Branch: Specializing Robotics Solvers in Warp

Anonymous submission

***Abstract*— The fastest GPU kernel for a robotics solve depends on the workload: the robot, the batch size, the horizon, and even the GPU. Peak throughput therefore means deriving and hand-writing a kernel for each case. In practice, libraries commit to one, or at best a handful of, elimination orders and thread mappings, and leave performance on the table. We present BRANCH, a library and compiler built within NVIDIA Warp for the structured linear solves at the heart of robotics optimizers and simulators. An application states its problem, its elimination plan, and its GPU schedule, and BRANCH generates the kernel for that configuration. BRANCH turns specialization into compilation: the problem is described once, so trying alternatives and keeping the fastest for the workload at hand becomes practical. We report two results. First, specialization matters. Over seven robots, batch sizes from 1 to 262 144, horizons to 2048 frames, and four GPUs, the best plan and schedule vary along every workload axis. The single best fixed BRANCH configuration is slower than the per-workload specialized configuration by up to 1.5× in single-frame inverse kinematics (IK), 1.3× in trajectory optimization, and 2.7× in contact response. Second, generated code holds up against hand-written kernels and existing libraries. This paper demonstrates BRANCH on inverse kinematics, from single poses to trajectories, and on contact response inside a Newton physics engine projected-Gauss–Seidel contact solver: replacing the hand-written contact kernels with generated ones runs the contact solver at 1.02–1.03× its native throughput. At matched accuracy BRANCH single-frame IK is faster than PyRoki and pytorch_kinematics at every tested batch size, and its trajectory solves are faster than PyRoki's at every tested horizon. Project site: branchsolver.io.**

I. INTRODUCTION

The same robotics software stack serves arms, hands, quadrupeds, and humanoids; batches from one instance to hundreds of thousands; single poses and long horizons; and contact-heavy scenes. The structured linear solves at the heart of its optimizers and simulators are symmetric positive-definite systems whose block structure follows the kinematic tree, the horizon, and the batch of instances. For such a solve, both the fastest elimination order and the fastest thread mapping depend on the workload. With thousands of instances, one thread per robot can eliminate joint by joint along the kinematic tree with little arithmetic and storage. With one instance, forming the dense joint-space system and factoring it with many cooperating threads does more arithmetic but finishes sooner. In a hand-written solver the equations, the elimination order, the temporary arrays, and the thread mapping appear together in one kernel, so trying an alternative means deriving, implementing, and validating another solver before learning whether it helps. Peak performance therefore means hand-writing a factorization and a thread mapping for each case and maintaining several implementations.

No single choice dominates. Over seven robots, batch sizes from 1 to 262 144, horizons to 2048 frames, and four GPUs (Sec. V), the fastest configuration changes along every axis of the configuration maps, one table of workloads per application and GPU. In single-frame IK the tree elimination wins all 49 cells and the thread mapping changes with robot and batch. In trajectory optimization, the optimal elimination order changes with batch and horizon: cyclic reduction of the frames wins every cell of the two tested arms and, on the two tested humanoids, the cells with $B \leq 64$ from $T{=}32$, and chronological elimination wins at $B{=}256$ on both humanoids, where the batch alone fills the device. In contact response the dense plan wins the robot-dominated worlds, and the tree plan wins the dense cube piles at large batch sizes. Even the best single configuration for a whole map is slower than the per-cell winner by up to 1.5× in single-frame IK, 1.3× in trajectory optimization, and 2.7× in contact.

Existing solvers commit to one of these choices (Table I). cuRobo fixes traversal, threads per problem, and shared-memory layout at compile time [1], GRiD generates one wave per tree level [2], and the hand-written single-frame IK, trajectory IK, and contact-response kernels this paper replaces each fix one elimination order under one mapping [3]. The fastest kernel for a new workload is yet another kernel.

BRANCH is a library and compiler for such solves, built within NVIDIA Warp [4]. The application states its problem, its elimination plan, and its GPU schedule (Fig. 1), and BRANCH generates the kernel for that configuration through shared machinery. The *problem* states the unknowns, the linear relations between them, and the residual terms. The *elimination plan* states which unknowns are eliminated together and in what order. The *GPU schedule* states how the plan's operations are grouped onto threads, cooperating blocks, or tiles, and where storage is placed. BRANCH is general over the structured symmetric positive-definite solves of robotics, for any problem of blocks with linear relations and typed residuals. BRANCH turns specialization into compilation: the problem is described once, and the kernel for any elimination plan and GPU schedule is generated from it. A developer therefore enumerates plans and schedules, keeps the fastest for the workload, and repeats the search when the robot, batch, horizon, or GPU changes.

This paper demonstrates BRANCH on inverse kinematics (IK), from single poses to trajectories, and on contact response inside a Newton physics engine [3] projected-Gauss–Seidel contact solver (Sec. IV): an IK solver moves a robot's joints so that its hands or tools reach target poses, a trajectory optimizer chooses a sequence of such configurations while penalizing abrupt motion, and a contact solver computes

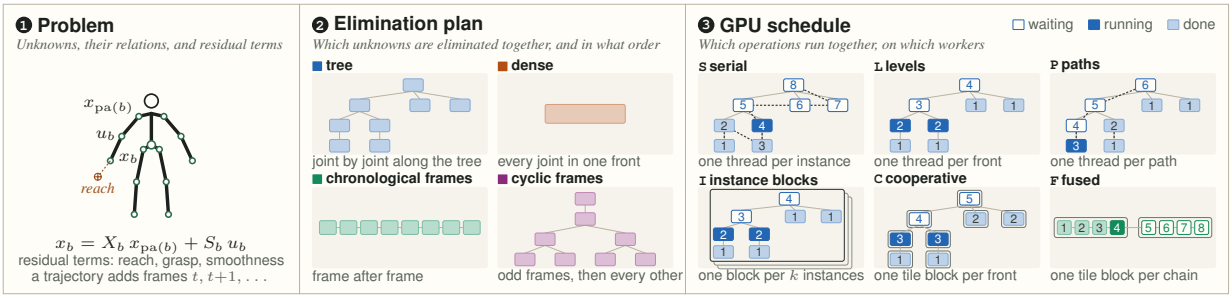

Fig. 1. **BRANCH takes three separate descriptions—the problem, the elimination plan, and the GPU schedule—and generates the kernel for that configuration.** Because BRANCH can generate a kernel for any plan and schedule, enumerating configurations and using the fastest one for the workload at hand becomes practical. Each box is a front: a group of unknowns eliminated together with the live unknowns they couple to; its result enters its parent. Numbers give each schedule's processing order (solid: running; tinted: done). S–C show the tree plan, F the chronological-frames plan, whose frames form the chains it fuses.

