GPU Acceleration for Real-time, Whole-body, Nonlinear Model Predictive Control

A dissertation presented

by

Brian Kyle Plancher

to

the Harvard John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Engineering Sciences

Harvard University

Cambridge, Massachusetts

May 2022

Dissertation Advisors: *Author:*

Professor Vijay Janapa Reddi   **Brian Kyle Plancher**

Professor Scott Kuindersma

GPU Acceleration for Real-time, Whole-body, Nonlinear Model Predictive Control

Abstract

Whole-body, nonlinear model predictive control (MPC) refers to the control strategy where a robot's state and input trajectories are continually optimized over a finite time horizon while taking into account the robot's full nonlinear dynamics. This has been referred to as the "Holy Grail" of robot motion planning and control, as it can enable robots to dynamically compute optimal trajectories and adapt to changes in their environment. Unfortunately, the underlying trajectory optimization algorithms traditionally used to solve these problems are computationally expensive and often too slow to run in real-time. Compounding this issue, the impending end of Moore's Law and the end of Dennard Scaling have led to a utilization wall that limits the performance a single CPU chip can deliver, requiring computer scientists to look beyond the CPU to exploit large-scale parallelism available on alternative computing platforms such as GPUs. This dissertation address these challenges by exposing, analyzing, and leveraging the structured sparsity and parallelism patterns found in the numerical optimization and rigid body dynamics algorithms commonly used for whole-body, nonlinear MPC. Through careful algorithmic refactoring and re-design, this work exploits these patterns to enable real-time MPC performance through GPU-acceleration. It also validates the feasibility of this approach in the presence of model discrepancies and communication delays between the robot and GPU by deploying the resulting implementations onto a physical manipulator arm. Overall, this dissertation finds that GPU acceleration can provide nearly order-of-magnitude speedups, and open-sources its implementations to aid the wider robotics community in accelerating both robotics computations and application development timelines.

Contents

v

List of Tables

List of Figures

Acknowledgments

This work would not have been possible without. . .

. . . **my advisor**, Professor Vijay Janapa Reddi. Thank you for taking a chance on a roboticist interested in exploring the intersection of robotics and computer architecture, for helping me develop my passion for teaching by providing me with amazing opportunities to teach lectures and co-develop courses, both online and in-person, and for being an all around amazing advisor, spending far too much time helping me wordsmith my faculty applications, and constantly pushing me to grow as a researcher, mentor, teacher, advisor and person.

. . . **my co-advisor**, Professor Scott Kuindersma. Thank you for taking a chance on a management consultant trying to get into robotics, for continuing to support me even after you had left academia, and for initially enabling, and supporting me in pursuing my passion for teaching.

. . . **my co-author and unofficial co-advisor**, Professor Zachary Manchester. Thank you for not only supporting me extensively in the first couple of months of my masters, when I was trying to get up to speed, but also for continuing to support me and welcome me into your robotics labs remotely, enabling me to stay deeply plugged into robotics.

. . . **my engineering community**. To my undergraduate thesis advisor, Professor Greg Morrisett, thank you for also taking a chance supporting a former economics major who switched late into computer science. To Professors Gu-Yeon Wei and David Brooks, thank you for all of your guidance and support and for originally showing me that there is more to fast computing than good software. To all of my collaborators – with a special shout-out to Sabrina M. Neuman – as well as all of the members of the Harvard Agile Robotics, Edge Computing, and VLSIArch Labs, and the Stanford/CMU Robotic Exploration Lab, thank you for all of your useful feedback, insights, help, and friendship throughout this process.

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

. . . **my family and friends**. Thank you for your unwavering, friendship, love, and support.

To mom, dad, pop, ah-ah, grammy, gramps, Jamie, and Megan, who have always been there to nurture, support, and guide me on the journey of life.

To Tess and Alvin, who light up every morning with a smile, a wiggle, and a bark, and always help remind me what really matters.

To Annie, whose unconditional love and support makes me the best me that I can be.

Chapter 1

Introduction

Whole-body, nonlinear model predictive control (MPC) refers to the control strategy where a robot's state and input trajectories are continually optimized over a finite time horizon while taking into account the robot's full nonlinear dynamics. This has been referred to as the "Holy Grail" of robot motion planning and control [4], as it can enable robots to dynamically compute optimal trajectories and adapt to changes in their environment. Unfortunately, this approach suffers from two fundamental constraints that limit its current application. First, this approach requires the solution of non-convex optimization problems, which often results in locally optimal solutions, and makes the algorithms extremely sensitive to hyperparameter choices. Second, even if these non-convexity issues can be overcome, the trajectory optimization algorithms traditionally used to solve these problems are computationally expensive and often too slow to run in real-time [3; 4].

Compounding this second issue, the impending end of Moore's Law and the end of Dennard Scaling have led to a utilization wall that limits the performance a single CPU chip can deliver [5; 6]. As such, for computationally bounded algorithms, like whole-body, nonlinear MPC, computer scientists have had to look beyond the CPU to exploit large-scale parallelism available on alternative computing platforms such as GPUs. This has led to, e.g., significant

performance improvements in the field of machine learning [7].

This dissertation address these computational challenges by exposing, analyzing, and leveraging the structured sparsity and parallelism patterns found in algorithms commonly used for whole-body, nonlinear MPC. Through careful algorithmic refactoring and re-design, my co-authors and I exploit these computational patterns in numerical optimization and rigid body dynamics algorithms to enable real-time MPC performance through GPU-acceleration. We also validate the feasibility of this approach in the presence of model discrepancies and communication delays between the robot and GPU by deploying the resulting open-source algorithms and implementations onto a physical manipulator arm. We also note that these algorithms and implementations can be used to accelerate additional state-of-the-art motion planning and control approaches that leverage offline MPC as a sub-routine to efficiently develop trajectory libraries and learned policies [8; 9; 10; 11; 12; 13; 14].

Throughout this dissertation, we use the process of Hardware-Software Co-Design [15] to ensure that our algorithms and implementations take advantage of the strengths and minimize the weaknesses of the CPUs and GPUs on which they are run. As such, we begin, in Chapter 2, by defining the three step algorithmic structure and underlying mathematical computations of the trajectory optimization algorithms that are commonly used for whole-body, nonlinear MPC: 1) approximate the cost and dynamics functions; 2) compute an update to the controls (and states); and 3) apply the update while ensuring improvement. We also, in Chapter 3, provide an introduction to multi-core CPU and GPU hardware, revealing the ability for the GPU to provide additional performance through large-scale parallelism at the cost of requiring more structured and naturally parallel algorithms. Combined, these two chapters set the stage for the algorithmic refactoring and re-design done in the later chapters to enable real-time performance through GPU acceleration.

We then expose the opportunities and limitations of both instruction-level and algorithm-level parallelism in the context of Differential Dynamic Programming (DDP) algorithms and demonstrate that higher MPC control rates generally lead to better real-world tracking performance.

This is done through the development of a GPU-optimized Parallel DDP solver with faster than state-of-the-art performance which is used to perform the first MPC experiments on physical robot hardware where the solver is running entirely on the GPU. These results, detailed in Chapter 4, show that not only is there a need for even faster trajectory optimization solvers to improve real-world performance, but also that performance tradeoffs exist between convergence behavior and time per iteration as the degree of algorithm-level parallelism increases, limiting the improvements from parallelism for DDP based algorithms.

We then demonstrate how algorithmic refactoring can expose hardware compatible computational patterns in rigid body dynamics algorithms and unlock the ability for the wider robotics community to run whole-body nonlinear MPC entirely on the GPU with their own custom robots. This is done through the development of GRiD, the first open-source GPU-accelerated spatial-algebra-based [16] rigid body dynamics library with analytical gradients. As detailed in Chapter 5, GRiD generates refactored code that outperforms state-of-the-art, multi-threaded, code-generated CPU libraries by as much as 7.2x when performing multiple computations of rigid body dynamics and their gradients and maintains as much as a 2.5x speedup when accounting for the I/O communication overhead between the CPU and GPU.

We then develop a sparsity exploiting and parallel friendly symmetric stair preconditioner for optimal control problems, leading to improvements in condition number, spectral radius, and iterations-to-converge on standard trajectory optimization problems. As detailed in Chapter 6, this not only reduces the average number of inner conjugate gradient (CG) iterations by a factor of up to 3.1x during trajectory optimization solves, but also enables the design of a GPU-optimized direct trajectory optimization solver with a structure exploiting parallel CG solver that can be used for GPU-accelerated, whole-body, nonlinear MPC.

Finally, this dissertation concludes in Chapter 7 and introduce opportunities for further acceleration by moving beyond the GPU to more custom parallel hardware architectures. We detail preliminary results indicating that FPGAs and custom ASICs can provide as much as 100x latency speedups over GPUs on rigid body dynamics computations.

Chapter 2

Model Predictive Control Background

In this chapter we provide background information on whole-body, nonlinear model predictive control and the trajectory optimization algorithms commonly used to solve their underlying nonlinear optimization problems at each control step.

2.1 Model Predictive Control (MPC)

Model Predictive Control (MPC) is an optimal control strategy in which the trajectory for a dynamical system is, at each control step, optimized over a finite time horizon. As the duration of the control step is generally shorter than the time horizon, very little of each trajectory is used. Instead a new optimal trajectory is continuously computed, adjusting to changes caused by the robot's interactions with its environment (e.g., wind blowing a quadrotor off course, unmodeled friction in a robot's joints). This process is diagrammed in Figure 2.1 where the execution of the first step of an optimized trajectory results in a deviation from the simulated plan, and the computation of a new optimal trajectory that takes this deviation into account. MPC has historically seen success in both the aerospace and chemical industries [17].
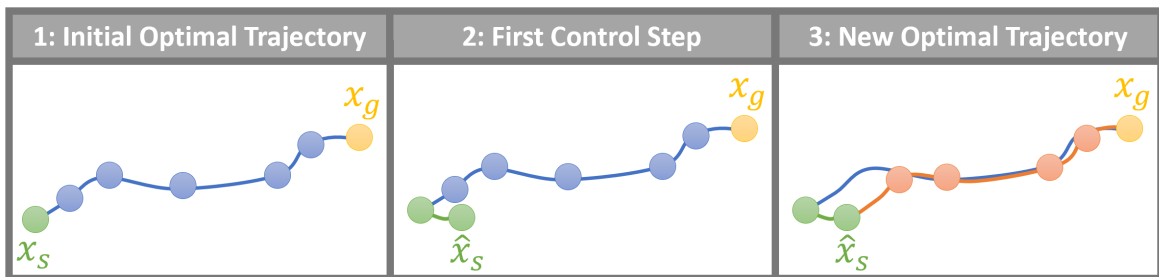
Figure 2.1: *A graphical depiction of MPC. On the left, an initial optimal trajectory shown in blue is computed from a start state, x_s, shown in green to a goal state, x_g, shown in yellow. In the center, the first step in the trajectory is executed on the real system resulting in a new start state that deviates slightly from the second state in the initial optimal trajectory. On the right, a new optimal trajectory shown in orange is constructed from the new start state to the goal state and the process repeats itself.*

Nonlinear Model Predictive Control refers to the case where MPC is applied to a nonlinear dynamical system. In robotics, whole-body, nonlinear MPC refers to the case where the robot's full nonlinear dynamics are used. This is in contrast to approaches that approximate the robot as a simpler system. For example, many successful approaches to bipedal and quadrupedal walking approximate the robot as a single rigid body by leveraging the simple linear inverted pendulum (SLIP) model, the zero-moment point, the hybrid-zero dynamics model, the centroidal dynamics model, or by simply ignoring leg dynamics [18; 19; 20; 21; 22; 23; 24; 25; 26].

2.2   Trajectory Optimization

Trajectory optimization [27], also known as numerical optimal control, is the traditional method used to solve the underlying nonlinear optimization problem at each control step of whole-body, nonlinear MPC. These problems optimize the robot's trajectory by minimizing an additive cost function,

$$J(X, U) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k), \tag{2.1}$$

subject to (nonlinear) constraints

$$c(X, U) \geq 0, \tag{2.2}$$

where $x_k \in \mathbb{R}^n$ are the robot's states and $u_k \in \mathbb{R}^m$ are the robot's control inputs along a discrete time trajectory with N *knot points*. Trajectories are commonly discretized into tens to hundreds of knot points to balance physical realism with computational complexity.

Common (nonlinear) state and input constraints include: joint and torque limits, obstacle avoidance constraints, contact constraints, and dynamics constraints which often leverage a discrete time integrator over a timestep h,

$$x_{k+1} = f(x_k, u_k, h). \tag{2.3}$$

A choice must be made as to the type of integrator to use which trades off integration accuracy and computational complexity. Common integrators, in order of increasing accuracy and complexity, include forward Euler, semi-implicit Euler, Midpoint, and Runge–Kutta methods [28]. Recent work has also explored the use of implicit and variations integrators which suggest improved accuracy for similar computational cost [29; 30; 31; 32].

In this dissertation we focus on the trajectory optimization problem with only dynamics constraints and an initial state constraint, that is,

$$
\begin{aligned}
\min_{X,U} \quad & \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k) \\
\text{s.t.} \quad & x_{k+1} = f(x_k, u_k, h) \;\; \forall k \in [0, N) \\
& x_0 = x_s
\end{aligned}
\tag{2.4}
$$

However, we note that, and detail in Sections 2.3 and 2.4, that algorithms aimed at solving this class of problems can be modified to support additional constraints.

The two most common ways to solve trajectory optimization problem are through direct methods and shooting methods. Direct methods solve the trajectory optimization problem by parameterizing both the state and input trajectories, X, U, and forming a large and sparse nonlinear program which can be solved using an off-the-shelf solver package [33]. In contrast, shooting methods, such as the Differential Dynamic Programming (DDP) algorithm,

6

parameterize only the input trajectory, U, and use Bellman's optimality principle to iteratively solve a sequence of much smaller optimization problems in order to compute the optimal input (and corresponding state) trajectories [34; 35].

Importantly, both of these approaches can be reduced to an iterative three step process:

1. Form an approximation (often through a Taylor expansion) of the cost and constraint functions around the nominal trajectory (X, U).

2. Compute an update that can be applied to the controls (and the states in the case of a direct method).

3. Apply the update, while ensuring descent on the original nonlinear problem, often through the use of a merit-function and a line-search or trust-region.

In both of these cases, Step 1 is dominated by the computation of dynamics (gradients), which consume 30% to 90% of the total computational time of whole-body, nonlinear MPC [1; 2; 3; 4]. Chapter 5 explores ways to accelerate these computations in detail.

Chapters 4 and 6 explore Steps 2 and 3 in more detail through the lens of both shooting and direct methods, as there are a variety of ways to both compute and apply the update, which admit different amounts of natural parallelism. We also introduce both algorithm types in more detail in the remainder of this chapter.

2.3 Differential Dynamic Programming (DDP)

Shooting methods, like the Differential Dynamic Programming (DDP) algorithm, parameterize only the input trajectory, U, and iteratively solve a sequence of much smaller optimization problems in order to compute the optimal input (and corresponding state) trajectories [34; 35]. This iterative process is based on Bellman's principle of optimality [36], which defines the

optimal cost-to-go, $V_k(x)$, by the recurrence relation:

$$V_N(x) = \ell_f(x_N)$$

$$V_k(x) = \min_u \ \ell(x, u) + V_{k+1}(f(x, u)). \tag{2.5}$$

When interpreted as an update procedure, this relationship leads to classical dynamic programming [36]. 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 (2.3).

DDP leverages a second-order Taylor expansion of a *local* approximations to the cost-to-go along a trajectory under perturbations, $\delta x, \delta u$ to avoid the curse of dimensionality:

$$Q(\delta x, \delta u) \approx \frac{1}{2} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_{xx} & Q_{xu} \\ Q_u & Q_{xu}^T & Q_{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}. \tag{2.6}$$

Where the block matrices are computed as:[1]

$$Q_{xx} = \ell_{xx} + f_x^T V_{xx}' f_x + V_x' \cdot f_{xx} \qquad Q_x = \ell_x + f_x^T V_x'.$$

$$Q_{uu} = \ell_{uu} + f_u^T V_{xx}' f_u + V_x' \cdot f_{uu} \qquad Q_u = \ell_u + f_u^T V_x'. \tag{2.7}$$

$$Q_{xu} = \ell_{xu} + f_x^T V_{xx}' f_u + V_x' \cdot f_{xu}.$$

Minimizing equation (2.6) with respect to δu results in the following control correction:

$$\delta u = -Q_{uu}^{-1} \left(Q_{ux} \delta x + Q_u \right) \equiv K \delta x + \kappa, \tag{2.8}$$

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

---

[1]Following the notation used elsewhere [37], 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 and are often omitted, resulting in the iLQR algorithm [38].

substituted back into equation (2.6) to obtain an updated quadratic model of V:

$$
\begin{aligned}
V_x &= Q_x - Q_{xu}\kappa \\
V_{xx} &= Q_{xx} - Q_{xu}K.
\end{aligned}
\tag{2.9}
$$

DDP, like other variants of Newton's method, can achieve quadratic convergence near a local optimum [35; 39]. However, a line search parameter, α, must be added to the forward pass to ensure a satisfactory decrease in cost, and a regularization must be applied to ensure that Q_{uu} in equation (2.8) is invertible [40].

As such, DDP's three step iterative process is:

1. Form a quadratic approximation of the cost function and a linear approximation of the constraints around a nominal trajectory (X, U).

2. Compute the control corrections K and κ along the whole trajectory by performing a backward pass starting at the final state, x_N, by setting $V_N = \ell_f(x_N)$, and iteratively applying Equations 2.6-2.9.

3. Recover the new state trajectory using the update controls through a forward nonlinear simulation pass starting at the initial state x_0, while ensuring descent on the nonlinear problem through the use of a line-search. This computes state $k+1$ for iteration $i+1$ as follows: $x_{k+1}^{i+1} = f(x_k^{i+1}, u_k^i + K_k^{i+1}(x_k^{i+1} - x_k^i) + \alpha\kappa_k^{i+1}, h)$.

For shooting methods, while constraints beyond the standard dynamics constraints cannot be included in the standard formulation, recent work has shown that they can be included by transforming the backward pass into a series of QPs [41; 42; 43; 44; 45; 46] or through augmented Lagrangian approaches that modify the cost function [47; 48].

2.4 Direct Trajectory Optimization

Direct methods solve the trajectory optimization problem by forming a large and sparse nonlinear program which can be solved using a variety of off-the-shelf solver packages. Popular

packages include the Sequential Quadratic Programming (SQP) solver SNOPT [49], and the Interior Point solver IPOPT [50].

While there are a variety of algorithmic approaches used to solve the nonlinear programs resulting from Equation 2.4, most methods can be reduced to the following three step iterative process: [33; 50; 49]:

1. Form a quadratic approximation of the cost function and a linear approximation of the constraints around a nominal trajectory (X, U), resulting in a quadratic program (QP) where $Z = [X, U]$, $G = \nabla^2 J$, $g = \nabla J$, and $C = \nabla c$.

2. Compute the update δZ^* by solving the associated Karush–Kuhn–Tucker (KKT) system:

$$\begin{bmatrix} G & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} -\delta Z \\ \lambda \end{bmatrix} = \begin{bmatrix} g \\ c \end{bmatrix} \tag{2.10}$$

3. Apply the update step, $\delta X^*, \delta U^*$, while ensuring descent on the nonlinear problem through the use of a merit-function and line-search or trust-region.

Therefore, while DDP style methods rely on an iterative forward and backward pass to compute and apply the update, direct methods instead solve the KKT system (Equation 2.10) directly and then apply the update to all of the states and controls at once. As such, these approaches expose more natural parallelism. Direct methods are also more flexible as they can more easily support the inclusion of additional constraints. While there are many different formulations of direct methods, they tend to differ in two main ways: the first is in how they handle additional inequality constraints,[2] and the second is in how they solve the associated KKT system.

---

[2]While we focus on a subset of equality constrained trajectory optimization problems in this dissertation (see Equation 2.4), we include a quick summary of common approaches used to solve trajectory optimization problems with inequality constraints here. While these approaches do require algorithmic modifications, they still reduce to algorithms with similar computational patterns as the algorithms discussed in this dissertation. Sequential Quadratic Programming (SQP) methods solve a sequence of QPs based on estimates of the *active set* of constraints. That is, the set of equality constraints and all inequality constraints upon which the solution resides on the boundary of the inequality, in other words, where $c(Z) = 0$. As the size of the active set is exponential in the number of inequality constraints, these formulations rely on good estimates of the active set to converge quickly. Interior Point methods instead augment inequality constraints with *slack variables*, s, such that the

Importantly, regardless of the ways in which both equality and inequality constraints are handled, all of these methods rely on inner solves of KKT systems. These linear systems can be solved in two main ways. The first is through direct factorization into matrices which make solving the resulting system easier. For example, the LL^T Cholesky Factorization or the LDL^T factorization result in a system with only triangular (and diagonal) matrices which can be efficiently solved through serial back substitution. The second method is through iterative fixed point methods, the most popular of which is the Conjugate Gradient Krylov subspace method. These methods start with a guess of the solution and rely on large (and often sparse) matrix-vector products and vector reductions to repetitively improve the guess, converging to the final solution. For more information on these different methods we suggest reading Chapter 5 of [33] and Chapter 4 of [52].

2.4.1 Merit Functions

As states and controls are both updated simultaneously in direct methods, a merit function is needed to ensure that each step provides a combined improvement to the nonlinear cost and constraints functions.

A common choice for a merit function is the L1 merit function [33],

$$M(Z; \mu) = J(Z) + \mu |c(Z)|. \tag{2.11}$$

When combined with an Armijo line search condition, an update step $\alpha \delta X^*, \alpha \delta U^*$ will be accepted if it satisfies the following, where ω_1, and ω_2 are hyperparameters for the minimum and maximum allowable deviations from the expected reduction [33; 40]:

$$\omega_1 \leq \frac{M(Z + \alpha \delta Z; \mu) - M(Z; \mu)}{\alpha \left(g^T \delta Z - \mu |c|\right)} \leq \omega_2. \tag{2.12}$$

constraints become $c(Z) - s = 0$ and then place log barrier penalties into the cost functions to penalize for large slack values. Finally, as mentioned previously, augmented Lagrangian methods penalize all of the constraints in the cost function by optimizing the augmented Lagrangian $\mathcal{L}_A(Z, \mu, \lambda) = J(Z) + \lambda^T c(Z) + c(Z)^T I_\mu c(Z)$ where I_μ ensures that only the active set is penalized quadratically [51]. Many commercial software packages use a mixture of these methods to leverage the respective strengths. For more information on these different methods we suggest reading Chapters 15-19 of [33].

2.4.2 Schur Complement Direct Trajectory Optimization

One way to solve the KKT system in Equation 2.10 is through the use of the symmetric positive semi-definite *Schur Complement*, S, which can be formed, and then used to solve for the optimal Lagrange multipliers, λ^*, as follows:

$$
\begin{aligned}
S &= -CG^{-1}C^T \\
\gamma &= c - CG^{-1}g \\
S\lambda^* &= \gamma
\end{aligned}
\tag{2.13}
$$

λ^* can then be used to compute the optimal state and control update, δZ^*:

$$
\delta Z^* = G^{-1}\left(g - C^T\lambda^*\right)
\tag{2.14}
$$

In most trajectory optimization scenarios, S and γ are relatively easy to form, and Equation 2.14 is relatively easy to solve, as the cost function is separable across timesteps by construction, resulting in a block-diagonal G matrix. As such, the dominant computational step in this approach is solving $S\lambda^* = \gamma$. As mentioned previously this can be solved either through efficient serial direct factorization approaches (e.g., Cholesky factorization) or through efficient iterative methods (e.g., Conjugate Gradient). As we focus on parallel algorithms in this dissertation, we focus on iterative methods as they traditionally admit more parallel computations.

2.4.3 Krylov Subspace Methods

Krylov subspace methods are iterative methods that can be used to efficiently compute the solution of large linear systems. That is, they solve the problem $S\lambda^* = \gamma$ for a given S and γ by iterative refining an estimate for λ up to some tolerance ϵ. The most popular of these methods is the Conjugate Gradient (CG) method, which is applicable to systems where S is positive semi-definite.

The convergence rate of CG is directly related to the spread of the Eigenvalues of $S \in \mathbb{R}^{n \times n}$, converging faster and avoiding round-off and overflow errors caused by iterative floating point

math when they are clustered and moderate in magnitude. CG is also guaranteed to converge in at most n steps if the spectral radius of S (the maximum absolute magnitude of the eigenvalues of S) is less than one [33; 53].

To improve the performance of these algorithms a preconditioning matrix $\Phi \approx S$ is often applied to instead solve the equivalent problem $\Phi^{-1}S\lambda^* = \Phi^{-1}\gamma$. A good preconditioner is one that is easy to invert and reduces the spread and magnitude of the eigenvalues of $\Phi^{-1}S$.

Finally, we note that the matrix $\Phi^{-1}S$ does not have to be explicitly formed. Instead the Preconditioned CG (PCG) algorithm leverages matrix-vector products with S and Φ^{-1}, as well as vector reductions, both parallel friendly operations (see Algorithm 1).

Algorithm 1: $PCG(S, \gamma, \lambda, \Phi, \epsilon) \to \lambda^*$

1: $r = \gamma - S\lambda$

2: $\tilde{r}, p = \Phi^{-1}r$

3: $\eta = r^T\tilde{r}$

4: **for** iter $i = 1 : \text{max\_iter}$ **do**

5: $\quad \alpha = \eta/(p^T S p)$

6: $\quad r = r - \alpha S p$

7: $\quad \lambda = \lambda + \alpha p$

8: $\quad \tilde{r} = \Phi^{-1}r$

9: $\quad \eta' = r^T\tilde{r}$

10: $\quad$ **if** $\eta' < \epsilon$ **then**

11: $\quad\quad$ return λ

12: $\quad \beta = \eta'/\eta$

13: $\quad p = \tilde{r} + \beta p$

14: $\quad \eta = \eta'$

15: return λ

2.4.4 Parallel Preconditioners

There are many different preconditioners that are optimized for computation on vector or parallel processors. The most popular of these is the Jacobi or Block-Jacobi preconditioner. This sets:

$$\Phi = \mathrm{diag}(S) \qquad \text{or} \qquad \Phi = \text{block-diag}(S). \tag{2.15}$$

Previous GPU based Krylov solvers mainly leveraged these preconditioners [54; 55]. For block banded matrices, alternating and overlapping block preconditioners have also been used in previous work. These methods compute Φ^{-1} as a sum of the inverse of block-diagonal matrices that compose S [56; 57]. Finally, Polynomial splitting preconditioners [57] follow the pattern $S = \Psi - R$ and compute a preconditioner where:

$$S^{-1} \approx \Phi^{-1} = (I + \Psi^{-1}R + (\Psi^{-1}R)^2 \dots)\Psi^{-1}. \tag{2.16}$$

Increasing the degree of the polynomial computes a better approximation of S and improves the convergence rate of the resulting PCG algorithm. However, this requires more computation to compute the preconditioner and also often creates a preconditioner with a larger bandwidth, requiring more memory. For block-banded matrices, like the Schur complement matrix for most trajectory optimization problems, as the values in the true inverse decay exponentially as one moves away from the diagonal [58], this creates a tradeoff between the accuracy and both the memory and computational complexity of the preconditioner.

Chapter 3

Computer Hardware Background

In order for computer programs to run efficiently they need to be designed to take advantage of the strengths and minimize the weaknesses of the computer hardware on which they are run. This process is refereed to as Hardware-Software Co-Design [15]. In this chapter we show why high performance parallel architectures are needed today and then provide an introduction to multi-core CPUs, GPUs, and co-design.

3.1   The Need for Parallelism

The impending end of Moore's Law and the end of Dennard Scaling have led to a utilization wall that limits the performance a single chip can deliver [5; 6]. As shown in Figure 3.1, since about 2005, CPU clock speed has flat-lined and single threaded performance has started to stall. In response, hardware vendors have drastically increased the number of CPU cores on each processor. However, for many cutting edge applications, like machine learning, this has not been enough, and computer scientists have had to look beyond the CPU to exploit large-scale parallelism available on alternative computing platforms such as GPUs [7].

