cuRobo:
Parallelized Collision-Free Minimum-Jerk Robot Motion Generation

Balakumar Sundaralingam[*] Siva Kumar Sastry Hari[*] Adam Fishman Caelan Garrett

Karl Van Wyk Valts Blukis Alexander Millane Helen Oleynikova

Ankur Handa Fabio Ramos Nathan Ratliff Dieter Fox

NVIDIA

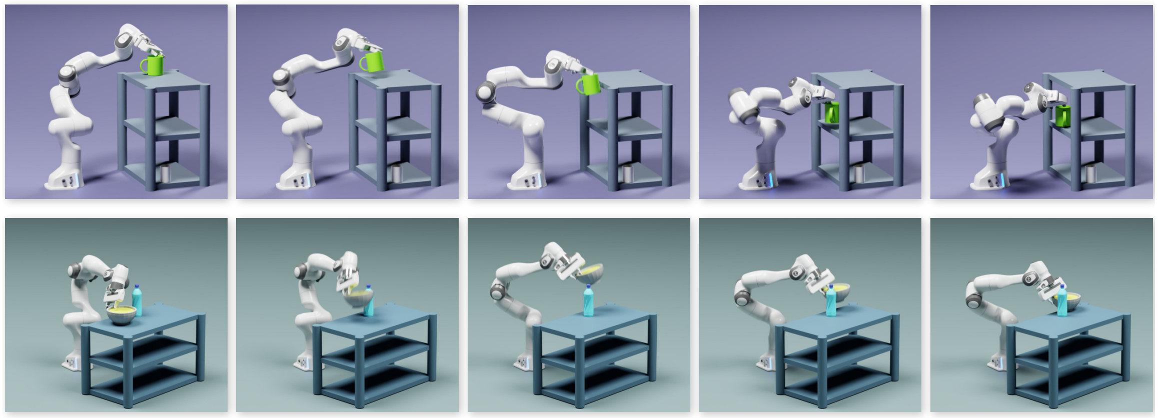


Figure 1: We present an approach to perform collision-free motion generation for a manipulator to move from a start configuration to a desired gripper pose. Few instances of using our approach is shown here, where the Franka Panda robot moves a grasped object around obstacles to place in a target location. We leverage parallel computing to generate motions within 30ms.

Abstract

This paper explores the problem of collision-free motion generation for manipulators by formulating it as a global motion optimization problem. We develop a parallel optimization technique to solve this problem and demonstrate its effectiveness on massively parallel GPUs. We show that combining simple optimization techniques with many parallel seeds leads to solving difficult motion generation problems within 50ms on average, 60× faster than state-of-the-art (SOTA) trajectory optimization methods. We achieve SOTA performance by combining L-BFGS step direction estimation with a novel parallel noisy line search scheme and a particle-based optimization solver. To further aid trajectory optimization, we develop a parallel geometric planner that plans within 20ms and also introduce a collision-free IK solver that can solve over 7000 queries/s. We package our contributions into a state of the art GPU accelerated motion generation library, *cuRobo* and release it to enrich the robotics community. Additional details are available at curobo.org.

[*]Equal Contribution.

Contents

List of Algorithms

List of Tables

1 Introduction

Safe navigation is fundamental to robotics [1], requiring robots to have a robust global motion generation system to traverse any environment structure encountered at deployment. Motion generation for high-dimensional systems is extremely challenging as satisfying complex constraints and minimizing cost terms in a very large C-Space is computationally expensive. Manipulators, for instance, can have many articulations, complex link geometries, entire goal regions beyond a single configuration, task constraints, and nontrivial kinematic and torque limitations. There has been a long history of problem decomposition in this field to mitigate complexity, leading to standard approaches that often first plan collision-free geometric paths [2, 3] and then smooth those paths for dynamic efficiency [3, 4]. But increasingly, research into the interconnections between optimization and planning [5–12] has shown that optimization can be a powerful tool well beyond trajectory smoothing, and trajectory optimization alone now has a breadth of applications [13–16]. Our modern understanding of this robot navigation problem is that it is a large *global* motion optimization problem [17, 18].

The global optimization literature suggests that finding the true global minimum is usually impractical, but strategies for robustly finding high-performing local minima can be effective [19]. Many strategies follow the simple pattern of selecting many seed candidates and performing a local optimization for each. This sample and optimize process can often realize substantial gains by leveraging distributed computation. However, most motion generation systems today remain sequential and slow, following a CPU-based design. State-of-the-art motion generation solutions take 0.5s to 10s depending on the task's complexity on modern CPUs [20]. This run-time is even slower on edge devices that operate under limited power budgets. This slow and sequential process has resulted in pipelined systems that compute only a single best candidate seed, which is then passed to an optimizer for local optimization [3]. Such systems fundamentally limit their ability to find better local optima by betting on a single seed.

The insights used to improve the speed and quality of the solution for global optimization problems may apply well to the problem of global motion generation. In this work, we present a collection of techniques and implementations that leverage parallel processing to accelerate motion planning and optimization, and for running many optimization instances in parallel to robustly address these global optimization problems. Existing literature supports these algorithmic principles and has shown that (1) the heuristic initialization for the problem can be effective [9], and (2) many restarts with randomized noise of the initial seed can dramatically improve performance [21].

In the realm of global motion generation, massively parallel compute is already being used to accelerate Probabilistic Road Map (PRM) pruning by using an FPGA with special circuits, leading to many orders of magnitude speedup [22]. However, instead of designing special circuits for motion generation, we leverage the massively parallel compute available on graphical processing units (GPUs). GPUs have become pervasive in both high- and low-powered configurations as they offer energy-efficient and high-throughput computation platforms, an important requirement for solving parallelizable compute intensive problems. We show how GPUs also offer programmability and flexibility to map sophisticated computations of motion optimization to hardware, allowing us to parallelize the entire motion generation pipeline. We achieve high speedups using GPUs compared to serial implementations in motion generation. While we demonstrate the benefits of using parallel compute for motion generation using NVIDIA GPUs, the approach is applicable to other parallel architectures.

Our effort to solve global motion generation starts with parallelizing the core blocks in a robotics stack – robot kinematics, robot self signed distance (i.e., between a robot's links), and robot world signed distance (i.e., world represented by cuboids, meshes, and a depth camera stream). We formalize these functions to use many threads per query, implement them efficiently in CUDA, and provide them as differentiable functions in pyTorch, enabling others to also use these functions as the backbone for their own robotic tasks. We then formulate a continuous collision checking algorithm that only requires a point signed distance function from the world representation. We then introduce parallel algorithms for numerical optimization and geometric planning, that aid in solving global motion generation. Our main contributions are summarized as follows:

Performant Kinematics and Signed Distance Kernels: We develop high-performance CUDA kernels for robot kinematics and signed distance computation which are up to 10,000x faster than existing CPU based methods.

Differentiable Continuous Collision Checking: Formulate continuous collision checking algorithm that only needs a point signed distance function (and closest point for gradients) to perform swept collision checks, enabling use across different world representations from primitives and meshes to occupancy maps.

Parallel Optimization: We develop a GPU batched L-BFGS optimizer, that uses an approximate parallel line search scheme, and a particle-based optimizer to solve difficult motion generation problems. Our solver is 23×, 80×, and 87× faster for inverse kinematics, collision-free inverse kinematics, and collision-free trajectory optimization respectively when compared to existing CPU based solvers.
SOTA IK Solver: Leveraging our performant kernels, we have developed a world-leading inverse kinematic solver, that can solve 37000 IK problems per second (23× faster than TracIK [23]) and also solve 7600 collision-free IK problems per second (80× faster than using TracIK + Bullet [24]).
Parallel Geometric Planner: We develop a geometric planner with a parallel steering algorithm to generate collision-free paths within 20 ms on a modern desktop machine with NVIDIA RTX 4090 and AMD Ryzen 9 7950x.
Global Motion Generation: Combining our above contributions, we have a global motion generation pipeline that can plan within 50ms, 60× faster than existing methods (Tesseract).
Validation on a Low-Power Device: We evaluate our GPU-accelerated motion generation stack and existing CPU-based methods on an NVIDIA Jetson AGX Orin at different power budgets. Results show that our approach is 28× and 21× faster on average for motion generation problems when the device was set to 60W and 15W budgets, respectively.
cuRobo Library: We developed *cuRobo*, a suite of GPU-accelerated robotics algorithms, providing SOTA implementations of robot kinematics, signed distance functions, optimization solvers, geometric planning, trajectory optimization, and model predictive control. We are releasing this library to enrich roboticists with the necessary tools to explore large-scale problems in robotics.

2 Motion Generation as Optimization

We define the problem of motion generation as the task of moving from an initial joint configuration θ_0 to a final joint configuration θ_T, at which state a task cost $C(\theta_T)$ is below a desired threshold. Additionally, the transition states from θ_0 to θ_T must also satisfy system constraints. In this work, we focus on the task of collision-free motion generation to reach a goal Cartesian pose $X_g \in \mathbb{SE}(3)$ with the robot's end-effector. Specifically, we want to obtain a joint-space trajectory $\theta_{[0,T]}$ that satisfies the robot's joint limits (position, velocity, acceleration, jerk), doesn't collide with itself or the environment, and reaches the goal pose X_g by the last timestep T.

We formulate this continuous-time motion problem as a time discretized trajectory optimization problem,

$$\begin{aligned}
\underset{\theta_{[1,T]}}{\arg\min} \quad & C_{\text{task}}(X_g, \theta_T) + \sum_{t=1}^{T} C_{\text{smooth}}(\cdot) && (1)\\
\text{s.t.,} \quad & \theta^- \le \theta_t \le \theta^+, \forall t \in [1,T] && (2)\\
& \dot{\theta}^- \le \dot{\theta}_t \le \dot{\theta}^+, \forall t \in [1,T] && (3)\\
& \ddot{\theta}^- \le \ddot{\theta}_t \le \ddot{\theta}^+, \forall t \in [1,T] && (4)\\
& \dddot{\theta}^- \le \dddot{\theta}_t \le \dddot{\theta}^+, \forall t \in [1,T] && (5)\\
& \dot{\theta}_T, \ddot{\theta}_T, \dddot{\theta}_T = 0 && (6)\\
& C_r(K_s(\theta_t)) \le 0, \forall t \in [1,T] && (7)\\
& C_w(K_s(\theta_t)) \le 0, \forall t \in [1,T] && (8)
\end{aligned}$$

where $C_{smooth}(\cdot)$ is a cost term that encourages smooth robot behavior. Joint limit constraints are enabled by Eq.2-5. We also constrain the robot to have zero velocity, acceleration and jerk at the final timestep by constraints in Eq. 6. A detailed discussion on this optimization problem and the formulation of cost terms is available in Appendix. A. We discuss the collision avoidance constraints Eq.7, and Eq.8 in Sec. 3.

A good initial seed can speedup convergence in the above defined trajectory optimization problem. One common way [14] to initialize the seed is to first optimize only for the terminal joint configuration θ_T and then initialize the trajectory with a linear interpolation from the start configuration θ_0 to the solved terminal configuration (interpolating through a predefined waypoint has also shown to be helpful [9]). In our problem setting of reaching a goal pose X_g, the terminal state optimization problem boils down to a collision-free inverse kinematics (IK) problem containing the pose cost, the collision constraints Eq. 8-Eq. 7 and the joint limit constraint Eq. 2. We hence first solve for collision-free IK, followed by seed generation, and then trajectory optimization. Once we run trajectory optimization, we find an optimal dt by scaling the

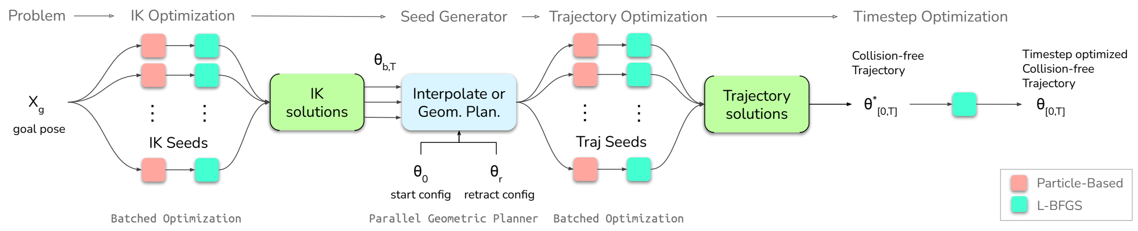


Figure 2: Our approach to global motion generation takes as input a goal pose X_g, initial joint configuration θ_0, and outputs a timestep optimized, collision-free trajectory $\theta_[0,T]$. We solve this by first running many instances of collision-free IK, followed by generating seeds for trajectory optimization by either linearly interpolating between start and IK solutions, passing through a retract config, or using our geometric planner. We then run trajectory optimization on many seeds in parallel to obtain a collision-free trajectory $\theta^*_{[0,T]}$, which is then re-optimized with an estimated dt to get a timestep optimized collision-free trajectory $\theta_{[0,T]}$. Our numerical optimization performs a few iterations of particle-based optimization to move the seed to a good region follow by L-BFGS to quickly converge to the minimum.

trajectory's velocity, acceleration, or jerk to the robot's limits and rerun trajectory optimization with this new dt to get the final result (see A.4). Our overall approach is illustrated in Figure 2.

3 Kinematics & Collision Avoidance

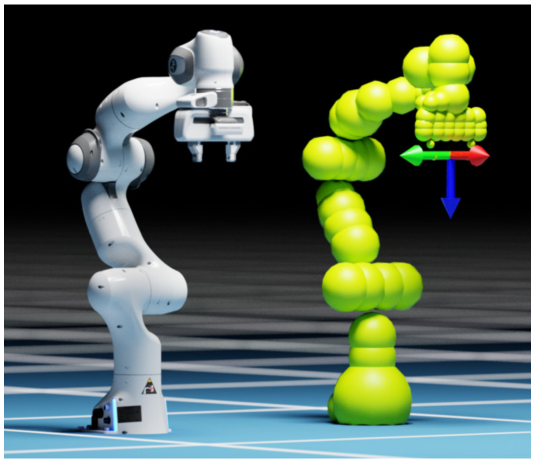
Collision avoidance is a critical component of motion generation as the robot needs to be able to avoid colliding with itself (self-collision) and with the world for safe operation. A standard approach to computing collisions involves transforming the robot's geometries (often represented as meshes) based on the current joint configuration (forward kinematics) and computing mesh-mesh distances [25–27, 27, 28]. Since we know the geometry of the robot, we can reduce the computation required for collision checking by representing the robot's volume with a set of spheres [29] as shown in Figure 3. With this sphere representation for the robot, our collision avoidance cost terms only need to check the distance between the origin of each sphere and the world, then subtract the radius to get the sphere distance. Similarly, for self collisions, we only have to compute the distance between pairs of spheres (i.e. compute point distance and subtract the radii of the two spheres). This enables our approach to scale to low-power edge devices and also accommodate very large batch queries. We discuss some techniques to approximate a mesh with spheres in Appendix D. We will next discuss how we map between the robot joint configuration and the location of the spheres.

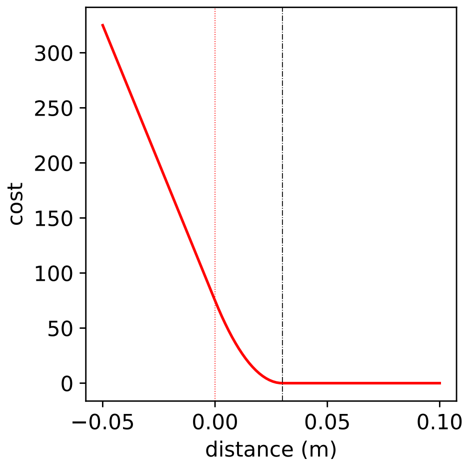
3.1 Robot Kinematics

Robot kinematics $K_s(\cdot)$ enables mapping between a robot's joint configuration and the Cartesian pose (SE(3)) of all geometries attached to the robot. This mapping is done by computing a sequence of transformations from the base link of the robot to the different links attached through joints. Each actuated joint adds an additional transformation based on it's value and type. Hence, traversing the robot's kinematic tree by design is sequential for serial manipulators. To overcome the sequential nature of computation, we represent the transformations as homogeneous matrices (4x4), enabling us to use four parallel threads to compute matrix multiplications. Once we build the pose of all links of the robot, we perform matrix vector products to compute the position of the spheres. We also output the pose of the end-effector as a position and quaternion. For computing the backward, we use 16 threads to read and project the gradients from the Cartesian space to the joint space. By using many threads for a single kinematics query, we overcome some of the memory overhead that comes with parallel compute devices. Our kinematics function supports single axis actuation across all three linear and three angular spaces. Extensive details are available in Appendix E.1. Figure 3-a shows the output of our forward kinematic function, given a joint configuration of the robot.

3.2 Self-Collision Avoidance

For avoiding self-collisions, we formulate a distance cost that computes the largest penetration distance between spheres from all links. Since most robots allow for safe contact between consecutive links and some


(a) Sphere representation of Franka Panda



(b) Quadratic collision cost profile

Figure 3: A sphere representation of the Franka Panda is shown in the left. We also visualize the end-effector frame by the axis near the gripper. cuRobo's kinematics function takes the joint configuration of a robot and outputs the location of these spheres and the pose of the end-effector. We generate bounding spheres for the robots using NVIDIA Isaac Sim's sphere generator (website). On the right, we show the quadratic cost profile for smoothly transitioning from collision (dotted vertical line at 0m) to non-collision space using an activation distance η (shown by dashed black vertical line at 0.03m).

link pairs will never be in collision due to kinematic limits, we build a set of sphere pairs S for which self-collision needs to be checked. We empirically found only 50% of the sphere pairs end up in this collision pair set across many widely used manipulators. We scale the largest penetration distance by a scalar weight β_1. Our self-collision term can be written as,

$$C_r(K_s(\theta_t)) = \beta_1 \max_{i,j \in S}(\max(0, s_{i,r} + s_{j,r} - ||s_{i,x} - s_{j,x}||)) \tag{9}$$

where $s_{i,x}, s_{j,x} \in \mathbb{R}^3$ are the positions, $s_{i,r}, s_{j,r} \in \mathbb{R}$ are the radii of of spheres i and j respectively.

3.3 World Collision Avoidance

Generating smooth obstacle avoidance behavior has been studied extensively in the the literature [7, 11, 28, 30]. We highlight common pitfalls in collision-free motion optimization and how our approach overcomes them by leveraging existing techniques and introducing novel contributions below,
Discontinuity at surface boundary: Discontinuity in the collision cost term near an obstacle surface leads to poor conditioning of the optimization problem, especially when collision cost term is non-convex. To mitigate this issue, we add a buffer distance η and change the cost to be quadratic when within η distance to the obstacle surface similar to [7] as shown in Fig. 3-(b). This modification of the collision distance d_c, given the signed distance d can be written as,

$$d_c = \begin{cases} d + 0.5\eta & \text{if } d > 0 \\ \frac{0.5}{\eta}(d+\eta)^2 & \text{if } -\eta < d < 0 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Speeding through obstacles: When a collision cost term only penalizes the position of the sphere, the optimization can attempt to move through obstacles (i.e., high penalty region) very fast to reach a lower cost region compared to being in collision for many timesteps (see our website for a visualization of this phenomenon). To mitigate this issue, we implement a speed metric, similar to [7], that scales the collision cost of a sphere by it's velocity $\dot{s}$ (calculated through finite-difference). This encourages the optimization to move around the obstacle instead of speeding through an high penalty region.

Collision at real-robot execution: Tuning a robot's control box to track a planned kinematic trajectory with millimeter accuracy at high speeds can be very time consuming. In addition, most manufacturers do not provide many parameters to tune their control box. Any Path deviation near obstacles could lead to catastrophic collisions with the world. To be robust to path deviations, we penalize the robot's velocity when within η distance to obstacles as robots can track with higher accuracy at slower speeds. This penalization is performed by enabling our speed metric when within η distance instead of only enabling at collision. This brings our collision term to $d_s = \dot{s}d_c$.

Collision with thin obstacles: Motion optimization is commonly done by discretizing the trajectory by some timesteps and computing collisions at these timesteps. However, if the trajectory does not have a fine resolution of discretization, collisions with very thin obstacles could be missed. To overcome this issue, we develop a novel formulation of continuous collision checking that only requires a point query signed distance function from the world representation, enabling continuous collision checking with a variety of world representations. We discuss this formulation in the next Section 3.4.

3.4 Continuous Collision Checking

Continuous collision checking is a well studied problem [31–33], with many methods building a swept volume followed by checking collision between the swept volume and the obstacles. However, collision checking using swept volumes requires the world representation to be able to compute signed distance between complex geometry (i.e., the swept volume) and the obstacles in the world. This is only possible for worlds represented by primitive shapes or meshes. Worlds represented using neural networks [34, 35] or voxels [36] will require special mechanisms to work with swept volumes. To avoid this complexity, we introduce a novel formulation of continuous collision checking that only requires a point signed distance query function from a world representation. We hope that this reduces the barrier for the perception community to deploy their world representations into cuRobo for global collision-free motion generation. Our method is related the iterative method from Bruce [37] and loosely related to the bubbles concept from Quinlan and Khatib [38]. The difference between our approach and Bruce's approach is we not only compute the signed distance but also the gradients. We also formulate the computation to run in parallel threads for each time step in the trajectory.

Our continuous collision checking algorithm is illustrated in Fig. 4. Given a trajectory of a sphere discretized by three timesteps $S_{[0,1,2]}$, we first check if the sphere S_1 is in collision. If it is in collision, we compute the collision cost and move by sphere radius. If it's not in collision, we compute the signed distance to the nearest obstacle and move this distance along the direction of motion between S_0 and S_1, which we term as *sweep backward.* If we hit a collision, then we compute the collision cost and then continue sweeping until we reach the midpoint between S_0 and S_1. Similarly, we *sweep forward* until midpoint between S_1 and S_2. For every sphere location in the trajectory, we *sweep forward* and *sweep backward* upto the mid distance as this enables our gradient computations to be parallelizable (i.e., gradient for a sphere location does not depend on the collisions at other sphere sweeps).

Our implementation of the algorithm assumes that the sphere is moving linearly between waypoints which is not true for links attached through a revolute joint. Empirically, we found that even with this assumption, the optimization was able to find paths in tight spaces. We leave incorporating exact movement path between sphere waypoints for future work and discuss the implementation of this algorithm in Appendix E.3.

3.5 World Representation

To compute the world collision cost term, we only require the ability to compute the closest point $c_r \in \mathbb{R}^3$ to a query point and also if the query point is inside or outside an obstacle $s_r \in [-1, 1]$. These quantities can be obtained from a differentiable point query signed distance function or through geometric processing. We implement three different world representations that output these quantities through geometric processing,

Oriented Bounding Box We implement an efficient cuboid query function as we found cuboids to be a common representation for collision avoidance in many real-world deployments. For this world representation, we assume the world is made of only oriented bounding boxes (i.e., cuboids with a SE(3) pose).

Mesh Leveraging NVIDIA warp's bounding volume hierarchy (BVH), which stores mesh's faces and vertices in an accelerated framework for fast closest point and inside/outside queries. This representation assumes the world is represented by watertight meshes.

Depth Camera Third, we write a wrapper to NVIDIA nvblox [39] which integrates Euclidean Signed Distance (ESDF) from truncated Signed Distance Fields (TSDF) streaming from a depth camera. This representation enables us to build a ESDF voxel representation of the world using a depth camera

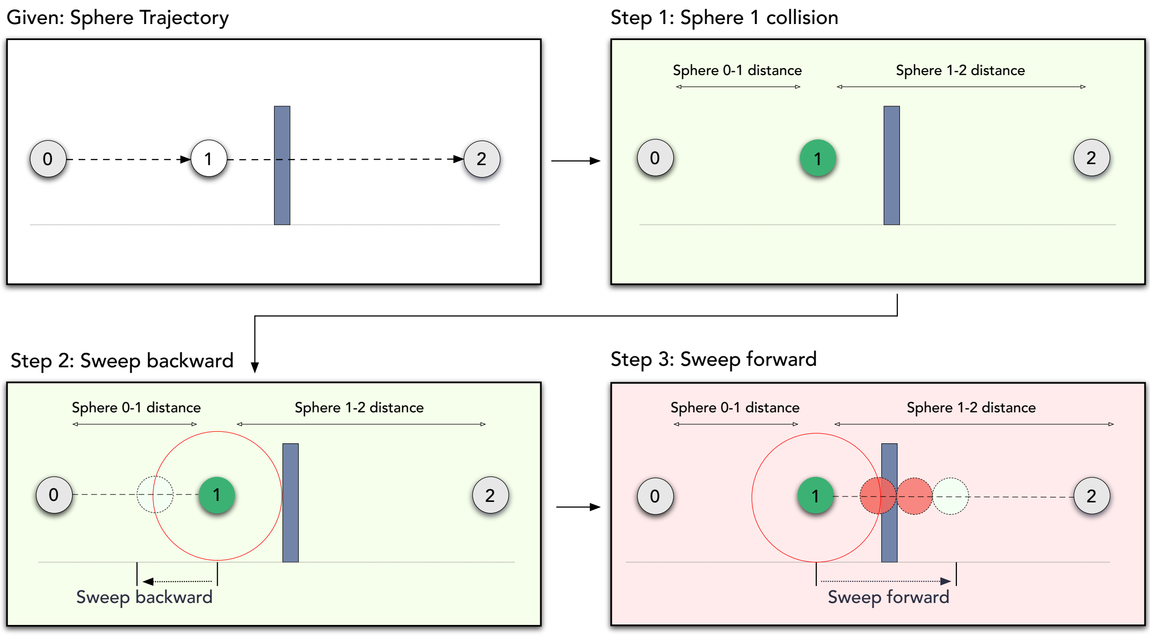


Figure 4: Our continuous collision checking algorithm is illustrated for a sphere moving from position 0 to 2 through 1. To compute the collision distance at a timestep S_1, we first check if S_1 is in collision (step 1). If it is not in collision, we compute the distance to the closest obstacle (shown as red circle in step 2) and *sweep backward* by checking for collision at this distance along the trajectory between S_0 and S_1. If no collision is found, we repeat until we reach midpoint between S_0 and S_1. We similarly do this iterative process between S_1 and S_2 in *sweep forward* (step 3).

and use this for computing collision distance. This representation assumes that we have access to a depth camera and accurate pose of the camera with respect to the robot's base at each frame for integrating into nvblox.

Our implementation of the collision cost also allows for using a combination of the three world representations as we sum over all collisions in the world. In addition to these representations, our robot sphere representation allows for interfacing with other methods that can output a differentiable signed distance [34, 35, 40]. For example, Tang *et al.* [41] integrated a learned SDF representation of the world [35] for collision avoidance in cuRobo.

The overall world collision term can be written as,

$$C_w(S_{t-1,t,t+1}) = \beta_2 \text{speed}(S_{t-1,t,t+1}) \text{smooth}(\text{sweep}(S_{t-1,t,t+1})) \tag{11}$$

where β_2 scales the cost by a large penalty to act as a soft constraint, sweep($\cdot$) computes the collision distance using our continuous collision checking algorithm from Sec. 3.4. The function smooth($\cdot$) adds the quadratic smoothing over the collision distance using Eq. 10, which is then scaled by the velocity of the sphere speed($\cdot$). We sum this term across all spheres that represent the robot.

4 Parallel Optimization Solver

There are several techniques to solve the optimization problem defined in Section 2, from particle-based optimization [8, 42] to gradient-based optimization [7, 9, 12, 21] methods. In particular many trajectory optimization methods [7, 12, 21] have approximated hard constraints as soft constraints by treating them as cost terms with large weights to transform the optimization problem from one with nonconvex constraints to a box-constrained nonconvex optimization problem. Motivated by these successes, we also approximate our constraints as cost terms and implement a quasi-newton solver to solve this nonconvex optimization problem.

L-BFGS, a quasi-newton optimization method that can solve very large optimization problems, is a common method shown to achieve superlinear convergence by estimating the Hessian using evaluated gradients. Our optimizers are built around L-BFGS because of its combined performance and relative simplicity that aids parallelization. Gauss-Newton solvers are also ubiquitous and important in robotics [10, 43, 44], but after an initial exploration we decided to focus our experiments on L-BFGS. Many formulations of Gauss-Newton restrict their presentation to the nonlinear least-squares problem [10, 45] where performance is best understood, but that's unduly restrictive in our setting. These methods can be generalized as a form of natural gradient descent [11] and related formulations of iLQG [46] demonstrate their empirical utility on more general costs using quadratic approximations. However, appropriately leveraging the problem structure within a GPU is not straightforward and the band-diagonal solve commonly used is inherently sequential leaving a number of open questions we would need to resolve. Our benchmarks indicate that even the simpler L-BFGS shows significant improvement over the state-of-the-art when GPU compute is properly leveraged; we leave a full exploration of Gauss-Newton to future work.

4.1 Parallel L-BFGS Optimization

Our L-BFGS optimizer has two steps, we first compute the step direction given the current optimization variables $\Theta = \theta_{t\in[0,T]}$ and the gradient $\Delta\Theta$ with respect to the sum of the cost terms using the standard L-BFGS steps as described in Nocedal and Wright[45]. Given this step direction $\Delta\Theta$, we perform line search by scaling the step direction with a discrete set of magnitudes $\alpha \in \mathbf{R}^n$ and computing the best magnitude from this set using Armijo and Wolfe conditions as shown in Alg. 1. Extensive details on our solver is available in Appendix A.7.

Our approach of trying a predefined discrete set of magnitudes instead of iteratively searching for the largest magnitude that satisfies the condition enables us to more effectively use parallel compute as the cost and gradient for the discrete set can be computed in parallel. For the case where none of the values in our discrete set satisfies the line search conditions, we use a very small magnitude (0.01) which acts as a noisy step update. This noisy step update also prevents NaN values in the the step direction computation as there is always a perturbation in the optimization variables between iterations. After every optimization iteration, we update our best estimate as the optimization could diverge due to noisy perturbation in line search. Empirically, we found that the use of a noisy perturbation instead of stopping the optimization when line search fails to find a magnitude that satisfied the conditions greatly increased the convergence rate on trajectory optimization problems as shown in Section 7.