how an impulse at a foot or finger changes the velocity of the whole robot. Replacing the hand-written contact kernels with generated ones runs the contact solver at 1.02–1.03× its native throughput. At matched accuracy BRANCH single-frame IK is faster than PyRoki and pytorch_kinematics at every tested batch size, and its trajectory solves are faster than PyRoki's at every tested horizon. Our contributions are:

1) Configuration maps over seven robots, batch sizes to 262 144, horizons to 2048, and four GPUs, showing that no single plan or schedule dominates, what the best fixed configuration costs, and what trying an alternative costs (Sec. V).
2) One representation of problem, elimination plan, and GPU schedule, with a two-rule planner (SUBSTITUTE, ELIMINATE) whose instances are the plans of Table I (Sec. III-B).
3) A library of generated kernels: six schedules plus right-hand-side tiling as lowerings of one plan, with plan and schedule reports that expose fill, depth, and storage before compiling (Sec. III-C).
4) Integration into Newton's inverse kinematics, from single poses to trajectories, and into a Newton physics engine projected-Gauss–Seidel contact solver, evaluated at matched accuracy against PyRoki and pytorch_kinematics and, end to end, against the hand-written contact kernels (Secs. IV and V).

II. RELATED WORK

Each solver BRANCH relates to commits to one elimination plan and one GPU mapping (Table I). The articulated-body and composite-rigid-body algorithms are the tree and dense recursions [5], [6]. Riccati recursion, cyclic reduction [7], and SPIKE [8] are three orders for the block-tridiagonal system of a horizon. The GPU libraries fix their mapping when they are built: cuRobo its traversal, threads per problem, and shared-memory layout [1]; GRiD one wave per tree level [2]; MuJoCo Warp a scalar, tiled, or sparse mass-matrix factorization per model [10]; PhysX and Newton one kernel per articulation [3], [11], [18]; MPCGPU a conjugate gradient on the Schur system [13]; SOCU a nested-dissection Cholesky [12]. PyRoki solves user residuals over a least-squares solver in JAX [14], [15], LoIK applies the articulated recursion to differential IK on the CPU [9], and GMR retargets motion on the CPU one frame at a time [19]. GTSAM builds an elimination tree from a variable ordering [16], Ceres exposes Schur elimination and sparse Cholesky as interchangeable solvers [17], and sparse direct solvers separate a fill-reducing ordering from numeric factorization [20]. BRANCH shares this separation but plans on the *lifted* articulated space, so the tree recursion, joint-space condensation, and temporal orderings are one representation with one executor. Like Halide [21], Exo [22], and Sympiler [23], it separates what is computed from how it is scheduled: anything that changes eliminated sets, separators, or fill is a *plan* transformation, and a *schedule* only groups fronts onto workers and places storage.

TABLE I

EXISTING SOLVERS COMMIT TO ONE ELIMINATION PLAN AND ONE GPU SCHEDULE; BRANCH GENERATES THE KERNEL FOR ANY OF THEM FROM THE SAME PROBLEM DESCRIPTION. PLAN NAMES AND SCHEDULE LETTERS AS IN FIG. 1, OTHERWISE THE MAPPING IN WORDS; – NOT STATED.

Solver	Plan	Schedule
articulated-body algorithm [5]	tree	–
composite rigid body, Cholesky [6]	dense	–
Riccati recursion	chronological frames	–
cyclic reduction [7]	cyclic frames	–
SPIKE [8]	partitioned	–
LoIK [9]	tree	CPU
GRiD [2]	tree	L
cuRobo [1]	dense	C
MuJoCo Warp [10]	dense	S or C, per model
PhysX articulations [11]	tree	per articulation
Newton single-frame IK [3]	dense	C
contact response for Newton [3]	tree and dense	per articulation
SOCU [12]	cyclic frames	–
MPCGPU [13]	iterative (PCG)	–
PyRoki, jaxls [14], [15]	dense or CG	XLA
GTSAM [16]	any ordering	CPU
Ceres [17]	any ordering	CPU or CUDA
BRANCH	all four	S L P I C F

III. BRANCH: PROBLEM, PLAN, AND SCHEDULE

A *problem* states blocks of unknowns, the linear relations between them, and residual terms. At the current state it induces the local quadratic

$$\min_z \ \tfrac{1}{2} z^{\mathsf{T}} H z + g^{\mathsf{T}} z, \quad H = \textstyle\sum_k \tilde{J}_k^{\mathsf{T}} \tilde{J}_k + Q, \quad g = \textstyle\sum_k \tilde{J}_k^{\mathsf{T}} \tilde{r}_k, \tag{1}$$

where $\tilde{r}_k = w_k r_k(0)$ and $\tilde{J}_k = w_k\, \partial r_k / \partial z$ are the weighted residuals and Jacobians of the problem's terms

and Q collects quadratic terms (damping λI, or a by-reference matrix such as an inertia). `br.plan(prob, order)` is the planning stage: it builds the elimination plan, a front program (Sec. III-B), for the chosen order. `br.compile(plan, schedule)` lowers the plan under a GPU schedule (Sec. III-C) and returns a solver, whose `solve()` returns the Gauss–Newton (or LM) step $z^\star = -H^{-1}g$ and whose `solve(rhs=v, on=blocks)` adds $A^\mathsf{T}v$ to g, the response operator of Sec. IV.

A. Problem

An `Articulation` is a rooted forest of bodies b with parent $\mathrm{pa}(b)$ and joint width $d_b \in \{0,\dots,6\}$. The unknowns are the joint tangents $u_b \in \mathbb{R}^{d_b}$; body twists $x_b \in \mathbb{R}^6$ are dependent blocks tied to them by the exact relation

$$x_b = X_b\, x_{\mathrm{pa}(b)} + S_b\, u_b, \qquad x_{\mathrm{world}} = 0, \tag{2}$$

with $S_b \in \mathbb{R}^{6\times d_b}$ the motion subspace and $X_b \in \mathbb{R}^{6\times 6}$ an optional frame change. Free blocks $o \in \mathbb{R}^w$ are declared with `br.blocks`, and block sets carry an instance axis B and optionally a frame axis T. A residual term is a typed function of *sample* arguments (Warp structs holding a block's state and a zero-initialized perturbation in its coordinates: `Body`, `Dof`, `Tangent`) and *value* arguments (Fig. 2). Helpers such as `position` and `rotation_error` carry custom adjoints equal to the analytic Jacobian rows, so autodiff of any residual linear in the perturbation reproduces Newton's hand-written Jacobian rows. `prob.add` binds samples to block *slices* (`x[hands]`, `u.dofs[:, 1:]`), and quadratic terms (`br.quadratic`, `br.damping`) bind coefficient arrays by reference. One generated kernel per term evaluates r and its Jacobian rows (in-kernel reverse-mode `wp.grad`, or the user's `.jacobian`) into a *slot* holding $\tilde{J}^\mathsf{T}\tilde{J}$, $\tilde{J}^\mathsf{T}\tilde{r}$, and $\tilde{r}$.

