2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

CuRobo: Parallelized Collision-Free 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

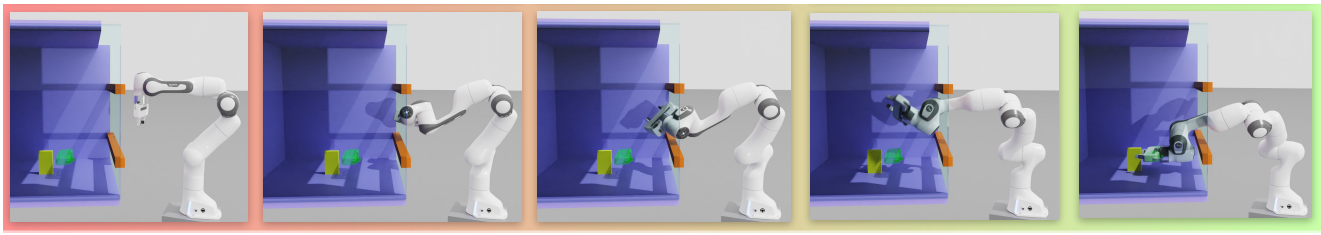

Fig. 1: We present an approach to perform motion generation for a manipulator to move from a start config to a desired gripper pose (shown as green box in the image) while avoiding collisions with the environment. Our approach leverages parallel compute to solve many instances of motion optimization from different seeds on the GPU to obtain a solution within an average time of 53ms.

***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 53ms on average, 62x faster than 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 is atleast 28x faster than SOTA RRTConnect implementations. We also introduce a collision-free IK solver that can solve over 9000 queries/s. We are releasing our GPU accelerated library *CuRobo* that contains core components for robot motion generation. Additional details are available at sites.google.com/nvidia.com/curobo.**

I. Introduction

Safe navigation is fundamental to robotics [1], but high-dimensional motion generation can quickly become prohibitively challenging as the dimensionality of the C-space and sophistication of criteria and constraints increases. 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].

*Equal Contribution. All authors are affiliated to NVIDIA. A. Fishman and D. Fox are also affiliated to University of Washington. F. Ramos is also affiliated to University of Sydney.

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 runtime is even slower on edge devices that operate under limited power budgets. This limitation mostly resulted in pipelined systems whereby a motion planner computes a single best candidate seed which is passed to an optimizer to perform a single 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 to leverage parallel processing to accelerate motion planning and optimization, and for running many optimization instances in parallel to robustly address these global optimization problems more effectively. Existing literature supports these algorithmic principles and has shows that that (1) the heuristic initialization to the problem can be effective, and (2) many restarts with randomized noise of the initial seed can dramatically improve performance. For instance, [9] studied the utility of heuristic initialization, and [21] effectively sampled many seeds in sequence using periodic random perturbations.

Massively parallel GPUs have become pervasive in both high- and low-powered configurations as they offer high throughput and energy efficiency, making them an important tool for parallelizable compute intensive problems. Even for the initial collision-free motion planning phase, paralleliza-