Algorithm 1: Parallel Noisy Line Search

```
   Param: $\Theta, \Delta\Theta, \alpha = [0.01, ...]$
1  $\Theta_l \leftarrow \text{clip}(\Theta + \alpha\Delta\Theta)$          ▹ get bounded variables
2  $c_0, c_l \leftarrow c(\Theta_0), c(\Theta_l)$          ▹ compute cost for magnitudes
3  $\delta\Theta_l \leftarrow \delta c_l$          ▹ compute batched gradients
4  $a \leftarrow c_l \le c_0 + \eta_1\alpha\Delta\Theta$          ▹ Armijo Condition
5  if Wolfe then
6      if Strong then $a_2 \leftarrow \text{abs}(\delta\Theta_l\alpha) \le \eta_2\Delta\Theta$
7      else $a_2 \leftarrow \delta\Theta_l\alpha \ge \eta_2\Delta\Theta$
8      $a \leftarrow a \mathbin{\&\&} a_2$
9  end
10 $i \leftarrow \text{largest\_true}(a)$          ▹ returns 0 when none is true
11 $\hat{c}, \hat{\Theta}, \delta\hat{\Theta} \leftarrow c_l[i], \Theta_l[i], \delta\Theta_l[i]$          ▹ return best
```

4.2 Particle-Based Optimization

To encourage L-BFGS to reach a local optima, we devise a strategy combining particle and gradient-based optimization. This is inspired by strong theoretical results in stochastic gradient Markov Chain Monte Carlo [47, 48], and sampling-based MPC controllers such as MPPI [49]. In our method, we first run a few iterations of particle-based optimization over the initialization before sending to L-BFGS. Given an initial mean trajectory of joint configurations $\Theta_\mu = \theta_{[1,T]}$ and a covariance Θ_σ, we sample n particles $\theta_{n,[1,T]}$ from a zero mean Gaussian and then update $\theta_{n,[1,T]} = \Theta_\mu + \sqrt{\Theta_\sigma} * \theta_s$. We compute the cost for these particles $C(\theta_{n,[1,T]}) \in \mathbb{R}^n$ and calculate the exponential utility $w = \frac{e^{c_i}}{\sum_{i=0}^n e^{c_i}}$, where $c = \frac{-1.0}{\beta}C(\theta_{n,[1,T]})$. We then update the mean and covariance as,

$$\Theta_\mu = (1 - k_\mu)\Theta_{\mu-1} + (k_\mu) w * \theta_{n,[1,T]}, \tag{12}$$

$$\Theta_\sigma = (1 - k_\sigma)\Theta_{\sigma-1} + w * (\theta^2_{n,[1,T]} - \Theta_{\mu-1}). \tag{13}$$

We found that the use of particle-based optimization to initialize L-BFGS led to better convergence as empirically validated in Sec. 7. To tackle very hard problems and further reduce the number of seeds required to converge, we develop a parallelized geometric planner that generates collision-free geometric paths between start and goal in the next section.

5 Parallel Geometric Planner

We develop a geometric planner to generate a collision-free path from the start configuration θ_0 to the goal configuration θ_T. This generated path is specified by a list of w waypoints $\theta_{[0,w]}$ through which the robot passes in a linear fashion. By studying common geometric planning methods [3], we found three main components in graph building that can benefit from parallel compute. Specifically, sampling collision-free nodes, finding k nearest nodes in graph, and steering from each sampled node to k nearest nodes. We implement algorithms to perform these tasks in parallel on the GPU in our geometric planner.

Algorithm 2: Parallel Geometric Planner

```
    Data  : $\theta_{b,0}$, $\theta_{b,g}$
    Param: $g_{max}$, $g_{refine}$, $c_{max}$,$c_{default}$, $k_{refine}$, $k_{explore}$, $p_{init}$, $p_{refine}$, $p_{explore}$
    Result: path_found, path($\Theta_{b,[0,w]}$)
    Init  : $k_n \leftarrow k_{explore}$, $c_{max} \leftarrow c_{default}$, $p_n \leftarrow p_{explore}$, $i \leftarrow 0$
 1  e = [[$\theta_{b,0}$, $\theta_{b,g}$], [$\theta_{b,g}$, $\theta_{b,0}$], [$\theta_{b,0}$, $\theta_r$], [$\theta_r$, $\theta_{b,0}$], [$\theta_{b,g}$, $\theta_r$], [$\theta_r$, $\theta_{b,g}$]]
 2  steer_connect(e)                              ▷ Connect start, goal, and retract
 3  path_found, path, min_len ← shortest_path($\theta_{b,0}$, $\theta_{b,g}$)
 4  if path_found then
 5      path, min_len, $c_{max}$ ← shortcut_path($\theta_{b,0}$, $\theta_{b,g}$)
 6      if min_len == 2 then return path
 7  end
 8  $c_{min} \leftarrow dist(\theta_{b,0}, \theta_{b,g})$
 9  while not path_found or $i < g_{refine}$ do
10      $id$ ←random(!path_found)     ▷ Pick an index from the set of queries that do not have a path yet.
11      $\theta_{s,k}$ ← sample_nodes($\theta_0$, $\theta_g$, $c_{max}$, $p_n$)       ▷ sample nodes within ellipse
12      $e \leftarrow near(k_n, \theta_{s,k})$      ▷ Find $k_n$ nearest samples $\theta_{s,k}$ to existing nodes in graph
13      steer_connect(e)                          ▷ Steer and connect to graph
14      path_found, path, min_len ← shortest_path($\theta_{b,0}$, $\theta_{b,g}$)
15      $i+ = 1$
16      if path_found && min_len > 3 then
17          path, min_len, $c_{max}$ ← shortcut_path(path)
18      else
19          $c_{max}[id] \leftarrow c_{max}[id] + c_{min}[id] * \eta_{explore}$
20          $p_n + = \eta_{explore} * p_n$
21          $k_n + = \eta_{explore} * k_n$
22      end
23  end
24  return path_found, path
```

Our geometric planner as shown in Alg. 2, first performs heuristic planning by checking if we can steer from start to goal configuration directly [50] or through a predefined retract configuration θ_r (lines 1-7). If this heuristic fails, we sample collision-free configurations v_{new} from an informed search region that samples within c_{max} of the straight line distance between start and goal similar to BIT* [51] (line 11). We then find the k_n nearest neighbours from the existing graph and try to steer from the graph nodes to the new vertices (lines 12-13). We repeat these steps until we find a path with only one waypoint (line 16). Between re-attempts we grow the number of sampled nodes p_n, the number of nearest neighbours k_n, and the search region c_{max} to grow the exploration space of the geometric planner (lines 19-21). To efficiently leverage parallel compute in geometric planning, we develop an algorithm to steer from s vertices $\theta_{s,0}$ in a graph with s sampled new configurations $\theta_{s,k}$ in parallel as described in Alg. 3.

Algorithm 3: Parallel Steering

```
   Input: $e = [\theta_{s,0}, \theta_{s,k}]$
   Parameters: r, $d_w$
1  $\vec{g} \leftarrow distvec(\theta_{s,0}, \theta_{s,k})$          ▷ distance between nodes
2  $n \leftarrow max(|\vec{g}|/r) + 1$                               ▷ find largest distance
3  $\vec{d} \leftarrow d[: n+1]/n$                                   ▷ discretize based on largest distance
4  $\vec{l} \leftarrow \theta_{b,0} + \vec{d} * \vec{g}/d_w$          ▷ get disretized edges
5  $mask \leftarrow \text{mask\_samples}(\vec{l})$                   ▷ check for validity
6  $h \leftarrow \text{first\_false}(mask) - 1$                      ▷ first collision index/edge
7  $h[h == -1] \leftarrow n$
8  $v_{new} \leftarrow l[h]$                                         ▷ store last valid point/edge
9  $d \leftarrow \text{dist}(\theta_{s,0}, v_{new})$                 ▷ store distance value in edge
10 $\text{graph\_add}(\theta_{b,0}, v_{new}, d)$
```

We also implement a `shortcut_path()` function that tries to connect each waypoint with every other waypoint in the path to try to find a shorter path. We leverage this geometric planner to also find paths between a batch of start and goal configurations or from a single start to a goal set. We achieve this by randomly choosing a query index (for which a path does not exist yet) and sampling in this query region to expand the graph (lines 10,11).

6 Results

We validate and compare our approach to existing methods on two motion planning datasets, the motion-benchmaker dataset [52] containing 800 problems and the mpinets dataset [53] containing 1800 problems. Both datasets contain motion planning problems for the Franka Emika Panda robot, which has 7 actuated joints. The datasets span 12 unique scene types, with each problem starting the robot at a collision-free joint configuration and defining the goal as a desired end-effector pose. A few instances of the motion planning problems from this set is shown in Fig. 5. We provide the planning problems along with code to compute different metrics at github.com/fishbotics/robometrics.

We first analyze the quality of solutions in Section 6.1, then compare compute times in Section 6.2. We also analyze our collision-free inverse kinematics solver in Section 6.2.4, followed by an analysis of our kinematics and distance query modules in Section 6.2.5, as they can also be used independently in other manipulation tasks.

6.1 Motion Generation Quality

We focus analysis of motion generation quality to metrics that affect the success rate, and execution time of the computed motions. We compute six different metrics that capture geometric and temporal qualities of the generated trajectories. First, we compute the standard metrics from geometric planning methods, specifically *Success* on a dataset within a given time and *C-Space Path Length* which is the distance traveled by the robot's joints to reach the target pose. In addition, we introduce four metrics that evaluates time parameterization of the generated motions. The *Motion Time* metric compares the trajectory times given the number of trajectory points n and the time dt between waypoints (i.e., $(n-1)*dt$). If a robot perfectly tracks the planned trajectory, then this *Motion Time* would be the execution time. When moving manipulators at high-speed, they become very sensitive to jerk profiles, especially when starting from or ending at zero velocity (i.e., idle). This has prevented motion generation approaches from executing at the full rated speed of a robot. We hence introduce a metric that measures the maximum jerk across the trajectory, which we call *Maximum Jerk* metric. We also measure the *Maximum Acceleration* and *Mean Velocity* across the trajectory to draw further comparisons between methods.

We compare our method to Tesseract [20] which uses Bullet's continuous collision detector [24], Open Motion Planning Library (OMPL) [54] for geometric planning, and TrajOpt [9] for trajectory optimization.

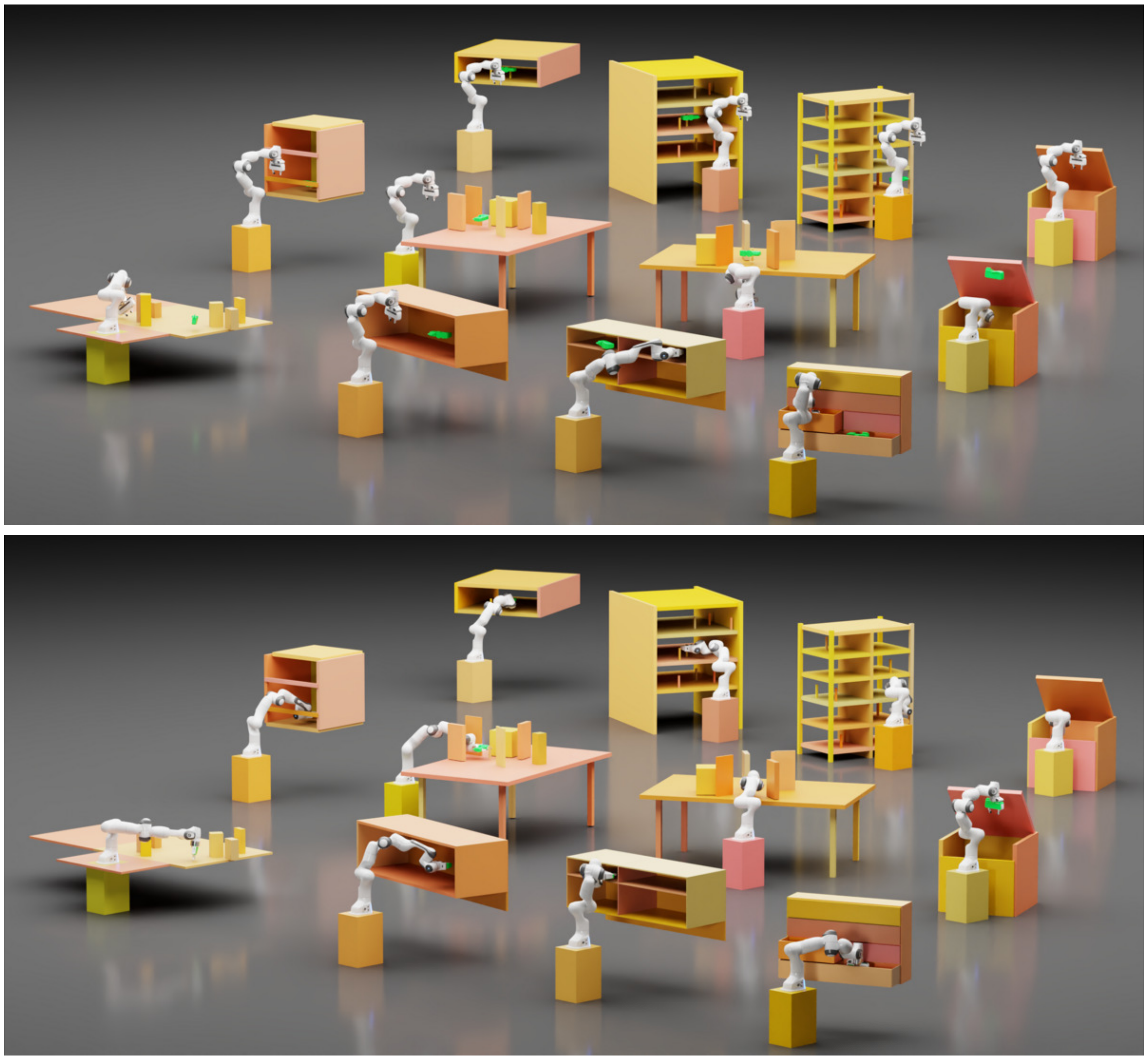
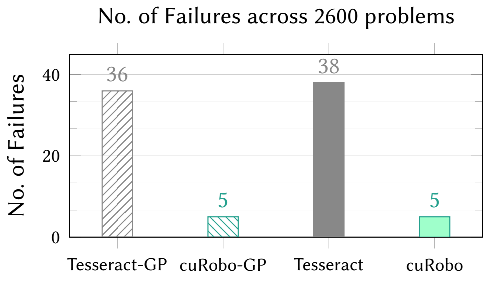


Figure 5: Motion planning problems across the 12 different scenes are visualized here. The top image shows the robot in the start configuration and the bottom image shows the robot at a final configuration after running cuRobo's motion generation. The top two rows show scenes from the motion benchmaker dataset [52] and the bottom row shows the scenes from motion policy networks [53]

We use two motion planning methods from Tesseract, one that only performs geometric planning with RRTConnect [55] which we call *Tesseract-GP* and one that uses the geometric plan as a seed to Trajopt for trajectory optimization which we call *Tesseract*.[2] We compare these baselines to two versions of motion generation from cuRobo, one that only does geometric planning *cuRobo-GP* using our algorithm from Section 5 and the other that does Trajectory Optimization *cuRobo*. For all methods, we timeout at 60 seconds and allow random restarts until this timeout is reached. More details on the baselines and the evaluation methods are available in Appendix B.

We report the success across the 2600 motion planning problems in Figure 6. We count a trajectory as *Success* when it is collision-free, it doesn't violate any joint limits, and is within 5mm and 5% of desired position and orientation respectively. We found our geometric planner *cuRobo-GP* to find a path on 99.8% of the dataset compared to *Tesseract-GP*'s 98.6%. Our geometric planner only failed on 5 problems compared to *Tesseract-GP* failing on 36 problems. When we compare trajectory optimization methods, *cuRobo* only failed on 5 problems while *Tesseract* failed on 38 problems, giving a success rate of 99.8% and 98.53% respectively. *cuRobo* and *cuRobo-GP* failed on the same set of five problems. When we look at these problems in Fig. 7,

[2]We also tried Tesseract's TrajOpt integration with a linear seed but it failed to find solutions on most problems.



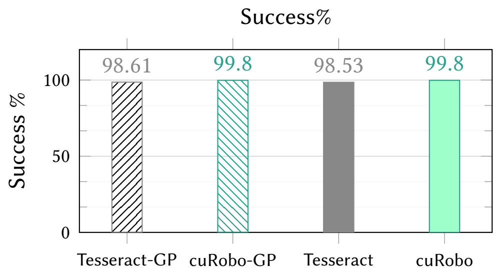


Figure 6: We plot the success of different methods on the motionbenchmaker and mpinets dataset in these plots. On the left we plot the number of failures on our evaluation dataset which contains 2600 problems. *cuRobo* only fails on 5 problems which are all invalid problems when evaluated with *cuRobo*'s collision representation. On the right plot, we see that *cuRobo* has a higher success % than *Tesseract*.

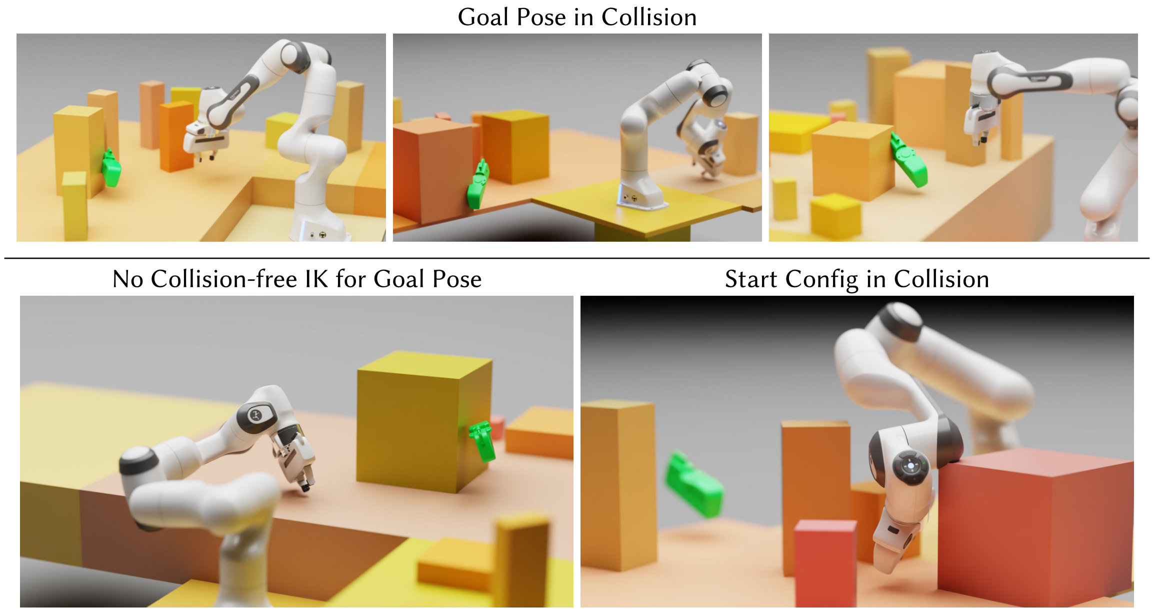


Figure 7: *cuRobo* failures in the dataset are visualized here. We found 3 instances the goal pose be in collision as shown in the top row. We found one instance where the all IK solutions to the goal pose was in collision as shown in the bottom left image and one instance where the start joint configuration was in collision as shown in the bottom right image.

we see that 3 problems have their goal pose in collision with the world, 1 problem as the start configuration in collision, and one problem does not have a collision-free IK solution to the goal pose. The reason for these problems to be invalid for *cuRobo* is because we evaluate *cuRobo* with our cuboid collision checker which approximates the obstacles represented by cylinders as cuboids. While *cuRobo* also has mesh-based collision checking and depth camera based collision checking, we leave evaluating these collision checkers for a future work.

Next, we plot *C-Space Path Length* across the four methods in Figure 8. First, we observe that our geometric planner *cuRobo-GP* has shorter path length than *Tesseract-GP* which uses RRTConnect. *cuRobo-GP*'s path lengths are [5.39, 7.13, 13.45] radians on mean, 75th, and 98th percentile of the dataset compared to *Tesseract-GP*'s [7.1, 8, 17.2] radians. We then observe methods that use trajectory optimization, *cuRobo* and *Tesseract* have shorter path length than geometric planning methods *cuRobo-GP* and *Tesseract-GP*. *Tesseract* reduces paths on average by 48% when compared to *Tesseract-GP*. *cuRobo* reduces the path length by 53% on average when compared with *Tesseract-GP*. *cuRobo* reduces path length by 53%, 39%, and 10% on average when compared to *Tesseract-GP*, *cuRobo-GP*, and *Tesseract* respectively. When we compare *cuRobo* with *Tesseract*, we found that *cuRobo*'s paths are 26% shorter than *Tesseract* on the 98thpercentile of the dataset.

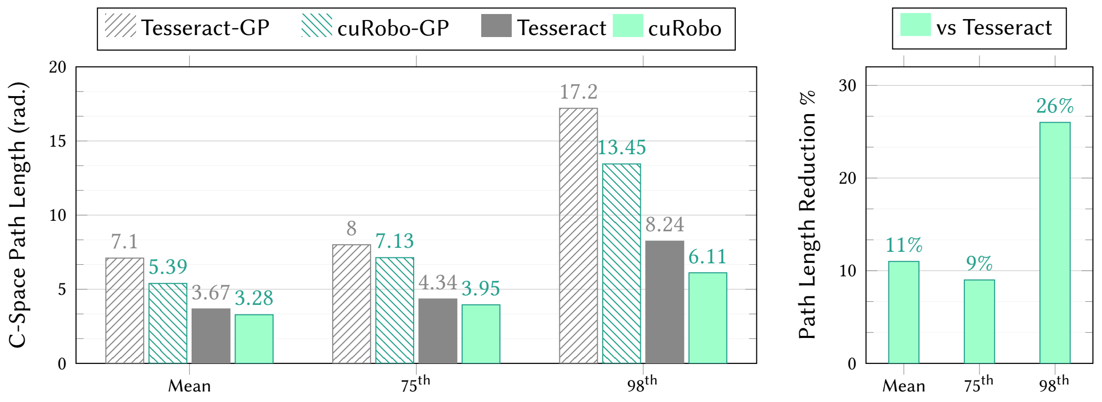


Figure 8: We compare the *C-Space Path Length* across different methods in the above plots. On the left, we see that *cuRobo* has shorter path lengths than all other methods and *cuRobo-GP* has shorter path than *Tesseract-GP*. On the right, we plot the reduction in path length with *cuRobo* over *Tesseract* across our test dataset.

From the geometric planning metrics, we find that trajectory optimization methods have comparable success percentage to geometric planning methods while also providing shorter path lengths than RRTConnect-like planners. While optimal geometric planners can generate shorter paths than RRTConnect, they require either more planning time or task-specific heuristics as shown in Appendix C.1. Our goal is not only to get shorter paths, but also trajectories that can be executed on robots with minimal post-processing. Geometric planning methods require time parameterization as a post-processing step to be able to execute on robots. Kunz and Stilman developed a bounded velocity and acceleration time-parameterization technique [4] that is extensively used in the robotics community, including MoveIt [56]. However, this time parameterization technique does not bound the jerk along the trajectory and as such can have very large jerks. We could not find any accessible software library that can post process geometric paths while bounding jerks, making geometric path planning not directly deploy-able on jerk sensitive manipulators. We hence only compare between *Tesseract* and *cuRobo* on the time parameterization metrics. We additionally take the trajectories obtained from *Tesseract* and post process with Kunz and Stilman's method, which we call *Tesseract-TG*. We add this to our comparisons to highlight the improvements we can get with trajectory optimization techniques, especially with *cuRobo*'s minimum jerk formulation.

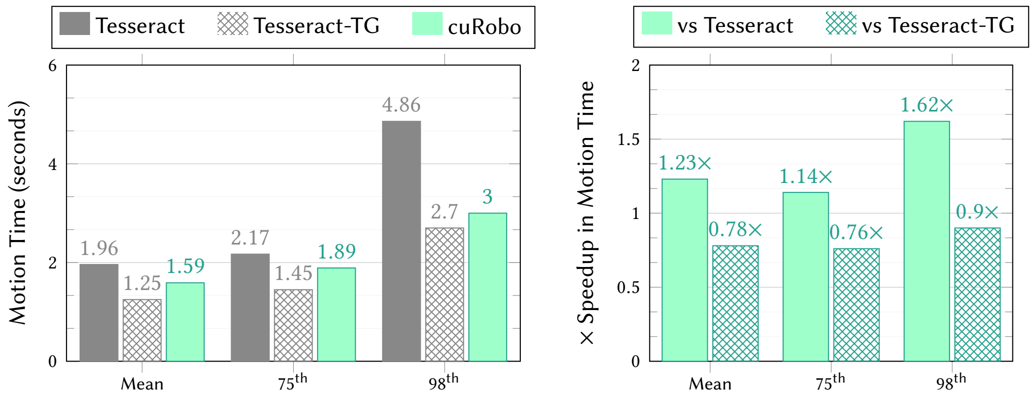


Figure 9: We compare the *Motion Time* (i.e., length of the trajectory in time) generated by *cuRobo* with *Tesseract* and also to *Tesseract-TG* which post processes the trajectory with a time-optimal reparameterization developed by Kunz and Stilman [4]. On the left plot, we see that *cuRobo* generates trajectories that are within 3 seconds on the 98thpercentile while *Tesseract* generates significantly slower trajectories at 5 seconds. On the right plot, we observe that *cuRobo* obtains trajectories that are 1.62× faster than *Tesseract*. *cuRobo* is within 0.9× of *Tesseract-TG* motion time while having 9× lower jerk.

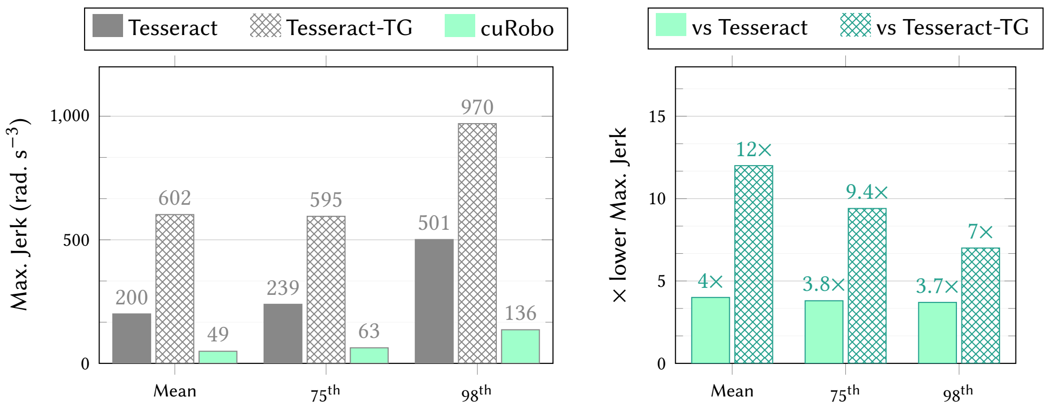


Figure 10: The maximum jerk along the trajectory across all joints is shown in the plots. On the left plot, we see that *cuRobo* has a maxumum jerk of 136 rad.s^{-3} on the 98th percentile of the dataset while *Tesseract* has 501 rad.s^{-3} a 7x higher value as shown by the plot on the right.

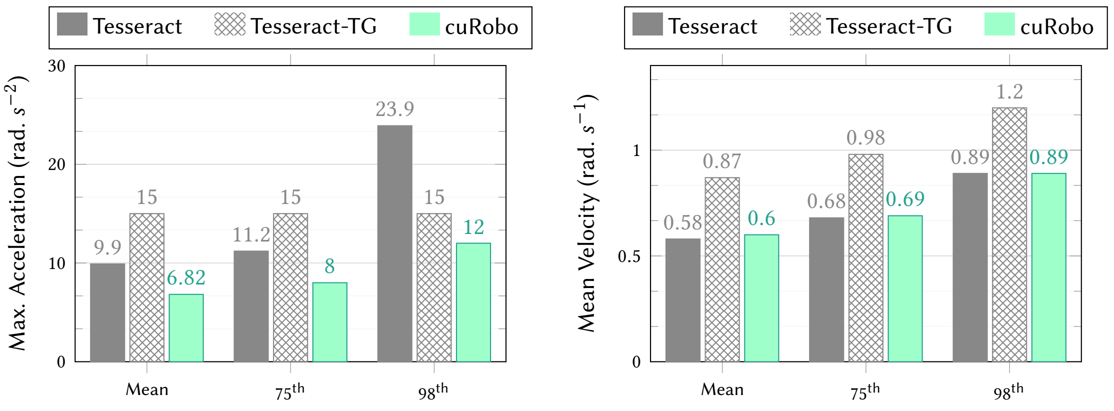


Figure 11: The maximum acceleration and mean velocity across the dataset is plotted for the different methods. *cuRobo* has the smallest acceleration (the left plot) across the methods while having similar mean velocity to *Tesseract* on the right plot.

The first in time parameterization metrics is *Motion Time* which we plot in Figure 9. *cuRobo*'s trajectories take [1.59, 1.89, 3] seconds compared to Tesseract's [1.96, 2.17, 4.86] seconds on the mean, 75th, and 98th percentiles. *cuRobo* produces trajectories that have a 1.23x lower mean and 1.62x lower 98th percentile motion time when compared to motions generated by *Tesseract*. This large reduction in motion time leads to *cuRobo*'s 98th percentile being 2.86 seconds quicker than *Tesseract*. When we compare *cuRobo*'s solutions to *Tesseract-TG* which uses time-optimal reparameterization, *cuRobo* generates trajectories that are 0.3 seconds slower both on average and 98th percentile of the dataset. This slow down is because *cuRobo* also optimizes for minimum-jerk, leading to trajectories with 12x lower jerk on average compared to *Tesseract-TG* as shown in Fig. 10. When comparing to *Tesseract*, we generate trajectories that have 4x lower jerk on average as *Tesseract* doesn't minimize jerk.

