Differentiable and Constrained Model Predictive Control on the GPU

Gabriel Bravo-Palacios, Jianghan Zhang, Zachary Pestrikov, Brian Plancher, Thomas Lew

***Abstract*—Differentiable model predictive control (MPC) enables the integration of control and learning by embedding optimization structure within end-to-end training pipelines. To ensure the maximal performance of these pipelines, both the models and the solvers must reside on the GPU. While GPU-based solvers have shown to scale well in past work, they largely remain limited to unconstrained problems, restricting their applicability. In this work, we introduce a differentiable, GPU-accelerated, nonlinear MPC framework that can handle state and input constraints via the Alternating Direction Method of Multipliers (ADMM). The method builds on a sequential quadratic programming (SQP) formulation and leverages GPU-optimized linear system solvers for scalability. We demonstrate our approach on drone obstacle avoidance, reinforcement learning, and humanoid imitation learning tasks.**

I. Introduction

Differentiable optimization enables the infusion of optimization into learning pipelines, improving data efficiency and enforcing known structure and constraints. For example, differentiable model predictive control (MPC) unlocks optimization-based policies that have found many applications ranging from parameter estimation, to planning, to reinforcement learning (RL) and imitation learning (IL) [1]–[7].

Despite these successes, the scalability of such approaches remains a key challenge due to the lack of efficient implementations of GPU-based solvers. This challenge is driven by a number of factors including: the sequential nature of optimal control solvers (e.g., leveraging sequential-in-time Riccati recursions), the need to handle structured-sparse linear systems, and historically poor GPU support for sparse matrix factorization. Therefore, approaches that handle general constraints and are differentiable only run on the CPU [8], while existing differentiable MPC solvers on the GPU are limited to unconstrained or box-constrained problems [7], [9], [10].

To address this challenge, we develop a differentiable, GPU-based solver that supports pointwise state and input inequality constraints, slack variables, costs with across-time couplings, and implicit dynamics. We provide this solver as a JAX-native library with a CUDA-native linear-solve backend that exploits parallelism across time steps and batched problem instances. We showcase the solver on control, RL, and IL tasks, demonstrating improvements over state-of-the-art GPU (`mpc.pytorch` [7]) and CPU (`acados` [8]) solvers.

Gabriel Bravo-Palacios, Jianghan Zhang, Zachary Pestrikov, and Brian Plancher are with Dartmouth College, Hanover, NH, USA (Email: {`gbravo, jianghan.zhang.gr, zpestrikov, plancher`}`@dartmouth.edu`)

Thomas Lew is with Toyota Research Institute, Los Altos, CA, USA. (Email: `thomas.lew@tri.global`)

Toyota Research Institute provided funds to support this work, but this article solely reflects the opinions and conclusions of its authors.

II. Differentiable Model Predictive Control

Model Predictive Control (MPC) is a control scheme that operates in a receding-horizon fashion. At each control step, it solves an optimal control problem (**OCP**), executes the first action, and repeats the process from the next state. In this work, we consider **OCP** of the following form, where $x_t \in \mathbb{R}^{n_x}$ and $u_t \in \mathbb{R}^{n_u}$ are states and control inputs, $\xi_t \in \mathbb{R}^m$ are slack variables, and the binary parameter $\delta_\xi \in \{0, 1\}$ toggles the use of slacks:

$$\min_{(x,u,\xi)} \sum_{t=0}^{N-1} \ell_t(x_t, u_t, x_{t+1}, u_{t+1}) + \ell_N(x_N, u_N) + \sum_{t=0}^{N} \tfrac{\gamma}{2}\|\xi_t\|^2 \tag{1a}$$

$$\text{s.t.} \quad f_t(x_{t+1}, u_{t+1}, x_t, u_t) = 0, \quad t = 0, \dots, N-1, \tag{1b}$$

$$h(x_0, u_0) = 0, \tag{1c}$$

$$\underline{g}_t \le g_t(x_t, u_t) + \delta_\xi \xi_t \le \overline{g}_t, \ t = 0, \dots, N. \tag{1d}$$

This formulation includes three features that go beyond what most solvers support, yet are essential in practice. First, the cost admits cross-time couplings (e.g., control-rate penalties $\|u_{t+1} - u_t\|^2$) that promote smooth trajectories and protect actuators. Second, the dynamics constraints accommodate implicit integrators, which are necessary for systems with stiff dynamics. Third, slack variables ensure recursive feasibility of the receding-horizon scheme. Finally, assuming that costs and constraints (ℓ, f, g, h) depend on parameters $\theta \in \mathbb{R}^p$, solutions to **OCP** also depend on θ. We refer to this as **OCP**$_\theta$.

