics community. We compare our CUDP algorithm against this method.

3.2 The CUDP Algorithm

The CUDP algorithm applies the augmented Lagrangian constraint formulation to the standard UDP algorithm. It does this by adding quadratic penalty and Lagrange multiplier terms for each constraint to the cost function and defining an outer loop to update λ and μ when the inner augmented UDP solution converges. We use the following augmented Lagrangian function:

$$\mathcal{L}_A(x_0, U; \mu, \lambda) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k) \; + \frac{1}{2} \sum_{k=0}^{N} c(x_k, u_k)^T I_{\mu_k} c(x_k, u_k) + \sum_{k=0}^{N} \lambda_k c(x_k, u_k), \tag{3.1}$$

where $c(x, u)$ is the vertical concatenation of all equality and inequality constraints:

$$\begin{aligned} &c_i(x, u) = 0, \quad i \in \mathcal{E} \\ &c_i(x, u) \geq 0, \quad i \in \mathcal{I}. \end{aligned} \tag{3.2}$$

Following [44] inequality constraints are handled through I_{μ_k} which is defined as a diagonal matrix that encodes the active constraints:

$$I_{\mu_k}(i,i) = \begin{cases} \mu_k^i & i \in \mathcal{E} \\ \mu_k^i & i \in \mathcal{I} \;\cap\; \left\{ c_i(x_k, u_k) < 0 \;\cup\; \lambda_k^i > 0 \right\} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Including I_{μ_k} in Equation (3.1) ensures that penalties are not incurred when inequality constraints are satisfied.

At each timestep during the backward pass of the UDP algorithm, the constraint functions and their gradients are evaluated. A modified version of $Q(\delta x, \delta u)$ from equations (2.5)–(2.6), denoted as $\hat{Q}$, is then defined to include a Gauss-Newton approximation of the constraint terms:

$$\begin{aligned}
\hat{Q}_{xx} &= Q_{xx} + \frac{\partial c(x,u)}{\partial x} I_\mu \frac{\partial c(x,u)}{\partial x} \\
\hat{Q}_{uu} &= Q_{uu} + \frac{\partial c(x,u)}{\partial u} I_\mu \frac{\partial c(x,u)}{\partial u} + \rho I \\
\hat{Q}_{xu} &= Q_{xu} + \frac{\partial c(x,u)}{\partial x} I_\mu \frac{\partial c(x,u)}{\partial u} \\
\hat{Q}_{x} &= Q_{x} + c(x,u) I_\mu \frac{\partial c(x,u)}{\partial x} + \text{diag}(\lambda) \frac{\partial c(x,u)}{\partial x} \\
\hat{Q}_{u} &= Q_{u} + c(x,u) I_\mu \frac{\partial c(x,u)}{\partial u} + \text{diag}(\lambda) \frac{\partial c(x,u)}{\partial u},
\end{aligned} \tag{3.4}$$

where the regularization parameter $\rho > 0$ is included. Equations (2.7) and (2.8) can then be used as usual to compute the feedback policy during the backwards pass.

The forward pass remains unchanged except that the computation of the total cost must be adjusted to include the cost from the augmented Lagrangian instead of the original cost function.

As is typically done in augmented Lagrangian methods, λ is updated only if the constraint violation of the local minimizer is less than a threshold value, ϕ, and otherwise μ is updated. The parameter ϕ is updated according to a predefined schedule to help guide the algorithm toward a solution while avoiding large increases in μ early on. While there are many different variations on this schedule in the literature, most suggest a monotonically decreasing schedule for ϕ, which results in a monotonically increasing μ [47; 35].

Note that while it is possible to use a single global value of μ, CUDP uses a separate μ_k^i and λ_k^i for each constraint at each timestep. Bertsekas notes that this approach gives the solver the most flexibility, at the expense of computational overhead [47]. Experiments indicate that

this is necessary for most robotics problems as the various constraints usually converge to very different final values for μ and λ. Intuitively, a torque constraint should not behave the same as a final state constraint as the former can be adjusted for at each timestep, while the latter is the result of the full trajectory. It is important to note that this flexibility sometimes comes with the price of additional major iterations, where adjacent timesteps pass large inputs back-and-forth until both μ values are sufficiently increased. Finally, the initial value of μ is set to be small relative to the primal cost function and increased slowly enough that ill-conditioning does not occur in early major iterations, but also fast enough that constraint tolerances are met in a reasonable amount of time.

In the end, the final algorithm proceeds by running the inner augmented UDP algorithm until convergence, and then updating μ and λ according to a set schedule until the desired feasibility tolerance is achieved. The full CUDP algorithm is summarized in Algorithm 2.

Algorithm 2: *CUDP*

1: Initialize the algorithm and load in initial trajectories
2: **while** $\max(c) > \epsilon_c$ **do**
3: **while** cost not converged **do**
4: Compute V_N and derivatives
5: **for** $k = N-1, \ldots, 0$ **do**
6: (2.12)–(2.13), (3.4) $\rightarrow \hat{Q}^k$
7: **if** $\hat{Q}^k_{uu}$ is invertible **then**
8: (2.7) $\rightarrow K_k, \kappa_k$
9: (2.8) $\rightarrow V_k$ and derivatives
10: **else**
11: Increase ρ go to line 5
12: **end if**
13: **end for**
14: $\alpha = 1$
15: $\tilde{x}_0 = x_0$
16: **for** $k = 0, \ldots, N-1$ **do**
17: $\tilde{u}_k = u_k + \alpha\kappa_k + K_k(\tilde{x}_k - x_k)$
18: $\tilde{x}_{k+1} = f(\tilde{x}_k, \tilde{u}_k)$
19: **end for**
20: Compute $\tilde{J}$ using (3.1) and $\tilde{X}, \tilde{U}$
21: **if** $\tilde{J}$ satisfies line search criteria **then**
22: Update $X \leftarrow \tilde{X}$, $U \leftarrow \tilde{U}$
23: **else**
24: Reduce α and go to line 16
25: **end if**
26: Quadratize cost at X, U
27: Quadratize dynamics at X, U
28: Linearize constraints, update I_λ at X, U
29: **end while**
30: **for** $k = 0, \ldots, N$ **do**
31: **for** $i \in \mathcal{E} \cup \mathcal{I}$ **do**
32: **if** $c_i^k < \phi_i^k$ **then**
33: Update λ_i^k using (2.19)
34: Reduce ϕ_i^k
35: **else**
36: Increase μ_k^i
37: **end if**
38: **end for**
39: **end for**
40: **end while**

Backward Pass

Forward Pass

Next Iteration Setup

Outer Loop Updates

3.3 Experiments

In this section, three numerical examples are provided to demonstrate the performance of CUDP. To do this, UDP and iLQR are compared using both penalty (-P) and augmented Lagrangian (-A) formulations.[2] All of the algorithms are implemented in MATLAB. Each uses the same scheduling of updates to μ and λ, and all integration is done with a 3rd-order Runge-Kutta [33] method and a time horizon of 4 seconds. The example systems are available as part of Drake [48]. A MATLAB implementation of the algorithm with these examples is available at `http://bit.ly/ConstrainedUDP`.

3.3.1 Inverted Pendulum

The first example is the classic swing-up task for the two degree of freedom inverted-pendulum system (see Figure 3.1). The state vector is defined as $x = [y, \theta, \dot{y}, \dot{\theta}]^T$, where y is the translation of the cart and θ is the angle of the pendulum measured from the downward equilibrium. The initial state is $x_0 = [0, 0, 0, 0]^T$, the stable downward equilibrium, and the goal state is $x_g = [0, \pi, 0, 0]^T$, the unstable upward equilibrium. A quadratic cost function of the form:

$$J = \frac{1}{2}(x_N - x_g)^T Q_N (x_N - x_g) \; + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T Q (x_k - x_g) + \frac{1}{2} u_k^T R u_k, \tag{3.5}$$

where $Q = 0.1 \times I_{4x4}$, $R = 0.01$, and $Q_N = 1000 \times I_{4x4}$ is used. $N = 120$ knot points are used for the discretization of the trajectory and the algorithm is initialized with all of the states and controls set to 0. An input constraint of $\pm 30\,\mathrm{N}$, and a final state constraint of $x_N = x_g$ are enforced. Optimizations are run at three different constraint tolerances: $1e^{-2}$ ("low precision"), $1e^{-4}$ ("medium precision"), and $5e^{-7}$ ("high precision"). In all cases, the intermediate cost convergence tolerance was set to $1e^{-2}$ and the final iteration cost convergence tolerance was set to $1e^{-6}$. Different conversion criteria are used because while a very precise final trajectory is

[2] Note that UDP-A is the CUDP algorithm.

desired, initial iterations that do not satisfy constraint tolerances do not converge to the final solution, so there is no need to spend time getting a very precise answer. A maximum value of $1e^{30}$ was set for μ, since allowing it to grow much larger led to poor numerical conditioning. For both penalty-based and augmented UDP, β was set to $1e^{-2}$.

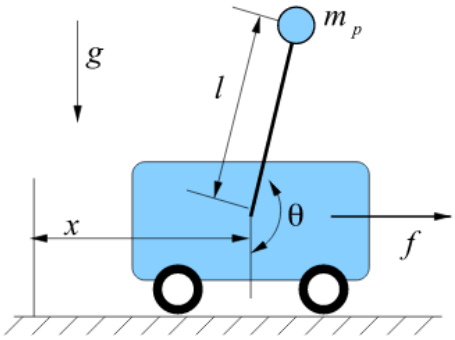

Figure 3.1: *Inverted Pendulum [1]*

Figure 3.2 provides some intuition for how the constrained UDP algorithm solves the inverted pendulum swing-up problem. The vertical dashed black lines indicate outer loop updates occurring at the end of major iterations. Intuitively, the first major iteration finds a local minimum of the primary objective despite large constraint violations. Later iterations reduce constraint violation while often increasing cost (unconstrained solutions give a lower bound on cost). Despite increasing values of μ in the final few iterations, the product of $c(x, u)$ and μ remains relatively constant due to the corresponding decrease in constraint violation. The algorithm also often violates input constraints temporarily to guide the state trajectory to feasible regions of state space. This makes intuitive sense as each input is independently decided and can be drastically changed at will, unlike each state, making them easy to violate temporarily and fix later. This behavior is qualitatively different from SQP and other hard constraint based methods, where linearized constraints are strictly satisfied during each iteration.