We compare the mean velocity and maximum acceleration between methods in Figure 11. *Tesseract-TG* has the largest values in mean velocity and maximum acceleration as Kunz and Stilman's time parameterization technique by design attempts to reach peak velocity by instantly jumping to maximum acceleration. *Tesseract* has larger max acceleration when compared to *cuRobo* as it doesn't have to optimize for jerk and as such can instantaneously change acceleration along the trajectory without any penalties. We also observed that *Tesseract* has a 98th percentile maximum acceleration of 23.9 rad.s^{-2} which is beyond the 15 rad.s^{-2} acceleration limit we set for the Franka Panda robot. *cuRobo* has the smallest maximum acceleration across the dataset as we minimize jerk across the trajectory, which penalizes instantaneous changes to acceler-

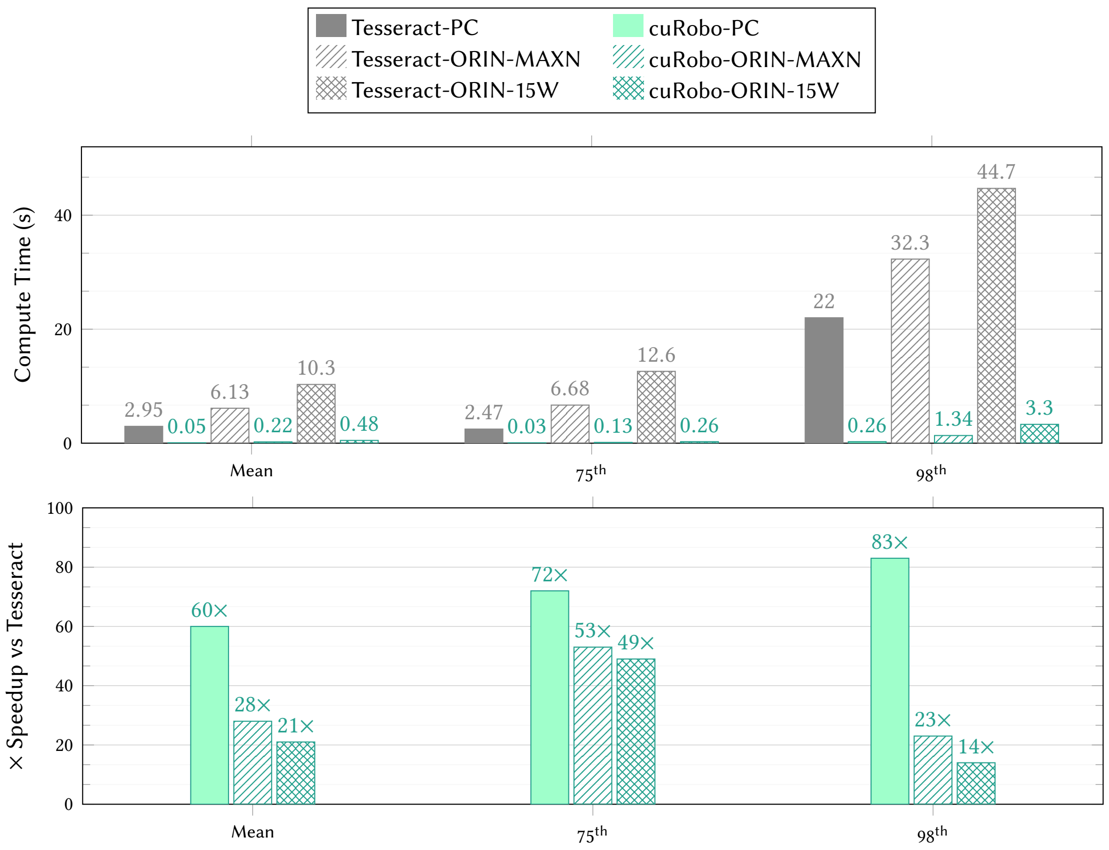


Figure 12: We compare the compute time for motion generation between *cuRobo* and *Tesseract* across three compute platforms. On all of the 2600 motion planning problems, we found *cuRobo* to take the least time, getting a 60× speedup on average on a desktop pc with NVIDIA RTX 4090 and AMD Ryzen 9 7950x, with a 83× speedup on the 98th percentile.

ation. Even with the smallest maximum acceleration, *cuRobo*'s mean velocity is comparable to *Tesseract*, slightly higher in the mean by 0.02 rad.s^{-1} and 0.01 rad.s^{-1} in the 75th percentile.

Summarizing across these metrics, *cuRobo* produces paths that are shorter in path length than any other method while also succeeding on all feasible problems in the dataset. In addition, *cuRobo* generates trajectories that have the least jerk, 4x lower than existing trajectory optimization techniques, and 12x lower than existing time parameterization methods. Trajectories generated with *cuRobo* also have motion times 1.23x lower than existing trajectory optimization techniques and is within 0.3 seconds of high-jerk time parameterization methods. We will next analyze the compute time taken by *cuRobo* to obtain these trajectories.

6.2 Compute Time

We calculate the compute time on three platforms, a PC with an AMD Ryzen 9 7950x CPU and NVIDIA RTX 4090 GPU, and an NVIDIA Jetson AGX Orin 64GB system configured to operate at MAXN (60W) and 15W power budgets. We measure runtime using Python's *time* utility after synchronizing device and host.

6.2.1 Motion Generation

The time it takes to compute motions using *cuRobo* is compared to *Tesseract* in Figure 12 across all three platforms. We observed that *Tesseract* takes on average 2.95 seconds compared to *cuRobo* taking 50ms, leading to a 60× speedup in motion planning. The gap in planning time increases at the 75th percentile of evaluation set, *cuRobo* taking 30ms to plan while *Tesseract* takes 2.47 seconds, leading to a 72× speedup in planning with our proposed method. On the 98th percentile of the dataset, *cuRobo* takes 260 milliseconds while *Tesseract* takes 22 seconds, giving *cuRobo* 83× speedup in planning. The difference in planning time

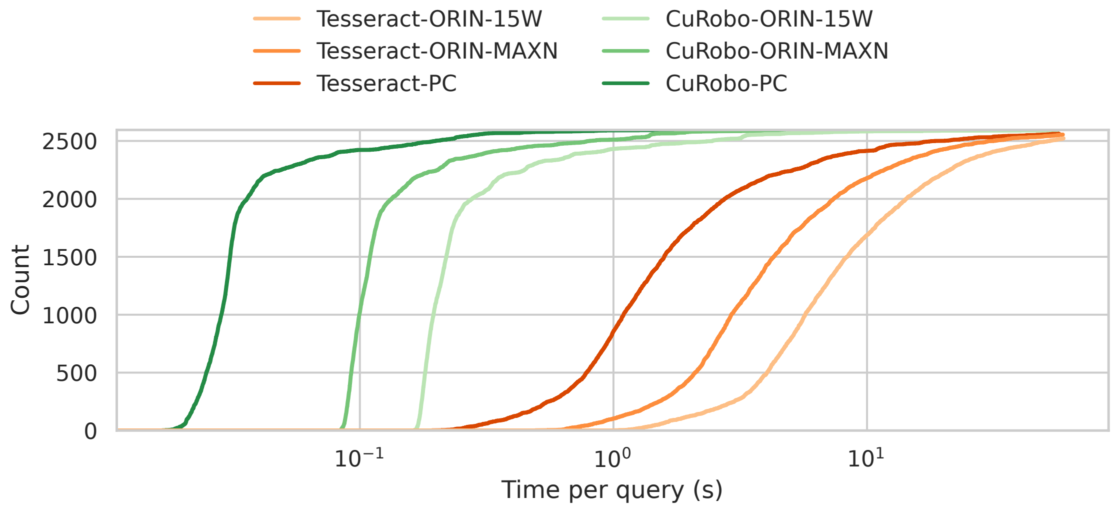


Figure 13: Motion generation time across compute platforms for *Tesseract* and *cuRobo* is shown with the x-axis in logarithmic scale to better highlight the difference in time across the dataset. *cuRobo* at 15W on the NVIDIA Jetson ORIN AGX (cuRobo-ORIN-15W) is faster than *Tesseract* on a full desktop PC with an i7 processor (Tesseract-i7) as seen by large gap in x-axis across the full dataset. In addition, we see that *cuRobo* slows down on the Jetson compared to a NVIDIA RTX 4090.

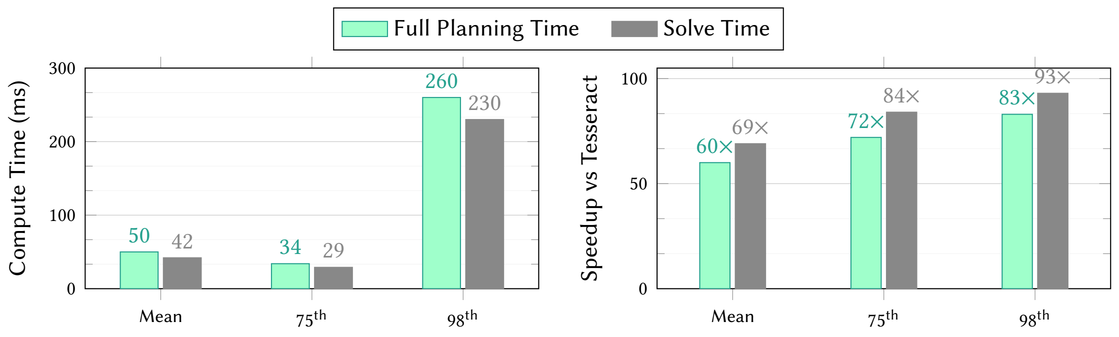


Figure 14: We measure the time taken by *cuRobo* in the optimization iterations *Solve Time* and compare it to the time taken by the full pipeline. As our library is implemented in python, we take measurable amount of time outside of the solver iterations which could be reduced by rewriting in a compiled programming language.

across the mean, 75th, and 98th is because *cuRobo* calls the geometric planner only after three failed attempts with linear seeds.

The speedup over *Tesseract* scales to the NVIDIA Jetson ORIN as well, in both power modes as shown in Figure 12. *cuRobo* takes 0.22 seconds and 0.48 seconds on average on at MAXN and 15w while *Tesseract* takes 6.13 seconds and 10.3 seconds respectively. On average, *cuRobo* is 28× and 21× faster than *Tesseract* at MAXN and 15W respectively. We also obeserved that *cuRobo* is faster on a NVIDIA Jetson ORIN at 15W than *Tesseract* running on a desktop PC as shown in Figure 13.

Our approach is implemented in python with key compute kernels in CUDA C++ called through python wrappers. We reduced the python overhead and the overhead of repeatedly launching CUDA kernels by recording optimization iterations and the `mask_samples` function in geometric planning (see Algorithm 3) in CUDA Graphs. We then replay the recorded CUDA Graphs with data from new planning problems. This use of CUDA Graphs reduced our planning time by 10x compared to calling the kernels individually from

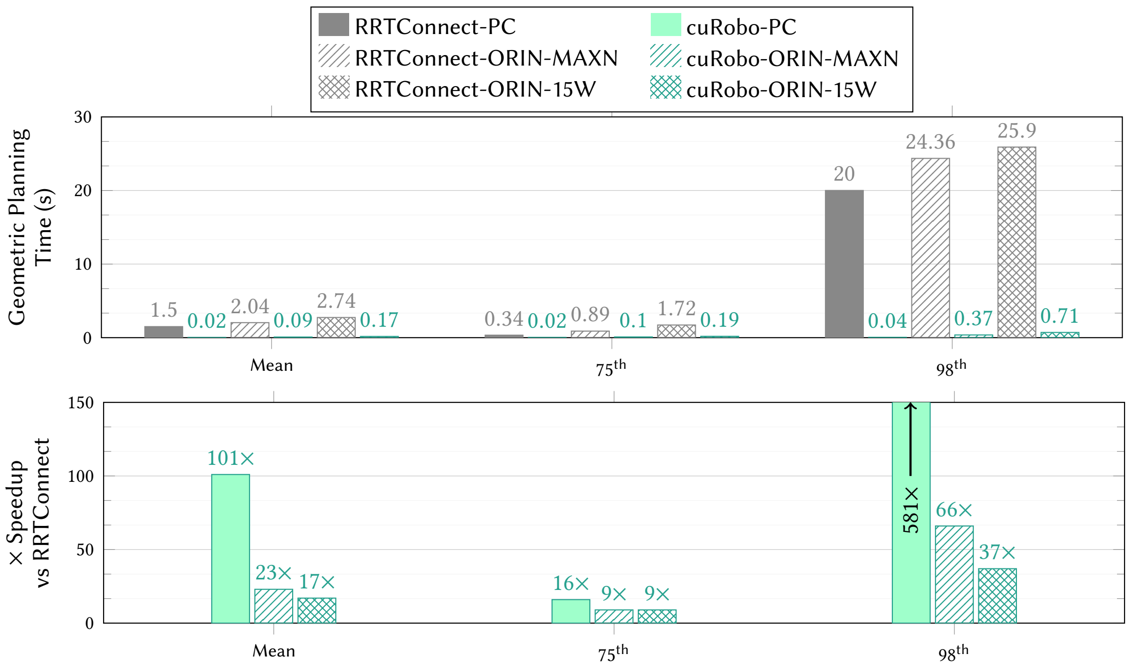


Figure 15: Our GPU accelerated geometric planner is 101× faster on average compared to OMPL's implementation of RRTConnect available in *Tesseract*. We observed an average speedup of 23× and 17× on the NVIDIA Jetson Orin at MAXN and 15W modes respectively. We see a much larger speedup on the 98th percentile of the dataset across the compute platforms as our GPU accelerated graph builder is able to explore the workspace in parallel while a CPU based approach scales linearly with edge validation.

python. Our implementation still has some components in python, calling many small cuda kernels to setup the optimization problems, and also to get the final result from the many parallel seeds. We timed these parts and found that *cuRobo* spent 8ms, 5ms, and 30ms on the mean, 75th, and 98th percentile as shown in Figure 14. The percentage of time spent in these steps compared to the solver time was 15%, 15%, and 12% on average, 75th, and 98th percentiles. We leave speeding up these components by rewriting directly in C++ and fusing the CUDA kernels to future work. We do share the speedup that could be gained if these components are optimized by only comparing the iterations time and geometric planning time to *Tesseract* in Figure 14. We see that our speedup of 60× becomes 69× on mean, 72× becomes 84× on 75th percentile, and 83× becomes 93× on the 98th percentile.

6.2.2 Geometric Planning

We compare the compute time in geometric planning between our GPU accelerated geometric planner introduced in Section 5 which we call *cuRobo-GP* to OMPL's RRTConnect implementation in Tesseract, which we call *Tesseract-GP* in Figure 15. *Tesseract-GP* takes 1.5 seconds on average while *cuRobo-GP* takes 0.02 seconds leading to a 101× speedup in geometric planning with *cuRobo-GP*. Looking at the 98th percentile planning time, *Tesseract-GP* takes 20 seconds while *cuRobo-GP* takes 0.04 seconds, giving us a 581× speedup. A very recent work from Thomason *et al.* [57] that explores vectorized geometric planning leveraging SIMD on CPU. The results from their paper show that it takes 0.1ms (mean) to plan on the motion benchmaker dataset. However, their code is not available at the time of this publication and we leave comparing to it for a future work.

6.2.3 Trajectory Optimization

We compute the time it takes to perform trajectory optimization in Figure 16 between *cuRobo*'s GPU accelerated approach and TrajOpt [9] implementation leveraged by *Tesseract*. We see that *cuRobo* is 87× and 145× faster in optimization than TrajOpt on average and 75th percentile respectively. *cuRobo* takes a mere 10ms to perform trajectory optimization compared to TrajOpt taking 1.79 seconds on 75th percentile of the

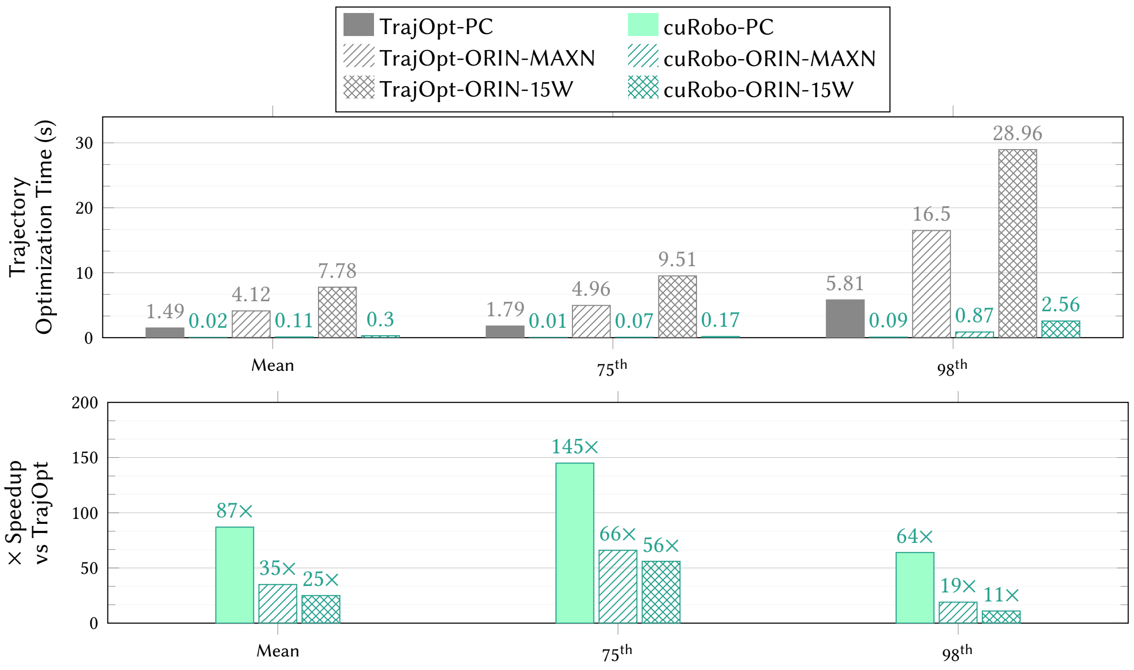


Figure 16: We compare our trajectory optimization to *TrajOpt* which is integrated in *Tesseract* for collision-free trajectory generation. On average, our approach is 87× faster than *TrajOpt* on a desktop PC. On a NVIDIA Jetson ORIN, our appraoch is 35× and 25× faster at MAXN and 15W modes.

evaluation set. We get speedups of 23× and 17× on Jetson device as well, taking 0.09 and 0.17 seconds on average on ORIN at MAXN and 15W respectively. These speedups are interesting because numerical optimization is predominantly iterative where we run many sequential iterations until convergence. These sequential iterations can make the entire computation graph memory access heavy. Our efficient parallelization of compute across the whole pipeline enables us to get these speedups. We not only run each seed of optimization in seperate threads but also split many of our workload heavy kernels across many threads on a per seed basis.

While these timings are based on optimizing collision-free trajectories, we hope that these speedups encourage roboticists to leverage *cuRobo*'s trajectory optimization implementation for other robotics tasks.

6.2.4 Inverse Kinematics

We compare the performance of our inverse kinematics solver against TracIK [23]. We perform two sets of experiments: (1) without collision checking and (2) with self and environment collision checking. We chose an instance of the *bookshelf-small-panda* scene from motion benchmaker [52] for the collision checking experiment. We sample feasible joint configs from a Halton Sequence and average the results across 5 trials for different batch sizes. Since TracIK does not account for collisions, we perform rejection sampling with PyBullet, allowing 10 reattempts. For all cuRobo IK queries, we run 30 seeds in parallel and return the best solution from these seeds. We evaluate IK with 5 different batch sizes – 1, 10, 100, 500, and 1000. For a single query (batch size=1), cuRobo takes 2.7ms while TracIK only takes 0.9ms. However, as we increase the batch size of IK queries, we see a speedup starting from a batch size of 10.

For the standard IK problem, we can generate 37134 solutions per second when we use a batch size of 1000 while TracIK can only generate 1590 solutions, 23.4× slower than our method. When we compare collision-free IK, our method (*cuRobo-Coll-Free*) can compute 7611 compared to rejection sampled TracIK (*TracIK-Coll-Free*) which can only obtain 95 collision-free solutions per second in our experiments, 80x slower than our approach to collision-free IK as shown in Fig. 37-B. We also found that rejection sampling approach to collision-free IK failed on 20% of the problems tested. BioIK [58] reports that their approach can solve IK in 0.7ms (1428 solutions per second), it is not clear from their paper whether the runtime includes collision-free IK. Even if we consider their timing to be for collision-free IK, our method is still faster starting from a batch size of 10, taking 0.48ms per solution.

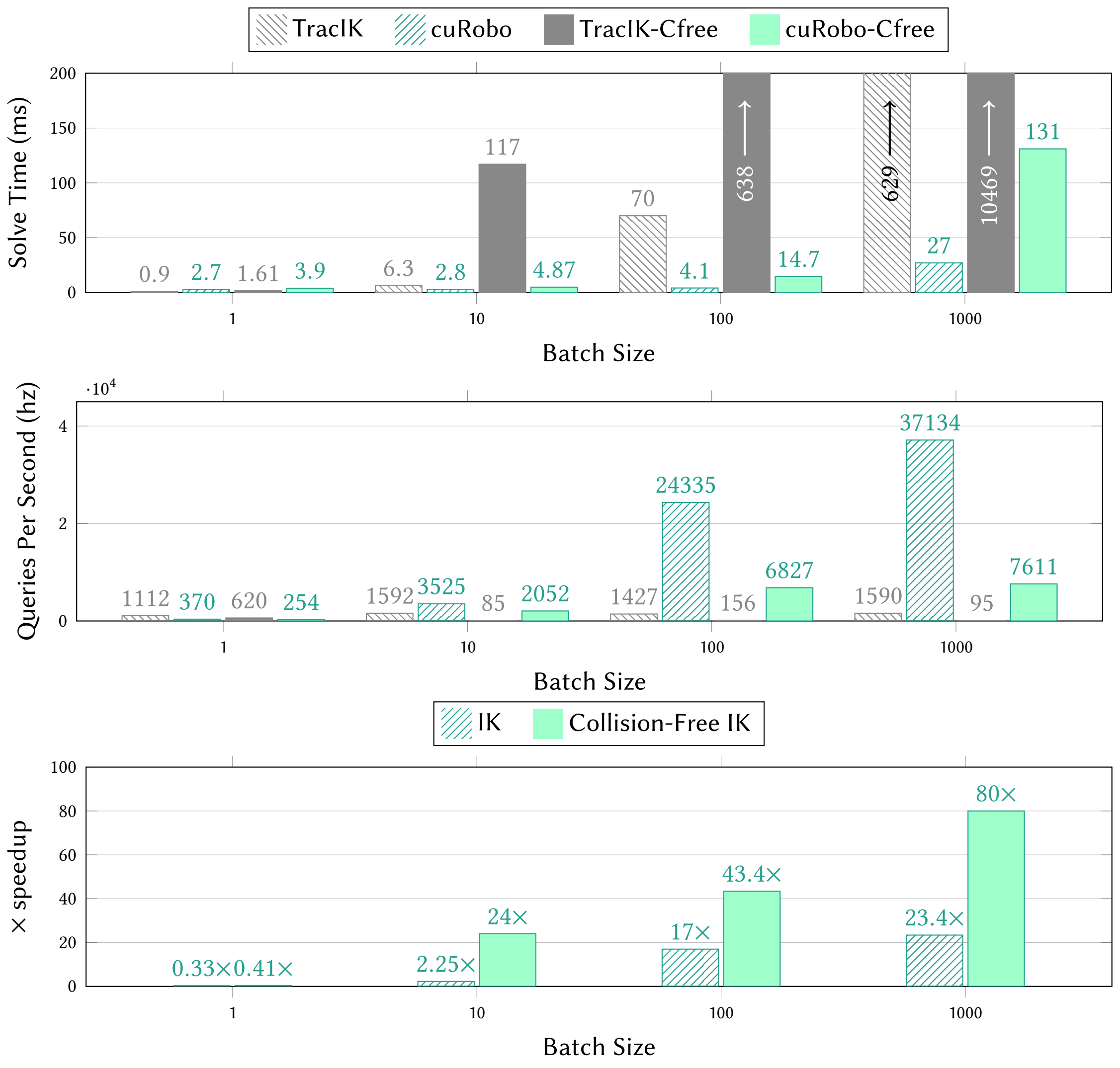


Figure 17: We compare the compute time in solving inverse kinematics (IK) between *cuRobo* and TracIK [23] across different batch sizes. *TracIK* is able to solve 1000 poses in 629ms while *cuRobo* only takes 27ms, 23.4× faster than *TracIK*. In solving collision-free IK, *cuRobo* solves 1000 poses in 131ms, 80× faster than rejection sampled *TracIK*.

6.2.5 Kinematics and Distance Queries

One of the major breakthroughs in accelerating our motion generation approach was on developing a parallel compute friendly implementation of robot kinematics. Most common manipulators have many serially connected links, making computation of kinematics a largely serial operation. Existing SOTA methods for forward kinematics on CPUs such as pinnochio [60] take 1μs on average for 7-dof robots while GPU accelerated kinematics implemented in PyTorch such as STORM [61] outmatch CPU methods only at a batch size of 1000. STORM improves upon implementation from Meier *et al.* [62] by keeping buffers in memory between calls without recreating them. This slowdown in GPU based kinematics is because existing implementations use many CUDA kernels to perform kinematics, e.g., STORM runs through 125 CUDA kernels to compute kinematics. In *cuRobo*, we implement the entire kinematics in a single CUDA kernel, discussed in Appendix E. This enables our approach to outmatch pinnochio's performance at a batch size of 100 as shown in Figure 19. We also compare our kinematics implementation with KDL's implementation which is used by traciky. We additionally show the improvement with CUDA Graphs in calling GPU methods by adding a suffix "-CG".

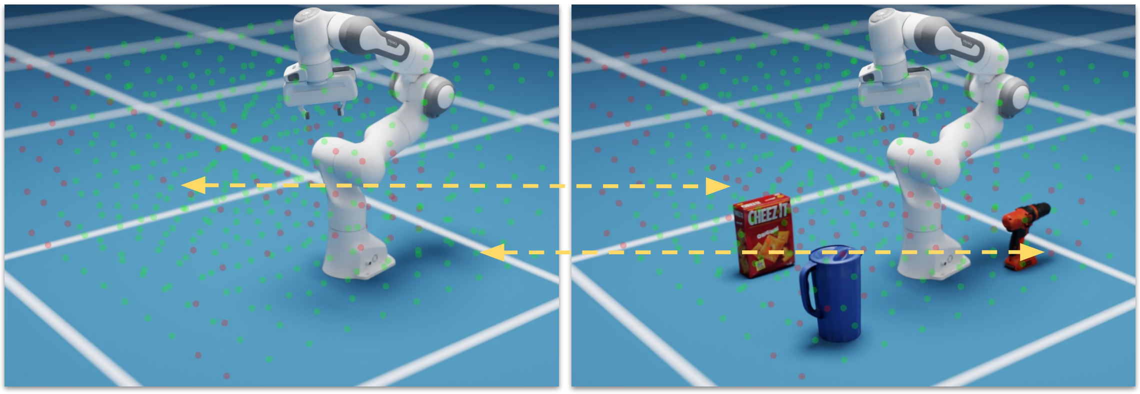


Figure 18: We show reachability analysis as an application of fast batched inverse kinematics in these images. We sample 500 poses in a grid shown by red and green spheres, and query *cuRobo*'s batched inverse kinematics solver for joint configurations to match these poses. We color the spheres as red if IK was unsuccessful and green if successful. *cuRobo* is able to run at 15Hz while also sharing the GPU resources with NVIDIA Isaac Sim. On the right we show our solver also reasoning about world collisions and marking poses near objects with red as they are not collision-free.

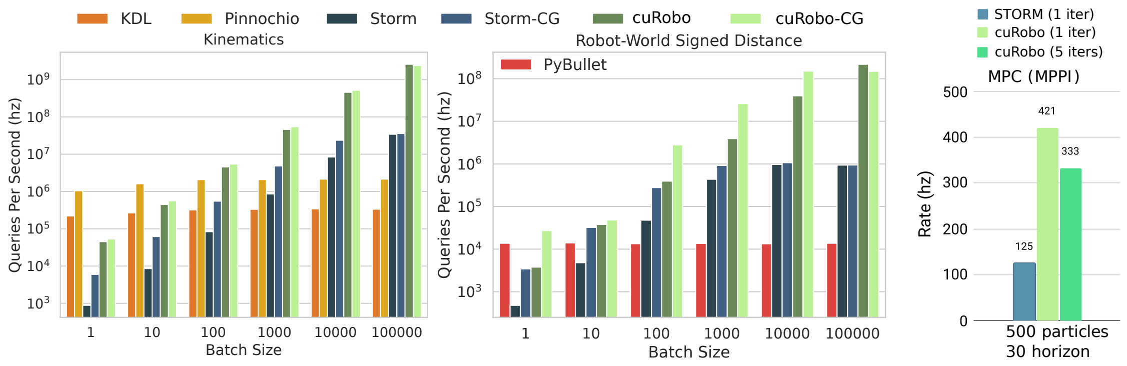


Figure 19: In the left plot, we see our forward kinematics match CPU methods at a batch size of 100 and becomes faster by upto 891x on 100k. In the middle plot, we see that our distance queries is upto 16,000x faster than *PyBullet* as the batch size grows to 100k. Note that the y-axis is in log scale plots. Leveraging our faster kinematics and distance queries, we can run MPPI at upto 421hz, 3.36× faster than STORM [59].

