MPCGPU: Real-Time Nonlinear Model Predictive Control through Preconditioned Conjugate Gradient on the GPU

Emre Adabag[1], Miloni Atal[1*], William Gerard[1*], Brian Plancher[2]

***Abstract*— Nonlinear Model Predictive Control (NMPC) is a state-of-the-art approach for locomotion and manipulation which leverages trajectory optimization at each control step. While the performance of this approach is computationally bounded, implementations of direct trajectory optimization that use iterative methods to solve the underlying moderately-large and sparse linear systems, are a natural fit for parallel hardware acceleration. In this work, we introduce MPCGPU, a GPU-accelerated, real-time NMPC solver that leverages an accelerated preconditioned conjugate gradient (PCG) linear system solver at its core. We show that MPCGPU increases the scalability and real-time performance of NMPC, solving larger problems, at faster rates. In particular, for tracking tasks using the Kuka IIWA manipulator, MPCGPU is able to scale to kilohertz control rates with trajectories as long as 512 knot points. This is driven by a custom PCG solver which outperforms state-of-the-art, CPU-based, linear system solvers by at least 10x for a majority of solves and 3.6x on average.**

I. Introduction

Nonlinear Model Predictive Control (NMPC) is a feedback control strategy which repeatedly solves finite horizon optimal control problems (OCP) in real time, enabling robots to adapt to changes in their environment. This approach has seen great recent success in applications to both locomotion and manipulation [1], [2], [3], [4], [5].

Most implementations of NMPC leverage trajectory optimization [6] to solve the underlying optimal control problems. Two popular classes of these algorithms are shooting methods and direct methods. Shooting methods parameterize only the input trajectory and use Bellman's optimality principle [7] to iteratively solve a sequence of smaller optimization problems [8], [9]. Direct methods explicitly represent the states, controls, dynamics, and any additional constraints, leading to moderately-large nonlinear programs with structured sparsity patterns [10].

There has been historical interest in parallel strategies [11] for solving trajectory optimization problems. This is growing increasingly important with the impending end of Moore's Law and the end of Dennard Scaling, which have led to a utilization wall that limits the performance a single CPU chip can deliver [12], [13]. Several more recent efforts have shown that significant computational benefits are possible by exploiting the natural parallelism in the computation of the (gradients of the) dynamics and cost functions on GPUs and FPGAs [14], [15], [16], [17], [18], [19], [20], [21]. However, multiple-shooting and consensus approaches to computing trajectory updates at each algorithmic iteration [22], [23], [24], [25] have only seen modest gains when implemented on alternative hardware platforms [26], [27].

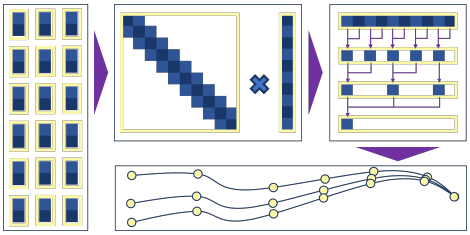

Fig. 1: At each control step, MPCGPU uses a parallel construction of the Schur complement of the Karush-Kuhn-Tucker (KKT) system, a parallel-friendly PCG solver, and a parallel line search to provide real-time performance through GPU acceleration.

On the other hand, direct methods naturally expose more parallelism that can be exploited through hardware acceleration. Importantly, these approaches are computationally dominated by the solutions of a moderately-large and sparse linear systems. Iterative methods, like the Preconditioned Conjugate Gradient (PCG) algorithm [28], are particularly well-suited for parallel solutions of linear systems, as they are computationally dominated by matrix-vector products and vector reductions [29], [30], and have shown past success in outperforming state-of-the-art CPU implementations for solving very-large linear systems on GPUs [31], [32].

In this work, we introduce MPCGPU, a GPU-accelerated, real-time NMPC solver that exploits the structured sparsity and the natural parallelism in direct trajectory optimization (see Figure 1). At our solver's core is a custom, accelerated implementation of PCG tuned for the Schur complement of the KKT systems of trajectory optimization problems.

We show that MPCGPU increases the scalability and real-time performance of NMPC, solving larger problems, at faster rates. In particular, for tracking tasks using the Kuka IIWA manipulator, MPCGPU is able to scale to kilohertz rates with trajectories as long as 512 knot points. This is driven by a custom, GPU-accelerated, PCG solver which outperforms state-of-the-art, CPU-based, linear system solvers by at least 10x for a majority of solves and 3.6x on average. We release our software and experiments open source at: https://github.com/a2r-lab/MPCGPU.

This material is based upon work supported by the National Science Foundation (under Award 2246022). 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]Emre Adabag, Miloni Atal, and William Gerard are with the School of Engineering and Applied Science, Columbia Unviersity, New York, NY. {ea2944, ma4338, wg2404}@columbia.edu

[2]Brian Plancher is with Barnard College, Columbia University, New York, NY. bplancher@barnard.edu

*These authors contributed equally to this work.