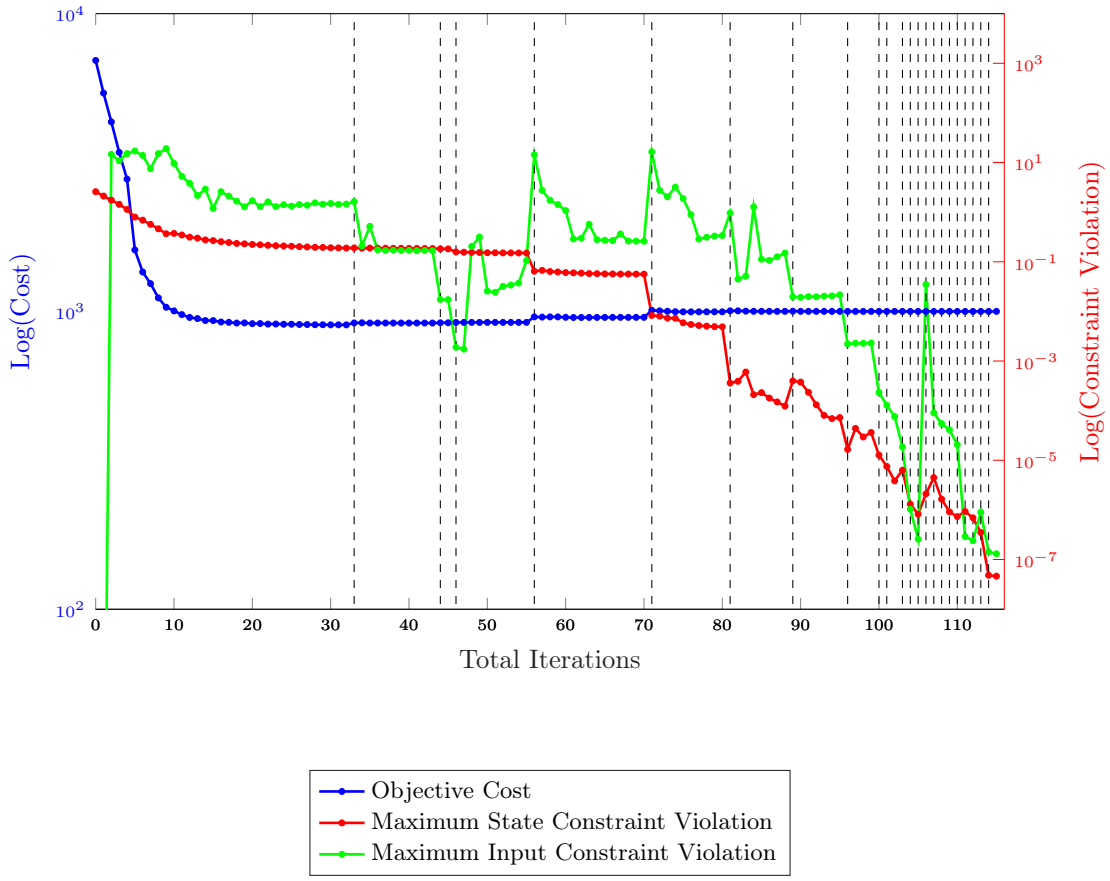

Figure 3.2: *Cost and constraint violation per iteration for CUDP succeeding on the inverted pendulum.*

Figure 3.3 shows a failing output from the augmented iLQR algorithm (iLQR-A) in the high-precision case, exiting after reaching a limit of 100 major iterations. The algorithm makes progress until the upper bound of $\mu \leq 1e^{30}$ is reached on the final state constraint around iteration 100. After that, as various inputs are increased beyond the torque limits to attempt to satisfy the final state constraint, their respective μ parameters rise, which decreases the relative weight of the penalty on the final state constraint. This further exacerbates the problem and prevents any further progress.

Complete results from the inverted pendulum experiments are given in Table 3.1. Results in red indicate a failure to meet the required constraint tolerance. All of the algorithms were able to converge in the low-precision setting with very similar total cost, but the penalty methods required much larger maximum values for μ. iLQR-A finished the fastest, although

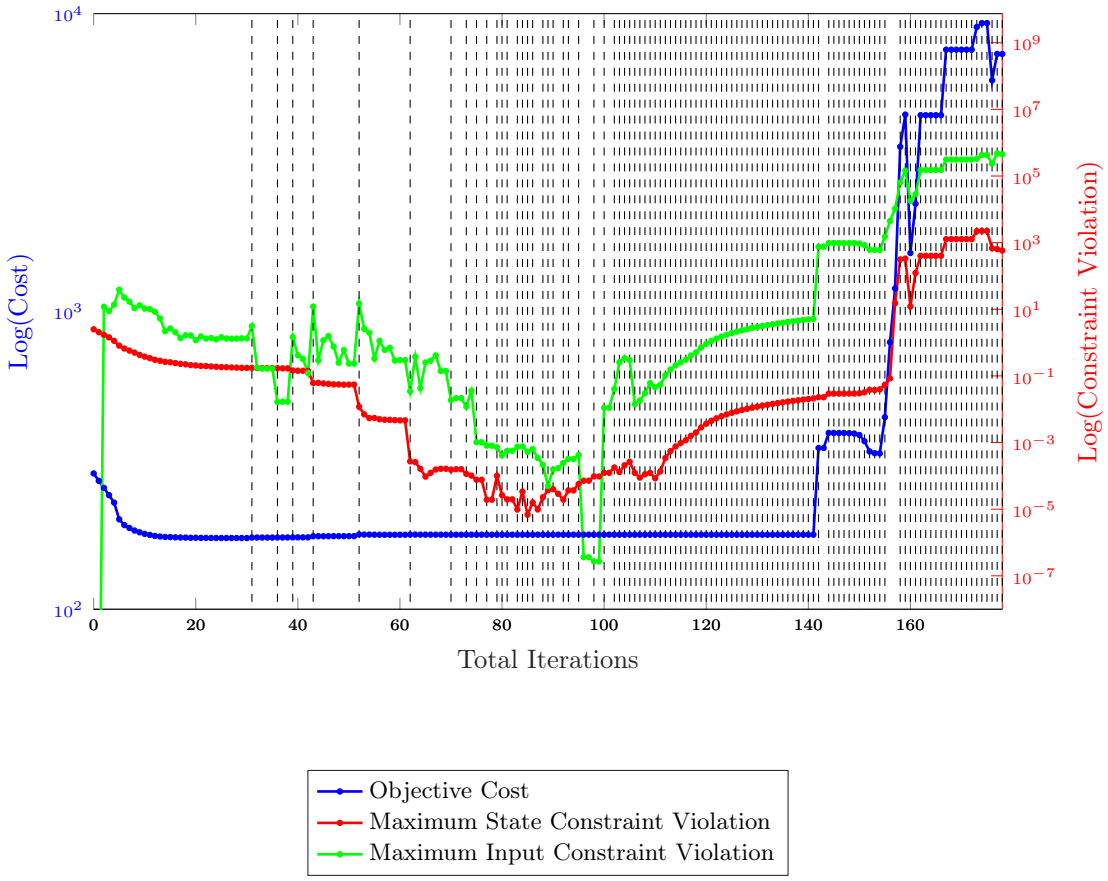

Figure 3.3: *Cost and constraint violation per iteration for iLQR-A failing on the inverted pendulum.*

UDP-A found a lower final cost. In the medium-precision setting, both augmented Lagrangian methods succeeded, while the penalty methods failed to achieve state constraint tolerance and exited after reaching 100 major iterations. In the high-precision setting, the CUDP algorithm (UDP-A) was able to find a feasible solution while all other algorithms failed. This experiment and those that follow show that, for loose constraint tolerances, the use of both iLQR and penalty methods may be sufficient, but the CUDP algorithm is superior when high-precision is desired.

Low precision: $\max(c) < 1e^{-2}$, initial $\phi = 1e^{-1}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	254	1006.1	$2.3e^{-4}$	$\mathbf{5.8e^{-7}}$	$1e^{7}$	$1e^{7}$
UDP-P	229	1022.4	$\mathbf{1.2e^{-4}}$	$1.1e^{-6}$	$1e^{8}$	$1e^{8}$
iLQR-A	**93**	1001.0	$8.2e^{-3}$	$1.8e^{-3}$	$\mathbf{1e^{5}}$	$\mathbf{1e^{3}}$
UDP-A	140	**999.6**	$8.8e^{-3}$	$4.8e^{-4}$	$\mathbf{1e^{5}}$	$\mathbf{1e^{3}}$

Medium precision: $\max(c) < 1e^{-4}$, initial $\phi = 1e^{-2}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	231	1011.9	$1.7e^{-4}$	$1.8e^{-4}$	$1e^{11}$	$1e^{11}$
UDP-P	258	1054.2	$4.1e^{-4}$	$3.4e^{-8}$	$1e^{16}$	$1e^{16}$
iLQR-A	**98**	1001.4	$2.4e^{-5}$	$7.7e^{-5}$	$1e^{7}$	$1e^{8}$
UDP-A	126	**999.8**	$\mathbf{9.0^{e-6}}$	$\mathbf{3.7e^{-5}}$	$\mathbf{1e^{6}}$	$\mathbf{1e^{4}}$

High precision: $\max(c) < 5e^{-7}$, initial $\phi = 5e^{-3}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-A	179	1001.7	$3.7e^{-5}$	$5.0e^{-5}$	$1e^{17}$	$1e^{13}$
UDP-A	**117**	**999.8**	$\mathbf{4.6e^{-8}}$	$\mathbf{1.3e^{-7}}$	$\mathbf{1e^{8}}$	$\mathbf{1e^{7}}$

Table 3.1: *CUDP inverted pendulum optimization results.*

3.3.2 Quadrotor

The objective for this example is to compute a trajectory that flies a quadrotor through a forest while avoiding collisions with trees. The quadrotor has four inputs corresponding to the thrust of each rotor, and twelve states corresponding to the position and Euler angles, along with their respective first derivatives. A quadratic cost function with $Q = 0.1 \times I_{12x12}$, $R = 0.01 \times I_{4x4}$, and $Q_N = 1000 \times I_{12x12}$ was used, with $N = 120$ knot points. All control inputs were initialized to 0, an input constraint of $-10 \leq u \leq 10$ was applied, and a no-collision constraint with the trees and a final state constraint of $x_N = x_g$ were enforced. The algorithms were again tested at three different constraint tolerance values. For the unscented variants, β was set to $1e-4$. Figure 3.4 shows a final trajectory computed by the constrained UDP algorithm.

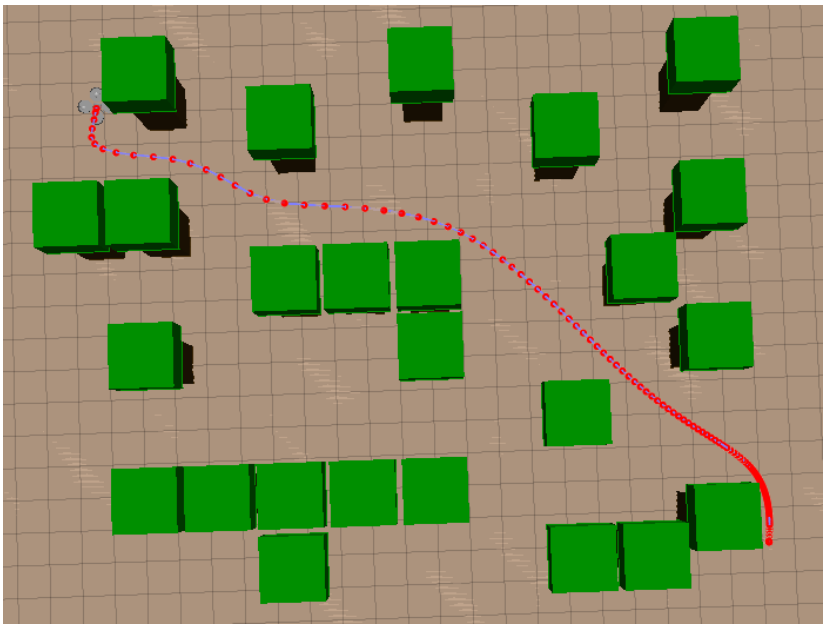

Figure 3.4: *Collision-free quadrotor trajectory computed by CUDP.*

The full results are shown below in Table 3.2. As in the previous example, all of the algorithms converged to the low-precision constraint tolerance, with the augmented Lagrangian variants requiring fewer iterations. For the medium-precision case, both iLQR methods failed to achieve the required tolerance. We hypothesize that the higher-order information provided by the UDP backup procedure is responsible for its improved convergence [29]. For tight constraint tolerances, only UDP-A is able to find a feasible trajectory.

