

pRRT: GPU-Parallel RRT-Connect for Fast, Consistent, and Low-Cost Motion Planning

Chih H. Huang^{1*}, Pranav Jadhav^{2*}, Brian Plancher^{3,4}, and Zachary Kingston²

Abstract—Sampling-based motion planning algorithms, like the Rapidly-Exploring Random Tree (RRT) and its widely used variant, RRT-Connect, provide efficient solutions for high-dimensional planning problems faced by real-world robots. However, these methods remain computationally intensive, particularly in complex environments that require many collision checks. To improve performance, recent efforts have explored parallelizing specific components of RRT such as collision checking, or running multiple planners independently. However, little has been done to develop an integrated parallelism approach, co-designed for large-scale parallelism. In this work we present pRRT, a RRT-Connect based planner co-designed for GPU acceleration across the entire algorithm through parallel expansion and SIMD-optimized collision checking. We evaluate the effectiveness of pRRT on the MotionBenchMaker dataset using robots with 7, 8, and 14 degrees of freedom (DoF). Compared to the state-of-the-art, pRRT achieves as much as a 10× speedup on constrained reaching tasks with a 5.4× reduction in standard deviation. pRRT also achieves a 1.4× reduction in average initial path cost. Finally, we deploy pRRT on a 14-DoF dual Franka Panda arm setup and demonstrate real-time, collision-free motion planning with dynamic obstacles. We open-source our planner to support the wider community.

I. INTRODUCTION

Motion planning is a fundamental problem in robotics that involves finding collision-free motion paths through a robot’s configuration space [1–3]. While there are many approaches to planning, sampling-based motion planning (SBMP) approaches are widely used due to their generality and efficiency in higher-dimensions. One of the most popular SBMP algorithms is the Rapidly-Exploring Random Tree (RRT) [4] and its bidirectional variant RRT-Connect [5]. RRT-Connect’s design biases it towards quickly solving problems involving large open spaces, even in high dimensions. However, its performance suffers in cluttered environments such as in constrained reaching tasks [3].

One potential avenue for improving planning performance is through the use of parallelism. Existing parallelized approaches fall into one of three categories: (1) *high-level parallelism* by executing multiple RRT instances simultaneously [6, 7], (2) *mid-level parallelism* by executing multiple

This material is based upon work supported by the National Science Foundation (under Award 2411369). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding organizations.

¹CHH is with Columbia College, Columbia University, New York, NY. chih.h@columbia.edu

²PJ and ZK are with the Department of Computer Science, Purdue University, IN. {jadHAV14, zkington}@purdue.edu

^{3,4}BP is with Dartmouth College, Hanover, NH and Barnard College, Columbia University, New York, NY. plancher@dartmouth.edu

*Equal Contribution.

sample-and-grow iterations simultaneously [8], and (3) *low-level parallelism* over primitive operations such as collision checking [9–12]. While prior works have mostly focused on one form of parallelism, little has been done on integrating an approach over multiple levels of parallelism to maximize performance on modern parallel hardware (e.g., GPUs).

To address this gap, we introduce pRRT, a parallel RRT-Connect-based algorithm designed for GPU acceleration across multiple parallelism levels: mid-level planning iteration parallelism and low-level primitive operation parallelism. Our approach’s performance is driven by three key design decisions: (1) concurrent sampling, expansion and connection of start and goal trees via GPU threads, (2) SIMD-optimized collision checking to quickly validate edges, inspired by the SIMD-optimized validation of Thomason et al. [9], and (3) efficient memory management between block and thread level parallelism, ultimately reducing expensive memory transfer overheads. Compared to other GPU-based SBMP algorithms (e.g., [7, 10, 13]), our approach supports high-dimensional manipulators and achieves sub-millisecond planning performance.

By integrating parallelism across planning iterations and primitive operations, pRRT achieves fast, consistent, and efficient planning. We evaluate pRRT against state-of-the-art CPU- and GPU-based motion planners on the MotionBenchMaker dataset [14] using robots with 7, 8, and 14 degrees of freedom (DoF). pRRT achieves as much as a 10× speedup on constrained reaching tasks over state-of-the-art planners in complex environments. Importantly, pRRT exhibits a 5.4× reduction in planning time standard deviation across all problems, indicating consistency across a variety of environments and robots. pRRT also produces low-cost initial paths, achieving a 1.4× reduction in average initial path cost compared to existing approaches. These results highlight the benefits of software-hardware co-designed GPU parallelism for accelerating SBMPs. Finally, we deploy pRRT on a 14 DoF dual Franka Panda arm setup and demonstrate real-time, collision-free motion planning with dynamic obstacles (see Fig. 7). We open-source our planner at: <https://github.com/CoMMLab/pRRT>.

II. BACKGROUND AND RELATED WORK

A. The RRT-Connect Planning Algorithm

There is a vast array of SBMP approaches each with their own modifications to core operations of sampling, nearest neighbors search, steering, and collision checking [1, 3, 15]. In this work, we focus on the RRT and RRT-Connect algorithms due to their fast single-query performance.

Many modifications have been proposed to the RRT algorithm, such as the growth of multiple trees [5, 16, 17], biased sampling [18–21], and environment preprocessing [22]. RRT-Connect [5] is a widely used variant that simultaneously expands two trees: one rooted at the start configuration and the other at the goal configuration. The balanced variant of RRT-Connect [23] selects the smaller tree for tree growth at each iteration, which is more efficient for problems with unbalanced difficulty around the start and goal configurations. Each iteration of the algorithm also includes an additional connect operation, where the newly added node is greedily extended toward the opposing tree until a collision is detected. The planner terminates when the two trees successfully connect, forming a complete path. Additionally, sampling heuristics such as dynamic domain sampling [18] can be easily integrated to improve performance.

RRT-Connect is well known for accelerating planning in large, open spaces [5]. However, many problems contain *narrow passages* created by environment geometry (e.g., when a robot must reach into a container), where motion is heavily constrained. In such cases, a large number of samples must be evaluated, motivating the extension of trees in parallel. A critical component of this process is motion validation, which depends on collision checking.

Collision checking is empirically recognized as a primary bottleneck in motion planning [10]. This has motivated diverse algorithmic strategies, including hierarchical collision detection [24], restructured computations for faster expected performance [25], delayed or lazy checking [26, 27], exact and swept-volume approaches [28–30], and caching techniques [31]. Given the independence of collision-checking operations across candidate samples, parallelism offers a natural and promising direction for accelerating this stage of the planning pipeline.