B. Elimination plan

The planner is symbolic: it manipulates block identities and integer recipes. Its state is the set of *live* blocks and a set of *sources*. A source is a quadratic (H_s, g_s) stored in a slot in its original coordinates, with a *recipe* of entries. Each entry maps a range of slot coordinates onto a live block through $E_{\mathrm{entry}} = \prod \text{factors}$, each factor an X_b or S_b, so that the source contributes $E_s^\mathsf{T} H_s E_s$ and $E_s^\mathsf{T} g_s$. Two rules rewrite this state. **SUBSTITUTE**(x_b): for every source touching x_b, replace each entry on x_b by an entry on $x_{\mathrm{pa}(b)}$ with factors $\cdot X_b$ (unless the parent is the world) and one on u_b with factors $\cdot S_b$ (unless $d_b = 0$). Then remove x_b from the live set and record it for recovery. **ELIMINATE**(V): let F be the sources touching V and sep the other blocks they touch. Emit the front $f = (V, \mathrm{sep}, F)$, remove F, and if $\mathrm{sep} \neq \emptyset$ add a source on sep with an identity recipe, the message M_f. Every front runs the same three operations. With the assembled front partitioned by eliminated and separator coordinates,

$$\bar{H}_f = \begin{bmatrix} D & G \\ G^\mathsf{T} & C \end{bmatrix} = \sum_{s\in F} E_s^\mathsf{T} H_s E_s, \qquad \bar{g}_f = \begin{bmatrix} h \\ c \end{bmatrix}, \tag{3}$$

TABLE II

PLAN REPORTS EXPOSE FACTOR SIZE, DEPTH, AND STORAGE BEFORE ANY KERNEL IS COMPILED. FOR THE G1 (n=35, 30 BODIES) SINGLE-FRAME IK PROBLEM THE TREE PLAN'S FACTOR HAS A THIRD OF THE DENSE PLAN'S ENTRIES BUT IS SPREAD OVER 30 FRONTS 11 DEEP. FOR THE G1 TRAJECTORY AT T=128, ELIMINATING BODIES BEFORE FRAMES IS 3821 FRONTS DEEP AND NEEDS 52 MIB PER INSTANCE, AND CYCLIC REDUCTION OF THE FRAMES TRADES A DEPTH OF 128 FOR 8 AT THE PRICE OF A SEPARATOR TWICE AS WIDE.

Problem / order	Fronts	Cls.	Depth	$n_s^{\max}$	$\mathrm{nnz}(L)$	Storage
single-frame IK / tree	30	3	11	6	224	41.8 KiB
single-frame IK / dense	1	1	1	0	630	51.0 KiB
traj. 128 / tree, then time	3840	43	3821	47	186 k	52.3 MiB
traj. 128 / frames, chrono.	128	2	128	35	236 k	6.94 MiB
traj. 128 / frames, cyclic	128	3	8	70	383 k	9.21 MiB

factorization, upward and downward sweeps are

$$L = \mathrm{chol}(D), \quad Z = L^{-1}G, \quad M_f = C - Z^\mathsf{T}Z, \tag{4}$$

$$y = L^{-1}h, \qquad m_f = c - Z^\mathsf{T}y, \tag{5}$$

$$e = -L^{-\mathsf{T}}(y + Zs), \qquad z_{x_b} = \textstyle\sum_{\text{entries}} E_{\text{entry}}\, z_{\text{dst}}, \tag{6}$$

(s: the parent's separator solution, e: the eliminated solution, and the last equation recovers the substituted twists). L, Z, M_f overwrite D, G, C in place. A non-positive pivot sets a per-front flag and continues. Fronts are classified by $(n_e, n_s, n_{\mathrm{own}})$ and one operator is compiled per class. The `PlanReport` lists fronts, classes, levels, fill $\mathrm{nnz}(L)$, *widening* (which block rode which fronts), and flop and float counts (Table II).

Orders are action lists built by four recipes over sets or slices, concatenated with + and `sum`. The classical plans are instances (Tables I and II):

- `order.tree(u)`: per frame, in postorder, SUBSTITUTE(x_b) then ELIMINATE$(\{u_b\})$, so that $\bar{H}_b = T_b^\mathsf{T}\big(Q_b + \sum_{c\in\mathrm{ch}(b)} M_c\big)T_b + \mathrm{diag}(R_b, 0)$ with $T_b = [S_b \;\; X_b]$: the articulated-body recursion [5] with classes $(d_b; 6; d_{\mathrm{pa}(b)}{+}6)$.
- `order.dense(u)`: SUBSTITUTE all x, then ELIMINATE all u of a frame as one supernode. The slots acquire the body Jacobian as factor lists, and the single front of width $n = \sum_b d_b$ is the normal matrix $J^\mathsf{T}J{+}\lambda I$ of dense single-frame IK. Its twists form one *composite* edge assembled by the composite-rigid-body recursion.
- `order.frames(sel, time)`: SUBSTITUTE all x, then ELIMINATE(U_t) chronologically (fronts $[U_t; U_{t+1}]$ of class $(n; n)$, depth T) or by cyclic reduction [7], odd frames level by level (fronts $[U_t; U_{t-1}, U_{t+1}]$ of class $(n; 2n)$, depth $\lceil \log_2 T \rceil + 1$).
- `order.eliminate(o)`: ELIMINATE of a free block. Its position in the order decides which fronts carry it, and a `memory_budget` refuses a plan whose report exceeds it before any kernel is generated.