For signed distance queries, we compare with two prior methods – PyBullet which uses Bullet to compute the signed distance [24] and STORM. We are faster beginning at a batch size of 1 as our approach uses many parallel threads on the GPU for a single query.

6.2.6 Summary

We summarize the median compute time across the different modules in *cuRobo* in Figure 20. *cuRobo*'s implementation of kinematics and collision checking can compute within 1 nanosecond and 10 nanoseconds respectively when using a batch size of 100k. For general inverse kinematics and collision-free inverse kinematics, *cuRobo* can compute within 27 μseconds and 130 μseconds. This low computation time can accelerate existing robotics pipelines that use inverse kinematics such as reachability analysis [63] and placement planning [64]. Geometric planning has been used in verifying transition feasibility in hierarchical planning [65] and task and motion planning (TAMP) [66], where the quality of solutions is not critical and knowing if a path exists is sufficient. For these applications, leveraging *cuRobo*'s geometric planning can lead to a 101× speedup compared to using OMPL's RRTConnect algorithm. Our implementation of collision-free trajectory optimization takes 10ms, which could accelerate other robotics problems. The full motion generation pipeline already runs at 30ms on median on a modern PC. In addition, *cuRobo*'s motion generation scales well to a NVIDIA AGX Orin running at 60W, taking only 100ms enabling deployment of motion generation on edge devices. *cuRobo* obtains these low compute times while having most of it's stack

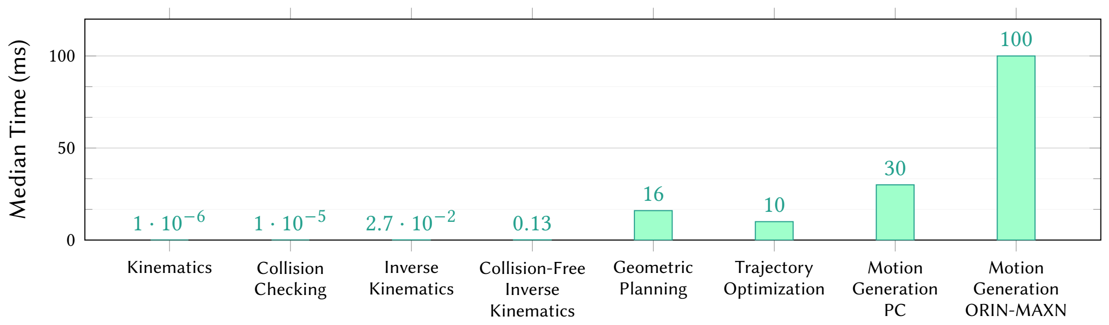


Figure 20: We plot the median compute time across the many modules we have developed in *cuRobo*. The compute time is within 100ms across all modules, with kinematics taking 1 nanosecond, and collision checking taking 10 nanoseconds when using a batch size of 100k. Inverse Kinematics takes 27 μseconds and collision-free inverse kinematics takes 130 μseconds with a batch size of 1000 on a NVIDIA RTX 4090. Trajectory optimization takes 10ms, followed by geometric planning taking 16ms, and motion generation taking 30ms across the benchmarking dataset.

in Python and with components implemented as separate modules. One could get even better compute times by fusing the modules at the CUDA kernel level, however that can make the library very rigid and inaccessible to robot practitioners.

6.3 Real Robot Tracking Performance

We study the tracking performance between the generated motions and executed motions by running *cuRobo* on two Universal Robots, an UR5e robot and an UR10 robot. We connect the robots to a NVIDIA Jetson ORIN AGX which is running a PREEMPT RT kernel and uses the ros driver from universal robots for communication. We run *cuRobo* on the same Jetson device and send the generated trajectories to Universal Robot's trajectory tracking controller. For both robots, we setup an obstacle and selected seven random poses scattered around the robot's workspace, which are shown in Figure 21. We then run motion generation to reach these seven poses in sequence five times, leading to a total of 35 reaching motion trials. The robots use an absolute magnetic encoder and an optical encoder together to measure the joint position. The accuracy of the magnetic encoder is +-0.0017 radians as obtained from [67]. We could not obtain the accuracy of the optical encoder and also the overall accuracy of the joint position measurement. We hence assume for all discussion below that the joint position is accurate up to 0.0017 radians.

We generate motions using *cuRobo* in two different modes, starting with *min-acc* where *cuRobo* performs trajectory optimization with acceleration minimization, without any jerk minimization, followed by *min-*

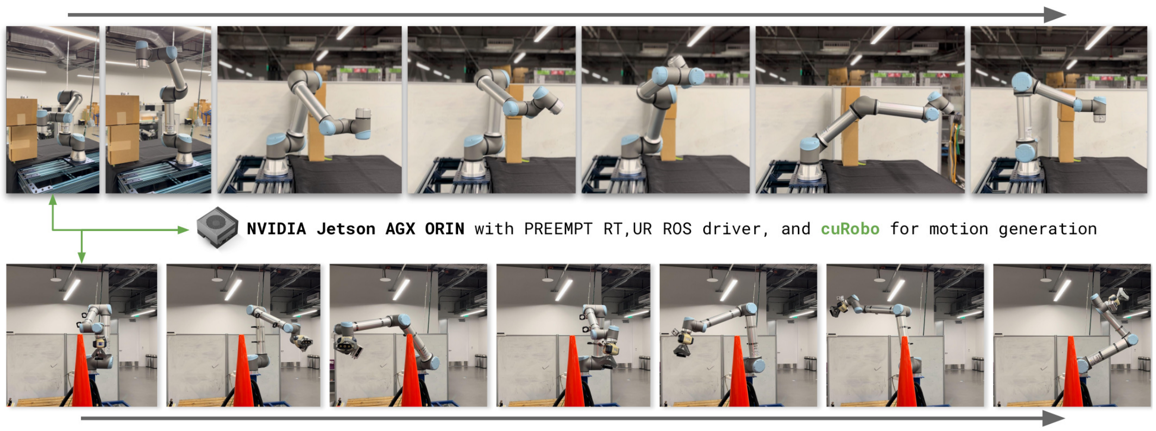

NVIDIA Jetson AGX ORIN with PREEMPT RT,UR ROS driver, and cuRobo for motion generation

Figure 21: We generate motions using *cuRobo* to reach the 7 poses shown here on the UR5e in the top and UR10 in the bottom. We use a NVIDIA Jetson AGX ORIN running at MAXN to generate motions using *cuRobo* and send the trajectories to the robot.

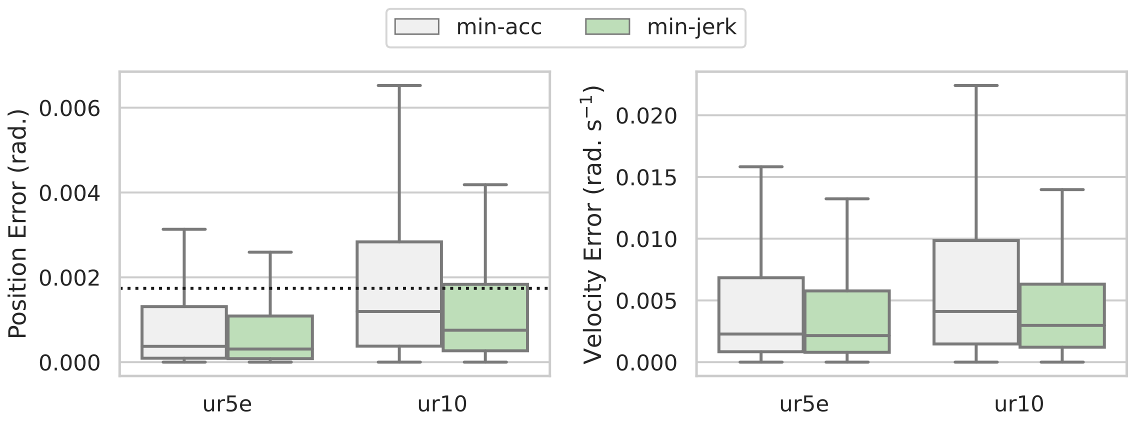


Figure 22: The tracking error in joint position space is plotted in the left and the error in velocity tracking is plotted on the right across the two real robots, UR5e and UR10. Here *min-acc* and *min-jerk* refers to *cuRobo*'s trajectory optimization with minimum jerk cost disabled and enabled respectively. The black dotted line in the left plot indicates the accuracy margin of the joint encoders on the robots.

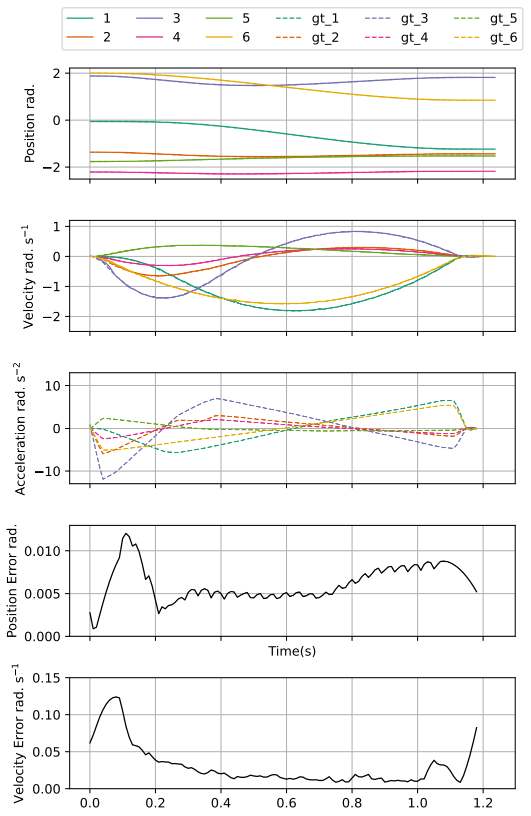
jerk where we minimize jerk along with acceleration. In both these modes, the trajectory is optimized over 32 timesteps, interpolated to a 0.01 second resolution, and sent to the robot. We found no difference between sending an interpolated trajectory (at a 0.01 resolution) and the optimized coarse trajectory (32 steps) to the UR10 and the UR5e. However, when executing trajectories with a low-level controller on many common robots, it might be necessary to interpolate the trajectory to a finer resolution before execution. We hence run all our experiments with interpolated trajectories.

We first measure the error in tracking the position and velocity across the trajectories. Deviations from the planned path can lead to critical failure as the robot could hit obstacles in the world. Poor velocity tracking can lead to the robot taking more motion time than planned, creating uncertainty in cycle time for tasks. As reported in our results in Figure 22, *min-jerk* has lower tracking errors in both position and velocity across both the robots. We observed a mean position error of 0.00117 radians and 0.00085 radians for *min-acc* and *min-jerk* on the UR5e robot, both within the 0.0017 radians accuracy margin of the joint encoders. On the UR10 robot, we found the mean position error to be 0.00250 radians and 0.00133 radians for *min-acc* and *min-jerk* respectively. We suspect the the larger position error on the UR10 to be because of the robot being physically larger, thereby requiring more dynamics compensation at high speeds compared to the UR5e. The position error for *min-acc* is also larger than the accuracy margin of the encoder.

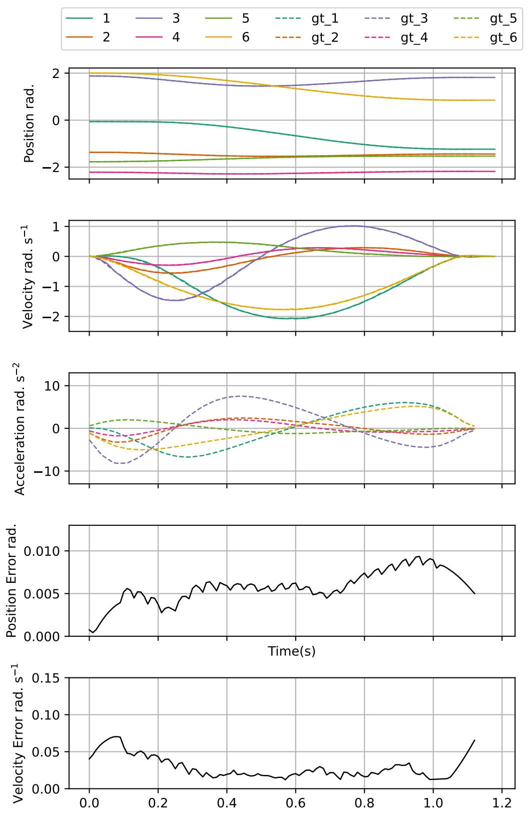
To closely examine the difference in position error between *min-acc* and *min-jerk* on the UR10, we plot one executed trajectory from the trials in Figure 23. We observe that the robot with *min-acc*, the robot has a large spike in position error at the start of the trajectory while in *min-jerk* there is no steep increase in error at the start. We suspect this spike in *min-acc* at the start to be because of the robot not being able to instantly accelerate to the maximum acceleration limit. With *min-jerk*, we gradually increase the acceleration, thereby minimizing tracking error due to delay in robot's acceleration. While one could feed a feed forward torque to help the robot accelerate more quickly, sending torque commands is not possible in many industrial robots including the UR10 and UR5e.

Our motion times for *min-jerk* were [0.98, 1.57, 2.45] seconds and [0.87, 1.34, 2.16] seconds on the UR5e and UR10 respectively, where the numbers map to mean, 75th percentile and 98th percentile. Our motion times for *min-acc* were [1.02, 1.59, 2.83] and [0.92, 1.52, 2.49] on the UR5e and UR10 respectively.

As part of our real robot experiments, we also timed the whole pipeline, starting with when *cuRobo* gets a planning query and completes computing a plan, followed by when the robot starts moving, and finally when the robot completes trajectory execution. We found *cuRobo* to complete planning within 100ms on average for both UR5e and UR10 robots. We observed on average a delay of 58ms and 68ms between when a trajectory is sent to the UR ROS driver and when the robot starts moving on the UR5e and UR10 respectively. We plot the time it takes overall reach a target pose and the split between planning, delay, and execution in Figure 24. We see that *cuRobo* takes [6.02%, 6.62%, 9.12%] and [6.67%, 7.93%, 9.13%] of the time in the full pipeline on the UR5e and UR10 respectively in *min-jerk* mode. The delay accounts for 5% and 6% of the time on 98th percentile on UR5e and UR10 respectively. The robot is in motion [90%, 92%, 94%] and [89%, 90%,



(a) Minimum Acceleration



(b)Minimum Jerk

Figure 23: The planned trajectory and the followed path is shown for the UR10 robot in position space and velocity space in the top two grid locations. We plot the planned trajectory's joint acceleration in the middle plot. We plot the sum of the absolute error in tracking position and velocity across all joints in the bottom two grid locations. We see the effect of large jerk at near the start time-steps of the velocity error and position error in (a) while (b) has a more flat error.

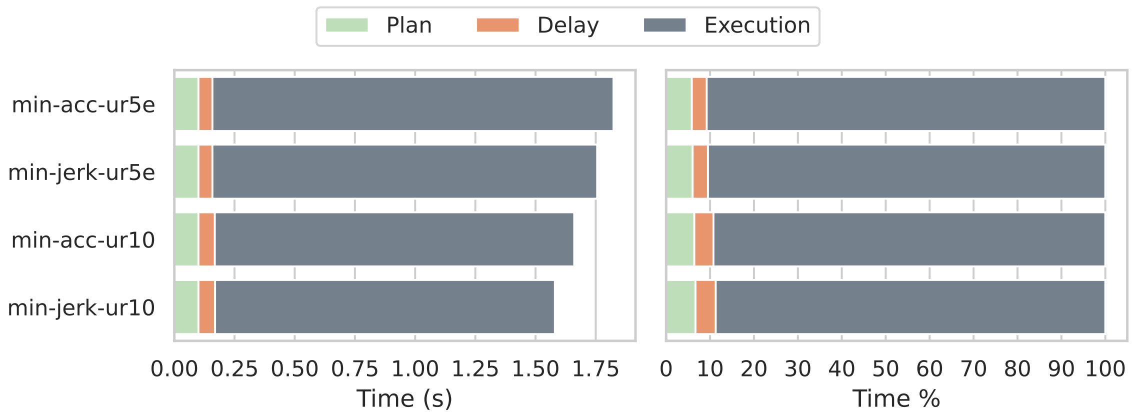


Figure 24: The time taken to plan and execute a trajectory across the two robots is plotted on the left. On the right, we plot the time split between planning (*Plan*), delay between sending a trajectory and the robot starting to move (*Delay*), followed by the time the robot is in motion (*Execution*).

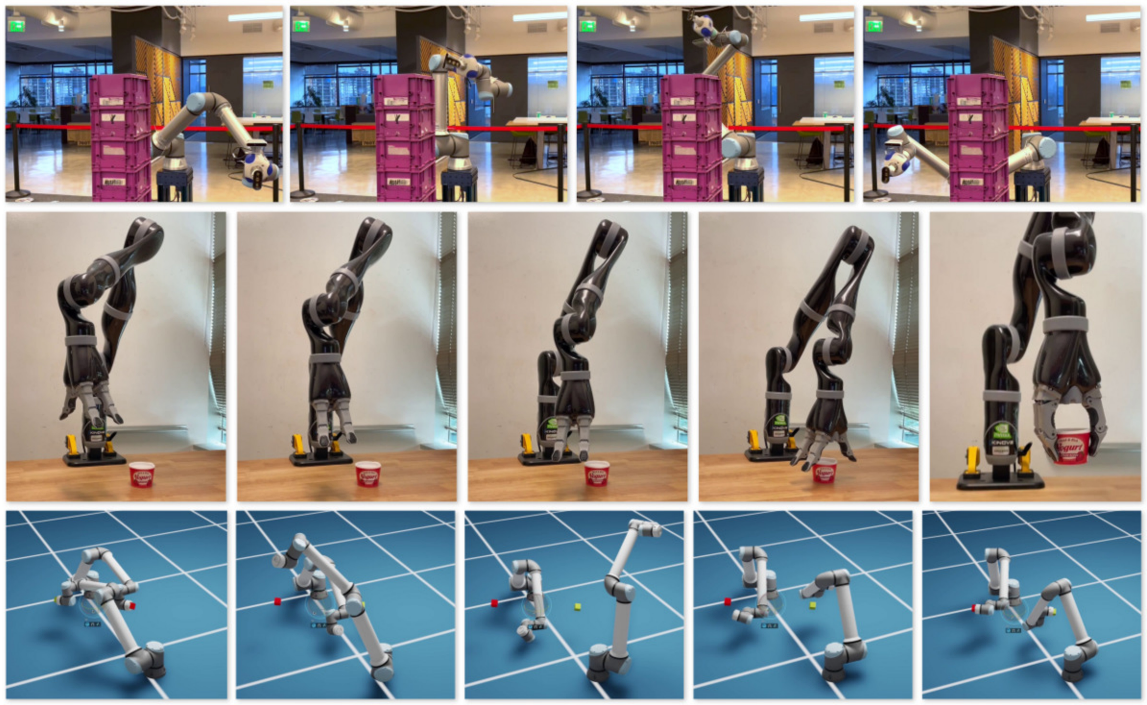


Figure 25: In the top row, we show the UR10e robot avoiding an obstacle by using cuRobo for motion generation in combination with an ESDF map that was built with *nvblox*. The second row shows a Kinova Jaco robot grasping an object by using *cuRobo* to generate motions for moving to the grasp pose and lifting the object. The bottom row shows coordinated motion generation with a dual UR10e robot setup in NVIDIA Isaac Sim. The two UR10e robots start at a collision-free configuration and move around each other to reach their respective target poses given by red and yellow colored cubes.

92%] on UR5e and UR10 respectively. A common technique to reduce planning overhead in cycle time is to plan for the next sequence of targets while the robot is executing it's current trajectory. This has been leveraged with existing planners as they can take significantly longer planning times, in the range of 2.5 seconds. However, this can prevent the robot from reacting to any world or task changes between motions. With *cuRobo* we can plan the next motion after executing the current trajectory as we only take 6% of the cycle time on average. This also simplifies the robot programming pipeline, as computational tasks can be executed in serial.

6.4 Deployment on different Robot Platforms

We deployed *cuRobo* on few different robot platforms as shown in Figure 25, with no changes to parameters in trajectory optimization. We created the robot spheres for these robots along with a collision-free rest configuration. We then called the inverse kinematics, geometric planning, and trajectory optimization methods.

First, we deployed *cuRobo* on a UR10e with nvblox [36, 39, 68] to perform collision checking between the world and the robot as shown in Figure 25. We use nvblox to generate a euclidean signed distance field (ESDF) map of the world, by scanning with a realsense D-415 camera attached to the end-effector. We then generate motions for the robot to go around obstacles. Next, we deployed on a Kinova Jaco arm as shown in Figure 25, where we implemented a PD controller in the velocity space to command the generated trajectory. We also tested coordinated motion generation for a dual arm UR10e robot setup in NVIDIA Isaac Sim and preliminary results are promising as *cuRobo* finds collision-free paths to move both arms to their targets as shown in Figure 25.

6.5 Summary

We first compared the quality of motions generated from *cuRobo* in Section 6.1 and showed that *cuRobo* generates better solutions than existing techniques. We then showed in Section 6.2 that *cuRobo* generates these high quality solutions in a fraction of the time taken by existing methods across different computing platforms including a 15W NVIDIA Jetson device. We then compared the compute time across sub-components, inverse kinematics, geometric planning, and trajectory optimization and showed double digit speedups compared to existing implementations. We validated our motion generation approach on two robots in Section 6.3, a UR5e and UR10 robot, both tracking the minimum jerk high-speed trajectories from *cuRobo* with position errors below the accuracy margin of the joint encoders. We also showed our approach working on different robot platforms in Section 6.4.

7 Component Analysis

We study the effect of different components in *cuRobo*'s trajectory optimization, starting with collision cost formulation, followed by the effect of number of parallel seeds in trajectory optimization, and then the effect of different parameters in our numerical optimization solvers.

7.1 Collision Cost Formulation

We first analyze the impact of different collision cost formulations on success of the optimization problem in Figure 26. We ran our motion generation pipeline without geometric planning and 500 IK seeds. We ran experiments without particle-based optimization, with particle-based optimization, and with 1 and many seeds. From the results of this experiment, we make the following observations:

- Increasing activation distance from 0cm to 2.5cm improves success rate by 27%, having the largest impact in success rate.
- Using continuous collision checking (swept) improves success rate further by 4% when compared to only using an activation distance of 2.5cm.
- Speed metric improves success rate further by 2% across the dataset.

We found activation distance is critical in improving success rate as we perform trajectory optimization at a coarse scale of a few timesteps(32-50) and then interpolate the trajectory to a fixed dt of 0.025 to validate success. After this interpolation, a collision-free trajectory can move into regions of collision and lead to failure. In addition, having an activation distance adds smoothness to the cost term, making it easier for an optimization solver to minimize collisions. Our continuous collision checker checks collisions between timesteps by linearly interpolating in the task space, approximating linear interpolation in the joint space. This improves the success rate by 7% with tuned TO seeds and zero activation distance. The improvement diminishes with activation distance where it only improves by 3%.

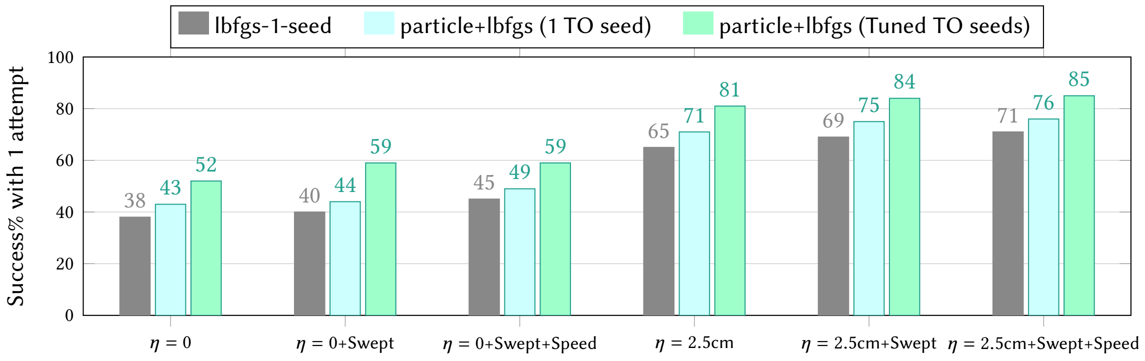


Figure 26: We compare the effect of different metrics in computing world collision cost. We start with a traditional formulation of collision cost where we only add a cost when robot is in collision with the world $\eta = 0$, followed by addition of our continuous collision checking implementation *Swept*, and then the speed metric introduced in [7]. We then add an activation distance of 2.5 cm to these metrics.

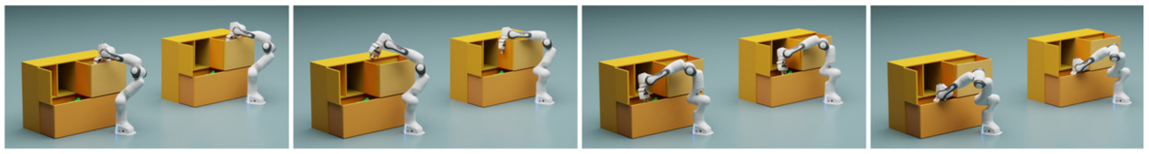


Figure 27: Effect of Speed Metric during Trajectory Optimization is visualized in the dresser environment. The left robot in each frame uses the speed metric during optimization, enabling the solver to move out of collision (in the middle image). The right robot in each frame does not use the speed metric and as a result speeds through the high cost collision region.

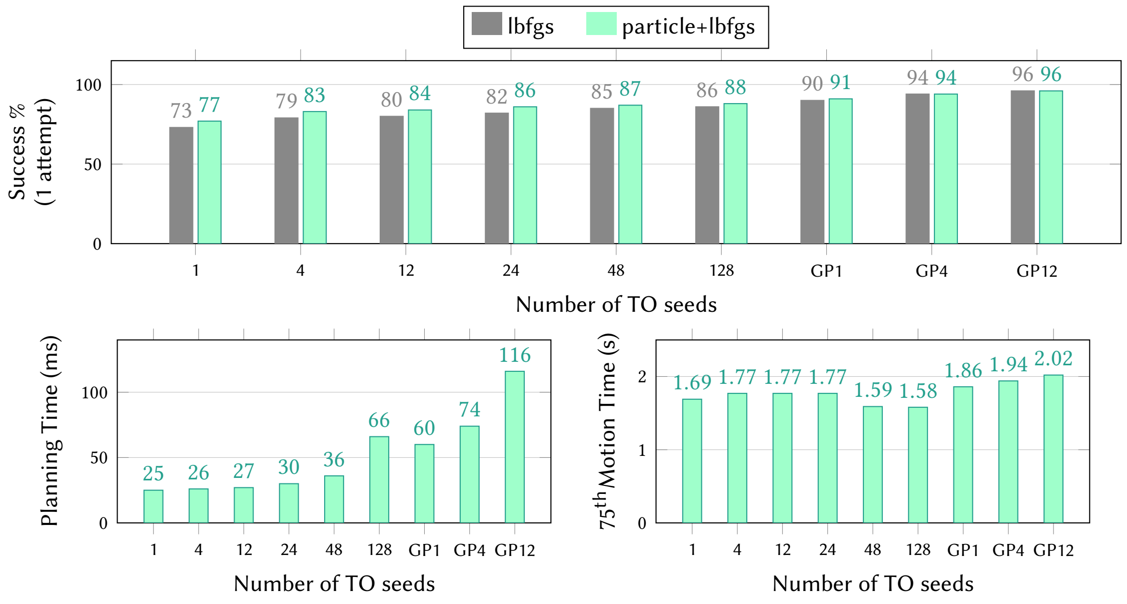


Figure 28: The improvement in success rate across the 2600 problems in our dataset as we increase the number of trajectory optimization seeds is shown in the top plot. We also see that the use of geometric planning to seed trajectory optimization (GP1, GP4, GP12) increases the success rate to 96% with an increase in planning time. We plot the 75th percentile *Motion Time* on the bottom right plot across the number of seeds.

The effect of speed metric is observable at an activation distance of 0.0 cm on trajectory optimization with 1 seed, where it provides a 5% improvement over LBFGS and also over particle+LBFGS. This effect diminishes when we use many parallel seeds as a collision-free path is obtained through other seeds. We visualize the effect of speed metric in Figure 27, where the use of speed metric enables the optimization solver to find a path that is collision-free while without the metric, the solver ends up speeding through the high cost region.

From the results in Figure 26, we see that a naive implementation of collision cost with a gradient based optimizer and 1 seed achieves a success rate of 38%. Adding an activation distance, as introduced by [7] would bring this to 65%, followed by addition of a continuous collision checking would bring this to 69%. Addition of the speed metric from [7] would increase this further to 71%. Adding a particle-based optimizer to initialize gradient-based solver would increase the success to 76%. Finally, running trajectory optimization across many seeds would increase the success rate to 85%.

Overall, our application of existing techniques from [7] in combination with our continuous collision checking module and many parallel seeds for trajectory optimization improves the success rate from 38% to 85%, enabling *cuRobo* to solve 85% of the motion generation problems within 1 attempt.

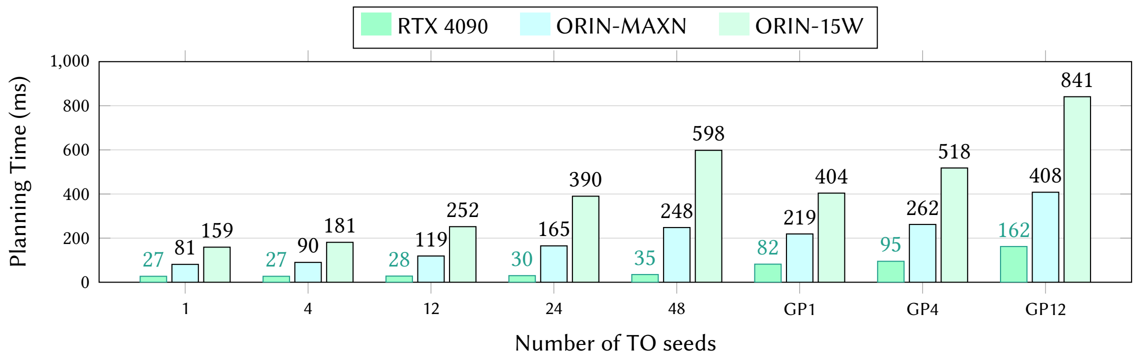


Figure 29: The slowdown in planning time as we increase number of seeds used in trajectory optimization is visualized across the 100 problems on *cage-panda* environment. The slowdown is more significant on the NVIDIA Jetson Orin as it has lower number of SMs compared to a RTX 4090.