Low precision: $\max(c) < 1e^{-2}$, initial $\phi = 1e^{-1}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	272	758.6	$\mathbf{2.8e^{-4}}$	$\mathbf{2.9e^{-6}}$	$1e^{6}$	$1e^{6}$
UDP-P	125	727.4	$2.0e^{-3}$	$7.8e^{-6}$	$1e^{5}$	$1e^{5}$
iLQR-A	**109**	712.4	$3.7e^{-3}$	$3.2e^{-3}$	$1e^{5}$	$\mathbf{1e^{2}}$
UDP-A	115	**707.3**	$7.6e^{-3}$	$2.3e^{-3}$	$\mathbf{1e^{4}}$	$\mathbf{1e^{2}}$

Medium precision: $\max(c) < 1e^{-4}$, initial $\phi = 1e^{-2}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	223	764.2	$1.4e^{-3}$	$3.2e^{-4}$	$1e^{12}$	$1e^{12}$
UDP-P	**114**	729.6	$\mathbf{2.9e^{-5}}$	$\mathbf{2.2e^{-7}}$	$1e^{10}$	$1e^{10}$
iLQR-A	253	712.6	$3.3e^{-4}$	$2.1e^{-4}$	$1e^{8}$	$1e^{5}$
UDP-A	158	**708.8**	$9.1e^{-5}$	$1.8e^{-6}$	$\mathbf{1e^{8}}$	$\mathbf{1e^{5}}$

High precision: $\max(c) < 1e^{-6}$, initial $\phi = 5e^{-3}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
UDP-P	201	729.6	$2.9e^{-5}$	$2.2e^{-7}$	$1e^{11}$	$1e^{11}$
UDP-A	**149**	**708.8**	$\mathbf{5.3e^{-7}}$	$\mathbf{1.7e^{-7}}$	$\mathbf{1e^{8}}$	$\mathbf{1e^{4}}$

Table 3.2: *CUDP quadrotor optimization results.*

3.3.3 Robotic Arm

The objective for this example is to compute a trajectory for a Kuka LBR IIWA 14 robotic arm to place a rigid object onto a shelf while avoiding an obstacle in its workspace. The state vector is comprised of the 7 joint positions and velocities. We again used a quadratic cost function with $Q = I_{14x14}$, $R = 1e^{-4} \times I_{7x7}$, and $Q_N = 1000 \times I_{14x14}$, with $N = 400$, all controls initialized to 0, input constraints of $-200 \leq u \leq 200\,\text{Nm}$ on each joint, a no collision constraint with the obstacle, and a final state constraint of $x_N = x_g$. The algorithms were again tested at three different constraint tolerance values. For the unscented variants, β was set to $1e^{-4}$. In all cases, the intermediate cost convergence tolerance was set to 10 and the final iteration cost tolerance to $1e^{-2}$. These higher values were used for practicality as the scale of the cost function is larger than the previous examples.

A screen shot of the trajectory computed by the constrained UDP algorithm in the high precision setting is shown in Figure 3.5.

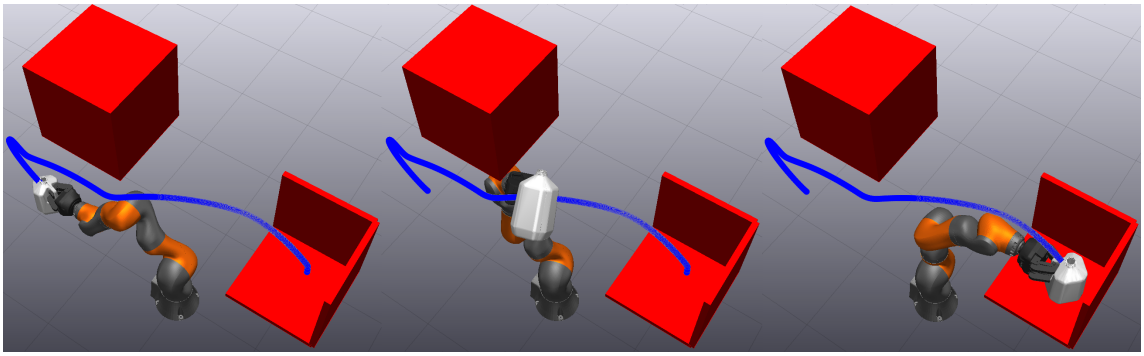

Figure 3.5: *Collision-free Kuka arm trajectory computed by constrained UDP.*

Table 3.3 contains the results of all trials. As before, all of the algorithms handled the low-precision case and produced similar overall costs. The penalty methods converged faster, with lower constraint violation, and larger μ values. Despite their success in the low precision case, however, both penalty methods failed in the medium precision case. Once again, in the high precision case, only the UDP-A method succeeded. This example further demonstrates how penalty and iLQR based methods may succeed with loose constraint tolerances, but the CUDP algorithm can support much more precise constraint satisfaction.

Low precision: $\max(c) < 1e^{-2}$, initial $\phi = 5e^{-1}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	**50**	2161.4	$9.5e^{-3}$	$6.5e^{-8}$	$\mathbf{1e^{6}}$	$1e^{6}$
UDP-P	53	**2155.4**	$8.6e^{-3}$	$\mathbf{5.1e^{-8}}$	$\mathbf{1e^{6}}$	$1e^{6}$
iLQR-A	89	2171.3	$7.1e^{-3}$	$6.1e^{-3}$	$\mathbf{1e^{6}}$	$\mathbf{1e^{2}}$
UDP-A	88	2174.9	$\mathbf{6.3e^{-3}}$	$5.4e^{-3}$	$\mathbf{1e^{6}}$	$1e^{3}$

Medium precision: $\max(c) < 1e^{-3}$, initial $\phi = 1e^{-2}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-P	155	2161.3	$9.6e^{-3}$	$6.2e^{-3}$	$1e^{17}$	$1e^{17}$
UDP-P	146	2226.8	$7.0e^{-3}$	$5.3e^{-3}$	$1e^{12}$	$1e^{12}$
iLQR-A	84	2688.4	$5.7e^{-4}$	$\mathbf{1.3e^{-5}}$	$\mathbf{1e^{8}}$	$\mathbf{1e^{4}}$
UDP-A	**82**	**2674.4**	$\mathbf{4.2e^{-4}}$	$1.8e^{-5}$	$\mathbf{1e^{8}}$	$\mathbf{1e^{4}}$

High precision: $\max(c) < 5e^{-5}$, initial $\phi = 5e^{-3}$

	Iters	Cost	c_x	c_u	μ_x	μ_u
iLQR-A	182	6471.1	$5.9e^{-3}$	$3.8e^{-5}$	$1e^{10}$	$1e^{3}$
UDP-A	**71**	**2674.7**	$\mathbf{2.9e^{-5}}$	$\mathbf{7.1e^{-6}}$	$\mathbf{1e^{10}}$	$\mathbf{1e^{6}}$

Table 3.3: *CUDP Kuka arm optimization results.*

Chapter 4

Parallelizing DDP[1]

Recently, computer scientists have looked toward parallelization as a potential solution to improve the throughput of many algorithms. These explorations have been conducted through the use of multi-core CPUs, massively parallel GPUs, and even through custom hardware implementations on FPGAs and eventually ASICs.

Prior work on parallel nonlinear optimization has broadly focused on exploiting the natural separability of operations performed by the algorithm to achieve *instruction-level parallelism*. For example, if a series of gradients needs to be computed for a list of static variables, that operation can be shifted from a serial loop over them to a parallel computation across them. Alternatively, if block diagonal matrices must be inverted many times by the solver, each of these instructions can be broken down into a parallel solve of several smaller linear systems. These parallelizations might instead be called "low-hanging fruit" parallelizations as they do not change the theoretical properties of the algorithm and therefore can and should be used whenever possible. This research has led to a variety of optimized QP solvers targeting CPUs [49; 50], GPUs [51; 52], and FGPAs [53**?**].

[1] Co-authored with Scott Kuindersma

These approaches have also been used to specifically improve the performance of a subclass of QPs that frequently arise in trajectory optimization problems on multi-threaded CPUs [54; 55; 56; 57] and GPUs [58; 59]. Antony and Grant [60] used GPUs to additionally exploit the inherent parallelism in the "next iteration setup" step of DDP.

In contrast to instruction-level parallelism, *algorithm-level parallelism* changes the underlying algorithm to create more opportunities for simultaneous execution of instructions. In the field of trajectory optimization, this approach was first explored by Bock and Plitt [61] and then Betts and Huffman [27], and has inspired a variety of "multiple shooting" methods [62; 63]. Recently, this approach been used to parallelize both an SQP algorithm [64]; as well as the forward [65; 66] and backward passes [17] of the iLQR algorithm. Experimental results from using these parallel iLQR variants on multi-core CPUs represent the current state of the art for real-time robotic motion planning. That said, these changes have various tradeoffs that we will further explore through our experiments in Chapter 5.

Therefore, while DDP is an inherently serial algorithm, in that it performs forward and backward sweeps along the discretized trajectory, there are still many opportunities to leverage parallel computations to improve the speed of the algorithm. In the remainder of this chapter we build on the literature to summarize the various parallelizations that can be made to DDP and present a unified parallel DDP algorithm.

4.1 Instruction Level Parallelizations for DDP

As shown in Algorithm 1, the forward pass is accepted when the new total cost satisfies the line search criteria. Therefore, after or during forward integration, the cost at each state must be evaluated. Since the cost is additive as shown in Equation 2.2, it can be computed fully in parallel following forward integration. The cost can then be summed up in $\log(N)$ operations using a parallel reduction operator as shown in Figure 4.1.