B. Parallel Acceleration of Planning Algorithms

Parallelization of planning algorithms has been studied since the advent of the field [32, 33]. In particular, Amato et al. [34] noted early on that SBMPs (e.g., [35]) were “embarrassingly parallel.” We categorize the parallelism of SBMPs into three categories: high-, medium-, and low-level parallelism. High-level parallelization refers to running multiple isolated planners in parallel; medium-level parallelization refers to running multiple iterations within the same planning framework in parallel; low-level parallelization refers to parallelizing primitive operations within each iteration of a single planning algorithm.

In terms of CPU parallelism, most parallel planners achieve high- or medium-level parallelization, such as running many instances of the planner simultaneously [6], running RRT iterations in parallel on one tree [8, 36], and parallelizing search with forests of trees [37, 38]. With regard to low-level parallelism, VAMP [9, 39] utilizes vectorized motion validation through CPU Single Instruction, Multiple Data (SIMD) instructions leading to data parallelism in the collision checking process. However, CPU-based parallelism

is limited in scope due to a relatively small number of cores and the limitations of current CPU SIMD operations.

GPUs provide massive parallelism through the Single Instruction, Multiple Thread (SIMT) model over thousands of cores. While powerful, GPUs require careful algorithm design to fully leverage the hardware. High-level parallelization strategies have been proposed by Hidalgo-Paniagua et al. [7] and Jacobs et al. [40], offering substantial speedups in environments with narrow passages. A number of different GPU-based SBMPs have been proposed such as GMT [13], Pk-RRT [41], Kino-PAX [42], and the RRT-like planner used by cuRobo [43] which leverages medium-level parallelism for parallel sampling, rollouts, and tree growth. Work on low-level parallelism has been focused on accelerating collision checking through clustering [44], spatial hashing [45], hierarchy construction [46], and checking discretized configurations along a motion in parallel [10–12].

Finally, we note that outside of SBMP algorithms there have similarly been many developments on GPU-parallelization of other classes of planning and control algorithms including both search-based planning [47–49], as well as optimal control-based techniques [43, 50–54].

Despite these many advances and clear evidence of performance gains, existing works that parallelize SBMP algorithms only utilize parallelism at one specific level, limiting their overall performance. To overcome this gap, we developed pRRT, a GPU-based planner that leverages co-designed parallelism across multiple levels to surpass state-of-the-art performance.

III. PRRTC ALGORITHM

In this section, we discuss the algorithmic design and implementation of pRRT as diagrammed in Fig. 1 and Algs. 1 and 2. Overall, pRRT retains similar primitive operations and structure as the vanilla RRT-Connect algorithm, but it implements aggressive mid- and low-level parallelism designed to leverage the computational structure of modern GPUs. At the mid-level, pRRT runs hundreds of parallel RRT-Connect iterations asynchronously across both trees (Alg. 1 lines 4–19). This enables pRRT to explore the configuration space faster and find higher quality paths. At the low-level, pRRT parallelizes nearest neighbors (NN) search and discretized edge collision checking (CC).

This parallelism is realized via blocks of parallel threads on the GPU. Each block is responsible for running independent RRT-Connect iterations with a constant number of threads t_1, \dots, t_n to parallelize low-level operations. NN search and CC against the robot itself and environment are parallelized across individual threads (shown in green). For forward kinematics (FK), threads are logically organized into groups of four to parallelize matrix operations (shown in pink) building on previous GPU designs [43].

Our block and thread organization is critical for efficiency. Threads within a block share a physical processor and low-latency shared memory, enabling fast synchronization for fine-grained parallel operations. Since independent RRT-Connect iterations require little coordination, blocks can op-

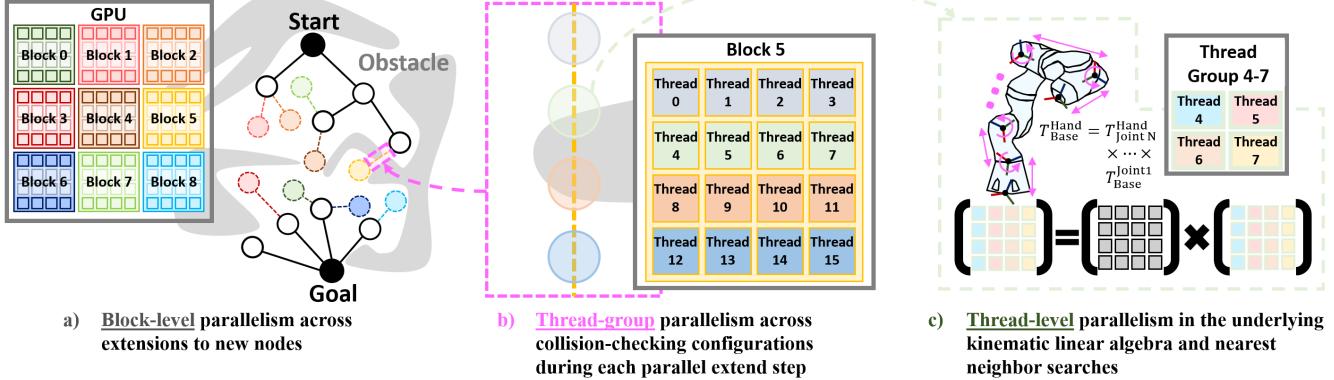


Fig. 1: A conceptual illustration of pRRTc in an abstract configuration space operating across three levels of GPU-parallelism. a) Separate GPU blocks extend both the start and goal trees with multiple samples in parallel. b) Within each block, parallel groups of four threads compute collision checks across the multiple configurations that correspond intermediate states of a motion. c) Individual threads parallelize the underlying kinematic linear algebra needed for these collision checks. Individual threads also parallelize nearest neighbors search.

