An Integrated FPGA Accelerator for Deep Learning-Based 2D/3D Path Planning

Keisuke Sugiura and Hiroki Matsutani, *Member, IEEE*

Abstract—**Path planning is a crucial component for realizing the autonomy of mobile robots. However, due to limited computational resources on mobile robots, it remains challenging to deploy state-of-the-art methods and achieve real-time performance. To address this, we propose P3Net (PointNet-based Path Planning Networks), a lightweight deep-learning-based method for 2D/3D path planning, and design an IP core (P3NetCore) targeting FPGA SoCs (Xilinx ZCU104). P3Net improves the algorithm and model architecture of the recently-proposed MPNet. P3Net employs an encoder with a PointNet backbone and a lightweight planning network in order to extract robust point cloud features and sample path points from a promising region. P3NetCore is comprised of the fully-pipelined point cloud encoder, batched bidirectional path planner, and parallel collision checker, to cover most part of the algorithm. On the 2D (3D) datasets, P3Net with the IP core runs 30.52–186.36x and 7.68–143.62x (15.69–93.26x and 5.30–45.27x) faster than ARM Cortex CPU and Nvidia Jetson while only consuming 0.255W (0.809W), and is up to 1278.14x (455.34x) power-efficient than the workstation. P3Net improves the success rate by up to 28.2% and plans a near-optimal path, leading to a significantly better tradeoff between computation and solution quality than MPNet and the state-of-the-art sampling-based methods.**

Index Terms—**Path planning, neural path planning, point cloud processing, PointNet, deep learning, FPGA.**

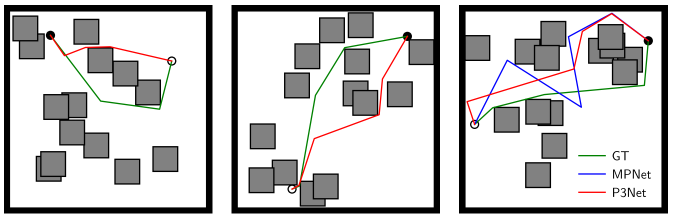

Fig. 1. Results on the P3Net2D dataset. While MPNet (blue) fails to plan feasible paths in the first two cases, the proposed P3Net (red) plans successfully in all these cases, while reducing the parameters by 32.32x.

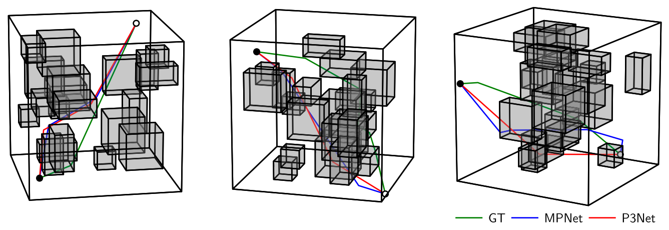

Fig. 2. Results on the P3Net3D dataset. P3Net (red) produces feasible paths in these cases with 5.43x less parameters than MPNet.

I. Introduction

Path planning aims to find a feasible path from a start to a goal position while avoiding obstacles. It is a fundamental component for mobile robots to autonomously navigate and accomplish a variety of tasks, e.g., farm monitoring [1], aerial package delivery [2], and mine exploration [3]. Such robotic applications are often deployed on resource-limited edge devices due to severe constraints on the cost and payload. In addition, real-time performance is of crucial importance, since robots may have to plan and update a path on-the-fly in the dynamic environments. To cope with the strict performance requirements, FPGA SoCs are increasingly used in robotic applications such as visual odometry [4] and SLAM [5]. FPGA SoC integrates an embedded CPU with a reconfigurable fabric, which allows to develop a custom accelerator tailored for a specific algorithm. Taking these into account, an FPGA-based efficient path planning implementation becomes an attractive solution, which would greatly broaden the application range, since mobile robots can now perform expensive planning tasks on its own without connectivity to remote servers.

In path planning, the sampling-based methods including Rapidly-exploring Random Tree (RRT) [6] and RRT* [7] are the most prominent; they explore the environment by incrementally building a tree that represents a set of valid robot motions. A number of RRT variants, e.g., Informed-RRT* [8] and BIT* [9], have been proposed to improve the sampling efficiency and convergence speed. While they offer better tradeoffs between computational effort and solution quality, they rely on carefully designed heuristics, which imply the prior knowledge of the environment and may not be effective in certain scenarios. Due to their increased algorithmic complexity, it even takes up to tens of seconds to complete a task on an embedded CPU. On top of that, their inherently sequential nature would require intricate strategies to map onto a parallel computing platform.