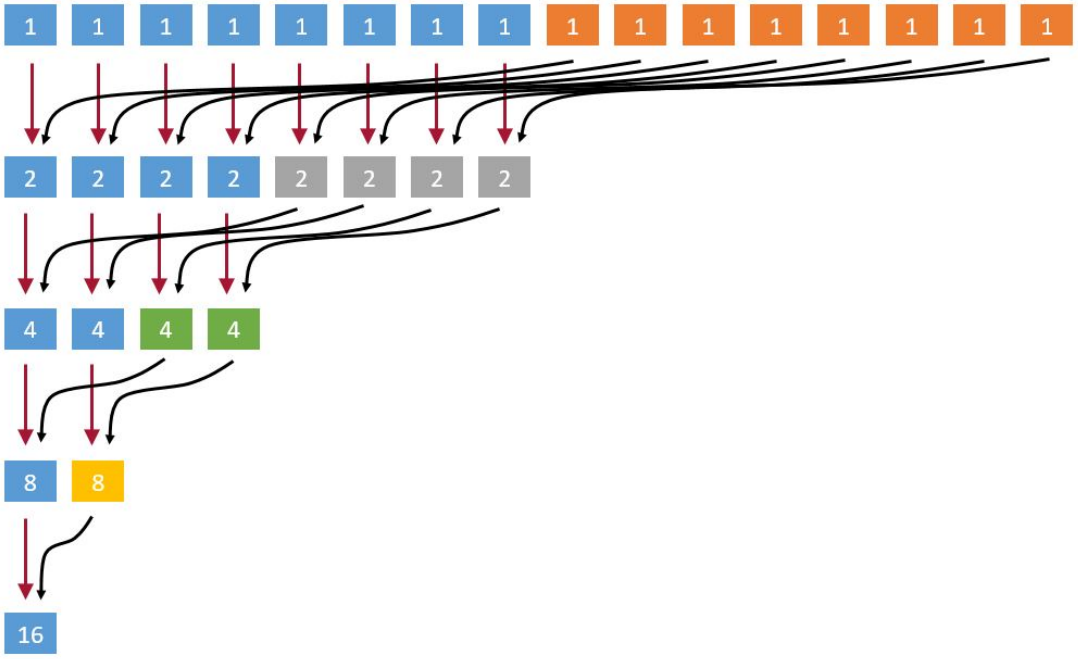

Figure 4.1: *Graphical representation of a parallel reduction.*

In addition, instead of computing the line search during the forward pass by sequentially reducing α, we can compute all resulting forward simulations for a set of possible α values in parallel. Furthermore, if all possible simulations are computed in parallel, then the algorithm could select the "best" trajectory across all options for α, rather than the first option that shows improvement.

In order to develop a well performing notion of "best" we start by using the improved line search criteria as described by Tassa in his thesis [32]. This uses an improved version of the expected cost reduction (Equation 2.9),

$$\delta V^*(\alpha) = -\alpha \kappa^T H_u + \frac{\alpha^2}{2} \kappa^T H_{uu} \kappa, \tag{4.1}$$

and accepts an iterate if the ratio of the expected reduction to the actual reduction,

$$z = \left(J - \tilde{J}\right) / \delta V^*(\alpha), \tag{4.2}$$

falls in the range

$$0 < c_1 < z < c_2 < \infty. \tag{4.3}$$

As Tassa explains, "[t]he unintuitive upper limit c_2 is used because sometimes the improvement is due to the initial trajectory being particularly bad, and the new trajectory has jumped from one bad region of state-space to another slightly-less-bad region, often introducing undesired features and landing in a local minimum. This situation might be likened to encountering a vertical drop when trying to get down a mountain. Jumping off might decrease one's altitude quickly in short run, but might be a bad idea with regards to reaching the bottom." Building on this line of thinking, the lower limit c_1 is used because steps that far under-perform their expected reduction indicate that their quadratic approximation of the cost-to-go was a bad approximation. Since the current model therefore cannot describe the resulting area of state-space well, these regions could also be bad parts of the state-space with undesired features. Therefore, by using Tassa's line search criteria, we exclude potentially bad trajectories, and can then safely take the lowest cost trajectory that satisfies both criteria as our best trajectory.

Finally in Algorithm 1, we also show the computation of all quadratizations of the cost function and dynamics (the "next iteration setup" step) as occurring in one step following the forward pass. As mentioned earlier, this is already parallelized in many implementations of DDP used for online robotic motion planning. This is one of the more expensive steps, and being able to efficiently parallelize this computation is critical for fast performance of the final implementation.

4.2 Algorithmic Level Parallelism for DDP

At its core, the serial nature of DDP stems from the fact that in the forward and backward passes information is propagated through all of the timesteps *in order* allowing it to solve many small problems at each timestep without being myopic. Therefore, in order to create more parallelism and to avoid the $O(N)$ steps in the forward and backward passes, we need to delay and modify this information flow while still preventing long term myopic behavior. In general we do this by breaking the trajectory of N states into equally sized and spaced blocks of M states which we operate on in parallel while leveraging boundedly stale information and global consensus operations to ensure correct long term behavior.

4.2.1 Backward Pass

We do this in the backward pass by leveraging the Stale Synchronous Parallel (SSP) model of computation, which has seen recent success in the machine learning community [67; 68]. The main idea behind SSP computation is that for an iterative algorithm, if the staleness of the data reaching parallel computation nodes is bounded, then the algorithm can theoretically still converge to the same solution as a non-stale implementation. This will often increase total work while improving throughput.

In the particular example of the backward pass of DDP, we break the N timesteps into M_b equally spaced parallel blocks of size $N_b = N/M_b$. In parallel, we can then compute the CTG within each block by passing information serially backwards in time, as is done in standard DDP. At each iteration we pass the information from the beginning of one block to the end of the adjacent block. We can thus ensure that CTG information is at worst case $(M_b - 1)$ iterations stale between the first and last block as shown in Figure 4.2.

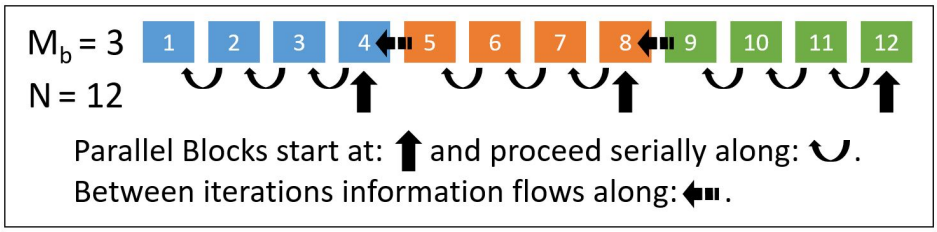

Figure 4.2: *Graphical representation of the backward pass algorithmic level parallelizations.*

Farshidian et. al. [17] note that this approach is somewhat naive and may fail if the trajectory in the next iterate is far enough away from the previous iterate as the stale CTG approximations from the previous iterate are defined in relative coordinates and are only valid locally. Therefore, they propose making an adjustment to the previous iterates CTG to re-center the quadratic approximation around the current iterate. Formally, a coordinate transformation function $f(\cdot)$ needs to be applied to points in iterate $i+1$ such that:

$$V^i\left(f\left(x^{i+1}\right)\right) = V^i\left(x^i\right). \tag{4.4}$$

For notational simplicity we define $V^{i+1}(\cdot) \equiv V^i(f(\cdot))$. We can then simply take iterate i's quadratic CTG approximation $\frac{1}{2}x^{i^T}V_{xx}^i x^i + x^{i^T}V_x^i$, and perform a linear coordinate transformation to re-center it about $x^{i+1} = x^i + \delta x$. After algebraic simplifications,[2] the quadratic term remains the same, while the linear term is updated:

$$V_x^{i+1} = V_x^i + V_{xx}^i(x^{i+1} - x^i). \tag{4.5}$$

This process is shown graphically in Figure 4.3.

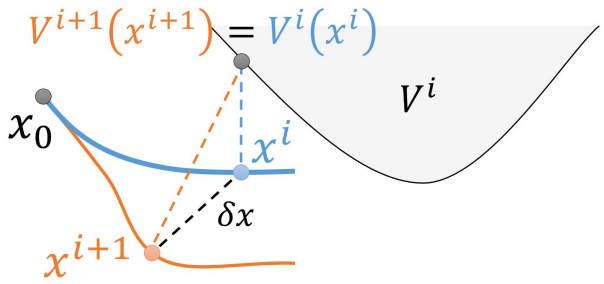

Figure 4.3: *Graphical representation of the linear coordinate transformation.*

[2] $\frac{1}{2}\left(x^i + \delta x\right)^T V_{xx}^i \left(x^i + \delta x\right) + \left(x^i + \delta x\right)^T V_x^i \quad \rightarrow \quad \frac{1}{2}x^{i^T}V_{xx}^i x^i + x^{i^T}\left(V_x^i + \delta x^T V_{xx}^i\right) + C$

While, in theory, this parallel process will still ultimately converge, in practice, some forward passes will fail to find a solution due to the information in the backward pass blocks being "too stale." When this occurs we want to re-run the backwards pass with a better current estimate of the CTG at the start of each block. We do this by using the previous failed iterate's CTG approximation and passing it back along the blocks as if it succeeded since it contains less stale information than the CTG approximation from the previous successful iterate. However, since we know that this estimate caused the previous forward pass to fail, we want to encourage the algorithm to stay closer to the previous successful state trajectory and take a smaller step in order to help the next forward pass succeed. We therefore update our regularization scheme and instead of regularizing the control by adding ρI_m to Q_{uu} from Equation (2.7), we follow Tassa again and add a state regularization term ρI_n to V'_{xx} in the computation of Q_{uu} and Q_{xu} as follows:

$$\begin{aligned} Q_{uu} &= \ell_{uu} + f_u^T \left(V'_{xx} + \rho I_n\right) f_u + V'_x \cdot f_{uu} \\ Q_{xu} &= \ell_{xu} + f_x^T \left(V'_{xx} + \rho I_n\right) f_u + V'_x \cdot f_{xu}. \end{aligned} \tag{4.6}$$

This also leads to an updated quadratic model of V (Equation 2.8):

$$\begin{aligned} V_x &= Q_x - K^T Q_{uu} \kappa - Q_{xu} \kappa - K^T Q_u \\ V_{xx} &= Q_{xx} - K^T Q_{uu} K - Q_{xu} K - K^T Q_{ux}. \end{aligned} \tag{4.7}$$

Together this improved regularization and the passing back of the failed iterate's CTG approximation allows for information to propagate and for safer, smaller steps to be taken, which improves the robustness of the algorithm.

It is also important to note that the forward pass can also fail in practice because the new trajectory moved too far from the previous trajectory, rendering the controls at later time steps sub-optimal. In order to address this issue, most standard DDP implementations increase their regularization parameter when the forward pass fails. Therefore, this approach attempts to solve both potential causes of a forward pass failure at the same time.

4.2.2 Forward Pass

In order to avoid the $O(N)$ serial forward simulation Giftthaler et. al. [65] introduce what they call Gauss-Newton Multiple Shooting which adapts iLQR for multiple shooting. It consists of a fast consensus forward sweep with linearized dynamics followed by a multiple shooting forward simulation from M_f equally spaced states of length $N_f = N/M_f$ timesteps as shown in Figure 4.4.