Algorithm 1: pRRTc($c_{start}, c_{goal}, \delta$)

```

1:  $T_a.init()$ ,  $T_b.init()$ 
2:  $T_a.add(c_{start})$ ,  $T_b.add(c_{goal})$ 
3: for block index  $i = 0 \dots N_{samples}$  do in parallel
4:   for iteration  $iter \leftarrow 0 \dots \text{max\_iters}$  do
5:      $T_s^i, T_o^i \leftarrow \text{argmin}(|T_a|, |T_b|)$ ,  $\text{argmax}(|T_a|, |T_b|)$ 
6:      $c_{rand}^i \leftarrow \text{RANDOM\_CONFIG}()$ 
7:      $c_{nns}^i \leftarrow \text{NEAREST\_NEIGHBOR}(c_{rand}^i, T_s^i)$ 
8:      $c_{new}^i, \text{valid}^i \leftarrow \text{pEXTEND}(c_{rand}^i, c_{nns}^i, \delta)$ 
9:     if  $\text{valid}^i$  then // Add and Greedily extend to  $T_o^i$ 
10:       $T_s^i.add(c_{new}^i)$ 
11:       $c_{nno}^i \leftarrow \text{NEAREST\_NEIGHBOR}(c_{new}^i, T_o^i)$ 
12:       $n_{ext} \leftarrow \text{DISTANCE}(c_{new}^i, c_{nno}^i) / \delta$ 
13:       $v_{ext} \leftarrow (c_{nno}^i - c_{new}^i) / \delta$ 
14:      for  $i_{ext} \leftarrow 1 \dots n_{ext}$  do
15:         $c_{ext}^i \leftarrow c_{new}^i + v_{ext}$ 
16:         $c_{new}^i, \text{valid}^i \leftarrow \text{pEXTEND}(c_{new}^i, c_{ext}^i, \delta)$ 
17:        if  $\text{valid}^i$  then  $T_s^i.add(c_{new}^i)$ 
18:        else break
19:        if  $i_{ext} = n_{ext}$  then return PATH( $T_a, T_b$ )
20: return Failed

```

erate asynchronously and access the global trees only when needed. This allows each block to explore the search space at its own pace with minimal synchronization overhead, while still leveraging the progress made by others. To handle potential race conditions on the shared global trees, pRRTc employs atomic primitives to ensure new configurations are written safely to memory.

Finally, when serial operations are required, they are executed by a designated lead thread t_1 within each block. This role is primarily reserved for intermittent, higher-level control flow tasks, such as selecting which tree to expand, sampling new states, and updating the global tree.

Algorithm 2: pEXTEND(c_{rand}, c_{nn}, δ)

```

1: valid  $\leftarrow$  True
2:  $c_{new} \leftarrow \text{INTERPOLATE}(c_{rand}, c_{nn}, \delta)$ 
3: for  $i \leftarrow 1 \dots N_{cc}$  do in parallel thread-groups
4:    $c_{check} \leftarrow \text{INTERPOLATE}(c_{rand}, c_{new}, i/N_{cc} * \delta)$ 
5:   valid &= COLLISION_CHECK( $c_{check}$ )
6: return  $c_{new}$ , valid

```

The complete algorithm is shown in Alg. 1 and Alg. 2 where c refers to joint space configurations, δ is the extension range parameter, T_a is the start tree, T_b is the goal tree, T_s^i and T_o^i refer to the tree being extended and the opposite tree for block i , and N_{cc} is the collision checking resolution. While omitted above for improved readability and clarity of our overall approach, we also use the dynamic domain sampling heuristic described in Yershova et al. [18].

A. SIMT Collision Checking

As mentioned previously, collision checking (CC) is widely recognized as a major computational bottleneck in motion planning [10], particularly when validating long or numerous edges in high-dimensional configuration spaces. Many prior approaches aim to accelerate CC by spatially distributing checks along an edge using binary subdivision [55] and SIMD parallelism [9]. We build on these approaches and use GPU parallelism to check a set of discrete points along an edge simultaneously.

Each edge in our planner is discretized according to a pre-defined CC resolution, and the entire set of points along that edge is validated in parallel. Specifically, we assign a group of four threads to each configuration to exploit the 4×4 matrix operations required in forward kinematics (FK) inspired by past GPU designs [43]. For example, if an edge is discretized into 32 configurations, the GPU block will contain $32 \times 4 = 128$ threads. Since the maximum extension length is bounded by the planner’s RRT range parameter, we can preallocate these threads at kernel launch time, avoiding iteration and synchronization overhead. Furthermore, unlike

approaches that store all intermediate matrices, we minimize shared memory usage by overwriting intermediate results whenever possible. For high-DOF robots such as Baxter, this strategy reduces shared memory consumption by up to 7 \times , enabling a larger number of concurrent blocks per streaming multiprocessor (SM), critical to overall planning throughput.

We further accelerate this routine by enabling early exits. When a collision is detected along an edge, the candidate edge is invalidated and all checks along that edge are halted. While conceptually simple, implementing this efficiently on the GPU is challenging as synchronizing across all threads in a block introduces undesirable overheads. To avoid this, we leverage warp⁴ primitives, which allow fast coordination among groups of 32 threads through low-latency GPU memory. Since our implementation assigns four threads per CC, each warp can evaluate and efficiently enable early exit synchronization for eight checks in parallel. For environment collisions, where the workload is balanced, threads assume all peers remain active. For self-collisions, where workloads vary due to differing numbers of collision spheres, threads use a single-cycle *active mask* [56] to identify which threads are active and share results only among them.

We use the `foam` tool [57] to compute a set of bounding spheres representing the robot offline. These spheres are used at runtime in a two-stage collision checking process. First, a low-resolution FK and CC pass is run, flagging links that may be in collision. Only those links are re-checked using a high-resolution pass, which leverages finer-grained sphere models for more precise analysis. This hierarchical approach reduces expensive CC calls while preserving accuracy.

For static environments, such as those from MotionBenchMaker [14], we represent obstacles using primitive geometry. In these settings, the small number of obstacles allows the full set of environment primitives to be cached in low- or mid-level GPU memory throughout most of the planning process, enabling low-latency access across threads in each block. For real-world experiments, we integrate with Nvblox [58] to perform CC against dynamically updated Euclidean Signed Distance Field (ESDF) maps.

B. Nearest-Neighbor (NN) Search

We use divide-and-conquer parallelism to accelerate our NN search. Within a block of n threads, pRRTc divides each NN query into n subproblems, with each thread assigned to search a disjoint subset of the existing tree for a local NN. Once these are found, pRRTc constructs the final global NN using a tree-based parallel reduction [59]. For a given tree of size T , serial NN requires T comparisons while parallel NN requires $\lceil T/n + \log(n/2) \rceil$. This approach avoids significant computational overhead and remains competitive with serial approaches even with low total node counts.

⁴A “warp” represents 32 contiguous threads on the same GPU-core. These threads work in lock-step due to the design of NVIDIA GPU hardware and as such have native implicit synchronization at the hardware level.