Manuscript received 3 July 2023; revised 4 January 2024; accepted 9 March 2024. Date of publication 18 March 2024; date of current version 10 May 2024. This work was supported by JSPS KAKENHI under Grant JP22J21699. Recommended for acceptance by J. Hormigo. *(Corresponding author: Keisuke Sugiura.)*

The authors are with the Department of Information and Computer Science, Keio University, Yokohama 223-8522, Japan (e-mail: sugiura@arc.ics.keio.ac.jp; matutani@arc.ics.keio.ac.jp).

This article has supplementary downloadable material available at https://doi.org/10.1109/TC.2024.3377895, provided by the authors.
Digital Object Identifier 10.1109/TC.2024.3377895

Motivated by the tremendous success of deep learning, the research effort is devoted to developing learning-based methods; the basic idea is to automatically acquire a policy for planning near-optimal paths from a large collection of paths generated by the existing methods. MPNet [10] is a such recently-proposed method, which employs two separate MLPs (Multi-Layer Perceptrons) for point cloud encoding and incremental path planning. Unlike sampling-based methods, MPNet does not involve operations on complex data structures (e.g., KNN (K-Nearest Neighbor) search on a K-d tree), and DNN (Deep Neural Network) inference is more amenable to parallel processing. This greatly eases the design of a custom processor and makes MPNet a promising candidate for the low-cost FPGA implementation. Its performance is however limited due to the following reasons; the encoder does not take into account the unstructured and unordered nature of point clouds, which degrades the quality of extracted features and eventually results in a lower success rate. Furthermore, the planning network has a lower parameter efficiency, and MPNet has a limited parallelism as it processes only one candidate path at a time until a feasible solution is found.

This paper addresses the above limitations of MPNet and proposes a new learning-based method for 2D/3D path planning, named **P3Net** (PointNet-based Path Planning Networks), along with its custom IP core for FPGAs (**P3NetCore**). While the existing methods often assume the availability of abundant computing resources (e.g., GPUs), which is not the case in practice, P3Net is designed to work on resource-limited edge devices and still deliver satisfactory performance. Besides, to our knowledge, P3NetCore is one of the first FPGA accelerators for fully learning-based path planning. The main contributions of this paper are summarized as follows:

1) To improve parameter efficiency and extract features that are permutation-invariant, we utilize a PointNet [11]-based encoder architecture, which is specifically designed for point cloud processing, together with a lightweight planning network.
2) We make two algorithmic modifications to MPNet; we introduce a batch planning strategy to process multiple candidate paths in parallel, which offers a new parallelism and improves the success rates without increasing the computation time. We then add a refinement phase at the end to iteratively optimize the path.
3) To significantly improve the tradeoff between computation time and success rate, we design a custom IP core for P3Net, which integrates a point cloud encoder, a bidirectional neural planner, and a collision checker.

II. Related Works

A. Sampling and Learning-Based Path Planning

The sampling-based methods, e.g., RRT [6] and RRT* [7], are prevalent in robotic path planning; they explore the environment by incrementally growing an exploration tree. Considering that the free space should be densely filled with tree nodes to find a high-quality solution, the computational complexity is at worst exponential with respect to the space dimension. The later methods introduce various heuristics to improve search efficiency; Informed-RRT* [8] uses an ellipsoidal heuristic, while BIT* [9] and its variants [12], [13] apply graph-search techniques. Despite the steady improvement, they still rely on sophisticated heuristics; deep learning-based methods have been extensively studied to automatically learn effective policies for planning high-quality paths.

Several studies have investigated the hybrid approach, where deep learning techniques are incorporated into the classical planners. Ichter et al. [14], [15] and Wang et al. [16] extend RRT by generating informed samples from a learned latent space. Neural A* [17] is a differentiable version of A*, while WPN [18] uses LSTM to generate path waypoints and then A* to connect them. Aside from the hybrid approach, an end-to-end approach aims to directly learn the behavior of classical planners via supervised learning; Inoue et al. [19] and Bency et al. [20] train LSTM networks on the RRT* and A*-generated paths, respectively. The other architectures, e.g., CNN [21] and Transformer [22] are also employed to construct end-to-end models. MPNet and our P3Net follow the end-to-end supervised approach, and perform path planning on a continuous domain by directly regressing coordinates of the path points. P3Net is unique in that it puts more emphasis on the computational and resource efficiency and builds upon a lightweight MLP network, making it suitable for the low-end edge devices.