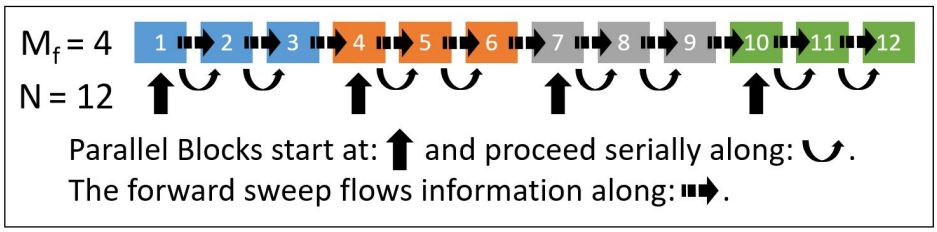

Figure 4.4: *Graphical representation of the forward pass algorithmic level parallelizations.*

While each of the M_f simulation blocks can be done completely in parallel, this will lead to defects d between the edges of each block which need to be driven down to 0 at convergence of the algorithm. Therefore, this defect d must also be taken into account in the approximation of the CTG as it changes the one step dynamics at the end of each block to:

$$x_{k+1} = \begin{cases} f(x_k, u_k) - d_k & k \text{ is the end of one block} \\ f(x_k, u_k) & \text{otherwise.} \end{cases} \tag{4.8}$$

This can be collapsed down to one function by setting $d_k = 0$ whenever it is not the end of one block. Therefore, the non-linear dynamics (Equation 2.1) can be re-written as:

$$x_{k+1} = f(x_k, u_k) - d_k. \tag{4.9}$$

Flowing this change through Equations 2.3-2.6 will result in a modified version of $Q(\delta x, \delta u)$, denoted as $\bar{Q}$, where the gradients are updated as follows:

$$\begin{aligned} \bar{Q}_x &= Q_x + f_x^T V'_{xx} d \\ \bar{Q}_u &= Q_u + f_u^T V'_{xx} d. \end{aligned} \tag{4.10}$$

We now also update our line search criteria to include the following acceptance criteria:

$$0 < \max_k ||d_k||_1 < c_3 < \infty. \tag{4.11}$$

The additional upper limit, c_3, excludes any trajectories that have large defects between the multiple shooting segments, as these trajectories represent an artificial mathematical reduction in cost that is infeasible in practice.

Finally, in order to update the starts of each block of states and to avoid myopic behavior in the blocks, a consensus forward sweep is performed. This is done by integrating the state trajectory using linearized dynamics (where $A = f_x$ and $B = f_u$) and feedback controls computed for the current state trajectory. Like in the original forward pass, a line search parameter, α, is needed to ensure a cost decrease. Therefore, the state at timestep $k+1$ and iteration $i+1$ is updated as follows:

$$x_{k+1}^{i+1} = x_{k+1}^i + \left(A_k^i + B_k^i K_k^i\right)\left(x_k^{i+1} - x_k^i\right) + \alpha B_k^i \kappa_k^i + d_k^i. \tag{4.12}$$

Thus, the forward pass from the standard DDP/iLQR algorithm now becomes the serial forward sweep followed by a parallel forward simulation on each of the M_f blocks. The whole forward-backward process is then repeated until convergence, just like standard DDP/iLQR.

4.3 Final Parallel Algorithm

In the end we combine the instruction level parallelizations, M_f multiple shooting intervals, and M_b blocks for the backward pass. This results in a complete algorithm that can adapt its level of parallelism depending on choices for M_f and M_b as shown in Algorithm 3.

Algorithm 3: *Parallel DDP*

1: Initialize the algorithm and load in initial trajectories
2: **while** cost not converged **do**
3: **for** all M_b blocks b **do** in parallel
4: **for** $k = b_0 : b_{N_b}$ **do**
5: d_k, (2.6), (4.6), (4.10) $\rightarrow \bar{Q}^k$
6: **if** $\bar{Q}^k_{uu}$ is invertible **then**
7: (2.7) $\rightarrow K_k, \kappa_k$
8: (2.8) $\rightarrow V_k$ and derivatives
9: **else**
10: Increase ρ go to line 3
11: **end if**
12: **end for**
13: **end for**
14: **for** all $\alpha[i]$ **do** in parallel
15: $\tilde{x}_0[i] = x_0$
16: **if** $M_f > 1$ **then**
17: **for** $k = 0 : N-1$ **do**
18: $\tilde{x}_k[i], x_k, K_k, \kappa_k, d_k$, (4.12) $\rightarrow \tilde{x}_{k+1}[i]$
19: **end for**
20: **end if**
21: **for** all M_f blocks b **do** in parallel
22: **for** $k = b_0 : b_{N_f} - 1$ **do**
23: $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$
24: $\tilde{x}_{k+1}[i] = f(\tilde{x}_k[i], \tilde{u}_k[i])$
25: $\tilde{d}_k[i] = 0$
26: **end for**
27: $k = b_{N_f}$
28: **if** $k < N$ **then**
29: $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$
30: $\tilde{d}_k[i] = x_{k+1} - f(\tilde{x}_k[i], \tilde{u}_k[i])$
31: **end if**
32: **end for**
33: $\tilde{X}[i], \tilde{U}[i]$, (2.2), (4.2) $\rightarrow \tilde{J}[i], \tilde{z}[i]$
34: **end for**
35: $i^* \leftarrow \arg\min_i \tilde{J}[i]$ s.t. $\tilde{z}[i], \tilde{d}[i]$ satisfy (4.3), (4.11)
36: **if** $i^* \neq \emptyset$ **then**
37: $X, U, d \leftarrow \tilde{X}[i^*], \tilde{U}[i^*], \tilde{d}[i^*]$
38: **else**
39: Increase ρ go to line 3
40: **end if**
41: Quadratize the cost at X, U
42: Quadratize the dynamics at X, U
43: **end while**

Backward Pass
Forward Sweep
Forward Simulation
Forward Pass
Next Iteration Setup

Chapter 5

Using Parallel DDP for Fast Robotic Motion Planning[1]

Our implementations were designed to target modern multi-core CPUs and GPUs in order to take advantage of their respective parallel processing power. At a high level, a multi-core CPU is simply a handful of modern complex CPUs that are designed to work together, often on different tasks, leveraging the multiple-instruction-multiple-data (MIMD) computing model. In contrast, a GPU is a much larger set of very simple processors, optimized for parallel computations of the same task, leveraging the single-instruction-multiple-data (SIMD) computing model. Therefore, as compared to a CPU processor, each GPU processor has many more arithmetic logic units (ALUs), but reduced control logic and a smaller cache memory (see Figure 5.1). In this work we specifically targeted NVIDIA GPUs by using the CUDA extensions to C++ and the NVCC compiler to take advantage of the increased general purpose computation abilities that NVIDIA has heavily invested in recently.

[1] Co-authored with Scott Kuindersma

Figure 5.1: *High level architecture differences between a single CPU and GPU processor [2].*

5.1 CUDA Background

Before discussing the GPU implementation, it is important to understand the target programming model, CUDA. CUDA is built around leverage *host* (CPU) and *device* (GPU) memory and code. It is very important to only try to access memory in the host or device by host or device functions respectively or a SEGFAULT will occur.

Global functions can be launched in parallel on the GPU through the use of a special syntax (`myFunc«<a,b,c,d»>(args)`) and are called *kernels*. In that syntax, `a` specifies how many blocks to launch each containing `b` threads. Each block is guaranteed to run all of its threads on the same processor but the order of the blocks launching and completing is not guaranteed. On most modern GPUs, those threads run in *warps* of 32 threads. It is very important that each of the threads in the same warp runs the same code and does not branch as branches will be run serially due to the SIMD architecture of modern GPUs. Also, memory reads/writes to *global memory* (RAM) need to be *coalesced* (sequential) and aligned, as global memory is also read and written to in chunks. Thus, coalesced reads/writes will greatly reduce the amount of I/O time transferring data from global memory.[2]

[2]While this penalty has been greatly reduced on the newest GPUs due to sophisticated engineering by GPU manufacturers, it is still best practice.

Each processor has its own dedicated L1 cache which is split into standard L1 cache memory and *shared memory*, which is managed by the programmer and can be accessed by all threads in the same block. Shared memory can be allocated statically in the kernel or dynamically through the `c` parameter on kernel launch. Using shared memory can be as fast as using registers if done properly, but if multiple threads try to read from the same shared memory address at the same time there will be a *bank conflict* and those reads will be serialized. There is also a limited amount of shared memory and if more memory is requested than available, the compiler will either throw an error, or store some of the data in global memory. Therefore, smart use of shared memory will greatly speed up the kernel and poor use may slow it down.

Each kernel launch will naturally run sequentially. However, if two kernels are doing completely unrelated operations they can be launched in parallel through the use of *streams* which are denoted on launch by parameter `d`. There are also thread safe atomic operations and a number of different ways to synchronize threads in a block, in a stream, or on the entire device.

It is important to note that kernel launches suffer overhead and so combining code into fewer kernels will improve the speed of the code. This is true for both custom user kernels and kernels from NVIDIA and third party libraries designed to do common computations (e.g., cuBLAS and cuSOLVE for matrix math [69]). Finally, when targeting GPUs it is important to remember that they are slower than CPUs on straight line serial code which makes them a poor choice for serial computations. For more information on CUDA and its programming model we suggest reading the NVIDIA CUDA programming guide [2] and/or CUDA by Example [70].

5.2 C++ Threading Background

As with CUDA, before discussing the multi-core CPU implementation, it is important to understand the C++ multi-threading programming model. For the purposes of this implementation we focus on the C++ thread library that provides a higher level and operating system agnostic interface to the standard UNIX POSIX thread library and can be simply included using `#include<thread>`. Not only is this library easy to include, but it also is easy to use as any function can be called in its own thread by invoking `myThread = thread(myFunc, args)`.

Each thread will run with its own set of registers and stack memory, and context switches between threads are scheduled asynchronously by the operating system. In fact, context switches can occur in the middle of a threads execution. Due to the increased cache size available on CPU processors, there is no need to manually manage a notion of shared memory. Instead, smart access patterns to memory addresses will allow the CPU to automatically cache often used data. Again, coalesced and aligned memory access and lack of branching is strongly suggested, as the CPU will often try to predict the next instructions and data and will be much faster if this pre-fetching is exploited.

It is important to note that thread launches do incur some overhead and since there are in general only a handful of cores on most systems, CPU multi-threading scales best to the tens of threads. As on a GPU, there are notions of atomic operations and synchronization through the `thread.join()` call, which, importantly, must be called on each thread before the process finishes or an error will occur.[3] Since the thread library is part of the standard library for C++11, there are a host of books and resources online that explain it in greater detail including *C++ Concurrency in Action* [71].

[3]Threads can also be detached from the current process and left to run independently through the `thread.detach()` call.

5.3 Implementation Details

Our GPU implementation was designed to target modern NVIDIA GPUs. We minimized memory bandwidth delay by doing most computations on the GPU, which removes the need for most memory transfers between the device and host. The CPU is instead in charge of high level serial control flow for kernel launches, such as the outer `while` loop, as well as the `if` and `else` statements that decide whether to accept or reject new trajectories. On the GPU we also condense as many computations onto as few kernels as possible, a process known as *kernel fusion* [72], to minimize kernel launch overhead. We also make heavy use of streams and asynchronous memory transfers to increase throughput wherever possible. For example, by running the quadratization of the cost and dynamics in separate streams, the throughput time for the next iteration setup was much closer to the maximum of the running times for those steps than the sum. Similarly, we also update all control trajectories for the parallel line search to the next iterate in parallel with the backward pass computations as both steps are independent and need to simply finish before the next forward pass.

This approach led to very fast code and required a lot of custom kernel design. This occurred because we found that the heavily optimized general purpose GPU matrix math libraries (e.g., cuBLAS) are optimized for very large matrix operations. DDP algorithms, on the other hand, require many sets of serial small matrix operations that suffer overhead from these implementations. Therefore, we implemented simple custom kernels which keep the data in shared memory throughout these computations and provide a large speedup. We further optimized our code by precomputing terms during parallel operations. For example, during the backward pass, A, B, K, and κ were already loaded into shared memory and thus computing $A + BK$ and $B\kappa$ only added a small overhead to the paralellizable backward pass, while greatly reducing the time for the serial forward sweep.