II. Related Work

There has been a significant amount of prior work developing general purpose sparse linear system solvers on the GPU both using factorization-based approaches [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], as well as iterative methods [29], [32], [43], [44], [45], [46], [47], [48]. There has also been work developing and implementing Block-Cyclic-Reduction and other tree-structured methods that are optimized for block-tridiagonal systems [49], [50], [51], [52]. These general purpose approaches have found speedups through GPU usage, but only once the problem size grows to more than tens if not hundreds of thousands of variables or for instances of (linear) (power-flow) problems [32], [39], [40], [41], [42], [53]. As such, these general purpose solvers are not performant for most trajectory optimization problems (e.g., our examples in Section V have 448 to 7,168 variables).

For the nonlinear trajectory optimization problem, evolutionary, particle-swarm, Monte-Carlo, and other sampling based approaches have been implemented on GPUs [54], [55], [56], [57], [58], [59], [60], [61], [62]. Most prior work on gradient-based parallel nonlinear trajectory optimization has been fully confined to the CPU [63], [24], [22], [23], [25], [64], relied on the CPU for many of the computations [65], [66], focused only on the problem of optimizing BLAS functions on the GPU [67], [68], or was limited to GPU acceleration of the naturally parallel (gradients of the) dynamics and cost functions [14], [16], [18], [19].

There are two existing lines of work which fully implemented gradient-based nonlinear trajectory optimization on the GPU. The first leveraged shooting based methods and found them to not expose much natural parallelism, limiting their performance [26], [27]. The second used a Block-Cyclic-Reduction-based direct method to exploit the particular structure exposed by position-based dynamics [15].

This work adds to the literature by designing a GPU-accelerated, gradient-based, direct trajectory optimization solver for standard reduced-coordinate dynamics [69] leveraging a custom parallel PCG solver at its core.

III. Background

A. Direct Trajectory Optimization

Trajectory optimization [6], also known as numerical optimal control, 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,

$$x_{k+1} = f(x_k, u_k, h), \quad x_0 = x_s, \tag{1}$$

with a timestep h, and minimize an additive cost function,

$$J(X, U) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k). \tag{2}$$

Direct methods for trajectory optimization form a moderately-large and sparse nonlinear program. While there are a variety of algorithmic approaches used to solve these problems, most methods can be reduced to a three step process which is repeated until convergence [10], [70], [71].

Step 1: Compute a second-order Taylor expansion of our problem along a nominal trajectory, resulting in the following quadratic program (QP) for the deviation (δX, δU):

$$\begin{aligned} &\min_{\delta X, \delta U} \tfrac{1}{2}\delta x_N^T Q_N \delta x_N + q_N^T \delta x_N + \\ &\sum_{k=0}^{N-1} \tfrac{1}{2}\delta x_k^T Q \delta x_k + q^T \delta x_k + \tfrac{1}{2}\delta u_k^T R \delta u_k + r^T \delta u_k \\ &\text{s.t.} \quad \delta x_0 = x_s - x_0, \\ &\quad \delta x_{k+1} - A_k \delta x_k - B_k \delta u_k = 0 \quad \forall k \in \mathbb{Z} \cap [0, N) \end{aligned} \tag{3}$$

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

$$\begin{bmatrix} G & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} -\delta Z \\ \lambda \end{bmatrix} = \begin{bmatrix} g \\ c \end{bmatrix} \tag{4}$$

where:

$$\begin{aligned} \delta z_k &= \begin{bmatrix} \delta x_k & \delta u_k \end{bmatrix}^T \qquad \delta z_N = \delta x_N \\ G &= \begin{bmatrix} Q_0 & & & \\ & R_0 & & \\ & & \ddots & \\ & & & Q_N \end{bmatrix} \\ g &= \begin{bmatrix} q_0 & r_0 & q_1 & r_1 & \dots & q_N \end{bmatrix}^T \\ C &= \begin{bmatrix} I & & & & & \\ -A_0 & -B_0 & I & & & \\ & & \ddots & -A_{N-1} & -B_{N-1} & I \end{bmatrix} \\ e_k &= x_{k+1} - f(x_k, u_k) \\ c &= \begin{bmatrix} x_s - x_0 & e_0 & e_1 & \dots & e_{N-1} \end{bmatrix}^T \end{aligned}$$

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 trust-region or line-search [10].