B. Hardware Acceleration of Path Planning

Several works have explored the FPGA and ASIC acceleration of the conventional graph and sampling-based methods, e.g., A* [23], [24], [25], [26] and RRT [27], [28], [29], [30]. Kosuge et al. [24] develops an accelerator for A* graph construction and search on the Xilinx ZCU102. Since A* operates on grid environments and is subject to the curse of dimensionality, it is challenging to handle higher dimensional cases or larger maps. For RRT-family algorithms, Malik et al. [27] proposes a parallelized architecture for RRT, which first partitions the workspace into grids and distributes them across multiple RRT processes. Chung et al. [30] devises a dual-tree RRT with parallel and greedy tree expansion for ASIC implementation. Some studies leverage GPU [31] or distributed computing techniques [32], [33] to speed up RRT, while it degrades power efficiency and not suitable for battery-powered edge devices. RRT-family algorithms repeat tree expansion and rewiring steps alternately; they are inherently sequential and difficult to accelerate without a sophisticated technique (e.g., space subdivision, parallel tree expansion). In contrast, our proposed P3Net offers more parallelism and is hardware-friendly, as it mainly consists of DNN inferences, and does not operate on complex data structures (e.g., K-d tree).

Only a few works [34], [35], [36] have considered the hardware acceleration of neural planners. Huang et al. [35] presents an accelerator for a sampling-based method with a CNN model, which produces a probability map given an image of the environment for sampling the next robot position. In [36], an RTL design of a Graph Neural Network-based path explorer rapidly evaluates priority scores for edges in a random

Algorithm 1 MPNet for 2D and 3D path planning

```
Require: Start $\mathbf{c}_{\mathrm{start}}$, goal $\mathbf{c}_{\mathrm{goal}}$, obstacle point cloud $\mathcal{P}$
Ensure: Path $\tau = \{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_T\}$ $(\mathbf{c}_0, \mathbf{c}_T = \mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}})$
 1: $\phi(\mathcal{P}) \leftarrow \mathrm{ENet}(\mathcal{P})$                         ▷ Point cloud feature extraction
    ▷ Initial coarse planning
 2: $\tau \leftarrow \mathrm{NeuralPlanner}(\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}}, \phi(\mathcal{P}))$
 3: if $\tau = \varnothing$ : return $\varnothing$                                   ▷ Failure
 4: $\tau \leftarrow \mathrm{Smoothing}(\tau)$
 5: if $\tau$ is collision-free : return $\tau$                          ▷ Success
    ▷ Replanning
 6: for $i = 0, \ldots, I_{\mathrm{Replan}} - 1$ do
 7:     $\tau \leftarrow \mathrm{Replan}(\tau, \phi(\mathcal{P}))$
 8:     $\tau \leftarrow \mathrm{Smoothing}(\tau)$
 9:     if $\tau \neq \varnothing$ and $\tau$ is collision-free : return $\tau$     ▷ Success
10: return $\varnothing$                                                     ▷ Failure
11: function NeuralPlanner($\mathbf{c}_s, \mathbf{c}_g, \phi(\mathcal{P})$)
12:     $\tau^{\mathrm{a}} \leftarrow \{\mathbf{c}_s\}$ , $\tau^{\mathrm{b}} \leftarrow \{\mathbf{c}_g\}$              ▷ Forward-backward paths
13:     $\mathbf{c}^{\mathrm{a}}_{\mathrm{end}} \leftarrow \mathbf{c}_s$, $\mathbf{c}^{\mathrm{b}}_{\mathrm{end}} \leftarrow \mathbf{c}_g$, $r = 0$                 ▷ Path endpoints
14:     for $i = 0, \ldots, I - 1$ do
15:         if $r = 0$ :                              ▷ Forward direction (start to goal)
16:             $\mathbf{c}^{\mathrm{a}}_{\mathrm{new}} \leftarrow \mathrm{PNet}(\phi(\mathcal{P}), \mathbf{c}^{\mathrm{a}}_{\mathrm{end}}, \mathbf{c}_g)$
17:             $\tau^{\mathrm{a}} \xleftarrow{+} \{\mathbf{c}^{\mathrm{a}}_{\mathrm{new}}\}$ , $\mathbf{c}^{\mathrm{a}}_{\mathrm{end}} \leftarrow \mathbf{c}^{\mathrm{a}}_{\mathrm{new}}$, $r = 1$
18:         else if $r = 1$ :                       ▷ Reverse direction (goal to start)
19:             $\mathbf{c}^{\mathrm{b}}_{\mathrm{new}} \leftarrow \mathrm{PNet}(\phi(\mathcal{P}), \mathbf{c}^{\mathrm{b}}_{\mathrm{end}}, \mathbf{c}_s)$
20:             $\tau^{\mathrm{b}} \xleftarrow{+} \{\mathbf{c}^{\mathrm{b}}_{\mathrm{new}}\}$ , $\mathbf{c}^{\mathrm{b}}_{\mathrm{end}} \leftarrow \mathbf{c}^{\mathrm{b}}_{\mathrm{new}}$, $r = 0$
21:         if $\tau^{\mathrm{a}}$ and $\tau^{\mathrm{b}}$ are connectable : return $\tau = \{\tau^{\mathrm{a}}, \tau^{\mathrm{b}}\}$
22:     return $\varnothing$                                                 ▷ Failure
23: function Replan($\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}, \phi(\mathcal{P})$)
24:     $\tau_{\mathrm{new}} \leftarrow \varnothing$
25:     for $i = 0, \ldots, T - 1$ do
26:         if $\mathbf{c}_i$ and $\mathbf{c}_{i+1}$ are connectable :
27:             $\tau_{\mathrm{new}} \xleftarrow{+} \{\mathbf{c}_i, \mathbf{c}_{i+1}\}$
28:         else         ▷ Plan a detour $\tau_{i,i+1} = \{\mathbf{c}_i, \mathbf{c}^{(1)}_i, \mathbf{c}^{(2)}_i, \ldots, \mathbf{c}_{i+1}\}$
29:             $\tau_{i,i+1} \leftarrow \mathrm{NeuralPlanner}(\mathbf{c}_i, \mathbf{c}_{i+1}, \phi(\mathcal{P}))$
30:             if $\tau_{i,i+1} = \varnothing$ : return $\varnothing$                        ▷ Failure
31:             $\tau_{\mathrm{new}} \xleftarrow{+} \tau_{i,i+1}$
32:     return $\tau_{\mathrm{new}}$                                             ▷ Success
```