In our multi-threaded CPU implementation we began by reusing the same baseline code to leverage the work that went into using best practices with memory access patterns and

overlapping computations. However, with the decreased number of threads for ideal performance, we needed to introduce loops in each threaded computation to minimize the number of thread launches while still computing all of the necessary results. In general, we tried to limit the number of threads to a small multiple of the number of CPU cores in order to maximize CPU usage while minimizing thread launch overhead. Thus, we did not parallelize the forward pass across all line search parameters as we often had fewer cores than the number of multiple shooting intervals. Therefore, we instead used the standard serial loop over line search parameters.

In both implementations, we made use of analytical dynamics and cost functions and their respective gradients and Hessians to maximize performance and to ensure that the control flow for computations on the GPU was simple and mainly required only ALU operations. For our simple systems we were able to hand derive these functions. For more complex systems we intended to use the RobCoGen software package to generate optimized C++ code for the forward dynamics and coordinate transforms from simple description files [73]. However, while the package is optimized and very fast when used by CPUs, it relies heavily on object oriented programming. Dynamically allocating those objects in CUDA kernels overflows the available shared memory and registers on each CUDA core, and loading those objects in from global memory proved to be very slow. Therefore, for the purposes of this work, we were limited to hand derived dynamics, and thus to simpler systems. Furthermore, we found that our initial MATLAB generated analytical derivatives included many repeated terms and needed to be hand optimized to meet our performance criteria. For example, for the quadrotor, our optimizations we were able to reduce the computation time by an order of magnitude.

In general, we found that the more we could special case the algorithm and make decisions at compile time, the faster we could make the algorithm. This raises an inherent tension in high performance software design as for practical real world use we need to provide some flexibility to the user, but still have very fast code.

5.4 Examples

In this section, three numerical examples are provided to demonstrate the performance of our parallel implementation of iLQR. We ran our experiments on a laptop with 16GB of RAM, a 2.8GHz quad-core Intel Core i7-7700HQ CPU, and a 1708MHz 1280 Cuda Core NVIDIA GeForce GTX 1060 6GB GDDR5 GPU. In all of our experiments we seeded the optimizer with a poor initialization. For the simple and inverted pendulums, all controls were initialized to a very small constant value,[4] while for the quadrotor, we initialized the algorithm with a simple hover. All of the experiments used a 3rd-order Runge-Kutta method for integration over a 4 second trajectory with $N = 128$ knot points. All of the solvers used the same scheme for updating ρ and the same set of line search options for α. For more consistent experimental comparisons, we also enforce staleness of the CTG during the backward pass and of the starting state of each forward simulation even if the asynchronous launches of the blocks would have allowed for some updated information to flow.[5]

We report results in terms of time per iteration and cost reduction as a function of the number of iterations. Time per iteration is a particularly useful metric when comparing our results to the current state of the art, as in those results, the solver is often run a fixed, small number of iterations at every control step. Our target performance was to match those solvers which operate at 40-200Hz (5-25ms per iteration of iLQR) [17; 6; 14; 15; 74; 16]. We note that many of these results are achieved on systems whose complexity is of the quadrotor or greater. Therefore, we only compare our timing from our quadrotor experiment to the literature. To ensure our results were representative for each experiment, we ran 100 trials with noise $\sim \mathcal{N}(0, \sigma^2)$ applied to the velocities of the initial trajectory.

[4] With a single backward pass we could initialize the controls with $\mathbf{0}$. However, in the parallel case we need to provide some control from which feedback gains can be computed on the first pass, otherwise the information from the final state may not propagate down the backward pass blocks fast enough for the algorithm to progress at all in the first few iterations. This often leads to a premature exit for "convergence." We ended up initializing all controls to 0.01 Nm.

[5] In general this simply degrades performance slightly and therefore our results provide a lower bound on throughput depending on the scheduling of threads at runtime and the hardware on which they are run.

5.4.1 Simple Pendulum

We first consider the classic pendulum system and swing-up task in simulation. We define the state vector to be $x = [\theta, \dot{\theta}]^T$ where θ is the angle of the pendulum measured from the downward equilibrium. The initial state is $x_0 = [0, 0]^T$, the stable downward equilibrium, and the goal state is $x_g = [\pi, 0]^T$, the unstable upward equilibrium. As in Section 3.3, we use a quadratic cost function where $Q = \texttt{diag}(0.1, 0.01)$, $R = 0.01$, and $Q_N = 1000 \times I_{2x2}$. We solve the problem on using both our GPU and multi-core CPU implementations with $M_b = M_f = M = 1, 2, 4, 8, 16, 32, 64$ and $\sigma = 0.01$.

The median cost per iteration across all levels of parallelism is shown in Figure 5.2. As we increase M, and add delays into the information propagating through the forward and backward passes, the algorithm requires more iterations to converge. Furthermore, for $M < 8$ the "best line search" option employed by the GPU leads to faster convergence than the standard method and is otherwise equivalent.

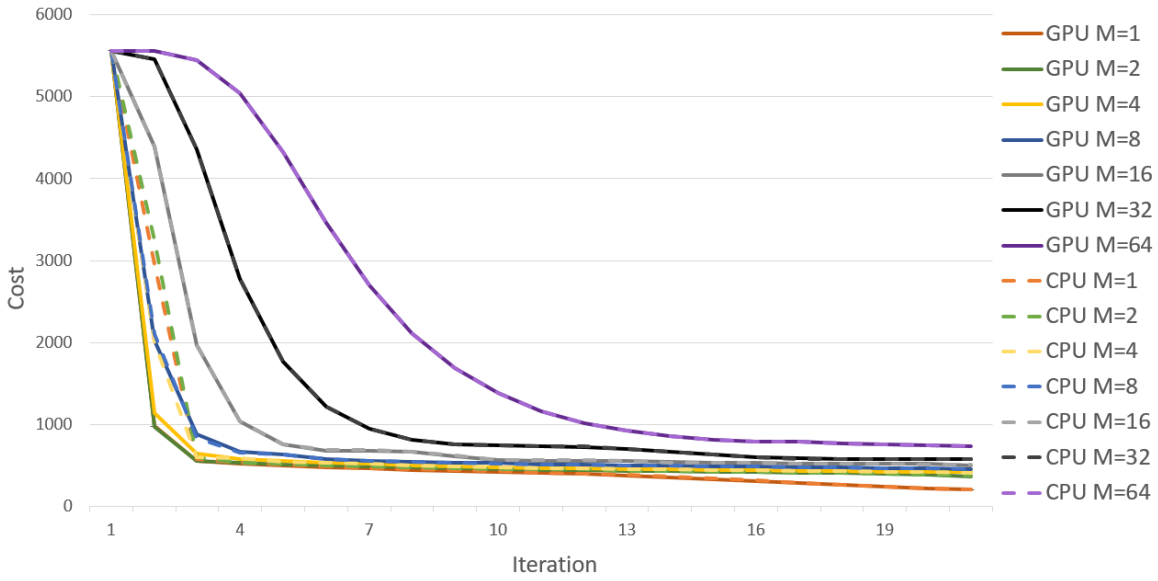

Figure 5.2: *Total cost per iteration for the simple pendulum swing up.*

The median time per iteration for each of the parallelization options for both implementations is shown in Figure 5.3. We report median time per iteration as the CPU implementation

will execute the forward simulation and sweep a variable number of times depending on the depth of the line search, unlike the GPU implementation which executes the full line search in parallel at every iteration.

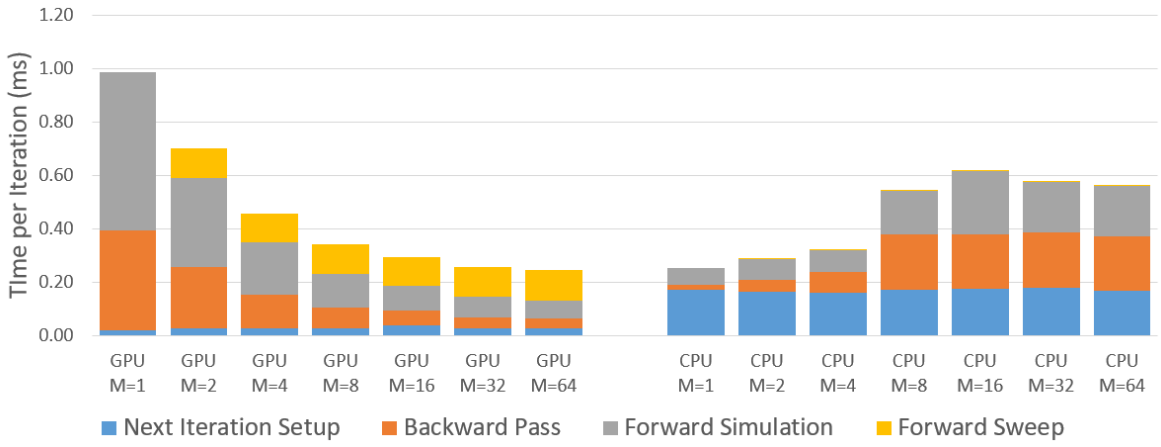

Figure 5.3: *Median time per iteration time for the simple pendulum swing up.*

With regard to the GPU implementation, we find that by parallelizing the forward simulation and backward pass we can drastically reduce the computation time per iteration. That said, there are diminishing returns to parallelization. First, kernel launch overhead begins to dominate the running time as parallelization is increased. Second, since the next iteration setup is always fully parallelized and the forward sweep does not parallelize at all, the running times for both steps do not decrease as we increase M. This leads to their time beginning to dominate the total time. In fact, starting at $M = 16$, the forward sweep is the most expensive step in the algorithm. This is a prime example of Amdahl's law [75], which shows diminishing returns to parallelism based on the percentage of the code that can be parallelized.

In regards to the multi-core CPU implementation, we find that the swing up task for the pendulum problem is too computationally easy to warrant parallelism on the CPU. A single thread can solve the problem fast enough that thread launch overhead is highly problematic. In addition, we run out of cores for parallel computation quite quickly, and need to serialize any further potentially parallel computations, this leads to a litany of context switches and attempts at "hyperthreading" (pretending one core is two and running two threads on that

core almost simultaneous), which ultimately slows down the code. This is most evident in the massive slowdown in the forward simulation and backward pass as M grows from 1 to 8. Once $M > 4$ we have run out of cores and therefore any future potentially parallel computations are serialized and thus the slowdown plateaus.

From the data we can also see that the CPU is able to leverage its higher clock rate to compute the serial forward sweep faster than the GPU, but the GPU is able to leverage its increased number of cores to compute the parallel next iteration setup step faster than the CPU, as we would expect.

5.4.2 Inverted Pendulum

We then again consider the classic inverted-pendulum system and swing-up task in simulation as done in Section 3.3.1 (this time without a final state constraint). We set $Q = \texttt{blkdiag}(0.01 \times I_{2x2}, 0.001 \times I_{2x2})$, $R = 0.0001$, $Q_N = 10000 \times I_{4x4}$. We solve the problem again using $M = 1, 2, 4, 8, 16, 32, 64$ and $\sigma = 0.001$.