B. The Schur Complement Method

One approach to solving Equation 4 is through a two step process which forms the symmetric positive definite *Schur Complement*, S, and first solves for λ^* and then δz^*:

$$\begin{aligned} S &= -CG^{-1}C^T & \gamma &= c - CG^{-1}g \\ S\lambda^* &= \gamma & \delta z^* &= G^{-1}(g - C^T\lambda^*). \end{aligned} \tag{5}$$

By defining the variables θ, ϕ, and ζ,

$$\begin{aligned} \theta_k &= A_k Q_k^{-1} A_k^T + B_k R_k^{-1} B_k^T + Q_{k+1}^{-1} \\ \phi_k &= -A_k Q_k^{-1} \\ \zeta_k &= -A_k Q_k^{-1} q_k - B_k R_k^{-1} r_k + Q_{k+1}^{-1} q_{k+1}, \end{aligned} \tag{6}$$

S and γ take the form:

$$\begin{aligned} S &= \begin{bmatrix} Q_0^{-1} & \phi_0^T & & & & \\ \phi_0 & \theta_0 & \phi_1^T & & & \\ & & \ddots & \phi_{N-2} & \theta_{N-2} & \phi_{N-1}^T \\ & & & & \phi_{N-1} & \theta_{N-1} \end{bmatrix} \\ \gamma &= c + \begin{bmatrix} Q_0^{-1} q_0 & \zeta_0 & \zeta_1 & \dots & \zeta_{N-1} \end{bmatrix}^T \end{aligned} \tag{7}$$

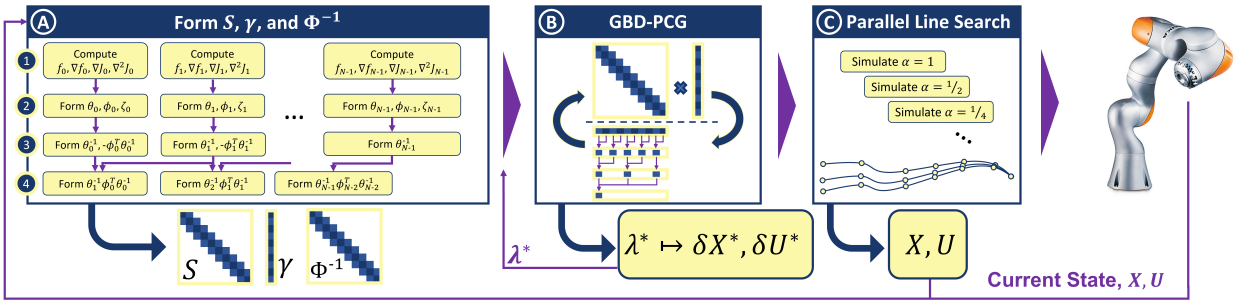

Fig. 2: A high level overview of MPCGPU which: 1) in parallel on the GPU computes S, γ, and Φ^{-1} and stores those values in an optimized dense format, 2) uses our GBD-PCG solver to compute λ^* and reconstructs $\delta X^*, \delta U^*$ through GPU-friendly matrix-vector multiplications and vector reductions, and 3) leverages a parallel line search to compute the final trajectory, X, U. This trajectory is then passed to the (simulated) robot and the current state of the (simulated) robot is measured and fed back into our solver which is run again, warm-started with our last solution.

C. 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 some tolerance ϵ. The most popular of these methods is the conjugate gradient (CG) algorithm which has been used for state-of-the-art results on large-scale optimziation problems on the GPU [53], [72]. 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 (see Algorithm 1).

D. Graphics Processing Units (GPUs)

Compared to a multi-core CPU, a GPU is a larger set of simpler processors, optimized for parallel execution of identical instructions. GPUs are best at computing regular and separable computations, over large data sets, with limited synchronization (e.g., large matrix multiplication) [73]. Our work uses NVIDIA's CUDA extensions to C++ [74].

IV. The MPCGPU Solver

In this section we describe the design of the MPCGPU solver which exploits the sparsity and natural parallelism found in direct trajectory optimization algorithms and iterative linear system solvers. To further promote efficient GPU acceleration, unlike generic approaches, which require a kernel launch and CPU-GPU synchronization for each matrix operation, MPCGPU uses only three kernels that are asynchronously queued, resulting in only a single CPU-GPU synchronization. We also only transfer the initial and final values between the CPU and GPU to reduce I/O overheads.

As shown in Figure 2, our approach can be broken down into a three step process. At each control step we first compute each block row of the Schur complement system, S and γ, as well as our preconditioner, Φ^{-1}, in parallel by taking advantage of the structured sparsity of those matrices. Next, we use our custom GPU-optimized, warm-started, PCG solver, GBD-PCG (Algorithm 2), to compute the optimal Lagrange multipliers, λ^*, and reconstruct the optimal trajectory update, $\delta X^*, \delta U^*$. Finally we leverage a parallel line search to compute the final trajectory X, U which we send to the (simulated) robot for execution and simultaneously measure the current state of the (simulated) robot to begin our next control step. In the remainder of this section we provide further details on our approach. Our open-source implementation can be found at: https://github.com/a2r-lab/MPCGPU.