geometric graph, and edges with high priority are selected to form a path. This paper extends our previous work [34]; instead of only accelerating the DNN inference in MPNet, we implement the whole bidirectional planning algorithm on FPGA. In addition, we derive a new path planning method, P3Net, to achieve both higher success rate and speedup. This paper is one of the first to realize a real-time fully learning-based path planner on a resource-limited FPGA device.

III. Preliminaries: MPNet

In this section, we briefly describe the MPNet [10] algorithm (Alg. 1), which serves as a basis of our proposal.

Let us consider a robot moving around in a 2D/3D environment $\mathcal{X} \subset \mathbb{R}^D$ $(D = 2, 3)$. We denote its position as $\mathbf{c} \in \mathbb{R}^D$. Given a pair of start and goal points $\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}} \in \mathbb{R}^D$, MPNet tries to find a collision-free path $\tau = \{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_T\}$ $(\mathbf{c}_0, \mathbf{c}_T = \mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}})$ if exists. As illustrated in Fig. 3 (left), MPNet assumes that obstacle information is represented as a point cloud $\mathcal{P} = \{\mathbf{p}_0, \ldots, \mathbf{p}_{N-1}\} \in \mathbb{R}^{N \times D}$ containing N points uniformly sampled from the obstacle region. The notation $\tau \xleftarrow{+} \mathbf{c}$ is a shorthand for $\tau \leftarrow \tau \cup \{\mathbf{c}\}$.

Importantly, MPNet uses two DNN models for encoding and planning, namely **ENet** and **PNet** (Fig. 3, right); ENet compresses the obstacle information $\mathcal{P}$ into a feature embedding $\phi(\mathcal{P}) \in \mathbb{R}^F$. PNet is responsible for sampling the next position

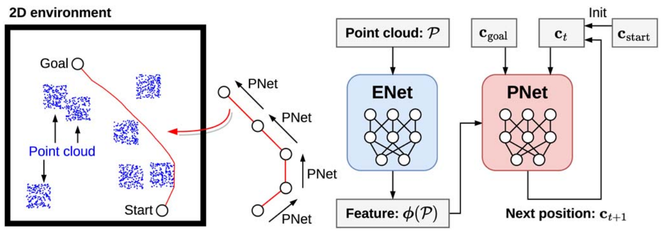
2D environment
Goal
Point cloud
Start
PNet
PNet
PNet
PNet
Point cloud: $\mathcal{P}$
ENet
Feature: $\phi(\mathcal{P})$
$\mathbf{c}_{\mathrm{goal}}$
$\mathbf{c}_t$
Init
$\mathbf{c}_{\mathrm{start}}$
PNet
Next position: $\mathbf{c}_{t+1}$

Fig. 3. Overview of the MPNet algorithm.