Like in the previous experiment, we find that for larger M, the algorithm requires more iterations to converge (see Figure 5.4). This time we find that the "best line search" option employed by the GPU leads to faster convergence across all levels of parallelism. The one exception is the $M = 64$ case in which it not only converged slower than the standard method but also caused many trials to converge to other less optimal local minima.

The GPU implementation shows similar decreases in time per iteration due to increased parallelism as it did during the simple pendulum experiment (see Figure 5.5). However, the more complex and computationally expensive system dynamics, lead to increased time per iteration, as compared to the simple pendulum experiment, across all choices for M. Interestingly, the always fully parallel "next iteration setp" takes a very similar amount of time to compute as it did in the simple pendulum experiment. We hypothesize that the combination of using an optimized dynamics gradient function and computing all states fully in parallel

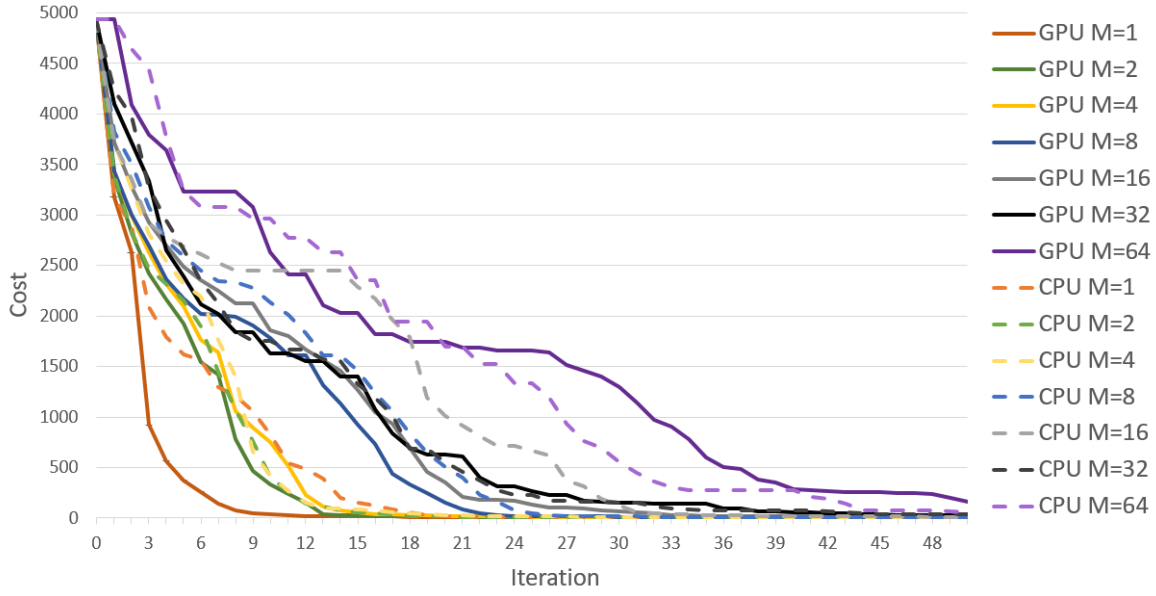

Figure 5.4: *Total cost per iteration for the inverted pendulum swing up.*

allows for the overall computation to take a similar amount of time.

The CPU implementation again struggles to parallelize well. We again believe that this example is too simple for the CPU to warrant effective multi-core parallelization. Finally, again, we find that the next iteration setup step parallelizes better on the GPU and the serial forward sweep runs faster on the CPU.

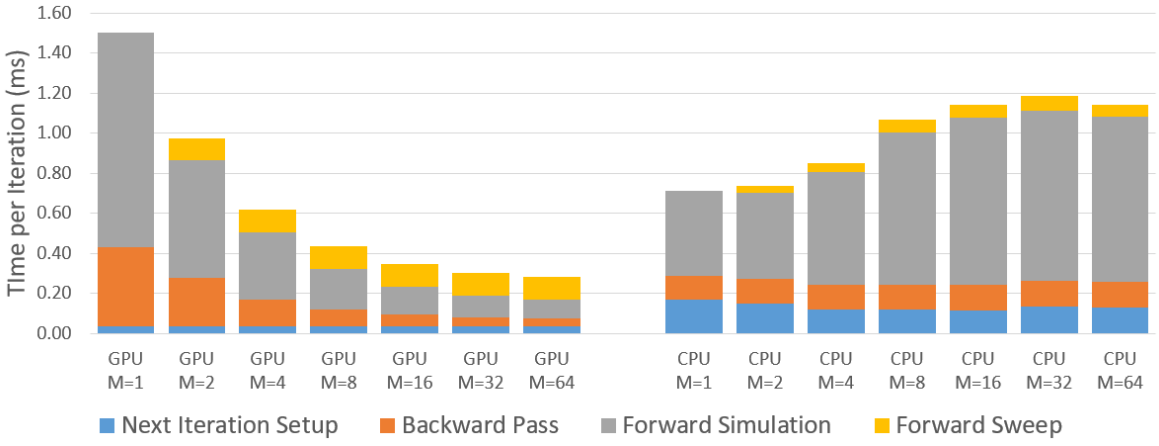

Figure 5.5: *Median time per iteration time for the inverted pendulum swing up.*

5.4.3 Quadrotor

We finally consider again the Quadrotor system and flight task from Section 3.3.2. (this time without constraints). We set $Q = \texttt{blkdiag}(0.01 \times I_{3x3}, 0.001 \times I_{3x3}, 2.0 \times I_{6x6})$, $R = 5.0 \times I_{4x4}$, $Q_N = 1000 \times I_{12x12}$. We again solve the problem using $M = 1, 2, 4, 8, 16, 32, 64$ and $\sigma = 0.01$. As mentioned previously, we initialize the algorithm with a torque input that counters gravity and allows the quadrotor to hover.

As with the other experiments, figure 5.6 clearly shows the median cost per iteration increasing as M increases. At the same time, the "best line search" option only shows improvement over the standard method in the $M = 64$ case and is otherwise equivalent. It is also important to note that all trials converged in the $M < 64$ cases but in the $M = 64$ case only $\sim 60\%$ of both the GPU and CPU trials converged.

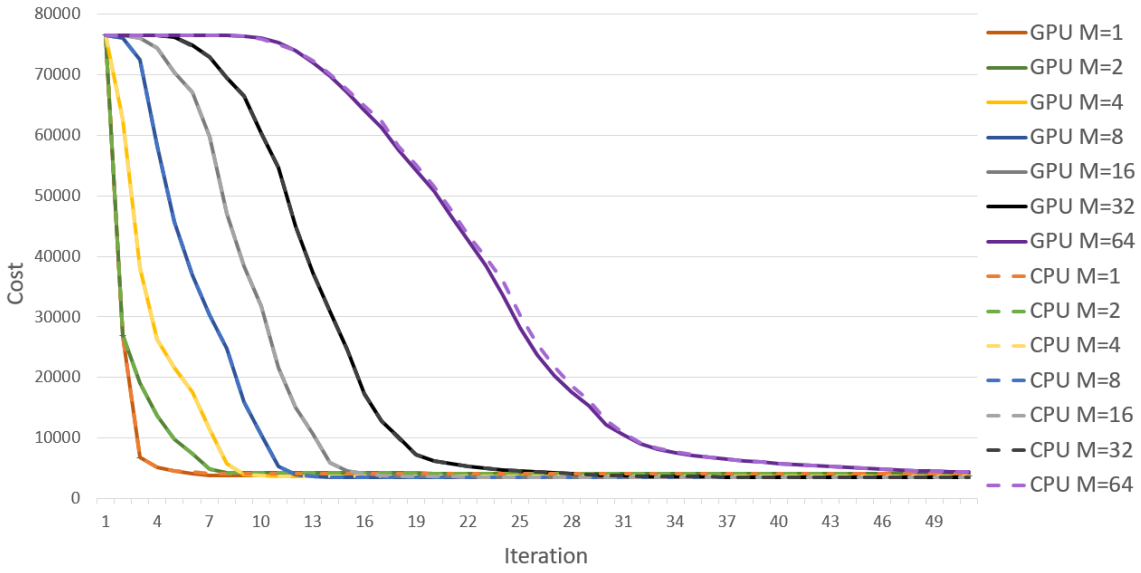

Figure 5.6: *Total cost per iteration for the quadrotor flight.*

Also as with the previous experiments, and shown in Figure 5.7, we find that the GPU shows strong improvements in results for increased levels of parallelism. We also see that in this more complex example, the CPU backward pass and forward simulation also get faster as M grows from $M = 1$ to $M = 4$. Once we have $M > 4$ we exceed the number of cores on the system

and incur serialization penalties such that the speed slows and plateaus. Again, we find that the next iteration setup step parallelizes better on the GPU and the serial forward sweep runs faster on the CPU. Finally, with time per iteration at 5ms for GPU $M = 1$ and under 3.5ms for all other GPU and CPU cases, our implementation is able to beat state-of-the-art reported rates of 5-25ms [74] on a similar system.

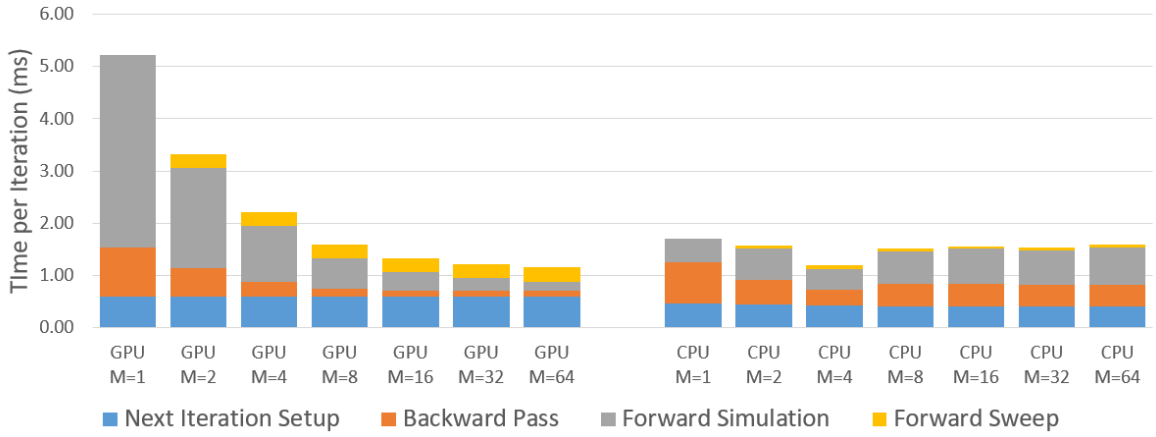

Figure 5.7: *Median time per iteration time for the quadrotor flight.*

Overall, across all of our experiments, we see that today's multi-core CPUs are incredibly powerful and compute serial and mildly parallel objects very quickly. However, we also see that for truly parallel computations, GPUs can provide greatly increased performance even on very simple tasks.

