2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

GPU Parallelization of Policy Iteration RRT#

R. Connor Lawson* Linda Wills* Panagiotis Tsiotras†

***Abstract*— Sampling-based planning has become a *de facto* standard for complex robots given its superior ability to rapidly explore high-dimensional configuration spaces. Most existing optimal sampling-based planning algorithms are sequential in nature and cannot take advantage of wide parallelism available on modern computer hardware. Further, tight synchronization of exploration and exploitation phases in these algorithms limits sample throughput and planner performance. Policy Iteration RRT# (PI-RRT#) exposes fine-grained parallelism during the exploitation phase, but this parallelism has not yet been evaluated using a concrete implementation. We first present a novel GPU implementation of PI-RRT#'s exploitation phase and discuss data structure considerations to maximize parallel performance. Our implementation achieves 3–4× speedup over a serial PI-RRT# implementation for a 77.9% decrease in overall planning time on average. As a second contribution, we introduce the Batched-Extension RRT# algorithm, which loosens the synchronization present in PI-RRT# to realize independent 12.97× and 12.54× speedups under serial and parallel exploitation, respectively.**

I. Introduction

Motion planning is foundational to robotics, and the subject has received much study over several decades of research. Sampling-based planning [1]–[6] has become the *de facto* choice for complex robots given its superior ability to rapidly explore high-dimensional configuration spaces. Yet, the problem is far from solved, and new techniques continue to be developed.

As path planning algorithms improve, their applications become more varied and demanding. Recently, the so-called Task And Motion Planning (TAMP, e.g. [7]) seeks to marry geometric path planning and semantic constraint satisfaction, requiring very fast geometric planning on novel scenarios as a subroutine to more sophisticated algorithms. Other extensions such as kinodynamic planning recast planning with differential constraints as a geometric planning problem in a higher dimensional space via projection onto the constraint manifold. As the cost of planning in general grows exponentially with dimension, faster algorithms enable planning for more realistic dynamical systems.

Since Karaman and Frazzoli's seminal paper in 2011 [3], focus on sampling-based planning has turned to finding not merely feasible paths, but optimal ones. Planners seek shortest paths in some sense – typically Euclidean distance, either in configuration space or in some projection thereof. Optimizing the solution can be seen as an "exploitation" phase utilizing connectivity information gained during sample generation and graph extension, the "exploration" phase [8].

Existing single-query sampling-based planning algorithms rely on tightly interleaved exploration and exploitation, using the result of one phase to inform the next in an iterative fashion. Many proofs of correctness and optimality depend intimately on this ordering with precise synchronization. For example, the optimal Rapidly Exploring Random Tree (RRT*) algorithm requires sequential sampling and extension to achieve Voronoi-biased "rapid" exploration of the configuration space, but rewires the solution tree after each new sample to achieve asymptotic optimality.

Synchronization of exploration and exploitation imposes a bound on parallelism of planning algorithms. Probabilistic completeness and asymptotic optimality hold only as the number of samples approaches infinity. Exploitation "in the loop" requires more work per iteration, reducing sample throughput. Excellent prior work, such as [9], [10], has focused on parallelizing and accelerating expensive subproblems of path planning – most commonly collision detection and nearest neighbor search – but these approaches sidestep the problem of synchronized exploitation, which ultimately limits the available parallelism.

Arslan and Tsiotras [11] introduce new opportunities for parallel exploitation. Their PI-RRT# planner exposes fine-grained parallelism within shortest-path search through a policy iteration method. However, this algorithm still retains tight synchronization and contains only theoretical predictions of parallel performance.

This work builds on [11] to move towards desynchronizing exploration and exploitation, and to evaluate parallel performance of PI-RRT# with a concrete implementation. We present a novel GPU implementation of PI-RRT# and compare its performance to a serial implementation. Additionally, we reduce the tight synchronization between exploitation and exploration in PI-RRT# by introducing batched graph extension and discuss future opportunities for further asynchronous exploitation with respect to exploration.

