2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

GATO: GPU-Accelerated and Batched Trajectory Optimization for Scalable Edge Model Predictive Control

Alexander Du[1], Emre Adabag[1,2], Gabriel Bravo-Palacios[3,4], Brian Plancher[3,4]

***Abstract*— While Model Predictive Control (MPC) delivers strong performance across robotics applications, solving the underlying (batches of) nonlinear trajectory optimization (TO) problems online remains computationally demanding. Existing GPU-accelerated approaches either parallelize single solves, handle large batches at sub-real-time rates, or sacrifice model generality for speed. This leaves a large gap in solver performance for many state-of-the-art MPC applications that require real-time batches of tens to low-hundreds of solves. As such, we present GATO, an open source, GPU-accelerated, batched TO solver co-designed across algorithm, software, and computational hardware to deliver real-time throughput for these moderate batch size regimes. Our approach leverages a combination of block-, warp-, and thread-level parallelism within and across solves for ultra-high performance. We demonstrate the effectiveness of our approach through a combination of: simulated benchmarks showing speedups of $18-21\times$ over CPU baselines and $1.4-16\times$ over GPU baselines as batch size increases; case studies highlighting improved disturbance rejection and convergence behavior; and finally a validation on hardware using an industrial manipulator. We open source GATO to support reproducibility and adoption.**

I. Introduction

Model Predictive Control (MPC) is a feedback control strategy which has seen great success in a wide variety of robotic applications [1], [2], [3], [4], [5]. Most implementations of (nonlinear) MPC leverage trajectory optimization (TO) [6] to solve the underlying optimal control problems. Historically, these TO problems are solved through 1st- or 2nd-order optimization methods. Unfortunately, such problems are computationally expensive and only deliver locally optimal solutions. As such, several recent efforts have leveraged careful approximations and simplifications of the underlying optimal control problem [7], [8], [9], as well as hardware acceleration, most commonly on GPUs, to help overcome these computational limitations. These GPU-accelerated efforts include both the development of 0th order methods that construct sample-based approximate gradients [10], [11], [12], [13], [14] as well as hybrid, 1st-, and 2nd-order methods targeting both the overall solvers [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], as well as the underlying numerical linear algebra and physics kernels [27], [28], [29], [30], [31]. Importantly,

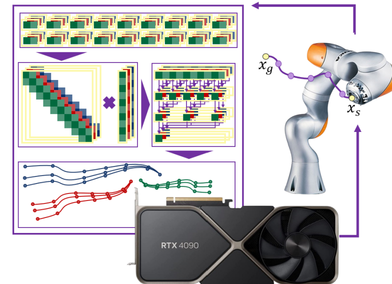


Fig. 1: The GATO solver parallelizes across batches of trajectory optimization solves on the GPU through algorithm-software-hardware co-design. This approach enables real-time performance for batch sizes of tens to low-hundreds of solves with tens to low-hundreds of knot points per solve.

this collection of works demonstrates robust, real-time, real-world usability though numerous deployments onto various modalities of physical robot hardware.

At the same time, there have been a number of recent applications in which batches of tens to low-hundreds of trajectory optimization solves can be leveraged for state-of-the-art MPC performance [22], [26], [32], [33], [34], [35], [36]. And while many of these results are demonstrated through GPU parallelism, in general, whether through 0th-, 1st-, 2nd-order, or hybrid approaches, existing GPU-accelerated solvers are designed to either parallelize a single solve across a GPU, implement large-scale (e.g., >1000) parallel batches of solves, or are special cased for a very limited setup. As such, to the best of the authors' knowledge, for batches of tens to low-hundreds of solves, prior solvers trade off latency, throughput, and generality: some hit kHz rates but only for a few parallel solves; others process large batches but miss real-time targets; still others attain speed by restricting the problem specification (e.g., point-mass models, a single linearization). This fundamentally limits their deployed use, despite their demonstrated real-world promise.

To overcome these challenges, we developed GATO (Figure 1), a GPU-accelerated, batched trajectory optimization solver designed to enable real-time batched solves of tens to low-hundreds of trajectory optimization problems. Our work is inspired by the MPCGPU solver [21], which demonstrated that GPU-acceleration through careful algorithm-hardware-software co-design can enable long-horizon, real-time performance. While MPCGPU is limited to a single solve per GPU, we designed GATO to solve tens to low-hundreds of problems simultaneously. This is done via block-, warp-, and thread-level parallelism both across and within underlying computations for efficient problem matrix formation, linear

This material is based upon work supported by the National Science Foundation (under Awards 2246022, 2411369). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding organizations.

[1] School of Engineering and Applied Science, Columbia University.
[2] University of Michigan
[3,4] Barnard College, Columbia University and Dartmouth College
Correspondence to: plancher@dartmouth.edu

system solves, and line search iterate computations.

We demonstrate the power of this GPU-first framework through a series of simulation benchmarks, case studies, and a hardware demonstration on an industrial manipulator. We find that GATO provides speedups of up to $18-21\times$ over CPU baselines and $1.4-16\times$ over GPU baselines as batch size increases. Our case studies highlight how such batched solves can improve disturbance rejection and convergence behavior of TO and MPC and can run in real-time on robot hardware. We release our software open source at: https://github.com/a2r-lab/GATO.