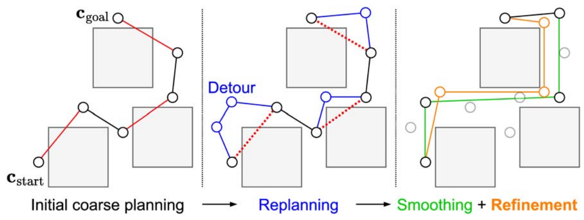

Fig. 4. Processing flow of the MPNet path planning.

$\mathbf{c}_{t+1}$ which is one step closer to the goal, from the current and goal positions $\mathbf{c}_t, \mathbf{c}_{\mathrm{goal}}$ as well as the obstacle feature $\phi(\mathcal{P})$. Fig. 4 outlines the algorithm. MPNet consists of two main steps, referred to as (1) **initial coarse planning** and (2) **replanning**, plus (3) a final **smoothing** step.

A. MPNet Algorithm

MPNet first extracts a feature $\phi(\mathcal{P})$ (Alg. 1, line 1) and proceeds to the **initial coarse planning** step (line 2) to roughly plan a path τ between $\mathbf{c}_{\mathrm{start}}$ and $\mathbf{c}_{\mathrm{goal}}$. The bidirectional planning with PNet, referred to as NeuralPlanner (lines 11-22), plays a central role in this step.

Given a pair of start-goal points $\mathbf{c}_s, \mathbf{c}_g$, NeuralPlanner plans two paths $\tau^{\mathrm{a}}, \tau^{\mathrm{b}}$ in forward and reverse directions alternately (lines 15, 18). The forward path $\tau^{\mathrm{a}} = \{\mathbf{c}_s, \ldots, \mathbf{c}^{\mathrm{a}}_{\mathrm{end}}\}$ is incrementally expanded from start to goal by repeating the PNet inference (lines 16-17). From the current path endpoint $\mathbf{c}^{\mathrm{a}}_{\mathrm{end}}$ and goal $\mathbf{c}_g$, PNet computes a new waypoint $\mathbf{c}^{\mathrm{a}}_{\mathrm{new}}$, which becomes a new endpoint of τ^{a} and used as input in the next inference. Similarly, the backward path $\tau^{\mathrm{b}} = \{\mathbf{c}_g, \ldots, \mathbf{c}^{\mathrm{b}}_{\mathrm{end}}\}$ is expanded from goal to start (lines 19-20). After updating τ^{a} or τ^{b}, NeuralPlanner attempts to connect them and create a path $\tau = \{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_T\}$ between $\mathbf{c}_0, \mathbf{c}_T = \mathbf{c}_s, \mathbf{c}_g$, if there is no obstacle between path endpoints $\mathbf{c}^{\mathrm{a}}_{\mathrm{end}}, \mathbf{c}^{\mathrm{b}}_{\mathrm{end}}$ (line 21). The above process, i.e., path expansion and collision checking, is repeated until a feasible path is obtained or the maximum number of iterations I is reached. The algorithm fails if $\tau^{\mathrm{a}}, \tau^{\mathrm{b}}$ cannot be connected after I iterations[1] (line 22).

The tentative path τ connecting $\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}}$ is obtained from NeuralPlanner (line 2); if τ passes the collision check, then MPNet performs **smoothing** and returns it as a final solution (lines 4-5). In the smoothing process (Fig. 4, right), the planner

[1]As more endpoints $\mathbf{c}^{\mathrm{a}}_{\mathrm{end}}, \mathbf{c}^{\mathrm{b}}_{\mathrm{end}}$ are sampled for the forward-backward paths $\tau^{\mathrm{a}}, \tau^{\mathrm{b}}$, it is more likely that τ^{a} and τ^{b} meet each other at some point and are connectable. More simply, setting a larger I will increase the chance of success of NeuralPlanner.

greedily prunes redundant waypoints from τ to obtain a shorter and smoother path; given three waypoints $\mathbf{c}_i, \mathbf{c}_j, \mathbf{c}_k$ $(i < j < k)$, the intermediate one $\mathbf{c}_j$ is dropped if $\mathbf{c}_i$ and $\mathbf{c}_k$ can be directly connected by a straight line. This involves a collision checking on the new edge $(\mathbf{c}_i, \mathbf{c}_k)$.