Our parallel GPU implementation results in 3–4× speedup in exploitation for large problems and an average decrease of up to 77.9% in overall planning time compared to a serial implementation. Batched graph extension, by contrast, achieves up to 12.97× and 12.54× reduction in planning time with serial and parallel implementations, respectively,

This work has been supported by the National Science Foundation under Grant No. DGE-1650044.

*School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332. rconnorlawson@gatech.edu, linda.wills@ece.gatech.edu

†Daniel Guggenheim School of Aerospace Engineering, Institute for Robotics and Intelligent Machines, Georgia Institute of Technology, Atlanta, GA 30332. tsiotras@gatech.edu

when compared to more tightly synchronized exploitation.

Section II discusses related work, including existing parallelized planning algorithms. Section III defines relevant concepts and notation of the PI-RRT# algorithm. Section IV presents a novel GPU implementation for exploitation in PI-RRT#. Section V further modifies the algorithm to support batched graph extension. Section VI evaluates the proposed planners. Section VII discusses future directions before concluding in Section VIII.

II. Related work

Existing sampling-based algorithms are many and varied. They can generally be divided into two classes: single-query and multi-query. Single query algorithms discover a tree of solutions rooted at the initial query point. Because they evaluate connectivity of the free planning space at query time, these algorithms are more adaptable to dynamic environments and very high dimensional spaces where complete coverage is infeasible. The prototypical single-query algorithm is the Rapidly Exploring Random Tree (RRT) [1]. Algorithms in this group find a collision-free path from a single starting point in configuration space to a goal region by building a directed tree of collision-free edges. Notable recent algorithms in this category include: RRT*, the optimal variant [3]; Fast Marching Tree (FMT*) [6], which explores a collection of samples in best-first order according to a selection heuristic; Batch Informed Trees (BIT*) [5], a generalization of FMT* to incorporate informed sampling; and RRT# [12], which uses value iteration-based exploitation to discover the best achievable paths through the current samples. PI-RRT#, as the name implies, is a modification of RRT# that uses policy iteration instead of value iteration as in the original RRT#.

Multi-query algorithms, by contrast, precompute a roadmap of the configuration space and connect pairs of start and goal query points to the map before solving a simple graph search. The separation of roadmap construction and path query allows very fast query-time execution, but requires a large memory footprint and expensive up-front computation. These algorithms are best suited for static environments with small configuration spaces, where thorough coverage by a roadmap is feasible. Algorithms in this category descend from Probabilistic Road Maps (PRM) [13].

There has been some previous work in parallelizing optimal single-query path planning. C-FOREST [14] replicates the path planning procedures and data structures over multiple CPUs, broadcasting solution paths to other processes as they are discovered. P-RRT* [15] uses multiple threads running RRT* on a shared lock-free data structure. Bialkowski *et al.* [9] exposes parallelism by offloading collision detection in RRT* to the GPU, but does not otherwise modify the algorithm. All of these approaches retain the underlying coupling of exploration and exploitation. As such, they are complementary to the proposed approach.

Group Marching Tree (GMT*) [16], akin to both FMT* and the Δ-stepping shortest path algorithm, expands multiple nodes per iteration in fixed-radius waves, similar to the proposed approach, but can only produce near-optimal paths. Further, GMT* operates on a precomputed batch of samples and neighbors, where the current work includes both sampling and neighbor detection at runtime. This design choice retains opportunity for sampling-time optimizations such as Informed Sampling [17] and lazily evaluates nearest neighbors only as necessary.