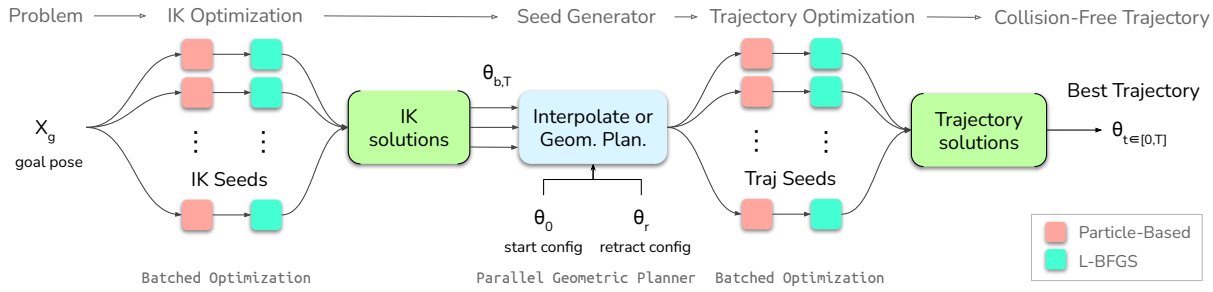

Fig. 2. Our approach to global motion generation takes as input a goal pose X_g, initial joint configuration θ_0, and computes $\theta_{b,T}$ solutions by running many parallel inverse kinematics optimization instances. We then generate seeds for trajectory optimization by linearly interpolating from the start joint configuration θ_0 to the IK solved goal configurations $\theta_{b,T}$. We use more than one IK solution in generating the seeds by leveraging the diverse IK solutions obtained from the previous step. We also generate seeds that pass through a robot retract configuration if available. We also have the option to use a Parallel Geometric Planner to get a collision-free geometric path to use as a seed on very hard problems. We then launch parallel optimization instances and choose the best trajectory $\theta_{t\in[0,T]}$ from the optimization solutions. Our Batched optimization solver first performs a few iterations of particle-based optimization to move the seed to a good region and then uses L-BFGS to converge quickly to the minimum.

tion can be effective. For example, a prior work leveraged FPGAs to implement special circuits to accelerate Probabilistic Road Map (PRM) pruning leading to many orders of magnitude speedup [22]. GPUs offer superior programmability and flexibility to map sophisticated computations of motion optimization to hardware, allowing us to parallelize the entire motion generation pipeline. We achieve similarly high speedups using GPUs compared to serial implementations. While we demonstrate the benefits of using parallel compute for motion generation using NVIDIA GPUs, the approach should be applicable to other parallel architectures. Our main contributions are as follows:

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 42x, 85x, and 116x faster for inverse kinematics, collision-free inverse kinematics, and collision-free trajectory optimization respectively when compared to existing CPU based solvers.

Parallel Geometric Planner: We implement a geometric planner with a parallel steering algorithm to generate collision-free paths within 23 ms on average, 28x faster than existing RRTConnect implementations.

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.

Validation on a Low-Power Device: We evaluate our GPU accelerated motion generation stack and existing CPU based methods on a NVIDIA Jetson AGX ORIN at different power budgets. Results show that our approach is 31x and 16x faster on average for motion generation problems when the device was set to 60W and 15W budgets, respectively.

Library: We have developed CuRobo, a suite of GPU-accelerated robot motion generation algorithms. We plan to release this library to enrich roboticists with the necessary tools to explore large-scale robot motion generation.

II. 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 paper, 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), doesn't collide with itself or the environment, and reaches the goal pose X_g at the last timestep T.

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

$$\begin{aligned}
\underset{\theta_{[1,T]}}{\arg\min} \quad & ||X_g - K_e(\theta_T)||_2 + \sum_{t=1}^{T} \gamma_1 ||\ddot{\theta}_t||_2 && (1)\\
\text{s.t.,} \quad & C_w(K_s(\theta_t)) \preceq 0, \forall t \in [1,T] && (2)\\
& C_r(K_s(\theta_t)) \preceq 0, \forall t \in [1,T] && (3)\\
& \theta^- \preceq \theta_t \preceq \theta^+, \forall t \in [1,T] && (4)\\
& \dot{\theta}^- \preceq \dot{\theta}_t \preceq \dot{\theta}^+, \forall t \in [1,T] && (5)\\
& \ddot{\theta}^- \preceq \ddot{\theta}_t \preceq \ddot{\theta}^+, \forall t \in [1,T] && (6)\\
& \dot{\theta}_T = 0 && (7)
\end{aligned}$$

where $K_e(\cdot)$ is the kinematic function that computes the pose of the end-effector given the joint configuration θ. The kinematic function $K_s(\cdot)$ computes the location of the spheres that fill the robot's volume. These spheres are used to check collisions with the world C_w and with the robot itself C_r. These collision functions (C_w, C_r) return the distance to the closest obstacle. We obtain the joint velocity $\dot{\theta}$ and joint acceleration $\ddot{\theta}$ through central difference. The first term in Eq. 1 is the pose cost. All cost terms and constraints are described in our website (sites.google.com/nvidia.com/curobo).

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 optimizing 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. 2-Eq. 3 and the joint limit constraint Eq. 4. Our overall approach is shown in Fig. 2.

III. Parallel Optimization Solver

There are several techniques to solve the optimization problem defined in Sec. II, from sampling-based optimization [8] 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. Specifically, we want a solver that can achieve superlinear convergence on our difficult optimization problem.