A *differentiable* MPC solver aims to not only solve **OCP**$_\theta$, but also compute the sensitivities (gradients) of the solution with respect to the parameters (weights) θ. This enables easy integration of the solver into machine learning pipelines. Traditionally, solving **OCP**$_\theta$ for a set value of θ is referred to as the *forward pass*, while leveraging sensistivities to update θ is referred to as the *backward pass*.

III. Forward Pass: Solving **OCP**$_\theta$ via `SQP-ADMM`

We solve **OCP**$_\theta$ via sequential quadratic programming (SQP) [11, Chapter 18], as outlined in Algorithm 1. We describe our implementation of this approach next.

A. Convexification: Approximating *OCP*$_\theta$ as a *QP*

With a slight abuse of notations, we denote the variables as

$$x_t := \begin{bmatrix} x_t^{\text{OCP}} \\ u_t^{\text{OCP}} \end{bmatrix} \in \mathbb{R}^n, \ x := \begin{bmatrix} x_0 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^{(N+1)n}, \ n := n_x + n_u.$$

Algorithm 1 SQP-ADMM (Forward Pass)

Inputs: Tolerance ϵ, Max iterations, linesearch parameters
Initial guess: (x, y)

1: **while** not converged **do**
2: **QP** $\leftarrow$ Convexify **OCP** Sec. III-A
3: $(x^+, y^+) \leftarrow$ Solve **QP** via ADMM Sec. III-B
4: $(x, y) \leftarrow$ Linesearch(x^+, y^+, x, y) Sec. III-C
5: Convergence Check Sec. III-C
6: **Return**: Solution (x, y)

At each SQP iteration, we quadratize the objective and linearize the constraints, yielding the quadratic program (QP)

$$\textbf{QP}: \quad \min_{(x,\xi)} \quad \tfrac{1}{2} x^\top P x + q^\top x \tag{2a}$$

$$\text{s.t.} \quad Cx = c, \quad \underline{G} \le Gx \le \overline{G}. \tag{2b}$$

$P \succeq 0$ and q are the Hessian and gradient of the costs (1a), $Cx = c$ include the linearized dynamics (1b) and initial-state equality constraints (1c), and Gx stacks the linearized inequalities (1d) with shifted bounds $\underline{G}, \overline{G}$.

B. Solving *QP* via ADMM (Algorithm 2)

We solve **QP** using the OSQP ADMM scheme [12], tailored to the block-sparse (P, C, G) structure of **QP**. We use ADMM as it can quickly solve problems to moderate precision and can be warm started with infeasible initial guesses. First, we introduce slack variables z and rewrite **QP** as

$$\min_{(x,z)} \quad \tfrac{1}{2} x^\top P x + q^\top x \quad \text{s.t.} \quad Ax - z = 0, \quad z \in [l, u], \tag{3}$$

with $A = \begin{bmatrix} C & G \end{bmatrix}^\intercal, l = \begin{bmatrix} c & \underline{G} \end{bmatrix}^\intercal, u = \begin{bmatrix} c & \overline{G} \end{bmatrix}^\intercal$.

ADMM solves this problem in three steps: a primal, slack, and dual update. At an optimal solution (x, z) of **QP**, the KKT conditions state that there exists a multiplier y such that

$$Px + q + A^\top y = 0, \; Ax - z = 0, \; z \in [l, u], \; y \in N_{[l,u]}(z).$$

Thus, we define the primal and dual residuals

$$r_{\text{prim}} = Ax - z, \quad r_{\text{dual}} = Px + q + A^\top y, \tag{4}$$

and terminate when:

$$\|r_{\text{prim}}\|_\infty \le \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}} \max\{\|Ax\|_\infty, \|z\|_\infty\}, \tag{5}$$
$$\|r_{\text{dual}}\|_\infty \le \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}} \max\{\|Px\|_\infty, \|A^\top y\|_\infty, \|q\|_\infty\}.$$

We use the hyperparameter selection rule in OSQP [12]. We partition all the variables into equality and inequality blocks (e.g., (z_f, z_g), $\text{diag}(\rho_f I, \rho_g I)$). All matrix-vector products exploit the block-sparse, stage-decoupled structure of (P, C, G). E.g., since the equality slack satisfies $z_f = c$ after the first iteration, we eliminate it entirely and parallelize the remaining operations over time steps. Finally, we avoid enlarging the QP by handling slack variables via the smooth projection of [13].