Algorithm 1: PI-RRT# [11]

```
1 V ← {x_init, x_goal}; E ← ∅; G ← (V, E)
2 B ← ∅
3 for k := 1 to N do
4 |   (G, B', g_π) ← Extend(G, B, g_π)
5 |   if |B'| > |B| then
6 |   |   B ← Replan(G, B', x_init, x_goal)
7 (V, E) ← G; E' ← ∅
8 foreach v ∈ V do
9 |   E' ← E ∪ { π(v) }
```

III. Overview of PI-RRT#

PI-RRT# is an extension of the Rapidly-exploring Random Tree algorithm (RRT) which uses explicit policy iteration to maintain an optimal policy $\pi : V \to E$ and cost-to-come $g_\pi : V \to \mathbb{R}_+$ for each vertex $v \in V$ during construction of the free-space connectivity graph $G = (V, E)$. Each vertex is associated with a state $x \in \mathcal{X}_{free}$ in the obstacle-free configuration space, and the cost of an edge in G is the cost to traverse between the states associated with its endpoints, denoted $c : E \to \mathbb{R}_+$. An admissible heuristic $h : V \to \mathbb{R}_+$ provides a lower bound on the cost-to-go from a given vertex to the goal, and with the cost-to-come g_π is used to track a promising set $B \subseteq V$ of vertices that could potentially be on the optimal path.

A high-level overview of PI-RRT# can be found in Algorithm 1. PI-RRT# performs exploitation in both the `Extend` and `Replan` procedures, but `Replan` is the more expensive and crucial exploitation. The `Extend` procedure extends the graph by random sampling and extension in the same manner as RRT. This is the exploration step. Once a new vertex v is added, the local minimum cost-to-go $g_\pi(v)$ and policy $\pi(v)$ is found by considering edges to each existing vertex in a small neighborhood. This small exploitation determines whether the new vertex is promising, in which case it is added to the set B, triggering a `Replan` before adding the next sample. `Replan` performs policy iteration to convergence in order to propagate updated path information from new vertices and maintain optimal π and g_π over the whole promising set.

As this work parallelizes the `Replan` procedure on the GPU, the serial version is reproduced for reference in Algorithm 2. First, the `Improve` procedure evaluates all neighbors of each promising vertex (denoted $neigh(G, v)$) and selects the locally optimal parent node, updating policy π accordingly. This is the same local relaxation performed on the new vertex during `Extend`. Next, `Evaluate` rebuilds

Algorithm 2: PI-RRT# Policy Iteration

1 **Procedure** `Replan`(G, B, x_{init}, x_{goal})
2 **loop**
3 $\pi, \Delta g \leftarrow$ `Improve`(G, c, B, g_π)
4 **if** $\Delta g < \epsilon$ **then**
5 **break**
6 $g_\pi, B \leftarrow$`Evaluate`(G, c, π, g_π, h)
7 **Procedure** `Improve`(G, c, B, g_π)
8 **foreach** $v \in B$ **do**
9 $\hat{g} \leftarrow g_\pi(v)$; $\Delta g \leftarrow 0$
10 **foreach** $n \in neigh(G, v)$ **do**
11 **if** $c(n, v) + g_\pi(n) < \hat{g}$ **then**
12 $\hat{g} \leftarrow c(n, v) + g_\pi(n)$
13 $\pi(v) \leftarrow n$
14 $\Delta g \leftarrow \max(\Delta g, g_\pi(v) - \hat{g})$
15 **return** $(\pi, \Delta g)$
16 **Procedure** `Evaluate`(G, c, π, g_π, h)
17 $Q \leftarrow \{ x_{init} \}$; $B \leftarrow \emptyset$
18 **while** $|Q| > 0$ **do**
19 $v \leftarrow$ pop(Q)
20 **foreach** $n \in child(\pi, v)$ **do**
21 $g_\pi(n) \leftarrow c(v, n) + g_\pi(v)$
22 **if** $h(v) + g_\pi(v) < g_\pi(x_{goal})$ **then**
23 push(Q, n)
24 $B \leftarrow B \cup \{ n \}$
25 **return** g_π, B

B and computes updated g_π for all promising vertices via a truncated breadth-first traversal of the policy tree. This process is repeated until convergence at tolerance ϵ (Line 4). Additional detail on PI-RRT# can be found in [11].