L-BFGS, a quasi-newton optimization method that can solve very large optimization problems, is a common powerful 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 which enables easier parallelization. Gauss-Newton solvers are also ubiquitous and important in robotics [10, 23, 24], 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, 25] 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 [26] demonstrate their empirical utility on more general costs using quadratic approximations, but appropriately leveraging the problem structure with the 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 show significant improvement over the state-of-the-art when the GPU is properly leveraged; we leave a full exploration of Gauss-Newton to future work.

A. 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[25]. We provide a detailed description of these steps in our website (sites.google.com/nvidia.com/curobo).

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. 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 can diverge because of the noisy perturbation in line search.

Algorithm 1: Parallel Noisy Line Search

Param: $\Theta, \Delta\Theta, \alpha = [0.01, ...]$

$\Theta_l \leftarrow \text{clip}(\Theta + \alpha\Delta\Theta)$ ▷ *get bounded variables*

$c_0, c_l \leftarrow c(\Theta_0), c(\Theta_l)$ ▷ *compute cost for magnitudes*

$\delta\Theta_l \leftarrow \delta c_l$ ▷ *compute batched gradients*

$a \leftarrow c_l \leq c_0 + \eta_1\alpha\Delta\Theta$ ▷ *Armijo Condition*

if *Wolfe* **then**

  if *Strong* **then** $a_2 \leftarrow \text{abs}(\delta\Theta_l\alpha) \leq \eta_2\Delta\Theta$

  else $a_2 \leftarrow \delta\Theta_l\alpha \geq \eta_2\Delta\Theta$

  $a \leftarrow a \;\&\&\; a_2$

end

$i \leftarrow \text{largest\_true}(a)$ ▷ *returns 0 when none is true*

$\hat{c}, \hat{\Theta}, \delta\hat{\Theta} \leftarrow c_l[i], \Theta_l[i], \delta\Theta_l[i]$ ▷ *return best*

B. 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 [27, 28], and sampling-based MPC controllers such as MPPI [29]. 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 then compute the cost for these particles $C(\theta_{n,[1,T]}) \in \mathbb{R}^n$. We 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,

$$\Theta_\mu = (1 - k_\mu)\Theta_{\mu-1} + (k_\mu) w * \theta_{n,[1,T]}, \tag{8}$$

$$\Theta_\sigma = (1 - k_\sigma)\Theta_{\sigma-1} + w * (\theta^2_{n,[1,T]} - \Theta_{\mu-1}). \tag{9}$$

We found that the use of particle-based optimization to initialize L-BFGS led to better convergence and reduced the number of seeds needed by L-BFGS. To further reduce the number of seeds required to converge, we generate a collision-free geometric path between start to goal using a parallelized geometric planner, described in the next section.

IV. Parallel Geometric Planner

We use 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. 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. 2.

Algorithm 2: 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 $mask \leftarrow$ mask_samples($\vec{l}$) ▷ *check for validity*
6 $h \leftarrow$ 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 $d \leftarrow$ dist($\theta_{s,0}, v_{new}$) ▷ *store distance value in edge*
10 graph_add($\theta_{b,0}, v_{new}, d$)

Equipped with this parallel steering method, we develop a geometric planner as shown in Alg. 3 that first performs heuristic planning by checking if we can steer from start to goal configuration directly [30] 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* [31] (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). We 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 can also use this geometric planner to find paths between a batch of start and goal configurations or from a single start to a goal set. This can be achieved 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).

V. Implementation Details

We implement the full stack in PyTorch (v1.12) motivated by recent libraries that have been able to leverage GPU acceleration in robotics [32–34] while also enabling easy programming of new tasks. Additionally, our PyTorch-based stack can leverage CUDA Graphs [35] as we use reference pointers for most of our tensors. Using CUDA Graphs lowers the GPU kernel setup and launching overhead, which can be a significant factor for short running kernels. This also reduces the overhead of using Python as our programming language.

