# cuRoboV2-Fast: Fast Torque-Limited Collision-Free Motion Planning

Balakumar Sundaralingam  Adithyavairavan Murali  Stan Birchfield
NVIDIA

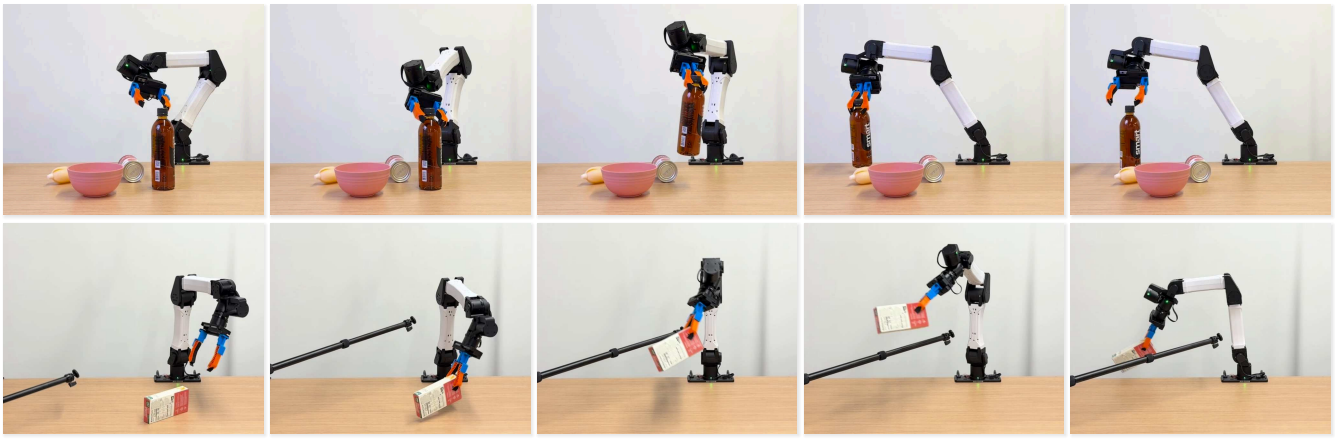


Fig. 1. Collision-free, torque-limited manipulation with cuRoboV2-Fast: payload transfer (top) and manipulation with a moving obstacle (bottom). Our stereo-camera-based deployment achieves 5.93 s mean pick-and-drop cycles, 34.6% shorter cycle time than cuRoboV2.

***Abstract*— We present cuRoboV2-Fast, a GPU-accelerated planner that reduces both planning latency and execution time for collision-free, payload-aware, torque-limited motions. B-spline trajectory optimization reduces the required iterations, while accelerated robot computations and differentiable inverse dynamics reduce refinement cost and accommodate changing payloads. TimeOpt optimizes traversal speed along the fixed path under sampled velocity, acceleration, jerk, and torque constraints, exploiting local spline support to solve banded Newton systems efficiently on the GPU. With a 3 kg payload, cuRoboV2-Fast achieves 4.4 ms mean planning time on RTX 5090, 14.7× faster than cuRoboV2. Our method on Jetson AGX Orin plans faster than cuRoboV1 + TOPPRA on RTX 5090. A stereo-camera-based deployment on YAM achieves 5.93 s mean completed pick-and-drop cycles, 34.6% shorter than cuRoboV2. Code (Apache 2.0): `github.com/NVlabs/curobo`. Project Page: `research.nvidia.com/labs/lpr/curobov2-fast/`.**

## I. Introduction

Fast manipulation requires motions that are both quick to plan and quick to execute. Motion planning has made substantial progress in finding collision-free paths through cluttered environments [1]. Executing these paths requires timing that respects velocity, acceleration, and actuator torque limits. Carried payloads change the dynamics and can render an otherwise feasible motion infeasible. Moreover, geometric path length alone does not determine execution time: a shorter path can require slower motion under actuator constraints. Efficient motion planning must therefore consider both the computational cost of generating a trajectory and the time required to execute it.

Prior work establishes the benefits of parallel planning, smooth trajectory representations, and efficient robot computations [2]–[4]. Planning executable motions nevertheless requires repeated collision and dynamics evaluations during trajectory optimization. Reducing this cost requires both fewer optimization iterations and cheaper computations within each iteration. Execution time presents a complementary challenge: uniformly rescaling the timestep cannot redistribute time along a path whose limiting actuator constraints vary with configuration. Efficient payload-aware planning therefore requires flexible optimization of trajectory geometry and timing alongside fast constraint evaluation.

We present cuRoboV2-Fast, a single-GPU planner for collision-free, torque-limited motions that reduces both planning latency and execution time. B-spline trajectory optimization refines geometry with fewer iterations, while TimeOpt optimizes traversal speed along the resulting fixed path. The timing spline's local support yields banded systems for efficient GPU optimization. A parallel GPU RRT provides route initialization when needed, and accelerated robot computations reduce the cost of refinement. Differentiable inverse dynamics incorporates payload-dependent torque limits and supports runtime inertial updates as carried objects change. We make six contributions toward fast, executable motion planning:

1) We use B-splines for collision-free, payload-aware, torque-limited trajectory optimization, matching the success of cuRoboV1 [2] with 25 versus 125 initial iterations in our convergence benchmark.
2) We introduce GPU-accelerated TimeOpt with banded Newton systems and a primal–dual predictor–corrector solver for fixed-path timing under sampled velocity, acceleration, jerk, and torque bounds. It reduces mean

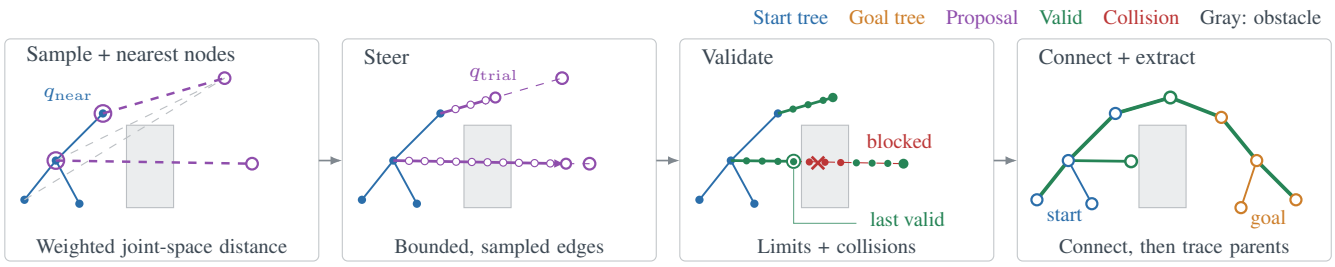



Fig. 2. Our RRT evaluates multiple tree extensions in parallel, validates their edges separately, and connects the start and goal trees to recover a route. A parallel scan inserts accepted endpoints, retaining valid prefixes of blocked edges. Purple segments show unchecked proposals, while green segments have passed validation.

motion duration by 15.7% beyond uniform timestep refinement and averages 0.28 ms on valid 100 g benchmark queries on RTX 5090.

3) We accelerate robot computations and differentiable inverse dynamics, supporting payload updates without recompilation. Trajectory optimization iterations are 2.7× faster than cuRoboV2 [5] on RTX 5090 and 2.6× faster on Jetson AGX Orin.
4) We achieve sub-millisecond route initialization with a parallel GPU RRT: 0.305 ms on average, 43× faster than cuRoboV1 PRM [2] and 1.48× faster than pRRTC [6] under coarse collision checks. Separating constraint validation from tree growth enables additional checks without changing growth kernels.
5) Our integrated pipeline achieves 4.4 ms mean motion-planning time with a 3 kg payload on RTX 5090, 14.7× faster than cuRoboV2 [5] while maintaining similar success. cuRoboV2-Fast on Jetson AGX Orin also plans faster than cuRoboV1 [2] + TOPPRA [7] on RTX 5090.
6) We demonstrate a stereo-camera-based pick-and-drop pipeline on YAM, integrating perception, grasp generation, and motion planning on a single GPU. Mean completed cycle time is 5.93 s, 34.6% shorter than cuRoboV2, with planning accounting for only 1.0% of cycle time.

We have open sourced the cuRoboV2-Fast implementation under the Apache 2.0 license at `github.com/NVlabs/curobo`.

## II. Related Work

Recent advances in motion planning exploit parallelism at several levels. Building on RRT-Connect [8], VAMP uses CPU SIMD instructions to accelerate kinematics and collision checking [1], while pRRTC distributes tree expansion, nearest-neighbor search, and edge validation across GPU threads [6]. cuRoboV1 combines geometric initialization with parallel trajectory optimization to generate collision-free motions [2]. These systems demonstrate the benefits of designing planning operations around the available hardware. Our RRT follows this direction while keeping validation separate from tree growth, so additional checks can be introduced without modifying the growth kernels.