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

1: $r = \gamma - S\lambda$
2: $\tilde{r}, p = \Phi^{-1}r$
3: $\eta = r^T\tilde{r}$
} **Initialization**

4: **for** iter $i = 1$: max_iter **do**
5: $\quad \alpha = \eta/(p^T Sp)$
6: $\quad r = r - \alpha Sp$
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** return λ
11: $\quad \beta = \eta'/\eta$
12: $\quad p = \tilde{r} + \beta p$
13: $\quad \eta = \eta'$
} **Main Loop**

14: return λ

A. Parallel Computation of S, γ, and Φ^{-1}

To efficiently compute S, γ, and Φ^{-1} on the GPU, we need to find a naturally parallel approach to form the values as well as an efficient data storage format that minimizes overheads. We also need to find an effective preconditioner that is parallel-friendly in its computation.

We first leverage the block-tridiagonal structure of the Schur complement, S, as shown in Equation 7, which is for the most part independent across timesteps, k, for each block-row. This pattern also extends to each block-row of the

γ vector. To further remove the need for synchronizations, for each k, we also compute the only cross-timestep quantities, Q_{k+1} and q_{k+1}. While this results in those terms being computed twice, it still proves to be more efficient than forcing a synchronization point between all block-rows. To ensure efficient computation of the underlying dynamics and kinematic quantities, we leverage the GRiD library, which was shown to outperform state-of-the-art CPU libraries even when taking into account I/O overheads [18].

We further parallelize across and within the many small matrix inversions and matrix multiplications within each parallel block-row computation. Leveraging best practices [16], [18], we also group together the various types of mathematical operations, storing intermediate values in shared memory, and re-ordering computations where needed.

We leverage the Symmetric Stair Preconditioner [30], [75] which is a parallel-friendly preconditioner optimized for block-tridiagonal systems wich has an analytical inverse and results in the block-tridiagonal matrix:

$$\Phi^{-1} = \begin{bmatrix} Q_0 & -Q_0\phi_0^T\theta_0^{-1} & \\ -\theta_0^{-1}\phi_0 Q_0^{-1} & \theta_0^{-1} & -\theta_0^{-1}\phi_1^T\theta_1^{-1} \\ & -\theta_1^{-1}\phi_1\theta_0^{-1} & \theta_1^{-1} \\ & & \ddots \end{bmatrix} \quad (8)$$

The structure of Φ^{-1} also permits mostly parallel computation as only the values of each θ_k^{-1} need to be shared across timesteps. Our approach thus requires only a single global synchronization across blocks, and allows us to store the block-tridiagonal S and Φ^{-1} matrices in a custom, compressed, dense format for increased IO bandwidth and memory efficiency.

B. GPU Parallel PCG for Block-Tridiagonal Systems

The core of our solver is a custom GPU Parallel PCG implementation specifically optimized for block-tridiagonal systems, GBD-PCG (Algorithm 2). That is, we leverage the sparsity structure of S and Φ^{-1} to maximize cache usage and natural parallelism resulting in a refactored, low-latency implementation with minimal synchronizations. These optimizations can be leveraged in the most computationally expensive part of the algorithm, the large matrix-vector products in lines 5, 6, and 8 of Algorithm 1, as each element of the product depends on at most $3n_b$ elements from the matrix and $3n_b$ elements from the vector, where n_b is the block dimension. We exploit this by grouping threads that access similar elements into thread blocks and storing S, Φ^{-1}, and all PCG iterates concurrently in shared (cache) memory on the GPU. We also operate as many steps of the algorithm fully in parallel as possible between the thread synchronizations needed for the parallel reductions of scalar values on lines 6, 12, and 21 of Algorithm 2. Similarly, we only use device memory (RAM) for those scalar reductions and for the values of p and r that need to be shared between blocks on lines 9 and 18 of Algorithm 2. This means that the choice of a sparse preconditioner not only enables its efficient computation and memory storage, but also reduces

Algorithm 2: GPU Parallel PCG for Block-Tridiagonal Systems (GBD-PCG) $(S, \Phi^{-1}, \gamma, \lambda, \epsilon) \rightarrow \lambda^*$