Algorithm 3: 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 = [[$\theta_{b,0}$, $\theta_{b,g}$], [$\theta_{b,g}$, $\theta_{b,0}$], [$\theta_{b,0}$, θ_r], [θ_r, $\theta_{b,0}$], [$\theta_{b,g}$, θ_r], [θ_r, $\theta_{b,g}$]]
2 steer_connect(e) ▷ *Connect start, goal, and retract*
3 path_found, path, min_len $\leftarrow$ shortest_path($\theta_{b,0}$, $\theta_{b,g}$)
4 **if** *path_found* **then**
5 path, min_len, $c_{max} \leftarrow$ 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 $id \leftarrow$random(!path_found) ▷ *Pick an index from the set of queries that do not have a path yet.*
11 $\theta_{s,k} \leftarrow$ 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 $\leftarrow$ shortest_path($\theta_{b,0}$, $\theta_{b,g}$)
15 $i+ = 1$
16 **if** *path_found && min_len* > 3 **then**
17 path, min_len, $c_{max} \leftarrow$ 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

We implement custom CUDA kernels for the L-BFGS step direction computation, robot forward and backward kinematics, robot-environment and robot-self distance queries. Our environment can be represented with oriented bounding boxes, meshes and depth images + camera poses of the environment. We implement a highly optimized CUDA kernel for Sphere-OBB distance queries and integrate with an existing CUDA kernel from *nvblox* [36] for distance queries from a mapped environment (map built with a depth camera attached to the robot's end-effector). For distance queries with meshes, we write CUDA kernels in NVIDIA's warp library and integrate it with PyTorch. We go into extensive details on our CUDA kernels in our website (sites.google.com/nvidia.com/curobo).

VI. Results

We validate our approach for all benchmarks on the Franka Emika Panda robot which has 7 actuated joints. Runtimes are measured using Python's *time* utility. We evaluate on two compute platforms and three total configurations. We

use a PC with an intel i7-7800X 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 evaluate motion generation on 8 scenes from the motion benchmaker dataset [37].

A. Kinematics and Distance Queries

We plot the number of queries per second (Hz) it takes to compute forward kinematics and distance queries in Fig. 3 and compare that to a PyTorch implementation of forward kinematics from STORM [32, 33], KDL's forward kinematics available through tracikpy and Pinocchio [38], which is a state of the art for CPU-based methods. We additionally show the effect of CUDA Graphs in calling GPU methods by adding a suffix "-CG". For distance queries, we compare with two prior methods – PyBullet which uses Bullet to compute the signed distance [39] and STORM. Our method is significantly faster starting at a batch size of 1000 for forward kinematics. For distance queries, we are faster beginning at a batch size of 1 as our approach uses many parallel threads on the GPU to compute distance across many obstacles.

B. Inverse Kinematics

We compare the performance of our inverse kinematics solver against TracIK [40]. 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 [37] 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 3.64ms while TracIK only takes 2ms. However, as we increase the batch size of IK queries, we see a speedup starting from a batch size of 10.

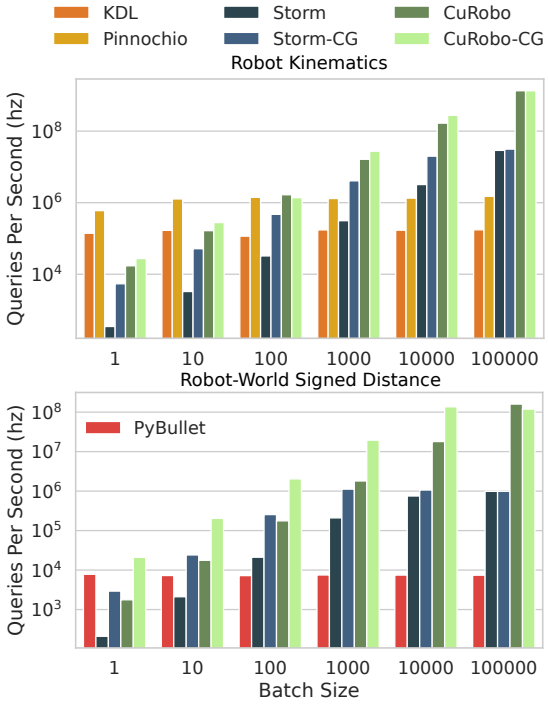

Fig. 3. In the top plot, we see our forward kinematics match CPU methods at a batch size of 100 and becomes faster by upto 891x on 100k. In the bottom plot, we see that our distance queries is upto 16,000x faster than *PyBullet* as the batch size grows to 100k. Note that the y-axis is in log scale.