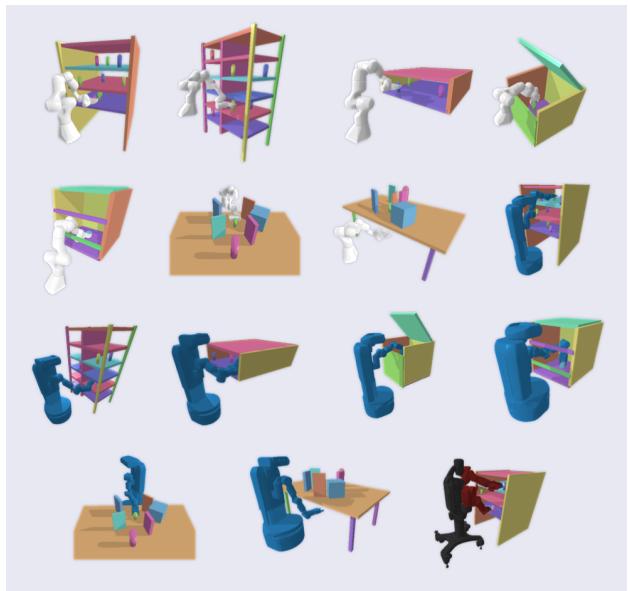


Fig. 2: The MotionBenchMaker scenes provide diverse manipulation problems for 7, 8, and 14 DoF robots. The problems include counter top manipulation, accessing shelves, and constrained reaching. Panda and Fetch have 7 scenes with 100 problems each, while Baxter has 3 scenes with 500 problems each.

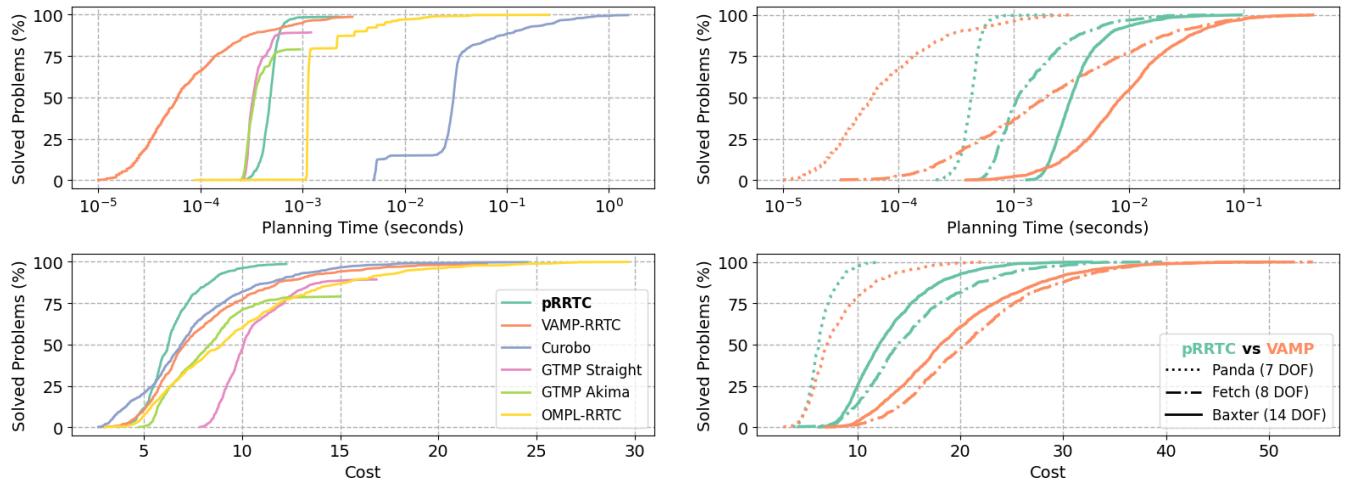
IV. EXPERIMENTS

A. Methodology

We evaluate pRRTc against state-of-the-art CPU and GPU baselines. On the CPU we use two CPU-based RRT-Connect implementations: the RRT-Connect provided by the Open Motion Planning Library (OMPL-RRTc) [60] and the implementation provided by VAMP (VAMP-RRTc) [9]. To ensure a fair evaluation, OMPL-RRTc was built with VAMP as the motion validation backend, as VAMP has shown orders of magnitude speedup compared to the default setup of OMPL. We choose these implementations because they are known to be the best performing planners on the MotionBenchMaker dataset in terms of planning time and solution cost [3, 9, 61]. We also compare against two state-of-the-art GPU-based planners: Curobo [43] and Global Tensor Motion Planning (GTMP) [62] with straight-line interpolation (GTMP Straight) and Akima splines (GTMP Akima). All experiments were conducted on an x86-based desktop computer with an AMD Ryzen Threadripper PRO 5965WX 24-Core CPU and an NVIDIA GeForce RTX 4090 GPU. We leverage all parallelism available in these planners whether on the CPU or GPU. Hyperparameters are controlled across planners to ensure equivalent implementations with identical collision checking resolution and motion extension range. Planner-specific hyperparameters are chosen to maximize percentage of problems solved and to minimize planning time. We measure the planning time of all algorithms excluding the environmental setup step for both the CPU and GPU to maintain a fair and consistent comparison. We use identical multi-dimensional Halton sequences for configuration sampling across pRRTc, VAMP-RRTc, and OMPL-RRTc, and default samplers for Curobo and GTMP.

System	Planning Time (ms)						Solution Cost (arclength)						Succ.
	Mean	Q1	Med.	Q3	95%	100%	Mean	Q1	Med.	Q3	95%		
Panda	Curobo	62.63	25.92	30.08	34.38	281.63	1537.75	7.52	5.45	6.89	8.96	14.09	100.00%
	OMPL-RRT	2.54	1.11	1.13	1.17	6.42	257.99	9.87	6.36	8.98	11.87	18.48	100.00%
	GTMP Akima	0.34	0.29	0.34	0.52	—	—	7.71	6.41	8.35	10.90	—	79.11%
	GTMP Straight	0.35	0.30	0.33	0.43	—	—	10.29	9.22	10.07	12.12	—	89.41%
	VAMP-RRT	0.17	0.03	0.06	0.13	0.81	3.08	8.05	5.80	7.04	9.46	14.84	100.00%
	pRRT	0.49	0.42	0.48	0.53	0.65	2.32	6.46	5.50	6.23	7.17	9.17	100.00%
Fetch	VAMP-RRT	11.93	0.52	1.90	8.52	56.48	368.69	21.34	16.20	20.49	25.19	34.54	100.00%
	pRRT	2.32	0.85	1.15	2.18	6.90	56.67	15.26	11.28	13.88	18.00	25.82	100.00%
	VAMP-RRT	18.53	4.47	8.67	18.25	66.53	398.39	19.47	14.44	18.05	23.32	32.75	100.00%
Baxter	pRRT	4.82	2.44	3.23	4.59	12.49	95.46	13.12	10.00	12.12	15.22	21.57	100.00%

TABLE I: Results for all robots and planners. Planning times shown in **milliseconds**, cost shown in units of configuration space.