1: **for** block $b = 0 : N$ **in parallel do** — **Initialization**
2: $\quad r_b = \gamma_b - S_b\lambda_{b-1:b+1}$
3: $\quad$ Load r_{b-1}, r_{b+1}
4: $\quad \tilde{r}_b, p_b = \Phi_b^{-1} r_{b-1:b+1}$
5: $\quad \eta_b = r_b^T \tilde{r}_b$
6: $\eta =$ **ParallelReduce**(η_b)
7: **for** iter $i = 1 :$ max_iter **do** — **Main Loop**
8: $\quad$ **for** block $b = 0 : N$ **in parallel do**
9: $\quad\quad$ Load p_{b-1}, p_{b+1}
10: $\quad\quad \Upsilon_b = S_b p_{b-1:b+1}$
11: $\quad\quad \upsilon_b = p_b \Upsilon_b$
12: $\quad \upsilon =$ **ParallelReduce**(υ_b)
13: $\quad$ **for** block $b = 0 : N$ **in parallel do**
14: $\quad\quad \alpha = \eta / \upsilon$
15: $\quad\quad \lambda_b = \lambda_b + \alpha p_b$
16: $\quad\quad r_b = r_b - \alpha \Upsilon_b$
17: $\quad$ **for** block $b = 0 : N$ **in parallel do**
18: $\quad\quad$ Load r_{b-1}, r_{b+1}
19: $\quad\quad \tilde{r}_b = \Phi_b^{-1} r_{b-1:b+1}$
20: $\quad\quad \eta'_b = r_b^T \tilde{r}_b$
21: $\quad \eta' =$ **ParallelReduce**(η'_b)
22: $\quad$ **if** $\eta' < \epsilon$ **then** return λ
23: $\quad$ **for** block $b = 0 : N$ **in parallel do**
24: $\quad\quad \beta = \eta' / \eta$
25: $\quad\quad p_b = \tilde{r}'_b + \beta p_b$
26: $\quad\quad \eta = \eta'$
27: return λ

the number of synchronizations and amount of memory that needs to be shared through RAM during each PCG iterate. This holistic co-design across algorithm stages is part of the reason why MPCGPU is so performant. Finally, we warm-start the values for λ based on the previous solve which we found greatly increased overall performance by reducing the number of PCG iterations needed for convergence.

C. Parallel Line Search

We leverage a parallel line search, computing all possible iterates for $\alpha \in \mathbb{A}$ in parallel[1] and selecting the iterate with the best value according to its L1 merit function [10]. This allows MPCGPU to evaluate all possible line search iterates in the same amount of time as it would take to compute a single iterate under a standard backtracking approach. Importantly, this not only reduces latency of this step, but has also been shown to improve the convergence of NMPC on similar whole-body trajectory tracking problems [26].

[1]In this work we use $\mathbb{A} = \{1, \frac{1}{2}, \cdots, \frac{1}{256}\}$, but any decaying set of fractional values can be used in practice.

V. Results

In this section we present a two-part evaluation of MPCGPU through a case study of online, dynamic, multi-goal, end-effector position tracking using whole-body NMPC for a simulated Kuka IIWA manipulator. First, we compare the performance of our underlying GBD-PCG iterative linear system solver with the state-of-the-art, CPU-based, QDLDL solver [76]. Second, we show how the end-to-end performance enabled by MPCGPU allows us to scale to long time horizons and fast control rates. Source code accompanying this evaluation can be found at https://github.com/a2r-lab/MPCGPU.

A. Methodology

Results were collected on high-performance workstation with a 3.2GHz 16-core Intel i9-12900K and a 2.2GHz NVIDIA GeForce RTX 4090 GPU running Ubuntu 22.04 and CUDA 12.1. Code was compiled with `g++11.4`, and time was measured with the Linux system call `clock_gettime()`, using `CLOCK_MONOTONIC` as the source. Our performance analysis is drawn from 100 NMPC trials of a 10 second, 5 goal, pick-and-place circuit for a simulated Kuka IIWA-14 (see Figure 3). Each NMPC trial consists of thousands of linear system solves which are needed for the many iterations of the underlying trajectory optimization problem for end-effector position tracking solved at each control step. All hyperparameter values can be found in our open-source source code and resulted in an average tracking error of $\sim$10cm, providing similar performance as previous experiments with GPU accelerated NMPC [27]. In particular, we note that all solvers used the same quadratic cost functions for each amount of knot points, and solver-specific hyperparameter values were independently tuned for maximal performance.

B. Linear System Solver Performance

We evaluate the performance of our underlying GBD-PCG linear system solver over its thousands of solves during each of our 100 NMPC trials running at a 500hz control rate and compare its performance to the state-of-the-art CPU-based QDLDL solver [76] operating in the same context.

Average Solve Time: Our results show that our GPU based solver outperforms QDLDL across most problem sizes and is only marginally slower at the smallest problem size, obtaining as much as a 3.6x average speedup (see Figure 4). This is driven by the GPU's ability to leverage large scale parallelism to gracefully scale to larger problem sizes. We note that the speedup plateaus at 256 knot points as we begin to run out of hardware resources on our specific GPU. These results show that unlike generic approaches that become performant at tens to hundreds of thousands of variables [32], [53], our domain-specific co-design approach enables the GPU to outperform the CPU even on moderately-sized linear systems (our experiments range from 448 to 7,168 variables).

Performance Distribution: Importantly, in most cases, the speedup is much larger than this. This is because iterative methods have a variable runtime as they can exit early depending upon the exit tolerance, ϵ. We demonstrate this using the 128 knot point problem as a case study in Figure 5.