Once an initial path is available, trajectory optimization can improve its smoothness and clearance, as demonstrated by CHOMP, STOMP, and TrajOpt [9]–[11]. The trajectory representation also shapes this refinement: B-splines have long supported collision-free smoothing and reactive deformation, using local support to modify portions of a path [3], [12]. Earlier whole-body optimization further shows how collision and torque constraints can be considered together [13]. Building on these ideas, we study how parallel spline evaluation and gradient accumulation make constrained refinement efficient within a complete planner.

Fixed-path timing methods optimize traversal speed while preserving geometry. Bobrow et al. [14] consider actuator torque bounds, and Verscheure et al. [15] formulate path timing as convex optimization. TOPPRA [7] handles velocity, acceleration, and torque constraints through backward reachability and forward construction using small linear programs. Its path-restricted torque is affine in path acceleration and squared speed, allowing dynamics coefficients to be precomputed and reused. TOPP3 [16] addresses third-order constraints by integrating maximum and minimum jerk profiles and connecting them through multiple shooting.

Our TimeOpt in cuRoboV2-Fast adopts the same fixed-path dynamics reduction and directly optimizes a cubic B-spline representation of squared speed under sampled velocity, acceleration, jerk, and torque bounds. Each sample depends on four timing controls, yielding banded Newton systems for a primal–dual predictor–corrector solver. This local support and parallel constraint evaluation enable efficient GPU optimization, with a mean TimeOpt latency of 0.28 ms across valid 100 g benchmark queries on RTX 5090.

Trajectory optimization repeatedly evaluates robot kinematics and collisions and propagates their derivatives. Efficient rigid-body algorithms and kinematic optimization libraries, including Pinocchio and PyRoki, provide reusable building blocks for this work [17], [18]. For collisions, broad-phase pruning reduces the geometry requiring detailed tests [19], while differentiable GPU evaluation supports parallel trajectory optimization in cuRoboV1 [2]. We build on these principles with shared kinematic gradient accumulation, parallel Jacobians, and pruning of self-collision and swept world queries. Their use in both trajectory refinement and grasp selection motivates evaluation at batch size one as well

as larger batches.

Torque-aware optimization adds repeated dynamics and derivative evaluations to this workload. GRiD accelerates these operations through robot-specific code generation, sparsity, and GPU parallelism [4]. We focus on directly computing gradients of the optimization loss and supplying chain schedules and inertial parameters as runtime data. This avoids forming full dynamics derivative matrices and permits payload updates without recompilation when the reduced topology is unchanged, while retaining specialization for robot dimensions. Integrating these methods with spline optimization and geometric initialization enables collision-free, torque-limited motion and grasp planning on a single GPU.

## III. Approach

Our motion planner combines collision-free initialization, B-spline optimization, TimeOpt for trajectory retiming, efficient robot computations, and differentiable inverse dynamics. Planning first refines a straight-line seed and invokes RRT when needed. Constrained motions, such as grasp approach and lift, use cost terms that heavily penalize deviations from the prescribed path.

### A. Collision-Free Initialization with RRT

When a straight-line seed is insufficient, we initialize optimization with a route from a bounded bidirectional RRT [8]. The planner grows trees from the start and goal in alternating waves, evaluating multiple extensions in parallel. Each extension samples a configuration, finds its nearest tree node, and steers toward the sample with a bounded step before checking joint limits and self/world collision (Fig. 2). Accepted endpoints extend the active tree, including the valid prefix of an otherwise blocked edge, and the opposite tree attempts to connect to them. Once the trees meet, tracing their parent chains recovers the route. The search stops when a connection is found or a preset limit on stored nodes or expansion rounds is reached.

Our implementation organizes growth into synchronized rounds of parallel extensions. In contrast, pRRTC [6] lets GPU blocks repeatedly sample and grow trees asynchronously, with greedy connection steps inside each block. Our nearest-node kernel distributes distance calculations across both stored nodes and joint coordinates, processing the tree in tiles and reducing the results to select each candidate's nearest node. In pRRTC, each thread computes complete distances for a subset of nodes before a block-wide minimum reduction. Thus, both planners parallelize nearest-node search, but distribute its work differently.

We check sampled states along all proposed edges together, then determine each edge's valid prefix. A parallel scan assigns accepted endpoints distinct storage slots, allowing concurrent insertion without reserving each slot atomically as in pRRTC. Each round also evaluates a bounded extension from the opposite tree. This organization keeps validation separate from growth, allowing additional checks without changing the growth kernels. Reusing allocated buffers and replaying growth operations through CUDA graphs reduces repeated dispatch overhead. Once a route is found, we shorten it and fit a spline seed for refinement.

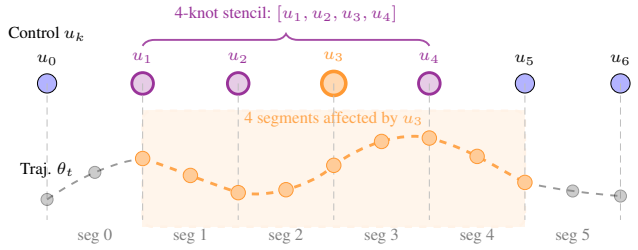



Fig. 3. Spline evaluation and backward gather: each sample uses four controls; $u_3$ gathers weighted sample gradients from the four orange segments and sums them using warp-level reduction primitives.

### B. B-Spline Trajectory Optimization

*a) Parallel evaluation.:* We use the local support of uniform cubic B-splines to parallelize trajectory evaluation and its backward pass. Let $U \in \mathbb{R}^{K\times d}$ contain the free controls for $d$ joints, $h$ be the knot interval, and $\alpha \in [0,1]$ the coordinate within segment $k$:

$$q^{(r)}(t) = h^{-r} \sum_{j=0}^{3} N_j^{(r)}(\alpha)\, \widetilde{u}_{k+j}, \qquad r = 0,1,2,3, \quad (1)$$

Here $N_j$ are cubic basis functions and $\widetilde{U}$ is the full control sequence. This established representation [3], [12] gives each sample four supporting controls (Fig. 3). We assign each sample–joint pair to a GPU thread, which evaluates position, velocity, acceleration, and jerk together from those controls. Samples therefore execute independently, with adjacent threads processing adjacent joints for coalesced memory access. Increasing the sampling density adds parallel evaluation work without increasing the number of optimized controls.

*b) Gradient accumulation.:* The backward pass reverses this mapping: each control gathers gradients only from samples within its support. In Fig. 3, $u_3$ gathers the sample gradients from the four orange segments, weights them by the corresponding basis derivatives, and sums them into its control gradient. Samples outside this region do not contribute. Writing sampled derivatives as $q_i^{(r)} = A_i^{(r)} U + b_i^{(r)}$, we accumulate

$$\nabla_U \mathcal{L} = \sum_i \sum_{r=0}^{3} (A_i^{(r)})^{\mathsf{T}}\, \nabla_{q_i^{(r)}} \mathcal{L}. \quad (2)$$

The kernel applies the basis weights directly rather than materializing the matrices $A_i^{(r)}$. Threads assigned to a control–joint pair split the sample contributions across warp lanes. Each lane combines the incoming position, velocity, acceleration, and jerk gradients with their corresponding basis weights; a warp reduction then sums the partial contributions. One lane writes the resulting control gradient. This gather avoids atomic accumulation from overlapping samples and a sample-by-control intermediate tensor, while allowing different controls and joints to be processed independently.

### C. TimeOpt: Fixed-Path Timing Optimization

In Sec. III-B, we optimize position controls at a fixed knot interval $\Delta t$ and reduce this interval between passes. This uniform reduction is limited by the most constrained part of the trajectory. TimeOpt optimizes timing along the fixed path, allowing other portions to be traversed faster while respecting actuator limits.

Let $q(s)$ be the fixed position spline, where $s \in [0, 1]$ denotes progress along the path. A second cubic B-spline represents squared path speed, with timing controls $r = (r_0, \ldots, r_{L-1}) \in \mathbb{R}^L$ as the optimization variables:

$$x_r(s) = \dot{s}^2 = \sum_{\ell=0}^{L-1} B_\ell(s) r_\ell. \tag{3}$$

Here $B_\ell(s)$ is a clamped cubic B-spline basis function and $r_\ell$ is its squared-speed coefficient. Both splines are functions of path progress $s$. Positive coefficients give $x_r(s) > 0$; forward traversal follows $\dot{s} = \sqrt{x_r(s)}$, where the dot denotes a time derivative.

TimeOpt seeks the minimum-duration traversal of this fixed path subject to velocity, acceleration, jerk, and torque limits:

$$\begin{aligned} \min_r \quad & \widehat{T}(r) = \frac{1}{N+1} \sum_{i=0}^{N} \frac{1}{\sqrt{x_r(s_i)}} \\ \text{subject to} \quad & r_\ell > 0, && \ell = 0, \ldots, L-1, \\ & |\dot{q}_r(s_i)| \le \bar{v}, && \forall i, \\ & |\ddot{q}_r(s_i)| \le \bar{a}, && \forall i, \\ & |\dddot{q}_r(s_i)| \le \bar{j}, && \forall i, \\ & \tau_{\min} \le \tau_r(s_i) \le \tau_{\max}, && \forall i. \end{aligned} \tag{4}$$

The objective averages inverse path speed at fixed uniform samples $s_i = i/N$ to approximate duration. The bounds $\bar{v}$, $\bar{a}$, $\bar{j}$, and $[\tau_{\min}, \tau_{\max}]$ limit joint velocity, acceleration, jerk, and torque, respectively. We also enforce these limits at timing knots and on both sides of internal path knots, and preserve the prescribed initial velocity and acceleration.