48 Years of Processor Trends
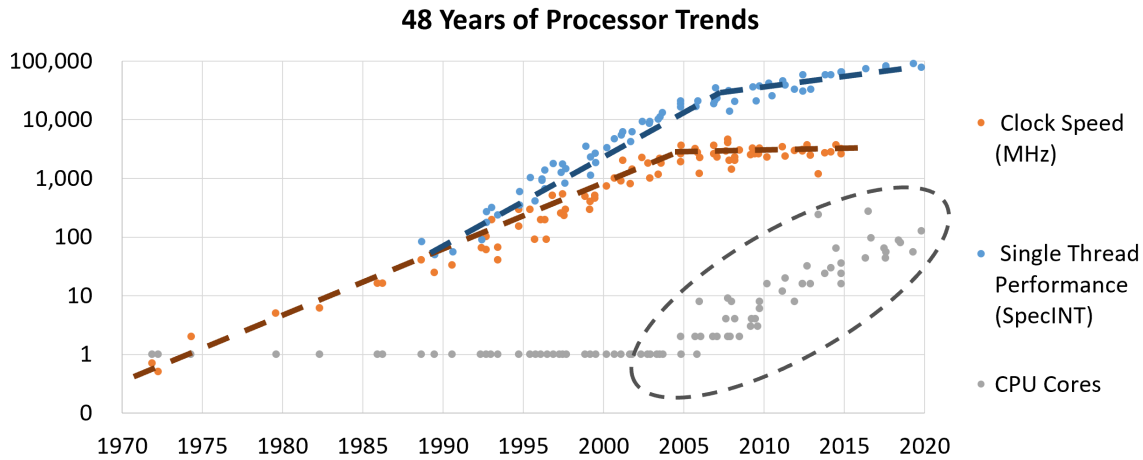
Figure 3.1: *48 years of processor trends plotted on a log scale. The data shows how CPU clock speed peaked in 2005 and has remained relatively constant since. At the same time while single threaded performance, indicated by* SpecINT *benchmark scores, has increased since 2005, the rate of increase has slowed dramatically. In response, hardware vendors have drastically increased the number of CPU cores on each processor. Original data up to the year 2010 collected by M. Horowitz, F. Labonet, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data for 2010-2019 collected by K. Rupp. All data available at* https://github.com/karlrupp/microprocessor-trend-data

3.2 Multi-Core CPU

A multi-core CPU can be roughly viewed as a handful of modern CPUs that are designed to work together, often on different tasks, leveraging the multiple-instruction-multiple-data (MIMD) computing model. A rich ecosystem of supporting software tools makes CPUs easy to program, and they are highly flexible with deep and varied instruction pipelines with sophisticated control logic and large caches (See Figure 3.2). This enables them to automatically amortize the delays cause by memory accesses for code that leverages regular memory access patterns and uses relatively small working sets of data that fit into the CPU caches. This is important as cache memory is often orders of magnitude faster than DRAM. CPUs are also fantastic at computing serial code due to their high clock rates.

Multi-core CPUs enable CPU threads to run on different logical processors in parallel. Each CPU thread runs with its own set of registers and stack memory, and context switches between threads, and the allocation of threads to logical hardware cores, are scheduled asynchronously

by the operating system. Due to the significant overheads of thread creation, *threadpools* have become popular as they amortise this cost through a create once, use many paradigm. For more information on CPU threading we suggest reading *C++ Concurrency in Action* [59].
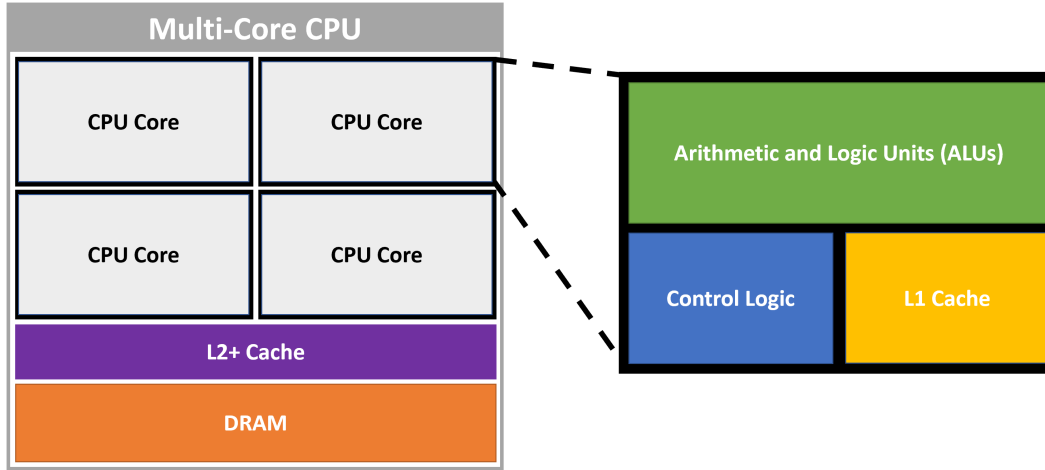
Figure 3.2: *High level architecture of a multi-core CPU. Each core has its own local L1 cache, control logic, and arithmetic and logic units, while all cores share access to a larger L2 cache and DRAM.*

3.3   Graphics Processing Unit (GPU)

In contrast to a multi-core CPU, 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. As such, each GPU processor has many more arithmetic logic units (ALUs), but reduced control logic and a smaller cache memory (see Figure 3.3).

For maximal performance, the GPU requires groups of threads within each thread block to compute the same operation on memory accessed via regular patterns. Like with the CPU, this enables the GPU to better leverage memory locality and amortize the order of magnitude slowdowns of accessing DRAM over cache memory. GPUs are therefore best at computing highly regular and separable computations over large working sets of data (e.g., large matrix-matrix multiplication). GPUs also typically run at about half the clock rate of CPUs, which further hinders their performance on purely sequential code.

17
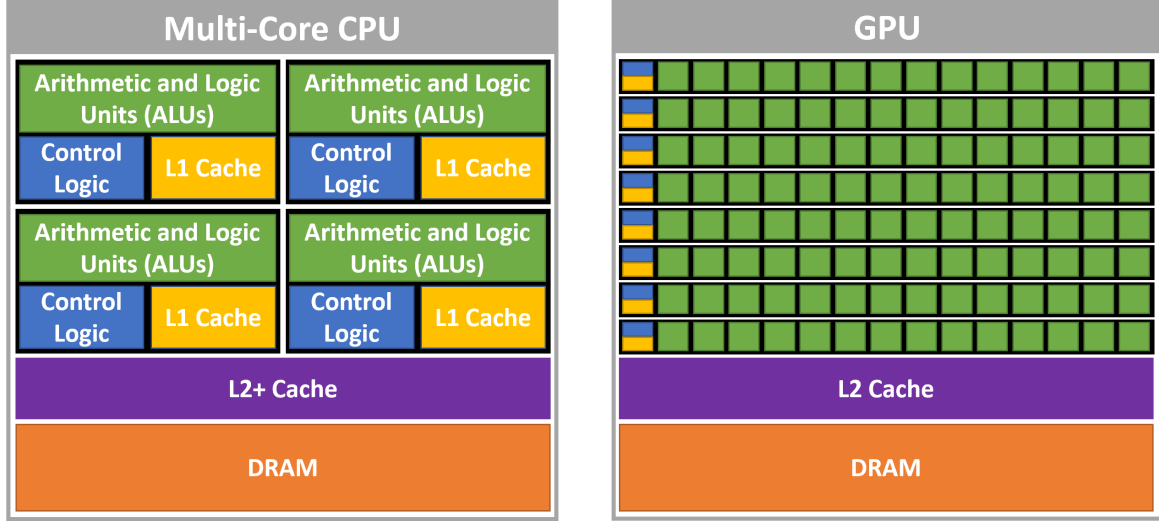
Figure 3.3: *High level architecture comparing a multi-core CPU (left) and GPU (right) processor. Importantly, a multi-core CPU has increased cache size and control logic, while a GPU has many more arithmetic and logic units (ALUs).*

When leveraging a GPU as an accelerator, for each independent computation, data must be transferred from the CPU to the GPU and then back again after computations are completed. This I/O communication overhead can be amortized by performing large amounts of arithmetic operations on the GPU per each round trip memory transfer. From a design perspective, GPUs are best suited for applications which require high throughput of a compute workload which has high computational intensity, and exhibits high degrees of natural parallelism.

Our work uses NVIDIA's CUDA [60] extensions to C++. CUDA is built around *host* (CPU) and *device* (GPU) memory and code. Special functions called *kernels* are launched from the host and then call device functions using parallel blocks of threads on the GPU.[1] On most modern GPUs, these threads run in *warps* of 32 threads. For maximal performance there should be no branching between threads in a single warp and all *global memory* (RAM)

---

[1]These functions are typically launched through the use of a special syntax (`myFunc«<a,b,c,d»>(args)`) where `a` specifies how many blocks to launch each containing `b` threads, `c` represents the amount of dynamically allocated shared (cache) memory, and `d` represents which *stream* of kernels this particular kernel launch should run in. Streams run in parallel of each other and kernels within each stream run sequentially.

accessed by each warp should be done in a *coalesced* (sequential) manner.[2] Each block's threads also access a small shared cache, which is split between a standard L1 cache and *shared memory*, which is manually managed by the programmer. Using shared memory can be as fast as using registers if done properly, and as such, historically, high performance CUDA code was dependent on smart use of the limited shared memory resources. To enable this resource sharing, all threads within a block are also guaranteed to run on the same processor, but the ordering of the blocks is not guaranteed. There are also a number of different ways to synchronize (groups of) threads in a block, in a stream, or on the entire device. Finally, it is important to note that kernel launches suffer very large overheads and so combining code into fewer kernels will improve overall performance. This process is generally referred to as *kernel fusion* [61]. For more information on the CUDA programming model we suggest reading the NVIDIA CUDA programming guide [60].

3.4 Hardware-Software Co-Design

Hardware-Software Co-Design is the process of collaboratively optimizing algorithms and computer hardware to ensure that target algorithms can take advantage of the strengths and minimize the weaknesses of the computer hardware on which they are run [15]. As diagrammed in Figure 3.4, co-design requires computer scientists to propagate up the hardware features and down the algorithmic features to redesign and refactor algorithms. This could mean simply reordering computations to expose different computational or memory access patterns. This could also mean redesigning algorithms or selecting alternate algorithms that better expose hardware friendly computational structures. Finally, in some cases this could even mean designing custom computer hardware to take advantage of the structured sparsity in key algorithms. During this process, if algorithms can be divided into known standard computational patterns, than existing optimized libraries can be used for efficient computations (e.g., the Berkeley Dwarfs [62; 63]). However, as we will see throughout this dissertation,

---

[2]While the penalty for out of order memory accesses has been greatly reduced on the newest GPUs due to sophisticated engineering by GPU manufacturers, it is still best practice to coalesce memory accesses.

there are often ways to develop further optimized and customized implementations that better target the exact sparsity and parallelism patterns found in robotics problems. As such, we endeavor to release our implementations as open-source libraries, and keep them updated to account for changes in the underlying computer hardware, in order to enable future robotics researchers to build on top of our optimized kernels, accelerating both robotics computations and application development timelines.
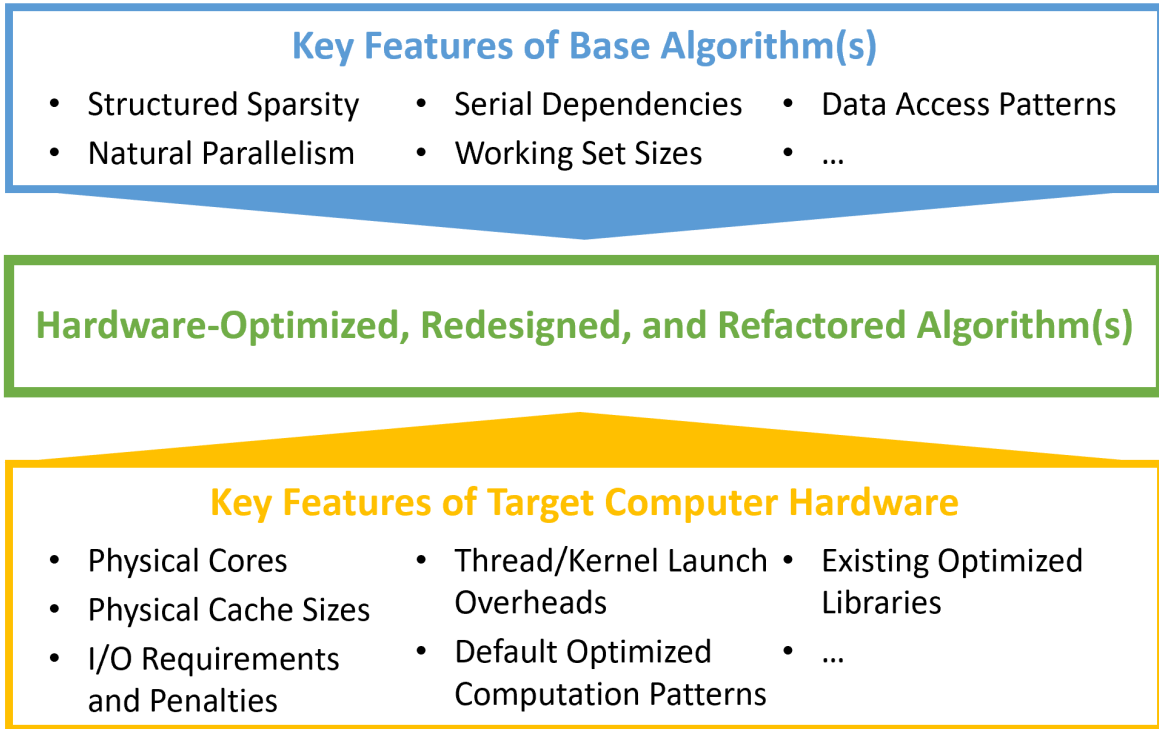
Key Features of Base Algorithm(s)

- Structured Sparsity
- Natural Parallelism
- Serial Dependencies
- Working Set Sizes
- Data Access Patterns
- …

Hardware-Optimized, Redesigned, and Refactored Algorithm(s)

Key Features of Target Computer Hardware

- Physical Cores
- Physical Cache Sizes
- I/O Requirements and Penalties
- Thread/Kernel Launch Overheads
- Default Optimized Computation Patterns
- Existing Optimized Libraries
- …

Figure 3.4: *Co-design requires computer scientists to propagate up the hardware features and down the algorithmic features to redesign and refactor algorithms in order to better map algorithms to hardware. Here we provide an incomplete list of some relevant algorithmic and hardware features. We explore these kinds of features in detail in the remainder of this dissertation.*

Chapter 4

MPC with GPU Accelerated DDP

This chapter aims to add to our understanding of the benefits and limitations of instruction-level and algorithm-level parallelization for whole-body, nonlinear MPC for a particular family of trajectory optimization algorithms based on Differential Dynamic Programming (DDP) [35]. We focus on DDP and its variants, particularly the iterative Linear Quadratic Regulator (iLQR) [38], as they have recently received increased attention due to growing evidence that online planning for model predictive control (MPC) is possible for high-dimensional robots [2; 4; 37; 64; 65; 66; 67]. Following the publication of the work upon which this chapter is based [3; 68], additional publications further reinforced the real-time applicability of these kinds of algorithms [69; 70; 71].

In this chapter, we describe our parallel implementation strategy on a modern NVIDIA GPU and present results demonstrating its performance compared to an equivalent multi-threaded CPU implementation using benchmark trajectory optimization tasks for a quadrotor and 7-DoF manipulator. We then deploy our implementation for whole-body, nonlinear MPC on a physical robot arm to demonstrate the feasibility of this approach in the presence of model discrepancies and communication delays between the robot and GPU. Our results suggest that GPU-based solvers can offer faster convergence than equivalent parallelized CPU implementations and

that in some cases these speedups lead to better real-world MPC performance. However, performance tradeoffs exist between convergence behavior and time per iteration as the degree of algorithm-level parallelism increases, limiting the possible improvements from parallelism for DDP based algorithms and implementations.

4.1   Related Work

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 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 [72; 73], GPUs [74; 75], and FGPAs [76; 77]. 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 [78; 79; 80; 81] and GPUs [82; 83]. Additionally, Antony and Grant [84] used GPUs to exploit the inherent parallelism in the "next iteration setup" step of DDP. Finally, there have been a series of recent works further exploiting this parallelism in dynamics gradient evaluations on the CPU, GPU, and FPGA [85; 86; 87], as well as through SIMD instructions on the CPU [88].

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 [89] and then Betts and Huffman [90], and has inspired a variety of "multiple shooting" methods [91; 92]. Recently, this approach has been used to parallelize both an SQP algorithm [93] and the forward [94; 95] and backward passes [64] of the iLQR algorithm. Experimental results from

22

using these parallel iLQR variants on multi-core CPUs achieve state-of-the-art results for real-time, whole-body, nonlinear MPC.

Our work aims to add to this literature by systematically comparing the performance of an parallel implementation of iLQR on a modern multi-core CPU and GPU to (1) better understand the performance implications of various implementation decisions that must be made on parallel architectures and (2) to evaluate the benefits and trade-offs of higher degrees of parallelization (GPU) versus a higher clock rate (CPU).

4.2   Parallelizing DDP

In this section we detail both the algorithm-level and instruction-level parallelism we implemented, for both the CPU and GPU, and present our final Parallel DDP (PDDP) algorithm. For additional detail on the algorithm and our implementation strategy we suggest reading Chapters 4 and 5 of [96].

4.2.1   Instruction-Level Parallelization

Since the standard DDP cost function (2.1) is additive and depends on each state and control independently, it can be computed in parallel following forward integration and summed in $\log(N)$ operations using a parallel reduction operator.

In addition, instead of computing the line search during the forward pass by sequentially reducing α, we can compute all forward simulations for a set of possible α values in parallel. Furthermore, if all simulations are computed in parallel, then the algorithm could select the "best" trajectory across all α values, rather than the first value that results in an improvement. We employ the line search criteria proposed by Tassa [40] and accept an iterate if the ratio of the expected to actual reduction,

$$z = \left(J - \tilde{J} \right) / \delta V^*(\alpha) \quad \text{where} \quad \delta V^*(\alpha) = -\alpha \kappa^T H_u + \frac{\alpha^2}{2} \kappa^T H_{uu} \kappa, \qquad (4.1)$$

falls in the range

$$0 < c_1 < z < c_2 < \infty. \tag{4.2}$$

Finally, since the dynamics are defined independently on each state and control pair, the Taylor expansions of the dynamics and cost (the "next iteration setup" step) can occur in parallel following the forward pass. As this is one of the more expensive steps, this is usually parallelized in CPU implementations of DDP currently being used for online robotic motion planning – though the number of knot points often exceeds the number of CPU processor cores.

4.2.2 Algorithm-Level Parallelization

Backward Pass

We break the N knot points into M_b equally spaced parallel blocks of size $N_b = N/M_b$. We compute the CTG within each block by passing information serially backwards in time, as is done in standard DDP. After each iteration we pass the information from the beginning of one block to the end of the adjacent block, ensuring that CTG information is at worst case $(M_b - 1)$ iterations stale between the first and last block as shown in Figure 4.1.

Farshidian et al. [64] note that this approach may fail if the trajectory in the next iterate is far enough away from the previous iterate as the stale CTG approximations are defined in relative coordinates and are only valid locally. Therefore, they propose a linear coordinate
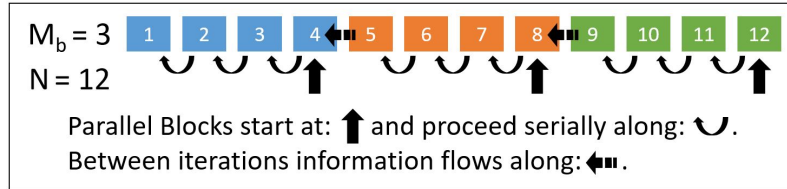
Figure 4.1: *Graphical representation of the backward pass algorithm-level parallelizations showing information flowing within each parallel block during each iteration and moving across adjacent block boundaries between iterations.*

transformation of the quadratic CTG approximation at iterate i to re-center it:

$$V_{xx}^{i+1} = V_{xx}^i, \qquad V_x^{i+1} = V_x^i + V_{xx}^i(x^{i+1} - x^i). \tag{4.3}$$

While this process will still usually converge [97; 98], in practice, some forward passes will fail to find a solution because either the CTG information was "too stale," or the new trajectory moved too far from the previous trajectory, rendering the controls at later knot points sub-optimal. Therefore, on the next pass we use the failed iterate's CTG approximation, as it is a less stale estimate, and again follow Tassa [40] and add a state regularization term ρI_n to V_{xx}' in the computation of Q_{uu} and Q_{xu} to stay closer to the last successful trajectory:

$$\begin{aligned} Q_{uu} &= \ell_{uu} + f_u^T \left(V_{xx}' + \rho I_n \right) f_u + V_x' \cdot f_{uu} \\ Q_{xu} &= \ell_{xu} + f_x^T \left(V_{xx}' + \rho I_n \right) f_u + V_x' \cdot f_{xu}. \end{aligned} \tag{4.4}$$

Forward Pass

Giftthaler et al. [94] introduced Gauss-Newton Multiple Shooting, which adapts iLQR for multiple shooting through a fast consensus sweep with linearized dynamics followed by a multiple shooting forward simulation from M_f equally spaced states of $N_f = N/M_f$ knot points as shown in Figure 4.2.

This parallel simulation leads to defects d between the edges of each block and changes the one step dynamics to $x_{k+1} = f(x_k, u_k) - d_k$. Incorporating this change into Equations 2.5-2.7
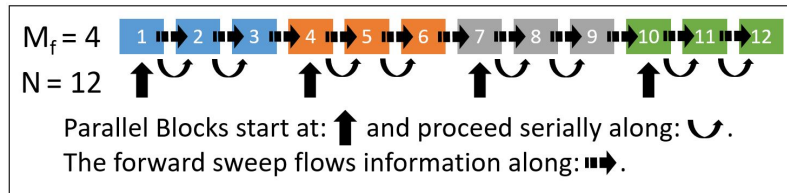
Figure 4.2: *Graphical representation of the forward pass algorithm-level parallelizations showing information flowing within each block during each forward simulation and along the entire trajectory through each forward sweep.*

results in a modified version of $Q(\delta x, \delta u)$:

$$Q_x = Q_x + f_x^T V'_{xx} d \qquad Q_u = Q_u + f_u^T V'_{xx} d. \tag{4.5}$$

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

$$0 < \max_k ||d_k||_1 < c_3 < \infty, \tag{4.6}$$

excluding any trajectories that have large defects which represent an artificial mathematical reduction in cost that is infeasible in practice.

Finally, in order to update the start states of each block, a serial consensus sweep is performed by integrating the state trajectory using the previously computed linearized dynamics ($A = f_x$, $B = f_u$) and feedback controls (K, κ), updating state $k+1$ for iterate $i+1$ (using a line search parameter α):

$$x_{k+1}^{i+1} = x_{k+1}^i + \left(A_k^i + B_k^i K_k^i\right)\left(x_k^{i+1} - x_k^i\right) + \alpha B_k^i \kappa_k^i + d_k^i. \tag{4.7}$$

Thus, the forward pass now becomes the serial consensus sweep followed by a parallel forward simulation on each of the M_f blocks.

4.2.3 The Parallel DDP Algorithm

Parallel DDP, shown in Algorithm 2 combines the instruction-level parallelizations, forward sweep, M_f multiple shooting intervals, and M_b backward pass blocks into a single algorithm with parameterizable levels of parallelism.

4.2.4 Implementation Details

As mentioned in Chapter 2, we used the NVIDIA CUDA extensions to C++ for our GPU implementation. We minimized memory bandwidth delays by doing most computations on a handful of fused GPU kernels, limiting CPU operations to high level serial control flow for kernel launches. We also made heavy use of streams and asynchronous memory transfers to increase throughput wherever possible. For example, by simultaneously computing the

26

Algorithm 2: *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** $k = b_{N_b} : b_0$ **do**

5:             d_k, (2.7), (4.4), (4.5) $\rightarrow Q^k$

6:             **if** Q_{uu}^k is invertible **then**   **Backward**

7:                 (2.8) $\rightarrow K_k, \kappa_k$ **Pass**

8:                 (2.9) $\rightarrow V_k$ and derivatives

9:             **else**

10:                 Increase ρ go to line 3   ⎭

11:     **for** all $\alpha[i]$ **do** in parallel   ⎫

12:         $\tilde{x}_0[i] = x_0$

13:         **if** $M_f > 1$ **then**

14:             **for** $k = 0 : N - 1$ **do**   **Consensus**

15:                 $\tilde{x}_k[i]$, (4.7) $\rightarrow \tilde{x}_{k+1}[i]$ **Sweep**

16:         **for** all M_f blocks b **do** in parallel   ⎫

17:             **for** $k = b_0 : b_{N_f} - 1$ **do**

18:                 $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$

19:                 $\tilde{x}_{k+1}[i] = f(\tilde{x}_k[i], \tilde{u}_k[i])$

20:                 $\tilde{d}_k[i] = 0$

21:             $k = b_{N_f}$ **Forward**

22:             **if** $k < N$ **then**   **Pass**

23:                 $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$ **Forward**

24:                 $\tilde{d}_k[i] = x_{k+1} - f(\tilde{x}_k[i], \tilde{u}_k[i])$ **Simulation**

25:         $\tilde{X}[i], \tilde{U}[i]$, (2.1), (4.1) $\rightarrow \tilde{J}[i], \tilde{z}[i]$

26:         $i^* \leftarrow \arg\min_i \tilde{J}[i]$ s.t. $\tilde{z}[i], \tilde{d}[i]$ satisfy (4.2), (4.6)

27:         **if** $i^* \neq \emptyset$ **then**

28:             $X, U, d \leftarrow \tilde{X}[i^*], \tilde{U}[i^*], \tilde{d}[i^*]$

29:         **else**

30:             Increase ρ go to line 3   ⎭

31:     Taylor approximate the cost at X, U **Next**

32:     Taylor approximate the dynamics at X, U **Iteration Setup**

Taylor approximations of the dynamics and cost in separate streams, the throughput of the next iteration setup step was much closer to the maximum of the running times for those calculations than the sum.

We also found that the general purpose GPU matrix math libraries (e.g., cuBLAS) were optimized for very large matrix operations, while DDP algorithms require many sets of serial small matrix operations. We implemented simpler custom fused kernels which provide a large speedup by keeping the data in shared memory throughout the computations. We further optimized our code by precomputing serial terms during parallel operations. For example, during the backward pass, A, B, K, and κ were loaded into shared memory, so computing $A + BK$ and $B\kappa$ only added a small overhead to the paralellizable backward pass, while greatly reducing the time for the serial consensus sweep.

Our multi-threaded CPU implementation leveraged the standard C++ `thread` library and reused the same baseline code to leverage the optimizations made during the GPU implementation and to provide an equivalent implementation for comparison. However, for optimal performance, we had to introduce serial loops within threads to limit the number of threads to a small multiple of the number of CPU cores.

As no sufficient GPU rigid body dynamics library existed at the time of publication (inspiring Chapter 5), we implemented a custom GPU optimized forward dynamics kernel for the manipulator based on the Joint Space Inertia Inversion Algorithm, the fastest parallel forward dynamics algorithm for open kinematic chain robots with a small number of rigid bodies [99]. For better direct comparisons, we used a looped version of that code for the CPU implementation.[1] Finally, following the state-of-the-art, we implemented the iLQR variant of DDP for our experiments.

---

[1]We note that while the Joint Space Inertia Inversion Algorithm is not the fastest serial forward dynamics algorithm, the difference is minimal for a 7-Dof manipulator.

4.3 Exploring the Benefits and Limitations of Parallelism

In this section we explore the benefits and limitations of parallelism across different problems, computing architectures, and levels of parallelism. These experiments suggest that practical performance improvements can be obtained through large-scale parallelization and GPU implementations, but that the tradeoffs between the degree of parallelism and convergence speed are strongly dependent on system dynamics and problem specification.