7.2 Effect of Parallel Seeds

In an ideal setting, we would want to run as many parallel seeds of optimization as possible and pick the best solution. However, as we reduce the compute available for motion generation, the number of parallel seeds used can have a significant impact on compute time. We study the interaction between number of seeds and compute time we ran motion generation on the 100 problems from the *cage-panda* environment with varying number of trajectory optimization seeds across the three compute platforms. We used 500 IK optimization seeds and also run particle based optimization to initialize L-BFGS in these experiments. We observed that on a RTX 4090, the compute time only changes by 8ms from using 4 seeds to 48 seeds while it changes by 158ms and 417ms on the ORIN at MAXN and 15W respectively. We plot these results in Figure 29.

Next, we study the impact of trajectory optimization seeds on success and motion time in Figure 28. We observed that increasing the number of parallel seeds for trajectory optimization increases the success rate and also decreases the motion time starting at 48 seeds. Initializing trajectory optimization with collision-free paths from our geometric planner also increases the success rate. However, using geometric planner to initialize trajectory optimization doubles the planning time on most problems. In addition, we found that the motion time was higher when trajectory optimization is initialized with a geometric planner. Based on these results, we use 12 trajectory optimization seeds for most environments in our evaluation dataset and increase this up to 28 for harder environments. We discuss the parameters used in Appendix B.3.

7.3 Line Search

The role of line search in gradient-based numerical optimization solvers is to find the best magnitude to scale the step direction. A common strategy to find the best magnitude is by backtracking search, where we start with a magnitude of 1 and reduce this value until some conditions are met. Two common conditions used in many modern numerical solver libraries are *weak-wolfe* and *strong-wolfe* which we term *wolfe* and *st-wolfe* in our comparisons. We also compare against no scaling of the step direction which we term *no-ls*. We compare these options with our noisy line search technique introduced in Section 4. We use [0.01,0.3,0.7,1.0] as the values for the noisy line search which we term *noisy-ls*. We also compare with only 1 value for noisy line search [0.01,1.0] which we term *noisy-ls-1*. For wolfe and strong wolfe, we use the values [0.0001,0.001,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0] in the line search. We evaluate these line search techniques across the full dataset with 1 trajectory optimization seed initialized by 2 iterations of particle-based optimization.

From Figure 30, we see that *no-ls*, *noisy-ls-1*, and *noisy-ls* all have a success rate higher than 70%, with *noisy-ls* having the highest success rate of 75.69%. While most of these techniques lead to good success rate, the impact of line search is more observable in the position error at the final timestep of the trajectory across the dataset as plotted in Figure 30-(b). We found noisy line search to have the lowest error of 2.86 mm while no line search ends up with an error of 4 mm. We observed wolfe and strong wolfe perform worse in both success and position error across the dataset as for these line search techniques, we use a magnitude scale of 0.0 if none of the chosen line search values satisfy the conditions.

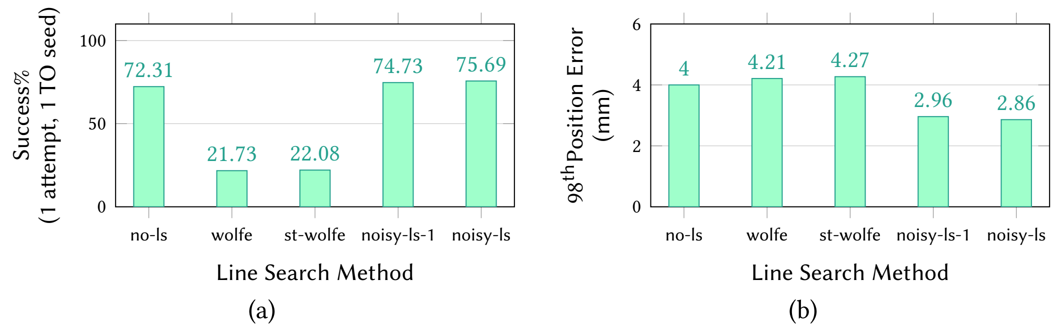


Figure 30: We compare the different line search techniques and their impact on trajectory optimization, starting with no line search *no-ls*, followed by common line search methods weak wolfe *wolfe* and strong wolfe *st-wolfe*. We compare these methods to our proposed noisy line search scheme *noisy-ls*. We also compare to noisy line search with only two values [0.01,1] which we term *noisy-ls-1*.

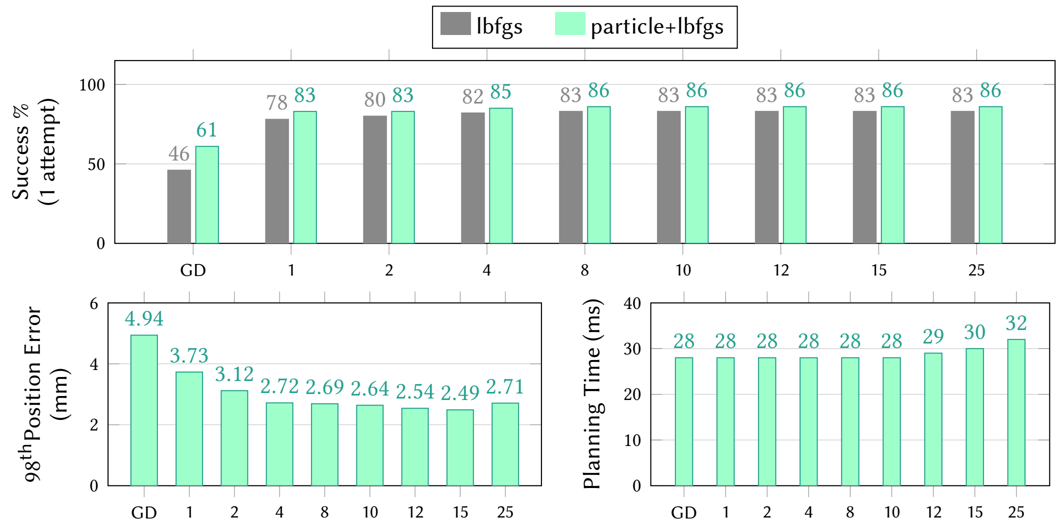
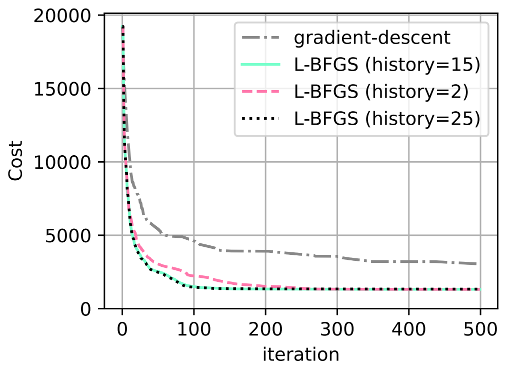


Figure 31: We compare Gradient Descent (GD) to L-BFGS with different history lengths, starting with the success rate in the top plot. We also plot the 98th percentile position error in bottom left and the average planning time on the bottom right.

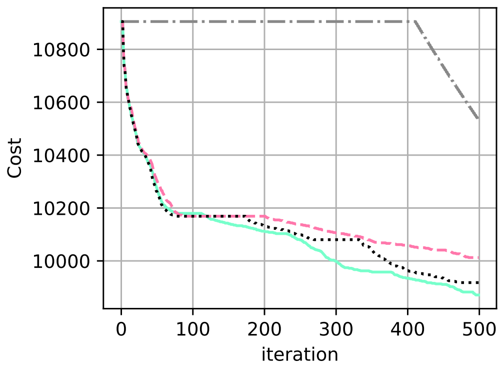
7.4 Gradient Descent and Effect of history length in L-BFGS

We ran trajectory optimization with gradient descent as the optimizer, swapping out L-BFGS, but keeping our noisy line search technique and found that it succeeded in 46% of the problems. Using particle-based optimization to initialize gradient descent increased the success to 61%. However, the 98th percentile position error was 4.94cm compared to L-BFGS's 2.72cm. In addition, our efficient implementation of L-BFGS enables us to compute step direction using L-BFGS within similar compute times to using Gradient Descent upto a history length of 12. The gap in performance between gradient descent and L-BFGS is also visualized on one trajectory optimization problem in Figure 32 where we see that L-BFGS with any history length converges to the minimum within 100-200 iterations while gradient descent has not converged even after 500 iterations.

We also ran experiments to study the impact of history length in L-BFGS. L-BFGS approximates the hessian by using the recent gradients and costs, which are stored in a history buffer. The more history L-BFGS uses to approximate the hessian, the closer the hessian gets to the true hessian. However, with increasing history, the compute required to compute the step direction also increases as shown in Figure 31-(b). To allow for the best chance between methods, we run our method with tuned number of parallel seeds across the dataset. We also compare the improvement we get with particle-based solver for initialization. From Figure 31, we found that increasing history indeed improves convergence as seen by the improvement in



(a) Trajectory Optimization



(b) Timestep Optimization

Figure 32: The cost at every iteration on one trajectory optimization problem from our evaluation set is plotted in (a) and the optimization after finding the optimal timestep is plotted in (b). We see that L-BFGS converges within 100 iterations at a history of 15 and 25, while a history of 2 leads to requiring 200 iterations to converge in (a). Gradient descent does not converge to a similar cost as L-BFGS even after 500 iterations. We also show the reduction in cost during timestep optimization in (b), where gradient descent doesn't start improving until 400 iterations.

success rate and reduction in position error. And increasing history starts having an impact on planning time at a value of 12. We chose a history of 4 for all our evaluations as we observed that the improvement in success rate and position error was not significant.

7.5 Effect of Particle Optimization

Across analysis of many of the components, we compare the improvement we obtain with particle optimization. We observed that since we only run 2 iterations of particle-based optimization, it only adds 2 ms to the planning time on average while improving success. We observed that using two iterations of particle-based optimization improved the success by 5% from 71% to 76% when running trajectory optimization with a single seed. With multiple seeds, the success improved by 3%, from 82% to 85%.

8 Concluding Remarks

We formulated the global motion generation problem as an optimization problem and introduced techniques from global optimization literature to efficiently solve them with accelerated parallel compute. We empirically validated our approach on a difficult set of motion generation problems for manipulators [52, 53] and showed a 60× speedup over SOTA motion generation methods on a modern PC and a speedup of 28× on a NVIDIA Jetson AGX ORIN at 60W. We release our implementations as a high performance CUDA accelerated library *cuRobo* with pytorch and python wrappers for easy integration in robotic pipelines. We will discuss some limitations of our work and potential extensions in the next section.

8.1 Limitations & Open Research Problems

There are several open research problems in motion generation that our approach does not solve in it's current form. We hope that our results and framework can be leveraged to solve these problems. We list some key problems below,
Global Reactive Motion Generation Our approach is currently limited to planning full motions, where the robot starts from a static state. While we can extend our trajectory optimization to start from a non-static state, reactive motion generation requires motions at a fixed solve rate and our approach can take longer time on hard problems as shown by our longer 98th percentile time.
Constrained Motion Generation Our preliminary evaluation on pose constraints [69] during motion such as maintaining orientation or height by adding a large weight to a running cost showed promise. However, we did not evaluate rigorously and also did not implement constrained graph planning which could be important for getting global guarantees.

Task Sequencing and Route Planning Finding the optimal sequence of poses to reach given an unordered list is an important problem in the industry, often called Task Sequencing [70] and route planning [71]. We do not explore this task in this work but do see point to point motion generation to be a critical component of finding the optimal sequence. Task and motion planning [66, 72, 73] could also leverage our point to point motion generation for faster feasibility checks and finding more optimal sequence of motions.
Contact-Rich and Full-Dynamics Trajectory Optimization Our current cost terms are focused on kinematic trajectory optimization with limits on position derivatives. Interacting with objects in the world would require optimizing over contacts, e.g. with contact implicit formulations [74] of trajectory optimization. Trajectories that minimize torques or other parts of dynamics would require optimizing over the robot's dynamics. Integrating existing GPU accelerated dynamics implementations [75–77] into *cuRobo* could potentially solve for trajectories over dynamics.
Robotics Solvers We implemented MPPI, gradient descent and L-BFGS in our framework to solve motion generation. Robotics focused numerical solvers are showing promise in quicker and better convergence compared to standard numerical solvers [78, 79] on robotics tasks. Formulating parallel compute friendly versions [80, 81] of these solvers could reduce the compute time even further and also enable new features in *cuRobo* such as solving with hard constraints.
Collision Avoidance from partial world sensing Our approach does not tackle partial perception and instead assumes that we can obtain a complete representation of the world. Incorporating learned representations of the world [34, 64, 82] into our collision cost term could extend our motion generation method to work in unknown environments.
Accelerating Computational Blocks Our approach encapsulates the key components in motion generation into modular components and enables researchers to develop improved algorithms for these components without requiring full expertise on the stack. We hope that this broadens our library's audience to experts in non-robotics fields. We think that this is important as researchers in computer architecture are starting to accelerate manipulator algorithms [75, 83] and providing a reference SOTA implementation along with benchmarks can greatly reduce the entry barrier for researchers to accelerate robotics. As a step in this direction, *cuRobo* has been used by computer architecture researchers to reduce memory bottlenecks in motion generation with reduced precision techniques [84].

8.2 Acknowledgements

We acknowledge several people who have helped in this work. Adithyavairavan Murali for helpful discussions on motion planning for grasping objects. Julen Urain for feedback on code design and early testing of collision checking functions. Rowland O'Flaherty for building the python template, CI, UR jetson docker, and parts of the USD API. Sandeep Desai for setting up the UR robots. Jonathan Tremblay for rendering nvblox meshes and testing cuRobo on a kinova Jaco. Buck Babich, Bryan Peele, and Avi Rudich for building the robot sphere generation utility in NVIDIA Isaac Sim and helpful discussions on minimum jerk trajectories. Miles Macklin and Lukasz Wawrzyniak for adding pytorch CUDA Graph integration into warp and also for helping build the mesh signed distance kernels. Alexander Zook and Hammad Mazhar for debugging CUDA contexts between pyTorch and NVIDIA Omniverse. Felix Wang, Claudia D'Arpino, and Xuning Yang for creating cuRobo's Isaac sim examples. YuShun Hsiao for testing an early prototype of the library and providing feedback on CUDA kernels.

A Trajectory Optimization: Cost Terms & Solvers

As mentioned in Section 2, we setup our trajectory optimization problem to optimize over the joint configurations $\theta_{t\in[1,T]}$ across t timesteps. As our implementation of L-BFGS can only handle box constraints on optimization variables, we write all our constraints as cost terms with large penalties. We discuss in detail the cost terms in our trajectory optimization, followed by our technique to optimize for the resolution of time discretization. We then discuss different trajectory profiles that can be obtained using cuRobo in Section A.6 and details on our solvers in Section A.7.

A.1 Pose Reaching Cost

The main goal of our trajectory optimization problem is to reach a target pose X_g for the robot's end-effector at the final timestep T. We implement this cost leveraging our differentiable kinematics function $K_e(\theta_T)$ which gives us the end-effector's position $p_T \in \mathbb{R}^3$ and the unit quaternion $\hat{q}_T$. We use a L-2 norm for computing the linear distance (i.e.,$||p_g - p_T||$) and use the quaternion distance for orientation $q_g^\top q_T$. We scale these distances d by $\alpha_0 \log\cosh(\alpha_2 d)$ to compute the cost. This scaling enables our cost to have both high accuracy near the goal and stable gradients when far away from the goal as shown in Figure 33-(a). Our pose reaching cost term is written as,

$$C_{\text{goal}}(X_g, \theta_T) = \alpha_0 \log\cosh(\alpha_2 ||p_g - p_T||_2) + \alpha_1 \log\cosh(\alpha_3 (q_g^\top q_T)) \tag{14}$$

While we can also add this cost with a smaller weight across timesteps to encourage the robot to reach the target pose as quickly as possible (i.e. a running cost), we found this running weight to be sensitive across environments and instead add a weighted joint space path length minimization cost which is discussed next. We use $\alpha_0 = 2000, \alpha_1 = 350, \alpha_2 = 100$, and $\alpha_3 = 100$ in our experiments.

For tasks that require reaching a joint configuration, we use a similar cost term in the joint position space,

$$C_{\text{cspace}}(\theta_g, \theta_T) = \alpha_4 \log\cosh(\alpha_5 ||\theta_g - \theta_T||_2^2) \tag{15}$$

where $\alpha_4 = 5000, \alpha_5 = 50$.

A.2 Path-Length Minimization & Smoothness Costs

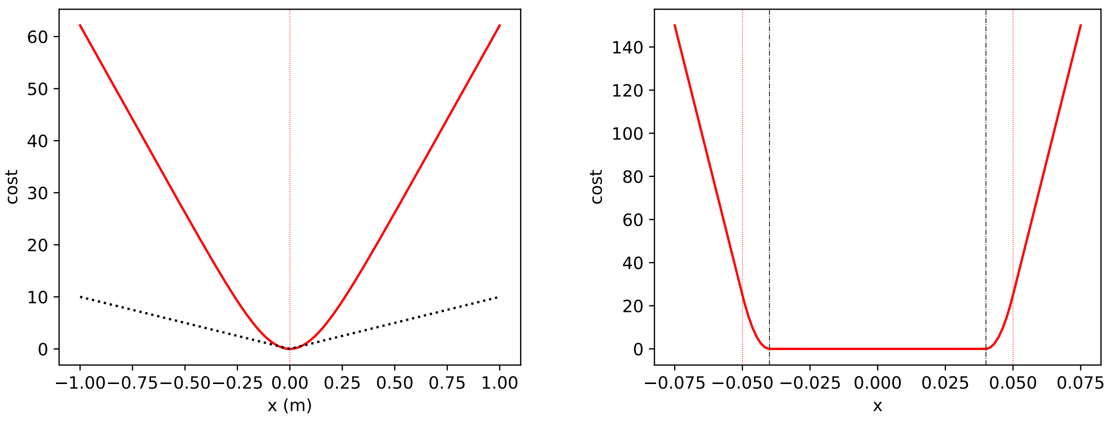
Our path length minimization cost term is a squared L-2 norm on the joint acceleration across timesteps $\sum_{t\in[0,T]} ||\ddot{q}_t||_2^2$. This term encourages the robot to find a trajectory that has the least acceleration, thereby resulting in ramp up of velocity in one direction for the joints when feasible to reach the goal as that velocity profile will result in the least acceleration. We also have a term that minimizes jerk $\sum_{t\in[0,T]} ||\dddot{q}_t||_2^2$, penalizing large instantaneous changes in acceleration which can cause poor trajectory tracking as discussed in Section 6.3.

We also want the robot to stop at the final timestep T which can be encouraged by adding a zero velocity cost term for the final timestep T. However, if we only penalize velocity to be zero at the final timestep, the trajectory obtained could have very large acceleration and jerk. We hence also want to have zero acceleration and zero jerk at the final timestep T. We can achieve this by penalizing velocity at the last three timesteps $t \in [T-3, T]$. Similarly, to encourage smooth acceleration at the start of the trajectory, we can add velocity constraints for the first three timesteps $t \in [0, 2]$. Empirically, we found adding these velocity constraints made our optimization become sensitive to the weights we used for these constraints between optimization problems. An alternative way to formulate this constant state criteria is to implicitly make the first and last three states be the same, similar to [10]. We hence use this implicit formulation to ensure smooth acceleration throughout the trajectory. The cost terms for smoothness are,

$$C_{\text{smooth}} = \sum_{t\in Q} \alpha_6 \log\cosh(\alpha_7 \dot{q}_t) + \sum_{t\in[0,T]} (\alpha_8 ||\ddot{q}_t||_2^2 + \alpha_9 ||\dddot{q}_t||_2^2) \tag{16}$$

where we use $\alpha_6 = 5000, \alpha_7 = 50, \alpha_8 = 5000$ and $\alpha_9 = 1.0$.

In addition to the above cost terms, we have three constraints for trajectory optimization: (1) enforcing joint limits, (2) avoiding self collisions between links of the robot, and (3) avoiding collisions between the robot and the world. We discussed the collision terms in Section 3. We will discuss enforcing joint limits next in Section A.3.



(a) Pose/CSpace Attraction cost (b) Bound cost

Figure 33: Our attraction cost profile has a tighter curve shown by the red line, compared to a standard l-2 norm which shown by dotted black line in (a). Our joint limit cost profile, shown in (b) transitions smoothly from a quadratic increase to a linear increase avoiding sudden linear jump that can happen right at a limit where the limit is shown by the red vertical dotted line.

A.3 Enforcing Joint Limits

To minimize the optimization from hitting joint position, velocity, acceleration, and jerk limits we add a smooth cost motivated by the potential function introduced by Ratliff *et al.* [7]. For a joint value q_t, we define the cost term as

$$c_p = \begin{cases} q_l - q_t + 0.5\eta_2 & \text{if } q_t < q_l \\ \frac{0.5}{\eta_2}(q_l - q_t + \eta_2)^2 & \text{if } q_l + \eta_2 > q_t \geq q_l \\ q_t - q_u + 0.5\eta_2 & \text{if } q_t > q_u \\ \frac{0.5}{\eta_2}(q_t - q_u + \eta_2)^2 & \text{if } q_u - \eta_2 < q_t \leq q_u \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

where q_u, q_l are the upper and lower joint limits respectively. The parameter η_2 defines the threshold before a joint limit from where we start penalizing the joint value. The profile of this cost is shown in Figure 33-(b). We use a similar cost term for velocity, acceleration, and jerk limits, all with an activation distance of $\eta_2 = 0.1$.

A.4 Time-step Descretization and Temporal Weight Scaling

To perform trajectory optimization across the full workspace of the robot, we need to be able to solve for both very short and very large motions. One way to do this is to attempt to solve the trajectory optimization problem, first with a very large number of timesteps t, and then sequentially reduce the timesteps until the optimization fails to converge as done by Ichnowski *et al.* [85]. However, this requires us to run sequential calls to trajectory optimization which can be slow on the GPU, especially when we have to reinitialize all buffers that are affected by the number of timesteps t. Another option is to treat dt also as part of the optimization variables, which we do not explore in this work.

In our current implementation, we fix the number of timesteps t and instead change the time discretization dt between timesteps to be able to solve motions of all ranges as shown in Algorithm 4. To run optimization with different dt, we scale all our cost terms that relate to velocity, acceleration, and jerk to account for the change in magnitude of dt. We first run trajectory optimization with a very large dt_i of 0.25 seconds and then scale the output trajectory to find a dt_o that pushes the velocity, acceleration or jerk to the robot's limits. We then scale our weights by dt_o and rerun trajectory optimization with dt_o to get the final trajectory. We then re-time this final trajectory to find the best dt_f. We empirically found that not penalizing jerk during the first trajectory optimization led to better convergence.

Algorithm 4: Time Discretization

```
Init   : $\Theta_i$, $dt_i$
Output: $\Theta_f$, $dt_f$
1 $\Theta_o \leftarrow$ traj_opt($\Theta_{init}$, $dt_i$) ▷ run trajectory optimization with initial dt
2 $dt_{opt} \leftarrow$ retime($\Theta_o$) ▷ find dt that pushes trajectory to robot limits (velocity, acceleration or jerk)
3 scale_traj_opt($dt_{opt}$) ▷ Scale weights by new dt
4 enable_jerk_cost() ▷ enable jerk minimization cost term
5 $\Theta_f \leftarrow$ traj_opt($\Theta_o$, $dt_{opt}$) ▷ run trajectory optimization with new dt
6 $dt_f \leftarrow$retime($\Theta_o$)▷ get final dt by retiming the final trajectory
```

A.5 Finite Difference for State Derivatives

Our optimization variables are in the joint position space $q_{t\in[0,T]}$ while we have cost terms in the velocity, acceleration, and jerk spaces. A standard technique to compute time derivatives is by backward finite difference, where we finite difference position to get velocity, then finite difference velocity to get acceleration, and finally finite difference acceleration to get jerk. However, we observed oscillations on the real robot when executing high-speed trajectories due to the low accuracy of backward finite difference in computing velocities. We then tried central difference and that got rid of the oscillations on real robot execution. The optimization also required more iterations to converge compared to backward difference.

We investigated finite difference approaches from Pose graph optimization literature and found the five point stencil method to be a good fit for computing time derivatives in our trajectory optimization problem. With a five point stencil method, we read five adjacent joint positions to compute velocity, acceleration, and jerk values with much higher precision than central or backward difference. An added benefit of five point stencil method for computing time derivatives is that any gradients from time derivatives have a blending effect on the position space [86]. The improved accuracy and the blending effect reduced the number of iterations required to converge by 50%. The trajectories obtained with the stencil approach also ran successfully on the real robot as shown by our tracking results in Section 6.3. Across all these methods for computing derivatives, we observed aliasing artifacts in the acceleration space when our jerk minimization cost was enabled. The aliasing was larger in backward difference, reduced in the central difference approach and even smaller in the five point stencil difference. This artifact could be reduced further by using even larger number of points in the stencil. However, we did not observe any issues on the real robot when executing the five point stencil trajectories. In our current implementation, we smooth out the acceleration artifacts with an average sliding window filter after convergence.

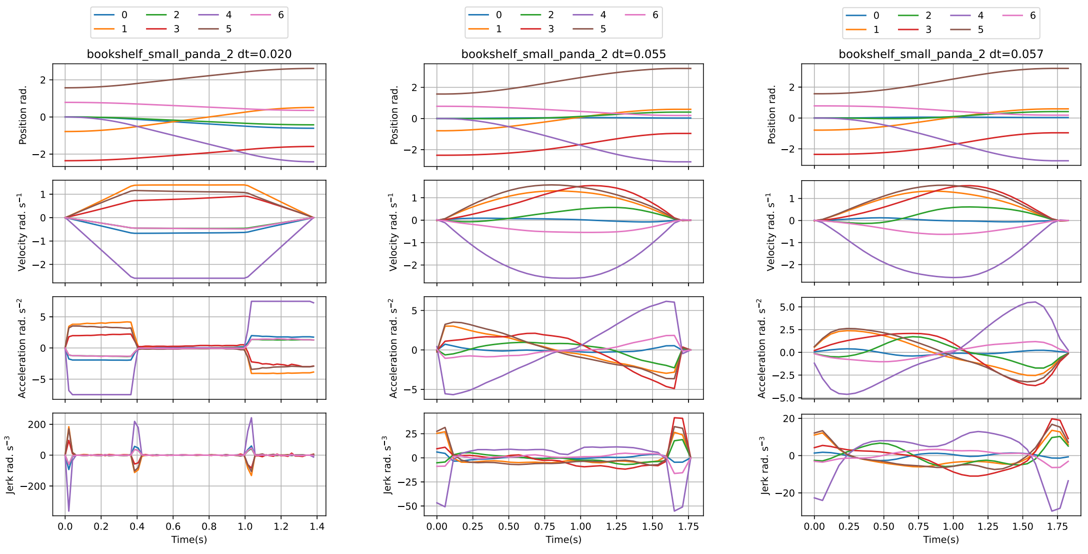
A.6 Trajectory Profiles

Trajectory profiles have been studied [4, 87] as a post processing step to geometric path planning, where methods try to obtain a time-optimal solution with bounded limits on velocity, acceleration, and possibly jerk. Kunz and Stilman [4] explored a technique to traverse waypoints without stopping at them by relaxing the accuracy in reaching the waypoints. Their approach was able to obtain trajectories with bounded velocity and acceleration. However, their trajectories have very large jerks as shown in Figure 34-(a). Ruckig [87] explored introducing jerk constraints, which constrained the solution to have zero velocity, acceleration, and jerk at the starting timestep and the final timestep. However, their solution was only optimal when the path had no waypoints between the start and goal configuration. They provide a solution to also solve for intermediate waypoints but do mention that it's not the optimal solution (ruckig documentation). In addition, we were unable to evaluate their waypoint solver as it is not accessible without a license.

Given that there is no technique that can output bounded jerk, acceleration, and velocity trajectories with zero velocity, acceleration, and jerk at start and final timesteps, we use our cost terms from Sec A.2 to encourage this trajectory profile. Our framework also enables getting different trajectory profiles by scaling the relative weights between cost terms as shown in Table 1 and Figure 34. We also integrate our library with Kunz and Stilman's method [4] through a python wrapper (github_link) to output bounded velocity and acceleration trajectories for applications that allow for large jerks.

Constraints	Acceleration-l2	Jerk-l2	Effect
None	None	None	Starts with non-zero velocity.
None	All timesteps	None	Starts with non-zero velocity.
$\dot{x}_{t\in[0,T]} = 0$	None	None	Not smooth.
$\dot{x}_{t\in[0,T]} = 0$	All timesteps	None	Large Jerk at first, last t.
$\dot{x}_{t\in[0,T]} = 0$	All timesteps	All timesteps	Low jerk.
$\dot{x}_{t\in[0,3],[T-3,T]} = 0$	All timesteps	All timesteps	Low jerk, hard to optimize.
$x_{t\in[1,3]}, x_{t\in[T-3,T-1]} = x_0, x_T$	All timesteps	All timesteps	Low jerk, easier to opimize.

Table 1: Trajectory profiles by penalizing position derivatives.