II. Background

A. Direct Trajectory Optimization

Trajectory optimization [6] solves an (often) nonlinear optimization problem to compute a robot's path through an environment as a series of states $X=\{x_0,\cdots,x_N\}$ and controls $U=\{u_0,\cdots,u_{N-1}\}$ for $x\in\mathbb{R}^n$ and $u\in\mathbb{R}^m$. These problems model the robot as a discrete-time dynamical system, $x_{k+1}=f(x_k,u_k,h)$, with initial condition $x_0=x_s$ and timestep h, and minimize an additive cost function, $J(X,U)$. Recent work has shown that direct methods, which explicitly represent the states, controls, dynamics, and any additional constraints, lead to moderately large nonlinear programs with structured sparsity patterns [37]. These approaches can be greatly accelerated on the GPU, especially as the size of the problem increases [21]. Direct methods follow a three-step process which is repeated until convergence [37], [38], [39]:

Step 1: Form the following quadratic program via a second-order Taylor expansion of the problem along a nominal trajectory, where Q, R and q, r are the Hessians and gradients of J, and A and B are the gradients of f, with respect to x and u, and $e_k=f(x_k,u_k,h)-x_{k+1}$:

$$\begin{aligned}\min_{\delta X,\delta U}\ & \tfrac{1}{2}\delta x_N^T Q_N\delta x_N+q_N^T\delta x_N+\\ & \sum_{k=0}^{N-1}\tfrac{1}{2}\delta x_k^T Q\delta x_k+q^T\delta x_k+\tfrac{1}{2}\delta u_k^T R\delta u_k+r^T\delta u_k, \quad (1)\\ \text{s.t.}\ & \delta x_0=x_s-x_0,\\ & \delta x_{k+1}-A_k\delta x_k-B_k\delta u_k=e_k,\ \forall k=0,\dots,N-1.\end{aligned}$$

Step 2: Compute $\delta X^*,\delta U^*$ by solving the KKT system:

$$\begin{bmatrix}G & C^T\\ C & 0\end{bmatrix}\begin{bmatrix}-\delta Z\\ \lambda\end{bmatrix}=\begin{bmatrix}g\\ c\end{bmatrix}, \tag{2}$$

where $\delta z_k=\begin{bmatrix}\delta x_k & \delta u_k\end{bmatrix}^T$, $\delta z_N=\delta x_N$,

$$\begin{aligned}G&=\begin{bmatrix}Q_0 & & & \\ & R_0 & & \\ & & \ddots & \\ & & & Q_N\end{bmatrix},\\ g&=\begin{bmatrix}q_0 & r_0 & q_1 & r_1 & \dots & q_N\end{bmatrix}^T,\\ C&=\begin{bmatrix}I & & & & & \\ -A_0 & -B_0 & I & & & \\ & & \ddots & -A_{N-1} & -B_{N-1} & I\end{bmatrix},\\ c&=\begin{bmatrix}x_s-x_0 & e_0 & e_1 & \dots & e_{N-1}\end{bmatrix}^T.\end{aligned}$$

Step 3: Apply the update step, $\delta X^*,\delta U^*$, while ensuring descent on the original nonlinear problem through the use of a merit-function and a line-search [37].

Within that framework, Adabag et al. [21], leveraged a symmetric stair preconditioner [40] to solve the KKT system (2) through a Schur complement, preconditioned conjugate gradient, iterative linear system solve, and a parallel line search with the L1 merit function:

$$\mathcal{M}(X,U)=J(X,U)+\mu|c|. \tag{3}$$

This approach maximizes parallel performance on the GPU, but is customized for solving only a single problem while utilizing the entire GPU. In Section III we develop a computational approach that leverages similar underlying algorithmic approaches but enables high-performance for batches of tens to low hundreds of solves.

B. Schur Complement Iterative Methods

Iterative methods solve the problem $S\lambda^*=\gamma$ for a given S and γ by iteratively refining an estimate for λ up to tolerance ϵ. The most popular of these methods is the conjugate gradient (CG) method, which is used in the current state-of-the-art, GPU-accelerated TO solver [21], and also for general, large-scale optimization problems on the GPU [27], [28]. The convergence rate of CG is directly related to the spread of the eigenvalues of S. Thus, a preconditoning matrix $\Phi\approx S$ is often applied to instead solve the equivalent problem with better numerical properties: $\Phi^{-1}S\lambda^*=\Phi^{-1}\gamma$. To do so, the preconditioned conjugate gradient (PCG) algorithm leverages matrix-vector products with S and Φ^{-1}, as well as vector reductions, both parallel friendly operations.

The PCG algorithm also requires the linear system S to be symmetric positive definite. As such, per [21], we form the *Schur Complement*, S, and then solve (2) as follows:

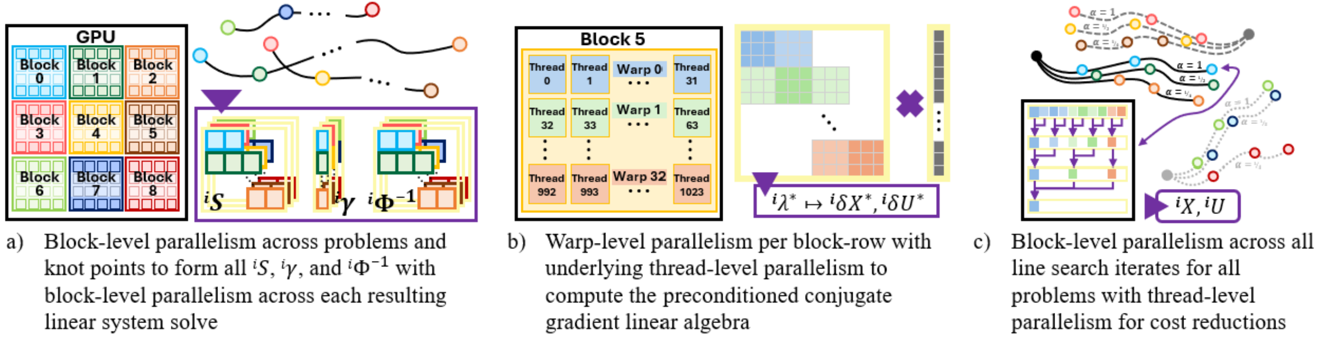
$$\begin{aligned}S&=-CG^{-1}C^T, & \gamma&=c-CG^{-1}g,\\ S\lambda^*&=\gamma, & \delta Z^*&=-G^{-1}(g-C^T\lambda^*).\end{aligned} \tag{4}$$

By defining the variables θ, ϕ, and ζ:

$$\begin{aligned}\theta_k&=A_kQ_k^{-1}A_k^T+B_kR_k^{-1}B_k^T+Q_{k+1}^{-1},\\ \phi_k&=-A_kQ_k^{-1},\\ \zeta_k&=-A_kQ_k^{-1}q_k-B_kR_k^{-1}r_k+Q_{k+1}^{-1}q_{k+1},\end{aligned} \tag{5}$$