We ran these experiments on a laptop with a 2.8GHz quad-core Intel Core i7-7700HQ CPU, a NVIDIA GeForce GTX 1060 GPU, and 16GB of RAM. In these experiments we initialized the algorithm with a gravity compensating input. Both the GPU and CPU implementations used the same scheme for updating ρ and the same set of options for α. We report convergence results (cost as a function of time and iteration) and the time per iteration. Total cost reduction as a function of time is a particularly useful metric when deploying algorithms in MPC scenarios where there is typically a fixed control time budget. To ensure our results were representative for each experiment, we ran 100 trials with noise distributed $\mathcal{N}(0, \sigma^2)$ applied to the velocities of the initial trajectory. Our solver implementations and these examples can be found at https://github.com/plancherb1/parallel-DDP.

4.3.1 Quadrotor

We first considered a quadrotor system with 4 inputs corresponding to the thrust of each rotor and 12 states corresponding to the position and Euler angles, along with their time derivatives. We solved a simple flight task from a stable hover 0.5 m above the origin to a stable hover at the same height and at 7 m in the x and 10 m in the y direction. We used a quadratic cost function of the form:

$$J = \frac{1}{2}(x_N - x_g)^T Q_N (x_N - x_g) \ + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T Q (x_k - x_g) + \frac{1}{2} u_k^T R u_k, \qquad (4.8)$$

setting $Q = \texttt{blkdiag}(0.01 \times I_{3x3}, 0.001 \times I_{3x3}, 2.0 \times I_{6x6})$, $R = 5.0 \times I_{4x4}$, $Q_N = 1000 \times I_{12x12}$. We solved over a 4 second trajectory with $N = 128$, $M_F = M_B = M = 1, 2, 4, 8, 16, 32, 64$, a

3rd-order Runge-Kutta integrator, and $\sigma = 0.001$.

Figure 4.3 reveals that the delayed flow of information due to the algorithm-level parallelizations (stale CTG information, fixed starting state of each simulation block) generally leads to smaller steps and therefore slower cost reduction per iteration. For example, for the CPU implementation, the median line search depth for $M = 1$ was between 0 and 1, while for $M = 4$ it was 5. It also shows that the GPUs ability to run a fully parallel line search, as compared to the CPUs partially parallel approach (due to limited number of hardware cores), allows the GPU to select a better "best line search" option and descend faster while avoiding local minima. For example, for the GPU implementation, the median line search depth for $M = 1$ was also between 0 and 1, while for $M = 4$ it was only 3. These trends were also mirrored in the success rate of the algorithm. While 0% of CPU and GPU runs failed for $M = 1, 2, 4$, and on the GPU only 5% failed for $M \geq 8$, on the CPU over 30% failed for $M \geq 8$.

The median time per iteration for each of the parallelization options for both implementations is shown in Figure 4.4. Our observed per-iteration times of under 3 ms for all GPU and CPU
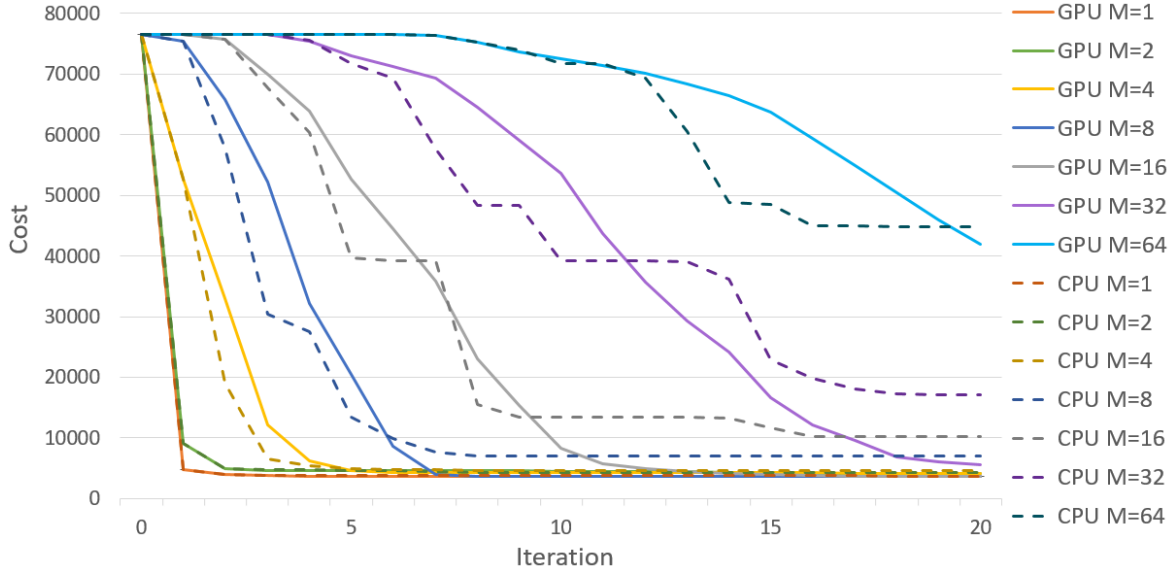
Figure 4.3: *Median cost vs iteration for the quadrotor experiment. We find that, for both the CPU and GPU, as the amount of algorithm-level parallelism increases, the convergence rate decreases.*

30

cases are comparable to state-of-the-art reported rates of 5-25 ms [2] on a similar UAV system. These results also match our expectations that the higher clock rate would allow the CPU to compute the serial consensus sweep faster while the GPU is able to leverage its increased number of cores to compute the parallel next iteration setup step faster.
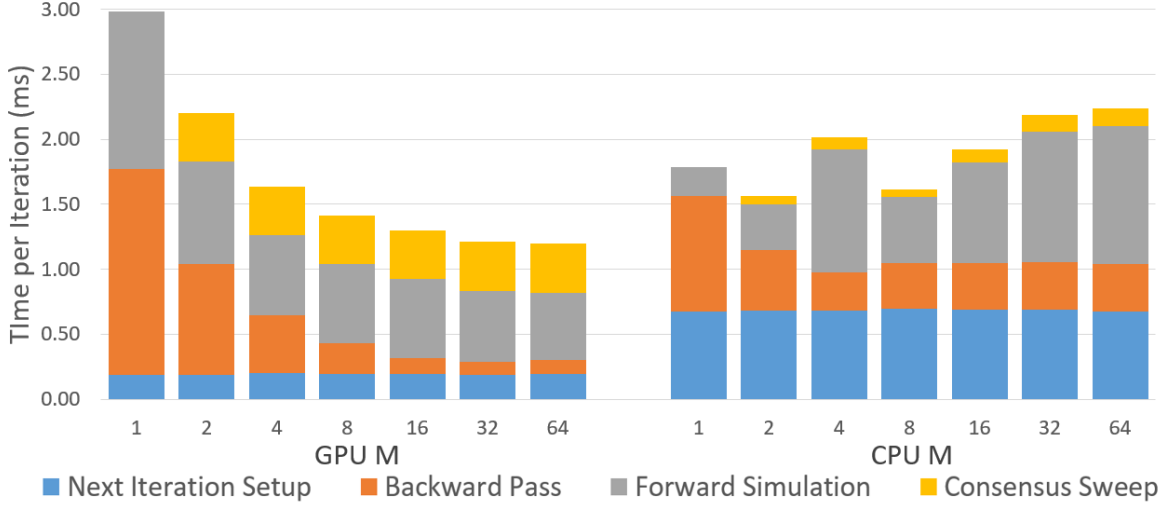
Figure 4.4: *Median time per iteration for the quadrotor experiment. We find that on the GPU, as the level of parallelism, M, increases, the time per iteration decreases, albeit with diminishing marginal returns. The CPU speedups in the backward pass stall at $M = 4$ as the CPU only has 4 cores, while the forward simulation is slower, for increased M, due to deeper serial line searches.*

For the CPU implementation, we also observed that parallelization can improve the speed of the backward pass until the number of available cores (4) is saturated at which point performance stagnates. For the forward simulation, we found that slower paths to convergence, and thus deeper line searches, quickly outweighed the running time gains due to parallelism. This is most evident in the increase in the time for the forward simulation as M grows from 8 to 64. By contrast, the GPU implementation is able to run the line search fully in parallel, which led to reductions in running time for both the backward pass and the forward simulation as M increases. However, there are diminishing returns. 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 for each line search option the consensus sweep cannot be parallelized, the running times for both steps remain constant. In fact, by the $M = 64$

case, the consensus sweep was almost a third of the total computational time for the GPU as compared to only 17 percent for the $M = 2$ case.

This increased speed per iteration and decreased convergence rate leads to a level of parallelism which optimizes the time to convergence, as shown in Figure 4.5. There we find that in the $M = 1$ and $M = 2$ cases, the CPU is able to leverage its higher clock rate to outperform the GPU. However, the GPU is able to better exploit the algorithm-level parallelism and outperform the CPU for $M > 2$. For this experiment the dynamics computations required are simple enough, and the problem size is small enough, that the fastest approach is CPU $M = 1$ indicating that on simple problems the overheads from parallelism outweighs the gains.
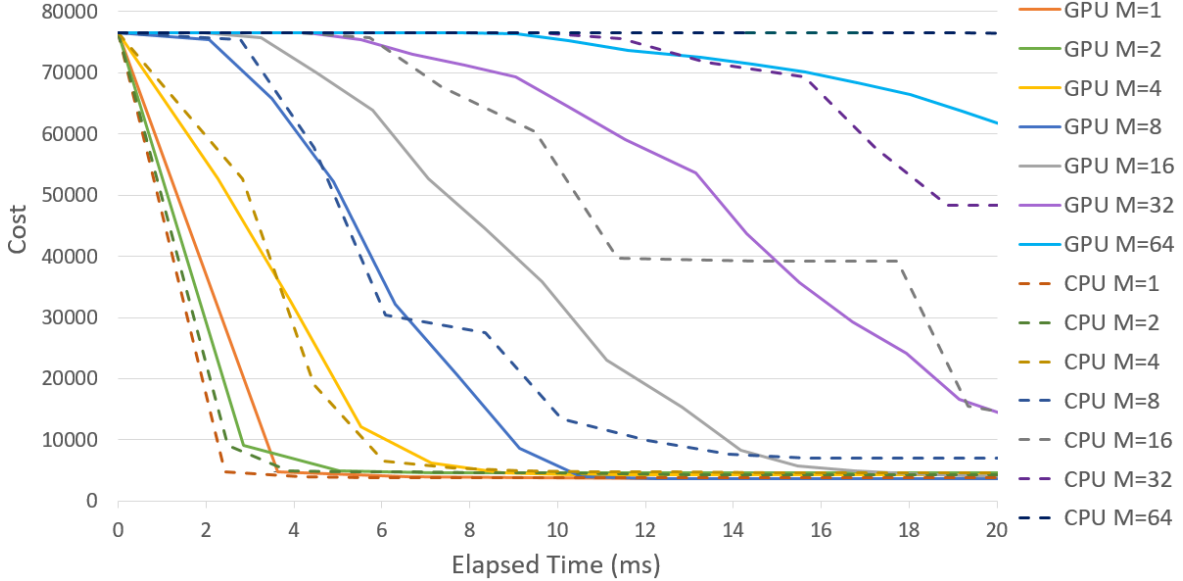
Figure 4.5: *Median cost for the first 20 milliseconds of the quadrotor experiment. We find that the decreases in time per iteration gained from parallelism are outweighed by the decreases in convergence rate indicating that on simple problems, the overheads from parallelism outweighs the gains.*

4.3.2   Manipulator

We then consider the Kuka LBR IIWA-14 manipulator which has 7 inputs corresponding to torques on the 7 joints. The nominal configuration is defined as the manipulator pointing straight up in the air. We solved a trajectory optimization task from a start state to a goal state

across the workspace depicted in Figure 4.6. We set $Q = \texttt{blkdiag}(0.01 \times I_{7x7}, 0.001 \times I_{7x7})$, $R = 0.001 \times I_{7x7}$, $Q_N = 1000 \times I_{14x14}$. We solved the problem over a 0.5 second trajectory with $N = 64$, $M_F = M_B = M = 1, 2, 4, 8, 16, 32$, a 1st-order Euler integrator, and $\sigma = 0.001$.
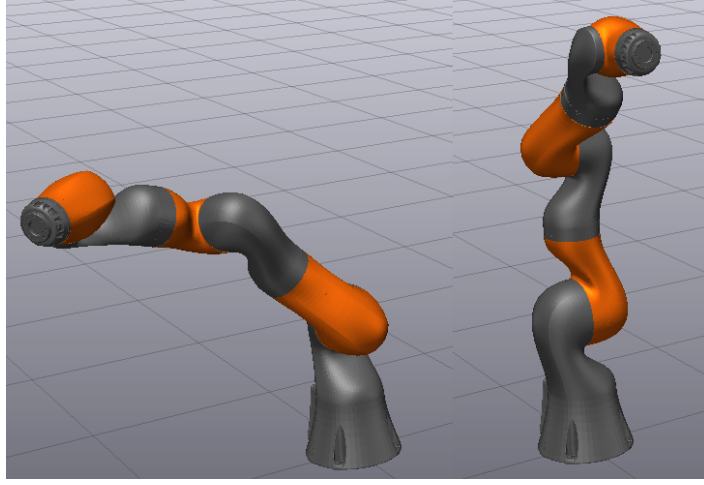
Figure 4.6: *Start (left) and goal (right) states for the manipulator experiment.*

Figure 4.7 shows that parallelism leads to speedups in time per iteration on the GPU and CPU. We see that again on the GPU both the forward simulation and backward pass decrease in time as M increases. On the CPU we again see that the backward pass decreases in time until the CPU runs out of cores at $M = 4$, while the forward simulation time varies non-monotonically for different values of M depending on the parallelization speedup and the depth of line search slowdown. With all cases having a median time per iteration under $5\,\mathrm{ms}$, both our GPU and CPU implementations are able to perform at speeds reported as state-of-the-art [2; 4; 37; 64; 65; 66].

Unlike in the quadrotor example, the more computationally expensive forward dynamics and increased problem size in this example led to performance gains from parallelism as shown in Figure 4.8. We find that the GPU is able to successfully exploit the algorithm-level parallelism with faster convergence from $M = 2, 4, 8, 16$ than $M = 1$ and that $M = 32$ on the GPU converges in about the same time as the CPU's fastest standard option, $M = 2$. We also tested various combinations of the number of blocks for the forward and backward passes until

we found the best possible CPU variant for this problem ($M_f = 2$ and $M_b = 4$) and found that GPU $M = 2, 4, 8, 16$ still converge faster.
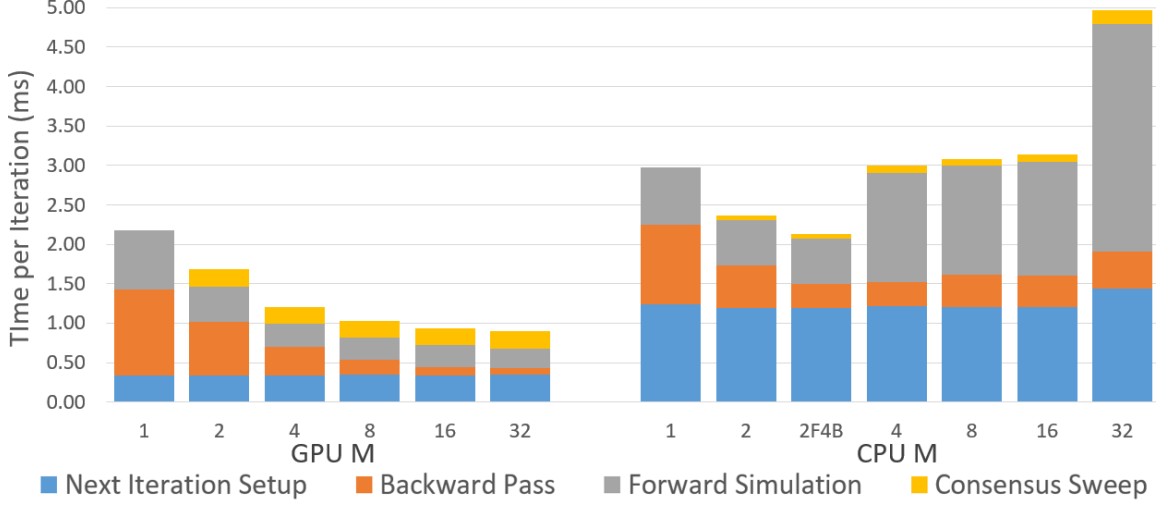
Figure 4.7: *Median time per iteration for the manipulator experiment. We again find that the GPU gets faster with increased parallelism. In this case the CPU also gets faster until $M = 4$. Also the fully parallel next iteration setup is much faster on the GPU than CPU. Taken together all of these effects show the increased speedups available from parallelism as the total computational complexity grows.*
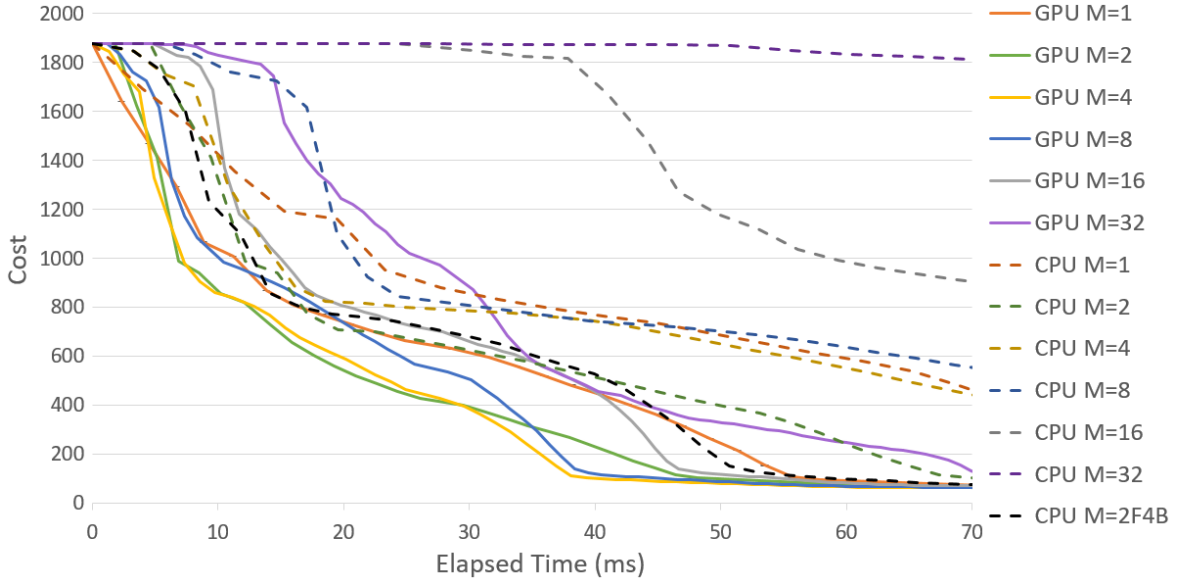
Figure 4.8: *Median cost for the first 70 milliseconds of the manipulator experiment. We find that the GPU is able to successfully exploit the algorithm-level parallelism with faster convergence from $M = 2, 4, 8, 16$ than $M = 1$ and that $M = 32$ on the GPU converges in about the same time as the CPU's fastest standard option, $M = 2$. This all shows the power of parallelism for improving the overall performance of computationally expensive tasks.*

4.4 Whole-body, Nonlinear MPC Experiments

To better understand these results and the applicability of our GPU implementation for online model-predictive control, we conducted a goal tracking experiment with the manipulator in simulation and on a physical robot. These experiments demonstrated the feasibility of this approach in the presence of model discrepancies and communication delays between the robot and GPU. We also found that higher control rates generally lead to better tracking performance across a range of parallelization options, further reinforcing the importance of improved end-to-end latency.

4.4.1 Simulation Experiments

We began by testing our implementation in simulation to prove the validity of using PDDP for whole-body, nonlinear MPC. At each control step we ran our fastest solver, the GPU $M = 4$ implementation, with a maximum time budget of $10\,\mathrm{ms}$. We warm started the iLQR algorithm by shifting all variables from the previous solve by the control duration and then rolling out a new initial state trajectory starting from the current measured state (with a gravity compensating input at the trailing knot points). During optimization periods, we simulated the system forward in realtime using the previously computed solution.

We considered a end-effector pose tracking task where the goal moved continuously along a figure eight path. We modified our cost function to include the end-effector error:

$$
\begin{aligned}
J = &\frac{1}{2}(\mathrm{ee}(q_N) - \mathrm{ee}_{goal})^T Q_N (\mathrm{ee}(q_N) - \mathrm{ee}_{goal}) \ + \frac{1}{2}\dot{q}_N^T \dot{Q}_N \dot{q}_N \\
&\sum_{k=0}^{N-1} \frac{1}{2}(\mathrm{ee}(q_k) - \mathrm{ee}_{goal})^T Q (\mathrm{ee}(q_k) - \mathrm{ee}_{goal}) + \frac{1}{2}\dot{q}_k^T \dot{Q}\dot{q}_k + \frac{1}{2}u_k^T R u_k,
\end{aligned}
\tag{4.9}
$$

where we include the quadratic penalty on $\dot{q}$ to encourage a stable final position. We set $Q = \mathtt{blkdiag}(0.01 \times I_{3x3}, 0 \times I_{3x3})$, $R = 0.0001 \times I_{7x7}$, $Q_N = \mathtt{blkdiag}(1000 \times I_{3x3}, 0 \times I_{3x3})$, $\dot{Q} = 0.1 \times I_{7x7}$, $\dot{Q}_N = 10 \times I_{7x7}$. At each control step we solved the problem using a first-order Euler integrator and $N = 64$ knot points over a 0.5 second trajectory horizon. The full figure eight trajectory we were tracking had a period of 10 seconds. To initialize the experiment, we

held the first goal pose constant until both $||\text{ee}(q) - \text{ee}_{goal}||_2^2$ and $||\dot{q}||_2^2$ were both less than 0.05 at which point the goal began moving along the figure eight path.

Figure 4.9 shows the trajectory computed by running the MPC experiment starting from an initial vertical state. Aside from confirming that good tracking performance is possible, we observed that the bookkeeping needed to implement MPC on a GPU does add delays in the control loop. In particular, the shifting of the previous variables and rolling out from the updated starting state takes almost $1.4\,\text{ms}$ on average. Since GPU $M = 4$ is able to compute iterations in about $1.2\,\text{ms}$, this MPC initialization step is quite expensive. We expect there are potential avenues for improving this overhead through better software engineering (e.g., by using circular buffers to reduce memory copy operations).
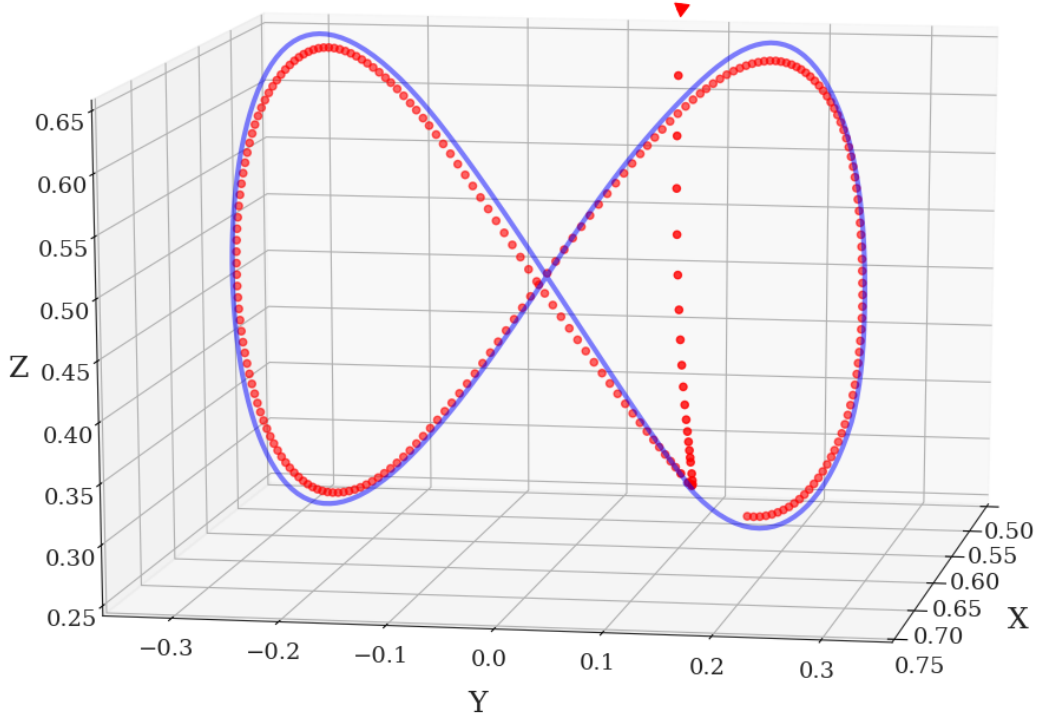
Figure 4.9: *Executed trajectory (red) vs. goal trajectory (blue) for the MPC experiment, showing good tracking performance for our GPU implementation of PDDP in simulation.*

4.4.2 Hardware Experiments

We then ran a figure eight goal tracking experiment with the physical Kuka arm sweeping control step duration and amount of algorithm level parallelism. We used the same cost function as in the simulation experiments in Section 4.4.1 and again warm started our solver by shifting all variables from the previous solve by the control step duration and then rolling out a new initial state trajectory starting from the current measured state (with a gravity compensating input at the trailing knot points). Simultaneously, we had another thread executing the previously computed feedback controller. To initialize the experiment, we again held the first goal pose constant until the 2-norm of the end-effector pose error and joint velocity were both less than 0.05 at which point the goal began moving along the figure eight path. Figure 4.10 shows the Kuka arm during one of these experiments.
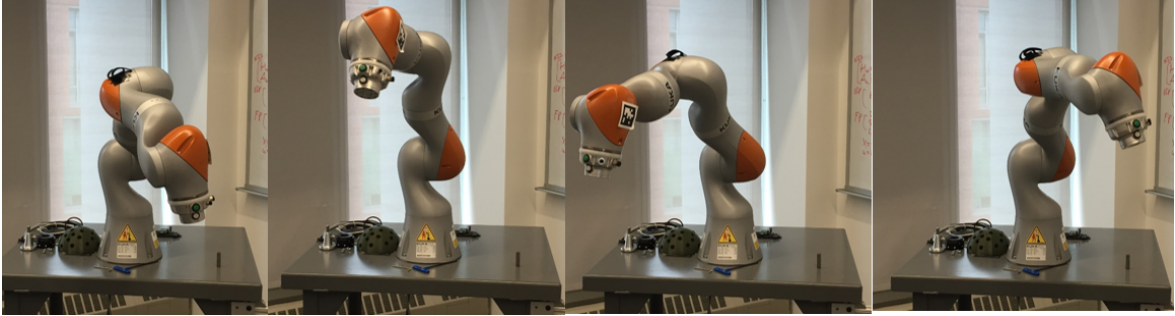
Figure 4.10: *The Kuka arm during a figure eight goal tracking experiment.*

Figure 4.11 shows the average tracking error plotted against control step duration. We found that good tracking performance is possible for a wide range of solvers, and a faster control step duration generally had better tracking performance. This is shown by the fact that while all of the points on the chart have a relatively small overall average tracking error, as the control step duration increases, so to does the tracking error. We also found that solvers start to fail when they had about as many (or less) iterations as the amount of algorithm level parallelism (e.g., $M = 4$ with 3 or less iterations). Finally, we found that for a similar control step duration, the solver that was able to take more iterations, a proxy for a more optimal

solution, had a lower tacking error. This indicates that, at least in this experimental setup, beyond some minimal level of optimality, while a more optimal solution was always preferred, delivering a sub-optimal solution faster outperformed a slow-to-update more optimal solution. We hypothesize that this is due to the fact that a faster update better accounts for feedback from the real world and can better overcome structural issues like model discrepancies between the physical robot and its simulated model. This tradeoff in optimality and control step duration further reinforces the importance of developing low-latency solvers.
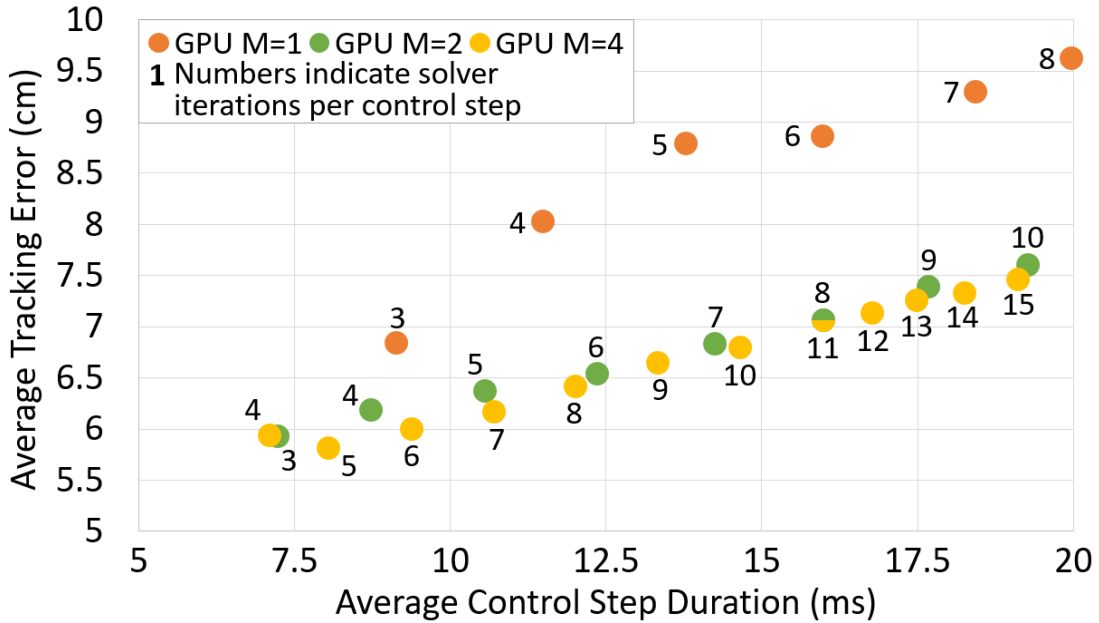
Figure 4.11: *Tracking error for a range of solvers vs. control step duration. We found that good tracking performance is possible for a wide range of solvers, and a faster control step duration generally had better tracking performance. As such, beyond some minimal level of optimality, while a more optimal solution was always preferred, delivering a sub-optimal solution faster outperformed a slow-to-update more optimal solution.*

4.5   Conclusion and Future Work

We presented an analysis of a parallel multiple-shooting iLQR algorithm that achieves state-of-the-art performance on example whole-body, nonlinear, trajectory optimization and MPC tasks both in simulation and on physical robot hardware. Our results show how parallelism, and GPU acceleration, can be used to increase the convergence speed of DDP algorithms

and lead to better real-world MPC performance in some situations. However, tradeoffs exist between convergence behavior and time per iteration as the degree of algorithm-level parallelism increases, limiting the possible improvements from parallelism for DDP based algorithms and implementations.