The computational bottleneck of Algorithm 2 is the linear system solve in the primal update (6). To accelerate it, we provide a CUDA-native backend designed to exploit parallelism across both time steps and across batched problem instances through the use of cuDSS [14].

Algorithm 2 ADMM for solving **QP**

Inputs: $P, q, A, l, u, \sigma, \rho, \alpha \in (0, 2)$
Initial guess: $x^0, z^0, \tilde{x}^0, \tilde{z}^0, y^0$
while not converged **do**
(**Primal update**):

$$(\tilde{x}^{k+1}, \tilde{z}^{k+1}) \leftarrow \arg\min_{\tilde{x}, \tilde{z}} \; \{\tfrac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \tfrac{\sigma}{2}\|\tilde{x} - x^k\|^2 + \tfrac{\rho}{2}\|\tilde{z} - z^k + \rho^{-1} y^k\|^2 \quad \text{s.t.} \quad A\tilde{x} = \tilde{z}\}. \tag{6}$$

(**Slack update**):

$$x^{k+1} \leftarrow \alpha \tilde{x}^{k+1} + (1 - \alpha) x^k, \tag{7a}$$
$$z^{k+1} \leftarrow \Pi_{[l,u]}\big(\alpha \tilde{z}^{k+1} + (1 - \alpha) z^k + \rho^{-1} y^k\big). \tag{7b}$$

(**Dual update**):

$$y^{k+1} \leftarrow y^k + \rho\big(\alpha \tilde{z}^{k+1} + (1 - \alpha) z^k - z^{k+1}\big). \tag{8}$$

Update the residuals $(r_{\text{prim}}, r_{\text{dual}}) = (4)$
Check for convergence with (5)
Update the stepsize parameter ρ

C. Linesearch and Convergence Check

We use a standard backtracking linesearch as described in [11, Algorithm 18.3]. We terminate SQP-ADMM once the first-order optimality conditions are sufficiently satisfied as follows

$$\|\nabla \ell(x) + y_f^\top \nabla f(x) + y_g^\top \nabla g(x)\|_\infty \le \epsilon_c, \tag{9}$$
$$\|f(x)\|_\infty \le \epsilon_c, \quad \|g(x)\|_{\infty,[\underline{g},\overline{g}]} \le \epsilon_c, \tag{10}$$
$$\|x^{i+1} - x^i\|_\infty \le \epsilon_c, \tag{11}$$

where $\|g(x)\|_{\infty,[\underline{g},\overline{g}]}$ denotes the maximum componentwise violation of the inequality bounds, and $\epsilon_c > 0$ is a tolerance.

IV. Backward Pass: Computing Sensitivities

We compute solution sensitivities of $\textbf{OCP}_\theta$ via the implicit function theorem, solving a linear system with the converged solution's active inequality and equality constraints rather than differentiating through unrolled solver iterations.

At convergence of SQP-ADMM, the primal-dual variables (x, y) satisfy the KKT conditions of $\textbf{OCP}_\theta$:

$$\nabla \ell_\theta(x) + y_f^\top \nabla f_\theta(x) + y_{g,\text{active}}^\top \nabla \Delta g_{\theta,\text{active}}(x) = 0, \tag{12a}$$
$$f_\theta(x) = 0, \tag{12b}$$
$$\Delta g_{\theta,\text{active}}(x) = 0, \tag{12c}$$

where we define $\Delta g_\theta(x) = \min\{g_\theta(x) - \underline{g}_\theta, \overline{g}_\theta - g_\theta(x)\}$, and note that active constraints satisfy $\Delta g_{\theta,\text{active}}(x) = 0$, and the inactive constraints satisfy $\Delta g_{\theta,\text{inactive}}(x) > 0$. In practice, we identify active constraints as those satisfying

$$\Delta g_{\theta,i}(x) \le \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}} \begin{cases} |\overline{g}_{\theta,i}| & \text{if } \Delta g_{\theta,i}(x) = \overline{g}_{\theta_i} - g_{\theta,i}(x), \\ |\underline{g}_{\theta,i}| & \text{otherwise.} \end{cases}$$

We can compactly write the KKT conditions (12) as

$$F((x, y)(\theta), \theta) = 0. \tag{13}$$

From (13), we compute sensitivities using the implicit function theorem as in [7], [8], [10].