C. GPU schedule

A schedule is a partition of the fronts into *groups*, each executed sequentially by one worker in topological order, plus a worker kind and a storage placement. The worker kinds are `THREAD` (register operators, one thread per group),

(a) single-frame IK: blocks, relations, residuals, plan, schedule

```
import warp as wp, warp.branch as br, newton.ik as ik
robot = br.Articulation.from_newton(model)   # topology only
u = br.tangents(robot, instances=B, q=q_dof)  # u_b in R^d_b
x = br.twists(u, pose=body_q, S=S)  # X_b x_pa(b) + S_b u_b
@br.residual                  # rows by in-kernel wp.grad ...
def reach(b: br.Body, tgt: wp.vec3, p: wp.vec3) -> wp.vec3:
    return tgt - br.position(b, p)
@br.residual
def limit(d: br.Dof, lo: float, hi: float) -> float:
    v = d.q + d.value
    return wp.max(v - hi, 0.0) + wp.max(lo - v, 0.0)
# ... or analytic rows: @limit.jacobian, same call site
prob = br.Problem()
prob.add(reach, b=x[hands], tgt=targets, p=off, weight=w)
prob.add(limit, d=u.dofs, lo=q_lo, hi=q_hi, weight=10.0)
prob.add(br.damping(u, weight=br.per_instance(lam)))  # lam I
plan = br.plan(prob, br.order.tree(u))  # fronts [u_b; x_pa]
sched = br.Schedule.instance_blocks(plan)   # or levels, ...
solver = br.compile(plan, sched, device=dev)  # + .report()s
for it in range(24):           # application-owned GN/LM loop
    ik.fill_kinematics(model, joint_q, body_q, S, q_dof)
    solver.linearize()
    solver.step().dofs(u, out=dq)   # factor + solve: -H^-1 g
    ik.retract_dofs(model, joint_q, dq, joint_q)
```

(b) trajectory IK: add a frame axis and one temporal residual

```
u = br.tangents(robot, instances=B, frames=T, q=q_dof)
@br.residual
def smooth(a: br.Dof, b: br.Dof, inv_dt: float) -> float:
    return ((b.q + b.value) - (a.q + a.value)) * inv_dt
prob.add(smooth, a=u.dofs[:, :-1], b=u.dofs[:, 1:],
         inv_dt=30.0, weight=0.05)
plan = br.plan(prob, br.order.frames(u, time="cyclic"))
solver = br.compile(plan, br.Schedule.fused(plan), dev)
```

(c) contact response: one inertia problem, two plans

```
K = br.Problem(); K.add(br.quadratic(x, inertia))   # by ref.
K.add(br.damping(u.dofs, weight=arm_K))  # armature + drive K
dense = br.plan(K, br.order.dense(u))       # joint-space H
dsol = br.compile(dense, br.Schedule.cooperative(dense), dev,
                  columns=K_MAX, specialize=True, resolve=False)
dsol.factor()         # fused composite-inertia assembly + chol
dsol.solve(rhs=br.DofRows(J, rows, scale=-1.0), on=u,
           dofs=br.DofRows(Y, rows), active_columns=n_act)
# Y = H^-1 J^T: K_MAX columns, n_act[b] active per instance
tree = br.plan(K, br.order.tree(u))               # ABA fronts
tsol = br.compile(tree, br.Schedule.instance_blocks(tree),
                  dev, columns=6 * MAX_ACTIVE)
tsol.factor(); tsol.response_blocks(x, bodies=act, out=R)
tsol.solve(rhs=neg_p, on=x, columns=1).dofs(u, out=dv)
```

Fig. 2. **Adding a residual, a coupling, or a frame axis takes a few lines, and the plan and the schedule are one call each.** BRANCH calls are blue. (a) Single-frame IK: blocks, relations, and typed residuals whose Jacobian rows come from autodiff or from a user-supplied analytic function, under the tree plan and the instance-blocks schedule inside the application's LM loop. (b) Adding a frame axis and one temporal residual on adjacent-frame slices turns (a) into trajectory IK; the frames are eliminated by cyclic reduction and one block walks each chain of fronts. (c) Contact response from one inertia problem: the dense plan as a dofs-only tile solver on the contact solver's own row tables, and the tree plan returning the per-body response blocks $(AM^{-1}A^{\mathsf{T}})_{bb}$ and, per sweep, $M^{-1}J^{\mathsf{T}}p$.

`LANES` (register operators, one cooperative block per k instances), and `BLOCK` (tile operators, one block per group). Per instance each front owns static segments $D \,|\, G \,|\, C$ of one flat array, overwritten by $L \,|\, Z \,|\, M_f$, plus right-hand-side slabs solved over tiles of `rhs_tile` columns. (i) The `serial` schedule runs all fronts in postorder, one thread per instance, with factor, up, and down fused into one launch per solve. (ii) The `levels` schedule runs one thread per (instance, front), one launch per level. (iii) The `paths` schedule runs one thread per heavy path, $O(\log n_{\text{fronts}})$ launches per phase. (iv) The `instance_blocks` schedule runs one cooperative block per k consecutive instances, lane i running front $\lfloor i/k \rfloor$ of the current level for instance $i \bmod k$ with `wp.tile_sync()` publishing the level, so a whole solve is one launch. (v) The `cooperative` schedule runs, per level, one gather launch that assembles the fronts (3) and one launch per class that runs each front in shared memory with Warp's tile primitives. (vi) The `fused` schedule walks a maximal single-child chain with one block as a loop over runs of equal classes, reloading the previous link's factored C into D in shared memory. `THREAD` and `LANES` kernels dispatch on the front's class from a group table, so every grouping over the same class set shares the kernel. `BLOCK` kernels assemble through host-built gather tables. A class whose plain tile factor exceeds the shared-memory budget (101 376 B on sm_120) is emitted as a *split factor*: an in-place Cholesky and lower solve produce L and Z, and the Schur update $C \mathrel{-}= Z^{\mathsf{T}}Z$ runs over contiguous row blocks of C as callee sub-functions, so the kernel needs only its widest step (78 400 B for the G1's $(70; 140)$ front, 62 432 B for the H1's $(51; 102)$). Fronts within the budget use the unsplit factor. Two options specialize further to a fixed plan. `specialize=True` emits one kernel per (plan, launch) with the front offsets as literals and passes messages between the fronts of one thread in registers, so a factored solve on the `serial` or `instance_blocks` schedule is one launch. `resolve=False` builds a *dofs-only* solver for plans whose fronts carry no substituted twist as a column (dense and frame orders), dropping the recoveries of (6) and reading and writing the caller's arrays through `DofRows` row tables (Fig. 2c). All schedules compute the same factorization, differing only in summation order, and everything after `compile` is CUDA-graph capturable.