Several directions for future research remain. First, we used hand-optimized, custom, analytical and numerical dynamics methods specific to the systems we considered to ensure good performance on the GPU. Integrating these implementations in recently developed code-generating CPU and GPU rigid body dynamics libraries [87; 100], which we discuss in Chapter 5, would enable us to test these algorithms on a wider variety of robot models. Second, it would be interesting to consider the performance impact of adding nonlinear constraints to parallel iLQR using augmented Lagrangian or QP-based methods [44; 45; 46; 47; 48].

Other types of trajectory optimization formulations and algorithms may be more suitable for large-scale parallelization on GPUs, such as direct transcription and solvers based on the alternating direction method of multipliers [101]. We intend to broaden our investigation beyond DDP algorithms in future work and present preliminary results in the development of GPU accelerated direct methods in Chapter 6.

Finally, in future work we would like to evaluate parallel trajectory optimization algorithms for MPC using low-power mobile parallel compute platforms such as the NVIDIA Jetson.

Chapter 5

GRiD: GPU Accelerated Rigid Body Dynamics with Analytical Gradients

Rigid body dynamics algorithms and their gradients are bottleneck computations for state-of-the-art implementations of whole-body, nonlinear MPC, consuming 30% to 90% of the total computational time as shown in Figure 5.1 [1; 2; 3; 4]. As such, in this chapter, we explore further opportunities to increase the performance of whole-body, nonlinear MPC by accelerating rigid body dynamics.
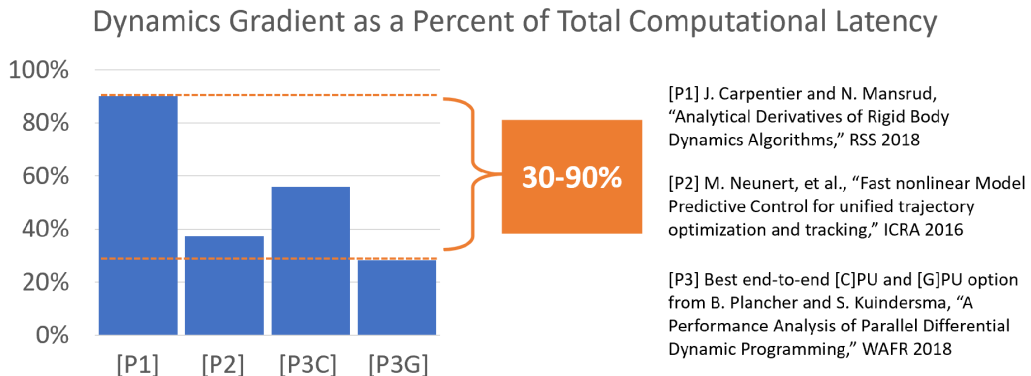
Figure 5.1: *Recent research indicates that rigid body dynamics gradients consume 30-90% of the total computational time of whole-body, nonlinear MPC [1; 2; 3].*

Despite being highly accurate and optimized, existing implementations of spatial-algebra-based approaches to rigid body dynamics do not take advantage of opportunities for parallelism present in the algorithm, limiting their performance [1; 100; 102; 103]. This is critical because there is natural parallelism in many computations involving rigid body dynamics. For example, the computation of the gradient of forward dynamics in most trajectory optimization algorithms is naturally parallel across the discrete points in the trajectory. Furthermore, as discussed in Section 3, the performance of multi-core CPUs has been limited by thermal dissipation, enforcing a utilization wall that restricts the performance a single chip can deliver [5; 6]. This has motivated increased use of GPUs, which can provide opportunities for higher performance by supporting larger-scale parallelism within a single chip.

In this chapter, we describe *GRiD*, a GPU-accelerated library for spatial-algebra-based rigid body dynamics and their analytical gradients. GRiD is designed to accelerate whole-body, nonlinear MPC by using blocks of GPU threads to compute the tens to hundreds of naturally parallel computations of rigid body dynamics and their gradients found in these algorithms. GRiD implements the more accurate spatial-algebra-based formulation [16] of rigid body dynamics used in state-of-the-art trajectory optimization [104; 105; 106; 107] and was the result of a series of papers exploring how to design easy-to-use, hardware-optimized, open-source implementations of rigid body dynamics algorithms [85; 86; 87].

GRiD not only unlocks the ability for whole-body, nonlinear trajectory optimization to run entirely on the GPU, but when performing multiple computations of rigid body dynamics and their gradients, it provides as much as a 7.2x speedup over a state-of-the-art, multi-threaded CPU implementation. GRiD also enables the use of a GPU as a rigid body physics accelerator for algorithms that are computed on a host CPU, maintaining as much as a 2.5x speedup when accounting for the I/O communication overhead between the CPU and GPU.

We released GRiD as an open-source library to enable robotics researchers to better explore and leverage the performance gains from large-scale parallelism on GPU platforms. GRiD can be found at https://github.com/robot-acceleration/grid.

5.1 Related Work

GRiD is designed to provide general-purpose, spatial-algebra-based dynamics with analytical gradients, and to accelerate them through large-scale parallelism on the GPU.

While there are many existing state-of-the-art spatial-algebra-based rigid body dynamics libraries [100; 102; 108; 109; 110], these libraries are not optimized for GPUs [3]. The exception is the recently released NVIDIA Isaac Sim [111] which supports spatial-algebra-based forward simulation, but not gradients, on the GPU.

As such, prior work using spatial-algebra-based approaches for planning and control on GPUs were either limited to cars, drones, and other lower degrees-of-freedom systems [112], or relied on manually-optimized implementations of rigid body dynamics and their gradients for a specific robot model [3]. Most machine learning approaches that leverage spatial-algebra-based rigid body dynamics rely on these aforementioned libraries [14; 111].

Using automatic differentiation, differentiable physics engines can also support gradient computations and have shown promise for real-time nonlinear MPC use on CPUs [113] and for accelerating machine learning, computer graphics, and soft robotics applications [107; 114; 115; 116; 117; 118; 119; 120] on CPUs and GPUs. However, existing GPU-based differentiable physics engines are optimized for simulating thousands of interacting bodies through contact using maximal coordinate, particle, and mesh-based approaches, which are less accurate when used for rigid body robotics applications over longer time step durations [104; 107].

GPUs have also historically been used to accelerate gradient computations through numerical differentiation [121; 122]. However, these methods have been shown to have less favorable numerical properties when used for robotic planning, control, and machine learning.

5.2 Rigid Body Dynamics Background

State-of-the-art spatial-algebra-based rigid body dynamics algorithms [16] operate in minimal coordinates and compute functions of the joint position $q \in \mathbb{R}^{n_q}$, velocity $\dot{q} \in \mathbb{R}^{n_v}$, acceleration $\ddot{q} \in \mathbb{R}^{n_v}$, and input torque $\tau \in \mathbb{R}^{m_\tau}$ that satisfy:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = B(q)\tau + J(q)^T F \tag{5.1}$$

where $M(q) \in \mathbb{R}^{n_q \times n_q}$ is the mass matrix, $C(q, \dot{q}) \in \mathbb{R}^{n_q \times n_v}$ is a Coriolis matrix, $G(q) \in \mathbb{R}^{n_q}$ is the generalized gravity force, $B \in \mathbb{R}^{n_q \times m_\tau}$ maps control inputs into generalized forces, and $J(q) \in \mathbb{R}^{n_q \times p}$ maps any external or constraint forces $F \in \mathbb{R}^p$ into generalized forces.

Common algorithms include: *Forward Dynamics*, computing $\ddot{q}$ when given $q, \dot{q}, \tau$, and optionally F; *Inverse Dynamics*, computing τ when given $q, \dot{q}, \ddot{q}$ and optionally F; as well as the computations of the various terms present in Equation 5.1. We note that common implementations of forward dynamics leverage either the Articulated Body Algorithm (ABA) or a combination of the Composite Rigid Body Algorithm (CRBA) to compute M and the Recursive Newton-Euler Algorithm (RNEA) for inverse dynamics, as follows:

$$\ddot{q} = \text{ABA}(q, \dot{q}, \tau, F)$$
$$\ddot{q} = M^{-1}(\tau - c) \text{ where } M = \text{CRBA}(q) \text{ and } c = \text{RNEA}(q, \dot{q}, 0, F). \tag{5.2}$$

Previous work has also showed that the most efficient way to compute the gradients of forward dynamics is through the direct computation of the inverse of the mass matrix, $M^{-1}(q)$, and the computation of the gradient of inverse dynamics [1],

$$\nabla \ddot{q} = -M^{-1}(q)\nabla \text{RNEA}(q, \dot{q}, \ddot{q}, F). \tag{5.3}$$

For more information on spatial-algebra-based rigid body dynamics we suggest reading Featherstone's *Rigid Body Dynamics Algorithms* [16].

5.3 The GRiD Library

In order to enable the broader robotics community to leverage GPU acceleration, our overarching design methodology was to make GRiD easily adoptable and extensible. As such, the resulting optimized CUDA C++ code is designed to be header-only with only a single dependency, the standard `cuda_runtime.h` library. In fact, even during URDF parsing and code generation, GRiD only requires the `beautifulsoup4`, `lxml`, `numpy`, and `sympy` Python libraries. Furthermore, we designed GRiD to be used by both GPU experts and novices. As such, we provide an API that allows users to integrate GRiD either directly into their existing CUDA code or through standard CPU C++ function calls, and also provide functions to automatically initialize and allocate all necessary memory on the CPU and GPU.

The GRiD library is built using a set of modular open-source packages (shown in Figure 5.2) to enable easy extension, and re-use by other robotics researchers. **GRiD** wraps and automates our GPU code generation engine (**GRiDCodeGenerator**), a self-contained URDF parser (**URDFParser**), and a set of reference implementations of rigid body dynamics algorithms (**RBDReference**) that can be used for code validation and testing. We also provide the benchmark experiments as described in Section 5.5.2 as a separate package (**GRiDBenchmarks**), as they require the support of additional external libraries to provide reference timings.
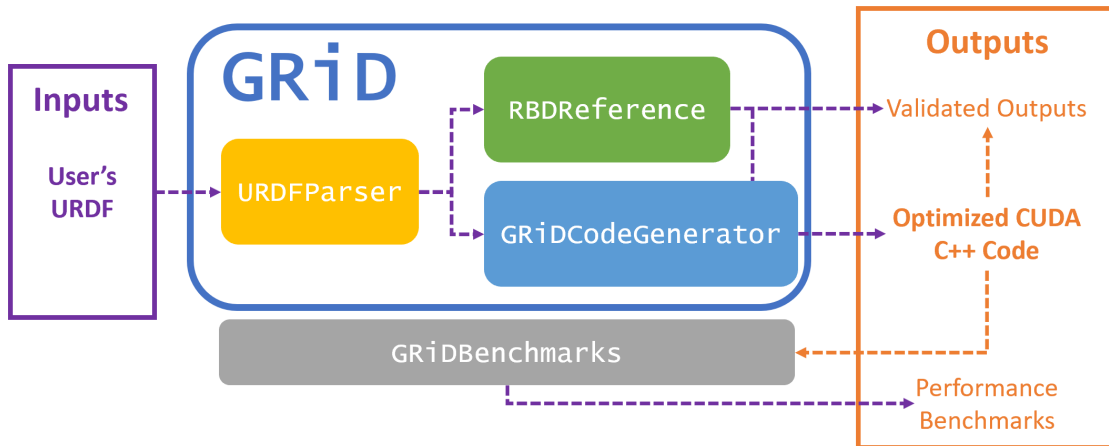
Figure 5.2: *The GRiD library package ecosystem takes an input URDF file and outputs optimized CUDA C++ code which can be validated against reference outputs and benchmarked for performance.*

The GRiD library currently fully supports any robot model consisting of revolute, prismatic, and fixed joints, and implements the following rigid body dynamics algorithms:

- The Recursive Newton Euler Algorithm (RNEA) for inverse dynamics [16];

- The direct inverse of mass matrix (M^{-1}) [1];

- Forward dynamics via $\ddot{q} = -M^{-1}(\tau - \text{RNEA}(q, \dot{q}, 0))$ [16];

- The analytical gradient of inverse dynamics (∇RNEA) with respect to the robot's position and velocity $(q, \dot{q})$ [1];

- The analytical gradient of forward dynamics with respect to the robot's position, velocity, and input torque $(q, \dot{q}, \tau)$ via $\nabla\ddot{q} = -M^{-1}\nabla\text{RNEA}(q, \dot{q}, \ddot{q})$ [1].

Directions for future work include extending this core with additional algorithms and joint types (see Section 5.6).

5.4   GRiD's Design and Optimizations

This section describes the hardware-software co-design of GRiD's optimized URDF driven code generation aimed at producing GPU-accelerated rigid body dynamics algorithms and their gradients for use with whole-body, nonlinear MPC. This section begins by introducing the key algorithmic features of spatial-algebra-based rigid body dynamics algorithms and their gradients (Section 5.4.1), and then describes how GRiD refactors these algorithms to better map them to GPU hardware (Section 5.4.2). In this section we use the gradient of inverse dynamics (Algorithm 3) to describe our refactoring approach, but note that the other rigid body dynamics algorithms implemented by GRiD, as described in Section 5.3, exhibit similar computational patterns and are optimized and accelerated in similar ways (see Appendix A for other refactorings).

<div align="center">**Algorithm 3:** $\nabla RNEA(\dot{q}, v, a, f, X, S, I) \rightarrow \partial c/\partial u$</div>

1: **for** frame $i = 1 : N$ **do**

2: $\qquad \frac{\partial v_i}{\partial u} = {}^i X_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} \left({}^i X_{\lambda_i} v_{\lambda_i}\right) \times S_i & u \equiv q \\ \\ S_i & u \equiv \dot{q} \end{cases}$

3: $\qquad \frac{\partial a_i}{\partial u} = {}^i X_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} \left({}^i X_{\lambda_i} a_{\lambda_i}\right) \times S_i \\ \\ v_i \times S_i \end{cases}$

4: $\qquad \frac{\partial f_i}{\partial u} = I_i \frac{\partial a_i}{\partial u} + \frac{\partial v_i}{\partial u} \times^* I_i v_i + v_i \times^* I_i \frac{\partial v_i}{\partial u}$

5: **for** frame $i = N : 1$ **do**

6: $\qquad \frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u}$

7: $\qquad \frac{\partial f_{\lambda_i}}{\partial u} += {}^i X_{\lambda_i}^T \frac{\partial f_i}{\partial u} + {}^i X_{\lambda_i}^T \left(S_i \times^* f_i\right)$

5.4.1   Key Features of Rigid Body Dynamics Algorithms

In order to design accelerated implementations of spatial-algebra-based rigid body dynamics algorithms for use with nonlinear MPC, it is important to first identify the key algorithmic features of the algorithms as these structural properties interact and impact the computation differently, and in compounding ways, on different hardware platform.

Spatial-algebra-based rigid body dynamics algorithms represents most intermediate quantities during computations as operations over vectors in $\mathbb{R}^6$ and matrices in $\mathbb{R}^{6 \times 6}$, defined in the frame of each rigid body. These frames are numbered $i = 1$ to n such that each body's parent λ_i is a lower number. Most rigid body dynamics algorithms operate via outward and inward loops over these frames collecting and transforming forces, accelerations, velocities, and inertias. Transformation matrices from frame λ_i to i are denoted as ${}^i X_{\lambda_i}$ and can be constructed from the rotation and translation between the two coordinate frames, which themselves are functions of the joint position q_i between those frames and constants derived from the robot's topology. The mass distribution of each link is denoted by its spatial inertia I_i, and S_i is a joint-dependent term denoting in which directions a joint can move (and is often a constant). Finally, spatial algebra uses spatial cross product operators $\times$ and $\times^*$, in which a

vector is re-ordered into a matrix, and then a standard matrix multiplication is performed. This reordering is shown in Equation 5.4 for $v \in \mathbb{R}^6$:

$$v\times = \begin{bmatrix} 0 & -v[2] & v[1] & 0 & 0 & 0 \\ v[2] & 0 & -v[0] & 0 & 0 & 0 \\ -v[1] & v[0] & 0 & 0 & 0 & 0 \\ 0 & -v[5] & v[4] & 0 & -v[2] & v[1] \\ v[5] & 0 & -v[3] & v[2] & 0 & -v[0] \\ -v[4] & v[3] & 0 & -v[1] & v[0] & 0 \end{bmatrix} \tag{5.4}$$

$$v\times^* = -v \times^T .$$

As mentioned above, in this section we'll explore the computational patterns found in these algorithms through the lens of the gradient of inverse dynamics (Algorithm 3).

Coarse-Grained Parallelism: As mentioned in Chapter 2, many modern nonlinear MPC implementations have a step that requires tens to hundreds of independent computations of the gradient of rigid body dynamics [3; 25; 37; 65; 66], offering parallelism across these long-running, independent computations.

Fine-Grained Parallelism: Algorithm 3 contains opportunities for additional shorter-duration parallelism between columns of partial derivatives and also in low-level mathematical operations, e.g., between the many independent dot-product operations within a single matrix-matrix, or matrix-vector, multiplication. For example, the computation of each column j of $\partial\{v, a, f, c\}/\partial u_j$ can be computed in parallel. And, within each column, each value i of $\partial\{v, a, f, c\}_i/\partial u_j$ can be computed through parallel dot-products and scalar additions.

Structured Sparsity: The underlying matrices used throughout the algorithm exhibit sparsity patterns that can be derived from the robot's topology. For example, robots with only revolute joints can be described such that all $S_i = [0, 0, 1, 0, 0, 0]^T$. In this way, all computations that are right-multiplied by S_i can be reduced to only computing and extracting the third column (or value), which can remove as much as 83% of the computations. There are also opportunities to exploit structured sparsity in the transformation matrices, $^iX_{\lambda_i}$, inertia matrices, I_i, and cross product matrices, $\times$ and $\times^*$, which are known to be 30% to

60% sparse [86]. These patterns are also defined at compile time, as they are based on the structure of the robot's kinematic tree or by construction, as shown in Equation 5.4.

Data Access Patterns: Algorithm 3 exhibits regular access patterns through its ordered loops (assuming the local variables for each frame i are stored regularly in memory). This enables quick and easy computations of memory address locations, batching of loads and stores, and even "pre-fetching" of anticipated memory loads in advance. However, the cross product operations are reorderings, leading to irregular memory access patterns.

Sequential Dependencies: Throughout the algorithm, local variables have references to parent or child frames whose values are computed in previous loop iterations. For example, the computations of $\partial\{v, a, f\}/\partial u$ in lines 2, 3, and 7.

Working Set Size: Algorithm 3 has a relatively small working set. It most frequently accesses only a few local variables between loop iterations, and a set of small matrices, in particular, I_i and ${}^iX_{\lambda_i} \in \mathbb{R}^{6 \times 6}$. This means that the working set can easily fit into small fast hardware memory structures, e.g., caches.

I/O Overhead: Algorithm 3 requires a handful of different input values and produces a matrix of output values which can require that a substantial amount of input and output (I/O) data is sent to and received from hardware accelerators like GPUs. This can introduce long latency delays in the system, slowing performance.

5.4.2 Mapping Rigid Body Dynamics Algorithms to the GPU

As mention in Chapter 3, as compared to a CPU, a GPU is a much larger set of very simple processors, optimized specifically for parallel computations with identical instructions over large working sets of data (e.g., large matrix-matrix multiplication). For maximal performance, the GPU requires groups of threads within each thread block to compute the same operation on memory accessed via regular patterns. As such, it is highly optimized for some types of native parallelism present in our application, but is inefficient on others. Table 5.1 offers

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

Table 5.1: *Algorithmic features of the gradient of rigid body dynamics and qualitative assessments of their suitability for different target hardware platforms. We find that in general, rigid body dynamics algorithms when used in whole-body, nonlinear MPC algorithms are naturally well suited for the CPU, but not for the GPU, outside of opportunities for coarse-grained parallelism between computations.*

qualitative assessments of how well these algorithmic features, described in Section 5.4.1, can be exploited by the CPU and GPU. This informed which advantageous features we could leverage, or disadvantageous bottlenecks we had to mitigate in our design.

As noted by the "excellent" rating in Table 5.1, the GPU can use blocks of threads to efficiently take advantage of large-scale, coarse-grained parallelism across many independent dynamics computations. However, while there exist many opportunities for fine-grained parallelism within each computation, as mentioned in the previous section, this parallelism can be harder to exploit effectively on a GPU as there are many different low-level operations that can occur simultaneously. For example, matrix vector multiplications, cross products, and additions all appear in lines 2, 3, 4, and 7 of Algorithm 3. Without careful refactoring, this leads to substantial amounts of thread divergence and low overall thread occupancy. This issue is compounded by the sequential dependencies both between parent and child frames as well as between the temporary variables $\partial\{v, a, f\}/\partial u$ which require multiple synchronization points within every loop iteration. Finally, GPUs typically run at about half the clock speed of CPUs (e.g., $1.44 - 1.7$GHz versus $3.6 - 3.8$GHz for the GPUs and CPUs evaluated in Section 5.5), further hindering their performance on sequential code. These issues are the major performance limitation of previous GPU implementations [3].

To address these bottlenecks, the core of our optimized GPU implementation is a refactored version of Algorithm 3, shown in Algorithm 4. In this refactoring, we moved computations with identical operations into the same parallel step to better fit the hardware's computational model. For example, in line 2 we compute a series of matrix vector multiplications, followed by a series of cross product operations in line 3. We also re-ordered the computations to minimize the amount of work done in serial loops, the main driver of sequential dependencies, at the expense of generating even more temporary values. For example, we precomputed α in lines 2 and 3 of the initial parallel computations to reduce the amount of work done in the later serial loop in lines 4-6.
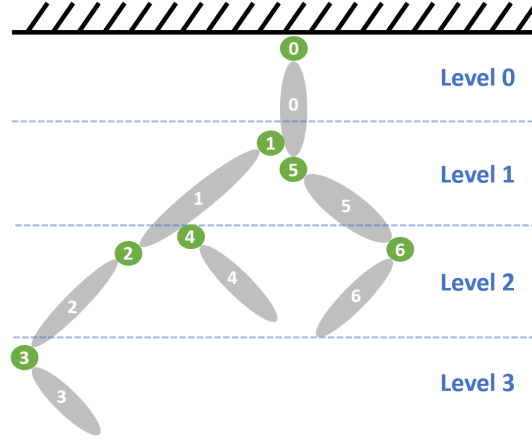
Figure 5.3: *An example robot topology.*

We also inject additional optimizations to take advantage of further parallelism offered by robot models with multiple branching points at different levels of the kinematic tree. Since dependencies in the serial passes of rigid body dynamics algorithms are between parent and child frames in the tree, we can compute "sibling" frames in parallel. For example, the first loop of the ∇RNEA algorithm (Algorithm 3) computes the temporary variables $\partial v_i, \partial a_i$ for frame i as a function of $\partial v_{\lambda_i}, \partial a_{\lambda_i}$ for its parent frame λ_i (Lines 2 and 3). Therefore, we can compute each $\partial v_i, \partial a_i$ by stepping serially through the levels of the tree, while computing all frames within each level in parallel (Algorithm 4 lines 4-6 and 9-10). For the robot shown in Figure 5.3, we compute the values associated with frame 0, then 1 and 5 in parallel, then 2, 4,

and 6 in parallel, and finally 3. GRiD also performs loop unrolling on these remaining serial loops to enable the compiler to easily optimize the resulting code. We also note that once all $\partial v, \partial a$ have been computed, all ∂f can be computed fully in parallel as each ∂f_i only references other values for frame i. That all said, due to the highly serial nature of the algorithm, serial operations and synchronization points still existed in our final implementation.