Vector-Jacobian Product (VJP): In machine learning applications, one typically uses the gradient of a function $\mathcal{L}$ (e.g., a loss or reward) of the solution x to $\mathbf{OCP}_\theta$. Such gradients are more efficiently computed using the VJP

$$\frac{\partial \mathcal{L}}{\partial \theta}^\top = -\frac{\partial F}{\partial \theta}^\top \left(\begin{bmatrix} \frac{\partial F}{\partial w} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial x} \\ 0 \end{bmatrix} \right). \tag{14}$$

Computing (14) involves solving the linear system $\begin{bmatrix} \frac{\partial F}{\partial w} \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{\lambda} \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$, with the sparsity structure $\begin{bmatrix} \frac{\partial F}{\partial w} \end{bmatrix} = \begin{bmatrix} \star & \star \\ \star & 0 \end{bmatrix}$.

V. Reinforcement Learning & Imitation Learning

Our MPC solver can be used as a *differentiable policy class* for reinforcement learning (RL) and imitation learning (IL):

$$\textbf{RL:} \quad \max_\theta \ \mathbb{E}\left[\sum_{t=1}^{H} R(x_t, \pi_0^\theta(x_t))\right], \ x_{t+1} = \texttt{Sim}(x_t, \pi^\theta(x_t)),$$

$$\textbf{IL:} \quad \min_\theta \ \mathbb{E}\left[\|(\hat{u}_0, \ldots, \hat{u}_T) - \pi_{0:T}^\theta(x_0)\|^2\right],$$

where $\pi_{0:T}^\theta(x_0)$ is the control input solution of $\mathbf{OCP}_\theta$, `Sim` is a differentiable simulator, and $(\hat{u}_0, \ldots, \hat{u}_T)$ is demonstration data from an expert policy. Compared to black-box neural networks, this MPC policy class leverages the structure of optimal control. Its inductive biases allow zero-shot generalizations to new problem instances, e.g., without the need to train a policy over multiple tasks if only the reference tracking cost changes, see [10]. Moreover, the solver is tailored to the GPU, so it supports using larger batch sizes and expressive models.

VI. Results & Discussion

We compare `DiffMPC` with two differentiable MPC solvers: the PyTorch-based iLQR solver `mpc.pytorch` [7] and the C-based solver `acados` [8]. `mpc.pytorch` can handle control bounds only, while `DiffMPC` and `acados` handle state and control constraints. We conduct the evaluation on constrained optimization, reinforcement learning, and imitation learning tasks.[1]

A. Constrained Optimization

We demonstrate state and control constraint support, with a 6-DoF drone obstacle avoidance task with three circular obstacles in the (x, y) plane, initial and final state constraints, and control box constraints. We test across five noisy initial states ($\sim \mathcal{N}(0, 0.05)$), two warm-start strategies (above-obstacle arc and straight line), and two time resolutions ($\Delta t \in \{1.0, 0.25\}$, $N = 50\,\text{s}/\Delta t$), using RK4 integration. `DiffMPC` uses $\{4, 8\}$ iterations per MPC solve. As shown in Figure 1, the drone avoids the obstacles reaching the target with a terminal error of 0.084 (line init) to 0.106 (arc init), and near-zero constraint violations ($< 1.3\%$ obstacle-radius penetration).

[1]We collect results on a NVIDIA GeForce RTX 5090 GPU and a 12th Gen Intel(R) Core(TM) i9-12900K CPU with Ubuntu 22.04 and CUDA 13.0.

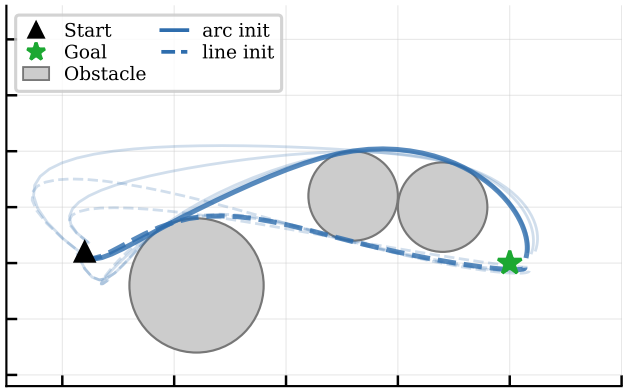

Fig. 1. Drone obstacle avoidance problem: Closed-loop trajectories initialized with 5 random initial conditions (faint lines) and 2 initial guesses (arc and line) (Δt=1.0 *sec.*, 8 max. SQP iterations).