S, γ, and the symmetric stair preconditioner, Φ^{-1} [40], take the following forms, where S and Φ^{-1} are block-tridiagonal:

$$\begin{aligned}S&=-\begin{bmatrix}Q_0^{-1} & \phi_0^T & \\ \phi_0 & \theta_0 & \phi_1^T\\ & \phi_1 & \theta_2\\ & & & \ddots\end{bmatrix},\\ \gamma&=c-\begin{bmatrix}Q_0^{-1}q_0 & \zeta_0 & \zeta_1 & \dots & \zeta_{N-1}\end{bmatrix}^T,\\ \Phi^{-1}&=-\begin{bmatrix}Q_0 & -Q_0\phi_0^T\theta_0^{-1} & \\ -\theta_0^{-1}\phi_0Q_0 & \theta_0^{-1} & -\theta_0^{-1}\phi_1^T\theta_1^{-1}\\ & -\theta_1^{-1}\phi_1\theta_0^{-1} & \theta_1^{-1}\\ & & & \ddots\end{bmatrix}.\end{aligned} \tag{6}$$



a) Block-level parallelism across problems and knot points to form all ${}^{i}S$, ${}^{i}\gamma$, and ${}^{i}\Phi^{-1}$ with block-level parallelism across each resulting linear system solve

b) Warp-level parallelism per block-row with underlying thread-level parallelism to compute the preconditioned conjugate gradient linear algebra

c) Block-level parallelism across all line search iterates for all problems with thread-level parallelism for cost reductions

Fig. 2: Overall design of our batched solver which a) forms problems in parallel across solves and timesteps, b) leverages warp-level parallelism within each block-based solve, and c) again leverages large-scale parallelism across the whole GPU for the line search and merit function calculations.

III. Design and Implementation

In this section, we describe the design of GATO as visualized in Figure 2 and described in the pseudocode in Algorithm 1. The solver architecture is optimized for GPU-parallel computations across tens to low-hundreds of trajectory optimization solves (M), each with tens to low-hundreds of timesteps (N). This paradigm, as noted in the introduction, is commonly found across robotics applications and is underserved by current solvers.

Our overall design is inspired by MPCGPU [21] and leverages a similar GPU-first architecture and overall 3-step design flow. However, while MPCGPU is customized for single-solve performance, we designed a new underlying solver to enable high-performance parallelism across multiple solves without sacrificing solver accuracy. We also implemented a number of additional fine-grained parallelism optimizations both across and within solves. As shown in Section IV this improves performance across all batch sizes.

At a high level, our design leverages block-based parallelism to divide up discrete naturally parallel components of each step of our batched solve. Depending on the stage of the solver this either happens at the timestep level or problem level. Within each block we leverage warp- and thread-level intrinsics and parallelism to maximize performance and minimize overheads. Finally, by moving all of the computation onto the GPU we avoid costly I/O penalties. In the remainder of this section we detail our design.

A. Batched Problem Setup and Line Search

GATO is designed to maximize all possible parallelism arising from the computational structure of the underlying (batched) optimal control problems. This is most apparent in the problem setup and line search steps (shown as steps a and c in Figure 2). Here we must form S, γ, and Φ^{-1} per equation 6 and solve a line search for the final update to Z, namely $Z \leftarrow Z + \alpha^* \delta Z^*$, under a merit function, $\mathcal{M}$:

$$\alpha^* = \arg\min_{\alpha_i} \mathcal{M}(Z + \alpha_i \delta Z^*) \quad \alpha \in [1/\beta^0, \ldots, 1/\beta^{\mathcal{A}}], \quad (7)$$

where $\beta > 1$ and $\mathcal{A}$ are positive integer values, with $\mathcal{A}$ representing the number of line search iterates. Throughout the steps, we compute gradients and Hessians of the costs and dynamics functions across all problems and timesteps ($N * M$ total timesteps), as well as compute the merit function values to support our line search, again requiring underlying cost and dynamics calculations across all problems, timesteps, and line search iterates ($N * M * \mathcal{A}$ total timesteps). As such, we exploit block-based parallelism for each timestep to maximize both the independent nature of these computations, as well as opportunities for within-computation thread-based parallelism for the underlying small-scale linear algebra. We use the GRiD [30] library for efficient dynamics (gradient) computations, which follows a similar computational model.

Importantly, because all computations to form S, γ, and each timestep's merit function are block-local, only cheap *intra*-block synchronizations are needed. Only a single *grid-wide* synchronization is required to finalize Φ^{-1}, and a block-level reduction is used to compute the merit function for each line search iterate across all batches of solves.

Throughout these computations, temporary variables are computed in fast shared memory and all final matrices and vectors are arranged densely and contiguously in global memory to maximize naturally coalesced loads and stores by the downstream PCG solver. We also reserve the device's persisting L2 cache to reduce global memory access during PCG. Most importantly, only the current system state(s) and goal(s), as well as the final optimized state and control trajectories, incur round-trip CPU-GPU data transfer overheads.

Algorithm 1: GATO ($X_{init}, U_{init}, N, M, \mathcal{A}, \rightarrow X^*, U^*$)

1: **for** $b = 0 \ldots N * M$ **do in parallel blocks**
2: $S_b, \gamma_b, \Phi_b^{-1}$ via (6) with **parallel threads** (III-A)
3: **for** $b = 0 \ldots M$ **do in parallel blocks**
4: δZ^* via (4) with **parallel warps of threads** (III-B)
5: **for** $b = 0 \ldots N * M * \mathcal{A}$ **do in parallel blocks**
6: $\mathcal{M}_b$ via (3) with **parallel threads** (III-A)
7: **for** $b = 0 \ldots M$ **do**
8: α_b^* via (7) with **parallel threads** (III-A)
9: **return** X^*, U^*

B. Batched PCG

A key factor of GATO's overall performance is our batched linear system solver which is built around per-block PCG solves with finer-grained warp-level[1] parallel linear algebra. This hardware-optimized design not only improves computational throughput, but also improves memory access patterns over MPCGPU, reduces synchronizations, and increases overall hardware resource utilization both for a single solve and, most importantly, for batches of solves.