The timing spline determines when the robot reaches each point on the path:

$$t_r(s) = \int_0^s \frac{1}{\sqrt{x_r(u)}}\, du, \qquad q_{\text{exec}}(t) = q(s_r(t)). \tag{5}$$

Here $s_r(t)$ is the inverse of $t_r(s)$. Thus, TimeOpt changes when each path point is reached while preserving the path (Fig. 4).

We use the standard fixed-path dynamics reduction [7], [15] to evaluate dynamics once and reuse them as timing changes. Torque depends on squared path speed through the exact affine relation

$$\tau_r(s) = \tfrac{1}{2} A(s) x_r'(s) + D(s) x_r(s) + g(s), \tag{6}$$

where primes denote path derivatives. The vectors $A(s)$ and $D(s)$ capture inertial effects, and $g(s)$ is gravity torque. Using the inverse-dynamics computation in Sec. III-E, we obtain them from three evaluations per path sample: $(q, 0, 0)$, $(q, 0, q')$, and $(q, q', q'')$, yielding $g$, $A+g$, and $D+g$, respectively, as in the TOPPRA implementation [20]. Subsequent timing updates use Eq. 6 without repeating inverse dynamics. Joint velocity, acceleration, and jerk follow from the chain rule using the fixed path derivatives.

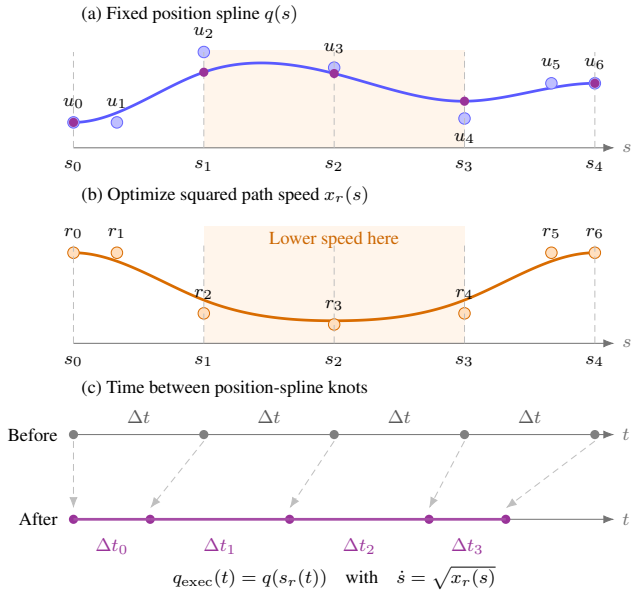



Fig. 4. TimeOpt schematic. (a) The position spline and its control points $u_i$ remain fixed; purple markers indicate positions at path knots. (b) Optimizing coefficients $r_i$ changes the squared path speed $x_r(s)$. (c) The resulting time map assigns unequal traversal times to the same path intervals. In this example, slower traversal through the shaded region and faster traversal elsewhere reduce the total duration.

We solve Eq. 4 with the primal–dual predictor–corrector method illustrated in Fig. 5. The solver updates timing controls $r$ and positive constraint multipliers $\lambda$. For each inequality, a positive margin $m_j$ measures the distance to its bound. The predictor targets zero complementarity, $m_j \lambda_j = 0$; the corrector balances these products across constraints to keep the search in the interior and accounts for their second-order change. The timing spline's local support yields banded Newton systems, and sampled constraints are evaluated in parallel on the GPU. Both direction calculations reuse the same Cholesky factorization. Backtracking selects a step satisfying the nonlinear bounds, and we retain the shortest feasible iterate. The final trajectory undergoes sampled actuator and collision checks and verification of its endpoint conditions.

TimeOpt targets efficient convergence to a local minimum and does not guarantee global optimality. It reduces mean motion duration by over 15% relative to trajectory refinement with uniform timestep rescaling (Sec. IV-B). These motion-time gains and low computational cost make TimeOpt useful for real-robot deployment.

### D. Accelerated Robot Computations

We reduce the cost of repeated robot evaluations by preparing topology-dependent data once and sharing intermediate results across computations. GPU tensor operations identify

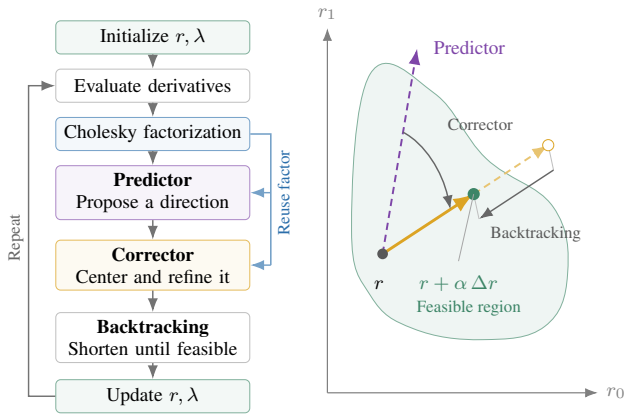



Fig. 5. TimeOpt solver. The predictor and corrector reuse one Cholesky factorization. In the schematic, the corrector changes the purple direction to yellow; backtracking shortens the step to the green feasible update.

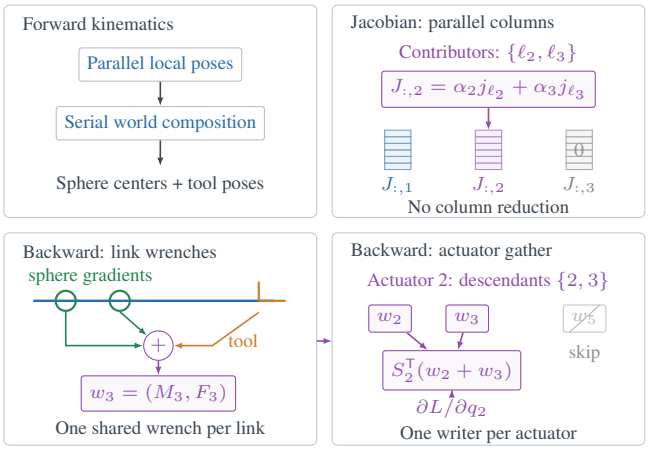



Fig. 6. Efficient kinematics: parallel local poses feed serial world composition, while prepared contributors enable independent Jacobian columns (top). The backward pass accumulates link wrenches once, then gathers them per actuator, avoiding repeated ancestor walks (bottom).

each joint's descendants and the joints contributing to each tool–actuator pair. Using this structure, forward kinematics computes local link poses in parallel, then composes world poses in tree order through shared memory. The resulting transforms provide the tool poses and collision spheres and support Jacobian evaluation.

For the backward pass, we first combine output gradients into one wrench per link. Each actuator independently gathers the wrenches of affected links, projects them onto its joint motion directions, and combines coupled-joint contributions before writing its gradient. Accumulating the link wrenches once avoids repeatedly walking the ancestor chain for every output. When IK requires an explicit Jacobian, we instead assign its columns to independent workers, each using the prepared contributor lists; trajectory optimization consumes the loss gradients directly (Fig. 6). These fast Jacobians support Levenberg–Marquardt initialization of the IK solver. In the figure, $S_2$ denotes a world-frame joint screw and $\alpha_\ell$ describes actuator-to-joint coupling.

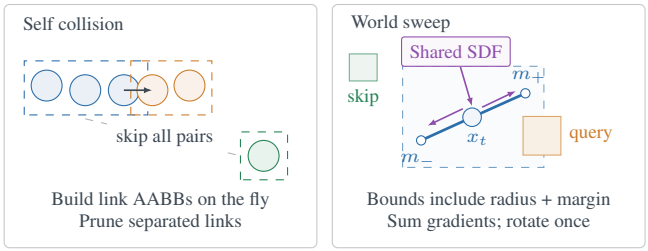



Fig. 7. Collision pruning and reuse. On-the-fly link AABBs eliminate sphere–sphere tests between separated links (left). Sweep bounds prune distant obstacles; adjacent half-sweeps share the SDF at $x_t$ and rotate their summed gradient once (right). $m_\pm$ denote neighboring midpoints.