For collision-free motion generation, we found a batch size of 100 for IK to sufficiently solve most problems in the motion benchmaker set. For the standard IK problem, we can generate upto 12600 solutions per second when we use a batch size of 100 while TracIK can only generate 300 solutions, 42x slower than our method. When we compare collision-free IK, our method (*CuRobo-Coll-Free*) can compute 9520 compared to rejection sampled Trac-IK (*TracIK-Coll-Free*) which can only obtain 112 collision-free solutions per second in our experiments, 85x slower than our approach to collision-free IK as shown in Fig. 5-B. While literature suggests that BioIK [41] can solve IK in 0.7ms, it is not clear whether the runtime includes collision-free IK. In either case, our method is still faster at 0.12ms for collision-free IK and 0.06ms for kinematic IK.

C. Trajectory Optimization Analysis

Before we compare our motion generation with existing methods, we analyze the impact of different components in our stack on the *cage-panda* environment shown in Fig. 1. We specifically show in Fig. 4 that increasing the number of parallel seeds significantly reduces the planning time, especially in difficult problems as indicated by the shorter lines on the tail end of the plot (i.e., closer to 100% on y-axis). We also show that increasing the parallel seeds to very large values such as 100 (*Opt-100*) results in much slower planning times throughout the test set. Through this analysis, we found the best performance in success rate and compute time can be achieved when we use 4 parallel seeds and use

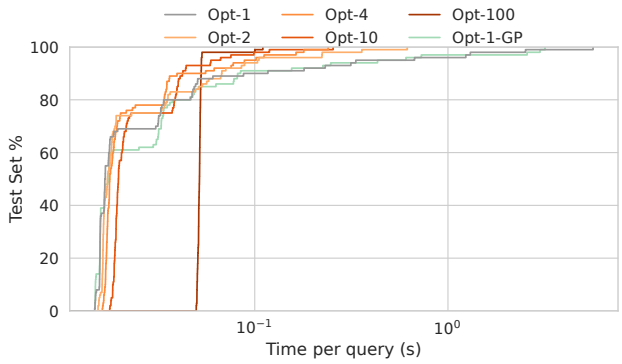

Fig. 4. We show how convergence rate is affected by the number of parallel seeds (suffixed number) we use in our optimizer. We show that adding a graph search based fallback (Opt-1-GP) does not significantly reduce planning time while increasing number of parallel seeds reduces the planning time on the tail end.

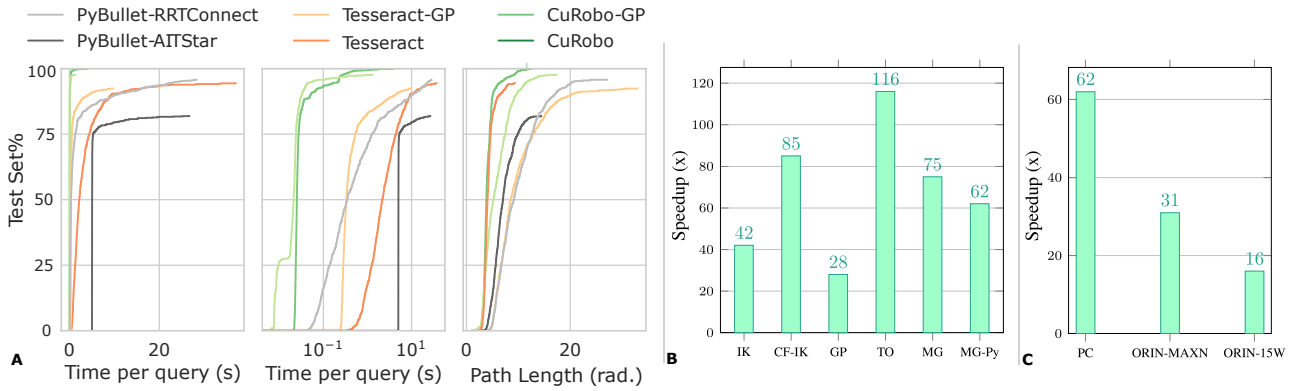

Fig. 5. We compare motion generation methods in (A), compare speedup of different components on PC in (B) and compare compute time across compute devices in (C). In (A), we see that *CuRobo*'s curve is always to the left of *Tesseract* across the full test set. In (B), we see the speedup across different components in CuRobo, where *MG-Py* shows the speedup when accounting for python overhead and post processing while *MG* referes to the speedup we achieve when only timing the optimization parts. In (C), we see our method *CuRobo* outperforming *Tesseract* across all devices, including on ORIN when running on only 15W.

geometric planning to seed only as a fallback after 3 random restarts. On harder problems (*table-pick-panda*, *table-under-pick-panda*), using 6 seeds gave us the best performance.