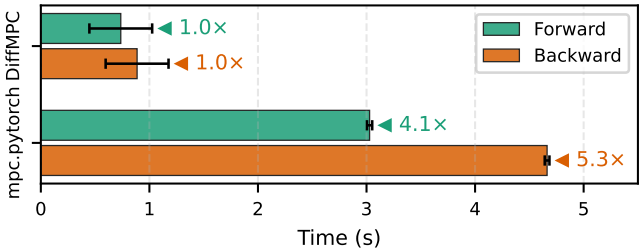

Fig. 2. Linear-System RL Problem: Mean solve time (seconds) per solver at batch size $B = 256$ and problem dimensions $n_x = 8$ and $n_u = 4$, averaged over planning horizons $N \in \{30, 40, 50\}$ and 10 repeated random seeds. Error bars denote 95% confidence intervals over seeds. Annotations indicate the relative slowdown factor with respect to `DiffMPC` (1.0×).

B. Reinforcement Learning

We consider the RL formulation in [10] for linear-quadratic optimal control problems with box inequality constraints. The discretized linear dynamics are $x_{t+1} = Ax_t + Bu_t + b$ for $t = 0, \ldots, T-1$, with $x_t \in \mathbb{R}^{n_x}$, $u_t \in \mathbb{R}^{n_u}$, and control bounds $\|u_t\|_\infty \leq u_{\max}$, with $u_{\max} \in \{1, 10\}$. The dynamics matrices are randomized as $A = I + 0.1\,\Delta A$, $\text{vec}(\Delta A) \sim \mathcal{N}(0, I)$, with eigenvalues clipped to $|\lambda(A)| \in (0, 0.99]$; $\text{vec}(B) \sim \mathcal{N}(0, I)$; $b \sim \mathcal{N}(0, 10^{-4} I)$; and initial conditions $x_0 \sim \mathcal{N}(0, 25I)$. The stage cost is quadratic, $c_t^x(x) = x^\top Q x$ with $Q = \text{diag}(\theta)$, $c_t^u(u) = \|u\|_2^2$, and the RL reward is $R(x, u) = -(\|x\|_2^2 + \|u\|_2^2)$. The learned parameters are $\theta = \text{diag}(Q)$. Nominally, we use the state-control dimensions $(n_x, n_u) = (8, 4)$, a planning horizon $T = 40$, an episode length $H = 50$, and batch sizes $B = 64$. Then, we vary each parameter independently. Statistics are computed over ten independent runs per configuration.

The GPU advantage of `DiffMPC` increases with the batch size and the problem size. `mpc.pytorch` is out-performed by the other solvers, with `DiffMPC` outperforming it by 4-5× at larger batch sizes (Figure 2). `acados` is the fastest method for single problem instances, and scales well on the CPU to larger problems and batch sizes thanks to its use of efficient linear algebra routines and parallelization across CPU cores. `DiffMPC`, thanks to leveraging GPU parallelization, the time-induced sparsity of the problem, and cuDSS for solving the