(a) MotionBenchMaker planning time and cost on 7 DoF Panda. All times are shown on a **logarithmic** scale. pRRT achieves lower cost initial solutions while performing on the order or better than state of the art on the most difficult problems. pRRT is the first planner to solve all problems, and only pRRT and VAMP-RRT solve both sub-millisecond planning time and 100% solve rate.

(b) MotionBenchMaker planning time and cost across all robots: the 7 DoF Panda, 8 DoF Fetch and 14 DoF Baxter. All times are shown on a **logarithmic** scale. As DoF and problem complexity grows, pRRT performs better relative to VAMP-RRT due to the speedups achieved via parallelism both in terms of average planning time as well as average final path cost.

Fig. 3: Empirical Cumulative Distribution Functions (ECDFs) for all Software Benchmarks.

We evaluated the planners on a set of realistic, challenging problems provided by the MotionBenchMaker dataset [14]. The robots used were the 7 DoF Franka Emika Panda, 8 DoF Fetch, and 14 DoF Rethink Robotics Baxter. We compare all five planners on Panda. We also compare pRRT and VAMP-RRT on Fetch and Baxter. Curobo and GTMP are omitted from the Fetch and Baxter experiments as they do not support these robots. OMPL-RRT is also omitted as it has been shown to be orders of magnitude slower than VAMP-RRT on MotionBenchMaker [9] and on our initial Franka evaluation. For Panda and Fetch, MotionBenchMaker incorporates a diverse set of seven environments each with 100 problems which test the planner’s ability to operate on table surfaces, reach into varying positions of bookshelves, and reach in highly constrained environments. For Baxter the dataset includes three bookshelf reaching scenes each with 500 problems. These benchmarks align with those used in [9, 62]. Visualizations of the environments are shown in Figure 2. Videos of our experiments are available online⁵.

B. Software Benchmarks

A summary of all results is shown in Table I. We present success rate versus performance curves of planning time and solution cost across all environments for the 7 DoF Panda robot using all baselines in Fig. 3a. We also present the comparison of pRRT vs. VAMP-RRT across all environments and robots in Fig. 3b.

On the 7 DoF Panda problems (Fig. 3a), pRRT is on average 128× faster than Curobo, 5× faster than OMPL-RRT, 3× slower than VAMP-RRT, and 1.4× slower than GTMP. Compared to GTMP, pRRT achieves a 100% solve rate, making pRRT and VAMP the only planners that solved all problems while averaging sub-millisecond planning time. In addition, pRRT has the best worst case performance, that is, it solves all problems in the fastest amount of time, and produces the lowest average cost paths.

As we scale to larger DoF problems (Fig. 3b), pRRT outperforms VAMP-RRT on average planning time, consistency, and initial path cost. On average, for the 8 DoF Fetch, pRRT offers 5× speedup compared to VAMP-RRT. We also present the distribution of results per-environment

⁵<https://youtu.be/okpqftLB6P8>

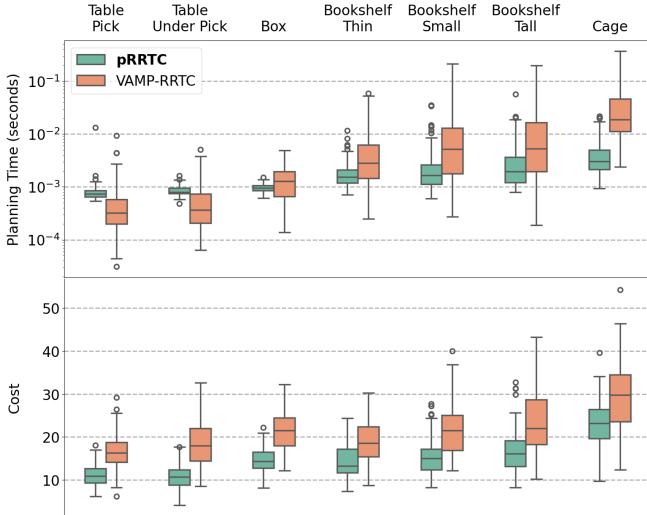


Fig. 4: MotionBenchMaker planning time and cost by problem on Fetch (8 DoF). Problems are ordered from left to right by increasing complexity. All times are shown on a **logarithmic** scale. pRRTTC achieves a 5× speedup in average planning time, 10× speedup on the computationally challenging *Cage* problems, and 8.4× decrease in planning time standard deviation.

in Fig. 4 and observe that the benefit of pRRTTC grows in more constrained environments. For example, on the *Cage* problem pRRTTC provides a 10× improvement on average planning time. pRRTTC also has an 8.4× decrease in the standard deviation of planning time compared to VAMP-RRTTC across all environments. This demonstrates pRRTTC is not only a faster planner on average, but it also offers much more reliable performance. The advantage of pRRTTC’s parallel design also extends to path cost. On average, pRRTTC produces initial paths with a 1.4× decrease in cost compared to VAMP-RRTTC while showing a 1.3× decrease in the standard deviation of cost. Thus, pRRTTC consistently finds lower cost initial paths.

Finally, for the 14 DoF Baxter, we see a similar trend of performance with pRRTTC providing a 3.9× average planning time speedup over VAMP-RRTTC and achieving a faster planning time on 93% of problems.