As already mentioned, the initial solution τ may contain edges that collide with obstacles (Fig. 4 (left, red lines)); if this is the case, the planner moves on to the **replanning** process (lines 7, 23-32). For every edge $\mathbf{c}_i, \mathbf{c}_{i+1} \in \tau$ that is in collision, MPNet tries to plan an alternative sub-path $\tau_{i,i+1} = \{\mathbf{c}_i, \mathbf{c}_i^{(1)}, \mathbf{c}_i^{(2)}, \ldots, \mathbf{c}_{i+1}\}$ between $\mathbf{c}_i, \mathbf{c}_{i+1}$ to avoid obstacles (Fig. 4 (center), line 29). NeuralPlanner is again called with $\mathbf{c}_i, \mathbf{c}_{i+1}$ as input start-goal points. The replanning fails if NeuralPlanner cannot plan a detour. The new waypoints $\mathbf{c}_i^{(1)}, \mathbf{c}_i^{(2)}, \ldots$ are then inserted to the path (line 31). If the resultant path is collision-free, MPNet returns it as a solution after smoothing (lines 8-9); otherwise, it runs the replanning again. In this way, MPNet gradually removes the non-collision-free edges from the solution. Replanning is repeated for I_{Replan} times at maximum, and MPNet fails if a feasible solution is not obtained (line 10).

PNet inference is a non-deterministic process and shows a stochastic behavior, as it utilizes dropout in the inference phase, unlike the typical case where the dropout is only enabled during training. The process $[\boldsymbol{\phi}(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\mathrm{goal}}] \mapsto \mathbf{c}_{t+1}$ can be viewed as sampling the next position $\mathbf{c}_{t+1}$ from a learned distribution $p(\boldsymbol{\phi}(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\mathrm{goal}})$. For this reason, NeuralPlanner and Replan are also non-deterministic. As a result, MPNet generates multiple different sub-paths in the replanning phase; by increasing I_{Replan}, MPNet has more chance of avoiding obstacles and finding a feasible path.

IV. Method: P3Net

In this section, we propose a new path planning algorithm, **P3Net** (Algs. 2-3), by making improvements to the algorithm and model architecture of MPNet.

A. Algorithmic Improvements

P3Net introduces two ideas, (1) **batch planning** and (2) **refinement step**, into the MPNet algorithm. As depicted in Figs. 4–5, P3Net (1) estimates multiple paths at the same time to increase the parallelism, and (2) iteratively refines the obtained path to improve its quality.

1) Batch Planning: According to the NeuralPlanner algorithm in MPNet (Alg. 1, lines 11-22), forward-backward paths τ_a, τ_b are incrementally expanded from start-goal points $\mathbf{c}_s, \mathbf{c}_g$ until their endpoints $\mathbf{c}_{\mathrm{end}}^{\mathrm{a}}, \mathbf{c}_{\mathrm{end}}^{\mathrm{b}}$ are connectable. In this process, PNet computes a single next position $\mathbf{c}_{\mathrm{new}}^{\mathrm{a}}$ $(\mathbf{c}_{\mathrm{new}}^{\mathrm{b}})$ from a current endpoint $\mathbf{c}_{\mathrm{end}}^{\mathrm{a}}$ $(\mathbf{c}_{\mathrm{end}}^{\mathrm{b}})$ and the destination $\mathbf{c}_g$ $(\mathbf{c}_s)$ (lines 16, 19). Due to the input batch size of one, PNet cannot fully utilize the parallel computing capability of CPU/GPUs, and also suffers from the kernel launch overheads and frequent data transfers. To amortize this overhead, two PNet inferences (lines 16, 19) can be merged into one with a batch size of two, i.e., two next positions $[\mathbf{c}_s, \mathbf{c}_g]$ are computed at once from concatenated inputs $\left[\mathbf{c}_{\mathrm{end}}^{\mathrm{a}}, \mathbf{c}_{\mathrm{end}}^{\mathrm{b}}\right], [\mathbf{c}_g, \mathbf{c}_s]$.