Each CUDA thread block is assigned to solve one linear system (6). Within a block, warps distribute work over knot points, and individual threads within a warp operate on rows/columns of the per-knot state/control blocks. This mapping eliminates inter-block coordination entirely: all vector updates, matrix-vector multiplications, and local reductions are resolved inside the block, avoiding the use of intra-block synchronization, e.g., the `cooperative groups` API used in [21]. This design improves both per-solve performance and portability across devices and launch contexts. This is because intra-block APIs require all blocks to be co-resident on the GPU, which constrains scalability and is particularly limiting on edge systems with restricted hardware resources.

All matrices/vectors are packed contiguously in row-major order by batch and knot points, exploiting the block tridiagonal structure of the S and Φ^{-1} matrices. This yields coalesced loads/stores for warp-strided accesses and makes warp shuffle intrinsics efficient for reductions. We also pad leading dimensions to multiples of the warp size to remove bounds checks and branch divergence. This enables us to implement a *warp-optimized* block tridiagonal matrix-vector multiplication routine that (i) uses shared memory tiles to stage the current and neighboring blocks, (ii) performs thread-parallel fused multiply–adds for the block-dense operations, (iii) pipelines loads to hide any memory latency (through the use of `cudaMemcpyAsync`), and (iv) avoids atomics or grid-wide barriers. The CUDA kernel's shared-memory footprint, block dimensions, and register usage are also tuned to maintain high occupancy while preventing register spilling for typical state/control sizes seen in MPC applications. As a result, each PCG solve proceeds efficiently and fully independently.

Finally, we partially unroll all inner loops over small, compile-time dimensions to reduce loop overhead, and compile with aggressive optimization flags (e.g., `-O3`, `-use_fast_math`). As shown in Section IV, these choices result in superior performance across our target batch sizes.

IV. RESULTS

In this section, we present a two-part evaluation of GATO. We first test our solver on a number of software benchmarks aimed to evaluate our approach against relevant baselines and explore the scalability of our design. We then demonstrate the usefulness of our solver through case studies of batched trajectory optimization for MPC applications. Our case studies are first demonstrated via simulation ablations. The final case study is also deployed onto a physical KUKA iiwa LBR14 manipulator. The source code accompanying this evaluation is released open source alongside our solver.

A. Methodology

Results were collected on a high-performance workstation with a 5.73GHz 24-core AMD Ryzen 9 7900X i9-12900K and a 2.2GHz NVIDIA GeForce RTX 4090 GPU running Ubuntu 22.04 and CUDA 12.6. Code was compiled with `g++11.4`, and time was measured using high-precision `timeit` package around `Python` wrappers for all CPU and GPU functions to provide realistic timing analysis for future users of our open source software.

Throughout our experiments, we compare our solver to ablations of itself, the state-of-the-art CPU QP solver OSQP [41] using the Pinocchio [42] dynamics library, and the state-of-the-art GPU solver MPCGPU [21] using the GRiD dynamics library [30]. We note that we also leverage the GRiD library in GATO as mentioned in Section III. All hyperparameter values can be found in our open source code. All solvers used the same cost functions, and solver-specific hyperparameter values were independently optimized.

We exclude `Jax`-based GPU solvers (e.g., [25], [26]) from our evaluations as both from their reported results in papers, and from our own evaluations on our computational hardware, they take tens of milliseconds to solve small batches of trajectory optimization problem: often an *order of magnitude slower* than our baselines. Similarly, OSQP's GPU backend is known to not be performant at our target problem sizes [21], [28] and is as such similarly excluded.

B. Scalability Benchmarks

We begin with a scalability study on a 6-DoF Neuromeka Indy7 manipulator executing a figure-8 tracking task. At each control step, we solve a batch of M trajectory-optimization problems with a fixed horizon of $N{=}64$, $h{=}0.01\,\text{s}$, warm-started with the previous control step's solution. Figure 3 (left) summarizes these results for $M = [1, 2, 4, \ldots, 128]$ comparing GATO against the aforementioned OSQP CPU baseline and MPCGPU GPU baseline. OSQP never matches the single-problem latency of either GPU method and, while its runtime scales reasonably with problem size, it is consistently the slowest. On the other hand, while MPCGPU is just $1.4\times$ slower than GATO for a single instance, since it is engineered to occupy the full GPU per solve, MPCGPU's latency grows nearly linearly with batch size, eventually falling behind GATO by a factor of $16\times$ for batch size $M = 128$. Overall, GATO achieves both lower single-solve latency and stronger scaling than baselines across our target range of batch-sizes. This yields an overall $18-21\times$ speedup over our CPU baseline and $1.4-16\times$ over our GPU baseline.

Figure 3 (right), shows a heat map of solve times for GATO while varying both batch size (M) and time horizon (N) for the same tracking problem. GATO is able to reach

[1]We note that a "warp" represents 32 contiguous threads on the same GPU-core. These threads work in lock-step due to the design of NVIDIA GPU hardware. By exploiting their native implicit synchronization at the hardware level, further acceleration of software can be achieved.