C. Ablation Studies

Fig. 5 uses the 14 DoF Baxter for an ablation analysis of key design choices and optimizations we implement in pRRTTC. We find that our design choices all individually contribute to our overall speedup in planning time. The most significant gain came from checking discretized motions in parallel rather than sequentially, which resulted in an average 9.4× speedup. Replacing CPU-based FK and collision checking routines (e.g., those used in VAMP [9]) with customized GPU-optimized kernels for pRRTTC provided a 3.3× average speedup. Using four threads per configuration to perform FK and CC, rather than a single thread, improved performance by an average of 1.8×. Incorporating dynamic domain sampling [18] contributed an average 1.3× gain. Enabling early termination of the CC routine upon detecting a collision led to an average 1.1× speedup. Finally, the combined impact

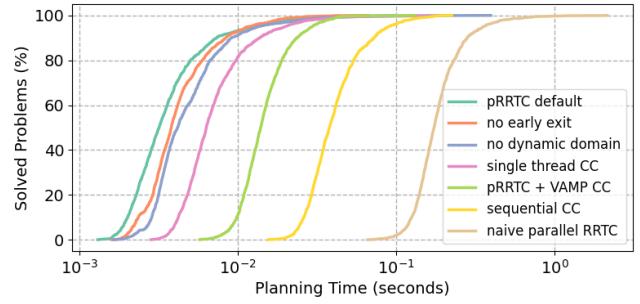


Fig. 5: MotionBenchMaker planning time across different ablations of pRRTTC on Baxter (14 DoF). All times are shown on a **logarithmic** scale. Each one of the design choices individually improved the efficiency of pRRTTC, with their combined effects achieving a 41× speedup over a naive parallel baseline.

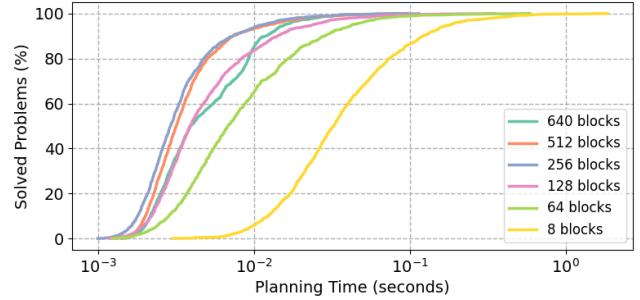


Fig. 6: MotionBenchMaker planning time across varying block level parallelism of pRRTTC on Baxter (14 DoF). All times are shown on a **logarithmic** scale. The planner benefits from increased parallelism until saturation around 256 blocks, after which block parallelism overhead led to suboptimal planning time.

of these optimizations is an average speedup of 41× over a naive baseline, showing how important careful co-design is for GPU-accelerated performance.

In addition to exploiting parallelism for FK and CC subroutines, the general block level parallelism of concurrent sampling and expansion also contributes to the efficiency of pRRTTC. As shown in Fig. 6, we see the planning time decreases as the number of blocks rise from 8 to 256, with a 14× speedup attributed to using 256 concurrent blocks rather than 8. As the block count increase beyond 256, the system becomes saturated, with the overhead of managing block saturation leading to worse planning time. This again shows that careful hardware-software co-design is needed for maximal algorithmic performance.

D. Hardware Validation

In order to test the feasibility of pRRTTC in real-time planning scenarios, we deployed it on dual 7 DoF Franka Panda arms, resulting in a total of 14 DoF. Our setup consists of the arms mounted on a platform, with an overhead Intel Realsense camera streaming depth images at 90 fps. We use Nvblox [58] to continuously integrate depth images into a Euclidean Signed Distance Field (ESDF). For real-time planning, we adapt our collision checking to query the ESDF rather than check against geometric primitives. To test real-

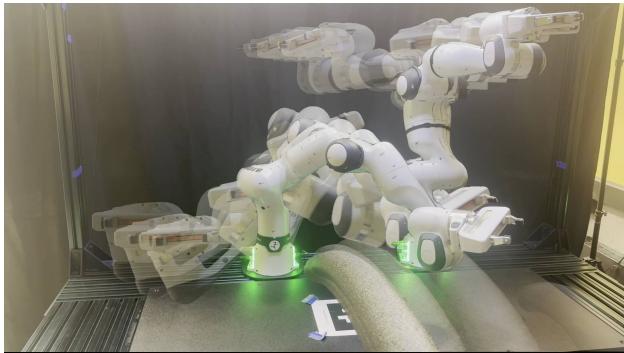


Fig. 7: pRRT_C achieves real-time replanning on a 14-DoF dual-arm Franka Panda setup, enabling dynamic obstacle avoidance. Shown here is a superimposed image from several frames of our supplementary video, illustrating the robot arms successfully avoiding moving pool noodles.

time replanning we have the robots sweep back and forth while obstructing their paths with dynamic obstacles. In our scenario, pRRT_C is able to achieve real-time replanning, which, when paired with the perception routines, is able to achieve an end-to-end pipeline frequency of 7.7 Hz. The complete hardware demonstration can be viewed in the supplemental video submission.

V. CONCLUSION AND FUTURE WORK

In this work, we present pRRT_C, a GPU-based parallel RRT-Connect algorithm coupled with a SIMT-optimized parallel collision checker. pRRT_C utilizes both low- and medium-level GPU parallelism, parallelizing both low-level primitives (collision checking and nearest neighbors) and expansion of the tree while introducing almost no additional computational or communication overhead. Compared to both CPU- and GPU-based state-of-the-art motion planners on the MotionBenchMaker dataset [14] using robots with 7, 8, and 14 degrees of freedom, pRRT_C provides as much as a 10× speedup and 5.4× reduction in planning time standard deviation while achieving a 1.4× reduction in average initial path cost compared to existing approaches. We deploy pRRT_C onto dual 7 DoF Franka Panda arms and demonstrate real-time, collision-free motion planning with dynamic obstacles.

Our future work includes transforming pRRT_C into an almost-surely asymptotically optimal sampling-based planner [63], as the parallelization of framework iteration, collision checking, and nearest neighbor search translates naturally into the path optimization setting, particularly for planners that benefit from fast edge validation [61]. We are also interested in leveraging real-time re-compilation to enable more dynamic co-designed optimizations [64] and aim to further improve our full end-to-end pipeline for additional real-world demonstrations in future work.