Algorithm 4: $\nabla RNEA\text{-}GRiD(\dot{q}, v, a, f, X, S, I) \rightarrow \partial f/\partial u$

1: **for** frame $i = 1 : n$ **in parallel do**

2: $\qquad \alpha_i = {}^iX_{\lambda_i} v_{\lambda_i} \quad \beta_i = {}^iX_{\lambda_i} a_{\lambda_i} \quad \gamma_i = I_i v_i$

3: $\qquad \alpha_i = \alpha_i \times S_i \quad \beta_i = \beta_i \times S_i \quad \delta_i = v_i \times S_i$

4: **for** level $l = 0 : l_{max}$ **do**

5: $\qquad$ **for** frame $i \in l$ **in parallel do**

6: $\qquad\qquad \frac{\partial v_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} \alpha_i & u \equiv q \\ S_i & u \equiv \dot{q} \end{cases}$

7: **for** frame $i = 1 : n$ **in parallel do**

8: $\qquad \rho_i = \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} \beta_i \\ \delta_i \end{cases}$

9: **for** level $l = 0 : l_{max}$ **do**

10: $\qquad$ **for** frame $i \in l$ **in parallel do**

11: $\qquad\qquad \frac{\partial a_i}{\partial u} = {}^iX_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \rho_i$

12: **for** frame $i = 1 : n$ **in parallel do**

13: $\qquad \frac{\partial f_i}{\partial u} = \frac{\partial v_i}{\partial u} \times^* \gamma_i \quad \eta_i = v_i \times^* I_i \quad \zeta_i = S_i \times^* f_i$

14: $\qquad \frac{\partial f_i}{\partial u} = \frac{\partial f_i}{\partial u} + I_i \frac{\partial a_i}{\partial u} + \eta_i \frac{\partial v_i}{\partial u} \qquad \zeta_i = {}^iX_{\lambda_i}^T \zeta_i$

15: **for** level $l = l_{max} : 0$ **do**

16: $\qquad$ **for** frame $i \in l$ **in parallel do**

17: $\qquad\qquad \frac{\partial f_{\lambda_i}}{\partial u} \mathrel{+}= {}^iX_{\lambda_i}^T \frac{\partial f_i}{\partial u} + \zeta_i$

18: **for** frame $i = n : 1$ **in parallel do**

19: $\qquad \frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u}$

Out-of-order computations and irregular data access patterns are also inefficient to compute on a GPU. It is important to avoid these, even at the expense of creating large numbers of temporary variables. Due to the small working set size, even after generating these additional temporary values we should, in theory, be able to fit everything in the GPU's shared memory (cache), minimizing latency penalties. Therefore, in our GPU implementation, and unlike in state-of-the-art CPU implementations [85; 86; 100; 102; 123], we did not exploit the sparsity in the X, $\times$, and $\times^*$ matrices. Instead, we used standard threaded matrix multiplication for the X matrices. For the $\times$ and $\times^*$ matrices we initially used a two-step process to first create temporary matrices to capture the re-ordering, and then used standard threaded matrix multiplication to compute the final values. However, when supporting arbitrarily large robots we realized that while the working set size of the serial algorithm is small, by refactoring the algorithm to expose more parallelism, we generated an algorithm with a working set size which grows proportionally to the number of frames in a robot model. Therefore, in order to ensure that all of the temporary variables fit into the GPU cache, at code generation time, GRiD determines if it is necessary to forgo any temporary memory computations in order to support robots with many degrees-of-freedom (dof). For example, for the 30 dof Atlas humanoid, GRiD does not compute each $v\times$ matrix in parallel and then use threaded matrix multiplication (like we can do for the IIWA manipulator [85]), but instead computes $v_1 \times v_2$ in a few parallel threads, trading off a slight latency penalty for a large savings in shared memory usage.

GRiD also leverages the robot's topology to determine sparsity patterns in the temporary variables needed for the gradient computations. As such, columns of temporary memory variables that would be all zeros are skipped and shared memory is compressed to effectively remove those columns. For most robot models this leads to significant savings. For example, reducing shared memory usage for the the quadruped robot HyQ [124] by more than 60%. While this does complicate the memory addressing schema, most required offsets are computed and cached at code generation time, minimizing the impact on latency. Furthermore, GRiD employs non-branching *if/else* constructs (e.g., `result = flag*val1 + !flag*val2`) to avoid branching for all other offsets and control flow switches where possible.

Finally, because data needs to be transferred between the GPU and a host CPU, I/O is a serious constraint for GPUs. As such, we took a series of steps to minimize its impact on overall performance. For example, to reduce the total I/O required by our GPU implementations, we preloaded the constant I matrices and constant parts of the X matrices at startup. We then explored a series of *split* and *fused* GPU kernels to analyze the I/O and compute trade-offs induced by different problem partitionings between the CPU and GPU. These experiments are detailed in Section 5.5 and ultimately reveal that the small latency tradeoffs of increased computation on the GPU generally outweigh the I/O and synchronization overhead of leveraging the CPU for those computations.

For memory transfers, we used the NVIDIA CUDA [60] library's built-in functions for transferring data to and from the GPU over PCIe Gen3. To better leverage the PCIe data bus, we ensured that all values were stored as a single contiguous block of memory. And, as suggested by NVIDIA, we copied all of the needed data over to the GPU memory once, let it run, and then copied all of the results back.

GRiD also employs further optimizations for certain classes of robot models. For example, for all single chain robots, the parent's frame number is always one less than the child's. For these robots, the code generated by GRiD will remove any indirect references to the parent (or child) frame number and instead simply subtract (or add) one.

Finally, as noted at the beginning of this section, while we use the gradient of inverse dynamics as the representative kernel for this section, GRiD applies similar patterns of refactorings, memory compressions, and computational optimizations across all of the algorithms described in Section 5.3. These refactorings can be found in Appendix A.

5.5   Benchmark Timing Results

We evaluated two timing metrics to understand the performance of our designs: the latency of a single computation of the algorithm, and the end-to-end latency (including I/O and

other overhead) to execute a set of N computations. We compared GRiD against baseline implementations on the CPU and GPU and against different problem partitionings of our own implementations. In general, we compare timing results across three robot models: the 7 degrees-of-freedom (dof) Kuka LBR IIWA-14 manipulator [125], the 12 dof HyQ quadruped [124], and the 30 dof Atlas humanoid [24; 126]. For single computation and multiple computation latency, we took the average of one million, and one hundred thousand trials, respectively. For clean timing measurements on the CPU, we disabled TurboBoost, and fixed the clock frequency to the maximum. We measured time with the Linux system call `clock_gettime()`, using `CLOCK_MONOTONIC` as the source. Overall, we find that GRiD scales to higher degrees of freedom systems and to higher numbers of parallel computations better than baseline GPU and state-of-the-art CPU implementations. We also find that this performance is maximized when we move as many of the computations onto the GPU as possible, limiting I/O and synchronization overheads. This results in GRiD providing as much as a 7.2x computational speedup over the CPU and maintaining as much as a 2.5x speedup when accounting for I/O communication overhead.

5.5.1 Proof-Of-Concept Evaluations

In this section we compare our proof-of-concept optimized GPU implementation of the gradient of forward dynamics for the Kuka LBR IIWA-14 manipulator [125] against a baseline CPU and GPU implementation. We analyze both the latency of a single computation, as well as problem partitioning between the CPU and GPU, and the throughput of multiple computations for use with nonlinear MPC.

For the GPU baseline we used an existing hand-optimized implementation of rigid body dynamics and its analytical gradients [3], as no full library exists for comparisons. For the CPU baseline, we implemented our own hand-optimized CPU implementation leveraging algorithmic and implementation insights from existing state-of-the-art CPU libraries to ensure a fair comparison with our optimized GPU implementation [100; 102; 103]. For more information on the CPU baseline implementation see Appendix B. For these evalua-

tions we used a high-performance workstation with a 3.6GHz quad-core Intel Core i7-7700 CPU and 1.7GHz NVIDIA GeForce GTX 2080 GPU running Ubuntu 18.04, CUDA 11.0, Clang 10, and g++7.4. Source code accompanying these evaluations can be found at https://github.com/plancherb1/fast-rbd-gradients.

Single Call Latency

The latency for a single computation of the baseline GPU implementation [3], our custom baseline CPU implementation, and our proof-of-concept optimized GPU implementation of the gradient of forward dynamics, excluding overheads is shown in Figure 5.4. We break the latency time down into three steps:

- Yellow (RNEA): Computing the initial temporary variables v, a, f, X, S, and I.

- Blue (∇RNEA): Using those values to compute the temporary variable $\partial c/\partial u$.

- Orange: Computing the final output by multiplying $\partial c/\partial u$ by the input M^{-1}.

	GPU Baseline	GPU Proof-of-Concept	CPU Baseline
■ dc/du	49.29	5.37	2.65
■ v, a, f	3.89	2.71	0.34
■ $d\ddot{q}/du$	1.23	0.36	0.44
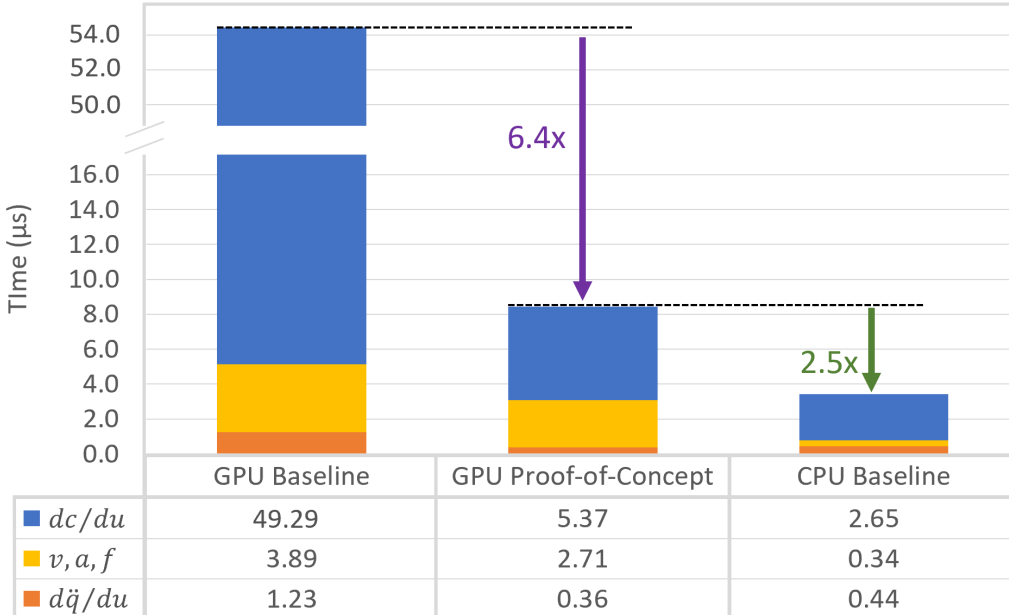
Figure 5.4: *Latency of one computation of the gradient of rigid body dynamics for the Kuka manipulator in the CPU and GPU baseline implementations, as compared to our proof-of-concept optimized GPU implementation. We find that our proof-of-concept outperforms the baseline by 6.4x but is still 2.5x slower than the CPU.*

The GPU implementations struggled against the CPU. Our proof-of-concept optimized implementation was outperformed by the CPU by 2.5x in this single computation test. This is due to the fact that GPUs derive their benefit from throughput offered by large-scale parallelism, not available with only one computation for the single branched IIWA manipulator. However, for our optimized implementation, while the GPU v, a, f latency is 8.0x slower than the CPU, the $\partial c/\partial u$ latency is only 2.0x slower. This improved scaling is the result of the re-factoring done in Section 5.4 to expose more parallelism, and the increased level of parallelism available in that step of the algorithm. Leveraging these optimizations, our GPU implementation is 6.4x faster than the existing state-of-the-art [3], driven by a 9.2x speedup in the $\partial c/\partial u$ step. We also note that the final M^{-1} multiplication step, shown in orange, is actually faster for the GPU (0.36μs) than the CPU (0.44μs) as it is a standard, parallel-friendly, matrix multiplication.

Problem Partitioning

In this section we explore the impact of changing the problem partitioning between the CPU and GPU on the end-to-end latency of our implementations. Figure 5.5 compares (from left to right within each group) "[s]plit", "[f]used", and "[c]ompletely fused" accelerated [G]PU kernels for the gradient of forward dynamics (Equation 5.3). In the *split* kernel we only computed the most parallel and compute-intensive section of the computation on the GPU, $\partial c/\partial u$ (Algorithm 4), while computing the inputs to the algorithm v, a, f, X, S, and I, as well as the final multiplication with M^{-1} on the host CPU. In the initial *fused* kernel we minimized I/O by computing both the inputs to the algorithm, v, a, f, X, S, and I, as well as $\partial c/\partial u$ on the GPU. Finally, we also designed a *completely-fused* kernel which additionally takes M^{-1} as an input and does the whole computation on the GPU. This lead to an increase in I/O as compared to the fused kernel, but reduces both the number of synchronization points with, as well as the total amount of computation done, on the CPU.

Figure 5.5 shows that by moving more of the computation onto the GPU increased the computational latency, as expected. However, since the $\partial c/\partial u$ computation is by far the most
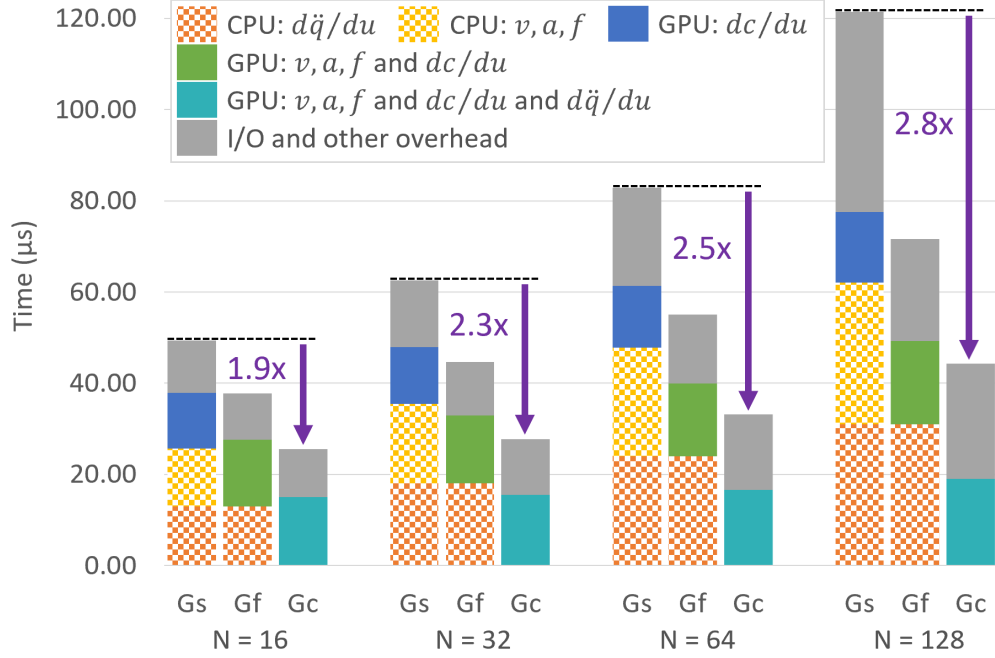
Figure 5.5: *Runtime of $N = 16, 32, 64,$ and 128 computations of our accelerated implementations of the dynamics gradient kernel for the Kuka manipulator using different problem partitionings between the CPU and [G]PU coprocessor: the [s]plit, [f]used, and [c]ompletely-fused kernels. We find that removing synchronization points and moving more computations onto the GPU reduces overall latency.*

expensive computation, and the GPU can easily compute many computations in parallel, the increase is quite small. This can be seen in the slight increase from the blue to green to teal bar in each group.

At the same time, the fused kernels have reduced I/O overhead, shown in the grey bars, which is more important as N increases. Furthermore, removing the high-level algorithmic synchronization points between the CPU and GPU, and the corresponding batched CPU computations, shown by the checkered orange and yellow bars, greatly reduced overall latency.

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

End-to-End CPU and GPU Comparisons

Figure 5.6 compares our most-optimized proof-of-concept implementations across all hardware platforms: the [C]PU implementation and the [G]PU [c]ompletely-fused implementations.

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

The second bar is the "Gc" kernel which was able to provide a 1.2x to 3.0x performance improvement over the CPU design for $N = 16, 128$ respectively. The GPU performed better as the number of computations increased and despite being much slower than the CPU on a single computation (Figure 5.4), outperformed the CPU on all tests using multiple computations. This is driven mainly by the GPU's ability to take advantage of both coarse-grained and fine-grained parallelism and efficiently scale to multiple computations.
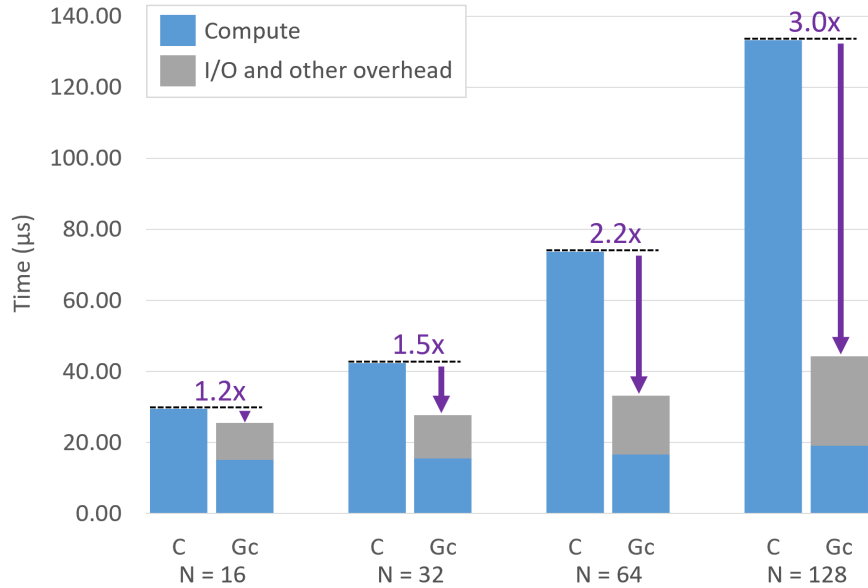
Figure 5.6: *Runtime of $N = 16$, 32, 64, and 128 computations of the accelerated dynamics gradient kernels for the Kuka manipulator for the [C]PU and [G]pu using the [c]ompletely-fused kernel. We find that the GPU outperforms the CPU by 1.2x (N=16) to 3.0x (N=128).*

5.5.2   GRiD Benchmark Evaluations

In this section we build on the results from the proof-of-concept implementation and benchmark the final GRiD library against a state-of-the art CPU baseline, the Pinocchio library [100][1], which supports optimized CPU code generation of both rigid body dynamics algorithms and their analytical gradients. As with the proof-of-concept evaluations we compare both the latency and throughput as well as the scalability of the implementations. In these benchmarks we used a high-performance workstation with a 3.8GHz eight-core Intel Core i7-10700K CPU and a 1.44GHz NVIDIA GeForce RTX 3080 GPU running Ubuntu 20.04, CUDA 11.4, `Clang 12`, and `g++9.4`. We compare timing results across three representative robot models: the 7 degrees-of-freedom (dof) Kuka LBR IIWA-14 manipulator [125] used in the proof-of-concept evaluations, the 12 dof HyQ quadruped [124], and the 30 dof Atlas humanoid [24; 126]. Source code accompanying these evaluations can be found at https://github.com/robot-acceleration/GRiDBenchmarks.

Single Computation Latency Scaling

In this secion we evaluate the latency scaling, excluding I/O overheads, of a single computation of each rigid body dynamics algorithm, from IIWA to HyQ and IIWA to Atlas, on both the CPU and GPU in Figure 5.7, and list absolute timings in Table 5.2. We also plot the scaling of the robots' dof as a measure of their computational complexity.

We find that the GPU is able to scale to more complex robots and algorithms better than the CPU by taking advantage of fine-grained parallelism induced by independent robot limbs and the independent columns of gradient computations. This is why GPUs are most advantageous when used on computations where they can take advantage of both fine and coarse-grained parallelism. As expected (and consistent with Section 5.5.1), the GPU is slower on a single computation than the CPU, but the GPU demonstrates better scalability across both algorithm and robot complexity. For example, as shown in Table 5.2, for ∇FD, the CPU is 4.4x faster

---

[1]We used the pinocchio3-preview branch to ensure we were using the latest, most optimized, code.

than the GPU ($2.9\mu s$ vs. $12.9\mu s$), but only 2.0x faster for Atlas ($20.9\mu s$ vs. $42.1\mu s$). That said, the CPU is still faster than the GPU for all individual computations, showing that GPU acceleration only makes sense when there is sufficient parallel work to be done.

On the CPU, the latency of each algorithm scales directly with its computational intensity, with the gradients requiring significantly more computation (see Table 5.2). The most computationally intensive algorithm, the forward dynamics gradient (∇FD), takes 2.9, 4.3, and 20.9 μs for IIWA, HyQ, and Atlas, while the simplest algorithm, inverse dynamics (ID) takes 0.3, 0.3, and 1.1 μs—a 9.7x to 19.0x slowdown.
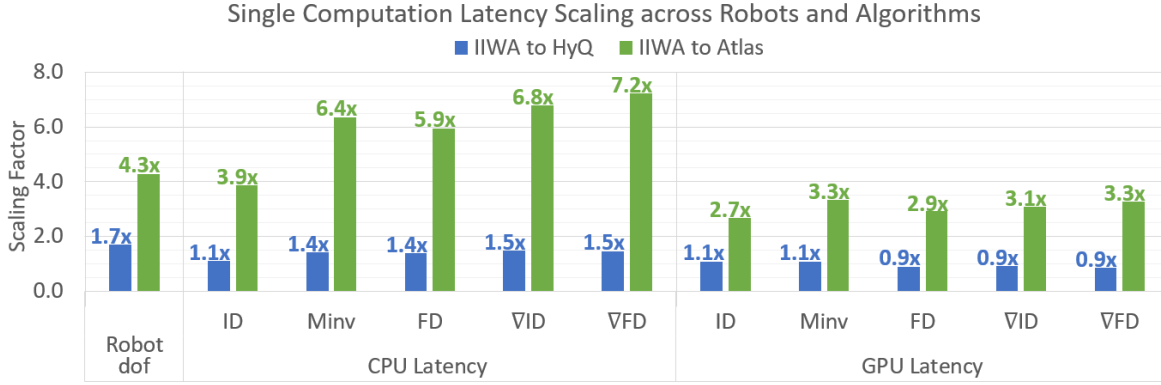
Figure 5.7: *The scaling of single computation latency from IIWA to HyQ and IIWA to Atlas for both the Pinocchio CPU baseline and the GRiD GPU library for various rigid body dynamics algorithms (ID = Inverse Dynamics, Minv = Direct Minv, FD = Forward Dynamics and ∇ indicates the gradient of that algorithm). We also plot the scaling of the robots' dof as a measure of their increased complexity. We find that the GPU is able to scale to more complex robots and algorithms better than the CPU by taking advantage of fine-grained parallelism induced by independent robot limbs and the independent columns of gradient computations.*

Table 5.2: *Single Computation Latency in μs Per Algorithm and Robot (ID = Inverse Dynamics, Minv = Direct Minv, FD = Forward Dynamics and ∇ indicates the gradient of that algorithm)*

Algorithm	CPU			GPU		
	IIWA	HyQ	Atlas	IIWA	HyQ	Atlas
ID	0.3	0.3	1.1	3.0	3.2	8.0
Minv	0.5	0.8	3.4	5.2	5.6	17.4
FD	0.9	1.2	5.3	7.7	6.9	22.4
∇ID	1.4	2.1	9.8	6.3	5.8	19.5
∇FD	2.9	4.3	20.9	12.9	11.0	42.1

60

CPU latency also scales with the dof of the robot (Figure 5.7). For example, as the robot's dof increases by a factor of 1.7x from IIWA to HyQ, the computation time also increase by 1.1x, for the $O(N)$ ID algorithm, up to 1.5x, for the $O(N^2)$ ∇FD algorithm. It appears that this strong performance is due to the code generation taking advantage of the the many shared computations in the gradients, as well as the sparsity induced by HyQ's independent limbs, which decrease the longest path through the rigid body tree from 7 on IIWA to 3 on HyQ. However, these optimizations are mitigated by the Atlas model, which has a much larger 30 dof, and a longest path through the rigid body tree of 8. Atlas has 4.3x the dof of IIWA, but has a 3.9x (ID) to 7.2x (∇FD) slowdown on the CPU.

By contrast, the GPU is able improve its scalability by not only taking advantage of sparsity and shared computations, but *also the opportunities for fine-grained parallelism* caused by both independent limbs in complex robot models, and independent columns of the gradient computations. For example, Table 5.2 shows that by taking advantage of parallelism in the gradient computations, the GPU is not only able to compute ∇ID faster than FD, but also only takes 12.9, 11.0, and 42.1 μs (for IIWA, HyQ, Atlas) for ∇FD as compared to 3.0, 3.2, and 8.0 μs for ID—a slowdown of only 3.4x to 5.3x, and a significant reduction from the CPU's 9.7x to 19.0x slowdown for these algorithms. Similarly, Figure 5.7 shows that by leveraging limb-based parallelism, the GPU computes forward dynamics (FD) and both gradients ($\nabla ID, \nabla FD$) faster for HyQ than for IIWA, and only has a 2.7x to 3.3x slowdown from IIWA to Atlas, again a significant reduction from the CPU's 3.9x to 7.2x.

Multiple Computation Latency

To characterize GRiD's performance in a typical nonlinear trajectory optimization scenario, which uses tens to hundreds of naturally parallel computations of dynamics algorithms, we evaluate the latency for $N = 16$, 32, 64, 128, and 256 computations of the gradient of forward dynamics using Pinocchio and GRiD across robot models in Figure 5.8. These times are broken down into computation time on the CPU or GPU and the GPU I/O overhead. The plot is overlaid with the speedup (or slowdown) of GRiD compared to Pinocchio in pure

computation alone, and also including I/O overhead. We use the gradient of forward dynamics as our representative kernel because it uses many of the other kernels as sub-routines and is the most computationally intense kernel, clearly demonstrating scaling trends.

The GPU outperforms the CPU on all but one of the multiple computation latency tests, even when accounting for I/O. In the one test where the CPU is faster—for the fewest computations, including I/O, for IIWA, the smallest robot with only a single limb—the GPU is still 0.9x as fast. We note that the improved performance on the CPU as compared to Section 5.5.1 can be explained by a combination of the high performance of the Pinocchio library's code generation, the addition of the computation of M^{-1}, as well as the increase from a 4 core to an 8 core CPU, greatly improving the CPU's parallel performance.

Even on the CPU, this benchmark shows how important it is to take advantage of coarse-grained parallelism between computations. For example, the gradient of forward dynamics
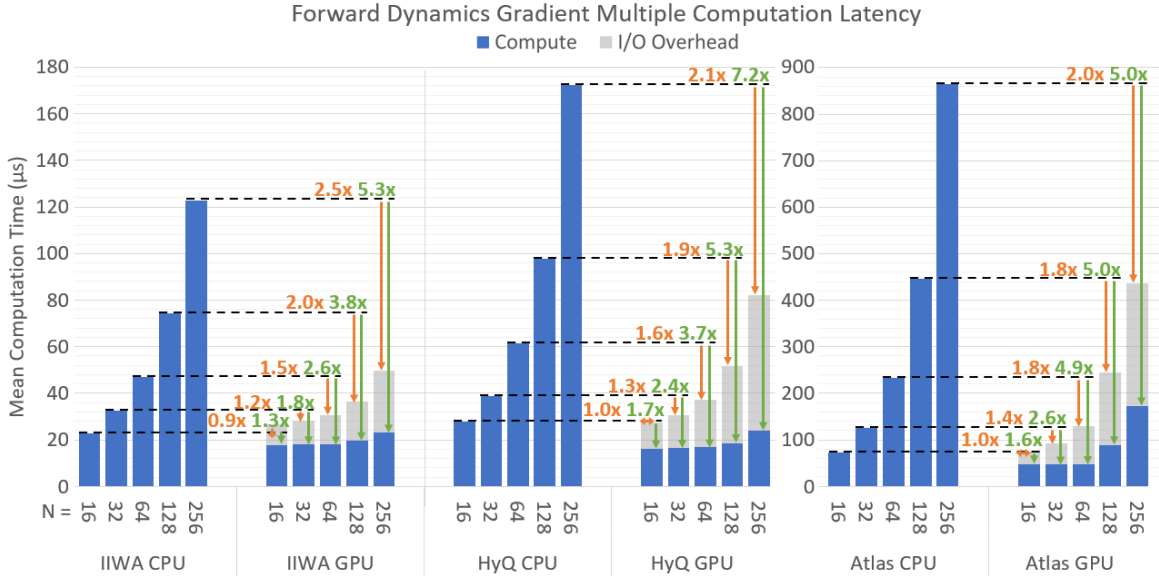
Figure 5.8: *Latency (including GPU I/O overhead) for N = 16, 32, 64, 128, and 256 computations of the gradient of forward dynamics for both the Pinocchio CPU baseline and the GRiD GPU library for various robot models (IIWA, HyQ, and Atlas). Overlayed is the speedup (or slowdown) of GRiD as compared to Pinocchio both in terms of pure computation and including I/O overhead. We find that in most cases the GPU outperforms the CPU and that outperformance increases as N increases. However, I/O overhead is an increasing concern as N grows.*

kernel (∇FD in Table 5.2) took 2.9, 4.3, and 20.9 μs for a single computation for IIWA, HyQ, and Atlas respectively. If we ran it 256 times serially it would therefore take over 742, 1091, and 5355 μs. As Figure 5.8 shows, computing it in parallel on 8-cores only takes 123, 172, 865 μs, saving 83-84% of the computation time.