Algorithm 2 P3Net (changed parts are highlighted in red)

```
Require: Start $\mathbf{c}_{\mathrm{start}}$, goal $\mathbf{c}_{\mathrm{goal}}$, obstacle point cloud $\mathcal{P}$
Ensure: Path $\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}$ $(\mathbf{c}_0, \mathbf{c}_T = \mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}})$
 1: Compute point cloud feature: $\boldsymbol{\phi}(\mathcal{P}) \leftarrow$ ENet($\mathcal{P}$)
    ▷ Initial coarse planning
 2: for $i = 0, \ldots, I_{\mathrm{Init}} - 1$ do
 3:     $\tau \leftarrow$ NeuralPlannerEx($\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}}, \boldsymbol{\phi}(\mathcal{P})$)
 4:     if $\tau \neq \varnothing$ : break                                   ▷ Success
 5: if $\tau = \varnothing$ : return $\varnothing$                                 ▷ Failure
 6: $\tau \leftarrow$ Smoothing($\tau$)
    ▷ Replanning
 7: if $\tau$ is not collision-free :
 8:     for $i = 0, \ldots, I_{\mathrm{Replan}} - 1$ do
 9:         $\tau \leftarrow$ Replan($\tau, \boldsymbol{\phi}(\mathcal{P})$)
10:         $\tau \leftarrow$ Smoothing($\tau$)
11:         if $\tau \neq \varnothing$ and $\tau$ is collision-free : break      ▷ Success
12:     if $\tau = \varnothing$ : return $\varnothing$                           ▷ Failure
    ▷ Refinement
13: $\tau_{\mathrm{best}} \leftarrow \tau$,  $c_{\mathrm{best}} \leftarrow$ Cost($\tau_{\mathrm{best}}$)          ▷ $\tau$ is now collision-free
14: for $i = 0, \ldots, I_{\mathrm{Refine}} - 1$ do
15:     $\tau_{\mathrm{new}} \leftarrow$ Refine($\tau_{\mathrm{best}}, \boldsymbol{\phi}(\mathcal{P})$)
16:     $\tau_{\mathrm{new}} \leftarrow$ Smoothing($\tau_{\mathrm{new}}$)
17:     $c_{\mathrm{new}} \leftarrow$ Cost($\tau_{\mathrm{new}}$)
18:     if $c_{\mathrm{new}} < c_{\mathrm{best}}$ : $c_{\mathrm{best}} = c_{\mathrm{new}}$,  $\tau_{\mathrm{best}} = \tau_{\mathrm{new}}$
19: return $\tau_{\mathrm{best}}$
```