(a) Kunz and Stillman [4] (b) Minimum acceleration (c) Minimum jerk

Figure 34: We show the different trajectory profiles possible leveraging cuRobo's trajectory optimization. The first way to get a bounded acceleration and velocity profile is by using Kunz and Stilman's trajectory post processing [4], shown in (a). This can have very high jerks due to instantaneous accelerations. We can obtain a better trajectory profile in cuRobo using a minimum acceleration cost as shown in (b), which can still have instantaneous jerks. By adding a minimum jerk cost term, we can get an even smoother trajectory profile as shown in (c).

A.7 Numerical Optimization Solvers

We design our solvers to take in a rollout class instance that provides a differentiable map from optimization parameters to the total cost. Our rollout class also takes in a batch of queries and outputs a batch of total costs. This batch query-able rollout class design allows us to use it for computing the cost for all samples in a particle based solver in parallel. This also enables us to compute the differentiable maps for all line search magnitudes in parallel. We illustrate thus design of our solvers and the rollout class in Figure 35. We detail the L-BFGS step update in Algorithm 6 which follows the algorithm from Nocedal and Wright [45]. Our algorithm for particle-based optimization is in Algorithm 5.

Once we compute the step direction, we have the option to clamp the step direction by $\gamma(q_H - q_L)$ to avoid the step direction from exploding (i.e., out of numerical precision) due to large violations of soft constraints (e.g., collision term). Scaling step direction is a common technique to add robustness to a numerical solver and has also been done in trajectory optimization previously by Ratliff *et al.* [7]. In the environments and problems we tested, we did not see significant improvement with scaling the step direction and hence did not use it.

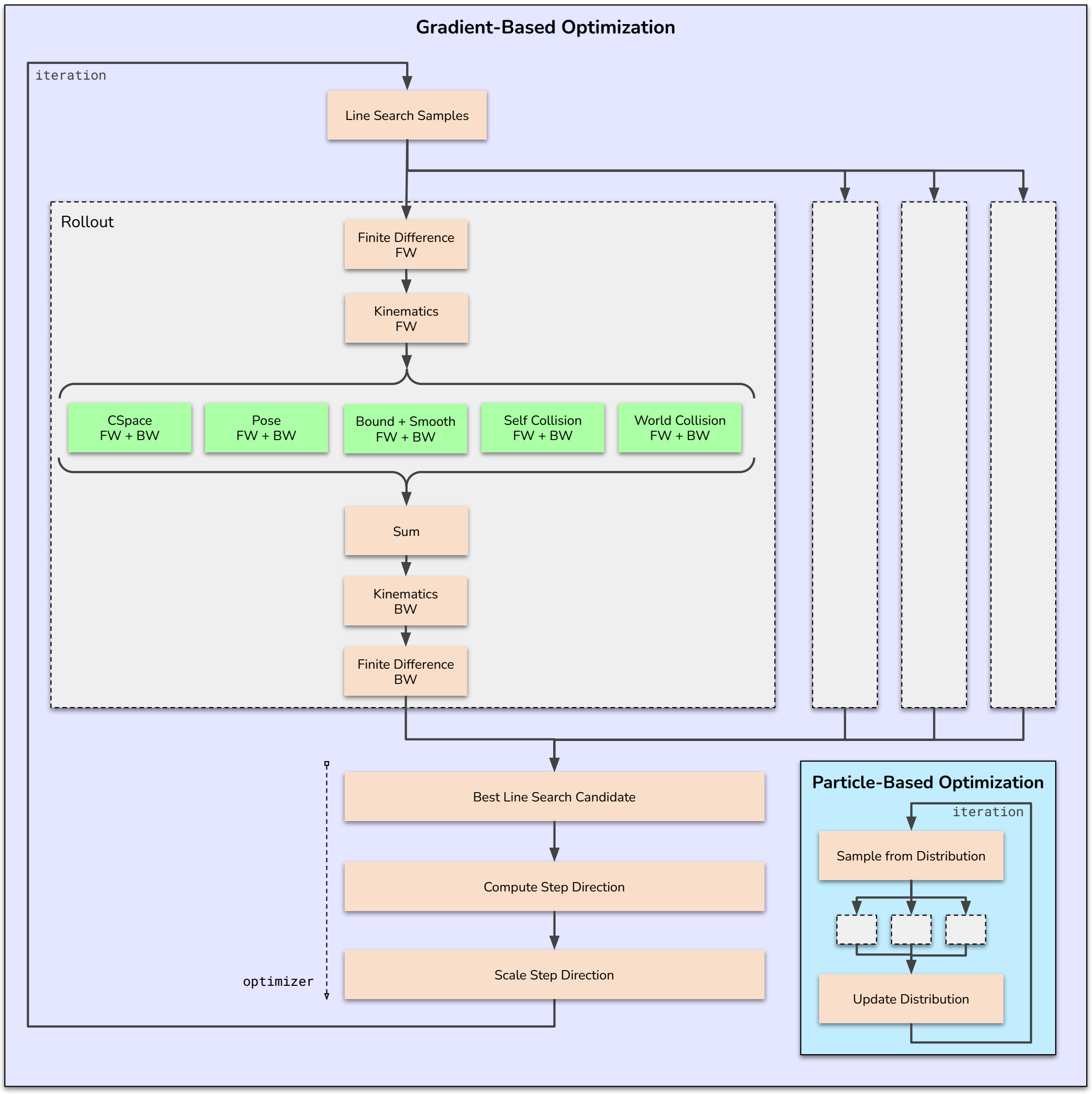


Figure 35: The compute graph in a single iteration of optimization is shown for both Gradient-based and Particle-based optimization in this figure (best viewed in color). The grey-block is the *Rollout* class that computes the cost given a batch of action trajectories. We leverage parallel execution of *Rollout* class to run parallel line search in gradient-based optimization and to compute cost across many samples in particle-based optimization. The green blocks highlight the different cost terms used for motion generation.

Algorithm 5: Particle-Based Optimization

```
  Data  : $\Theta_{\text{init}}, C(\cdot)$
  Param: $\sigma_0, c_{\min}$
  Result: $\mu$
  Init  : $\mu \leftarrow \Theta_{\text{init}}, \sigma \leftarrow \sigma_0$
1 for $n \leftarrow 0$ to $N$ do                                  ▷ Run optimization
2     $\Theta_l \leftarrow$SAMPLE$(\mu, \sigma)$
3     $c_l \leftarrow C(\Theta_l)$
4     $\mu, \sigma \leftarrow$ UPDATE$(c_l, \Theta_l, \mu, \sigma)$
5 end
```

Algorithm 6: L-BFGS Step Update

```
    Init    : $q \leftarrow \delta\Theta$
 1  $y_{[0,k]}, s_{[0,k]}, \rho_{[0,k]} \leftarrow \text{sh}(y_{[0,k]}, -1), \text{sh}(s_{[0,k]}, -1), \text{sh}(\rho_{[0,k]}, -1)$    ▷ Shift Buffers
 2  $y_k \leftarrow \delta\theta - \delta\Theta_{-1}$    ▷ Update Buffers
 3  $s_k \leftarrow \Theta - \Theta_{-1}$
 4  $\rho_k \leftarrow 1/y_k^\top s_k$
 5  $\Theta_{-1}, \delta\Theta_{-1} \leftarrow \Theta, \delta\Theta$    ▷ Store for next iteration
    /* Serial compute */
 6  for $i = k - 1$ to $k - m$ do
 7      $\alpha_i \leftarrow \rho_i s_i^\top q$
 8      $q \leftarrow q - \alpha_i y_i$
 9  end
10  $r \leftarrow H_K^0 q$
    /* Serial compute */
11  for $i = k - m$ to $k - 1$ do
12      $\beta \leftarrow \rho_i y_i^\top r$
13      $r \leftarrow r + s_i(\alpha_i - \beta)$
14  end
15  $\Delta\Theta \leftarrow r$
```

A.8 Tuning Weights & Parameters

Trajectory optimization with numerical solvers is often very sensitive to the weights used between cost terms, especially when the solvers don't solve for hard constraints directly. We found the following procedure very helpful in finding a good set of weights for the different cost terms,

1. Start the tuning process with a large number of seeds for IK (e.g., 500) and trajectory optimization (100). Also use a large number of timesteps (40) for trajectory optimization.
2. Tune pose cost term and collision cost term to solve IK.
3. Then, tune Pose cost and smoothness cost for trajectory optimization with collision costs disabled.
4. Finally, tune collision cost weights for trajectory optimization.

Once you have a good set of weights, start reducing the number of seeds until the weights do not work anymore. At this step in the process, we found increasing the weights for the pose cost improved success rate. Once we tuned our weights following these steps, the weights transferred across different environments and also across different robot platforms.

A.9 Evaluating Optimized Trajectories

Our inverse kinematics solver and trajectory optimization solver optimizes over many parallel seeds. After completing iterations, we often found more than one valid solution from these seeds. For inverse kinematics, we return the solution with a lowest weighted sum of pose error and the distance of the solution to the current joint configuration. For choosing between the valid trajectory optimization seeds, we use a blended sum of the pose error, maximum jerk, and motion time.

B Benchmarking Details

B.1 Dataset & Evaluator

We benchmarked motion planning with the 800 problems from the motion benchmaker dataset [52] and the 1600 problems from the mpinets dataset [53](Version 1.0.2). The motion planning problems available through the motion benchmaker dataset [52] had the Franka Panda's gripper at a 16cm opening which is outside of the real Franka Panda's 8cm limit. We reduce the radius of the obstacle (cylinder) that was the between the gripper by 4cm to make it fit within the Franka Panda's real gripper limits. The mpinets dataset [53] uses a gripper width of 2.5cm, which we set in our robot configuration during benchmarking.

Planning Environment	IK seeds	TO seeds	Time-steps	Force Graph
Bookshelf Small Panda	30	12	32	False
Bookshelf Tall Panda	30	12	32	False
Bookshelf Thin Panda	30	12	32	False
Box Panda	30	12	32	False
Box Panda Flipped	30	12	32	False
Cage Panda	100	16	32	False
Table Pick Panda	30	12	32	False
Table under Pick Panda	112	28	44	True
Table Top	30	12	32	False
Merged Cubby	30	12	32	False
Cubby	30	12	32	False
Dresser	30	12	32	False
Cubby-task-oriented	100	16	32	False
Dresser-task-oriented	100	20	32	False

Table 2: Planning Parameters used in *cuRobo* across the benchmark.

We retooled the evaluator that was written by [53], which is available at github.com/fishbotics/robometrics along with the dataset.

B.2 Tesseract Baseline

We used Tesseract planning with commit `1627231f3d`, compiled with `Release` CMAKE flag for our experiments. As discussed in Section A.6, we add costs to Trajopt that have the first three and last three timestep velocity to be at zero. We also set `smooth_acceleration=True` and set `smooth_velocity=False` to get the solution as mentioned by Tesseract developers (github-discussion). We tried setting `smooth_jerk=True`, however this caused the planner to fail on many planning problems and the trajectories also exhibited aliasing in the acceleration.

In addition, we couldn't run the Cartesian planning from Tesseract. We compute 20 IK solutions using cuRobo's collision-free IK solver for each problem and use one of these solutions per attempt of planning in Tesseract. We found that Tesseract failed to find many trajectories without offsetting the collision margin by -1.5cm. Hence, for all our comparisons we use a collision margin of -1.5cm. In addition, the OMPL planning phase failed to find solutions with an offset of -1.5cm, hence we further added an offset of -2cm only to the OMPL planning phase. We enable bullet continuous collision checking for the whole pipeline in Tesseract to leverage the fastest implementation of collision checking in their pipeline. We provide the Tesseract workspace with all changes at tesseract_ws and also develop a python wrapper for benchmarking which is available at github.com/balakumar-s/tesseract_wrapper.

B.3 cuRobo Parameters

We use an activation distance η of 2.5cm and use a scalar weight of 5000 for all cost terms that are soft constraints. We tuned the number of trajectory optimization (TO) seeds on a per scene level depending on average number of attempts needed to succeed when we ran with only 4 seeds. We used 30 seeds for Inverse Kinematics for most problems. For motion planning problems in *Table under Pick Panda*, we generate seeds for trajectory optimization from our geometric planner (Section 5) as we found that we need more than 100 linearly interpolated seeds to succeed. For trajectories that require many switching points, we use a larger number of timesteps. The values used across the scenes are shown in Tab. 2. We run 100 fixed iterations followed by 300 variable iterations for the trajectory optimization. We run 2 iterations of particle-based optimization before running L-BFGS for IK and Trajectory optimization. For our evaluations, we have a 2 second warm-up phase where the tensors need to be initialized, followed by CUDA Graph creation on the GPU. After this warm-up phase is complete, we can change the environment obstacles, the start configuration, and the goal before querying a solution for a motion generation problem.

C Additional Results

C.1 Comparison to pyBullet and RRTStar

In addition to comparing to *Tesseract*, we compare our method with pybullet wrapped OMPL implementations which are commonly used in many research efforts as they are accessible from Python [66]. We specifically compare to geometric planning methods, RRTConnect [55] which is a bidirectional feasible planner that is shown to be the fastest in finding a path (while the path may not be optimal) and AIT-Star [88] which is an asymptotically optimal planner that is proven to converge to the shortest path given infinite time. We use their implementations from OMPL [54] with PyBullet [24] for collision checking. We call them *PyBullet-RRTConnect* and *PyBullet-AITStar*, respectively. We only compare on the motionbenchmaker dataset [52] as we found it. To be significantly slower compared to trajectory optimization methods while also producing longer c-space path lengths as shown in Figure 36. We found *PyBullet-RRTStar* to be faster than *Tesseract-GP* which uses RRTConnect from ompl wrapped with Bullet's continuous collision checker.

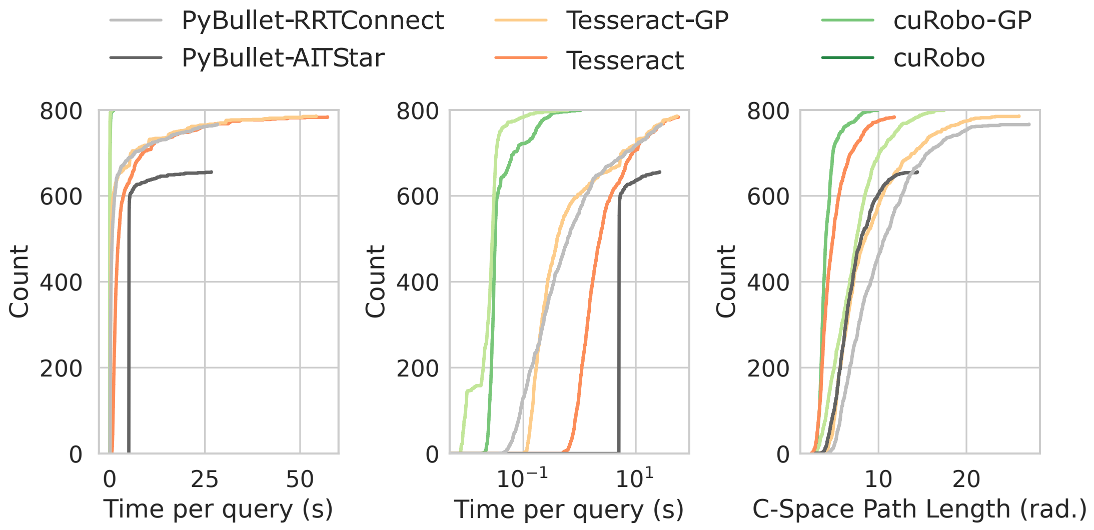


Figure 36: We compare the time taken by different motion generation methods in linear time scale on the left and log time scale in the middle. We see that *cuRobo*'s curve is significantly faster than *PyBullet*. On the right, we plot the C-Space path length across methods where we see again that *cuRobo* produces shorter paths than *PyBullet-AITStar*.

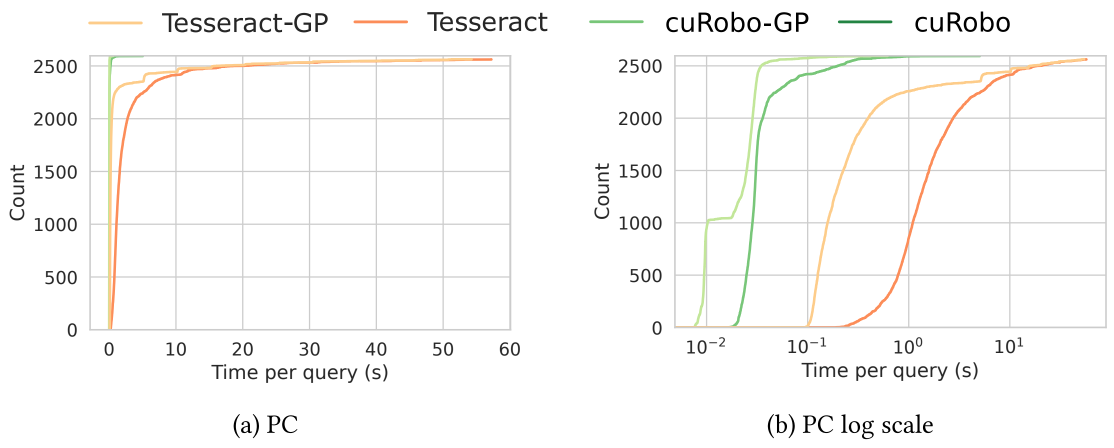
C.2 Compute Time Coverage Plot

We plot the distribution of compute time across the dataset for methods in *Tesseract* and *cuRobo* in Figure 37. From the distribution across different compute devices, we can observe that *cuRobo* is faster than both *Tesseract* and *Tesseract-GP*. Our geometric planner *cuRobo-GP* is faster than our trajectory optimization approach *cuRobo* across different compute platforms.

C.3 Real Robot Quirks

We record some real robot quirks we observed when deploying our trajectories on the Universal Robots. First, we observed poor tracking as we increased the maximum allowed acceleration during trajectory optimization, and also occasionally at peak velocities of the robot as shown in Figure 38. The UR robots come with safety features enabled by default, which restricts the speed of the robot. We overcame this issue by selecting the least conservative safety configuration which set the maximum power limit of the robot to 1000W (from 300W).

A second issue we observed with the Universal Robots was that trajectories with velocities computed through backward difference caused oscillations at high speeds. Switching to higher accuracy finite difference approaches such as central difference or five point stencil difference fixed this issue.



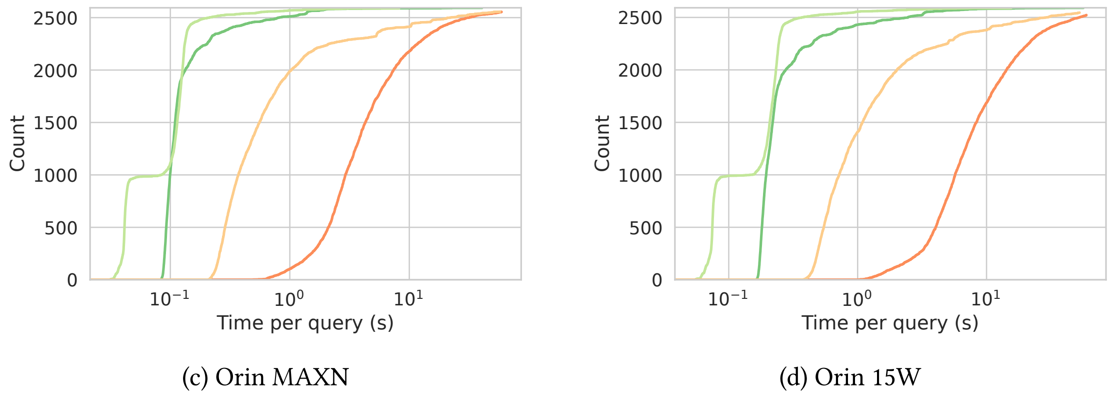


Figure 37: We compare the planning time obtained by our approach *cuRobo* to *Tesseract*.

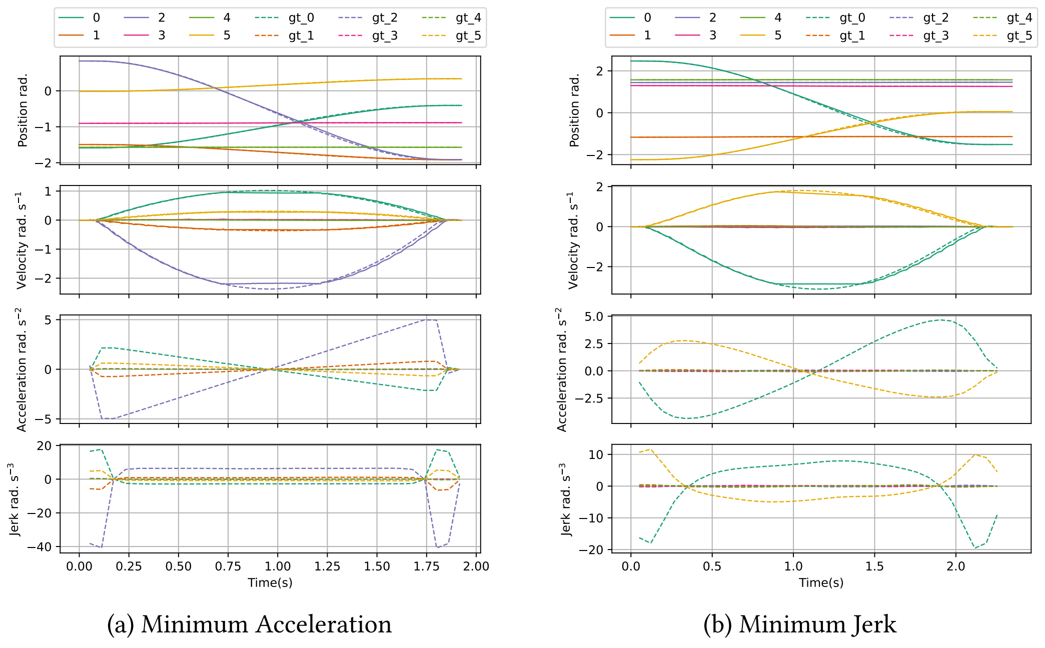


Figure 38: UR5e robot clipped peak velocities when executing some motions as show in (a) and (b). This happens when the robot's safety mode is enabled. Once we disabled the safety mode, the robot was able to execute trajectories with accelerations up to 12 radianss^{-2}, reaching maximum joint velocities.

D cuRobo Library

Implementing the algorithms discussed in this work requires a robotics toolkit that works seamlessly with CUDA code, interfaces with robot configuration files (urdf, usd), and also enables users to implement their own modules and cost terms without adding large overheads to the pipeline. Additionally, most of our algorithms run native on a GPU, requiring the framework to handle GPU tensors. We couldn't find a robotics toolkit that had these features so we built our own robotics framework *cuRobo*. We build our framework with PyTorch as the front-end enabling us to leverage the vast range of tools built by the PyTorch community. We specifically leverage the following from PyTorch:

1. Differentiable mapping between operations on tensors, enabling us to build a compute graph that contains the forward and backward passes for use with our numerical optimization solvers.
2. Profiling tools to analyze compute graphs and optimize bottlenecks.
3. CUDA code compilation and execution with pyTorch tensors enables us to write sophisticated high-performance algorithms as CUDA kernels and access them from python.
4. CUDA Graph creation and execution reduces kernel launch overheads, we found this to reduce our compute time significantly (10× faster) as we run 25 iterations of our optimization as a single CUDA graph call.

In addition to the above tools available in PyTorch, many existing GPU libraries also expose interface to PyTorch such as Warp [89], Kaolin [90], and PyPose [91], making it easier for users to use these libraries with our library. We leverage NVIDIA Warp [89] in our library to write simple user-defined cost terms such as squared l2-distance Eq. 15, our smoothness cost term Eq. 16, and our joint limit cost term Eq. 17. An example warp kernel and it's interface with PyTorch is shown in Fig. 40. We also leverage Warp's geometry kernels to compute the collision cost term for obstacles represented as meshes. We also developed a PyTorch API to nvblox [68] which we use to compute collision cost when obstacles come from a depth stream. We are releasing the PyTorch API to nvblox as a separate library and provide interfacing code for use in cuRobo.

Our library is designed as shown in Fig. 39 with a *Rollout* module for rolling out the dynamics and computing the cost terms, an *Optimization* module which contains numerical solvers, a *Geometry* module that contains signed distance functions, a *Geometric Planner* module for graph-based planning, and a wrapper module that provides a high-level api to motion generation tasks. We also provide differentiable PyTorch layers for kinematics and collision costs, for potential applications inside a neural network and also as part of a loss function motivated by recent successes in SDF-based rewards for contact-based manipulation [92].

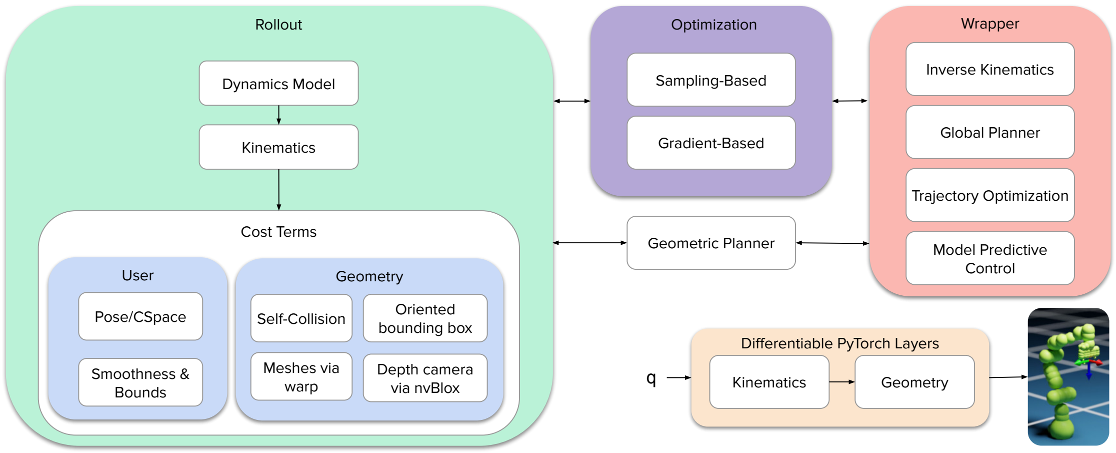


Figure 39: We illustrate the design of modules in our GPU accelerated library *cuRobo*. We have two core modules, a *Rollout* module that computes the cost given actions, and an *Optimization* module that has solvers to run numerical optimization. In addition, we have a *Wrapper* module that provides an high-level API to motion generation. We also provide differentiable PyTorch layers for kinematics and collision checking for use in neural networks.

```
1  @wp.kernel
2  def forward_l2_warp(
3      pos: wp.array(dtype=wp.float32),
4      target: wp.array(dtype=wp.float32),
5      target_idx: wp.array(dtype=wp.int32),
6      weight: wp.array(dtype=wp.float32),
7      out_cost: wp.array(dtype=wp.float32),
8      out_grad_p: wp.array(dtype=wp.float32),
9      batch_size: wp.int32,
10     horizon: wp.int32,
11     dof: wp.int32,
12 ):
13     tid = wp.tid()
14     # initialize variables:
15     b_id = wp.int32(0)
16     h_id = wp.int32(0)
17     d_id = wp.int32(0)
18     b_addrs = wp.int32(0)
19     target_id = wp.int32(0)
20     w = wp.float32(0.0)
21     c_p = wp.float32(0.0)
22     target_p = wp.float32(0.0)
23     g_p = wp.float32(0.0)
24     c_total = wp.float32(0.0)
25     # we launch batch * horizon * dof kernels
26     b_id = tid / (horizon * dof)
27     h_id = (tid - (b_id * horizon * dof)) / dof
28     d_id = tid - (b_id * horizon * dof + h_id * dof)
29     if b_id >= batch_size or h_id >= horizon or d_id
         >= dof:
30         return
31     # read weight
32     w = weight[0]
33     b_addrs = b_id * horizon * dof + h_id * dof + d_id
34     # read buffers
35     c_p = pos[b_addrs]
36     target_id = target_idx[b_id]
37     target_id = target_id * dof + d_id
38     target_p = target[target_id]
39     # calculate error
40     c_p = c_p - target_p
41     c_total = w * c_p
42     g_p = 2.0 * w
43     # write cost
44     out_cost[b_addrs] = c_total
45     # write gradient
46     out_grad_p[b_addrs] = g_p
```

(a) Warp kernel in Python

```
1  class SquaredL2DistFunction(torch.autograd.Function):
2      @staticmethod
3      def forward(
4          ctx,
5          pos,
6          target,
7          target_idx,
8          weight,
9          out_cost,
10         out_gp,
11     ):
12         wp_device = wp.device_from_torch(pos.device)
13         b, h, dof = pos.shape
14         wp.launch(
15             kernel=forward_l2_warp,
16             dim=b * h * dof,
17             inputs=[
18                 wp.from_torch(pos.view(-1), dtype=wp.
       float32),
19                 wp.from_torch(target.view(-1), dtype=
       wp.float32),
20                 wp.from_torch(target_idx.view(-1),
       dtype=wp.int32),
21                 wp.from_torch(weight, dtype=wp.float32
       ),
22                 wp.from_torch(out_cost.view(-1), dtype
       =wp.float32),
23                 wp.from_torch(out_gp.view(-1), dtype=
       wp.float32),
24                 b,
25                 h,
26                 dof,
27             ],
28             device=wp_device,
29             stream=wp.stream_from_torch(pos.device),
30           )
31           cost = torch.sum(out_cost, dim=-1)
32         ctx.save_for_backward(out_gp)
33         return cost
34     @staticmethod
35     def backward(ctx, grad_out_cost):
36         (p_grad,) = ctx.saved_tensors
37         p_g = None
38         if ctx.needs_input_grad[0]:
39             p_g = p_grad * grad_out_cost
40         return p_g, None, None, None, None, None, None
```

(b) Launching kernel from PyTorch

Figure 40: An example of using NVIDIA's Warp language for writing CUDA kernel is shown in (a), followed by it's interface with PyTorch in (b). As shown in line 29 of (b), we execute the warp kernel in the same CUDA stream as PyTorch's current stream to enable capture of kernel operations across pyTorch, Warp, and also custom CUDA kernels. This code was tested on PyTorch 1.13, along with Warp 0.9.0.