However, since the CPU only has 8 cores, it is unable to efficiently scale to take advantage of high numbers of naturally parallel computations, taking 5.4x, 6.2x, and 11.8x as long to compute $N = 256$ as compared to $N = 16$ for IIWA, HyQ, and Atlas respectively.

The GPU, on the other hand, is designed to scale to higher numbers of computations without incurring a latency penalty by launching independent blocks of threads for each computation. In fact, for IIWA and HyQ, $N = 256$ takes only 1.3x and 1.5x as long as $N = 16$. This leads to the GPU outperforming the CPU by 5.3x and 7.2x for $N = 256$, and maintaining a 2.5x and 2.1x speedup when including I/O.

For the much higher-dof Atlas robot, the GPU still outperforms the CPU for $N = 256$ by 5.0x, and 2.0x when including I/O. However, unlike with IIWA and HyQ, this performance increase is almost identical to the increase at $N = 64, 128$. This stall in performance improvement is caused by the large amount of shared memory needed for Atlas's 30 dof which limits the number of parallel blocks of threads that can fit concurrently on the GPU hardware. This shows that while GPUs offer better scalability than CPUs, they too have physical hardware limitations that need to be considered.

We also note that for both the CPU and GPU, the variance in timing results decreases as the computational complexity increases. The GPU is very reliable with more than 90% of times within 1% of the mean for all experiments and achieves over 99% of times within 1% of the mean for $N \geq 64$ for Atlas. The CPU, on the other hand, only has 95%, 87%, and 86% of times within 1% of the mean for $N = 256$ for Atlas, HyQ, and IIWA respectively, which drops to 81%, 73%, and 54% for $N = 16$. This suggests that GPU acceleration may be able to provide both improved end-to-end latency, as well as improved reliability.

Finally, we note that for the GPU, I/O overhead accounts for 53-71% of the total time for $N = 256$. This indicates that GRiD can provide the highest performance if integrated directly into an entirely GPU-based algorithm, instead of being used to accelerate a step of a CPU-based algorithm. In either case, however, if there is sufficient parallel work to be done, GRiD can reduce overall computational latency.

5.6 Conclusion and Future Work

In this chapter, we introduced GRiD, a GPU-accelerated rigid body dynamics library with analytical gradients. We found that by leveraging large-scale parallelism when performing multiple computations of rigid body dynamics algorithms, GRiD can provide as much as a 7.2x speedup over a state-of-the-art, multi-threaded CPU implementation and maintains as much as a 2.5x speedup when including I/O overhead.

There are many promising directions for future work to extend the functionality and versatility of the GRiD library. We have current work under development to expand GRiD to support the full breadth of rigid body dynamics algorithms and robot models supported by current state-of-the-art CPU spatial-algebra-based rigid body dynamics libraries [100; 102; 108; 109; 110]. Additionally, we are developing wrappers to our C++ host functions in higher level languages to make it even easier to leverage GRiD.

In future work, we would like to explore emerging rigid body dynamics algorithms and alternate formulations and implementations of rigid body dynamics, which may improve overall performance by exposing additional opportunities for parallelism, structured sparsity, and improved computational efficiency [99; 127; 128; 129; 130; 131; 132; 133; 134].

We would also like to add support for differentiating through model parameters [135; 136], as well as for contact, and hope to integrate these accelerated dynamics implementations into existing robotics software frameworks [3; 48; 137; 138; 139]. This would increase both GRiD's ease-of-use and applicability to more robotics researchers.

Finally, building out increased support for more trajectory optimization, MPC, and ML algorithms running entirely on the GPU would further increase the performance benefits from integrating GRiD into these approaches as full GPU algorithms would eliminate the I/O and synchronization overheads between the CPU and GPU.

Chapter 6

Towards MPC with GPU Accelerated Direct Trajectory Optimization

This chapter explores the design of direct trajectory optimization methods that admit more natural parallelism than the DDP based methods explored in Chapter 4, enabling more efficient GPU parallelism and improved acceleration. We first derive our GPU accelerated direct method, leveraging a structure exploiting Krylov subspace solver and a parallel symmetric stair preconditioner. We then show preliminary results indicating that this algorithmic approach results in improvements in condition number and spectral radius, leading to up to a 3.1x reduction in iterations-to-converge on standard trajectory optimization problems. We also find that our solver can be used for whole-body, nonlinear MPC.

6.1   Related Work

There has been a significant amount of prior work developing general purpose sparse linear system solvers on the GPU. Much of this work focus on designing factorization based approaches [140; 141; 142; 143; 144; 145], and Krylov subspace solvers [54; 55; 57; 74; 146; 147; 148], that can efficient support general purpose sparse matrices and perform efficient

sparse matrix linear algebra operations. There has also been work developing and implementing Block-Cyclic-Reduction and other tree-structured methods [149; 150; 151; 152] that are optimized for block-tridiagonal systems.

For the nonlinear trajectory optimization problem, evolutionary, particle-swarm, Monte-Carlo, and other sampling based approaches have been implemented on GPUs [83; 112; 153; 154; 155; 156; 157; 158; 159]. Most prior work on gradient-based parallel nonlinear trajectory optimization has been fully confined to the CPU [64; 93; 94; 160], relied on the CPU for many of the computations [161; 162], focused on optimizing BLAS functions on the GPU [75; 82], or was limited to the naturally parallel Taylor expansions of the dynamics and cost functions [84; 85; 87].

There are two existing lines of work fully implementing gradient-based parallel nonlinear trajectory optimization on the GPU. The first, detailed in Chapter 4, leveraged shooting based methods and found them to not expose much natural parallelism, limiting their performance [3; 68]. The second, used a Block-Cyclic-Reduction-based direct method to exploit the particular structure exposed by position based dynamics [163].

This chapter adds to this literature by designing a parallel symmetric stair preconditioner and a GPU-accelerated, gradient-based, nonlinear, direct trajectory optimization solver that is designed for the the more accurate spatial-algebra-based dynamics discussed in Chapter 5.

6.2   GPU-Accelerated Direct Trajectory Optimization

In this section we describe the design of our GPU-accelerated parallel direct trajectory optimization algorithm. Our algorithm is optimized to use blocks of parallel threads to exploit the sparsity and parallelism exposed by direct methods through a structure exploiting Krylov subspace solver and a parallel symmetric strair preconditioner. Our algorithm is particularly optimized for the problem described in Equation 2.4, a trajectory optimization problem that only has dynamics and an initial state constraint. However, as mentioned previously, this

formulation can support arbitrary constraints through the use of an augmented Lagrangian as it only impacts the the numerical values of the cost and its Jacobian and Hessian, and not their structure [47; 48]. Accordingly, we treat these terms as dense vectors and matrices where appropriate below to provide a more general optimized trajectory optimization solver.

6.2.1   A Structure Exploiting PCG Solver for the GPU

In order to maximize the performance of our PCG solver on the GPU, we need to maximize the amount of work done in parallel and minimize the amount of memory that needs to be shared by the various blocks of threads. We note that the computations in PCG are dominated by the large matrix vector multiplications with S and ϕ^{-1} in lines 5, 6, and 8 of Algorithm 1. These operations produce a vector of size $n * N$ where each of size n blocks of rows can be done naturally in parallel by blocks of threads. Furthermore, if we know the bandwidth of the Schur complement matrix S, which in our case is block tridiagonal, and the bandwidth of the inverse of the preconditioner, Φ^{-1}, which we call κ, we can minimize the amount of blocks of vectors r, p that need to be shared across the blocks of threads. This results in the minimal memory sharing GPU implementation of PCG as the remaining operations only require parallel reductions of scalar values. This structure exploiting implementation of the PCG algorithm optimized for the GPU is shown in Algorithm 5.

6.2.2   A Parallel Block-Tridiagonal Preconditioner

To further optimize the structure exploiting nature of the underlying PCG solver we also need to be able to leverage a parallel preconditioner with a small bandwidth.

One polynomial splitting for block tridiagonal matrices that has been shown to outperform Jacobi and Block-Jacobi methods is the stair based splitting [164; 165]:

$$
S = \begin{bmatrix} A_0 & B_0 & \\ B_1 & A_1 & B_2 \\ & B_3 & A_2 \end{bmatrix} \qquad \Psi = \begin{bmatrix} A_0 & \mathbf{0} & \\ B_1 & A_1 & B_2 \\ & \mathbf{0} & A_2 \end{bmatrix}, \tag{6.1}
$$

<div align="center">

Algorithm 5: *GPU-PCG* $(S, \Phi^{-1}, \gamma, \lambda, \epsilon) \rightarrow \lambda^*$

</div>

1: **for** block $b = 0 : N$ **in parallel do**

2:      $r_b = \gamma_b - S_b \lambda_{b-1:b+1}$

3:      Load $r_{b-\kappa:b-1}, r_{b+1:b+\kappa}$

4:      $\tilde{r}_b, p_b = \Phi_b^{-1} r_{b-\kappa:b+\kappa}$

5:      $\eta_b = r_b^T \tilde{r}_b$

6: $\eta = \textbf{ParallelReduce}(\eta_b)$

} **Initialization**

7: **for** iter $i = 1 : \text{max\_iter}$ **do**

8:      **for** block $b = 0 : N$ **in parallel do**

9:          Load p_{b-1}, p_{b+1}

10:          $\Upsilon_b = S_b p_{b-1:b+1}$

11:          $\upsilon_b = p_b \Upsilon_b$

12:      $\upsilon = \textbf{ParallelReduce}(\upsilon_b)$

13:      **for** block $b = 0 : N$ **in parallel do**

14:          $\alpha = \eta / \upsilon$

15:          $\lambda_b = \lambda_b + \alpha p_b$

16:          $r_b = r_b - \alpha \Upsilon_b$

17:      **for** block $b = 0 : N$ **in parallel do**

18:          Load $r_{b-\kappa:b-1}, r_{b+1:b+\kappa}$

19:          $\tilde{r}_b = \Phi_b^{-1} r_b$

20:          $\eta_b' = r_b^T \tilde{r}_b$

21:      $\eta' = \textbf{ParallelReduce}(\eta_b')$

22:      **if** $\eta' < \epsilon$ **then**

23:          return λ

24:      **for** block $b = 0 : N$ **in parallel do**

25:          $\beta = \eta' / \eta$

26:          $\eta = \eta'$

27:          $p_b = \tilde{r}_b' + \beta p_b$

} **Main Loop**

which has an analytical inverse,

$$\Psi^{-1} = \begin{bmatrix} A_0^{-1} & \mathbf{0} \\ -A_1^{-1}B_1A_0^{-1} & A_1^{-1} & -A_1^{-1}B_2A_2^{-1} \\ \mathbf{0} & A_2^{-1} \end{bmatrix}. \tag{6.2}$$

In order to maximize the structure exploiting nature of our PCG solver, we limit the bandwidth of our preconditioner to be block-tridiagonal. As such we use a 0th order polynomial approximation, setting $\Phi^{-1} = \Psi^{-1}$. We then note that as S is symmetric, its inverse is also symmetric. We can therefore compute a more accurate preconditoner by enforcing symmetry on the preconditoner (and its inverse). This results in the following block-tridiagonal preconditioner:

$$\Phi^{-1} = \begin{bmatrix} A_0^{-1} & -A_0^{-T}B_1^TA_1^{-T} \\ -A_1^{-1}B_1A_0^{-1} & A_1^{-1} & -A_1^{-1}B_2A_2^{-1} \\ -A_2^{-T}B_2^TA_1^{-T} & A_2^{-1} \end{bmatrix}. \tag{6.3}$$

6.2.3 Optimizing for the Trajectory Optimization Problem

At each iteration of the direct trajectory optimization problem we form a KKT system (Equation 2.10) using the dynamics Jacobians $A_k = f_{x_k}$, $B_k = f_{u_k}$, and cost Jacobians and Hessians $Q_k = J_{x_k x_k}$, $R_k = J_{u_k u_k}$, $q_k = J_{x_k}$, $r_k = J_{u_k}$ as follows:

$$G = \begin{bmatrix} Q_0 \\ & R_0 \\ & & \ddots \\ & & & Q_N \end{bmatrix} \qquad C = \begin{bmatrix} I \\ -A_0 & -B_0 & I \\ & & \ddots & -A_{N-1} & -B_{N-1} & I \end{bmatrix}$$

$$g = \begin{bmatrix} q_0 & r_0 & q_1 & r_1 & \dots & q_N \end{bmatrix}^T \qquad e_k = x_{k+1} - f(x_k, u_k)$$

$$c = \begin{bmatrix} x_s - x_0 & e_0 & e_1 & \dots & e_{N-1} \end{bmatrix}^T. \tag{6.4}$$

We can then define the following variables:

$$\theta_k = -A_0 Q_k^{-1} A_k^T - B_k R_k^{-1} B_k^T - Q_{k+1}^{-1}$$

$$\phi_k = A_k Q_k^{-1} \tag{6.5}$$

$$\zeta_k = A_k Q_k^{-1} q_k + B_k R_k^{-1} r_k - Q_{k+1}^{-1} q_{k+1},$$

and use them to define our block-tridiagonal Schur complement system as follows:

$$S = \begin{bmatrix} -Q_0^{-1} & \phi_0^T & & & \\ \phi_0 & \theta_0 & \phi_1^T & & \\ & \ddots & \phi_{N-2} & \theta_{N-2} & \phi_{N-1}^T \\ & & & \phi_{N-1} & \theta_{N-1} \end{bmatrix} \tag{6.6}$$

$$\gamma = c + \begin{bmatrix} -Q_0^{-1} q_0 & \zeta_0 & \zeta_1 & \cdots & \zeta_{N-1} \end{bmatrix}^T.$$

From Equation 6.3 this defines our custom parallel symmetric stair preconditioner as:

$$\Phi^{-1} = \begin{bmatrix} -Q_0 & -Q_0^T \phi_0^T \theta_0^{-T} & & \\ -\theta_0^{-1} \phi_0 Q_0 & \theta_0^{-1} & -\theta_0^{-1} \phi_1^T \theta_1^{-1} & \\ & -\theta_1^{-T} \phi_1 \theta_0^{-T} & \theta_1^{-1} & \\ & & & \ddots \end{bmatrix}. \tag{6.7}$$

As Q is usually symmetric, θ is also symmetric by construction, thus $Q = Q^T$, $\theta = \theta^T$, and $\theta^{-1} = \theta^{-T}$, and each block row k of Φ^{-1} follows the pattern:

$$\Phi_k^{-1} = \begin{bmatrix} -\theta_k^{-1} \phi_k \theta_{k-1}^{-1} & \theta_k^{-1} & -\theta_k^{-1} \phi_{k+1}^T \theta_{k+1}^{-1} \end{bmatrix}. \tag{6.8}$$

Therefore, as each row of S and Φ^{-1} only references cost and integrator information for two or three knot points along the trajectory, the computation of the preconditioner can be folded in with the formation of the quadratic approximation of the problem in one unified set of parallel operations as shown in Algorithm 6.

71

Algorithm 6: *Setup for PCG - Stair ($Z, \lambda \to S, \Phi^{-1}$)*

1: **for** block $b = 0 : N$ **in parallel do**

2:     $\tilde{x}_{b+1}, A_b, B_b = f, \nabla f(x_b, u_b)$

3:     $Q_b, R_b, q_b, r_b, Q_{b+1}, q_{b+1} = \nabla l(x_b, u_b, x_{b+1})$

4:     Compute $\theta_b, \phi_b, \gamma_b$ per Equation 6.5 and 6.6

5:     Compute θ_b^{-1}

6:     Load $\theta_{b-1}^{-1}, \theta_{b+1}^{-1}$, and ϕ_{b+1}^T to finish S_b per Equation 6.6

7:     Compute the off-diagonal terms to finish Φ_b^{-1} per Equation 6.8

6.2.4 The Overall Algorithm

The final overall algorithm then combines Algorithms 5, and 6 with a parallel line search as shown in Algorithm 7. We leverage a parallel line search, as we showed that it can improve convergence in Chapter 4. We do so by computing all possible iterates for $\alpha \in \mathbb{A}$ in parallel and selecting the iterate with the best merit function value, solving the following minimization problem for the optimal line search iterate, α^*:

$$
\begin{aligned}
\arg\min_{\alpha \in \mathbb{A}} \quad & M(Z + \alpha \delta Z; \mu) \\
\text{s.t.} \quad & \text{Equation 2.12}
\end{aligned}
\tag{6.9}
$$

Algorithm 7: *GPU-Accelerated Direct Trajectory Optimization ($Z, \lambda, \epsilon, \mathbb{A} \to Z^*$)*

1: **for** iter $i = 1 : \max\_\text{iter}$ **do**

2:     Setup for PCG with Algorithm 6

3:     Solve for λ^* with Algorithm 5

4:     Compute δZ^* per Equation 2.14

5:     **if** $|\delta Z^*| < \epsilon$ **then**

6:         return Z

7:     Find α^* per Equation 6.9

8:     $Z = Z + \alpha^* \delta Z^*$

6.3 Preliminary Experiments

In this section we describe a series of preliminary experiments that validate our algorithmic approach. We find that not only does this algorithmic approach results in improvements in condition number and spectral radius, leading to up to a 3.1x reduction in iterations-to-converge on standard trajectory optimization problems, but we also find that our solver can be used for whole-body, nonlinear MPC.

Both our baselines and proof-of-concept implementation of our symmetric stair preconditioner and Algorithm 7 were implemented in Python. All experiments were run on a high-performance workstation with a 3.8GHz eight-core Intel Core i7-10700K CPU running Ubuntu 20.04.

6.3.1 Proof-Of-Concept Parallel Preconditioner Evaluation

In this section we compare the efficacy of our symmetric stair preconditioner to baseline approaches to parallel preconditioning across four dynamical systems: a double integrator, a pendulum, a cart pole, and a Kuka LBR IIWA-14 manipulator [125].

As noted in Section 2.4.4, there are a variety of parallel preconditioning methods that have been used in the literature, and deployed onto the GPU, with the most popular being the jacobi and block-jacobi methods [54; 55]. For block banded matricies, like our block tridiagonal Schur complement matrix, alternating and overlapping block preconditioners have also been used in previous work [56; 57]. We use all four of these preconditioners as baselines.

For all four dynamics systems we use a quadratic cost function of the form:

$$J = \frac{1}{2}(x_N - x_g)^T Q_N(x_N - x_g) + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T Q(x_k - x_g) + \frac{1}{2}u_k^T R u_k. \qquad (6.10)$$

For the double integrator we set $Q = 0.01 \times I_{2x2}$, $R = 0.0001$, $Q_N = 10 \times I_{2x2}$, and solve a 1 second trajectory with $N = 10$ knot points from $x_s = [0,0]$ to $x_g = [1,0]$. For the pendulum we solve the swing-up task from $x_s = [0,0]$ to $x_g = [\pi,0]$, set $Q = I_{2x2}$, $R = 0.1$, $Q_N = 100 \times I_{2x2}$, and solve a 2 second trajectory with $N = 20$ knot points. For the cart pole we solve the

73

swing-up task from $x_s = [0, 0, 0, 0]$ to $x_g = [0, \pi, 0, 0]$, set $Q = \texttt{blkdiag}(I_{2x2}, 0.1 \times I_{2x2})$, $R = 0.001$, $Q_N = 1000 \times I_{4x4}$, and solve a 0.5 second trajectory with $N = 40$ knot points. For the manipulator we solve a motion task across the workspace from Section 4.3.2, set $Q = \texttt{blkdiag}(0.1 \times I_{7x7}, 0.01 \times I_{7x7})$, $R = 0.001 \times I_{7x7}$, $Q_N = 1000 \times I_{14x14}$, and solve a 0.5 second trajectory with $N = 20$ knot points.

Figure 6.1 shows the distribution of the magnitude of the Eigenvalues of the Schur complement matrix for the initial conjugate gradient solve for each of the four dynamical systems on a log scale. We report the values for both the original system, highlighted in orange, as well as after apply the various preconditioners from the literature, highlighted in blue, and our symmetric stair preconditioner, highlighted in green. The boxes denote the first, second, and third quartiles of magnitudes and the whiskers denote either the maximum and minimum values or 1.5 times the interquartile range (whichever is closer to the first and third quartile). Outlier points are denoted with circles and only appear above the top whisker for no preconditioner for the manipulator as well as for the alternating block preconditioner for the cart pole and manipulator.

Figure 6.2 additionally shows the condition number, the ratio of the absolute maximum to the absolute minimum Eigenvalue, resulting from each of those sets of Eigenvalues on a log scale, as well as the improvement achieved by our symmetric stair preconditioner over both the original system, highlighted in yellow, and the best alternative from the literature, highlighted in purple. As mentioned previously, closer grouped Eigenvalues, resulting in a lower condition number, will both cause the algorithm to converge faster and improve the numerical stability of the resulting conjugate gradient algorithm.

We see across all four systems that all of the preconditioners greatly reduce both the spectral radius and the condition number of the Schur complement matrix. This reinforces the importance of using preconditioning with iterative methods.

We also find that our symmetric stair preconditioner always produces the most well conditioned

Eigen Values of the Preconditioned Schur Complement Matrix
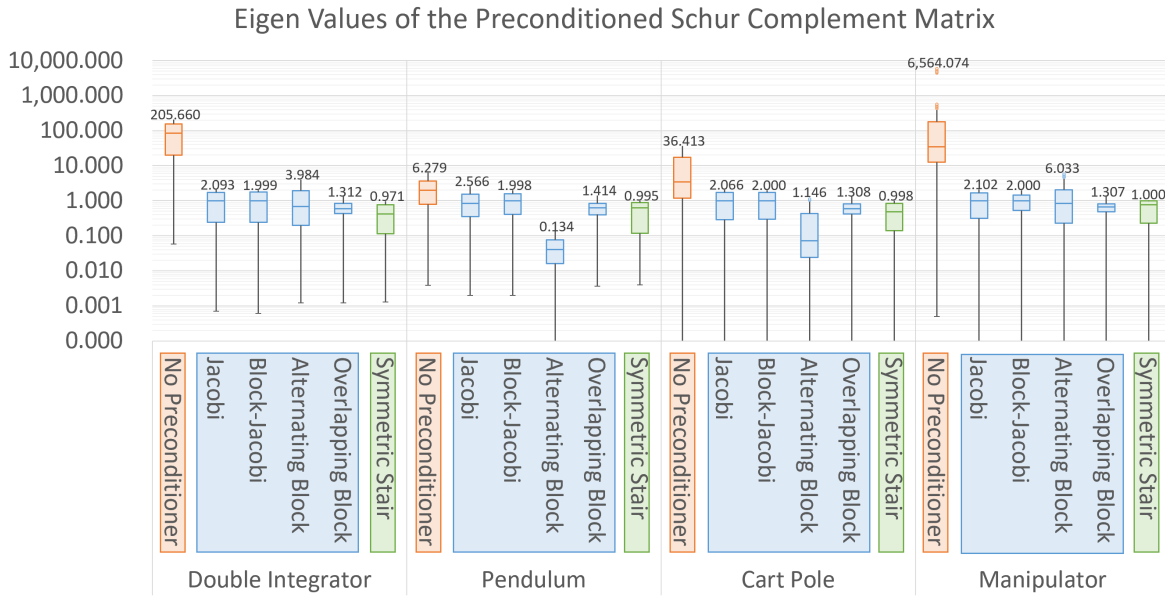
Figure 6.1: *Log scale box plots showing the range of the magnitude of the Eigenvalues in the Schur complement matrix for each of the four dynamical systems both for the original system (in orange), as well as after apply the various preconditioners from the literature (in blue), and our symmetric stair preconditioner (in green). We find that, for all four systems, all of the preconditioners reduce the overall magnitude of the Eigenvalues, and group them closer together, and that the symmetric stair preconditioner is the only preconditioner which keeps the spectral radius ≤ 1.*

Condition Number of the Preconditioned Schur Complement Matrix
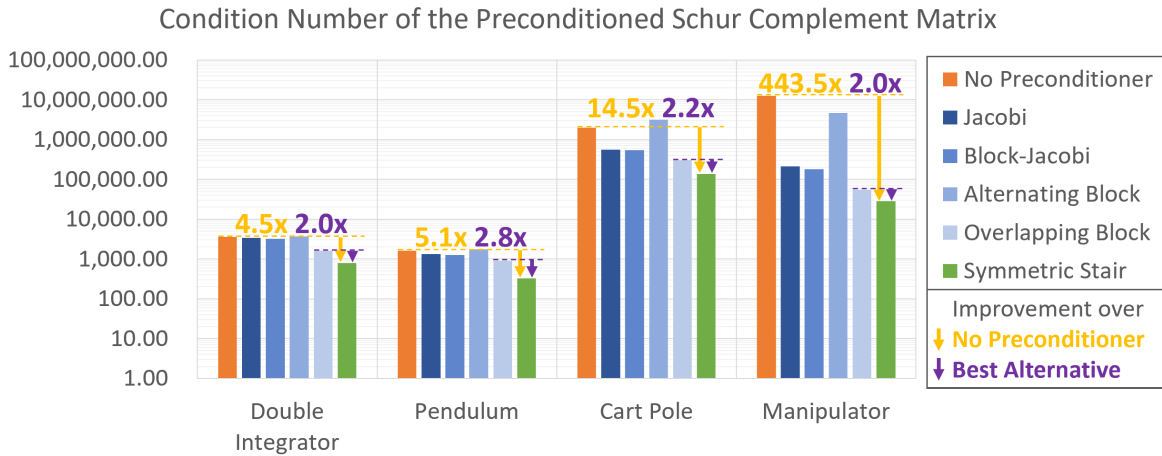
Figure 6.2: *A log scale bar chart showing the condition number of the Schur complement matrix for each of the four dynamical systems both for the original system (in orange), as well as after apply the various preconditioners from the literature (in blue), and our symmetric stair preconditioner (in green). We find that, for all four systems, all of the preconditioners improve the numerical conditioning and that the symmetric stair preconditioner results in the lowest condition number outperforming the best alternatives by more than 2x.*

75

system, 4.5x to 443.5x better than the original system, and 2.0x to 2.8x better than the best preconditioner from the literature, the overlapping block preconditioner. We also find that our preconditioner is the one that is able to keep the spectral radius to less than or equal to 1 for all four systems. Taken together these results show that our preconditioner not only improves the numerical stability of the resulting PCG solves, but is also the only preconditioner that can guarantee convergence for the resulting PCG solve.
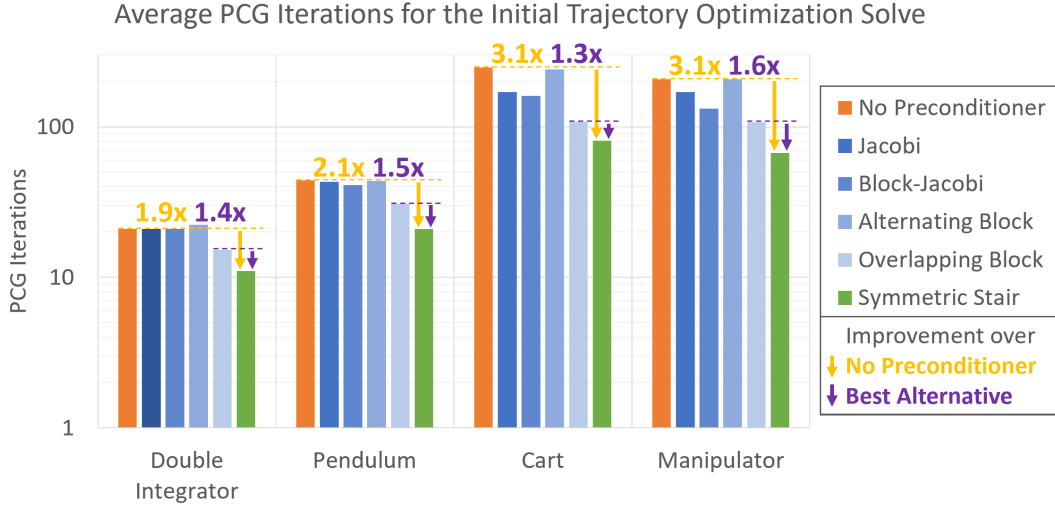
Figure 6.3: *A log scale bar chart showing the average number of inner (preconditioned) conjugate gradient iterations needed to solve the initial trajectory optimization problems for each of the four dynamical systems both for the original problem (in orange), as well as after apply the various preconditioners from the literature (in blue), and our symmetric stair preconditioner (in green). We find that, for all four systems, all of the preconditioners reduce the number of iterations and that the symmetric stair preconditioner outperforms the best alternative by up to 1.6x.*

Figure 6.3 shows the the average number of inner (preconditioned) conjugate gradient iterations needed to solve the initial trajectory optimization problems for each of the four dynamical systems both for the original problem, highlighted in orange, as well as after apply the various preconditioners from the literature, highlighted in blue, and our symmetric stair preconditioner, highlighted in green. We also plot the improvement achieved by our symmetric stair preconditioner over both the original system, highlighted in yellow, and the best alternative from the literature, highlighted in purple. We find that our preconditioner results in 1.9x to 3.1x reduction in the number of iterations over the original problem and maintains a

1.3x to 1.6x reduction over the best alternative from the literature, the overlapping block preconditioner. This shows that not only does our preconditioner improve the numerical conditioning of the systems, but also that this improved conditioning translates into improved performance on trajectory optimization problems used for whole-body, nonlinear MPC. This is crucial as a reduction in the number of inner PCG iterations directly translates to a reduction in the overall computational latency of the outer trajectory optimization algorithm.