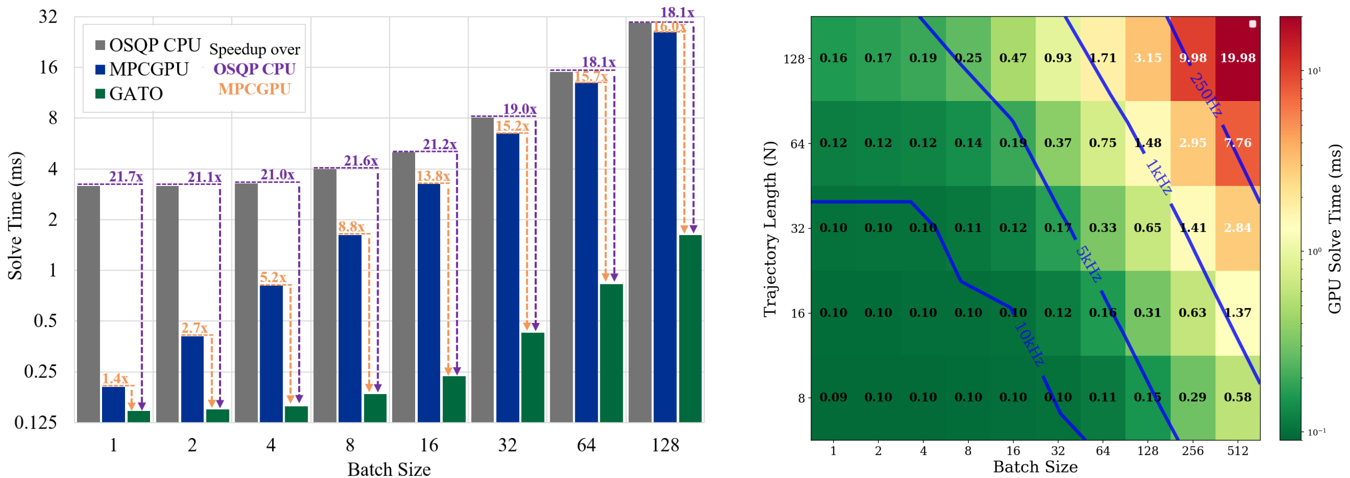


Fig. 3: (Left) Solve times for 6-DoF manipulator motions while varying the batch size (M) and underlying solver. $N = 64$ for all solves. GATO shows far improved scalability as compared to state-of-the-art CPU and GPU solutions. (Right) A heat map of solve times while varying both batch size (M) and time horizon (N). GATO is able to reach kHz control rates for real-time iterations of large batches (512) of short horizon ($N = 8$) trajectories, as well as smaller batches (32) of longer horizon trajectories ($N = 128$), showing the flexibility of the design.

kHz control rates for real-time iterations of large batches ($M = 512$) of short horizon ($N = 8$) trajectories, as well as smaller batches ($M = 32$) of longer horizon trajectories ($N = 128$), showing the flexibility of its design. We also find that this scalability is mostly related to the total number of knot points in the overall problem ($N * M$). For example, all points under 512 total points, e.g., $(N, M) = (64, 8), (8, 64), (16, 32)$, can run a real-time iteration in about 100μs (10kHz control rate) indicating that our solver efficiently utilizes GPU resources up to hardware limits, and then scales linearly for subsequent increases in problem size. We note that this surpasses the 512 point 1kHz-scaling shown in MPCGPU [21] and similar 1kHz max-scaling in other CPU-based state-of-the-art results [43].

In the next sections, we present three case studies that demonstrate the practical value of GATO's ability to solve batches of tens–to-hundreds of TO problems in real-time.

C. Case Study 1: Online Hyperparameter Optimization

Our first case study addresses hyperparameter selection in MPC, traditionally a time-consuming and sensitive process. We consider motion planning for the 7-DoF KUKA iiwa LBR14 with horizon $N = 64$ and timestep $h = 0.05$ s, running the solver for 100 SQP iterations from zero-initialized states and controls on 100 randomly sampled points within the robot's workspace.

Our batched solver is used to sample over ρ, a damping parameter often added to the diagonal of Q_k in (5) in deployed trajectory optimization solvers to improve numerical stability. We initialize the single-solve baseline with $\rho = 10^{-1}$. For batch size M, we initialize ρ by log-spacing values between 10^{-8} and 10^{1}. In both cases, ρ is adjusted after each SQP iteration based on the status of the line search, similar to the scheme in [44].

Figure 4 shows that larger batches consistently reduce

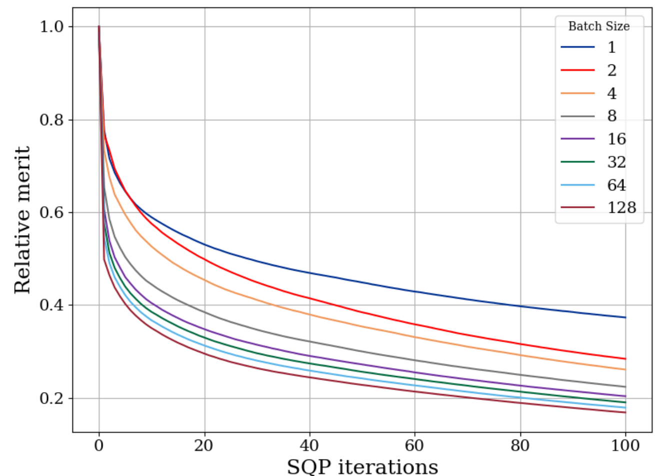


Fig. 4: Average (normalized) merit function value across SQP iterations over 100 runs each with 81 different random values for the cost function parameters Q and R in (5). For all solves $N = 64$, $h = 0.05$, ρ ranges from 10^{-8} to 10^{1}.

merit function values faster (indicating faster convergence to an optimal solution). Batches larger than 16 outperform the minimum merit achieved by the single-solve baseline after only 20 iterations, and $M = 32$ through 128 achieve nearly half the initial merit after only a single SQP iteration. However, we observe that these gains begin to saturate beyond $M \approx 32$ due to the limited range in ρ, reconfirming the importance of batch sizes of tens to low-hundreds. Overall, this improved convergence rate would enable a deployed solver to achieve comparable optimality at higher control rates, or increased optimality at a set nominal control rate.