Chapter 6

Conclusion

In this thesis, we addressed some of the practical challenges of using DDP in dynamic environments. First we presented the constrained UDP algorithm, a computationally efficient DDP variant capable of satisfying nonlinear state and input constraints with high accuracy through the use of an augmented Lagrangian. Second, we presented an implementation of the iLQR algorithm for multi-core CPU and GPU usage that can solve iLQR iterations at speeds comparable to the state of the art, revealing the power of and challenges with parallel computations.

Several directions for future research remain. For CUDP, combining multiple constraint-handling approaches may prove beneficial. For example, box constraints on inputs are captured using cost terms in our experiments. It would be straightforward to instead use existing QP techniques in the backward pass to compute input constraints [37; 38], while using augmented Lagrangian terms for state constraints. An empirical comparison including barrier methods in the DDP setting would also be interesting.

Also, as highlighted in prior work on UDP [29], the sigma point scaling parameter, β, must be chosen ahead of time for each example. Automatic approaches to setting β remain an interesting open problem. Similarly, more work is needed to determine optimal—or even good

suboptimal—schedules for ϕ and μ. We believe our results could be significantly improved if more effort was spent exploring update schemes.

We also hope that parallelization of trajectory optimization algorithms will continue to increase their speed and performance, but note that for widespread applicability, software needs to be developed that generates GPU optimized dynamics functions for arbitrary systems. We were able to hand optimize dynamics for our simple systems, but don't see that scaling well to more complex systems. In addition, adding support for constraints at similar speeds must be completed before these algorithms can be used on physical hardware in constrained environments.

In future work, we plan to develop parallel implementations of CUDP, augmented Lagrangian constrained iLQR, as well as other constrained DDP variants, targeting online MPC applications. We also plan to run experiments on hardware systems to validate our current simulation results both on more complex systems and to prove their validity in the real world. We also hope to analyze various flavors of QP solvers with various constraint formulations to see if alternate non-linear program solvers will be more amenable to parallelization and therefore may ultimately provide higher performance.

We hope that this work will eventually lead to that elusive Holy Grail of whole-body motion generation—real time optimal trajectory generation.

References

[1] R. Tedrake, "Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832).."

[2] NVIDIA, *NVIDIA CUDA C Programming Guide*. version 9.1 ed.

[3] J. T. Betts, *Practical Methods for Optimal Control Using Nonlinear Programming*, vol. 3 of *Advances in Design and Control*. Society for Industrial and Applied Mathematics (SIAM).

[4] K. D. Mombaur, "Using optimization to create self-stable human-like running," vol. 27, no. 3, pp. 321–330.

[5] M. Posa, M. Tobenkin, and R. Tedrake, "Lyapunov analysis of rigid body systems with impacts and friction via sums-of-squares," in *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, pp. 63–72.

[6] 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*.

[7] W. Xi and C. D. Remy, "Optimal Gaits and Motions for Legged Robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[8] M. Posa, S. Kuindersma, and R. Tedrake, "Optimization and stabilization of trajectories for constrained dynamical systems," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pp. 1366–1373, IEEE.

[9] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP Algorithm for Large-scale Constrained Optimization," vol. 47, no. 1, pp. 99–131.

[10] A. Wächter and L. T. Biegler, "On the Implementation of an Interior-point Filter Line-search Algorithm for Large-scale Nonlinear Programming," vol. 106, no. 1, pp. 25–57.

[11] D. Q. Mayne, "A second-order gradient method of optimizing non- linear discrete time systems," vol. 3, p. 8595.

[12] D. H. Jacobson and D. Q. Mayne, *Differential Dynamic Programming*. Elsevier.

[13] W. Li and E. Todorov, "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems," in *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics*.

[14] T. Erez, K. Lowrey, Y. Tassa, V. Kumar, S. Kolev, and E. Todorov, "An integrated system for real-time model predictive control of humanoid robots," in *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pp. 292–299.

[15] J. Koenemann, A. Del Prete, Y. Tassa, E. Todorov, O. Stasse, M. Bennewitz, and N. Mansard, "Whole-body Model-Predictive Control applied to the HRP-2 Humanoid," in *Proceedings of the IEEERAS Conference on Intelligent Robots*.

[16] M. Neunert, F. Farshidian, A. W. Winkler, and J. Buchli, "Trajectory Optimization Through Contacts and Automatic Gait Discovery for Quadrupeds," vol. 2, no. 3, pp. 1502–1509.

[17] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli, "Real-time motion planning of legged robots: A model predictive control approach," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pp. 577–584.

[18] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376, ACM.

[19] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "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*, ASPLOS XV, pp. 205–218, ACM.

[20] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278.

[21] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12.

[22] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. Konidaris, "Robot Motion Planning on a Chip," in *Robotics: Science and Systems*.

[23] 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*, pp. 4090–4097.

[24] 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)*, pp. 3572–3577.

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

[26] B. Ichter, E. Schmerling, A. a Agha-mohammadi, and M. Pavone, "Real-time stochastic kinodynamic motion planning via multiobjective search on GPUs," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5019–5026.

[27] J. T. Betts and W. P. Huffman, "Trajectory optimization on a parallel processor," vol. 14, no. 2, pp. 431–439.

[28] S. J. Julier and J. K. Uhlmann, "Unscented filtering and nonlinear estimation," vol. 92, no. 3, pp. 401–422.

[29] Z. Manchester and S. Kuindersma, "Derivative-Free Trajectory Optimization with Unscented Dynamic Programming," in *Proceedings of the 55th Conference on Decision and Control (CDC)*.

[30] R. Bellman, *Dynamic Programming*. Dover.

[31] L.-z. Liao and C. A. Shoemaker, "Advantages of Differential Dynamic Programming Over Newton's Method for Discrete-time Optimal Control Problems."

[32] Y. Tassa, "Theory and Implementation of Biomimetic Motor Controllers."

[33] J. R. Dormand and P. J. Prince, "A family of embedded Runge-Kutta formulae," vol. 6, no. 1, pp. 19–26.

[34] E. Todorov, "A convex, smooth and invertible contact model for trajectory optimization," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

[35] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, 2nd ed.

[36] B. Plancher, Z. Manchester, and S. Kuindersma, "Constrained Unscented Dynamic Programming," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[37] J. F. O. D. O. Pantoja and D. Q. Mayne, "A sequential quadratic programming algorithm for discrete optimal control problems with control inequality constraints," in *Proceedings of the 28th IEEE Conference on Decision and Control,*, pp. 353–357 vol.1.

[38] Y. Tassa, T. Erez, and E. Todorov, "Control-Limited Differential Dynamic Programming," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

[39] F. Farshidian, M. Neunert, A. W. Winkler, G. Rey, and J. Buchli, "An Efficient Optimal Planning and Control Framework For Quadrupedal Locomotion,"

[40] Z. Xie, C. K. Liu, and K. Hauser, "Differential dynamic programming with nonlinear constraints," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 695–702.

[41] A. Forsgren, P. Gill, and M. Wright, "Interior Methods for Nonlinear Optimization," vol. 44, no. 4, pp. 525–597.

[42] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," vol. 33, no. 9, pp. 1251–1270.

[43] J. van den Berg, "Iterated LQR smoothing for locally-optimal feedback control of systems with non-linear dynamics and non-quadratic cost," in *American Control Conference (ACC), 2014*, pp. 1912–1918.

[44] M. Toussaint, "A Novel Augmented Lagrangian Approach for Inequalities and Convergent Any-Time Non-Central Updates,"

[45] T. C. Lin and J. S. Arora, "Differential dynamic programming technique for constrained optimal control," vol. 9, no. 1, pp. 27–40.

[46] G. Lantoine and R. P. Russell, "A Hybrid Differential Dynamic Programming Algorithm for Constrained Optimal Control Problems. Part 1: Theory," vol. 154, no. 2, pp. 382–417.

[47] D. P. Bertsekas, "Multiplier Methods: A Survey," vol. 12, no. 2, pp. 133–145.

[48] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems."

[49] D. Leineweber, "Efficient reduced SQP methods for the optimization of chemical processes described by large sparse DAE models."

[50] D. P. Word, J. Kang, J. Akesson, and C. D. Laird, "Efficient Parallel Solution of Large-scale Nonlinear Dynamic Optimization Problems," vol. 59, no. 3, pp. 667–688.

[51] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pp. 917–924, ACM.

[52] L. Yu, A. Goldsmith, and S. Di Cairano, "Efficient Convex Optimization on GPUs for Embedded Model Predictive Control," in *Proceedings of the General Purpose GPUs*, GPGPU-10, pp. 12–21, ACM.

[53] K. V. Ling, S. P. Yue, and J. M. Maciejowski, "A FPGA implementation of model predictive control," in *2006 American Control Conference*, pp. 6 pp.–.

[54] H. Joachim Ferreau, A. Kozma, and M. Diehl, "A Parallel Active-Set Strategy to Solve Sparse Parametric Quadratic Programs arising in MPC," vol. 45, no. 17, pp. 74–79.

[55] J. V. Frasch, S. Sager, and M. Diehl, "A parallel quadratic programming method for dynamic optimization problems," vol. 7, no. 3, pp. 289–329.

[56] G. Frison, "Algorithms and Methods for Fast Model Predictive Control."

[57] R. Quirynen, "Numerical simulation methods for embedded optimization."

[58] Y. Huang, K. V. Ling, and S. See, "Solving Quadratic Programming Problems on Graphics Processing Unit,"

[59] D.-K. Phung, B. Hérissé, J. Marzat, and S. Bertrand, "Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit," vol. 50, no. 1, pp. 11883–11888.

[60] T. Antony and M. J. Grant, "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures," vol. 54, no. 5, pp. 1081–1091.

[61] H. G. Bock and K.-J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," vol. 17, no. 2, pp. 1603–1608.

[62] D. M. Garza, "Application of automatic differentiation to trajectory optimization via direct multiple shooting."

[63] M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber, "Fast Direct Multiple Shooting Algorithms for Optimal Robot Control," in *Fast Motions in Biomechanics and Robotics*, pp. 65–93, Springer, Berlin, Heidelberg.

[64] D. Kouzoupis, R. Quirynen, B. Houska, and M. Diehl, "A Block Based ALADIN Scheme for Highly Parallelizable Direct Optimal Control," in *Proceedings of the American Control Conference*.

[65] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl, "A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control,"

[66] E. Pellegrini and R. P. Russell, "A Multiple-Shooting Differential Dynamic Programming Algorithm," in *AAS/AIAA Space Flight Mechanics Meeting*.

[67] M. Zinkevich, J. Langford, and A. J. Smola, "Slow Learners are Fast," in *Advances in Neural Information Processing Systems 22* (Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, eds.), pp. 2331–2339, Curran Associates, Inc.

[68] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," vol. 2013, pp. 1223–1231.

[69] NVIDIA, "GPU-Accelerated Libraries for Computing."

[70] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed.

[71] A. Williams, *C++ Concurrency in Action : Practical Multithreading*. Manning.

[72] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing CUDA Code By Kernel Fusion—Application on BLAS," vol. 71, no. 10, pp. 3934–3957.

[73] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, "RobCoGen: A code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages," vol. 7, no. 1, pp. 36–54.

[74] M. Neunert, C. de Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear Model Predictive Control for unified trajectory optimization and tracking," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1398–1404.

[75] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, ACM.