IV. Applications and Boundaries

This paper demonstrates BRANCH on inverse kinematics, from single poses to trajectories, and on contact response inside a Newton physics engine projected-Gauss–Seidel contact solver. Each integration fixes what stays in the application and what BRANCH generates, and each exposes a different decision of Fig. 1.

a) Inverse kinematics: Newton's batched Levenberg–Marquardt (LM) optimizer [3] forms the damped normal equations $(J^{\mathsf{T}}J + \lambda I)\,\delta = -J^{\mathsf{T}}r$ of its position, rotation, and joint-limit residuals with a tiled dense Cholesky, and accepts or rejects each step. `IKOptimizerBranch` subclasses it and overrides residual evaluation, the Jacobian step, and the linear solve, inheriting everything else. The one divergence is that BRANCH refreshes kinematics before every linearization, so J and r are consistent after a rejected step. Single-frame IK exposes the plan (tree or dense) and, within a plan, the schedule.

b) Kinematic trajectory optimization: Trajectory IK, also called kinematic trajectory optimization, chooses joint configurations $q_0, \ldots, q_{T-1}$ that track pose targets under smoothness, limit, and posture penalties. It runs the same LM iteration over T frames, and a single pose is the case $T{=}1$. The trajectory solver assembles the per-frame blocks into block-tridiagonal superblocks and masks fixed and padded frames. BRANCH receives the assembled blocks as by-reference quadratic terms on free frame blocks and replaces the factorization and substitution only, so for $T \geq 2$ the frames are eliminated as assembled joint-space blocks. This application exposes the temporal order (the chronological-frames or cyclic-frames plan) and the schedule.

c) Articulated contact response: A GPU projected-Gauss–Seidel (PGS) contact solver for articulated bodies [3] repeatedly needs the change in the whole robot's joint velocities caused by an impulse on one body. It exposes two response strategies, which are two plans of one problem. The dense plan forms, once per step, the joint-space response columns $Y = M^{-1}J^{\mathsf{T}}$ and reads them during the sweeps. The tree plan keeps per-body 6×6 response blocks $(AM^{-1}A^{\mathsf{T}})_{bb}$ and propagates the accumulated impulses along the tree once per sweep. BRANCH owns the factorization and the response operations of either from one inertia problem (Fig. 2c). Collision detection, row construction, friction, the sweeps, integration, and the choice of plan stay in the contact solver.

V. EVALUATION

The maps below show that no single configuration dominates and that the generated kernels are competitive with hand-written ones. Four questions organize them. A configuration is a plan and a schedule.

A. Protocol

All measurements except those of Sec. V-D run on one NVIDIA RTX PRO 6000 Blackwell GPU (188 SMs) with a fork of Newton and one revision of Warp, float32 device arithmetic, and float64 host oracles. Each timed region is captured into a CUDA graph after one eager call and replayed three times warm, then 20 times (single-frame IK, contact) or 10 times (trajectory) with device synchronization. We report medians excluding compilation and transfers. Timed boundaries are complete application work on B instances: a 24-iteration LM solve for single-frame IK, the 32-iteration LM solve including native assembly for trajectory optimization, and one collision-plus-solver substep for contact. In the single-frame IK comparison of Fig. 5 every solver starts from the same 17 configurations per problem, the nominal pose and 16 sampled uniformly within the joint limits, runs 24 iterations from each, and keeps the best solution by the matched cost. Its time is the whole batched multi-start solve per problem. A row enters a figure only if it passes the matched-accuracy gate against the native implementation of its cell. It passes if either its final cost is within 0.2 % of the reference's, or every quality metric is within 0.2 % of the reference's value plus an absolute allowance of 1 mm on end-effector position (RMS and maximum), 10^{-3} rad on rotation, and 10^{-5} rad on joint-limit violation. Trajectory rows also keep a linear-solve gate (relative residual below 10^{-4}, step error below 5×10^{-4}), and contact rows must match the native response blocks, active body sets, impulses, and joint velocities within 2×10^{-4} and stay inside the native five-step run-to-run band. The same gate applies to BRANCH and every external baseline. Robots (tangent coordinates): UR10 (6), Franka FR3 with hand (9), Allegro hand (16), ANYmal-C (18), Unitree G1 (35, free root), G1 with hands (49), H1 with hands (51). G1 single-frame IK tracks 14 body poses from a recorded jogging clip retargeted to the robot. The other robots track forward-kinematics targets of a perturbed nominal pose. The trajectory task tracks waypoints of such targets over T frames at 30 Hz. The G1 with hands and the H1 use a velocity-only objective (smoothness penalties on joint velocities only, so each frame couples only to its neighbours).

B. Q1: Which configuration wins?

It depends on the application and the workload: in single-frame IK the tree plan under a schedule that changes with robot and batch, in trajectory optimization the cyclic-frames plan except at large batch, and in contact the dense plan except on dense cube piles at large batch.

a) Single-frame IK: Single-frame IK covers seven robots and $B \in \{1, \ldots, 16384\}$ (49 cells; the $T{=}1$ row of Fig. 3 draws $B \leq 256$ and panel (a) of Fig. 4 all seven batch sizes). The candidates are the tree and dense plans under the six schedules, with the two tile schedules both table-driven and plan-specialised (16 candidates). The tree plan wins all 49 cells, and within it the schedule changes with robot and batch. The instance-blocks schedule wins 23 cells, the paths schedule 19, the serial schedule 6, and the plan-specialised cooperative schedule 1. In 21 cells the runner-up is within 3 %.

b) Kinematic trajectory optimization: The $T \geq 8$ rows cover $B \in \{1, 16, 64, 256\}$ and $T \in \{8, \ldots, 2048\}$ on the seven robots (137 cells). The candidates are the chronological-frames and cyclic-frames plans under the fused and cooperative schedules at block sizes 64 and 128. At $B{=}256$, $T{=}2048$ the G1, the G1 with hands, and the H1 are out of memory: the native reference solver does not fit. The cyclic-frames plan wins 114 cells and the chronological-frames plan 23, and all 40 cells of the two arms are cyclic. On the G1 and H1 the cyclic-frames plan, whose $(70; 140)$ and $(51; 102)$ fronts run as split factors (Sec. III-C), wins at $B \leq 64$ from $T{=}32$. At $B{=}1$ it is 1.7–23× (G1) and 3.0–37× (H1) faster than the chronological-frames plan from $T{=}32$, and at $B{=}256$ the chronological-frames plan wins on both, where the batch alone fills the device. The cyclic-frames plan wins 15 of the G1 with hands' 19 cells. Within a plan the schedule changes less: the fused schedule wins 113 cells and the cooperative schedule 24, and the runner-up is within 3 % in 81 cells.

c) Articulated contact response: Panel (i) of Fig. 4 maps four contact workloads at $B \leq 16384$ on the RTX PRO 6000. In all, ten workloads were measured at B from 1 to 262144 (54 valid cells) with eight candidates. The plan changes with the workload and, on the cube piles, with batch. The dense plan wins all 29 robot-dominated cells (the G1 and H1 worlds) by 1.08–2.25× over the best tree-plan row, and the four Isaac Lab tasks by 1.17–2.03×. On the FR3 cube piles the tree plan wins 12 of 21 cells, where the pile is dense or the batch large: the cube shower from $B{=}64$ and the falling cubes at B 1024–4096 (up to 2.67×), while at $B \leq 256$ the falling cubes go to the dense plan by 1.27–1.53×. The dense plan holds the standing G1 to $B{=}262144$ and the H1 to 32768 (larger batches do not fit). The schedule never changes a winner within a plan by more than 3 %, and in 45 of the 54 cells the runner-up is within 3 %.