D. Case Study 2: Fixed Disturbance Rejection

Our second case study explores disturbance rejection, a common problem in robotic control tasks. Here, a 6-DoF

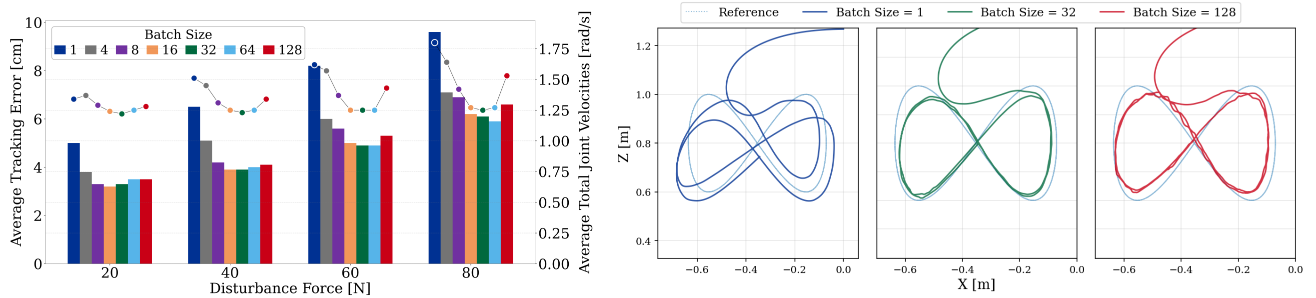


Fig. 5: Figure-8 tracking task, with an external disturbance applied at the end effector. (Left) Bar chart shows tracking error, scatter plot shows average total joint velocities. Increasing GATO's batch size enables increased disturbance reject, lowering tracking error and joint velocities until the increased latency from a larger batch size outweighs the optimality gains. (Right) End-effector trajectories realized during this experiment when 50N of external force is applied at the end effector, again showing that modest batch sizes lead to the best performance.

manipulator tracks a figure-8 end-effector trajectory like in IV-B, but now faces an unmodeled constant external force applied at the end effector in the $-Z$ direction.

Batched TO enables an "online hypothesize-and-test" strategy: evaluate multiple candidate disturbance models in parallel and apply the control from the most consistent one. At each control step, we solve a batch of M TO problems differing only in the assumed external force, f_j for $j \in [0, M)$. Candidate forces are generated by sampling directions uniformly on a sphere and adding them to the current estimated disturbance, exploring both direction and magnitude around the prior hypothesis. After solving this batch of problems, we use the optimized trajectory whose dynamics model best matches the measured evolution of the robot's state after one control step. We then update our disturbance estimate for our next solve by re-centering it around the selected f_j.

As shown in Figure 5, this simple sampling approach proves effective, consistent with batched roll-outs as noted in [45] and batched contact estimates as noted in [46]. In particular, Figure 5 (left) shows that tracking error and joint velocities decrease with increasing M until reaching a sweet spot at around $M = 32$. Beyond this point, increased solve times offset the benefit of finer hypothesis granularity and increase closed-loop error. Figure 5 (right) illustrates end-effector trajectories for a representative 50 N disturbance, where $M = 32$ tracks the figure-8 substantially better than a single-solve baseline, while very large batches, e.g., the $M = 128$ shown, lose effectiveness due to higher latency.

E. Case Study 3: Planning Under Uncertainty

In our final case study, we consider a 7-DoF KUKA iiwa LBR14 executing a multi-point pick–and–place task with an *unmodeled* suspended load attached to the end effector (see Figure 6). The swinging payload induces time-varying, direction-dependent forces that degrade controller performance. To account for this, at each control step, and as done in Section IV-D, GATO warm-starts from the previous solution and solves a batch of trajectory-optimization problems in parallel, each conditioned on a different disturbance hypothesis. Controls are then selected or blended according to consistency with the observed motion, and the hypothesis

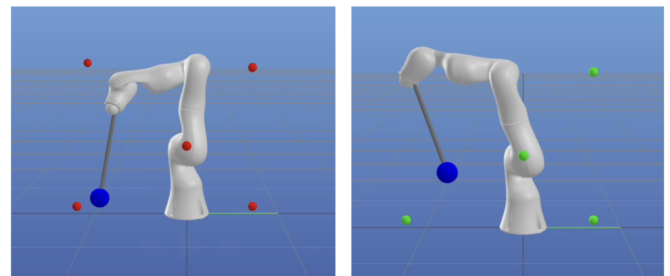


Fig. 6: Simulation visualization at the last timestep of the pick-and-place task in Section IV-E, with a 15kg pendulum attached to the last joint. Batch size $M = 1$ on the left, and $M = 32$ on right.

set is re-centered for the next step. Throughout, the task enforces tight accuracy requirements with success requiring the end effector to reach within 5 cm of each goal in under 5 seconds. We also require the sum of joint-velocities to be under 1.0 rad/s. If time is exceeded, the target is considered a failure and we move onto the next target. This experiment demonstrates GATO's robustness and shows why our target batch sizes are practical: they offer sufficient disturbance coverage without sacrificing real-time performance.

1) Simulation Studies: In simulation the solver uses a horizon of $N{=}16$ with a timestep $h = 0.01$ s, and is limited to 5 SQP iterations with a PCG tolerance of 10^{-6}. We simulate the plant at 1 kHz (RK4 with $h{=}0.001$ s). We use a constant 15kg mass and run 100 scenarios varying pendulum length $\ell \in [0.3, 0.7]$ m, initial angle $\|\theta\| \in [0, 0.6]$ rad, and damping constant $b \in [0.1, 0.6]$ Nms/rad.

Table I summarizes the solver's performance and Figure 7 shows the distribution of solve times for GATO across different batch sizes. We can see that the success rate dramatically increases and the task-completion time falls significantly as M grows from 1 to 8. Following that, performance continues to increase albeit at a slower rate. Ultimately, at our largest batch size of $M = 128$ (shown in red), we are able to not only achieve a 99.2% success rate but also solve almost all problems faster than any other solver, with $M = 64$ and 32 (shown in light blue and green) not far behind. Figure 6 provides a visualization of the simulated experiments, with

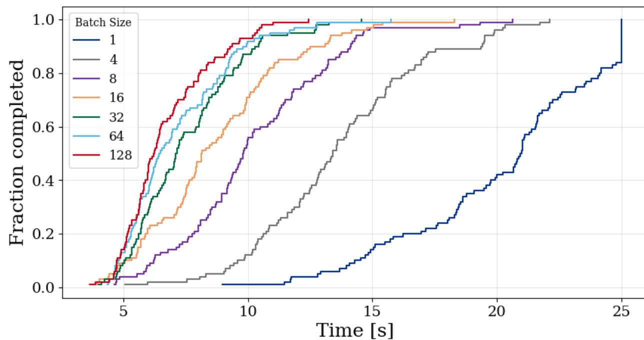


Fig. 7: Cumulative density function of the solve times for a trajectory length N=16 for GATO across different batch sizes and varied pendulum configurations. For each batch size, the solver accounts for 100 disturbance scenarios. We see how larger batch sizes enable more accurate unmodeled disturbance rejection.