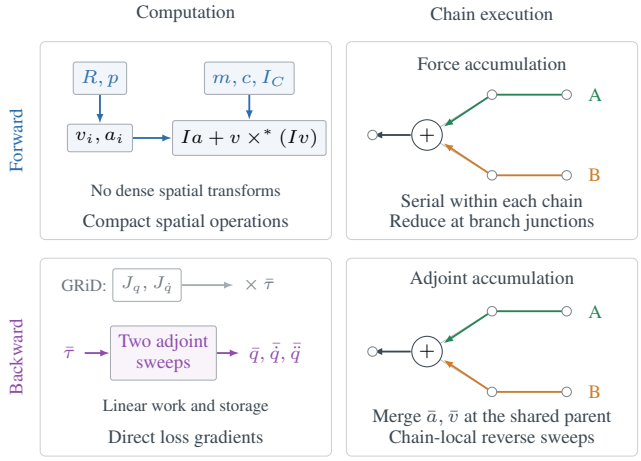



Fig. 8. Efficient inverse dynamics. Forward RNEA uses compact spatial operations (top); backward computes loss gradients directly instead of forming full derivative matrices (bottom). Independent chains run concurrently, reducing at shared parents. Trees show leaf-to-root accumulation. Payload updates require no recompilation.

Self-collision checking tests for overlap between collision spheres on the robot's links. cuRoboV1 [2] evaluates configured sphere–sphere pairs in parallel, but the number of checks can become costly for robots represented by many spheres. We compute an axis-aligned bounding box (AABB) for each link on the fly from its current sphere positions and padded radii. These bounds are built in parallel across links and stored in shared memory within the collision kernel. When two link AABBs do not overlap, we skip all sphere–sphere checks between those links. This inexpensive pruning can substantially reduce detailed pair tests, particularly when links are well separated. Each GPU block combines the remaining tests, cost reduction, and sphere gradients for one robot state.

Swept world collision uses the same principle to prune distant cuboids. Bounds that include the sphere radius, activation margin, and adjacent half-segments first reject distant cuboids. The remaining half-sweeps share the distance evaluation at their common trajectory point and combine their gradients before a single rotation to world coordinates (Fig. 7). This reduces repeated geometry calculations while retaining the chosen discrete sampling resolution for validation.

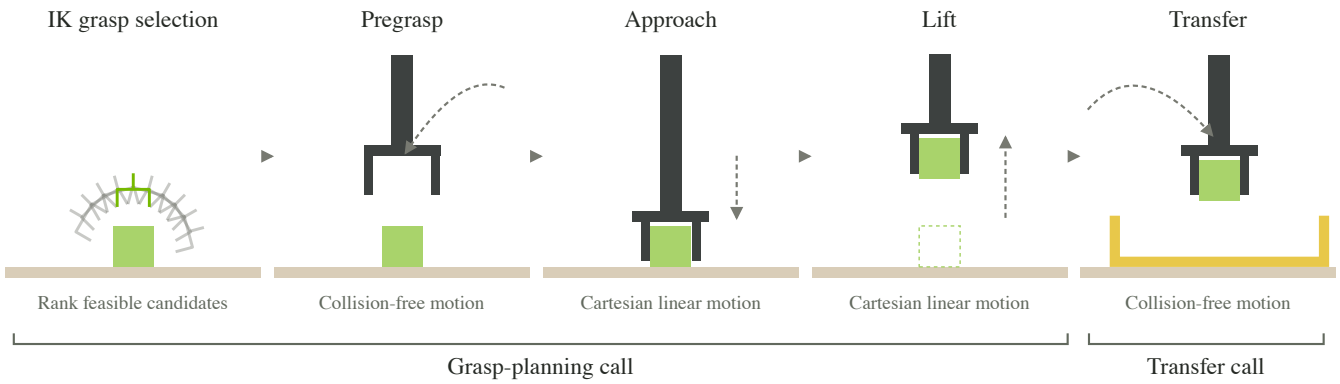



Fig. 9. Grasp-planning integration. IK selects a grasp, followed by pregrasp, approach, and lift motions in one planning call. Approach and lift follow Cartesian linear motions. A separate call plans transfer to the placement pose.

### *E. Differentiable Inverse Dynamics and Payloads*

GRiD exploits robot-specific code generation to optimize dynamics and its derivatives [4]. Our implementation instead keeps the chain schedule and inertial parameters as data, while specializing kernel dimensions and execution options at compile time. Efficiency comes from compact spatial operations, chain-local execution, and direct loss-gradient computation; payload updates do not require regenerating or recompiling the kernels.

At setup, we merge fixed bodies and prepare the chain schedule on the CPU. Independent chains execute concurrently, with each GPU worker traversing a serial chain locally. This avoids synchronization between successive joints within a chain; workers synchronize only at inter-chain dependencies. Spatial transforms use rotations and translations directly rather than dense $6 \times 6$ transform matrices. The recursive Newton–Euler algorithm computes link velocities, accelerations, and forces as

$$\begin{aligned} v_i &= X_i v_{\pi(i)} + S_i \dot{q}_i, \\ a_i &= X_i a_{\pi(i)} + S_i \ddot{q}_i + v_i \times S_i \dot{q}_i, \\ f_i &= I_i a_i + v_i \times^* (I_i v_i) - f_i^{\text{ext}}, \end{aligned} \tag{7}$$

where $X_i$ is a spatial transform, $S_i$ the joint motion subspace, and $I_i$ the spatial inertia. Forces are then accumulated from leaves toward the root, and projection onto each joint's motion subspace gives $\tau_i = S_i^{\mathsf{T}} f_i$. Payload changes update mass, center of mass, and inertia at runtime; the schedule remains valid while the reduced topology is unchanged.

Our analytical backward pass maps torque-loss gradients directly to $(q, \dot{q}, \ddot{q})$ and then to spline controls through Eq. 2. This avoids forming and contracting the full configuration and velocity derivative matrices returned by GRiD [4], reducing both derivative computation and output storage. Both passes combine sibling contributions with warp reductions and reuse shared-memory storage after its values are consumed or saved. The backward pass also reuses forward intermediates to avoid recomputation. Figure 8 summarizes the computation and chain execution. These savings preserve runtime updates to the robot's inertial parameters.

### *F. Grasp-Planning Integration*

An object can be grasped from many positions and orientations, so grasp planning begins by choosing which pose to plan for. Given a set of candidate grasp poses, we use IK to find feasible robot configurations and rank them by pose error and distance from the current configuration. We found this sufficient for grasp selection in our evaluated task, allowing us to replace trajectory optimization over the grasp-pose goal set with a less expensive IK solve. The selected pose defines the target for subsequent motion planning; IK feasibility alone does not establish that the robot can reach it along a collision-free path.

We then plan three connected motions (Fig. 9): a move to a pregrasp pose offset from the selected grasp, an approach along the grasp axis, and a lift of the object. Spline optimization generates the pregrasp motion, with RRT initialization when needed to navigate obstacles, while constrained spline optimization produces Cartesian linear motions for approach and lift. Each motion starts from the preceding motion's terminal state, with collision settings adjusted to permit the intended contact. A separate planning call generates a collision-free transfer of the grasped object to the placement pose.

## IV. Experiments

We evaluate route initialization, torque-limited trajectory optimization, robot computations, and pick-and-place planning. Motion benchmarks use Franka Panda (7-DoF), computation benchmarks also include G1 (29-DoF), and manipulation tasks use YAM. Offline planning runs on RTX 5090, with additional computation timings on Jetson; real-robot deployment uses RTX 6000 Ada.

### *A. Coarsely Checked Route Initialization*

We evaluate routes used to initialize trajectory optimization, with coarse collision checks at up to 0.03 rad per-joint spacing. On 2,575 queries from MotionBenchMaker [21] and MPINets [22], our RRT implementation achieves 98.68% initialization success, compared with 91.92% for cuRoboV1 PRM [2] and 88.27% for cuRoboV2 PRM (Fig. 10). Mean planning time is 0.305 ms, versus 13.09 ms and 8.49 ms for

the two PRM baselines, corresponding to $43.0\times$ and $27.9\times$ speedups. Our RRT is also $1.48\times$ faster than pRRTC [6], which achieves 99.07% success at 0.450 ms. VAMP [1] achieves 99.22% success at 0.0573 ms; although it initializes routes faster, VAMP + TOPPRA [7] has lower motion-planning success with payloads than our pipeline (Fig. 12).

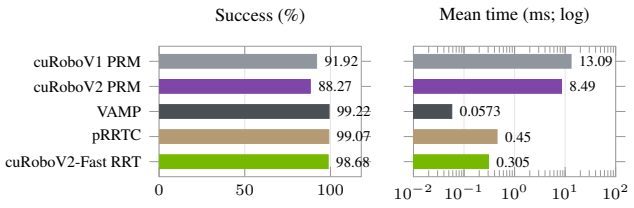



Fig. 10. Coarsely checked route initialization on MotionBenchMaker and MPINets (2,575 queries). Our RRT implementation is $1.48\times$ faster than pRRTC. VAMP plans routes faster, but VAMP + TOPPRA achieves lower motion-planning success with payloads than cuRoboV2-Fast (Fig. 12).

Timing and path length use 2,101 shared-success queries. Planning time includes native shortcutting and excludes IK. Mean path length is 4.93 rad for ours, compared with 5.28 rad for cuRoboV1 PRM, 3.70 rad for cuRoboV2 PRM, 5.00 rad for VAMP, and 5.32 rad for pRRTC. Routes that fail the coarse collision checks are rejected. Accepted routes serve as optimization seeds and require further refinement and validation.