IV. GPU Implementation

PI-RRT# exposes parallelism during exploitation that is not present in previous approaches. In particular, the Jacobi relaxation in the `Improve` procedure is directly parallelized by processing each vertex in parallel. Additionally, the `Evaluate` procedure is a truncated breadth-first tree traversal, in which each vertex on the frontier can be evaluated in parallel. This section presents a novel GPU implementation of `Replan` to take advantage of this parallelism. Note that this parallel implementation does not change the algorithm, rather it aims to make maximum use of parallelism already available.

A. Data Movement and Data Structure Preparation

As with any GPU algorithm, data structure and memory movement is of primary importance, as low bandwidth data movement between the CPU and GPU is the strictest memory bottleneck. [18] The data in this algorithm can be represented as a graph $G = (V, E)$ with properties on its edges and vertices. The data structures chosen to store these properties must facilitate both efficient bus utilization and convenient memory access patterns. Data such as the world description and the parameters of the cost function c are constant and therefore negligible. Each vertex has the following properties:

1) Cost-to-come $g(v) \in \mathbb{R}_+$
2) Heuristic cost-to-go $h(v) \in \mathbb{R}_+$
3) Parent $p \in \{V \cup \emptyset\}$, s.t. $\pi(v) = (p, v)$
4) Promising $b \in \{0, 1\}$, s.t. $b = 1 \iff v \in B$

All vertex properties are stored in structure of arrays format. Several of these data (g, p, and b) are transferred to and from and GPU at each `Replan`, so performant transfer is highest priority. Structure of arrays format stores each property as a contiguous buffer, facilitating constant time access and high bandwidth transfer.

The edges have only a single property: the cost of traversal c. This cost is evaluated once for each edge during `Extend` in order to evaluate the locally optimal g_π and π, and its value is cached for use during `Replan`. This removes the need to copy potentially high-dimensional state data to the GPU, further reducing data transfer overhead.

Storage for the edges and edge properties requires careful consideration, as the graph structure chosen will impact both the communication and computation time of the algorithm. Three graph structures were considered:

1) Adjacency List Graph: This structure has favorable memory access patterns for vertex expansion and graph extension, as costs and outgoing edges for each vertex are stored in dynamic arrays which provide (amortized) constant-time access. This is the format used for graph representation in the CPU benchmark implementation (See Section VI). However, synchronizing adjacency lists between the CPU and GPU is prohibitively expensive. While edge lists for each vertex are contiguous in memory, the collection of lists is not, as each list must be able to grow dynamically. As a result, synchronization requires many small memory transfers, incurring a significant bandwidth penalty.

2) Coordinate List Graph: This structure requires only a few large transfers to synchronize, leading to high bandwidth utilization, but incurs some overhead due to storing the source node of each edge redundantly. Compared to adjacency lists, vertex expansion in this format is much more expensive, as edges are not necessarily contiguous per vertex. Without sorting or other additional structure, vertex expansion requires searching the entire structure for edges with the desired source node.

3) Compressed Sparse Row (CSR) Graph: This structure has the advantage of contiguous edges per vertex, as in the Adjacency List graph, and does not suffer the space penalty of Coordinate lists. However, edges are not easily added to existing vertices in CSR. In general, adding new edges requires a rebuild step that is linear in the size of the graph.

Consideration of data flow yields helpful insight into data structure selection. There exists a producer-consumer relationship between the `Extend` and `Replan` procedures for edges and costs. New edges and costs are generated but unused in `Extend`, and received but unmodified in `Replan`.

Given the one-way dataflow, the best possible scenario for graph data is to transfer each edge to the GPU exactly once. Therefore, the CPU-side data structure can be optimized

strictly for write performance and fast memory transfer. We found that the Coordinate List format on the CPU best supports this use case, as new edges can be appended unordered to the list. Additionally, since any new vertex will have at most one edge to each neighbor, Coordinate List format is more space efficient than CSR in this case.