TABLE I: Episode performance vs batch size, collected by varying pendulum configuration (length, angle, damping).

Batch size	Success rate [%]	Mean time [s]
1	33.0	20.1
4	78.2	13.7
8	91.8	10.2
16	95.0	8.7
32	96.4	7.5
64	97.6	7.1
128	99.2	6.7

the red and green spheres denoting unreached and reached targets respectively for $M = 1$ (left) and $M = 32$ (right).

2) Hardware Deployment: Finally, we run two variants of the simulation experiments from IV-E on a physical KUKA iiwa LBR14 robot: (a) five-target goal reaching with no load at 100Hz, and (b) three-target goal reaching with an unmodeled 4kg load at 1000Hz. Both used horizon N=32, timestep $h = 0.02\,\text{s}$, one SQP iteration, and PCG tolerance of 10^{-6}. Our goal is to show real-world effectiveness of our GPU-accelerated approach, handling not only unmodeled forces but also control loop delays, system identification errors, and noisy sensor measurements. We compare results from a single solve against a batch size of $M = 32$.

As shown in Figure 8, Table II, and our supplementary videos, the batched solver outperforms single solves, reaching targets in less time and successfully rejecting model errors, sensor noise, and the time-varying external disturbance.

V. Conclusion and Future Work

In this work, we introduce GATO, an open source, GPU-accelerated, batched TO solver that is co-designed across algorithm, software, and computational hardware to deliver real-time throughput for batches of tens to low-hundreds of solves. GATO achieves its performance through co-designed parallelism at the block-, warp-, and thread-level, taking full advantage of the GPU computational model. Our experiments demonstrate not only superior performance, providing

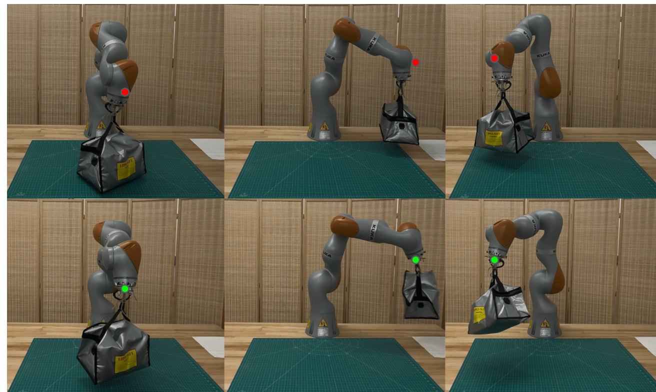


Fig. 8: Hardware experiment (b) showing the solver successfully ($M = 32$) and unsuccessfully ($M = 1$) account for the time-varying unmodeled disturbance and reach the targets.

TABLE II: Performance metrics for the hardware pick-and-place tasks showing the improved performance resulting from the use of our batched solver.

Experiment	***(a)***		***(b)***	
Batch Size	1	32	1	32
Completed	5/5	5/5	0/3	3/3
Time [s]	16.93	9.49	24.00	6.71

speedups of as much as 18–21× over CPU and 1.4–16× over GPU baselines as batch size increases, but also that such moderate batch sizes are useful for deployed applications, improving convergence, and rejecting disturbances both in simulation and on a physical manipulator.

There are many promising directions for future work, including: integration with actor-critic reinforcement learning to guide agent exploration [47], use of branch-and-bound-based methods for contact-implicit trajectory optimization [48], and evaluation of our approach on mobile robots at the edge using low-power GPU platforms such as the NVIDIA Jetson [49].

VI. Acknowledgments

We thank Ludovic Righetti and the Machines in Motion Laboratory for their support with hardware demonstrations.

References

[1] F. R. Hogan, E. R. Grau, and A. Rodriguez, "Reactive planar manipulation with convex hybrid mpc," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 247–253.

[2] 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.

[3] M. Tranzatto, T. Miki, M. Dharmadhikari, L. Bernreiter, M. Kulkarni, F. Mascarich, O. Andersson, S. Khattak, M. Hutter, R. Siegwart, *et al.*, "Cerberus in the darpa subterranean challenge," *Science Robotics*, vol. 7, no. 66, p. eabp9742, 2022.

[4] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. Del Prete, "Optimization-based control for dynamic legged robots," *IEEE Transactions on Robotics*, 2023.

[5] S. Kuindersma, "Taskable agility: Making useful dynamic behavior easier to create," Princeton Robotics Seminar, April 2023.

[6] J. T. Betts, *Practical methods for optimal control and estimation using nonlinear programming*. SIAM, 2010.

[7] 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 the atlas humanoid robot," *Autonomous robots*, vol. 40, pp. 429–455, 2016.

[8] H. Li and P. M. Wensing, "Cafe-mpc: A cascaded-fidelity model predictive control framework with tuning-free whole-body control," *arXiv preprint arXiv:2403.03995*, 2024.

[9] K. Nguyen, S. Schoedel, A. Alavilli, B. Plancher, and Z. Manchester, "Tinympc: Model-predictive control on resource-constrained microcontrollers," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 1–7.

[10] 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.

[11] B. Vlahov, J. Gibson, M. Gandhi, and E. A. Theodorou, "Mppi-generic: A cuda library for stochastic trajectory optimization," *arXiv preprint arXiv:2409.07563*, 2024.

[12] H. Xue, C. Pan, Z. Yi, G. Qu, and G. Shi, "Full-order sampling-based mpc for torque-level locomotion control via diffusion-style annealing," *arXiv preprint arXiv:2409.15610*, 2024.

[13] J. Alvarez-Padilla, J. Z. Zhang, S. Kwok, J. M. Dolan, and Z. Manchester, "Real-time whole-body control of legged robots with model-predictive path integral control," *arXiv preprint arXiv:2409.10469*, 2024.