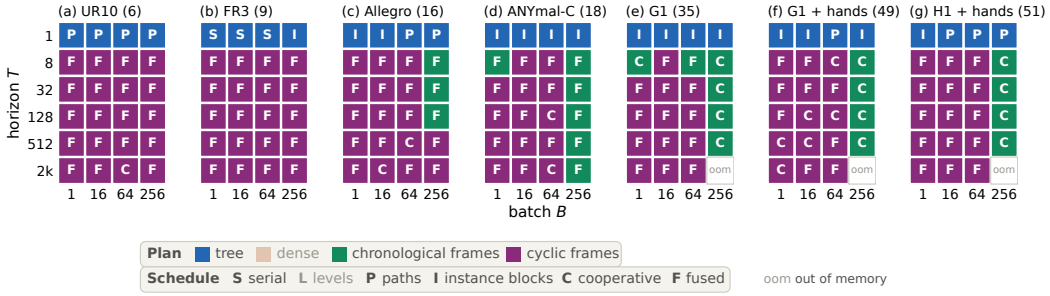

Fig. 3. **The best plan and schedule move with robot, batch, and horizon.** The fastest gate-passing BRANCH configuration per cell, one panel per robot, horizon by batch. The $T{=}1$ row is single-frame IK and $T \geq 8$ is trajectory optimization; the G1 with hands and the H1 penalize joint velocities only (Sec. V-A). Colour marks the plan and the letter the schedule. The tree plan wins every single-frame IK cell. Over the 137 trajectory cells the cyclic-frames plan wins 114 and the chronological-frames plan 23, and the fused schedule wins 113 and the cooperative schedule 24; oom marks out of memory.

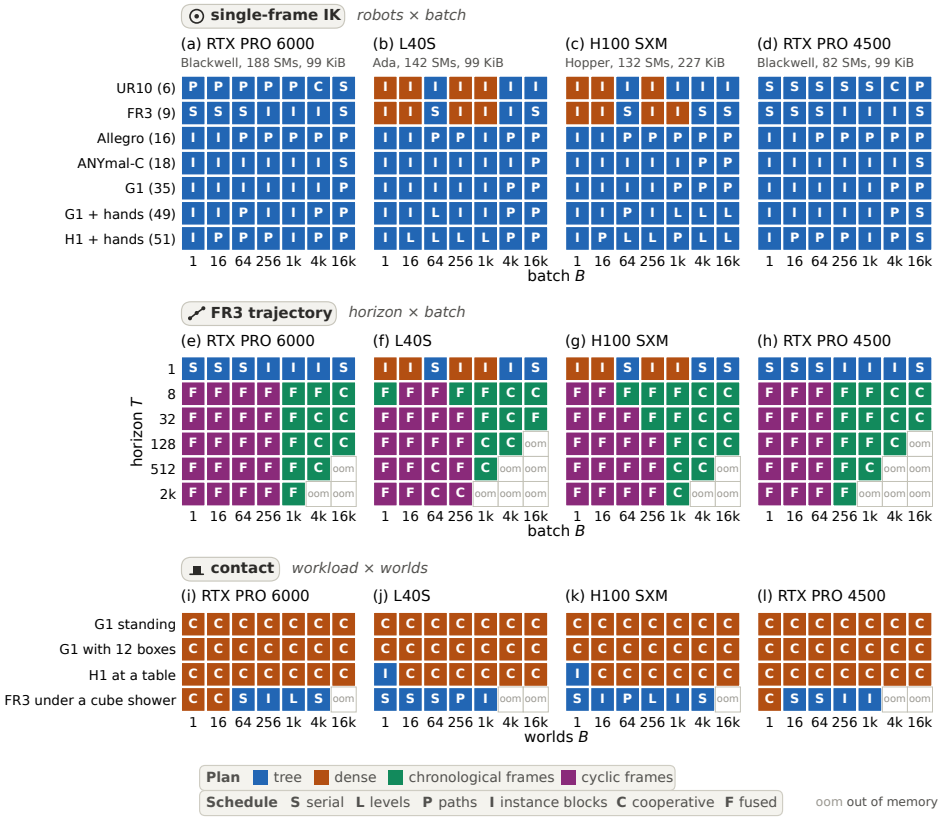

Fig. 4. **The best plan and schedule also depend on the GPU.** On the H100 and L40S the dense plan wins single-frame IK on the two arms at $B \leq 1024$ (7 and 8 cells) and the levels schedule wins 7 and 5 humanoid cells; neither wins a cell on the two Blackwell GPUs. Row 1 is single-frame IK over robots $\times$ batch, row 2 the FR3 trajectory map, and row 3 the contact map, one column per GPU (architecture, SMs, and KiB of shared memory per block in the header); oom marks out of memory. In the FR3 trajectory row the chronological-frames plan wins every cell at $B \geq 1024$ on every GPU.

C. Q2: How much does choosing correctly matter?

A near miss usually costs little. The second-fastest candidate is 4 % slower than the fastest in the median single-frame IK cell and 14 % in the worst, 3 % and 4.7× over the trajectory cells (the worst is the H1 at $B{=}16$, $T{=}2048$, where the chronological-frames plan is second), and 0.2 % and 2.03× in contact. One configuration for a whole map costs more. The best single configuration per map, the one with the best geometric-mean ratio to the per-cell winner, is at most 1.53× slower in single-frame IK (tree plan, instance-blocks schedule; H1, $B{=}4096$), 1.34× in trajectory optimization (cyclic frames, fused; FR3, $B{=}16384$, $T{=}128$), and 2.67× in contact (dense, cooperative; cube shower, $B{=}4096$).