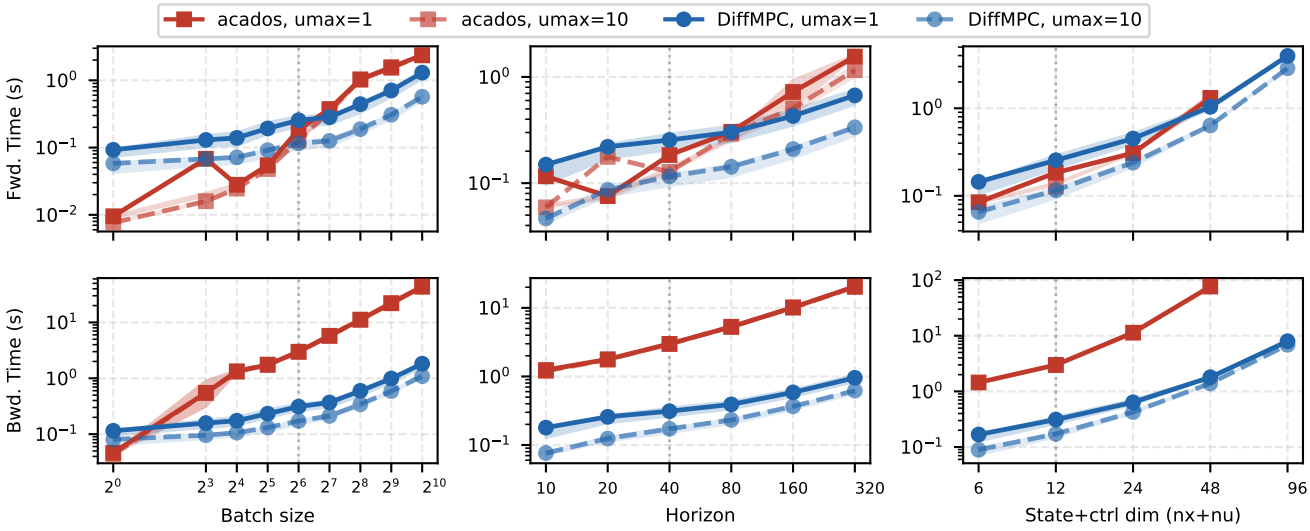

Fig. 3. Linear-system RL: Mean solve time (seconds), averaged over 10 random seeds, as a function of batch size (left), planning horizon (center), and state + control dimension (right) for $u_{max} = 1, 10$, demonstrating that at scale `DiffMPC` can provide order-of-magnitude improvements.

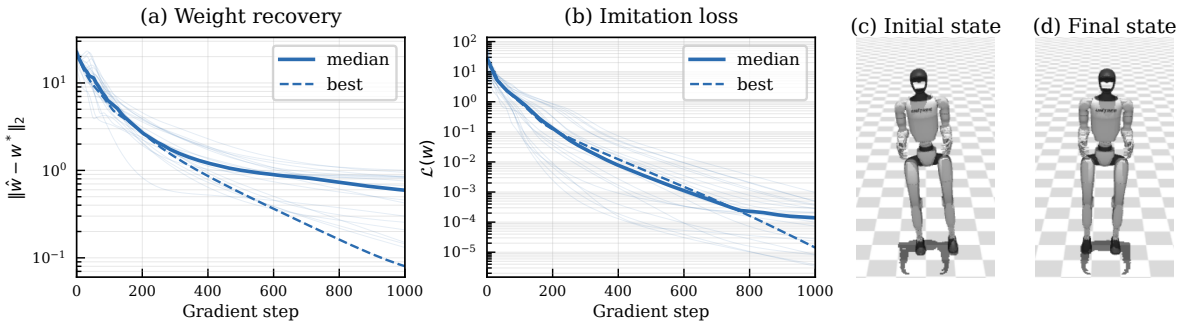

Fig. 4. Imitation learning on SRB balancing: (a) weight recovery and (b) imitation loss over gradient steps for `DiffMPC` across 20 random initializations. Solid lines show the median; dashed lines show the best run. Right: Closed-loop balancing in MuJoCo with MPC weights learned via imitation learning showing an (c) initial perturbed state and an (d) equilibrium state reached using recovered weights.

primal problems in the `ADMM` loop, outperforms `acados` for larger problems and batch sizes. In particular, for the more computationaly intensive backward pass, `DiffMPC` is able to acheive order-of-magnitude speedups at large batch sizes and horizons. As shown in Fig. 3, the forward-pass crossover occurs between batches 64 and 128. The highest speedups are achieved in the batch sweeps ($Horizon = 40$, $dim = 12$, $u_{max} = 1$), with the forward pass peaking at ≈2.3× (batch 256) and the backward pass reaching ≈24× at batch 1024.

C. Imitation Learning for Legged Robotics

We address the problem of recovering unknown cost weights $w^\star \in \mathbb{R}^{12}$ from expert demonstrations of a humanoid robot performing a standing-balance task. As expert and student policies, we use an MPC using Single Rigid Body (SRB) centroidal dynamics, friction-cone box constraints, with the state $x \in \mathbb{R}^{12}$ (CoM position, ZYX Euler angles, linear and angular velocities), and the control $u \in \mathbb{R}^6$ representing ground reaction forces. We consider the Unitree G1 humanoid.