### *B. Trajectory Optimization Convergence*

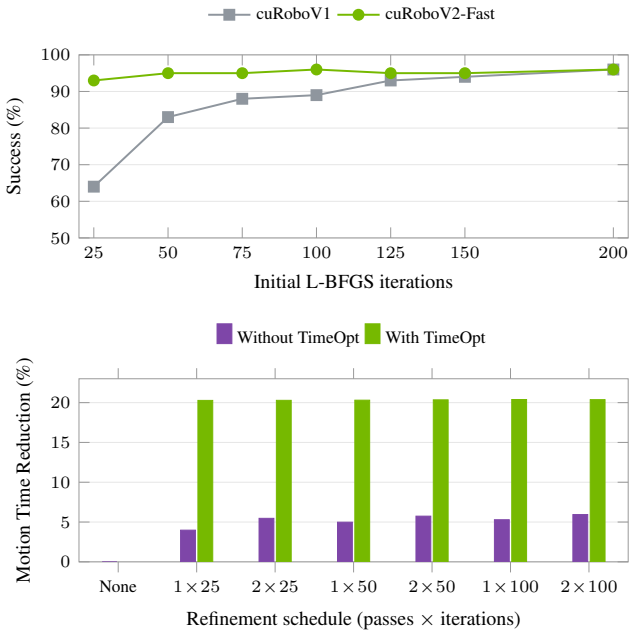



Fig. 11. Trajectory optimization convergence on 100 MotionBenchMaker `cage_panda` queries. Top: initial-only success. Bottom: cuRoboV2-Fast mean motion-time reduction on 93 shared valid queries after 25 initial iterations. Our B-spline formulation achieves higher success at small initial budgets. TimeOpt reduces mean motion time by 16% compared with finetuning using fixed timestep rescaling.

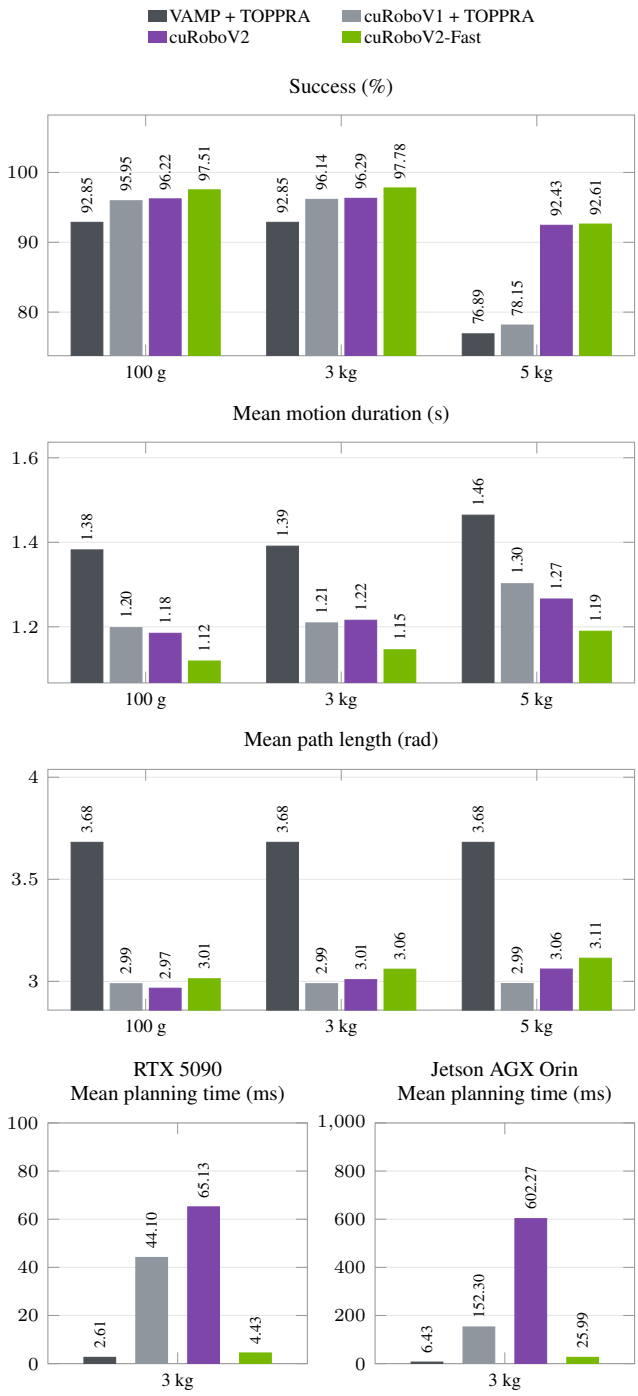



Fig. 12. Motion Planning with different payloads. cuRoboV2-Fast achieves similar success to cuRoboV2 while planning faster and producing shorter motions. Both V2 methods retain higher success than the TOPPRA baselines at the heaviest payload. Quality panels use RTX 5090 results; Fast success averages two repeats of 2,590 queries per payload. Motion duration and path length use 1,599 queries valid for all methods and payloads, including both Fast repeats. Separate planning-time panels show the 3 kg payload on RTX 5090 and Jetson AGX Orin, using different timing scales.

We isolate the effects of initial optimization, timestep refinement, and TimeOpt on 100 MotionBenchMaker `cage_panda` queries. cuRoboV1 [2] and cuRoboV2-Fast use the same four saved IK seeds per query and a single attempt, with dynamics and particle optimization disabled. Figure 11 shows that our B-spline formulation achieves higher success at small iteration budgets: cuRoboV2-Fast reaches 93% validated success with 25 initial L-BFGS iterations, compared with 64% for cuRoboV1. cuRoboV1 requires $5\times$ as many initial iterations to match this success rate. Increasing our initial budget to 200 raises success by only three percentage points.

We then fix our initial budget at 25 iterations and compare one or two refinement passes of 25, 50, or 100 iterations, using a timestep-shrink factor of 0.8. All settings retain 93% success. On the 93 queries valid across all settings, two 25-iteration passes reduce mean motion duration by 5.5% relative to initial optimization alone. Adding TimeOpt increases the total reduction to 20.3%, corresponding to a further 15.7% reduction relative to the refined trajectory. Longer refinement schedules improve motion duration after TimeOpt by less than 0.2%.

For the reported cuRoboV2-Fast timings and real-robot deployment, we use 50 initial iterations followed by one 25-iteration refinement pass ($50 + 1 \times 25$), giving a total budget of 75 iterations.

### C. Motion Planning with Payloads

We study the full pipeline for generating a collision-free, torque-limited motion for execution, given a start state and a target pose. For methods that do not consider torque limits during planning, we postprocess their trajectories using TOPPRA [7], which changes timing along a fixed path. We compare four pipelines: cuRoboV2-Fast, cuRoboV2 [5], cuRoboV1 [2] + TOPPRA, and VAMP [1] + TOPPRA.

As shown in Figure 12, cuRoboV2-Fast and cuRoboV2 achieve similar success rates across payloads, exceeding those of cuRoboV1 + TOPPRA and VAMP + TOPPRA. At the heaviest payload, success decreases for all methods, with a larger drop for the two TOPPRA baselines; cuRoboV2-Fast retains a 14.5–15.7 percentage-point advantage over them.

On queries valid for all methods at all payloads, cuRoboV2-Fast produces the shortest mean motion duration, reducing duration by 5.8% relative to cuRoboV2 on average across payloads. Path length alone can be misleading when comparing motion efficiency. cuRoboV2-Fast produces mean joint-space paths that are 0.8–4.2% longer than those of cuRoboV1 + TOPPRA, yet its motions are 6.8% shorter in duration on average across payloads.

For the 3 kg planning-time comparison, cuRoboV2-Fast takes 4.43 ms on RTX 5090, $14.7\times$ faster than cuRoboV2 and $10.0\times$ faster than cuRoboV1 + TOPPRA. VAMP + TOPPRA plans $1.7\times$ faster than cuRoboV2-Fast, but achieves lower success and excludes IK from its timing. On Jetson AGX Orin, cuRoboV2-Fast takes 25.99 ms, $23.2\times$ faster than cuRoboV2 and $5.9\times$ faster than cuRoboV1 + TOPPRA (Fig. 12). Its relative speedup over cuRoboV2 is greater than on RTX 5090. **cuRoboV2-Fast on Jetson AGX Orin plans faster than cuRoboV1 + TOPPRA on RTX 5090.**

### D. Robot Computation Performance

Having established the planning-time speedups of the full motion planning pipeline, we now examine computation times for individual modules. We benchmark median latency on RTX 5090 for Franka Panda and G1 at batches 1 and 1,024, comparing our implementation with cuRoboV1, cuRoboV2, Frax, Newton, PyRoki, PyT-Kin (pytorch_kinematics), Pinocchio, and GRiD where applicable. Module speedups average latency ratios over the two batch sizes; comparisons covering both robots also average over Franka and G1.