D. Does the optimal configuration depend on the GPU?

Yes. Fig. 4 repeats the maps on an H100 (132 SMs), an L40S (142 SMs), and an RTX PRO 4500 (82 SMs, Blackwell). In single-frame IK the tree plan wins every cell on the RTX PRO 4500 and one winner changes by more than 3 %. On the H100 and L40S two configurations that win nothing on the RTX PRO 6000 appear: the one-front dense plan wins the two arms at $B \leq 1024$ (7 and 8 cells) and the tree plan under the levels schedule wins 7 and 5 humanoid cells, so the winner changes in 24 of 49 cells on

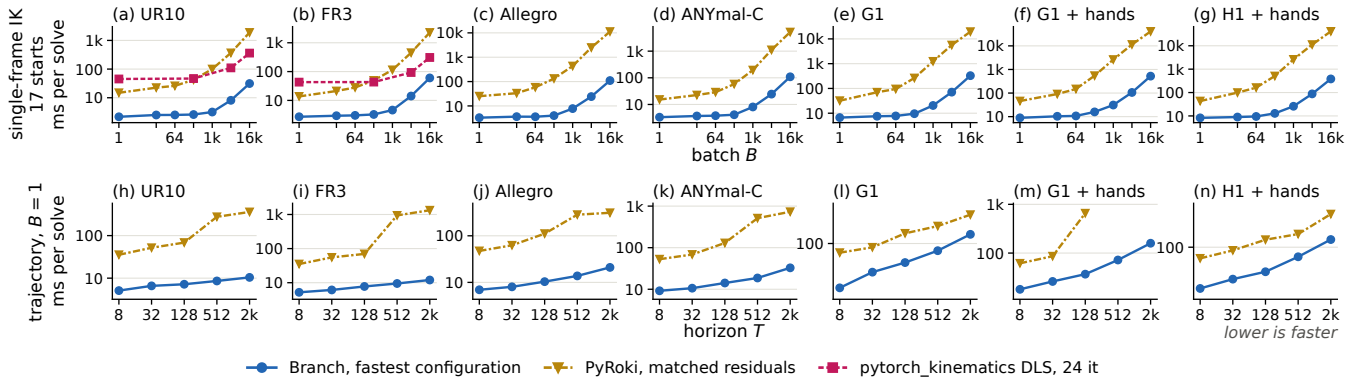

Fig. 5. **BRANCH is faster than PyRoki and pytorch_kinematics at every tested batch size in single-frame IK, and faster than PyRoki at every tested horizon in trajectory optimization, on the same problems at the same iteration budget.** Milliseconds per solve on a log axis, lower is faster, one panel per robot. (a)–(g) Single-frame IK against batch size: every solver starts from the same 17 configurations per problem, the nominal pose and 16 sampled within the joint limits, runs 24 iterations from each, and keeps the best solution by the matched cost; the time is the whole multi-start solve per problem. (h)–(n) Trajectory optimization at B=1 against horizon, single start, 32 iterations. BRANCH is its fastest configuration per cell; PyRoki/jaxls [14], [15] solves the matched residuals; pytorch_kinematics [24] runs its damped least squares.

the H100 (13 by more than 3 %) and in 19 on the L40S (11). In contact the winner changes by more than 3 % in 2 cells on the L40S and 3 on the H100, all at $B \leq 64$, and in none on the RTX PRO 4500. At T=8 the cyclic-frames and chronological-frames plans are within 2 % of each other at $B \leq 64$ on every GPU, and the drawn winner flips within that band. From B=256 the chronological-frames plan leads by 1–9 % on the three other GPUs, while the RTX PRO 6000 keeps the cyclic-frames plan through B=256. From B=1024 the chronological-frames plan wins every FR3 cell that fits on every GPU (by 8–37 % at T=8). Matching the fastest kernel to each GPU by hand would mean maintaining GPU-specific kernels; with BRANCH the change is a plan or schedule name.

E. Q3: How competitive is the fastest BRANCH *configuration?*

Faster than PyRoki and pytorch_kinematics at every tested batch size and horizon, and at native throughput in contact.

a) Single-frame IK: Panels (a)–(g) of Fig. 5 compare complete multi-start solves at matched accuracy against single-frame IK libraries on all seven robots at B from 1 to 16384, with PyRoki's root pose as a variable in Newton's tangent convention on the free-root robots. BRANCH is faster than PyRoki [14] and than pytorch_kinematics' damped-least-squares solver [24] (a Python loop of small kernels, joint limits by clamping) at every tested batch size. With 17 starts every BRANCH and native row passes the gate. PyRoki misses it once, on the UR10 at B=256 by 3 mm, and pytorch_kinematics misses on the FR3 from B=256 by 3–12 mm.

b) Trajectory optimization: Panels (h)–(n) of Fig. 5 compare complete LM solves on the matched objective at B=1. BRANCH is faster than PyRoki/jaxls on every robot at every tested horizon.

c) Hand-written kernels: BRANCH is at or above native end to end: the dense-plan adapter runs the full Isaac Lab [18] G1 velocity task at 1.02× and 1.03× the native environment-step throughput at $B \in \{4096, 16384\}$ and the physics step at 1.00× and 1.02×.

TABLE III

PLANNING COSTS MILLISECONDS, COMPILING SECONDS TO MINUTES. SETUP COST OF ONE CONFIGURATION BEFORE ITS FIRST TIMED STEP, AS RANGES OVER THE CONFIGURATIONS OF THE G1 SINGLE-FRAME IK PROBLEM, THE FR3 TRAJECTORY AT T=128, AND THE G1 CONTACT SOLVER. PLANNING: BUILDING THE ELIMINATION PLAN AND ITS REPORTS. COMPILE: KERNEL GENERATION AND JIT, FROM SCRATCH (COLD CACHE) AND WITH THE KERNELS ALREADY IN WARP'S KERNEL CACHE (WARM). FIRST CALL: MODULE LOAD AND FIRST LAUNCH. COMPILE TIME GROWS WITH THE SIZE OF THE GENERATED KERNELS: THE PLAN-SPECIALISED SERIAL AND INSTANCE-BLOCKS SCHEDULES UNROLL EVERY FRONT OF THE 35-DOF G1 WITH LITERAL OFFSETS (THE 210 S AND 241 S MAXIMA), AND THE FR3 TRAJECTORY FRONTS ARE 9 WIDE.