D. Motion Generation

We compare our method with geometric planning methods RRTConnect [42] 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 AITStar [43] which is an asymptotically optimal planner that is proven to converge to the shortest path given infinite time. We use their implementations from Open Motion Planning Library (OMPL) [44]. We use PyBullet [39] for collision checking in these methods and hence call them *PyBullet-RRTConnect* and *PyBullet-AITStar*, respectively. These methods are commonly used in many research efforts as they are accessible from Python [45].

We also compare to Tesseract [20] which uses Bullet's continuous collision detector [39], OMPL [44] for geometric planning, and TrajOpt [9] for trajectory optimization. We use two motion planning methods from Tesseract, one that only performs geometric planning which we call *Tesseract-GP* and one that uses the geometric plan as a seed to Trajopt for trajectory optimization which we call *Tesseract*.[1] We compare these baselines to two versions of motion generation from CuRobo, one that only does geometric planning *CuRobo-GP* and the other that does Trajectory Optimization *CuRobo*. For all methods, we timeout at 30 seconds and allow random restarts until then. For PyBullet and Tesseract, we use *TracIK-Coll-Free* to obtain a collision-free goal config. For our methods, we use our IK solver (*CuRobo-Coll-Free*) with 100 parallel seeds.

As seen in Fig. 5-A, our geometric planner *CuRobo-GP*'s path length is significantly shorter than AITStar while also taking only 27ms on average across the full dataset. When compared with *Tesseract-GP*, our geometric planner is 28x faster while obtaining paths that are 3.5 radians (60%) shorter in C-space on average. When comparing trajectory optimization methods, we first observe that both *Tesseract* and *CuRobo* get significantly shorter paths than geometric planning methods as seen by the gap between these methods and the others in x-axis of the C-Space path length plot in Fig. 5-A. In the middle plot of Fig. 5-A where the x-axis is plotted in log scale, we see that *CuRobo* is 62x faster, taking only 25ms on median to solve these problems, compared to 2.04s needed by *Tesseract*. We now run *Tesseract* and *CuRobo* on low-power NVIDIA Jetson AGX Orin to evaluate motion generation on an edge device that has power envelope similar to that of on-robot computers. As seen in Fig. 5-C, our approach is 31x faster at 60W and 16x faster at 15W on average.

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

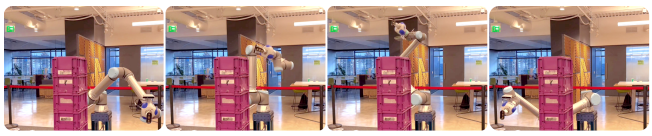

Fig. 6. We show the UR10 robot avoiding an obstacle by using CuRobo for motion generation in combination with a ESDF map that was built with *nvblox*.

We also demonstrate our approach on an UR10 robot, where we first scan the world using a realsense D-415 camera attached to the end-effector and build an ESDF map using *nvblox* [36, 46]. We then generate motions for the robot to go around obstacles using CuRobo as shown in Fig. 6.

VII. Conclusion

We formulated the global motion generation problem as an optimization problem and introduced techniques from global optimization literature to efficiently solve these motion generation problems leveraging parallel compute. Our approach achieves a 62x speedup compared to existing motion generation methods on average and can plan motions within 25ms on median. However, our 98% planning time is at 312ms while 52x faster than baselines is still much slower than our median time of 25ms. Our future work will explore reducing this gap between median and 98% planning time leveraging learning to seed optimization.

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. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," 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," 06 2016.

[23] 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

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

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

[26] 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.

[27] M. Welling and Y. W. Teh, in *Proceedings of the International Conference on Machine Learning*, L. Getoor and T. Scheffer, Eds. Omnipress, 2011, pp. 681–688.

[28] 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.

[29] N. Wagener, C.-A. Cheng, J. Sacks, and B. Boots, "An online learning approach to model predictive control,"

arXiv preprint arXiv:1902.08967, 2019.

[30] 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.

[31] 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.

[32] 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

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

[34] 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.

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

[36] ——, "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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.