Following the imitation-learning paradigm, we collect a dataset $\mathcal{D} = \{(x_0^{(i)}, u_{0:N-1}^{*(i)})\}_{i=1}^{d=100}$ of expert demonstrations, and use the imitation loss $\mathcal{L}(w) = \sum_{i=1}^{100} \sum_{k=0}^{H} \left\| u_k^*(x_0^{(i)}; w) - u_k^{*(i)} \right\|^2$. The weight recovery quality is measured by the model loss $\|\hat{w} - w^*\|_2$. We minimize $\mathcal{L}$ with respect to $\log w$ using the Adam optimizer from $N_{\text{inits}} = 20$ random initializations ($w_0 \sim \mathcal{U}(0.5, 12.0)$, per dimension).

Figure 4 summarizes the results. At step 1000, the median weight-recovery error is $\|\hat{w} - w^*\|_2 = 0.59$, with 19 of 20 runs finishing below 1.0 and the best reaching 0.080. The median imitation loss reaches 1.4×10^{-4}, with the best run at 3.4×10^{-6}. The training traces cluster tightly and descend early (Fig. 4), suggesting that gradients through the GPU-batched KKT system are consistent across initializations. Figure 4 (far right and center right) shows the start and end of a single MPC solve from a perturbed initial state under closed-loop whole-body control via `MuJoCo XLA` [15], as a first step toward a full-locomotion pipeline running entirely on the GPU.

VII. Conclusion and Future Work

This paper presents a new JAX-based, GPU-accelerated, differentiable optimization framework for model predictive control. Our framework leverages the use of ADMM to handle state and control constraints while efficiently scaling to the requirements of learning-based methods like RL and IL. In future work we aim to extend our work to support alternative linear system solver backends (e.g., [16], [17]), deploy the solver onto physical robots, and open-source the code.

REFERENCES

[1] P. Karkus, B. Ivanovic, S. Mannor, and M. Pavone, "Diffstack: A differentiable and modular control stack for autonomous vehicles," in *Conf. on Robot Learning*, 2023.

[2] M. Cummins, A. Padoan, K. Moffat, F. Dörfler, and J. Lygeros, "DeePC-Hunt: Data-enabled predictive control hyperparameter tuning via differentiable optimization," in *Learning for Dynamics & Control Conference*, 2025.

[3] B. Landry, Z. Manchester, and M. Pavone, "A differentiable augmented lagrangian method for bilevel nonlinear optimization," in *Robotics: Science and Systems*, 2019.

[4] M. Bhardwaj, B. Boots, and M. Mukadam, "Differentiable Gaussian process motion planning," in *Proc. IEEE Conf. on Robotics and Automation*, 2020.

[5] A. Romero, Y. Song, and D. Scaramuzza, "Actor-critic model predictive control," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 14 777–14 784.

[6] R. Grandia, F. Farshidian, E. Knoop, C. Schumacher, M. Hutter, and M. Bächer, "DOC: Differentiable optimal control for retargeting motions onto legged robots," *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–14, Jul. 2023.

[7] B. Amos, I. D. J. Rodriguez, J. Sacks, B. Boots, and Z. Kolter, "Differentiable MPC for end-to-end planning and control," *Conf. on Neural Information Processing Systems*, vol. 31, 2018.

[8] J. Frey, K. Baumgartner, G. Frison, D. Reinhardt, J. Hoffmann, L. Fichtner, S. Gros, and M. Diehl, "Differentiable nonlinear model predictive control," *Available at https://arxiv.org/abs/2505.01353*, 2025.

[9] R. Frostig, S. Sindhwani, and S. Tu, "Trajax," 2021. [Online]. Available: http://github.com/google/trajax

[10] E. Adabag, M. Greiff, J. Subosits, and T. Lew, "Differentiable model predictive control on the GPU," in *Int. Conf. on Learning Representations*, 2026.

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

[12] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: an operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.

[13] T. Lew, M. Greiff, J. Subosits, and B. Plancher, "Solving quadratic programs with slack variables via ADMM without increasing the problem size," in *European Control Conference*, 2026.

[14] NVIDIA, "cudss," 2026. [Online]. Available: https://developer.nvidia.com/cudss

[15] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[16] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "MPCGPU: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the GPU," in *Proc. IEEE Conf. on Robotics and Automation*, 2024.

[17] R. Schwan, D. Kuhn, and C. N. Jones, "GPU-accelerated Cholesky factorization of block tridiagonal matrices," 2026, available at https://arxiv.org/abs/2601.03754.