Application	Planning (ms)	Compile (s) cold cache	Compile (s) warm cache	First call (ms)
single-frame IK (G1)	1.7–2.2	3.2–210	<0.1	10–30
trajectory (FR3, T=128)	8.5–11	5.7–33	0.02–0.54	38–954
contact (G1)	–	2.8–241	<0.1	–

single-frame IK: 10 of 16 configurations compiled from scratch; the other 6 found their kernels compiled earlier in the same process; trajectory: 8 configurations, each compiled with an empty and with a reused kernel cache; contact: 8 configurations; the adapters record the constructor total (plan, compile and storage allocation): cold = the cell that compiled the kernels, warm = a later cell of the same configuration; – not recorded

F. Q4: What does trying an alternative cost?

Milliseconds to plan and seconds to minutes to compile. Table III lists the setup cost of a new configuration: planning is symbolic, takes at most 3.4 ms for the single-frame IK plans of Table II and 0.4–1.2 s for trajectory plans at T=2048, and its reports refuse an infeasible plan before compilation. Compilation dominates: from scratch, up to 210 s for a G1 single-frame IK configuration, 6–33 s for an FR3 trajectory configuration, and up to 241 s for a G1 contact configuration, and under a second in every row once the kernels are in the cache.

VI. Discussion and Limitations

BRANCH contributes no new factorization. Its plans and schedules are the known recursions and mappings of Table I, generated from one description. Each alternative costs seconds to minutes to compile (Sec. V-F), and BRANCH does not choose the configuration: the maps are measured, and the crossovers move with the GPU (Sec. V-D). The library solves the local quadratic and leaves outer loops, damping, and retraction to the application. Gradients through the factorization are not provided. Orders are recipes plus explicit elimination, without fill-reducing heuristics. Loop-closing joints are unsupported. Arithmetic is float32, and a failed pivot sets a flag without repair. On the worst ill-conditioned G1 single-frame IK instance ($\mathrm{cond}(H)$ near 5×10^4) the float32 tree and dense solves deviate 0.26 and 0.17 from a float64 solve (largest relative ∞-norm step error over the batch) and native's tiled Cholesky 0.035. The medians are 10^{-5} to 10^{-7}.

VII. Conclusion

Performance requires specialization, the right specialization changes with robot, batch, horizon, and GPU, and building each alternative by hand is costly. BRANCH turns that specialization into compilation for the structured linear solves at the heart of robotics optimizers and simulators. The problem is described once, the application states its elimination plan and its GPU schedule, and BRANCH generates the kernel for that configuration in Warp. Specialization matters: the best plan and schedule vary along every workload axis, and even the best single configuration for a whole map is slower than the per-cell winner (Sec. V). Generated code holds up against hand-written kernels and existing libraries: a Newton physics engine projected-Gauss–Seidel contact solver runs at 1.02–1.03$\times$ its native throughput with generated kernels in place of the hand-written ones, and at matched accuracy BRANCH is faster than PyRoki and pytorch_kinematics at every tested batch size in single-frame IK and faster than PyRoki at every tested horizon in trajectory optimization (Sec. V). A developer therefore specializes a solver to a new workload by enumerating plans and schedules, without re-deriving and re-implementing each alternative.

Acknowledgments

Generative AI disclosure: the code of the released library and of the experiment scripts, the text of Sections I–VII, Figs. 1–5, and Tables I–III were produced with Claude Fable 5.1 (Anthropic). The authors specified the technical content, the outlines, and the constraints of each section, directed and reviewed each step of the work, edited all text, and verified every claim, equation, algorithm, experimental result, figure, and citation against primary sources and the recorded measurements. All reported measurements are recorded runs of the released code on the stated hardware. No AI system is an author.

References

[1] B. Sundaralingam, A. Murali, and S. Birchfield, "cuRoboV2: Dynamics-aware motion generation with depth-fused distance fields for high-DoF robots," arXiv:2603.05493v2, 2026.

[2] B. Plancher, S. M. Neuman, R. Ghosal, S. Kuindersma, and V. Janapa Reddi, "GRiD: GPU-accelerated rigid body dynamics with analytical gradients," in *IEEE International Conference on Robotics and Automation*, 2022, pp. 6253–6260.

[3] Newton contributors, "Newton physics engine," Software, 2026, https://github.com/newton-physics/newton.

[4] NVIDIA Warp contributors, "Warp: a Python framework for high performance GPU simulation and graphics," Software, version 1.16.0, 2026, https://github.com/NVIDIA/warp.

[5] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *The International Journal of Robotics Research*, vol. 2, no. 1, pp. 13–30, 1983.

[6] ——, *Rigid Body Dynamics Algorithms*. New York: Springer, 2008.

[7] D. Heller, "Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems," *SIAM Journal on Numerical Analysis*, vol. 13, no. 4, pp. 484–496, 1976.

[8] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: the SPIKE algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006.

[9] B. Wingo, A. S. Sathya, S. Caron, S. Hutchinson, and J. Carpentier, "Linear-time differential inverse kinematics: an augmented Lagrangian perspective," in *Robotics: Science and Systems*, 2024.

[10] MuJoCo Warp contributors, "MuJoCo Warp," Software, revision d8a241ef1643, 2026, https://github.com/google-deepmind/mujoco_warp.

[11] NVIDIA Corporation, "PhysX SDK," Software, version 5, 2026, https://github.com/NVIDIA-Omniverse/PhysX.

[12] R. Schwan, D. Kuhn, and C. N. Jones, "GPU-accelerated Cholesky factorization of block tridiagonal matrices," arXiv:2601.03754, 2026.

[13] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "MPCGPU: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the GPU," in *IEEE International Conference on Robotics and Automation*, 2024.

[14] 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*, 2025.

[15] B. Yi, "jaxls: Sparse nonlinear least squares in JAX," Software, 2025, https://github.com/brentyi/jaxls.

[16] F. Dellaert and M. Kaess, "Factor graphs for robot perception," *Foundations and Trends in Robotics*, vol. 6, no. 1–2, pp. 1–139, 2017.

[17] S. Agarwal, K. Mierle, and The Ceres Solver Team, "Ceres Solver," Software, version 2.2, 2023, https://github.com/ceres-solver/ceres-solver.

[18] M. Mittal, P. Roth, J. Tigue, *et al.*, "Isaac Lab: A GPU-accelerated simulation framework for multi-modal robot learning," arXiv:2511.04831, 2025.

[19] Y. Ze, J. P. Araújo, J. Wu, and C. K. Liu, "GMR: General motion retargeting," Software, 2025, https://github.com/YanjieZe/GMR.

[20] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia: SIAM, 2006.

[21] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 519–530.

[22] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022, pp. 703–718.

[23] K. Cheshmi, S. Kamil, M. M. Strout, and M. Mehri Dehnavi, "Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 13:1–13:13.

[24] S. Zhong, T. Power, A. Gupta, and P. Mitrano, "PyTorch Kinematics," Software, v0.7.1, 2024, https://github.com/UM-ARM-Lab/pytorch_kinematics.