REFERENCES

- [1] S. M. LaValle. *Planning Algorithms*. Cambridge university press, 2006.
- [2] L. E. Kavraki and S. M. LaValle. “Motion Planning”. In: *Springer Handbook of Robotics*. Springer, 2016, pp. 139–162.
- [3] A. Orthey, C. Chamzas, and L. E. Kavraki. “Sampling-based motion planning: A comparative review”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 7 (2023).
- [4] S. M. LaValle and J. J. Kuffner. “Rapidly-Exploring Random Trees: Progress and Prospects”. In: *Algorithmic and computational robotics: new directions* 5 (2001), pp. 303–307.
- [5] J. J. Kuffner and S. M. LaValle. “RRT-Connect: An Efficient Approach to Single-Query Path Planning”. In: *IEEE International Conference on Robotics and Automation*. Vol. 2, 2000, pp. 995–1001.
- [6] M. Otte and N. Correll. “C-FOREST: Parallel Shortest Path Planning with Superlinear Speedup”. In: *Transactions on Robotics* 29.3 (2013), pp. 798–806.
- [7] A. Hidalgo-Paniagua, J. P. Bandera, M. Ruiz-de-Quintanilla, and A. Bandera. “Quad-RRT: A Real-Time GPU-Based Global Path Planner in Large-Scale Real Environments”. In: *Expert Systems with Applications* 99 (2018), pp. 141–154.
- [8] J. Ichnowski and R. Alterovitz. “Parallel Sampling-Based Motion Planning with Superlinear Speedup”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 1206–1212.
- [9] W. Thomason, Z. Kingston, and L. E. Kavraki. “Motions in Microseconds via Vectorized Sampling-Based Planning”. In: *IEEE International Conference on Robotics and Automation*. 2024, pp. 8749–8756.
- [10] J. Bialkowski, S. Karaman, and E. Frazzoli. “Massively Parallelizing the RRT and the RRT”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2011, pp. 3513–3518.
- [11] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. D. Konidaris. “Robot Motion Planning on a Chip”. In: *Robotics: Science and Systems*. Vol. 6. Ann Arbor, Michigan, June 2016.
- [12] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin. “The Microarchitecture of a Real-Time Robot Motion Planning Accelerator”. In: *IEEE/ACM International Symposium on Microarchitecture*. 2016, pp. 1–12.
- [13] B. Ichter, E. Schmerling, and M. Pavone. “Group Marching Tree: Sampling-Based Approximately Optimal Motion Planning on GPUs”. In: *IEEE International Conference on Robotic Computing*. 2017, pp. 219–226.
- [14] C. Chamzas, C. Quintero-Peña, Z. Kingston, A. Orthey, D. Rakita, M. Gleicher, M. Toussaint, and L. E. Kavraki. “MotionBenchMaker: A Tool to Generate and Benchmark Motion Planning Datasets”. In: *IEEE Robotics and Automation Letters* 7.2 (2021), pp. 882–889.
- [15] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT press, 2005.
- [16] W. Wang, X. Xu, Y. Li, J. Song, and H. He. “Triple RRTs: An Effective Method for Path Planning in Narrow Passages”. In: *Advanced Robotics* 24.7 (2010), pp. 943–962.
- [17] W. Wang, Y. Li, X. Xu, and S. X. Yang. “An Adaptive Roadmap Guided Multi-RRTs Strategy for Single Query Path Planning”. In: *IEEE International Conference on Robotics and Automation*. 2010, pp. 2871–2876.
- [18] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle. “Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain”. In: *IEEE International Conference on Robotics and Automation*. 2005, pp. 3856–3861.
- [19] C. Urmson and R. Simmons. “Approaches for Heuristically Biasing RRT Growth”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 2, 2003, pp. 1178–1183.
- [20] Z. Liu, F. Lan, and H. Yang. “Partition Heuristic RRT Algorithm of Path Planning Based on Q-Learning”. In: *IEEE Advanced Information Technology, Electronic and Automation Control Conference*. Vol. 1, 2019, pp. 386–392.
- [21] S. Rodriguez, X. Tang, J.-M. Lien, and N. M. Amato. “An Obstacle-Based Rapidly-Exploring Random Tree”. In: *IEEE International Conference on Robotics and Automation*. 2006, pp. 895–900.
- [22] X. Shu, F. Ni, Z. Zhou, Y. Liu, H. Liu, and T. Zou. “Locally Guided Multiple Bi-RRT* for Fast Path Planning in Narrow Passages”. In: *IEEE International Conference on Robotics and Biomimetics*. 2019, pp. 2085–2091.
- [23] J. Kuffner and S. LaValle. “An Efficient Approach to Path Planning Using Balanced Bidirectional RRT Search”. In: *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep* (2005).
- [24] T.-Y. Li and J.-S. Chen. “Incremental 3D Collision Detection with Hierarchical Data Structures”. In: *ACM Symposium on Virtual Reality Software and Technology*. 1998, pp. 139–144.
- [25] G. Sánchez and J.-C. Latombe. “A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking”. In: *International Symposium on Robotics Research*. 2003, pp. 403–417.