6.3.2 Proof-Of-Concept Nonlinear MPC Evaluation

We then evaluate our algorithms suitability for nonlinear MPC by comparing it to two baseline approaches to solving the nonlinear trajectory optimization problem at each MPC control step for both the pendulum and cart pole problems as defined in the previous section. At each control step we warm started each solver by shifting all variables from the previous solve and simulated the system forward using the previously computed solution.

For our first baseline, iLQR, we use the standard iLQR algorithm (PDDP with $M = 1$). As discussed in Chapter 4, there is a growing evidence base that these approaches can be used for realtime nonlinear MPC. For our second baseline, KKT-F, we use a standard factorization approach to directly solve the KKT system produced by a direct trajectory optimization approach (defined in Equation 2.10) through the use of the `numpy.linalg` package. We use this second baseline as it represents the current standard method for direct trajectory optimization on a CPU. Ideally our method would recover the same solution as the KKT-F approach as it is solving the exact same direct trajectory optimization problem and simply using a different numerical method to arrive at the solution (an iterative method instead of a factorization method).

We plot the final pendulum trajectory and control inputs used during the nonlinear MPC experiment for our symmetric stair, preconditioned conjugate gradient, Schur complement based solver, Schur-PCG-SS, against the two baselines in Figure 6.4. We find that not only are all three approaches able to converge to the goal state, but that they also produce very similar

trajectories. As hoped, our approach produces an almost identical solution to the KKT-F baseline, showing their equivalence when used for nonlinear MPC.

In Figure 6.5 we plot the error between the current state and goal state at each control step as well as the control inputs used during the cart pole nonlinear MPC experiment. We again
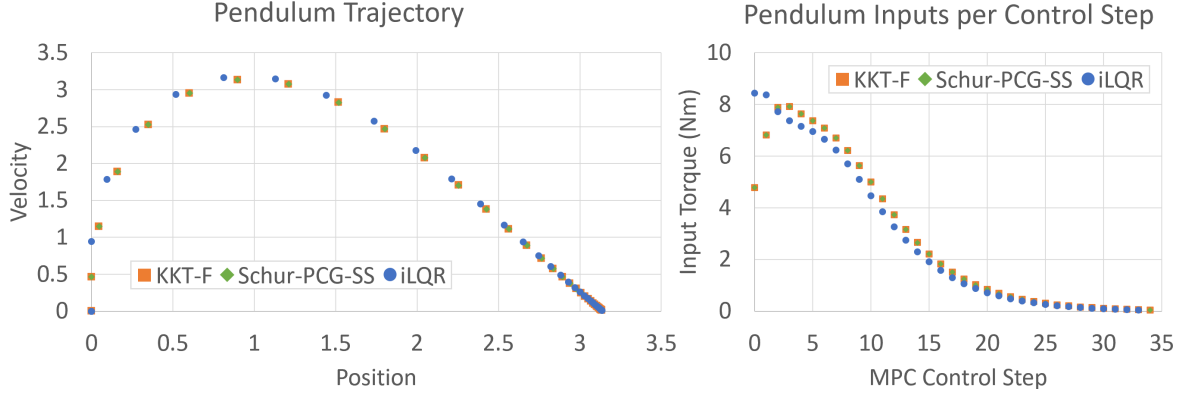
Figure 6.4: *On the left, a scatter plot showing the resulting pendulum trajectory when using all three approaches to solve the trajectory optimization problem at each control step: our symmetric stair, preconditioned conjugate gradient, schur complement based solver (Schur-PCG-SS in green), the baseline standard factorization approach to solve the direct trajectory optimization problem (KKT-F in orange), and the baseline iLQR algorithm (iLQR in blue). On the right, the corresponding input torques per control step. We find that not only are all three approaches are able to converge to the goal position, but that our approach produces an almost identical solution to the KKT-F baseline.*
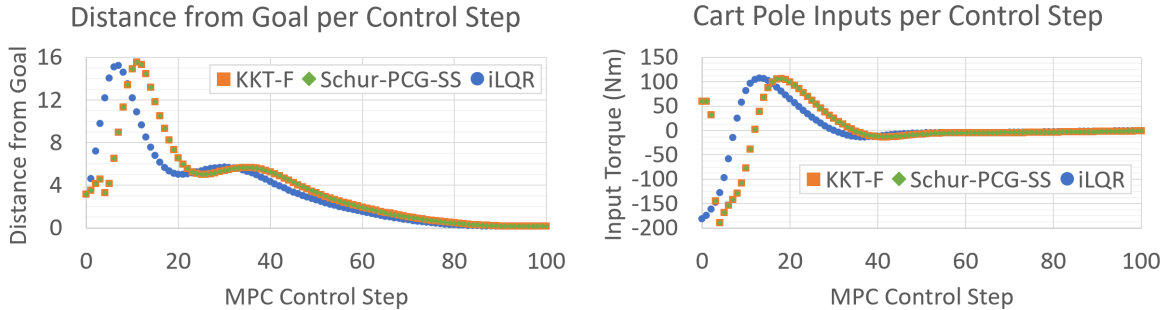
Figure 6.5: *On the left, a scatter plot showing the error between the current state at each control step and goal state of the cart pole experiment using all three approaches to solve the trajectory optimization problem: our symmetric stair, preconditioned conjugate gradient, schur complement based solver (Schur-PCG-SS in green), the baseline standard factorization approach to solve the direct trajectory optimization problem (KKT-F in orange), and the baseline iLQR algorithm (iLQR in blue). On the right, the corresponding input torques per control step. We find that not only are all three approaches able to converge to the goal state position, but that our approach again also produces an almost identical solution to the KKT-F baseline.*

78

plot results for our approach as well as the two baseline approaches. As with the previous experiment we find that not only are all three approaches able to converge to the goal state, but that our approach again produces an almost identical solution to the KKT-F approach.

Taken together, these experiments suggest that our approach to solving the nonlinear trajectory optimization problems can be used for efficient whole-body, nonlinear MPC.

6.4   Conclusion and Future Work

In this chapter we described the design of a GPU accelerated direct trajectory optimization algorithm leveraging a structure exploiting Krylov subspace solver and a parallel symmetric stair preconditioner. We then showed preliminary results indicating that this algorithmic approach results in better conditioned problems than many existing approaches, resulting in improved performance in terms of iterations-to-convergence, and is applicable to nonlinear MPC problems.

In future work we hope to implement this algorithm in CUDA and benchmark it on a GPU for performance against other state-of-the-art CPU and GPU implementations of nonlinear MPC. We hope to leverage the GRiD library (Chapter 5) in our implementation to ensure we are using fully optimized rigid body dynamics kernels, and hope to release our resulting implementation open-source for use by the wider robotics community.

We would also like to explore other approaches to solving the trajectory optimization problem which may also expose significant amounts of natural parallelism (e.g., solvers based on the alternating direction method of multipliers [101]), as well as other parallel-friendly approaches to solving the KKT system (e.g., block-Cholesky factorizations [166]). Finally we would like to compare trust region methods to our line search approach to better understand their relative robustness, globalization abilities, and parallel computational patterns.

Chapter 7

Conclusion and Future Work

In this dissertation we explored the opportunities, benefits, and challenges of leveraging GPU acceleration to develop real-time, whole-body, nonlinear MPC algorithms and implementations. Through hardware-software co-design, we developed algorithms and implementations that leveraged the strengths and minimized the weaknesses of the GPU. We also deployed our implementations on a physical manipulator arm to demonstrate the feasibility of this approach in the presence of model discrepancies and communication delays between the robot and GPU. Overall, we found that GPU acceleration can provide nearly order-of-magnitude speedups over state-of-the-art CPU implementations. To promote reproducibility and further development and use of this work, we released our implementations open-source for use by the broader robotics community.

Several directions for future research remain. Most importantly, we need to finish and release open-source our CUDA implementation of our direct trajectory optimization algorithm and benchmark it on a GPU for performance against other state-of-the-art CPU and GPU implementations of nonlinear MPC. We also hope to integrate GRiD into our direct trajectory optimization implementation, as well as into our PDDP implementation, and any additional trajectory optimization algorithms we implement in the future.

Also, as noted in the previous chapters there are ample opportunities to explore alternate formulations of both trajectory optimization algorithms [101; 166], as well as rigid body dynamics algorithms [99; 127; 128; 129; 130; 131; 132; 133; 134], which may result in high performance parallel algorithms. There are also many additional features that needed to be added to our implementations to make them most useful to the wider robotics community. In particular we hope to add support for the full breadth of rigid body dynamics algorithms and robot models supported by current CPU libraries [100; 102; 108; 109; 110], as well as support for additional nonlinear constraints (e.g., contact constraints) in our trajectory optimization algorithms using augmented Lagrangian or QP-based methods [44; 45; 46; 47; 48].

Finally, we aim to develop a front-end to our algorithms in higher level languages, such as Python and Julia, to reduce the barrier to entry for our GPU accelerated algorithms and implementations. We hope that this work serves as a foundation to enable the broader robotics community to access GPU-accelerated, real-time, whole-body, nonlinear MPC.

7.1 Hardware Acceleration Beyond the GPU

While this dissertation focuses on the fact that GPU acceleration can greatly improve the computational performance of robotics algorithms, it may be possible to achieve further acceleration by leveraging other parallel computer hardware architectures such as Field Programmable Gate Arrays (FPGAs) and custom Application-Specific Integrated Circuits (ASICs). While these types of computer hardware require significantly more work to design algorithms and implementations, and result in designs that are much harder, if not impossible, to modify, they can provide substantial amounts of parallelism without suffering from the rigid computational model found on GPUs.

To explore the possibility of additional acceleration from FPGAs and ASICs, we performed an initial experiment comparing the single computation latency of our proof-of-concept GPU and hand-optimized CPU baseline implementations for the Kuka manipulator from Section 5.5.1 to a proof-of-concept FPGA and synthesized ASIC implementation (see Appendix C for details

on their designs). For this experiment, we used a 3.6GHz quad-core Intel Core i7-7700 CPU running Ubuntu 18.04 and CUDA 11, a 1.7GHz NVIDIA GeForce GTX 2080 GPU, and a Virtex UltraScale+ VCU-118 FPGA synthesized at a clock speed of 55.6MHz. We synthesized our ASIC using a Global Foundries 12nm technology node at the typical process corner.

The results of this experiment are shown in Figure 7.1. As noted in Section 5.5.1, the GPU's computational model is not well optimized for this single computation, and although it outperforms the CPU in the multiple computation experiments, it is 2.5x slower than the CPU in this experiment. The more customized computational models of the FPGA and ASIC do not suffer from these issues and the FPGA outperforms the CPU by 5.6x, while the ASIC outperforms the FPGA by an additional 7.2x. This means that a custom ASIC outperforms the GPU by 99.4x. Notably, synthesis shows that our custom ASIC would be 65x smaller than a standard CPU, implying that we could support many parallel computational units for multiple computation use cases. Overall, this final experiment suggests that these alternative parallel hardware architectures should be explored more in future work.
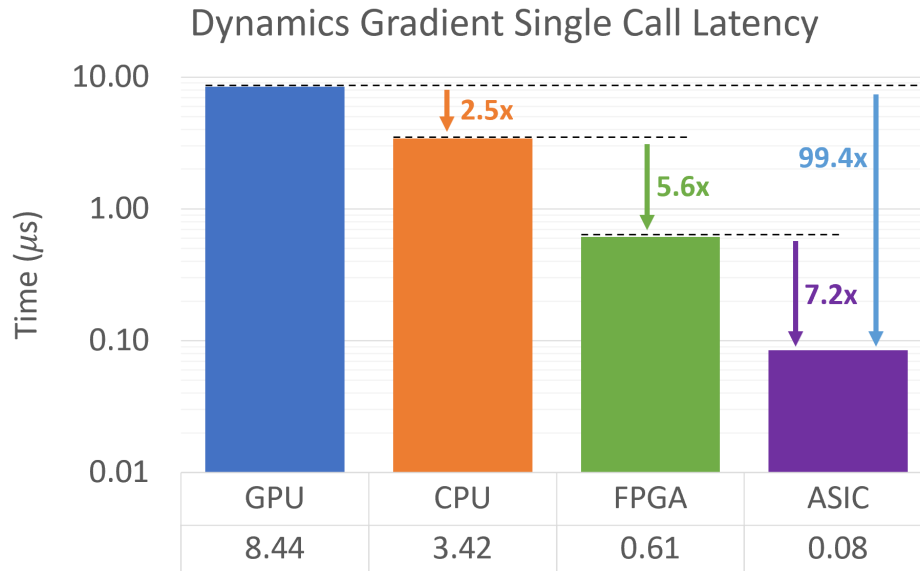
Figure 7.1: *Latency of one computation of the gradient of rigid body dynamics for the Kuka manipulator for our proof-of-concept optimized GPU implementation, optimized CPU baseline, and a proof-of-concept FPGA and ASIC implementation. We find that FGPAs and ASICs can significantly accelerate computations beyond the speeds available on CPUs and GPUs.*

References

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

[2] 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)*, 2016, pp. 1398–1404.

[3] B. Plancher and S. Kuindersma, "A Performance Analysis of Parallel Differential Dynamic Programming on a GPU," in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.

[4] 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 IEEE/RAS Conference on Intelligent Robots*, 2015.

[5] 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)*. ACM, pp. 365–376.

[6] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, pp. 205–218.

[7] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[8] S. Kuindersma, "Recent progress on atlas, the world's most dynamic humanoid robot," 2020, Robotics Today - A series of technical talks. [Online]. Available: https://www.youtube.com/watch?v=EGABAx52GKI

[9] L. Grossman, "Reinforcement learning to enable robust robotic model predictive control," Cambridge, MA, USA, May. 2020.

[10] J. Carius, F. Farshidian, and M. Hutter, "Mpc-net: A first principles guided policy search," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, p. 2897–2904, Apr 2020.

[11] M. Omer, R. Ahmed, B. Rosman, and S. F. Babikir, "Model predictive-actor critic reinforcement learning for dexterous manipulation," in *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, 2021, pp. 1–6.

[12] D. Hoeller, F. Farshidian, and M. Hutter, "Deep value model predictive control," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100, 30 Oct–01 Nov 2020, pp. 990–1004.

[13] S. Gros and M. Zanon, "Reinforcement learning for mixed-integer problems based on mpc," 2020.

[14] A. S. Morgan, D. Nandha, G. Chalvatzaki, C. D'Eramo, A. M. Dollar, and J. Peters, "Model predictive actor-critic: Accelerating robot skill acquisition with deep reinforcement learning," 2021.

[15] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, 1997.

[16] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[17] E. F. Camacho and C. B. Alba, *Model predictive control*. Springer science & business media, 2013.

[18] S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi, and H. Hirukawa, "The 3d linear inverted pendulum mode: A simple modeling for a biped walking pattern generation," in *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, vol. 1. IEEE, 2001, pp. 239–246.

[19] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa, "Biped walking pattern generation by using preview control of zero-moment point," in *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, vol. 2. IEEE, 2003, pp. 1620–1626.

[20] E. R. Westervelt, J. W. Grizzle, and D. E. Koditschek, "Hybrid zero dynamics of planar biped walkers," *IEEE transactions on automatic control*, vol. 48, no. 1, pp. 42–56, 2003.

[21] K. Sreenath, H.-W. Park, I. Poulakakis, and J. W. Grizzle, "A compliant hybrid zero dynamics controller for stable, efficient and fast bipedal walking on mabel," *The International Journal of Robotics Research*, vol. 30, no. 9, pp. 1170–1193, 2011.

[22] D. E. Orin, A. Goswami, and S.-H. Lee, "Centroidal dynamics of a humanoid robot," *Autonomous robots*, vol. 35, no. 2, pp. 161–176, 2013.

[23] H. Dai, A. Valenzuela, and R. Tedrake, "Whole-body Motion Planning with Centroidal Dynamics and Full Kinematics," in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2014.

[24] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for Atlas," *Auton. Robots*, vol. 40, no. 3, pp. 429–455, 2016.

[25] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1–9.

[26] E. R. Westervelt, J. W. Grizzle, C. Chevallereau, J. H. Choi, and B. Morris, *Feedback control of dynamic bipedal robot locomotion*. CRC press, 2018.

[27] J. T. Betts, *Practical Methods for Optimal Control Using Nonlinear Programming*, ser. Advances in Design and Control. Society for Industrial and Applied Mathematics (SIAM), 2001, vol. 3.

[28] J. R. Dormand and P. J. Prince, "A family of embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980.

[29] T. Fan, J. Schultz, and T. Murphey, "Efficient computation of higher-order variational integrators in robotic simulation and trajectory optimization," in *International Workshop on the Algorithmic Foundations of Robotics*. Springer, 2018, pp. 689–706.

[30] Z. Manchester, N. Doshi, R. J. Wood, and S. Kuindersma, "Contact-implicit trajectory optimization using variational integrators," *The International Journal of Robotics Research*, vol. 38, no. 12-13, pp. 1463–1476, 2019.

[31] W. Jallet, N. Mansard, and J. Carpentier, "Implicit differential dynamic programming," 2021.

[32] I. Chatzinikolaidis and Z. Li, "Trajectory optimization of contact-rich motions using implicit differential dynamic programming," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2626–2633, 2021.

[33] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. Springer, 2006.

[34] D. Q. Mayne, "A second-order gradient method of optimizing non- linear discrete time systems," *Int J Control*, vol. 3, p. 8595, 1966.

[35] D. H. Jacobson and D. Q. Mayne, "Differential dynamic programming," 1970.

[36] R. Bellman, *Dynamic Programming*. Dover, 1957.

[37] 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*, 2012.

[38] 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*, 2004.

[39] L. Liao and C. A. Shoemaker, "Advantages of Differential Dynamic Programming Over Newton's Method for Discrete-time Optimal Control Problems," 1992.

[40] Y. Tassa, "Theory and Implementation of Biomimetic Motor Controllers," 2011.

[41] "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,*, 1989.

[42] Y. Tassa, T. Erez, and E. Todorov, "Control-Limited Differential Dynamic Programming," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2014.

[43] F. Farshidian, M. Neunert, A. W. Winkler, G. Rey, and J. Buchli, "An Efficient Optimal Planning and Control Framework For Quadrupedal Locomotion," 2016.

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

[45] Y. Aoyama, G. Boutselis, A. Patel, and E. A. Theodorou, "Constrained differential dynamic programming revisited," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 9738–9744.

[46] W. Jallet, N. Mansard, and J. Carpentier, "Implicit differential dynamic programming," May 2022.

[47] 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)*, 2017.

[48] T. Howell, B. Jackson, and Z. Manchester, "Altro: A fast solver for constrained trajectory optimization," in *Proceedings of (IROS) IEEE/RSJ International Conference on Intelligent Robots and Systems*, November 2019, pp. 7674 – 7679.

[49] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP Algorithm for Large-scale Constrained Optimization," *SIAM Rev.*, vol. 47, no. 1, pp. 99–131, 2005.

[50] A. Wächter and L. T. Biegler, "On the Implementation of an Interior-point Filter Line-search Algorithm for Large-scale Nonlinear Programming," *Math Program*, vol. 106, no. 1, pp. 25–57, 2006.

[51] M. Toussaint, "A Novel Augmented Lagrangian Approach for Inequalities and Convergent Any-Time Non-Central Updates," 2014.

[52] P. E. Gill, W. Murray, and M. H. Wright, *Numerical linear algebra and optimization*. SIAM, 2021.

[53] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," USA, Tech. Rep., 1994.

[54] G. Flegar *et al.*, "Sparse linear system solvers on gpus: Parallel preconditioning, workload balancing, and communication reduction," Ph.D. dissertation, Universitat Jaume I, 2019.

[55] M. Schubiger, G. Banjac, and J. Lygeros, "Gpu acceleration of admm for large-scale quadratic programming," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 55–67, 2020.

[56] E. Galligani and V. Ruggiero, "A polynomial preconditioner for block tridiagonal matrices," *PARALLEL ALGORITHM AND APPLICATIONS*, vol. 3, no. 3-4, pp. 227–237, 1994.

[57] Y. Saad, *Iterative methods for sparse linear systems.* SIAM, 2003.

[58] S. Demko, W. F. Moss, and P. W. Smith, "Decay rates for inverses of band matrices," *Mathematics of computation*, vol. 43, no. 168, pp. 491–499, 1984.

[59] A. Williams, *C++ Concurrency in Action : Practical Multithreading.* Manning, 2012.

[60] NVIDIA, *NVIDIA CUDA C Programming Guide*, version 11.4 ed., 2022. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[61] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing.* IEEE, 2010, pp. 344–350.

[62] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[63] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, "Opencl and the 13 dwarfs: a work in progress," in *Proceedings of the 3rd acm/spec international conference on performance engineering*, 2012, pp. 291–294.

[64] 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*, 2017.

[65] 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*, 2013.

[66] M. Neunert, F. Farshidian, A. W. Winkler, and J. Buchli, "Trajectory Optimization Through Contacts and Automatic Gait Discovery for Quadrupeds," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1502–1509, 2017.

[67] M. Neunert, M. Stäuble, M. Giftthaler, C. D. Bellicoso, J. Carius, C. Gehring, M. Hutter, and J. Buchli, "Whole-body nonlinear model predictive control through contacts for quadrupeds," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1458–1465, 2018.

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

[69] E. Dantec, R. Budhiraja, A. Roig, T. Lembono, G. Saurel, O. Stasse, P. Fernbach, S. Tonneau, S. Vijayakumar, S. Calinon, *et al.*, "Whole body model predictive control with a memory of motion: Experiments on a torque-controlled talos," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 8202–8208.

[70] J.-P. Sleiman, F. Farshidian, M. V. Minniti, and M. Hutter, "A unified mpc framework for whole-body dynamic locomotion and manipulation," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4688–4695, 2021.

[71] S. Kleff, A. Meduri, R. Budhiraja, N. Mansard, and L. Righetti, "High-frequency nonlinear model predictive control of a manipulator," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 7330–7336.

[72] D. Leineweber, "Efficient reduced SQP methods for the optimization of chemical processes described by large sparse DAE models," 1999.

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

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

[75] 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*, ser. GPGPU-10. ACM, 2017, pp. 12–21.

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

[77] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *2009 European Control Conference (ECC)*, 2009, pp. 156–161.

[78] H. J. Ferreau, A. Kozma, and M. Diehl, "A Parallel Active-Set Strategy to Solve Sparse Parametric Quadratic Programs arising in MPC," *IFAC Proceedings Volumes*, vol. 45, no. 17, pp. 74–79, 2012.

[79] J. V. Frasch, S. Sager, and M. Diehl, "A parallel quadratic programming method for dynamic optimization problems," *Math. Prog. Comp.*, vol. 7, no. 3, pp. 289–329, 2015.

[80] G. Frison, "Algorithms and Methods for Fast Model Predictive Control," 2015.

[81] R. Quirynen, "Numerical simulation methods for embedded optimization," 2017.

[82] Y. Huang, K. V. Ling, and S. See, "Solving Quadratic Programming Problems on Graphics Processing Unit," *ASEAN Eng. J.*, 2011.

[83] D.-K. Phung, B. Hérissé, J. Marzat, and S. Bertrand, "Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 11 883–11 888, 2017.

[84] T. Antony and M. J. Grant, "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures," *J. Spacecr. Rockets*, vol. 54, no. 5, pp. 1081–1091, 2017.

[85] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. J. Reddi, "Accelerating robot dynamics gradients on a cpu, gpu, and fpga," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2335–2342, 2021.

[86] S. M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, and V. J. Reddi, "Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology," in *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021, p. 674–686.

[87] B. Plancher, S. M. Neuman, R. Ghosal, S. Kuindersma, and V. J. Reddi, "Grid: Gpu-accelerated rigid body dynamics with analytical gradients," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2022.

[88] A. Astudillo, J. Gillis, G. Pipeleers, W. Decré, and J. Swevers, "Speed-up of nonlinear model predictive control for robot manipulators using task and data parallelism," in *2022 IEEE 17th International Conference on Advanced Motion Control (AMC)*, 2022, pp. 201–206.

[89] H. G. Bock and K.-J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," *IFAC Proc. Vol.*, vol. 17, no. 2, pp. 1603–1608, 1984.

[90] J. T. Betts and W. P. Huffman, "Trajectory optimization on a parallel processor," *J. Guid. Control Dyn.*, vol. 14, no. 2, pp. 431–439, 1991.

[91] D. M. Garza, "Application of automatic differentiation to trajectory optimization via direct multiple shooting," 2003.

[92] 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*. Springer, Berlin, Heidelberg, 2006, pp. 65–93.

[93] 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*.

[94] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl, "A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control," 2017.

[95] E. Pellegrini and R. P. Russell, "A Multiple-Shooting Differential Dynamic Programming Algorithm," in *AAS/AIAA Space Flight Mechanics Meeting*, 2017.

[96] B. Plancher, "Parallel and constrained differential dynamic programming for model predictive control," Master's thesis, Harvard University, Cambridge, MA, USA, May. 2019.

[97] M. Zinkevich, J. Langford, and A. J. Smola, "Slow Learners are Fast," in *Advances in Neural Information Processing Systems 22*. Curran Associates, Inc., pp. 2331–2339.

[98] 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," *Advances in Neural Information Processing Systems (NIPS)*, vol. 2013.

[99] Y. Yang, Y. Wu, and J. Pan, "Parallel Dynamics Computation Using Prefix Sum Operations," *IEEE Robot. Autom. Lett.*, vol. 2, no. 3, pp. 1296–1303, 2017.

[100] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "The pinocchio c++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, 2019, pp. 614–619.

[101] S. Boyd, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundational Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2010.

[102] 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," *J. Softw. Eng. Robot. JOSER*, vol. 7, no. 1, pp. 36–54, 2016.

[103] S. Neuman, T. Koolen, J. Drean, J. Miller, and S. Devadas, "Benchmarking and workload analysis of robot dynamics algorithms," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[104] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

[105] J.-P. Sleiman, F. Farshidian, M. V. Minniti, and M. Hutter, "A unified mpc framework for whole-body dynamic locomotion and manipulation," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4688–4695, 2021.