Kinematics scales to large batches with little additional latency: increasing the batch from 1 to 1,024 adds less than 10% on either robot. Our forward and backward computations are $1.1\times$ faster than cuRoboV2 on Franka and $1.3\times$ faster on G1 (Fig. 13). Jacobian gains are larger on the higher-DoF G1 robot (Fig. 13). Our implementation is $1.2\times$ faster than cuRoboV2 on Franka and $1.6\times$ faster on G1. On G1, the speedup over cuRoboV1 is approximately $38\times$. Our IK solver uses explicit Jacobians during Levenberg–Marquardt initialization. On 100 shared Franka Panda targets, all four IK methods achieve 100% success under common pose, joint-limit, and collision checks. cuRoboV2-Fast is $1.9\times$ faster than cuRoboV2 for collision-aware IK, preserving success while reducing batch latency (Fig. 13).

Inverse dynamics takes approximately half as long as cuRoboV2 while supporting runtime changes to inertial parameters (Fig. 13). Increasing the batch from 1 to 1,024 adds less than 20% latency on either robot. Relative to GRiD [4], the reported latency ratios average $2.4\times$ on Franka and $6.4\times$ on G1 across the two batch sizes. At batch 1, our GPU implementation is also faster than CPU Pinocchio on both robots, with an average speedup of $1.5\times$.

Collision computations improve over cuRoboV2 at both batch sizes (Fig. 13), with mean speedups of $1.22\times$ for self-collision and $1.15\times$ for swept world-collision. G1 self-collision shows a much larger gain over cuRoboV1, with a speedup of approximately $52\times$.

### E. Trajectory Optimization Computation

cuRoboV2-Fast reduces trajectory optimization time through faster iterations and a smaller iteration budget (Fig. 14). With inverse dynamics enabled, each iteration is $2.8\times$ faster than cuRoboV2 on RTX 5090 and $2.6\times$ faster on Jetson AGX Orin.

Part of this gain comes from avoiding repeated work during line search. By reusing the cost at the current solution, cuRoboV2-Fast evaluates 972 robot states per iteration instead of cuRoboV2's 1,296. The RTX 5090 module profiles show combined forward and backward inverse dynamics taking 19.38 $\mu$s versus 31.58 $\mu$s for cuRoboV2 ($1.6\times$ faster). Other costs and reductions are $4.2\times$ faster. Combined scene and self-collision kernel time is 14.01 $\mu$s for cuRoboV2-Fast versus 16.22 $\mu$s for cuRoboV2. Compared with cuRoboV1,

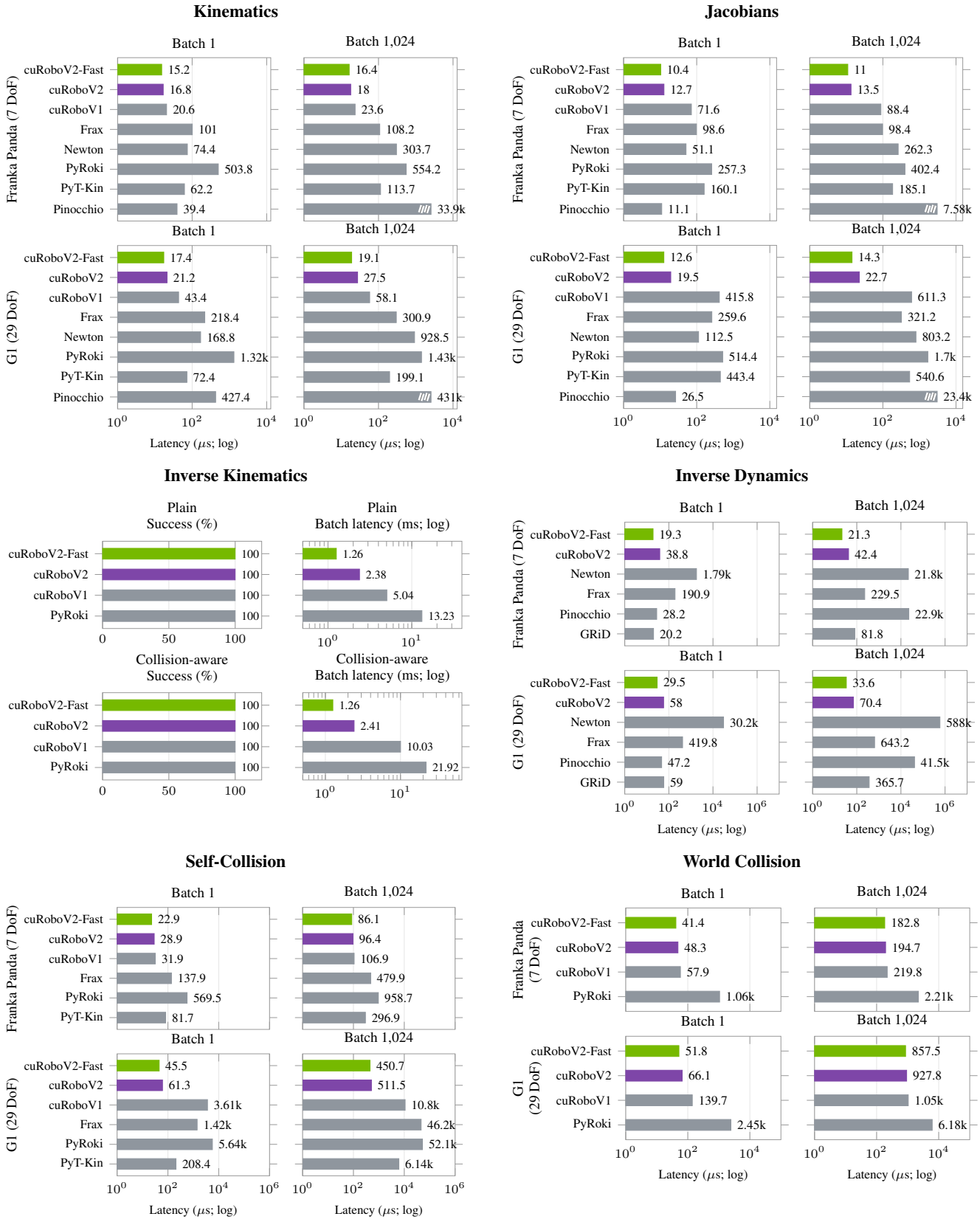



Fig. 13. Robot computation performance on RTX 5090. cuRoboV2-Fast accelerates robot computations and collision-aware IK. Latencies are medians; IK uses 100 shared Franka Panda targets. Other modules compare batches 1 and 1,024 on Franka Panda and G1. Broken-bar labels show full values; k denotes $10^3$.