- [26] G. Sánchez and J.-C. Latombe. “On Delaying Collision Checking in PRM Planning: Application to Multi-Robot Coordination”. In: *The International Journal of Robotics Research* 21.1 (2002), pp. 5–26.
- [27] K. Hauser. “Lazy Collision Checking in Asymptotically-Optimal Motion Planning”. In: *IEEE International Conference on Robotics and Automation*. 2015, pp. 2951–2957.
- [28] F. Schwarzer, M. Saha, and J.-C. Latombe. “Exact Collision Checking of Robot Paths”. In: *Algorithmic Foundations of Robotics* (2004), pp. 25–41.
- [29] D. Joho, J. Schwinn, and K. Safronov. “Neural Implicit Swept Volume Models for Fast Collision Detection”. In: *IEEE International Conference on Robotics and Automation*. IEEE. 2024, pp. 15402–15408.
- [30] D. Son, H. Jung, and B. Kim. “NeuralSVCD for Efficient Swept Volume Collision Detection”. In: *arXiv preprint arXiv:2509.00499* (2025).
- [31] J. Bialkowski, S. Karaman, M. Otte, and E. Frazzoli. “Efficient Collision Checking in Sampling-Based Motion Planning”. In: *Algorithmic Foundations of Robotics*. 2013, pp. 365–380.
- [32] J. Barraquand and J.-C. Latombe. “A Monte-Carlo Algorithm for Path Planning with Many Degrees of Freedom”. In: *IEEE International Conference on Robotics and Automation*. 1990, pp. 1712–1717.
- [33] D. Henrich. “Fast Motion Planning by Parallel Processing—A Review”. In: *Journal of Intelligent and Robotic Systems* 20 (1997), pp. 45–69.
- [34] N. M. Amato and L. K. Dale. “Probabilistic Roadmap Methods Are Embarrassingly Parallel”. In: *IEEE International Conference on Robotics and Automation*. Vol. 1. May 1999, pp. 688–694.
- [35] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces”. In: *Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.
- [36] S. Xiao, N. Bergmann, and A. Postula. “Parallel RRT* Architecture Design for Motion Planning”. In: *International Conference on Field Programmable Logic and Applications*. 2017, pp. 1–4.
- [37] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki. “Sampling-Based Roadmap of Trees for Parallel Motion Planning”. In: *Transactions on Robotics* 21.4 (2005), pp. 597–608.
- [38] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato. “A Scalable Method for Parallelizing Sampling-Based Motion Planning Algorithms”. In: *IEEE International Conference on Robotics and Automation*. 2012, pp. 2529–2536.
- [39] C. W. Ramsey, Z. Kingston, W. Thomason, and L. E. Kavraki. “Collision-Affording Point Trees: SIMD-Amenable Nearest Neighbors for Fast Collision Checking”. In: *Robotics: Science and Systems*. 2024.
- [40] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato. “A Scalable Distributed RRT for Motion Planning”. In: *IEEE International Conference on Robotics and Automation*. 2013, pp. 5088–5095.
- [41] G. Tran Thi Cam, D. M. Do, Q. H. Do, T. T. B. Huynh, and T. H. L. Dinh. “GPU-Based Parallel Path Planning for Mobile Robot Navigation in Dynamic Environments”. In: *International Symposium on Information and Communication Technology*. 2023, pp. 878–885.
- [42] N. Perrault, Q. H. Ho, and M. Lahijanian. “Kino-PAX: Highly Parallel Kinodynamic Sampling-Based Planner”. In: *arXiv preprint arXiv:2409.06807* (2024).
- [43] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. Van Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, et al. “Curobo: Parallelized Collision-Free Robot Motion Generation”. In: *IEEE International Conference on Robotics and Automation*. 2023, pp. 8112–8119.
- [44] J. Pan and D. Manocha. “GPU-Based Parallel Collision Detection for Fast Motion Planning”. In: *International Journal of Robotics Research* 31.2 (2012), pp. 187–200.
- [45] S. Pabst, A. Koch, and W. Straßer. “Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces”. In: *Computer Graphics Forum*. Vol. 29. 5. 2010, pp. 1605–1612.
- [46] C. Lauterbach, Q. Mo, and D. Manocha. “gProximity: Hierarchical GPU-Based Operations for Collision and Distance Queries”. In: *Computer Graphics Forum*. Vol. 29. 2. 2010, pp. 419–428.
- [47] Y. Zhou and J. Zeng. “Massively Parallel A* Search on a GPU”. In: 29.1 () . p. 7.
- [48] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto. “A Survey of Parallel A*”. 2017.
- [49] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto. “Parallel A* for State-Space Search”. In: *Handbook of Parallel Constraint Reasoning*. Ed. by Y. Hamadi and L. Sais. Springer International Publishing, 2018, pp. 419–455.
- [50] G. Williams, A. Aldrich, and E. A. Theodorou. “Model Predictive Path Integral Control: From Theory to Parallel Computation”. In: *Journal of Guidance, Control, and Dynamics* 40.2 (2017), pp. 344–357.
- [51] B. Plancher and S. Kuindersma. “A Performance Analysis of Parallel Differential Dynamic Programming on a GPU”. In: *International Workshop on the Algorithmic Foundations of Robotics*. Merida, Mexico, 2018, pp. 656–672.
- [52] E. Adabag, M. Atal, W. Gerard, and B. Plancher. “MPGPU: Real-Time Nonlinear Model Predictive Control Through Preconditioned Conjugate Gradient on the GPU”. In: *IEEE International Conference on Robotics and Automation*. Yokohama, Japan, 2024.
- [53] S. H. Jeon, S. Hong, H. J. Lee, C. Khazoom, and S. Kim. “Cusadi: A GPU Parallelization Framework for Symbolic Expressions and Optimal Control”. In: *IEEE Robotics and Automation Letters* (2024).
- [54] G. M. Chari, A. G. Kamath, P. Elango, and B. Acikmese. “Fast Monte Carlo Analysis for 6-DoF Powered-Descent Guidance via GPU-Accelerated Sequential Convex Programming”. In: *AIAA SciTech Forum*. 2024, p. 1762.
- [55] G. Sanchez and J.-C. Latombe. “Using a PRM Planner to Compare Centralized and Decoupled Planning for Multi-Robot Systems”. In: *IEEE International Conference on Robotics and Automation*. Vol. 2. 2002, pp. 2112–2119.
- [56] NVIDIA. *NVIDIA CUDA C++ Programming Guide*. Version 13.0. Sept. 2025.
- [57] S. Coumar, G. Chang, N. Kodkani, and Z. Kingston. “Foam: A Tool for Spherical Approximation of Robot Geometry”. In: *arXiv preprint arXiv:2503.13704* (2025).
- [58] A. Millane, H. Oleynikova, E. Wirbel, R. Steiner, V. Ramasamy, D. Tingdahl, and R. Siegwart. “nvblox: GPU-Accelerated Incremental Signed Distance Field Mapping”. In: *IEEE International Conference on Robotics and Automation*. 2024, pp. 2698–2705.
- [59] M. Harris et al. “Optimizing Parallel Reduction in CUDA”. In: *NVIDIA Developer Technology* 2.4 (2007), p. 70.
- [60] I. A. Şucan, M. Moll, and L. E. Kavraki. “The Open Motion Planning Library”. In: *IEEE Robotics and Automation Magazine* 19.4 (2012). <https://ompl.kavrakilab.org>, pp. 72–82.
- [61] T. S. Wilson, W. Thomason, Z. Kingston, L. E. Kavraki, and J. D. Gammell. “Nearest-Neighbourless Asymptotically Optimal Motion Planning with Fully Connected Informed Trees (FCIT*)”. In: *arXiv preprint arXiv:2411.17902* (2024).
- [62] A. T. Le, K. Hansel, J. Carvalho, J. Watson, J. Uraian, A. Biess, G. Chalvatzaki, and J. Peters. “Global Tensor Motion Planning”. In: *arXiv preprint arXiv:2411.19393* (2024).
- [63] J. D. Gammell and M. P. Strub. “Asymptotically Optimal Sampling-Based Motion Planning Methods”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 4.1 (2021), pp. 295–318.
- [64] J. Hu, J. Wang, and H. Christensen. *cpRRTC: GPU-Parallel RRT-Connect for Constrained Motion Planning*. 2025.