We plot the distribution of solve times for QDLDL against GPU-PCG $\epsilon = 1e^{-4}$, resulting in our 1.9x average speedup in Figure 4, as well as $\epsilon = 5e^{-5}$ and $\epsilon = 1e^{-5}$. QDLDL

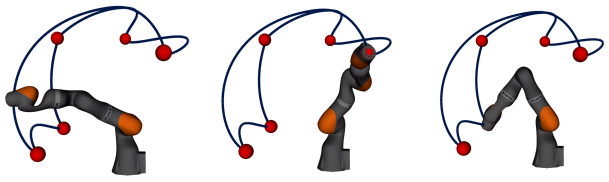

Fig. 3: Screenshots from the 5 goal, pick-and-place circuit used in our experiments.

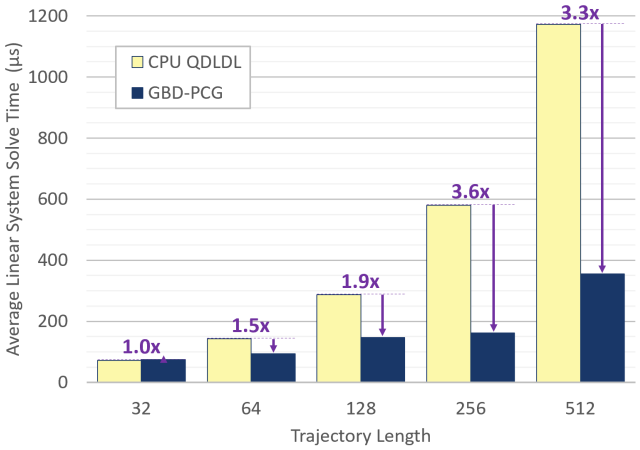

Fig. 4: Average linear system solve time for the thousands of solves within each of the 100 iterations of NMPC. As the problem scales, so does the advantage of GBD-PCG over QDLDL achieving up to a 3.6x average speedup.

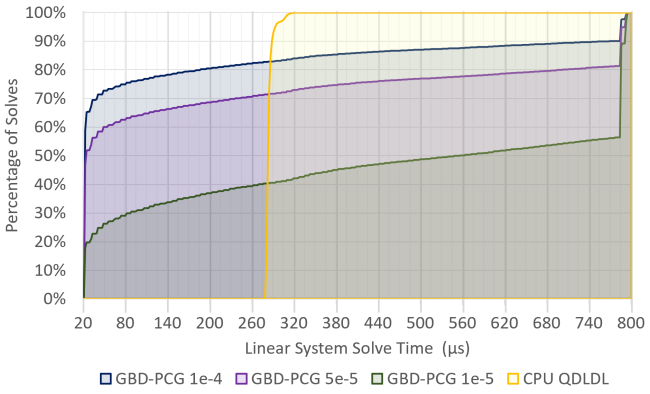

Fig. 5: Cumulative density function of the linear system solve times for a trajectory length of 128 for QDLDL and GBD-PCG under multiple different exit tolerances, ϵ. We note that the bi-modal distribution for GBD-PCG is often much faster than the uni-modal distribution for QDLDL. For example, for $\epsilon = 1e^{-4}$, 65% of GBD-PCG solves are $\geq$10x faster than the fastest QDLDL solve, and the slowest GBD-PCG solve is only 2.5x slower than the slowest QDLDL solve (with only 10% of solves $\geq$2x slower).

presents a uni-modal timing distribution with almost all solves occurring between 280 and 305 μs. GBD-PCG, on the other hand, presents a bi-modal timing distribution clustered both much faster and a little slower than QDLDL. For example, for $\epsilon = 1e^{-4}$, 65% of GBD-PCG solves are $\geq$10x faster than the fastest QDLDL solve, and the slowest GBD-PCG solve is only 2.5x slower than the slowest QDLDL solve (with only 10% of solves $\geq$2x slower).

Furthermore, while all values of ϵ shown in the plot were able to successfully track the target trajectory, the lower the exit tolerance, the more of the distribution mass shifted to being $\geq$10x faster than QDLDL (65%, 52%, 20% for $\epsilon = 1e^{-4}, 5e^{-5}, 1e^{-5}$ respectively). However, when ϵ was reduced even farther, our entire NMPC controller was unable to accurately track our target trajectory. These results present interesting directions for future work to find ways to eliminate the second slower mode of the solve time distribution while ensuring robust NMPC convergence.