For fast vertex expansion during the `Improve` kernel, data is converted to CSR format after transfer to the GPU. The new edges are sorted in linear time using radix sort, and a parallel merge over the old and new edge-lists rebuilds the CSR. This rebuild occurs once per `Replan`, so the cost is amortized over many vertex expansions.

B. Policy Improvement Kernel

During policy improvement, each promising vertex searches over its outgoing edges for the least cost path using cost-to-go estimates for its neighbors based on the current policy. In this kernel, each CUDA thread is assigned a single vertex in the promising set, as suggested in the original PI-RRT# paper [11]. Each thread searches all outgoing edges from its assigned vertex for the lowest cost parent. This option suffers from load imbalance since each vertex may have a variable number of neighbors, causing threads to terminate at different times. A load-balanced implementation based on the reduce-by-key primitive from the Thrust library [19], however, resulted in slower execution than the unbalanced version in practice.

C. Policy Evaluation Kernel

`Improve` may change the parent of each promising vertex based on its local neighborhood, but it does not propagate cost reductions caused by upstream rewiring to the leaves of the policy tree. To correct this inconsistency, `Evaluate` walks the policy tree outward in a breadth-first fashion, computing a globally consistent cost-to-come for every visited vertex. Along the way, the promising set is rebuilt using the newly updated cost-to-come, and non-promising branches are pruned from the traversal. Only promising vertices and their neighbors are re-evaluated, since only these vertices are relevant to finding an optimal path.

Many GPU implementations of Breadth First Search already exist. State-of-the-art algorithms, such as [20], use a single GPU kernel launch to process a whole frontier of vertices in parallel. Our implementation follows this approach, but takes advantage of the fact that π is a simple tree, rather than a general graph. Because of this added structure, our implementation does not require notions of open and closed sets typical of BFS, reducing GPU global memory pressure and increasing performance. Additionally, each outgoing edge from any frontier will always lead to a unique vertex.

V. Batch-Extension PI-RRT#

For our second contribution, we offer a key insight, namely, that tightly coupled exploration and exploitation leads to decreased sample throughput and therefore decreased planning performance. This is due to the high cost of tightly synchronized exploitation in the loop. In the example of PI-RRT#, policy iteration may require, in the worst case, a number of iterations two greater than the length of the optimal path in edges [11, Theorem 2].

Algorithm 3: Batched-Extension PI-RRT#

```
1  V ← {x_init, x_goal}
2  E ← ∅
3  G ← (V, E)
4  B ← ∅
5  for k := 1 to N/S do
6  |   B' ← B
7  |   repeat S times
8  |   |   (G, B') ← Extend(G, B', x_init, x_goal)
9  |   if |B'| > |B| then
10 |   |   B ← Replan(G, B', x_init, x_goal)
11 (V, E) ← G; E' ← ∅
12 foreach v ∈ V do
13 |   E' ← E ∪ { p_v }
```