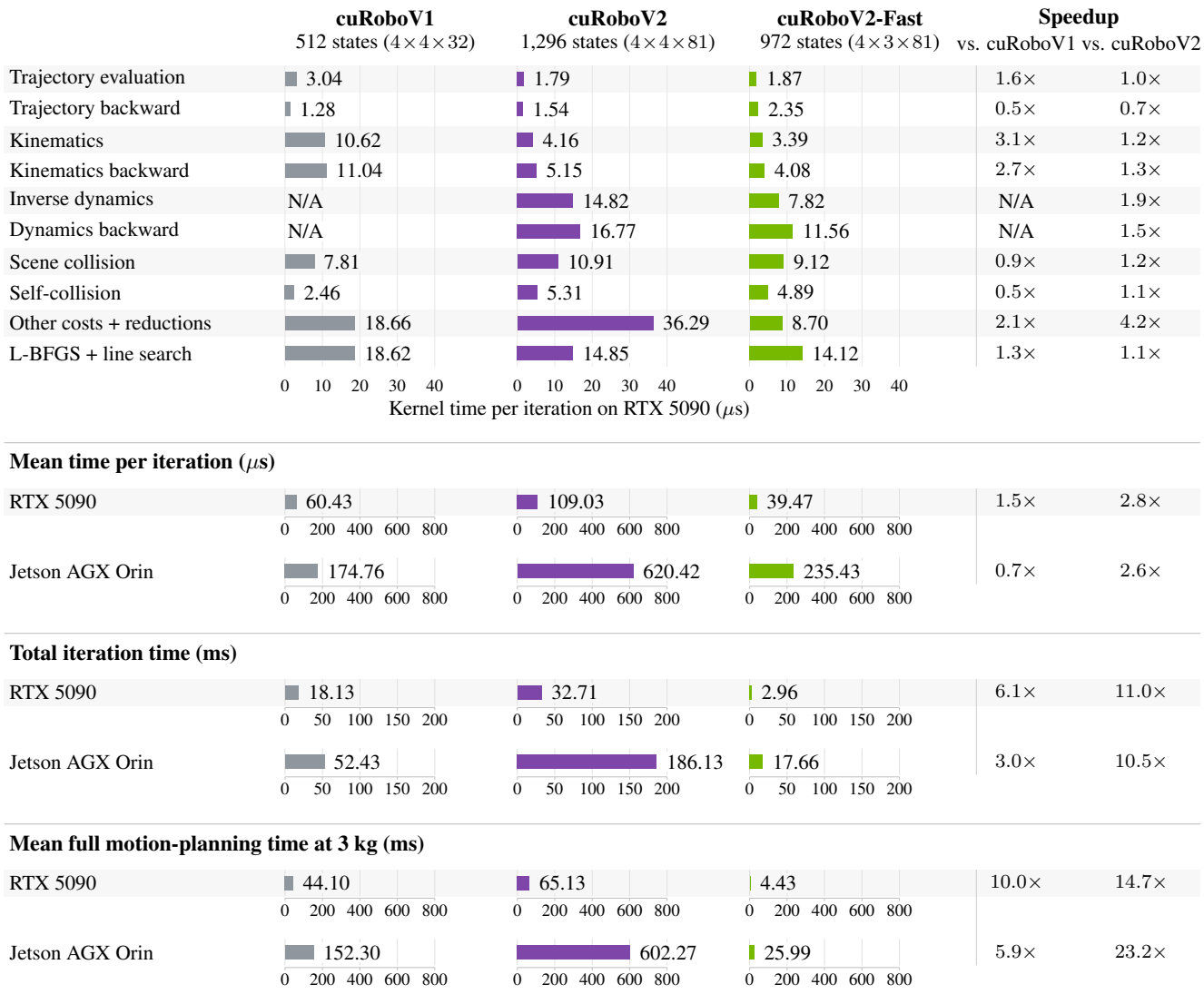



Fig. 14. Trajectory-optimization and full motion-planning times. Faster iterations and a smaller iteration budget compound to reduce optimization time. cuRoboV2-Fast achieves 11.0× and 10.5× shorter optimization-loop times than cuRoboV2 on RTX 5090 and Jetson AGX Orin, respectively. Iteration totals use 75 iterations for cuRoboV2-Fast and 300 for both baselines. Iteration timings on both devices use 100 repeated `bookshelf_small_panda` solves at 100 g.

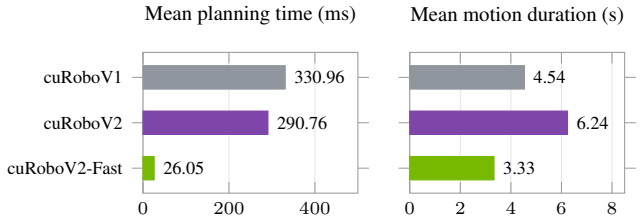



Fig. 15. Mean per-cube planning time and planned motion duration for the five-cube simulated YAM tray arrangement. cuRoboV2-Fast plans over 11× faster and produces shorter motions than both baselines.

combined forward and backward kinematics are 2.9× faster, while L-BFGS and line search are 1.3× faster.

The smaller iteration budget compounds these savings. Following the convergence study in Section IV-B, we use 75

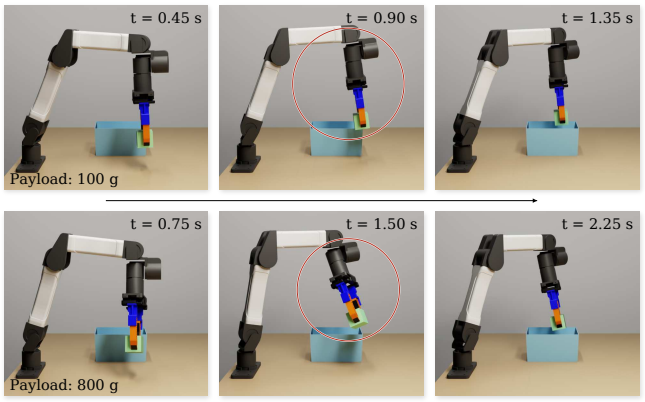



Fig. 16. Simulated YAM transfers with 100 g (top) and 800 g (bottom) payloads. The heavier payload shifts motion from joint 2 to joint 5 to respect joint 2's torque limit.

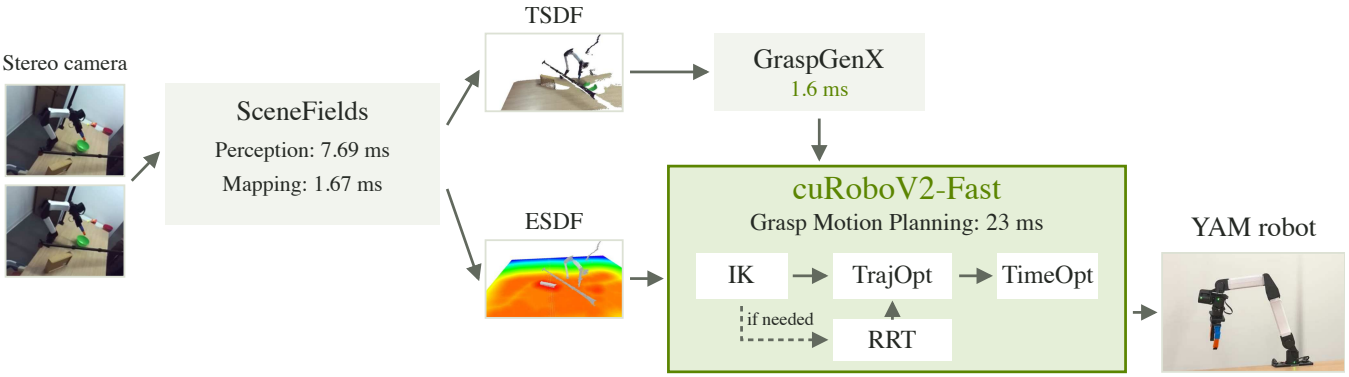



Fig. 17. Real-robot manipulation on one RTX 6000 Ada GPU. SceneFields provides depth, detections, and robot masks; its maps supply object surfaces to GraspGenX and collision distances to cuRoboV2-Fast. Our planner selects a grasp and plans collision-free, torque-limited pregrasp, approach, and lift motions in 23 ms. RRT initializes trajectory optimization when needed; transfer motions are planned separately.

iterations ($50 + 1 \times 25$) for cuRoboV2-Fast, compared with 300 for both baselines. Together with the faster iterations, this reduces total time spent in the optimization loop by a factor of 11.0 on RTX 5090 and 10.5 on Jetson AGX Orin relative to cuRoboV2. The corresponding reductions relative to cuRoboV1 are factors of 6.1 and 3.0, even though cuRoboV1 omits inverse dynamics. These timings cover the optimization iterations; initialization and other full-solve overhead are excluded.

### *F. Simulated Planning Experiments*

We compare cuRoboV1 [2], cuRoboV2 [5], and cuRoboV2-Fast on a simulated YAM pick-and-place task that arranges five cubes from one cluttered layout into a tray on RTX 5090. Planning includes grasp selection, pregrasp, approach, lift, and transfer. All three methods complete all five cubes, and the saved lift and transfer trajectories pass independent held-object collision and payload torque audits.

Figure 15 shows that cuRoboV2-Fast reduces both planning time and planned motion duration. Mean planning time per cube is 26.05 ms, giving speedups of 12.7× over cuRoboV1 and 11.2× over cuRoboV2. Mean planned motion duration is 3.33 s, which is 26.5% shorter than cuRoboV1 and 46.6% shorter than cuRoboV2. Together, these gains reduce mean cycle time, measured as planning plus planned motion, to 3.36 s per cube: 31.0% shorter than cuRoboV1 and 48.6% shorter than cuRoboV2.

In a separate simulated YAM transfer between fixed pickup and placement locations, we vary the payload from 100 g to 800 g (Fig. 16). The heavier payload changes the optimized motion: movement shifts from joint 2 to joint 5 to respect joint 2's torque limit, illustrating how payload-aware planning adapts the path to the load.

### *G. Real-Robot Deployment*

We deploy cuRoboV2-Fast on the I2RT YAM for tabletop clearing and bin picking (Fig. 19). One RTX 6000 Ada runs SceneFields [23], GraspGenX [24], and our planner (Fig. 17). SceneFields provides depth, detections, and robot masks. It builds a TSDF with 5 mm voxels for object surfaces used by GraspGenX and an ESDF with 20 mm voxels for collision checking by our planner.

Measured joint feedback shows that cuRoboV2-Fast uses more of the available velocity range during arm execution (Fig. 18). Relative to cuRoboV2, the fraction of motion time at 90–100% velocity utilization nearly doubles, while the fraction below 50% utilization decreases by 21.7%. Torque utilization remains below 90% for 99.6% of motion time with cuRoboV2-Fast, and all recorded torques remain within limits for both methods. Executed velocities can exceed limits despite velocity-feasible plans, highlighting the importance of considering torque constraints alongside velocity constraints when planning motions for execution.

Spending a larger fraction of motion time at higher velocities translates into shorter arm execution times. Across approximately 50 completed object-transfer cycles for cuRoboV2-Fast and 15 for cuRoboV2, mean arm execution time decreases by 44.8%. This reduction leads to a 34.6% decrease in mean cycle time (Fig. 18).