C. End-to-End NMPC Performance

To validate efficacy for use in NMPC for robotics applications, we also demonstrate the impact of our approach on the number of iterations of MPCGPU we could achieve at each control step for varying control rates and trajectory lengths. Figure 6 shows the resulting number of average trajectory optimization solver iterations we can compute while meeting the specified control rates and trajectory lengths using both our GBD-PCG solver as well as QDLDL to solve the thousands of underlying linear systems.[2]

Regardless of the linear system solver, our GPU-first approach, with both fast parallel construction of the Schur complement and fast parallel computation of the line search, enables trajectories as long as 128 knot points to operate at a 1kHz control rate, and achieve at least 4 iterations at a 500Hz control rate, for a per-iteration rate of 2kHz. Furthermore, similar to what we witness in the case of average linear system solve times, as the problem gets larger and the control rate increases, our fully GPU-based approach is increasingly performant. Highlights include our approach's ability to scale to 512 knot points at a 1kHz control rate and execute 8 iterations for 128 knot points at a 500Hz control rate, for a per-iteration rate of 4kHz. These results compare favorably to previously reported results in the literature of about 500hz to 1kHz per-iteration rates for trajectories of 30 to 120 knot points using state-of-the-art CPU-based [77], [78] and GPU-based [26] solvers for similar NMPC tasks. As such, our GPU-first approach opens up the possibility for our NMPC solver to either leverage longer horizon trajectories, run at faster control rates, produce more optimal solutions for the same horizon and control rate, or include some combination of those highly beneficial traits.

[2]We note that in the QDLDL case, as the NMPC loop is running on the GPU, data needs to be copied onto the host before executing the solve and converted into the sparse CSR matrix format. To ensure fair comparisons and avoid overheads for unnecessary data transfers and transformations, we implemented a variant of our parallel Schur complement computation which directly stores data in the CSR format expected by QDLDL.

MPCGPU with QDLDL		Knot Points				
		32	64	128	256	512
Control Rate	250Hz	21.0	14.0	8.0	4.0	2.0
	500Hz	10.0	6.5	4.0	2.0	1.0
	1kHz	4.0	3.0	1.0	X	X

MPCGPU with GBD-PCG		Knot Points				
		32	64	128	256	512
Control Rate	250Hz	22.2	19.7	15.4	5.2	4.4
	500Hz	10.3	10.6	8.0	4.6	3.0
	1kHz	4.9	5.2	3.7	2.4	1.7

Fig. 6: Average number of trajectory optimization iterations of MPCGPU executed at each control step for varying control rates and trajectory lengths. Our GPU-first approach to Schur complement construction and our parallel line search enables high control rates and long trajectories regardless of the underlying solver. However, the improved scalability of GBD-PCG enables our approach to scale to 512 knot points at 1kHz and execute 8 iterations for 128 knot points at 500Hz, for a per-iteration rate of 4kHz.

VI. Conclusion and Future Work

In this work, we introduce MPCGPU, a GPU-accelerated, real-time NMPC solver built around a parallel PCG solver. MPCGPU exploits the structured sparsity and natural parallelism in both direct trajectory optimization algorithms and iterative linear system solvers. Our experiments show that our approach is able to scale NMPC to larger problems, and operate it at faster rates, than is possible with existing state-of-the-art solvers. In particular, for tracking tasks using the Kuka IIWA manipulator, MPCGPU is able to scale to kilohertz control rates with trajectories as long as 512 knot points. For this problem, our GPU-based PCG solver outperforms a state-of-the-art CPU-based linear system solver by as much as 10x for a majority of solves and 3.6x on average.

There are many promising directions for future work to improve the functionality and usability of our approach. Most importantly, like all iterative methods, our GPU-based PCG solver exhibits variability in its solve times. Future work which learns when to leverage iterative methods vs. factorization-based methods, or which learns dynamic values for hyperparameters to reduce the worst-case runtimes, without sacrificing overall NMPC robustness, would greatly improve average-case performance. Furthermore, it would be interesting to explore the performance implications of adding additional constraints either through expanding the KKT system, or through augmented Lagrangian or operator splitting methods [10], [79], [80]. Finally, we would like to evaluate our approach on physical robots at the edge using low-power GPU platforms such as the NVIDIA Jetson [81].

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, F. Mascarich, L. Bernreiter, C. Godinho, M. Camurri, S. Khattak, T. Dang, V. Reijgwart, J. Loeje, D. Wisth, S. Zimmermann, H. Nguyen, M. Fehr, L. Solanka, R. Buchanan, M. Bjelonic, N. Khedekar, M. Valceschini, F. Jenelten, M. Dharmadhikari, T. Homberger, P. D. Petris, L. Wellhausen, M. Kulkarni, T. Miki, S. Hirsch, M. Montenegro, C. Papachristos, F. Tresoldi, J. Carius, G. Valsecchi, J. Lee, K. Meyer, X. Wu, J. Nieto, A. Smith, M. Hutter, R. Siegwart, M. Mueller, M. Fallon, and K. Alexis, "Cerberus: Autonomous legged and aerial robotic exploration in the tunnel and urban circuits of the darpa subterranean challenge," *arXiv preprint arXiv:2201.07067*, 2022.

[4] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. Del Prete, "Optimization-based control for dynamic legged robots," *arXiv preprint arXiv:2211.11644*, 2022.

[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 Using Nonlinear Programming*, ser. Advances in Design and Control. Society for Industrial and Applied Mathematics (SIAM), vol. 3.