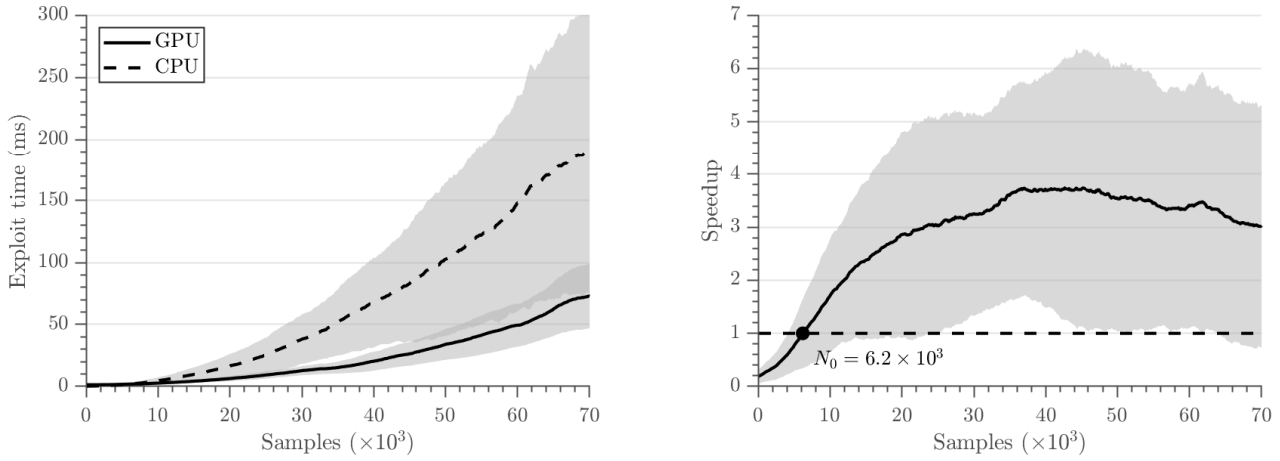
To address this issue, we introduce Batched-Extension PI-RRT# (BERRT#, Algorithm 3). This algorithm utilizes a batch size parameter, S, which delays the expensive `Replan` operation until S samples have been added th the graph (Lines 7–8). Allowing the user to select a batch size exposes a tunable balance between exploration and exploitation. In the case $S = 1$, BERRT# reduces to PI-RRT#. When $S = N$, BERRT# is a variant of PRM*, computing all samples and neighbors first followed by a single shortest path query.

Correctness: Arslan and Tsiotras [11], in their proof that `Replan` results in an asymptotically optimal policy over the set of promising vertices, consider G as a whole and an arbitrary initial policy. Thus the same theorem implies correctness of BERRT#, which differs from PI-RRT# only in the number of samples per iteration and thus in the quality of its initial policies.

Indeed, because BERRT# does not immediately propagate path improvements discovered during `Extend` to the remainder of the promising set, it is possible that within the `Extend` loop policy π and cost-to-come g_π are temporarily suboptimal. However, ensuring that `Replan` runs on the complete graph after all desired samples have been added recovers a fully optimal solution.

VI. Numerical Evaluation

In this section, we analyze the performance of our GPU-based `Replan` against a baseline CPU implementation using an Adjacency List graph. Further, we investigate the impact of synchronized exploitation on planner performance by varying the batch size in BERRT# for cases of both serial and parallel exploitation. Experiments were performed on a Intel Core i7-6820HQ 2.7 GHz CPU running Ubuntu 16.04. GPU results were realized on an NVIDIA Quadro M2000M mobile GPU with CUDA v10.2. Trials used independent random sample sequences on a query for a two-dimensional system with polygonal obstacles. Each test was run for five



(a) Performance

(b) Speedup

Fig. 1: Performance comparison of parallel (GPU) and serial (CPU) exploitation. GPU exploitation outperforms CPU implementation above N_0, marked with a star.

trials, and averaged results are reported. Where confidence intervals are shown, they reflect one standard deviation from the reported mean.

First, we compare the performance of parallel exploitation to serial exploitation. Figure 1a shows the time spent for exploitation as problem size increases for both implementations. Given that the GPU implementation incurs overhead due to memory movement and data structure reconstruction as discussed in Section IV, it is unsurprising that the CPU-based implementation is faster on small problems. Above the labelled threshold N_0, the superior performance of the GPU algorithm is able to overcome this overhead. Figure 1b reports the relative speedup of the GPU exploitation per `Replan`. The peak mean speedup is 3.74× at $N = 42,600$, before tapering back to 3× as the problem size increases. More study is needed to understand this reduction in relative performance on very large problems, but it can likely be attributed to the growing cost of data movement and an increasingly random vertex access pattern during `Evaluate`, reducing GPU utilization.