[106] L. Drnach and Y. Zhao, "Robust trajectory optimization over uncertain terrain with stochastic complementarity," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1168–1175, 2021.

[107] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, "Brax – a differentiable physics engine for large scale rigid body simulation," 2021.

[108] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Proceedings of the IEEE-RAS International Conference on Intelligent Robots*, 2012.

[109] T. Koolen and R. Deits, "Julia for robotics: simulation and real-time control in a high-level programming language," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 604–611.

[110] K. Werling, D. Omens, J. Lee, I. Exarchos, and C. K. Liu, "Fast and feature-complete differentiable physics for articulated rigid bodies with contact," *arXiv preprint arXiv:2103.16021*, 2021.

[111] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State, "Isaac gym: High performance gpu-based physics simulation for robot learning," 2021.

[112] G. Williams, A. Aldrich, and E. A. Theodorou, "Model predictive path integral control: From theory to parallel computation," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 2, pp. 344–357, 2017.

[113] M. Neunert, M. Giftthaler, M. Frigerio, C. Semini, and J. Buchli, "Fast Derivatives of Rigid Body Dynamics for Control, Optimization and Estimation," in *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2016.

[114] J. Bender, M. Müller, and M. Macklin, "A survey on position based dynamics, 2017," in *Proceedings of the European Association for Computer Graphics: Tutorials*, 2017, pp. 1–31.

[115] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, "End-to-end differentiable physics for learning and control," in *Advances in neural information processing systems*, 2018, pp. 7178–7189.

[116] J. Degrave, M. Hermans, J. Dambre, *et al.*, "A differentiable physics engine for deep learning in robotics," *Frontiers in neurorobotics*, vol. 13, p. 6, 2019.

[117] Y. Hu, J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik, "Chainqueen: A real-time differentiable physical simulator for soft robotics," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 6265–6271.

[118] J. Austin, R. Corrales-Fatou, S. Wyetzner, and H. Lipson, "Titan: A parallel asynchronous library for multi-agent and soft-body robotics using nvidia cuda," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 7754–7760.

[119] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, "Difftaichi: Differentiable programming for physical simulation," in *International Conference on Learning Representations (ICLR)*, 2020.

[120] E. Heiden, M. Macklin, Y. S. Narang, D. Fox, A. Garg, and F. Ramos, "DiSECt: A Differentiable Simulation Engine for Autonomous Robotic Cutting," in *Proceedings of Robotics: Science and Systems*, Virtual, July 2021.

[121] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, 2009, pp. 79–84.

[122] D. Michéa and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010.

[123] R. Featherstone, "Exploiting sparsity in operational-space dynamics," *The International Journal of Robotics Research*, vol. 29, no. 10, pp. 1353–1368, 2010.

[124] C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, "Design of hyq - a hydraulically and electrically actuated quadruped robot," *IMechE Part I: Journal of Systems and Control Engineering*, vol. 225, no. 6, pp. 831–849, 2011.

[125] KUKA AG, "Lbr iiwa | kuka ag," Accessed in 2020. [Online]. Available: https://www.kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa

[126] Boston Dynamics, "Atlas," Accessed in 2021. [Online]. Available: https://www.bostondynamics.com/atlas

[127] R. Featherstone, "A divide-and-conquer articulated-body algorithm for parallel o (log (n)) calculation of rigid-body dynamics. part 1: Basic algorithm," *The International Journal of Robotics Research*, vol. 18, no. 9, pp. 867–875, 1999.

[128] K. Yamane and Y. Nakamura, "Comparative Study on Serial and Parallel Forward Dynamics Algorithms for Kinematic Chains," *The International Journal of Robotics Research*, vol. 28, no. 5, pp. 622–629, 2009.

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

[130] J. Nganga and P. M. Wensing, "Accelerating second-order differential dynamic programming for rigid-body systems," *IEEE Robotics and Automation Letters*, 2021.

[131] S. Singh, R. P. Russell, and P. M. Wensing, "Efficient analytical derivatives of rigid-body dynamics using spatial vector algebra," 2021.

[132] S. Echeandia and P. M. Wensing, "Numerical methods to compute the coriolis matrix and christoffel symbols for rigid-body systems," *Journal of Computational and Nonlinear Dynamics*, vol. 16, no. 9, 07 2021.

[133] W. Agboh, D. Ruprecht, and M. Dogar, "Combining coarse and fine physics for manipulation using parallel-in-time integration," *ISRR 2019 Springer Tracts in Advanced Robotics*, July 2019.

[134] W. Agboh, O. Grainger, D. Ruprecht, and M. Dogar, "Parareal with a learned coarse model for robotic manipulation," *Computing and Visualization in Science*, vol. 23, no. 1, pp. 1–10, 2020.

[135] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, "NeuralSim: Augmenting differentiable simulators with neural networks," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[136] K. Werling, D. Omens, J. Lee, I. Exarchos, and C. K. Liu, "Fast and feature-complete differentiable physics engine for articulated rigid bodies with contact constraints," in *Robotics: Science and Systems*, 2021.

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

[138] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[139] M. Giftthaler, M. Neunert, M. Stäuble, and J. Buchli, "The control toolbox — an open-source c++ library for robotics, optimal and model predictive control," *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pp. 123–129, 2018.

[140] J. H. Jung and D. P. O'Leary, "Cholesky decomposition and linear programming on a gpu," *Scholarly Paper, University of Maryland*, 2006.

[141] D. Yang, G. D. Peterson, and H. Li, "Compressed sensing and cholesky decomposition on fpgas and gpus," *Parallel Computing*, vol. 38, no. 8, pp. 421–437, 2012.

[142] I. E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. H. Sameh, "A direct tridiagonal solver based on givens rotations for gpu architectures," *Parallel Computing*, vol. 49, pp. 101–116, 2015.

[143] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, "A sparse symmetric indefinite direct solver for gpu architectures," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 1, pp. 1–25, 2016.

[144] X. Hu, C. C. Douglas, R. Lumley, and M. Seo, "Gpu accelerated sequential quadratic programming," in *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*. IEEE, 2017, pp. 3–6.

[145] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse qr factorization on the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, pp. 1–29, 2017.

[146] H. Liu, J.-H. Seo, R. Mittal, and H. H. Huang, "Gpu-accelerated scalable solver for banded linear systems," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[147] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, "Preconditioned krylov solvers on gpus," *Parallel Computing*, 05 2017.

[148] H. Anzt, M. Kreutzer, E. Ponce, G. D. Peterson, G. Wellein, and J. Dongarra, "Optimization and performance evaluation of the idr iterative krylov solver on gpus," *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, pp. 220–230, 2018.

[149] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using gpus," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 2012, pp. 1–11.

[150] L.-W. Chang and W.-m. W. Hwu, *A Guide for Implementing Tridiagonal Solvers on GPUs.* Springer International Publishing, 2014, pp. 29–44.

[151] A. P. Diéguez, M. Amor, and R. Doallo, "New tridiagonal systems solvers on gpu architectures," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC).* IEEE, 2015, pp. 85–94.

[152] A. Lamas Daviña and J. Roman, "Mpi-cuda parallel linear solvers for block-tridiagonal matrices in the context of slepc's eigensolvers," *Parallel computing*, vol. 74, pp. 118–135, 2018.

[153] 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)*, 2015, pp. 3572–3577.

[154] Q. Wu, F. Xiong, F. Wang, and Y. Xiong, "Parallel particle swarm optimization on a graphics processing unit with application to trajectory optimization," *Engineering Optimization*, vol. 48, no. 10, pp. 1679–1692, 2016.

[155] P. Hyatt and M. D. Killpack, "Real-time evolutionary model predictive control using a graphics processing unit," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids).* IEEE, 2017, pp. 569–576.

[156] S. Ohyama and H. Date, "Parallelized nonlinear model predictive control on gpu," in *2017 11th Asian Control Conference (ASCC).* IEEE, 2017, pp. 1620–1625.

[157] K. M. M. Rathai, O. Sename, and M. Alamir, "Gpu-based parameterized nmpc scheme for control of half car vehicle with semi-active suspension system," *IEEE Control Systems Letters*, vol. 3, no. 3, pp. 631–636, 2019.

[158] Y. Wang, X. Luo, F. Zhang, and S. Wang, "Gpu-based model predictive control for continuous casting spray cooling control system using particle swarm optimization," *Control Engineering Practice*, vol. 84, pp. 349–364, 2019.

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

[160] J. V. Frasch, M. Vukov, H. J. Ferreau, and M. Diehl, "A dual newton strategy for the efficient solution of sparse quadratic programs arising in sqp-based nonlinear mpc," *Optimization Online 3972*, 2013.

[161] Y. Gang and L. Mingguang, "Acceleration of mpc using graphic processing unit," in *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*. IEEE, 2012, pp. 1001–1004.

[162] N. F. Gade-Nielsen, J. B. Jørgensen, and B. Dammann, "Mpc toolbox with gpu accelerated optimization algorithms," in *10th European workshop on advanced control and diagnosis*. Technical University of Denmark, 2012.

[163] Z. Pan, B. Ren, and D. Manocha, "Gpu-based contact-aware trajectory optimization using a smooth force model," in *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '19. New York, NY, USA: ACM, 2019, pp. 4:1–4:12.

[164] H.-B. Li, T.-Z. Huang, Y. Zhang, X.-P. Liu, and H. Li, "On some new approximate factorization methods for block tridiagonal matrices suitable for vector and parallel processors," *Mathematics and Computers in Simulation*, vol. 79, no. 7, pp. 2135–2147, 2009.

[165] H.-B. Li, T.-Z. Huang, Y. Zhang, X.-P. Liu, and T.-X. Gu, "Chebyshev-type methods and preconditioning techniques," *Applied Mathematics and Computation*, vol. 218, no. 2, pp. 260–270, 2011.

[166] E. Rothberg and A. Gupta, "An efficient block-oriented approach to parallel sparse cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 15, no. 6, pp. 1413–1439, 1994.

[167] J. Carpentier, "Analytical inverse of the joint space inertia matrix," 2018.

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

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

[170] 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.

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

[172] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 13–22.

[173] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[174] E. Fayneh, M. Yuffe, E. Knoll, M. Zelikson, M. Abozaed, Y. Talker, Z. Shmuely, and S. A. Rahme, "14nm 6th-generation core processor soc with low power consumption and improved performance," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 72–73.

95

Appendix A

Additional Rigid Body Dynamics Algorithms and Refactorings

In this section we present the Recursive Newton Euler Algorithm (RNEA) and the direct inverse of the mass matrix algorithm (M^{-1}). We begin by presenting the standard algorithms and then present the refactored algorithms that are optimized for GPU operation in the GRiD library (Chapter 5).

We note that the direct M^{-1} algorithm is based off of the derivation by Carpentier [167]. In it, π_i refers to the frames of the subtree rooted at frame i. That is, frame i and all of its descendants. For example, for the random robot in Figure 5.3, $\pi_2 = [2,3]$, $\pi_1 = [1,2,3,4]$ and $\pi_6 = [6]$. The notation $[i, i :]$ refers to row i, columns i to n where n is the total number of columns in that matrix. Finally, for $q \in \mathbb{R}^n$ then $M^{-1} \in \mathbb{R}^{n \times n}$ and $F_i \in \mathbb{R}^{6 \times n}$.

Algorithm 8: $RNEA(q, \dot{q}, \ddot{q}, f^{ext}) \to c$

1: $v_0 = 0$, $a_0 = \text{gravity}$

2: **for** frame $i = 1 : n$ **do**

3:      Compute ${}^i X_{\lambda_i}, S_i, I_i$

4:      $v_i = {}^i X_{\lambda_i} v_{\lambda_i} + S_i \dot{q}_i$

5:      $a_i = {}^i X_{\lambda_i} a_{\lambda_i} + S_i \ddot{q}_i + v_i \times S_i \dot{q}_i$

6:      $f_i = I_i a_i + v_i \times^* I_i v_i - f_i^{\text{ext}}$

7: **for** frame $i = n : 1$ **do**

8:      $c_i = S_i^T f_i$

9:      $f_{\lambda_i} \mathrel{+}= {}^i X_{\lambda_i}^T f_i$

Algorithm 9: $M^{-1}(q) \to M^{-1}$

1: $M^{-1}, F = 0$

2: **for** frame $i = 1 : n$ **do**

3:      Compute ${}^i X_{\lambda_i}, S_i, I_i$

4: **for** frame $i = n : 1$ **do**

5:      $U_i = I_i S_i$

6:      $D_i = S_i^T U_i$

7:      $M^{-1}[i, i] = D_i^{-1}$

8:      $M^{-1}[i, \pi_i] \mathrel{-}= D_i^{-1} S_i^T F_i[:, \pi_i]$

9:      **if** $\lambda_i \neq 0$ **then**

10:          $F_i[:, \pi_i] \mathrel{+}= U_i M^{-1}[i, \pi_i]$

11:          $F_{\lambda_i}[:, \pi_i] \mathrel{+}= {}^i X_{\lambda_i}^T F_i[:, \pi_i]$

12:          $I_{\lambda_i} \mathrel{+}= {}^i X_{\lambda_i}^T \left(I_i - U_i D_i^{-1} U_i^T \right) {}^i X_{\lambda_i}$

13: **for** frame $i = 1 : n$ **do**

14:      **if** $\lambda_i \neq 0$ **then**

15:          $M^{-1}[i, i :] \mathrel{-}= D_i^{-1} U_i^{T\,i} X_{\lambda_i} F_{\lambda_i}[:, i :]$

16:      $F_i[:, i :] = S_i M^{-1}[i, i :]$

17:      **if** $\lambda_i \neq 0$ **then**

18:          $F_i[:, i :] \mathrel{+}= {}^i X_{\lambda_i} F_{\lambda_i}[:, i :]$

Algorithm 10: $RNEA\text{-}GRiD(q, \dot{q}, \ddot{q}, f^{ext}) \to c$

1:   $v_0 = 0$, $a_0 = \text{gravity}$

2:   **for** frame $i = 1 : n$ **in parallel do**

3:      Compute $^{i}X_{\lambda_i}, S_i, I_i$

4:      $\alpha_i = S_i \dot{q}_i \quad \beta_i = S_i \ddot{q}_i$

5:   **for** level $l = 0 : l_{max}$ **do**

6:      **for** frame $i \in l$ **in parallel do**

7:        $v_i = {}^{i}X_{\lambda_i} v_{\lambda_i} + \alpha_i$

8:   **for** frame $i = 1 : n$ **in parallel do**

9:      $\beta_i \mathrel{+}= v_i \times \alpha_i$

10:   **for** level $l = 0 : l_{max}$ **do**

11:      **for** frame $i \in l$ **in parallel do**

12:        $a_i = {}^{i}X_{\lambda_i} a_{\lambda_i} + \beta_i$

13:   **for** frame $i = 1 : n$ **in parallel do**

14:      $\gamma_i = I_i v_i \quad f_i = I_i a_i - f_i^{\text{ext}}$

15:      $f_i \mathrel{+}= v_i \times^{*} \gamma_i$

16:   **for** level $l = l_{max} : 0$ **do**

17:      **for** frame $i \in l$ **in parallel do**

18:        $f_{\lambda_i} \mathrel{+}= {}^{i}X_{\lambda_i}^{T} f_i$

19:   **for** frame $i = 1 : n$ **in parallel do**

20:      $c_i = S_i^{T} f_i$

<div align="center">

Algorithm 11: $M^{-1}\text{-}GRiD(q) \to M^{-1}$

</div>

1:   $M^{-1}, F = 0$

2: **for** frame $i = 1 : n$ **in parallel do**

3:      Compute ${}^i X_{\lambda_i}, S_i, I_i$

4:      $U_i = I_i S_i$

5:      $D_i = S_i^T U_i$

6:      $M^{-1}[i, i] = D_i^{-1}$

7:      $\alpha_i = M^{-1}[i, i] S_i^T \qquad \beta_i = M^{-1}[i, i] U_i^T$

8:      $\gamma_i = U_i \beta_i \qquad\qquad\quad \delta_i = \beta_i {}^i X_{\lambda_i}$

9: **for** level $l = l_{max} : 0$ **do**

10:      **for** frame $i \in l$ **in parallel do**

11:          $M^{-1}[i, \pi_i] \mathrel{-}= \alpha_i F_i[:, \pi_i]$

12:          **if** $l > 0$ **then**

13:              $F_i[:, \pi_i] \mathrel{+}= U_i M^{-1}[i, \pi_i]$

14:              $F_{\lambda_i}[:, \pi_i] \mathrel{+}= {}^i X_{\lambda_i}^T F_i[:, \pi_i] \qquad \gamma_i = {}^i X_{\lambda_i}^T (I_i - \gamma_i)$

15:              $I_{\lambda_i} \mathrel{+}= \gamma_i {}^i X_{\lambda_i}$

16: **for** level $l = 0 : l_{max}$ **do**

17:      **for** frame $i \in l$ **in parallel do**

18:          **if** $l > 0$ **then**

19:              $M^{-1}[i, i :] \mathrel{-}= \delta_i F_{\lambda_i}[:, i :]$

20:          $F_i[:, i :] = S_i M^{-1}[i, i :]$

21:          **if** $l > 0$ **then**

22:              $F_i[:, i :] \mathrel{+}= {}^i X_{\lambda_i} F_{\lambda_i}[:, i :]$

Appendix B

CPU Optimized Dynamics Gradients

Our optimized CPU implementation of the gradient of forward dynamics builds on the algorithmic and implementation insights from existing state-of-the-art CPU libraries [100; 102; 103] and is further optimized for use with nonlinear model predictive control based on the algorithmic features presented in Section 5.4.1.

In order to efficiently balance the coarse-grained parallelism exposed by tens to hundreds of computations of the dynamics gradient across a handful of processor cores, each core must work through several computations sequentially. In fact, in order to minimize performance limiting context switches during thread execution, we limit the number of simultaneous threads to the number of logical cores.

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

We also used the Eigen library [168] to vectorize many linear algebra operations, taking advantage of some limited fine-grained parallelism by leveraging the CPU's modest-width vector operations.

The current fastest CPU forward dynamics package, RobCoGen, exploits structured sparsity to increase performance [102; 103]. Building on this approach, we wrote custom functions to exploit the structured sparsity in the dynamics gradient using explicit loop unrolling and zero-skipping.

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

Finally, we note that the Pinocchio Library [100] now supports optimized code-generation of rigid body dynamics algorithms, and their gradients, superseding this implementation.

Appendix C

FGPA and ASIC Optimized Dynamics Gradients

FPGAs have reconfigurable logic blocks and programmable interconnects that allow them to implement custom hardware functional units, data flows, and processing pipelines. FPGA designs often also use *fixed-point* arithmetic to perform math faster while using less energy and area per computation as compared to *floating-point* arithmetic. The trade-off is that the dynamic range and precision of fixed-point numbers are reduced. However, for many applications, this reduction still produces high quality end-to-end solutions (e.g., quantization for neural networks) [169; 170; 171].

To leverage this feature, we first validated that a fixed-point implementation of the dynamics gradient kernel would provide sufficient end-to-end accuracy. We solved the Kuka manipulator motion task from Section 4.4.1, sweeping the numerical data types used in the gradient computation. As shown in Figure C.1, a wide range of fixed-point data types were able to solve the problem successfully. We chose to use a 32-bit integer with 16 bits before and after the decimal point. This is safely in the valid datatype range (as low as 12 bits before and 5 bits after), and integrates easily with the CPU's 32-bit floating-point math.
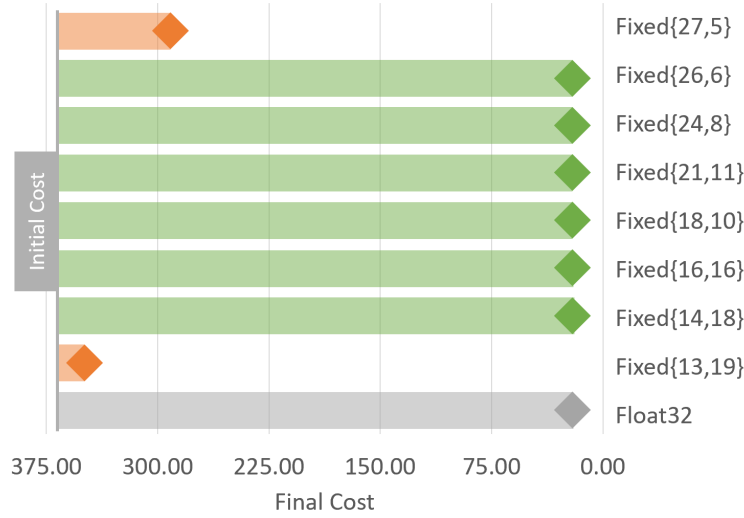
Figure C.1: *A range of fixed-point numerical types, highlighted in green, delivered comparable numerical performance, converging to the same final trajectory cost as the baseline 32-bit floating-point numerical datatype used in our CPU and GPU implementations, highlighted in grey. Types that resulted in algorithms that did not converge to an equivalent solution are highlighted in Orange. The various fixed-point types are labeled as "Fixed{integer bits, decimal bits}".*

Initial tests leveraging fixed-point conversion on the CPU incurred considerable overhead, as such, we instead used dedicated Xilinx IP cores on the FPGA to compute the numerical transformation, reducing overhead latency by as much as 3.6x.

We designed custom functional units and datapaths to exploit fine-grained parallelism. First, we refactored the algorithm, as exemplified in Algorithm 12, to better map the computational patterns to the FPGAs strengths. For example, we computed each column j of $\partial c / \partial u_j$ in parallel datapaths routed through the hardware and exploited parallelism between each brief linear algebra operation, as well as within those operations, by instantiating many multiplier and adder units in parallel. The FPGA can also fully exploit the fine-grained parallelism between different linear algebra operations as all operations are performed natively in independent, custom, parallel hardware circuits. For example, in Lines 3 and 6, cross products and matrix-vector products are executed in parallel using different hardware circuits.

Coarse-grained parallelism, however, was limited in our design because we made a design choice to prioritize reducing latency. This resulted in heavy utilization of limited digital

---

Algorithm 12: $\nabla RNEA\text{-}FPGA(\dot{q}, v, a, f, X, S, I) \rightarrow \partial c/\partial u$

1: **for** link $i = 1 : N$ **do**

2:    $\alpha_i = I_i v_i$

$$\frac{\partial v_i}{\partial u} = {}^i X_{\lambda_i} \frac{\partial v_{\lambda_i}}{\partial u} + \begin{cases} {}^i X_{\lambda_i} v_{\lambda_i} \times S_i & u \equiv q \\[2mm] S_i & u \equiv \dot{q} \end{cases}$$

3:    $\beta_i = \frac{\partial v_i}{\partial u} \times^* \alpha_i \qquad \gamma_i = I_i \frac{\partial v_i}{\partial u}$

$$\frac{\partial a_i}{\partial u} = {}^i X_{\lambda_i} \frac{\partial a_{\lambda_i}}{\partial u} + \frac{\partial v_{\lambda_i}}{\partial u} \times S_i \dot{q}_i + \begin{cases} {}^i X_{\lambda_i} a_{\lambda_i} \times S_i \\[2mm] v_i \times S_i \end{cases}$$

4:    $\frac{\partial f_i}{\partial u} = I_i \frac{\partial a_i}{\partial u} + \beta_i + v_i \times^* \gamma_i$

5: **for** link $i = N : 1$ **do**

6:    $\frac{\partial c_i}{\partial u} = S_i^T \frac{\partial f_i}{\partial u} \quad \delta_i = {}^i X_{\lambda_i}^T \frac{\partial f_i}{\partial u} \quad \zeta_i = {}^i X_{\lambda_i}^T \left(S_i \times^* f_i \right)$

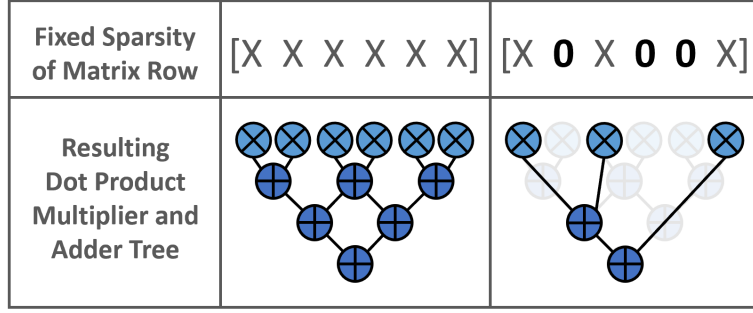7: $\frac{\partial f_{\lambda_i}}{\partial u} \mathrel{+}= \delta_i + \zeta_i$

---

signal processing (DSP) resources on our FPGA platform, with 77.5% of the 6840 DSP blocks used for a single computation in our final design. This is despite careful re-use or *folding* of multiplier resources, which, e.g., reduced the resource requirements for the forward pass of our implementation by about 7x.

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

By creating processing units tailored to the dataflow of the algorithm, our implementation also streamlines sequential chains of dependencies between links by creating dedicated hardware to iterate over those loops with minimal overhead. All of these reductions in overhead are crucial to obtaining a performance advantage on an FPGA, as designs on the reconfigurable circuit

Fixed Sparsity of Matrix Row	[X X X X X X]	[X **0** X **0 0** X]
Resulting Dot Product Multiplier and Adder Tree		

Figure C.2: *An example of a tree of multipliers and adders for a dot product with a dense and known sparse vector. The known sparse vector allows us to reduce a 4-level tree with 6 multiplications and 5 additions to a 3-level tree with 3 multiplications and 2 additions.*

fabric typically have much slower clock speeds than CPUs and GPUs (e.g., 55.6MHz for our design versus 1.7GHz and 3.6GHz for the GPU and CPU in Section 5.5.1).

Finally, we used a framework called Connectal [172] to implement the I/O transfer between the FPGA and CPU. Connectal's support for the FPGA platform we used is currently restricted to PCIe Gen1, limiting our I/O bandwidth. However, by pipelining the I/O with the dynamics gradient computations, we were able to achieve I/O overhead comparable to the GPU.

For our proof-of-concept ASIC, we ran synthesis based on our FPGA design using the Global Foundries 12nm technology node at the typical process corner using the Vivado Design Suite [173]. The maximum clock speed of the core computational pipeline on the ASIC is 7.2x faster than the clock speed of our FPGA implementation immediately accelerating all computations by 7.2x as shown in Figure 7.1. While we did not tape-out a full system-on-chip, our synthesized design had an area of 1.9 mm^2, nearly 65$\times$ smaller than Intel's 14 nm quad-core SkyLake processor [174]. This suggests that many processing pipelines could be fit on a single chip, allowing our ASIC to easily scale to multiple computations.

For more information on our proof-of-concept FPGA and ASIC implementations we suggest reading our RA-L and ASPLOS papers [85; 86].

Appendix D

Reusable Threads

Efficient threading implementations can have a large impact on the final design. As mentioned in Appendix B, good threading implementations can provide as much as a 1.9x speedup in overall system performance.

In many applications that leverage trajectory optimization and rigid body dynamics algorithms, the number of parallel computations is know at compile time. Furthermore, these parallel computations each often repeatedly touch the same memory addresses. To improve cache coherence and to reduce the overhead from launching and joining threads, we designed a custom, header-only, `ResuableThreads` library to enable the easy construction of indexable and synchronizeable persistent threads. This enables specific threads to be sent specific jobs that touch specific areas of memory, repetitively. Our library combines the features of the standard C++ thread library as well as the LLVM ThreadPool library and the C++ header only ThreadPool library, which all provide a higher level and operating system agnostic interface to the UNIX POSIX thread library.

To support the broader robotics research community, our `ResuableThreads` library is publicly available at: https://github.com/plancherb1/ReusableThreads.

Finally, an example usage of our library's C++ API is shown below:

```
// initialize

ReusableThreads<NUM_THREADS> threads;


// add a task where:

//    thread_id is an integer in range(0,NUM_THREADS)

//    function_ptr points to the desired task function

//    varargs is a common seperated list of input arguments

threads.addTask(thread_id, function_ptr, varargs);


// wait until all threads finish their tasks

threads->sync();
```