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

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

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

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

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

[12] 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*, ser. ISCA '11. ACM, pp. 365–376.

[13] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, pp. 205–218.

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

[15] 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.

[16] 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.

[17] 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," ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 674–686. [Online]. Available: https://doi-org.ezp-prod1.hul.harvard.edu/10.1145/3445814.3446746

[18] 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, pp. 6253–6260. [Online]. Available: https://ieeexplore.ieee.org/document/9812384/

[19] 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.

[20] S. M. Neuman, R. Ghosal, T. Bourgeat, B. Plancher, and V. J. Reddi, "Roboshape: Using topology patterns to scalably and flexibly deploy accelerators across robots," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589104

[21] Y. Yang, X. Chen, and Y. Han, "Rbdcore: Robot rigid body dynamics accelerator with multifunctional pipelines," *arXiv preprint arXiv:2307.02274*, 2023.

[22] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl, "A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control." [Online]. Available: http://arxiv.org/abs/1711.11006

[23] 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*.

[24] 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*.

[25] Y. Jiang, J. Oravec, B. Houska, and M. Kvasnica, "Parallel mpc for linear systems with input constraints," *IEEE Transactions on Automatic Control*, vol. 66, no. 7, pp. 3401–3408, 2020.

[26] 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)*.

[27] ——, "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.

[28] S. C. Eisenstat, "Efficient implementation of a class of preconditioned conjugate gradient methods," *SIAM Journal on Scientific and Statistical Computing*, vol. 2, no. 1, pp. 1–4, 1981.

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

[30] B. Plancher, "Gpu acceleration for real-time, whole-body, nonlinear model predictive control," Ph.D. dissertation, Harvard University, Cambridge, MA, USA, April 2022.

[31] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," vol. 236, no. 15, pp. 3584–3590. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042711002196

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] K. Świrydowicz, E. Darve, W. Jones, J. Maack, S. Regev, M. A. Saunders, S. J. Thomas, and S. Peleš, "Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers," *Parallel Computing*, vol. 111, p. 102870, 2022.

[40] 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.

[41] 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.

[42] F. Pacaud, S. Shin, M. Schanen, D. A. Maldonado, and M. Anitescu, "Accelerating condensed interior-point methods on simd/gpu architec-

tures," *Journal of Optimization Theory and Applications*, pp. 1–20, 2023.

[43] 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, pp. 917–924. [Online]. Available: http://doi.acm.org/10.1145/1201775.882364

[44] 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.

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

[46] 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.

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

[48] M. Tiwari and S. Vadhiyar, "Strategies for efficient execution of pipelined conjugate gradient method on gpu systems," in *International Conference on High Performance Computing*. Springer, 2022, pp. 77–89.

[49] 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.

[50] L.-W. Chang and W.-m. W. Hwu, *A Guide for Implementing Tridiagonal Solvers on GPUs*. Springer International Publishing, 2014, pp. 29–44. [Online]. Available: https://doi.org/10.1007/978-3-319-06548-9_2

[51] 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.

[52] 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.

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

[54] S. Heinrich, A. Zoufahl, and R. Rojas, "Real-time trajectory optimization under motion uncertainty using a GPU," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3572–3577.

[55] 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.

[56] 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.

[57] D.-K. Phung, B. Hérissé, J. Marzat, and S. Bertrand, "Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit," vol. 50, no. 1, pp. 11 883–11 888. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896317319614

[58] 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.

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

[60] 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.

[61] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S096706611830710X

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038234

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

[70] A. Wächter and L. T. Biegler, "On the Implementation of an Interior-point Filter Line-search Algorithm for Large-scale Nonlinear Programming," vol. 106, no. 1, pp. 25–57. [Online]. Available: https://doi.org/10.1007/s10107-004-0559-y

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

[72] M. Schubiger, G. Banjac, and J. Lygeros, "GPU Acceleration of ADMM for Large-Scale Quadratic Programming," vol. 144, pp. 55–67. [Online]. Available: http://arxiv.org/abs/1912.04263

[73] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," vol. 6, pp. 40–53.

[74] NVIDIA, *NVIDIA CUDA C Programming Guide*, version 9.1 ed. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[75] X. Bu and B. Plancher, "Symmetric stair preconditioning of linear systems for parallel trajectory optimization," in *IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, May. 2024.

[76] 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. [Online]. Available: https://doi.org/10.1007/s12532-020-00179-2

[77] C. Mastalli, R. Budhiraja, W. Merkt, G. Saurel, B. Hammoud, M. Naveau, J. Carpentier, L. Righetti, S. Vijayakumar, and N. Mansard, "Crocoddyl: An efficient and versatile framework for multi-contact optimal control," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 2536–2542.

[78] 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.

[79] 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.

[80] Z. Zhou and Y. Zhao, "Accelerated admm based trajectory optimization for legged locomotion with coupled rigid body dynamics." IEEE, 2020, pp. 5082–5089.

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