Figure 2 shows the average planning time for BERRT# under both serial and parallel exploitation implementations, across a variety of problem sizes N and batch sizes S. The problem size is the number of total samples considered before terminating the planner.

The number of samples in a batch has a profound effect on the performance of both the serial and parallel implementations, with larger batch sizes resulting in faster planner execution in most cases. When $N = 10^4$, increasing the batch size one order of magnitude from $S = 10$ to $S = 100$ results in a speedup of 4.84× on the CPU and 6.94× on the GPU. Compared to PI-RRT# ($S = 1$), BERRT# with $S = 100$ achieves an 8.83× speedup on CPU and 9.52× on GPU.

The largest observed speedup in this experiment was when $N = 3 \times 10^4$: increasing $S = 3$ to $S = 300$ resulted in 12.97× and 12.54× speedup on CPU and GPU implementations, respectively. The largest decrease in total planning time due to only GPU exploitation occurred when $N = 3 \times 10^4, S = 30$, a decrease of 77.9%.

The steep increase in performance with increasing batch size confirms the hypothesis that tightly coupled exploitation results in low planner performance. However, there is a limit to how much decoupling is effective. When $N = 10^4$ and $N = 3 \times 10^4$, increasing the sample size from 1% to 10% of the total problem size resulted in slight performance regressions. A possible cause of this regression is that larger batches without replanning introduce more suboptimality to the policy, requiring more policy iterations to converge back to the optimal policy. Additionally, for more challenging search problems, sampling-time optimizations based on the results of incremental exploitation are more important than in the 2D problem considered here. More frequent exploitation allows new information to be incorporated into sampling more quickly.

VII. Future work

As noted above, due to data movement and data structure preparation overhead, the proposed GPU-based implementation is only beneficial above a certain threshold. This threshold likely depends not only on machine parameters such as bus speed, memory bandwidth, and CPU cache size, but also on difficult-to-observe problem parameters such as configuration-space connectivity. Future work may investigate methods to adaptively select serial or parallel exploitation in order to maximize performance across all regimes.

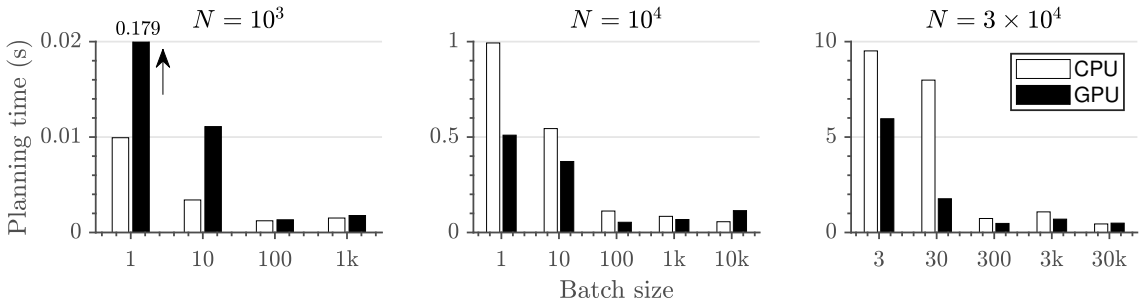


Fig. 2: Performance comparison of CPU and GPU exploitation for PI-RRT#. Graphs are labelled with corresponding problem size, N. Increasing the batch size results in significant reduction in overall planning time at all problem sizes.

Another promising avenue is asynchronous policy iteration exploitation concurrent with exploration. The BERRT# algorithm loosens the synchronization and demonstrates the potential speedup available, but the `Replan` procedure still does not run concurrently with graph extension. Since our GPU implementation successfully migrates policy iteration to a separate device, it may proceed concurrently and asynchronously with graph extension. Updated policy information and new edges in the graph may be exchanged periodically between devices. This method would allow progress on both exploration and exploitation to continue in parallel, and exposes opportunity to use complementary techniques such as C-FOREST on the CPU to parallelize graph extension.