We have also developed interfacing code to read kinematic structures (e.g., robots) from URDF and USD. To make environment configuration easier, we have developed a USD parser that can read obstacles and the robot directly from a USD stage. This USD parser enables our stack to work inside simulation engines such as NVIDIA Isaac sim without requiring extensive API integration as we can directly read the robot and the obstacles from the current USD stage. We provide an example in our code that shows how cuRobo can update it's obstacles based on an Isaac Sim world, with dynamic loading of new obstacles and obstacle pose changes.

To obtain a sphere representation of the robot for collision cost terms, we leverage an existing utility from NVIDIA's Isaac sim (sphere-generator). The sphere-generator gives a UI for manually adjusting the generated spheres to more accurately match the robot's geometry. In addition, we also develop a sphere approximation algorithm that can fit spheres to a mesh leveraging volume approximation techniques as shown in Fig. 41. The first technique samples the surface of the mesh evenly. We also develop a second method that voxelizes the mesh and treats each occupied voxel as a sphere and combines this with surface sampled spheres. We currently use this technique only for approximating the geometry of objects that are attached to a gripper during manipulation to enable collision avoidance between a grasped object and the world. One limitation of this technique is that it works well only when the number of spheres is greater than 100. We do not use this automatic technique to approximate the robot's geometry as this can lead to requiring thousands of spheres, potentially slowing down our pipeline on low-power compute devices such as the

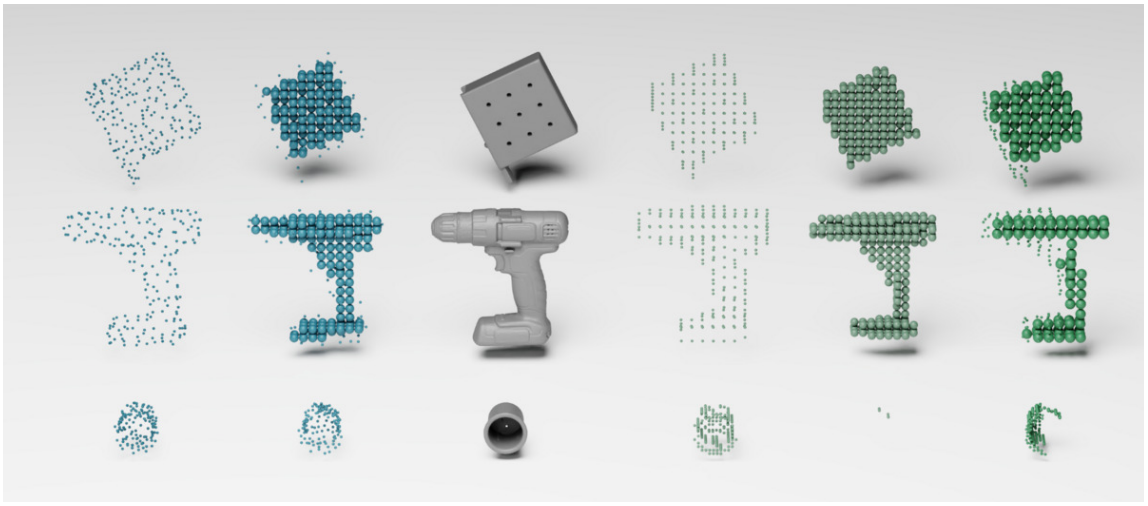


Figure 41: We illustrate some techniques to fit spheres to a mesh. We show three objects from the YCB dataset [93] in the third column. The last three columns show different ways to get spheres from a voxelization of the mesh. The fourth column projects all occupied voxels to the surface and then takes these points as surface points, the fifth column only takes voxels that are within the surface of the mesh and uses the voxel pitch to compute the radius of the spheres. The last column combines fourth and fifth column spheres to get spheres that are in the volume and also the surface. However this representation does not cover the full geometry of the object, missing some key details on the surface as seen by *cup* object in the last row. We hence only use the voxelization for computing spheres internal to the mesh and combine it with evenly sampled surface points (surface samples shown in the first column) to get a more accurate sphere representation in column 2. All approximations shown use 200 spheres per mesh.

NVIDIA Jetson ORIN. In addition, our current implementation of the self collision kernel is limited to 1024 spheres as we leverage warp-wide primitives to find the largest penetration distance.

E Parallelized Compute Kernels

GPUs are specialized for highly parallel computations as they have more transistors devoted to data processing compared to caching and flow control. This enables GPUs to hide memory access latencies in compute bottle-necked scenarios by running more parallel compute. A GPU has significantly more instruction throughput and memory bandwidth that a CPU within a similar power envelope and cost. To efficiently leverage GPU compute for motion generation, we found the following to be important:

1. Reducing reads and writes to global memory to avoid hitting memory access latency.
2. Skip writing zeros for gradients by keeping track of sparsity. As the optimization converges, most gradients will go to zero and skipping rewrites of zeros greatly reduces the memory access bottlenecks in our cost terms.
3. Reducing number of kernels called in an optimization iteration by combining small kernels into a single larger kernel that does all the work from the small kernels.
4. Using shared memory to share data across a thread block, enabling for-loops to be run in parallel.
5. Leveraging warp-wide operations for computing values across small thread groups such as reductions and finding the maximum.

We discuss the implementation of some of the key kernels in our framework in the following sections, starting with our kinematics kernels in Section E.1, followed by our self-collision kernel in Section E.2, our continuous signed distance kernel in Section E.3, and then our L-BFGS kernel in Section E.4. We then report compute times for these kernels across compute devices in Section E.5. There are many more kernels in our library and urge the readers to look at our code for the full set.

E.1 Kinematics

The role of the forward kinematics function is to map a robot's joint configuration to the pose of the robot's geometry in world coordinates. In cuRobo, we represent the robot's geometry by spheres and also provide task space poses for any links thats required in computing costs (e.g , the pose of the end-effector for motion generation) as shown in Figure 3. To perform gradient-based optimization, we require the backward mapping to project gradients from the world coordinates for poses and spheres, to the joint configuration which we call backward kinematics (i.e., this is also called as the Kinematic Jacobian). There are many ways to represent a robot's kinematics [94]. We wanted a representation that not only has less number of operations but also allows for running many operations in parallel. We found representing the transformations as 4×4 homogeneous transformation matrices to be the best representation as it enables us to run 4 parallel threads to compute matrix multiplications using shared memory.

The work for forward kinematics consists of computing a 4×4 matrix (which we call *cumul*) per link, which is then used to compute robot sphere locations for each of the links. We distribute the work per batch across a number of threads such that the compute resources are well utilized for the matrix multiplications and sphere transformations. We use four threads per batch in our implementation as shown in Algorithm 7.

For backward kinematics, the *cumul* matrix is used to compute the gradients for each of the spheres. Similar to the above implementation, we distribute the work to multiple threads (16 threads/batch) as shown in Algorithm 8. The *cumul* matrix can either be saved to memory in forward and reused by the backward kernel, or regenerated by the backward. Reusing the matrix may be slower than regenerating it for large batch sizes on systems with low memory bandwidth. We hence use a flag to choose between reusing from memory and recomputing for this kernel. We also write out the transformation matrices that map a joint value to a matrix in Table 6 and their gradients in Table 7.

E.2 Self-Collision

Our CUDA kernel checks for self-collision for a batch of robot configurations by reading the location of the spheres in the joint configuration along with the radii. Per batch, we compute distances between all pairs in the set of spheres S that represent the robot and find the pair with the minimum distance.

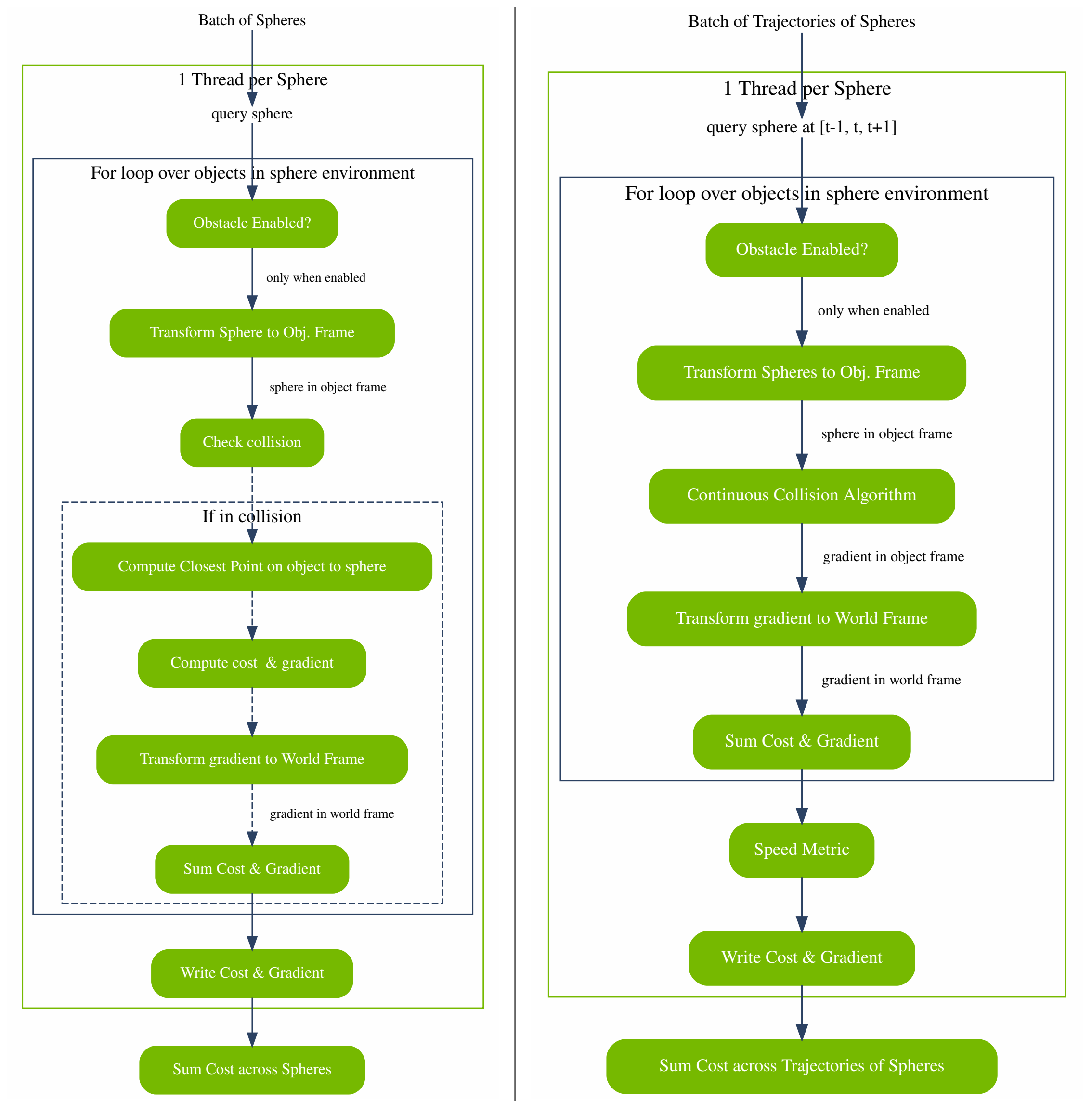
We map the work per batch to one thread-block so that the parallel reduction can be performed relatively quickly using intra-warp sharing primitives and shared memory, avoiding atomic accesses altogether. For distance computations we evaluated two versions. First version maps 32×32 distance computations to a warp such that the memory accesses are coalesced and we reuse the first loaded sphere for 32 distance computations. To avoid extra work, this version checks that the first sphere's index is greater than second sphere's index. In doing so, some of the fetches get wasted. The second version, assigns a set of distance computations (set size being 8, 16, 32, etc) per thread. The assignment consists of sphere index pairs and is pre-computed. Two spheres are fetched by index per computation. To reduce the fetch overhead, we store the spheres in shared memory. We use this version in our evaluations and is written in Algorithm 9.

E.3 Signed Distance

We distribute the collision checking work per batch across a number of threads equal to the number of spheres and horizons. Each thread loads a sphere along with two others from adjacent horizons. For each of the OBBs, the distance between the sphere and OBB is computed by first transforming the sphere to OBB coordinate frame and then checking if the sphere is within bounds as it becomes an axis-aligned bounding box (AABB). If there's a potential collision, gradient is computed. If not, we check whether a potential collision is possible between two adjacent time horizons to compute gradients based on the continuous collision checking algorithm described in Section 3. For inverse kinematics and graph planning, we only check collisions at discrete times which is written in Algorithm 10. The continuous collision kernel is written in Algorithm 11 followed by the steps in computing continuous collision distance in Algorithm 12. The flow of the algorithms are also illustrated in Figure 42.

E.4 L-BFGS

L-BFGS consists of a series of dot products to compute the step direction from the history of gradients and optimization parameters. We use many threads in one thread-block to perform the product and reduce (sum-up) the products in parallel. This will work as long as the history is less than 16. The increase in run-time with batch size will follow a step function as GPUs include multiple SMs that can execute thread-



(a) Sphere-World Signed Distance

(b) Sphere-World Continuous Signed Distance

Figure 42: The functions to compute signed distance and gradient between a query sphere and objects in the world is shown in (a). In (b), a continuous signed distance algorithm is shown for computing signed distance across a trajectory taken by a query sphere.

blocks simultaneously (e.g., Jetson Orin AGX includes 16 SMs in 60W mode). Another compute component in L-BFGS is shifting buffer elements, which when done in parallel and asynchronously requires storing the values in a temporary buffer and then updating the buffer to avoid data corruption. In our implementation, the buffer size was ≤ 32, typical size of a warp in a GPU. Based on this fact and the GPU's SIMT execution model, we leverage warp shuffle operation to shift values down, avoiding explicit memory loads/stores for temporary values [95]. This logic can also be extended to shift larger buffers by taking care of the elements at the warp boundaries.

E.5 Kernel Timing Benchmark

We profile trajectory optimization for one problem in our evaluation and write the timings for all kernels in an iteration of the L-BFGS solver in Table 3. Our trajectory optimization first runs 100 iterations with 12 parallel seeds, selects a good time-step discretization value and optimizes 1 seed for upto 300 seeds. We plot the time it takes for running 1 seed in Table 4.

We compare the kernel time for computing the world collision cost between three different world representations. First, we represent the world as oriented bounding boxes (OBB) and use our custom CUDA kernel to compute the collision cost. Second, we represent the world with meshes and use Warp's BVH based signed distance function to compute the collision cost. Third, we render depth images of the same world and build an ESDF map using nvblox and compute collision cost with this map using our custom nvblox CUDA kernel. We compute times for both 12 TO seeds and 1 TO seed in Table 5. We did not implement the continuous algorithm for the nvblox kernel and only compute collision cost at the sphere waypoint. We compute the speed metric for all kernels.

From the compute times, we observe that cuboid collision cost *OBB-Collision* kernel is much faster ($8\times$) than using meshes for collision cost. Using a mesh representation also uses more registers (207 vs 77). The slowdown with the use of meshes was even more significant on the Jetson which has lower number of SMs and lower memory bandwidth. Our nvblox kernel takes less time than even the cuboid collision kernel on a RTX 4090. We do not implement the continuous collision algorithm for nvblox, which makes the comparison weak as the cuboid and mesh kernels are doing more work than nvblox. It's still promising see that collision cost computation with nvblox does not add significant overhead compared to other world representations. We leave implementation of the continuous collision algorithm in nvblox for a future work.

E.6 Profiling CUDA Flags

PyTorch exposes optional CUDA optimizations, some require recompilation of the CUDA kernels such as `fast-math` while others only require enabling a flag at runtime such as using `tf32` for arithmetic operations. In addition, pyTorch uses it's own memory allocator by default. We compared the solve time on our 2600 problems and found no significant difference between `cudaMallocAsync` memory allocator and pyTorch's default allocator. We found that not using `tf32` made our approach 15% slower and not using `fast-math` made our approach 18% slower. In addition to flags, PyTorch also exposes CUDAGraphs which enables capturing a sequence of kernel launches on a CUDA device and replaying this captured graph for repeated calls. This fits very well to numerical optimization, where we call a sequence of functions to map the optimization variables to cost, followed by computation of step direction and a better set of optimization variables at every iteration. We found that capturing 25 iterations of our solver in a CUDAGraph and replaying this capture gave us a $10\times$ speedup compared to calling these kernels native.

Kernel	grid	block	registers	sh. mem.	Time (μs)		
					4090	Orin MAXN	Orin 15W
Tensor Step-FW	84	128	30 [36]	0	2	13	26
Tensor Step-BW	84	128	38	0	1	4	6
Kinematics-FW	48	128	33 [34]	28672	10	44	112
Kinematics-BW	192	128	39 [40]	7168	12	25	34
Self-Collision	1536	400	33 [36]	1280	7	77	208
OBB-Collision	768	128	77 [78]	0	8	64	157
Pose-Cost	12	128	61 [58]	0	2	5	6
Bound-Smooth	42	256	54	0	2	6	9
L-BFGS Step	12	224	48	3728	4	11	13
Line Search	12	224	42	260	3	8	9
Update Best	21	128	16	0	1	3	4
Reduce (4)				0	7	21	28
Elementwise (3)				0	4	28	41
Concatenate		512	21 [18]	0	2	4	6
Jit	21	128	16	0	1	3	3
All (20 kernels)					**66**	**316**	**662**

Table 3: Time taken by kernels in 1 iteration of Trajectory Optimization(12 seeds, 32 timesteps).

Kernel	grid	block	registers	sh. mem.	Time (μs)		
					4090	Orin MAXN	Orin 15W
Tensor Step-FW	7	128	30 [36]	0	2	5	6
Tensor Step-BW	7	128	38	0	1	2	2
Kinematics-FW	4	128	33 [34]	28672	9	16	17
Kinematics-BW	16	128	39 [40]	7168	4	7	7
Self-Collision	128	400	33 [36]	1280	2	9	20
OBB-Collision	64	128	77 [78]	0	4	10	19
Pose-Cost	1	128	61 [58]	0	2	3	4
Bound-Smooth	4	256	54	0	2	4	5
L-BFGS Step	1	224	48	3728	4	7	7
Line Search	1	224	42	260	3	5	5
Update Best	2	128	16	0	1	2	2
Reduce (4)				0	7	13	15
Elementwise (3)				0	3	6	7
Concatenate		512	21 [18]	0	1	3	3
Jit	7 [2, 4]	128	16	0	1	2	2
All (20 kernels)					**46**	**94**	**121**

Table 4: Time taken by kernels in 1 iteration of Trajectory Optimization(1 seed, 32 timesteps).

Kernel	grid	block	registers	Time (μs)		
				4090	Orin MAXN	Orin 15W
OBB-Collision (1-TO)	64	128	77 [78]	4	10	19
Mesh-Collision (1-TO)	68	256	207 [210]	27	115	260
nvblox-Collision* (1-TO)	64	128	56	2	-	-
OBB-Collision (12-TO)	768	128	77 [78]	8	64	157
Mesh-Collision (12-TO)	340	256	207 [210]	37	353	874
nvblox-Collision* (12-TO)	768	128	56	5	-	-

Table 5: World Collision Checking Representation.

Joint Type	Joint Transformation	Full Link Transformation
Fixed Joint	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 & f_1 & f_2 & f_3 \\ f_4 & f_5 & f_6 & f_7 \\ f_8 & f_9 & f_{10} & f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Prismatic X	$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 & f_1 & f_2 & f_0 d_x + f_3 \\ f_4 & f_5 & f_6 & f_4 d_x + f_7 \\ f_8 & f_9 & f_{10} & f_8 d_x + f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Prismatic Y	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 & f_1 & f_2 & f_1 d_y + f_3 \\ f_4 & f_5 & f_6 & f_5 d_y + f_7 \\ f_8 & f_9 & f_{10} & f_9 d_y + f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Prismatic Z	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 & f_1 & f_2 & f_2 d_z + f_3 \\ f_4 & f_5 & f_6 & f_6 d_z + f_7 \\ f_8 & f_9 & f_{10} & f_{10} d_z + f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Revolute X	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 & f_1 \cos\theta_x + f_2 \sin\theta_x & -f_1 \sin\theta_x + f_2 \cos\theta_x & f_3 \\ f_4 & f_5 \cos\theta_x + f_6 \sin\theta_x & -f_5 \sin\theta_x + f_6 \cos\theta_x & f_7 \\ f_8 & f_9 \cos\theta_x + f_{10} \sin\theta_x & -f_9 \sin\theta_x + f_{10} \sin\theta_x & f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Revolute Y	$\begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 \cos\theta_y - f_2 \sin\theta_y & f_1 & f_0 \sin\theta_y + f_2 \cos\theta_y & f_3 \\ f_4 \cos\theta_y - f_6 \sin\theta_y & f_5 & f_4 \sin\theta_y + f_6 \cos\theta_y & f_7 \\ f_8 \cos\theta_y - f_{10} \sin\theta_y & f_9 & f_8 \sin\theta_y + f_{10} \cos\theta_y & f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Revolute Z	$\begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} f_0 \cos\theta_z + f_1 \sin\theta_z & -f_0 \sin\theta_z + f_1 \cos\theta_z & f_2 & f_3 \\ f_4 \cos\theta_z + f_5 \sin\theta_z & -f_4 \sin\theta_z + f_5 \cos\theta_z & f_6 & f_7 \\ f_8 \cos\theta_z + f_9 \sin\theta_z & -f_8 \sin\theta_z + f_9 \cos\theta_z & f_{10} & f_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Table 6: Joint Transformation matrices based on axis of actuation.

Joint Type	Position Gradient	Rotation Gradient
Prismatic X	$\vec{g}_d^\top(\vec{x})$	0
Prismatic Y	$\vec{g}_d^\top(\vec{y})$	0
Prismatic Z	$\vec{g}_d^\top(\vec{z})$	0
Revolute X	$\vec{g}_d^\top(\vec{x} \times (\vec{l} - \vec{d}))$	$\vec{g}_r^\top \langle [\vec{x}]_\times R_p \rangle$
Revolute Y	$\vec{g}_d^\top(\vec{y} \times (\vec{l} - \vec{d}))$	$\vec{g}_r^\top \langle [\vec{y}]_\times R_p \rangle$
Revolute Z	$\vec{g}_d^\top(\vec{z} \times (\vec{l} - \vec{d}))$	$\vec{g}_r^\top \langle [\vec{z}]_\times R_p \rangle$

Table 7: Gradient for different joint types, where $\langle \cdot \rangle$ refers to flattening a matrix to a vector.

Algorithm 7: Forward Kinematics using 4 threads

```
Kernel Launch Data: Launch 4 threads per batch, tid = thread id
Input: q$\in \mathbb{R}^{B\times D}$
Output: link_pos$\in \mathbb{R}^{B\times N\times 3}$, link_quat$\in \mathbb{R}^{B\times N\times 4}$, b_robot_spheres$\in \mathbb{R}^{B\times M\times 4}$,
        b_cumul_mat$\in \mathbb{R}^{B\times N\times 16}$
Kinematics Data: fixedTransform$\in \mathbb{R}^{L\times 4\times 4}$, robotSpheres$\in \mathbb{R}^{M\times 4}$, linkMap$\in \mathbb{Z}^{L}$, jointMap$\in \mathbb{Z}^{L}$,
        jointMapType$\in \mathbb{Z}^{L}$, storeLinkMap$\in \mathbb{Z}^{N}$, linkSphereMap$\in \mathbb{Z}^{M}$, B,M,N,L,D
/* B = batch size, M = number of spheres, N = number of links to write, L = number of links, D = number of
   actuated joints */
1  extern shared cumul_mat    ▷ store cumul matrix in shared memory $\in \mathbb{R}^{bpb\times l\times 16}$ (bpb= batches per block)
2  col_idx = tid % 4    ▷ Using four threads per batch index
3  b_idx = tid / 4    ▷ batch index for current thread
4  m_base = b_idx * L * 16    ▷ matrix index
5  cumul_mat[m_base + col_idx * 4] = fixedTransform[col_idx * 4]    ▷ read a column of the base link matrix
6  if write_cumul then
7      b_cumul_mat[b_idx * L * 16 + col_idx * 4] = cumul_mat[m_base + col_idx * 4]
8  for l$\leftarrow$ 1to L do    ▷ loop over links
9      ft_base = l * 16
10     in_base = m_base + linkMap[l] * 16
11     out_base = m_base + l * 16
12     j_type = jointMapType[l]    ▷ read joint type
13     angle = q[b_idx * D * jointMap[l]]    ▷ joint articulation value
       /* compute local transformation matrix from articulation value using table 6 */
14     j_col = joint_transform(j_type, angle, fixedTransform[ft_base + col_idx], col_idx)
15     for i$\leftarrow$ 0 to 3 do    ▷ multiply local transform with previous link transform to get global transform
16         cumul_mat[out_base + i * 4 + col_idx] = dot(cumul_mat[in_base + i * 4], j_col)
17     end
18     if write_cumul then    ▷ write out transforms for use in backward
19         b_cumul_mat[b_idx * L * 16 + l * 16 + col_idx * 4] = cumul_mat[out_base + col_idx*4]
20 end
   /* compute sphere positions and write to memory */
21 mpt = (M + 3) / 4    ▷ spheres per thread in a batch index
22 for i$\leftarrow$ 0 to mpt do
23     m_idx = i * 4 + col_idx
24     if m_idx $\geq$ M then
25         break
26     read_cumul_idx = linkSphereMap[m_idx]    ▷ read link index for sphere
27     transform_sphere(robotSpheres[sph_idx * 4], cumul_mat[mat_base + read_cumul_idx * 16],
         b_robot_spheres[b_idx + m_idx*4])
28 end
   /* write link poses to memory */
29 for i$\leftarrow$ 0 to N do
30     l_map = storeLinkMap[i]
31     l_base = b_idx * N
32     out_mbase = m_base + l_map * 16
33     quat = mat_to_quat(cumul_mat[out_mbase])
34     link_quat[l_base * 4 + i *4 + col_idx] = quat[col_idx]    ▷ write one value per thread
35     if col_idx<3 then
36         link_pos[l_base * 3 + i* 3 + col_idx] = cumul_mat[out_mbase]
37 end
```

Algorithm 8: Backward Kinematics using 16 threads

```
Kernel Launch Data: Launch 16 threads per batch
Input: grad_link_pos, grad_link_quat, grad_spheres, global_cumul_mat
Kinematics Data: robotSpheres∈ $\mathbb{R}^{M\times 4}$, linkMap∈ $\mathbb{Z}^L$, jointMap∈ $\mathbb{Z}^L$, jointMapType∈ $\mathbb{Z}^L$,
                 storeLinkMap∈ $\mathbb{Z}^N$, linkSphereMap∈ $\mathbb{Z}^M$, B,M,N,L,D
Output: grad_out_q
1  extern shared cumul_mat
2  b_idx = tid / 16
3  elem_idx = tid % 16
4  for l ←0 to N do                                  ▷ read global cumul matrix to shared memory
5      cumul_mat[e_idx] = global_cumul_mat[b_idx * L * 16 + l * 16 + elem_idx]
6  end
7  psum_grad = []
8  mpt = (M+15)/16
9  for i in M do                                     ▷ project sphere gradients to joints
10     m_idx = elem_idx * mpt + i
11     if m_idx ≥ M then
12         break
13     loc_grad_sph = grad_spheres[(b_idx * M + m_idx) * 4]
14     if loc_grad_sph == 0 then
15         continue
16     read_cumul_idx = linkSphereMap[m_idx]
17     sphere_mem = transform_sphere(robotSpheres[m * 4], cumul_mat)
       /* assuming all joints affecting current sphere has an index below read_cumul_idx */
18     for j in [read_cumul_idx, 0] do
19         if  linkChainMap[read_cumul_idx, j] == 0 then
20             continue                              ▷ skip links that are not in current serial chain
21         j_type = jointMapType[j]
22         psum_grad[jointMap[j]] += point_backward_grad(cumul_mat[j * 16], sphere_mem,
             loc_grad_sph, j_type)
23     end
24 end
25 for i ← 0 to N do                                 ▷ project link gradients to joints
26     g_pos = grad_link_pos[(b_idx + i)*3]
27     g_quat = grad_link_quat[(b_idx + i)*4]
28     if g_pos == 0 && g_quat == 0 then
29         continue                                  ▷ skip computation when gradients are zero
30     l_map = storeLinkMap[i]
31     l_base = b_idx * N
32     out_mbase = m_base + l_map * 16
33     dpt = (l_map + 15) / 16                       ▷ number of links per thread
34     for k in [dpt, 0] do
35         j = k * 4 + elem_idx
36         if j > l_map or j < 0 or linkChainMap[l_map, j] == 0 then
37             continue                              ▷ skip links that are not in current serial chain
38         j_idx = jointMap[j]
39         j_type = jointMapType[j]
           /* compute gradient using table 7 */
40         psum_grad[j_idx] += pose_backward_grad(cumul_mat[], l_pos, g_pos, g_quat)
41     end
42 end
43 psum_grad = warpReduce(psum_grad)                 ▷ sum gradient across warp
44 grad_out_q[b_idx * D] = psum_grad                 ▷ write out gradients in a for loop on thread 0
```