[14] R. Enrico, M. Mancini, and E. Capello, "Comparison of nmpc and gpu-parallelized mppi for real-time uav control on embedded hardware," *Applied Sciences*, vol. 15, no. 16, p. 9114, 2025.

[15] B. Plancher and S. Kuindersma, "A performance analysis of parallel differential dynamic programming on a gpu," in *Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics*. Springer, 2018, pp. 656–672.

[16] 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*, 2019, pp. 1–12.

[17] Y. Lee, M. Cho, and K.-S. Kim, "Gpu-parallelized iterative lqr with input constraints for fast collision avoidance of autonomous vehicles," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 4797–4804.

[18] D. Cole, S. Shin, F. Pacaud, V. M. Zavala, and M. Anitescu, "Exploiting gpu/simd architectures for solving linear-quadratic mpc problems," in *2023 American Control Conference (ACC)*. IEEE, 2023, pp. 3995–4000.

[19] S. Shin, F. Pacaud, and M. Anitescu, "Accelerating optimal power flow with gpus: Simd abstraction of nonlinear programs and condensed-space interior-point methods," *arXiv preprint arXiv:2307.16830*, 2023.

[20] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. Van Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, *et al.*, "Curobo: Parallelized collision-free robot motion generation," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 8112–8119.

[21] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 9787–9794.

[22] Y. Lee, K. H. Choi, and K.-S. Kim, "Gpu-enabled parallel trajectory optimization framework for safe motion planning of autonomous vehicles," *IEEE Robotics and Automation Letters*, 2024.

[23] S. H. Jeon, S. Hong, H. J. Lee, C. Khazoom, and S. Kim, "Cusadi: A gpu parallelization framework for symbolic expressions and optimal control," *IEEE Robotics and Automation Letters*, 2024.

[24] A. L. Bishop, J. Z. Zhang, S. Gurumurthy, K. Tracy, and Z. Manchester, "Relu-qp: A gpu-accelerated quadratic programming solver for model-predictive control," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 13 285–13 292.

[25] K. Tracy and Z. Manchester, "On the differentiability of the primal-dual interior-point method," *arXiv preprint arXiv:2406.11749*, 2024.

[26] L. Amatucci, J. Sousa-Pinto, G. Turrisi, D. Orban, V. Barasuol, and C. Semini, "Primal-dual ilqr for gpu-accelerated learning and control in legged robots," *arXiv preprint arXiv:2506.07823*, 2025.

[27] M. Naumov, "Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas," *Nvidia white paper*, vol. 3, 2011.

[28] 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.

[29] 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.

[30] B. Plancher, S. M. Neuman, R. Ghosal, S. Kuindersma, and V. J. Reddi, "Grid: Gpu-accelerated rigid body dynamics with analytical gradients," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 6253–6260.

[31] F. Pacaud, S. Shin, M. Schanen, D. A. Maldonado, and M. Anitescu, "Accelerating condensed interior-point methods on simd/gpu architectures," *Journal of Optimization Theory and Applications*, pp. 1–20, 2023.

[32] M. Hamer, L. Widmer, and R. D'andrea, "Fast generation of collision-free trajectories for robot swarms using gpu acceleration," *IEEE Access*, vol. 7, pp. 6679–6690, 2018.

[33] D. Guhathakurta, F. Rastgar, M. A. Sharma, K. M. Krishna, and A. K. Singh, "Fast joint multi-robot trajectory optimization by gpu accelerated batch solution of distributed sub-problems," *Frontiers in Robotics and AI*, vol. 9, p. 890385, 2022.

[34] F. Rastgar, H. Masnavi, K. Kruusamäe, A. Aabloo, and A. K. Singh, "Gpu accelerated batch trajectory optimization for autonomous navigation," in *2023 American Control Conference (ACC)*. IEEE, 2023, pp. 718–725.

[35] I. Tsikelis and K. Chatzilygeroudis, "Gait optimization for legged systems through mixed distribution cross-entropy optimization," in *2024 IEEE-RAS 23rd International Conference on Humanoid Robots (Humanoids)*. IEEE, 2024, pp. 1011–1018.

[36] T. Lew, M. Greiff, F. Djeumou, M. Suminaka, M. Thompson, and J. Subosits, "Risk-averse model predictive control for racing in adverse conditions," *arXiv preprint arXiv:2410.17183*, 2024.

[37] J. Nocedal and S. J. Wright, *Numerical optimization*. Springer, 1999.

[38] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, pp. 25–57, 2006.

[39] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM review*, vol. 47, no. 1, pp. 99–131, 2005.

[40] X. Bu and B. Plancher, "Symmetric stair preconditioning of linear systems for parallel trajectory optimization," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 9779–9786.

[41] 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.

[42] 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)*. IEEE, 2019, pp. 614–619.

[43] 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.

[44] M. K. Transtrum and J. P. Sethna, "Improvements to the levenberg-marquardt algorithm for nonlinear least-squares minimization," 2012. [Online]. Available: https://arxiv.org/abs/1201.5885

[45] T. Howell, N. Gileadi, S. Tunyasuvunakool, K. Zakka, T. Erez, and Y. Tassa, "Predictive sampling: Real-time behaviour synthesis with mujoco," *arXiv preprint arXiv:2212.00541*, 2022.

[46] H. J. T. Suh, T. Pang, and R. Tedrake, "Bundled gradients through contact via randomized smoothing," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 4000–4007, 2022.

[47] E. Alboni, G. Grandesso, G. P. R. Papini, J. Carpentier, and A. Del Prete, "Cacto-sl: Using sobolev learning to improve continuous actor-critic with trajectory optimization," in *6th Annual Learning for Dynamics & Control Conference*. PMLR, 2024, pp. 1452–1463.

[48] T. Marcucci and R. Tedrake, "Warm start of mixed-integer programs for model predictive control of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 66, no. 6, pp. 2433–2448, 2020.

[49] M. Ditty, "Nvidia orin system-on-chip," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 2022, pp. 1–17.