VIII. Conclusion

We have presented a GPU-based implementation of policy iteration for use in sampling-based planning problems. The presented solution achieves an empirical speedup of 3–4$\times$ for each iteration when compared to a serial implementation, resulting in up to 77.9% decrease in end-to-end planning time. Additionally, employing the key insight that tightly synchronized exploitation slows sample throughput, we present the Batched-Extension RRT# algorithm and demonstrate its effectiveness, reducing total planning time by up to an order of magnitude.

References

[1] S. M. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," 1998.

[2] S. M. LaValle, *Planning Algorithms*. Cambridge: Cambridge University Press, 2006.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.*, vol. 30, no. 7, pp. 846–894, Jun. 1, 2011.

[4] M. Elbanhawi and M. Simic, "Sampling-Based Robot Motion Planning: A Review," *IEEE Access*, vol. 2, pp. 56–77, 2014.

[5] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch Informed Trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *IEEE Int. Conf. on Robot. and Automat.*, May 2015, pp. 3067–3074.

[6] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *Int. J. Robot. Res.*, vol. 34, no. 7, pp. 883–921, Jun. 1, 2015.

[7] L. P. Kaelbling and T. Lozano-Perez, "Hierarchical task and motion planning in the now," in *IEEE Int. Conf. on Robot. and Automat.*, Shanghai, China: IEEE, May 2011, pp. 1470–1477.

[8] M. Rickert, A. Sieverling, and O. Brock, "Balancing Exploration and Exploitation in Sampling-Based Motion Planning," *IEEE Trans. Robot.*, vol. 30, no. 6, pp. 1305–1317, Dec. 2014.

[9] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT," in *IEEE/RSJ Int. Conf. on Intell. Robots and Syst.*, Sep. 2011, pp. 3513–3518.

[10] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *Int. J. Robot. Res.*, vol. 31, no. 2, pp. 187–200, Feb. 2012.

[11] O. Arslan and P. Tsiotras, "Incremental sampling-based motion planners using policy iteration methods," in *IEEE 55th Conf. on Decis. and Control*, Dec. 2016, pp. 5004–5009.

[12] ——, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *IEEE Int. Conf. on Robot. and Automat.*, Karlsruhe, Germany, May 2013, pp. 2421–2428.

[13] L. E. Kavraki, P. Svestka, J.-. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Autom.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[14] M. Otte and N. Correll, "C-FOREST: Parallel Shortest Path Planning With Superlinear Speedup," *IEEE Trans. Robot.*, vol. 29, no. 3, pp. 798–806, Jun. 2013.

[15] J. Ichnowski and R. Alterovitz, "Parallel sampling-based motion planning with superlinear speedup," in *IEEE/RSJ Int. Conf. on Intell. Robots and Syst.*, Oct. 2012, pp. 1206–1212.

[16] B. Ichter, E. Schmerling, and M. Pavone, "Group Marching Tree: Sampling-Based Approximately Optimal Motion Planning on GPUs," in *First IEEE Int. Conf. on Robot. Comput.*, Apr. 2017, pp. 219–226.

[17] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *IEEE/RSJ Int. Conf. on Intell. Robots and Syst.*, Sep. 2014, pp. 2997–3004.

[18] NVIDIA, "CUDA C++ Programming Guide, v10.2," NVIDIA, Inc., Nov. 2019. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 02/29/2020).

[19] N. Bell, J. Hoberock, and C. Rodrigues, "THRUST: A productivity-oriented library for CUDA," in *Programming Massively Parallel Processors*, Elsevier, 2017, pp. 475–491.

[20] Y. Wang, Y. Pan, A. Davidson, *et al.*, "Gunrock: GPU Graph Analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, 3:1–3:49, Aug. 23, 2017.