Algorithm 9: Robot Self-Collision Checking using sphere-pair threads

```
Input: b_robot_spheres$\in \mathbb{R}^{b\times M\times 4}$
Output: out_distance$\in \mathbb{R}^{b}$, out_grad$\in \mathbb{R}^{b\times M\times 4}$
Data: sparse_index$\in \mathbb{B}^{b\times M}$
Kinematics Data: offsets$\in \mathbb{M}$, weight$\in \mathbb{R}$, locations$\in \mathbb{Z}^{M\times M}$
Kernel Launch Data: NDPT, NBPB, B, M, Launch (max_pairs + NDPT )/NDPT threads
b_idx = block_id * NBPB
nbpb = min(NBPB, B - b_idx)
if $nbpb == 0$ then
    return
extern shared rs_shared
if $tid<M$ then
    for $l\in [0, nbpb]$ do
        sph = b_robot_spheres[4 * ((b_idx + l) * M + tid)]    ▷ read sphere from global memory
        sph[3] += offsets[tid]                                ▷ add offset to sphere radius
        rs_shared[NBPB * tid + l] = sph                       ▷ copy sphere to shared memory
    end
sync_threads()
indices[NDPT * 2]
for $i \in [0, NDPT * 2]$ do                                  ▷ read indices of sphere pairs for this thread
    indices[i] = locations[tid * 2 * NDPT + i]
end
max_d = {d:0, i:-1, j:-1}
for (k = 0; k< NDPT; k++) do          ▷ compute sphere pair distances and store the largest in this thread
    i = indices[k*2]
    j = indices[k * 2 + 1]
    for (l = 0; l< nbpb; l++) do
        sph1 = rs_shared[NBPB * i + l]
        sph2 = rs_shared[NBPB * j + l]
        if sph1.radius ≤ 0.0 or sph2.radius ≤ 0.0 then
            continue
        dist = sphere_distance(sph1, sph2)
        if dist > max_d.d then
            max_d.d = dist
            max_d.i = i
            max_d.j = j
    end
end
w_max_d = WarpMax(max_d)                          ▷ Find the largest distance across threads in warp
if $tid< M$ then
    for (l = 0; l< nbpb; l++) do
        if $sparse\_index[(b\_idx + l) * M + tid]! = 0$ then
            out_grad[(b_idx + l) * M * 4 + tid*4] = 0             ▷ reset all gradients to zero
            sparse_index[(b_idx + l) * M + tid] = 0
    end
sync_threads()
if $tid == 0$ then
    for (l = 0; l< nbpb; l++) do
        max_d = best_d[l*32]
        for (i = 1; i< (blockid+31 )/ 32; i++) do
            if $w\_max\_d[l * 32 + i].d > max\_d$ then
                max_d = w_max_d[l * 32 + i]           ▷ Find the largest distance across different warps
        end
        if $max\_d.d ! = 0$ then
            write_distance_gradient(max_d, b_robot_spheres[b_idx * M * 4], sparse_idx[b_idx * M])
    end
```

Algorithm 10: World Collision Distance

```
Kernel Launch Data: Launch 1 thread per sphere
World Model Input: obb_bounds, obb_pose, obb_enable, max_nobs, nboxes
Collision Config Input: activation_distance, weight
Input: b_robot_spheres, env_idx, B, H, M
Output: out_distance, out_grad
Data: sparsity_idx
bid = tid / (H, M)
hid = (tid - bid * H * M) / M
sid = (tid - bid * H * M - hid * M)
sph_idx = bid * H * M + hid * M + sid                ▷ compute ids from thread indices
sph = b_robot_spheres[sph_idx]                       ▷ read sphere from global memory
if sph.radius < 0.0 then
    return   ▷ we use negative sphere radius to deactivate spheres (e.g., spheres for a grasped object)
max_dist = 0
sum_grad = 0
eta = activation_distance
sph.radius += eta                                    ▷ add activation distance to sphere radius
start_box_idx = env_idx * max_nobs
for (box_idx = 0; box_idx< nboxes; box_idx++) do     ▷ loop over obstacles
    if obb_enable[start_box_idx + box_idx] == 0 then
        continue                                     ▷ check if obstacle is enabled
    loc_obb_pose = obb_pose[start_box_idx + box_idx] ▷ read obstacle pose into register
    loc_sph = transform_sphere(loc_obb_pose, sph)    ▷ transform sphere from world frame to obstacle frame
    loc_bounds = obb_bounds[start_box_idx + box_idx] ▷ read obstacle data
    loc_bounds = loc_bounds / 2
    if check_sphere_aabb(loc_bounds, loc_sphere) then ▷ check if sphere collides with obstacle
        loc_bounds += loc_sphere.radius
        cl = compute_sphere_gradient(loc_bounds, loc_sphere, eta)
        max_dist += cl.distance
        sum_grad += project_gradient_global_frame(loc_obb_pose, cl)
end
if max_dist == 0 then
    if sparsity_idx[sph_idx] == 0 then
        return
    sparsity_idx[sph_idx] = 0
    out_grad[sph_idx * 4] = 0
    out_distance[sph_idx] = 0
end
max_dist = weight * max_dist
sum_grad = weight * sum_grad
out_distance[sph_idx] = max_dist
out_grad[sph_idx * 4] = sum_grad
sparsity_idx[sph_idx] = 1
```

Algorithm 11: World Continuous Collision Distance

```
Kernel Launch Data: Launch 1 thread per sphere
World Model Input: obb_bounds, obb_pose, obb_enable, max_nobs, nboxes
Collision Config Input: activation_distance, weight, steps, speed_dt
Input: b_robot_spheres, env_idx, B, H, M
Output: out_distance, out_grad
Data: sparsity_idx
bid = tid / (H, M)
hid = (tid - bid * H * M) / M
sid = (tid - bid * H * M - hid * M)                 ▷ compute ids from thread indices
sph1 = b_robot_spheres[(b_addrs + (hid * M ) + sid) * 4]   ▷ read sphere from global memory
if sph1.radius < 0.0 then
    return    ▷ we use negative sphere radius to deactivate spheres (e.g., spheres for a grasped object)
max_dist = 0
sum_grad = 0
sweep_fwd = False
sweep_bwd = False
eta = activation_distance
dt = speed_dt
start_box_idx = env_idx * max_nobs
sph1.radius += eta
if hid > 0 then
    sph0 = b_robot_spheres[(b_addrs + ((hid-1) * M ) + sid) * 4]
    sph0.radius += eta
    sph0_distance = sphere_distance(sph0, sph1)
    sph0_len = sph0_distance + sph0.radius * 2
    if sph0_distance > 0.0 then              ▷ read sphere position in previous time-step
        sweep_bwd = True
if hid < horizon -1 then                     ▷ read sphere position in next time-step
    sph2 = b_robot_spheres[(b_addrs + ((hid+1) * M ) + sid) * 4]
    sph2.radius += eta
    sph2_distance = sphere_distance(sph2, sph1)
    sph2_len = sph2_distance + sph2.radius * 2
    if sph2_distance > 0.0 then
        sweep_fwd = True
/* Perform continuous collision computation using Algorithm 12 */
max_dist, sum_grad = compute_continuous_collision_distance()
if max_dist == 0 then                        ▷ check if collision cost is zero
    if sparsity_idx[sph_idx] == 0 then
        return                ▷ use sparsity data to exit early if tensors are already zero
    sparsity_idx[sph_idx] = 0
    out_grad[sph_idx * 4] = 0
    out_distance[sph_idx] = 0
max_dist = weight * max_dist
sum_grad = weight * sum_grad
out_distance[sph_idx] = max_dist
out_grad[sph_idx * 4] = sum_grad
sparsity_idx[sph_idx] = 1
```

Algorithm 12: Continuous Collision Distance

```
for (box_idx = 0; box_idx< nboxes; box_idx++) do                          ▷ loop over obstacles
    if obb_enable[start_box_idx + box_idx] == 0 then
        continue ;                                                 ▷ check if obstacle is enabled
    in_obb_pose = obb_pose[start_box_idx + box_idx] ;                        ▷ read obstacle pose
    loc_sph = transform_sphere(in_obb_pose, sph);  ▷ transform sphere from world frame to obstacle frame
    loc_bounds = obb_bounds[start_box_idx + box_idx];
    loc_bounds = loc_bounds / 2;
    if check_sphere_aabb(loc_bounds, loc_sphere) then               ▷ check if sphere collides with obstacle
        loc_bounds += loc_sphere.radius;
        cl = compute_sphere_gradient(loc_bounds, loc_sphere, eta);
        max_dist += cl.distance;
        sum_grad += project_gradient_global_frame(in_obb_pose, cl);
        jump_distance = sph1.radius ;                     ▷ start with a jump distance of sphere radius
    else
        jump_distance = compute_distance(loc_bounds, loc_sphere) ;        ▷ compute distance to obstacle
    end
    jump_d = jump_distance;
    if sweep_bwd && jump_d < sph0_distance then
        loc_sph0 = transform_sphere(in_obb_pose, sph0);
        for (j = 0; j< steps; j++) do                                          ▷ loop over sweep steps
            if jump_d ≥ sph0_distance then
                break ;   ▷ jump distance is greater than half distance between current and previous time-steps
            k0 = 1 - jump_d / (sph0_len);
            compute_jump_distance(loc_sph, loc_sph0, k0, eta, loc_bounds, grad_loc_bounds,
              sum_pt, jump_d);
        end
    end
    if sweep_fwd && jump_d < sph2_distance then
        loc_sph2 = transform_sphere(in_obb_pose, sph2);
        for (j = 0; j< steps; j++) do
            if jump_d ≥ sph2_distance then
                break ;        ▷ jump distance is greater than half distance between current and next time-steps
            k0 = 1 - jump_d / (sph2_len);
            compute_jump_distance(loc_sph, loc_sph2, k0, eta, loc_bounds, grad_loc_bounds,
              sum_pt, jump_d);
        end
    end
    if sum_pt.w > 0 then
        max_dist += sum_pt.w;
        project_gradient_global_frame(in_obb_mat, sum_pt, max_grad);
    end
end
```

F Changes since ICRA 2023

We have made significant improvements to this work since *cuRobo*'s publication at ICRA 2023 [96]. We added jerk minimization to our trajectory optimization as we found large jerks to trigger safety stops on the real robot and also lead to worse tracking. We also switched from using backward difference to five point stencil difference for computing derivatives of state from position. This increased the speed at which we could move the UR10, enabling acceleration, and velocity to reach the robot's limits. In addition, we now implicitly account for the zero velocity, acceleration, and jerk at the final timestep and start by duplicating states. This removed overshoot that often happens when using a central difference scheme. All these changes to our state representation reduced the number of iterations needed to converge by 50%.

We found that our evaluation dataset had goals that were not very close to the start state of the robot. This led our weights to work only for medium and long range motions. To make our approach work for small motions, we had to increase the weights for smoothness cost terms. We were unable to find a set of weights that would work for any length trajectory. To overcome this problem, we added a time-step optimization step that will re-optimize the trajectory with a dt estimate from an initial trajectory optimization. We also made changes to the Tesseract planner to closely match the trajectory optimization problem we are trying to solve. Specifically, we found that Tesseract did not optimize for trajectories that stop at the last timestep. We hence added velocity constraints to Tesseract's trajopt implementation, which enabled us to obtain a cuRobo-like trajectory profile as seen in Fig. 34-(b). This increased the planning time for Tesseract to 5.8 seconds on a desktop PC with an i7-7800x. We upgraded our desktop PC with a more recent CPU, an AMD Ryzen 9 7950x which reduced Tesseract's planning time to 2.9 seconds. The upgraded CPU also reduced our geometric planning time to 20ms. We introduced a procedure to tune weights in Appendix A.8 which allowed us to solve collision-free IK with 30 IK seeds instead of 100. This retuning of weights also enabled us to improve on position and rotation accuracy.

We also improved on our CUDA kernels, implementing higher performing versions of the algorithms. We reduced memory latency by packing vectors with 3 floats into padded float4, enabling our kernels to use vectorized loads (cuda-blog).

G Author Contributions

Balakumar Sundaralingam led the overall project, designed, and developed the *cuRobo* library. He ran evaluations in the paper, including real world experiments and wrote the paper.

Siva Kumar Sastry Hari led the effort on accelerating compute bottlenecks with high performance CUDA kernels, contributed to *cuRobo* library, and provided insights on GPU acceleration that led to design changes in *cuRobo* library. He also wrote parts of the paper.

Adam Fishman implemented the evaluator, metrics, and provided a format for loading datasets. He also evaluated the *pybullet-RRTConnect* and *pybullet-AITStar* baselines.

Caelan Garett provided insights on geometric planning and implemented an API to post process path plans from pddlstream using cuRobo.

Karl Van Wyk provided initial implementations of Jacobian computations, discussed cost shaping heuristics to improve trajectory optimization, especially when handling constraints and increasing pose reaching accuracy.

Valks Blukis implemented the nvblox pyTorch wrapper library and also developed an interface for rendering depth images from ground truth world representations.

Alexander Millane and Helen Oleynikova implemented the signed distance and closest point functions in nvblox and exposed CUDA kernels for easy use within cuRobo.

Ankur Handa advised on using MPPI as a sampling-based solver, provided insights on stencil methods for smoothing higher order derivatives and edited the paper.

Fabio Ramos provided insights on numerical optimization, discussed L-BFGS and sampling-based methods, and edited the paper.

Nathan Ratliff helped set the research direction, advised on trajectory optimization techniques including collision cost shaping, provided implementation insights to efficiently compute derivatives, and wrote parts of the paper.

Dieter Fox advised on the project, helped set the research direction, and provided ideas for experiments.

H Revision History

For the most recent version of this report, see the PDF from curobo.org. We track revisions made since the initial arxiv submission in this section.

Arxiv v2

- Changed name from 'CuRobo' to 'cuRobo'.
- Added flow chart illustration of collision checking algorithms (Figure 42).
- Fixed typo in Equation 14.
- Added citations [39, 73].

References

[1] V. S. Medeiros, E. Jelavic, M. Bjelonic, R. Siegwart, M. A. Meggiolaro, and M. Hutter, "Trajectory optimization for wheeled-legged quadrupedal robots driving in challenging terrain," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4172–4179, 2020.

[2] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[3] S. M. LaValle, *Planning Algorithms.* Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[4] T. Kunz and M. Stilman, "Time-optimal trajectory generation for path following with bounded acceleration and velocity," *Robotics: Science and Systems VIII*, pp. 1–8, 2012.

[5] ——, "Probabilistically complete kinodynamic planning for robot manipulators with acceleration limits," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2014, pp. 3713–3719.

[6] M. Toussaint, "Robot trajectory optimization using approximate inference," in *Proc. of the Int. Conf. on Machine Learning (ICML).* ACM, 2009, pp. 1049–1056.

[7] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *2009 IEEE International Conference on Robotics and Automation.* IEEE, 2009, pp. 489–494.

[8] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *2011 IEEE international conference on robotics and automation.* IEEE, 2011, pp. 4569–4574.

[9] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.

[10] M. Toussaint, "KOMO: Newton methods for k-order Markov constrained motion problems," e-Print arXiv:1407.0414, 2014.

[11] N. Ratliff, M. Toussaint, and S. Schaal, "Understanding the geometry of workspace obstacles in motion optimization," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2015.

[12] M. Mukadam, J. Dong, X. Yan, F. Dellaert, and B. Boots, "Continuous-time gaussian process motion planning via probabilistic inference," *The International Journal of Robotics Research*, vol. 37, no. 11, pp. 1319–1340, 2018.

[13] M. Posa and R. Tedrake, "Direct trajectory optimization of rigid body dynamical systems through contact," in *Algorithmic foundations of robotics X.* Springer, 2013, pp. 527–542.

[14] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[15] T. Apgar, P. Clary, K. Green, A. Fern, and J. W. Hurst, "Fast online trajectory optimization for the bipedal robot cassie." in *Robotics: Science and Systems*, vol. 101, 2018, p. 14.

[16] B. Sundaralingam and T. Hermans, "Relaxed-rigidity constraints: kinematic trajectory optimization and collision avoidance for in-grasp manipulation," *Autonomous Robots*, vol. 43, no. 2, pp. 469–483, 2019.

[17] J. Ichnowski, M. Danielczuk, J. Xu, V. Satish, and K. Goldberg, "Gomp: Grasp-optimized motion planning for bin picking," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 5270–5277.

[18] D. P. Bertsekas, *Reinforcement Learning and Optimal Control.* Athena Scientific, 2018.

[19] N. Hansen, "The CMA evolution strategy: A tutorial," *CoRR*, vol. abs/1604.00772, 2016. [Online]. Available: http://arxiv.org/abs/1604.00772

[20] "Tesseract," https://github.com/tesseract-robotics/tesseract, accessed: 2022-09-05.

[21] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa, "Chomp: Covariant hamiltonian optimization for motion planning," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1164–1193, 2013.

[22] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "Robot motion planning on a chip," in *Proceedings of Robotics: Science and Systems*, AnnArbor, Michigan, June 2016.

[23] P. Beeson and B. Ames, "Trac-ik: An open-source library for improved solving of generic inverse kinematics," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids).* IEEE, 2015, pp. 928–935.

[24] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.

[25] J. Pan and D. Manocha, "Gpu-based parallel collision detection for fast motion planning," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 187–200, 2012.

[26] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation.* IEEE, 2012, pp. 3859–3866.

[27] L. Montaut, Q. Lidec, V. Petrík, J. Sivic, and J. Carpentier, "Collision Detection Accelerated: An Optimization Perspective," in *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022.

[28] K. Tracy, T. A. Howell, and Z. Manchester, "Differentiable collision detection for a set of convex primitives," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

[29] M. Greenspan and N. Burtnyk, "Obstacle count independent real-time collision avoidance," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2. IEEE, 1996, pp. 1073–1080.

[30] K. Van Wyk, M. Xie, A. Li, M. A. Rana, B. Babich, B. Peele, Q. Wan, I. Akinola, B. Sundaralingam, D. Fox *et al.*, "Geometric fabrics: Generalizing classical mechanics to capture the physics of behavior," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 3202–3209, 2022.

[31] S. Redon, M. C. Lin, D. Manocha, and Y. J. Kim, "Fast Continuous Collision Detection for Articulated Models," *Journal of Computing and Information Science in Engineering*, vol. 5, no. 2, pp. 126–137, 02 2005. [Online]. Available: https://doi.org/10.1115/1.1884133

[32] Y. J. Kim, G. Varadhan, M. C. Lin, and D. Manocha, "Fast swept volume approximation of complex polyhedral models," in *Proceedings of the eighth ACM symposium on Solid modeling and applications*, 2003, pp. 11–22.

[33] S. Redon, A. Kheddar, and S. Coquillart, "Fast continuous collision detection between rigid bodies," in *Computer Graphics Forum*, vol. 21, no. 3, 2002, pp. 279–287.

[34] M. V. der Merwe, Q. Lu, B. Sundaralingam, M. Matak, and T. Hermans, "Learning Continuous 3D Reconstructions for Geometrically Aware Grasping," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2020. [Online]. Available: https://sites.google.com/view/reconstruction-grasp/home

[35] J. Ortiz, A. Clegg, J. Dong, E. Sucar, D. Novotny, M. Zollhoefer, and M. Mukadam, "iSDF: Real-Time Neural Signed Distance Fields for Robot Perception," in *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022.

[36] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[37] J. R. Bruce, "Real-time motion planning and safe navigation in dynamic multi-robot environments," *Ph. D. dissertation*, 2006.

[38] S. Quinlan and O. Khatib, “Elastic bands: Connecting path planning and control,” in *[1993] Proceedings IEEE International Conference on Robotics and Automation.* IEEE, 1993, pp. 802–807.

[39] A. Millane, H. Oleynikova, E. Wirbel, R. Steiner, V. Ramasamy, D. Tingdahl, and R. Siegwart, “nvblox: Gpu-accelerated incremental signed distance field mapping,” 2023.

[40] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, “Deepsdf: Learning continuous signed distance functions for shape representation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 165–174.

[41] Z. Tang, B. Sundaralingam, J. Tremblay, B. Wen, Y. Yuan, S. Tyree, C. Loop, A. Schwing, and S. Birchfield, “Rgb-only reconstruction of tabletop scenes for collision-free manipulator control,” in *2023 IEEE International Conference on Robotics and Automation*, 2023.

[42] A. Lambert and B. Boots, “Entropy regularized motion planning via stein variational inference,” *ArXiv*, vol. abs/2107.05146, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235795323

[43] T. Schmidt, R. Newcombe, and D. Fox, “DART: Dense Articulated Real-time Tracking with consumer depth cameras,” *Autonomous Robots*, vol. 39, no. 3, pp. 239–258, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10514-015-9462-z

[44] F. Dellaert, “Factor graphs: Exploiting structure in robotics,” *Annual Review of Control; Robotics; and Autonomous Systems*, 2021.

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

[46] E. Todorov and W. Li, “A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems,” in *In proceedings of the American Control Conference*, vol. 1, 2005, pp. 300–306.

[47] M. Welling and Y. W. Teh, “Bayesian learning via stochastic gradient langevin dynamics,” in *Proceedings of the 28th international conference on machine learning (ICML-11).* Citeseer, 2011, pp. 681–688.

[48] Y.-A. Ma, T. Chen, and E. Fox, “A complete recipe for stochastic gradient mcmc,” in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015.

[49] N. Wagener, C.-A. Cheng, J. Sacks, and B. Boots, “An online learning approach to model predictive control,” *arXiv preprint arXiv:1902.08967*, 2019.

[50] S. S. Srinivasa, A. M. Johnson, G. Lee, M. C. Koval, S. Choudhury, J. E. King, C. M. Dellin, M. Harding, D. T. Butterworth, P. Velagapudi *et al.*, “A system for multi-step mobile manipulation: Architecture, algorithms, and experiments,” in *International Symposium on Experimental Robotics.* Springer, 2016, pp. 254–265.

[51] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, “Batch informed trees (bit): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs,” in *2015 IEEE international conference on robotics and automation (ICRA).* IEEE, 2015, pp. 3067–3074.

[52] C. Chamzas, C. Quintero-Pena, Z. Kingston, A. Orthey, D. Rakita, M. Gleicher, M. Toussaint, and L. E. Kavraki, “Motionbenchmaker: A tool to generate and benchmark motion planning datasets,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 882–889, 2021.

[53] A. Fishman, A. Murali, C. Eppner, B. Peele, B. Boots, and D. Fox, “Motion policy networks,” in *Proceedings of the 6th Conference on Robot Learning (CoRL)*, 2022.

[54] I. A. Sucan, M. Moll, and L. E. Kavraki, “The open motion planning library,” *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[55] J. J. Kuffner and S. M. LaValle, “Rrt-connect: An efficient approach to single-query path planning,” in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 2. IEEE, 2000, pp. 995–1001.

[56] D. Coleman, I. Sucan, S. Chitta, and N. Correll, “Reducing the barrier to entry of complex robotic software: a moveit! case study,” *arXiv preprint arXiv:1404.3785*, 2014.

[57] W. Thomason*, Z. Kingston*, and L. E. Kavraki, “Motions in microseconds via vectorized sampling-based planning,” in *arxiv*, 2023.

[58] S. Starke, N. Hendrich, and J. Zhang, “Memetic evolution for generic full-body inverse kinematics in robotics and animation,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 3, pp. 406–420, 2018.

[59] M. Bhardwaj, S. Choudhury, B. Boots, and S. Srinivasa, "Leveraging experience in lazy search," *Autonomous Robots*, vol. 45, no. 7, pp. 979–996, 2021.

[60] 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 *IEEE International Symposium on System Integrations (SII)*, 2019.

[61] M. Bhardwaj, B. Sundaralingam, A. Mousavian, N. D. Ratliff, D. Fox, F. Ramos, and B. Boots, "STORM: An Integrated Framework for Fast Joint-Space Model-Predictive Control for Reactive Manipulation," in *Proceedings of the 5th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, vol. 164. PMLR, 2022, pp. 750–759. [Online]. Available: https://proceedings.mlr.press/v164/bhardwaj22a.html

[62] F. Meier, A. Wang, G. Sutanto, Y. Lin, and P. Shah, "Differentiable and learnable robot models," *arXiv preprint arXiv:2202.11217*, 2022.

[63] A. Makhal and A. K. Goins, "Reuleaux: Robot base placement by reachability analysis," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 137–142.

[64] A. Murali, A. Mousavian, C. Eppner, A. Fishman, and D. Fox, "Cabinet: Scaling neural collision detection for object rearrangement with procedural scene generation," in *International Conference on Robotics and Automation (ICRA)*, 2023.

[65] Y. Zhu, J. Tremblay, S. Birchfield, and Y. Zhu, "Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 6541–6548.

[66] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 440–448.

[67] renishaw, "Universal Robots Joint Encoder," https://www.renishaw.com/en/aksim-supports-universal-robots-for-smart-factory-automation--40389, accessed: 2023-08-23.

[68] NVIDIA, "GitHub - nvidia-isaac/nvblox: A GPU-accelerated TSDF and ESDF library for robots equipped with RGB-D cameras," https://github.com/nvidia-isaac/nvblox, 2022, accessed: 2022-09-14.

[69] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: A framework for pose-constrained manipulation planning," *The International Journal of Robotics Research*, vol. 30, no. 12, pp. 1435–1460, 2011.

[70] S. Alatartsev, S. Stellmacher, and F. Ortmeier, "Robotic task sequencing problem: A survey," *Journal of intelligent & robotic systems*, vol. 80, pp. 279–298, 2015.

[71] I. Gentilini, F. Margot, and K. Shimada, "The travelling salesman problem with neighbourhoods: Minlp solution," *Optimization Methods and Software*, vol. 28, no. 2, pp. 364–378, 2013.

[72] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual review of control, robotics, and autonomous systems*, vol. 4, pp. 265–293, 2021.

[73] M. Görner, R. Haschke, H. Ritter, and J. Zhang, "Moveit! task constructor for task-level motion planning," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 190–196.

[74] Z. Manchester and S. Kuindersma, "Variational contact-implicit trajectory optimization," in *Robotics Research: The 18th International Symposium ISRR*. Springer, 2020, pp. 985–1000.

[75] 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.

[76] B. Plancher and S. Kuindersma, "A performance analysis of parallel differential dynamic programming on a gpu," in *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer, 2020, pp. 656–672.

[77] ——, "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*, 2019.

[78] A. Bambade, S. El-Kazdadi, A. Taylor, and J. Carpentier, "PROX-QP: Yet another Quadratic Programming Solver for Robotics and beyond," in *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022.

[79] T. A. Howell, K. Tracy, S. Le Cleac'h, and Z. Manchester, "Calipso: A differentiable solver for trajectory optimization with conic and complementarity constraints," in *The International Symposium of Robotics Research.* Springer, 2022, pp. 504–521.

[80] 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.

[81] L. Pineda, T. Fan, M. Monge, S. Venkataraman, P. Sodhi, R. Chen, J. Ortiz, D. DeTone, A. Wang, S. Anderson, J. Dong, B. Amos, and M. Mukadam, "Theseus: A Library for Differentiable Nonlinear Optimization," *arXiv preprint arXiv:2207.09442*, 2022.

[82] J. Yamada, C.-M. Hung, J. Collins, I. Havoutis, and I. Posner, "Leveraging scene embeddings for gradient-based motion planning in latent space," in *International Conference on Robotics and Automation (ICRA)*, 2023.

[83] D. Shah, N. Yang, and T. M. Aamodt, "Energy-efficient realtime motion planning," in *International Symposium on Computer Architecture*, 2023.

[84] Y.-S. Hsiao, S. Hari, B. Sundaralingam, J. Yik, T. Tambe, C. Sakr, S. Keckler, and V. Reddi, "Vapr: Variable-precision tensors to accelerate robot motion planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023.

[85] J. Ichnowski and R. Alterovitz, "Motion planning templates: A motion planning framework for robots with low-power cpus," in *2019 International Conference on Robotics and Automation (ICRA).* IEEE, 2019, pp. 612–618.

[86] A. Wedel, T. Pock, C. Zach, H. Bischof, and D. Cremers, "An improved algorithm for tv-l 1 optical flow," in *Statistical and Geometrical Approaches to Visual Motion Analysis: International Dagstuhl Seminar, Dagstuhl Castle, Germany, July 13-18, 2008. Revised Papers.* Springer, 2009, pp. 23–45.

[87] L. Berscheid and T. Kröger, "Jerk-limited real-time trajectory generation with arbitrary target states," *Robotics: Science and Systems XVII*, 2021.

[88] M. P. Strub and J. D. Gammell, "Adaptively informed trees (ait*): Fast asymptotically optimal path planning through adaptive heuristics," in *2020 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, 2020, pp. 3191–3198.

[89] NVIDIA, "GitHub - NVIDIA/warp: A Python framework for high performance GPU simulation and graphics," https://github.com/NVIDIA/warp, 2022, accessed: 2022-09-14.

[90] C. Fuji Tsang, M. Shugrina, J. F. Lafleche, T. Takikawa, J. Wang, C. Loop, W. Chen, K. M. Jatavallabhula, E. Smith, A. Rozantsev, O. Perel, T. Shen, J. Gao, S. Fidler, G. State, J. Gorski, T. Xiang, J. Li, M. Li, and R. Lebaredian, "Kaolin: A pytorch library for accelerating 3d deep learning research," https://github.com/NVIDIAGameWorks/kaolin, 2022.

[91] C. Wang, D. Gao, K. Xu, J. Geng, Y. Hu, Y. Qiu, B. Li, F. Yang, B. Moon, A. Pandey *et al.*, "Pypose: A library for robot learning with physics-based optimization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 22 024–22 034.

[92] B. Tang, M. A. Lin, I. Akinola, A. Handa, G. S. Sukhatme, F. Ramos, D. Fox, and Y. Narang, "Industreal: Transferring contact-rich assembly tasks from simulation to reality," *arXiv preprint arXiv:2305.17110*, 2023.

[93] B. Calli, A. Singh, J. Bruce, A. Walsman, K. Konolige, S. Srinivasa, P. Abbeel, and A. M. Dollar, "Yale-cmu-berkeley dataset for robotic manipulation research," *The International Journal of Robotics Research*, vol. 36, no. 3, pp. 261–268, 2017.

[94] N. T. Dantam, "Robust and efficient forward, differential, and inverse kinematics using dual quaternions," *The International Journal of Robotics Research*, vol. 40, no. 10-11, pp. 1087–1105, 2021. [Online]. Available: https://doi.org/10.1177/0278364920931948

[95] NVIDIA, "Programming Guide :: CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2022, accessed: 2022-09-14.

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