We break down the time in a cycle across motion planning, arm execution, gripper execution, and other processing and waits (Fig. 19). For cuRoboV2-Fast, these average 0.06, 3.75, 1.53, and 0.60 s, respectively. Arm execution accounts for 63.1% of cycle time, while planning accounts for only 1.0%.

## V. Discussion

The planning-time improvements come from reducing both the number of optimization iterations and the cost of each iteration. The convergence experiment shows that our B-spline formulation reaches comparable success with fewer initial iterations in the evaluated configuration, while accelerated robot computations reduce the cost of refinement with inverse dynamics. Together with fast route initialization, these improvements yield full-pipeline speedups while maintaining similar success to cuRoboV2 across payloads. The gains also extend to Jetson AGX Orin, where cuRoboV2-Fast plans faster than cuRoboV1 + TOPPRA on RTX 5090, supporting deployment on embedded hardware.

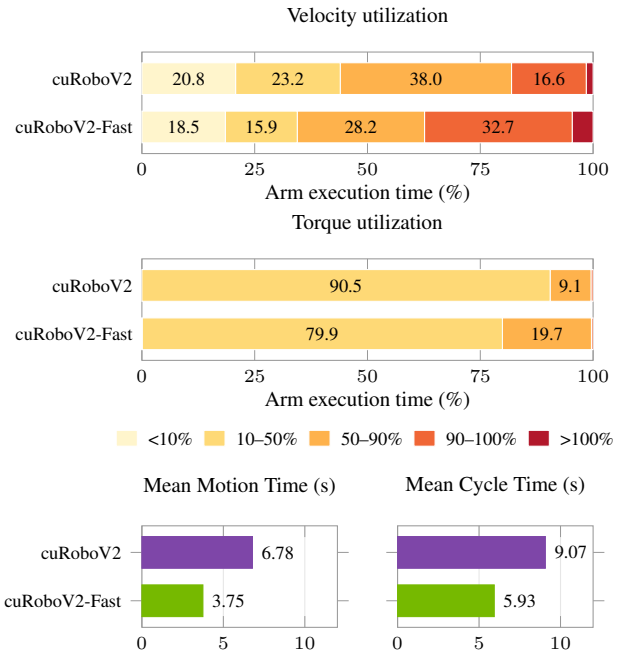



Fig. 18. Arm utilization, motion time, and cycle time for real-robot object transfers. Utilization uses the most utilized joint at each instant. cuRoboV2-Fast nearly doubles time at 90–100% velocity utilization and reduces motion and cycle times by 44.8% and 34.6%, respectively.

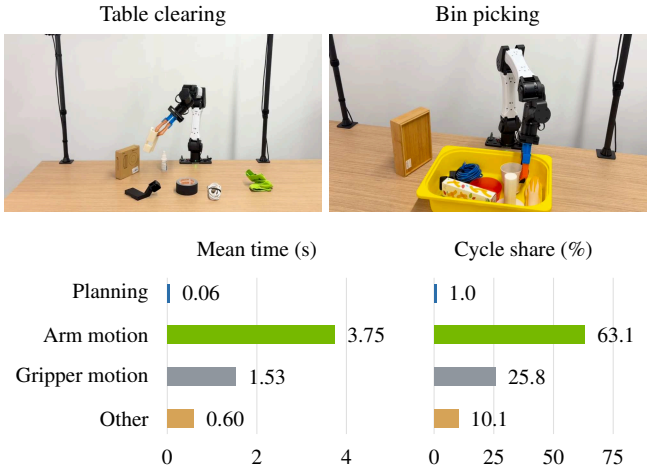



Fig. 19. Real YAM table clearing and bin picking with cuRoboV2-Fast. Mean cycle breakdown over 53 completed sequences; planning accounts for only 1.0%, including failed planning calls and retries within each cycle. Other includes residual processing and waits.

Efficient execution requires optimizing both the path and its timing. The payload benchmarks show that shorter joint-space paths do not necessarily produce shorter motions under actuator constraints. Payload-aware trajectory optimization can adapt the path to the load, while TimeOpt redistributes traversal time along the resulting path. Its 15.7% reduction in mean motion duration beyond uniform timestep refinement shows that timing remains an opportunity for improvement after geometry optimization. Local spline support yields banded Newton systems and parallel constraint evaluation, making this additional optimization inexpensive on the GPU. These results support TimeOpt as a practical way to improve execution time without requiring globally optimal timing.

The real-robot experiments connect these improvements to execution: cuRoboV2-Fast spends a larger fraction of arm-motion time near the velocity limits and reduces mean arm execution time by 44.8% relative to cuRoboV2. Shorter arm motions also reduce completed cycle time, although gripper execution and other processing remain substantial contributors. Planning accounts for only 1.0% of the measured cycle time, so further planning-latency reductions alone would have limited impact on these cycles. Improving the motions produced by the planner therefore remains central to faster manipulation, alongside integration with perception, grasp generation, and execution.

### A. Limitations

Our approach addresses motion planning holistically, enabling high success in cluttered scenes while reducing both computation time and motion duration. Although it does not guarantee global optimality, it achieves a mean pick-and-transfer cycle time of 5.9 s on a real robot. Planning accounts for only 1.0% of cycle time in this deployment and is no longer a bottleneck. The main remaining bottleneck lies in contact-rich manipulation, particularly achieving reliable grasps and successful object interactions.

## References


[1] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 8749–8756.

[2] 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.

[3] J. Pan, L. Zhang, and D. Manocha, "Collision-free and curvature-continuous path smoothing in cluttered environments," in *Robotics: Science and Systems (RSS)*, 2011. [Online]. Available: https://www.roboticsproceedings.org/rss07/p32.pdf

[4] 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.

[5] B. Sundaralingam, A. Murali, and S. Birchfield, "cuRoboV2: Dynamics-aware motion generation with depth-fused distance fields for high-DoF robots," 2026. [Online]. Available: https://arxiv.org/abs/2603.05493

[6] C. H. Huang, P. Jadhav, B. Plancher, and Z. Kingston, "pRRTC: GPU-parallel RRT-connect for fast, consistent, and low-cost motion planning," in *2026 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2026. [Online]. Available: https://arxiv.org/abs/2503.06757

[7] H. Pham and Q.-C. Pham, "A new approach to time-optimal path parameterization based on reachability analysis," *IEEE Transactions on Robotics*, vol. 34, no. 3, pp. 645–659, 2018.

[8] 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.

[9] 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.

[10] 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.

[11] J. Schulman, Y. Duan, J. Ho, A. X. 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, pp. 1251 – 1270, 2014.

[12] D. Chiaravalli, F. Califano, L. Biagiotti, D. De Gregorio, and C. Melchiorri, "Physical-consistent behavior embodied in B-spline curves for robot path planning," *IFAC-PapersOnLine*, vol. 51, no. 22, pp. 306–311, 2018.

[13] S. Lengagne, J. Vaillant, E. Yoshida, and A. Kheddar, "Generation of whole-body optimal dynamic multi-contact motions," *The International Journal of Robotics Research*, vol. 32, no. 9–10, pp. 1104–1119, 2013. [Online]. Available: https://staff.aist.go.jp/e.yoshida/papers/ijrr-lengagne-2013.pdf

[14] J. E. Bobrow, S. Dubowsky, and J. S. Gibson, "Time-optimal control of robotic manipulators along specified paths," *The International Journal of Robotics Research*, vol. 4, no. 3, pp. 3–17, 1985.

[15] D. Verscheure, B. Demeulenaere, J. Swevers, J. De Schutter, and M. Diehl, "Time-optimal path tracking for robots: A convex optimization approach," *IEEE Transactions on Automatic Control*, vol. 54, no. 10, pp. 2318–2327, 2009.

[16] H. Pham and Q.-C. Pham, "On the structure of the time-optimal path parameterization problem with third-order constraints," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 679–686.

[17] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "The Pinocchio C++ library – a fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, 2019, pp. 614–619.

[18] C. M. Kim, B. Yi, H. Choi, Y. Ma, K. Goldberg, and A. Kanazawa, "PyRoki: A modular toolkit for robot kinematic optimization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2025. [Online]. Available: https://arxiv.org/abs/2505.03728

[19] 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.

[20] "TOPPRA: Joint torque constraint implementation," https://hungpham2511.github.io/toppra/_modules/toppra/constraint/joint_torque.html, accessed October 8, 2026.

[21] 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, 2022.

[22] 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*, ser. Proceedings of Machine Learning Research, vol. 205. PMLR, 2023, pp. 967–977.

[23] B. Sundaralingam, A. Murali, and S. Birchfield, "SceneFields: Fast scene representation for robotic manipulation," NVIDIA, Technical Report, 2026.

[24] B. Han, Y.-W. Chao, E. Coumans, C. Eppner, J. Deng, S. Birchfield, and A. Murali, "GraspGen-X: Cross-embodiment 6-DOF diffusion-based grasping," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026, pp. 20 878–20 889.