Algorithm 3 Batch Planning and Refinement Step in P3Net

```
 1: function NeuralPlannerEx($\mathbf{c}_s, \mathbf{c}_g, \boldsymbol{\phi}(\mathcal{P})$)
    ▷ Initialize batch of current and goal positions
 2:     $\mathbf{C}_{\mathcal{B}} \leftarrow [\mathbf{c}_0^{\mathrm{a},0}, \mathbf{c}_0^{\mathrm{b},0}, \ldots, \mathbf{c}_{B-1}^{\mathrm{a},0}, \mathbf{c}_{B-1}^{\mathrm{b},0}] \in \mathbb{R}^{2B \times D}$
          $(\forall j$ $\mathbf{c}_j^{\mathrm{a},0} = \mathbf{c}_s, \mathbf{c}_j^{\mathrm{b},0} = \mathbf{c}_g)$
 3:     $\mathbf{C}_{\mathcal{B}}^{\mathrm{goal}} \leftarrow [\mathbf{c}_g, \mathbf{c}_s, \mathbf{c}_g, \mathbf{c}_s, \ldots, \mathbf{c}_g, \mathbf{c}_s] \in \mathbb{R}^{2B \times D}$
    ▷ Initialize batch of paths and lengths
 4:     $\tau_{\mathcal{B}}^{\mathrm{a}} \leftarrow [\tau_0^{\mathrm{a}}, \tau_1^{\mathrm{a}}, \ldots, \tau_{B-1}^{\mathrm{a}}]$, $\forall j$ $\tau_j^{\mathrm{a}} = \{\mathbf{c}_s\}$
 5:     $\tau_{\mathcal{B}}^{\mathrm{b}} \leftarrow [\tau_0^{\mathrm{b}}, \tau_1^{\mathrm{b}}, \ldots, \tau_{B-1}^{\mathrm{b}}]$, $\forall j$ $\tau_j^{\mathrm{b}} = \{\mathbf{c}_g\}$
 6:     $\ell_{\mathcal{B}} \leftarrow [\ell_0^{\mathrm{a}}, \ell_0^{\mathrm{b}}, \ldots, \ell_{B-1}^{\mathrm{a}}, \ell_{B-1}^{\mathrm{b}}]$, $\forall j$ $\ell_j^{\mathrm{x}} = |\tau_j^{\mathrm{x}}| = 1$
 7:     for $i = 0, \ldots, I - 1$ do
 8:         $\mathbf{C}_{\mathcal{B}}^{\mathrm{next}} \leftarrow$ PNet($\boldsymbol{\phi}(\mathcal{P}), \mathbf{C}_{\mathcal{B}}, \mathbf{C}_{\mathcal{B}}^{\mathrm{goal}}$)      ▷ Next positions
            $(\mathbf{C}_{\mathcal{B}}^{\mathrm{next}} = [\mathbf{c}_0^{\mathrm{a},i+1}, \mathbf{c}_0^{\mathrm{b},i+1}, \ldots, \mathbf{c}_{B-1}^{\mathrm{a},i+1}, \mathbf{c}_{B-1}^{\mathrm{b},i+1}])$
 9:         for $j = 0, \ldots, B - 1$ do                       ▷ Collision checks
10:             if $(\mathbf{c}_j^{\mathrm{a},i+1}, \mathbf{c}_j^{\mathrm{b},i})$ are connectable :
11:                 expandA $\leftarrow 1$, expandB $\leftarrow 0$, $s \leftarrow 1$
12:             else if $(\mathbf{c}_j^{\mathrm{a},i}, \mathbf{c}_j^{\mathrm{b},i+1})$ are connectable :
13:                 expandA $\leftarrow 0$, expandB $\leftarrow 1$, $s \leftarrow 1$
14:             else if $(\mathbf{c}_j^{\mathrm{a},i+1}, \mathbf{c}_j^{\mathrm{b},i+1})$ are connectable :
15:                 expandA $\leftarrow 1$, expandB $\leftarrow 1$, $s \leftarrow 1$
16:             else
17:                 expandA $\leftarrow 1$, expandB $\leftarrow 1$, $s \leftarrow 0$
18:             if expandA : $\tau_j^{\mathrm{a}} \xleftarrow{+} \{\mathbf{c}_j^{\mathrm{a},i+1}\}$, $\ell_j^{\mathrm{a}} \leftarrow \ell_j^{\mathrm{a}} + 1$
19:             if expandB : $\tau_j^{\mathrm{b}} \xleftarrow{+} \{\mathbf{c}_j^{\mathrm{b},i+1}\}$, $\ell_j^{\mathrm{b}} \leftarrow \ell_j^{\mathrm{b}} + 1$
20:             if $s = 1$ : return $\tau = \{\tau_j^{\mathrm{a}}, \tau_j^{\mathrm{b}}\}$                ▷ Success
21:         $\mathbf{C}_{\mathcal{B}} \leftarrow \mathbf{C}_{\mathcal{B}}^{\mathrm{next}}$
22:     return $\varnothing$                                          ▷ Failure

23: function Refine($\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}, \boldsymbol{\phi}(\mathcal{P})$)
24:     $\tau_{\mathrm{new}} \leftarrow \varnothing$
25:     for $i = 0, \ldots, T - 1$ do
26:         $\tau_{i,i+1} \leftarrow$ NeuralPlannerEx($\mathbf{c}_i, \mathbf{c}_{i+1}, \boldsymbol{\phi}(\mathcal{P})$)
27:                          ▷ Compute a new path connecting $\mathbf{c}_i$ and $\mathbf{c}_{i+1}$
28:         if $\tau_{i,i+1} \neq \varnothing$ and $\tau_{i,i+1}$ is collision-free :
29:             $\tau_{\mathrm{new}} \xleftarrow{+} \tau_{i,i+1}$                          ▷ Use new path
30:         else
31:             $\tau_{\mathrm{new}} \xleftarrow{+} \{\mathbf{c}_i, \mathbf{c}_{i+1}\}$                 ▷ Use current solution
32:     return $\tau_{\mathrm{new}}$
```

As shown in Fig. 5, our NeuralPlannerEx algorithm (Alg. 3, lines 1-22) takes this idea further by creating a total of B pairs of forward-backward paths (i.e., $\tau_{\mathcal{B}}^{\mathrm{a}} = \left[\tau_0^{\mathrm{a}}, \ldots, \tau_{B-1}^{\mathrm{a}}\right]$, $\tau_{\mathcal{B}}^{\mathrm{b}} = \left[\tau_0^{\mathrm{b}}, \ldots, \tau_{B-1}^{\mathrm{b}}\right]$). It serves as a drop-in replacement for

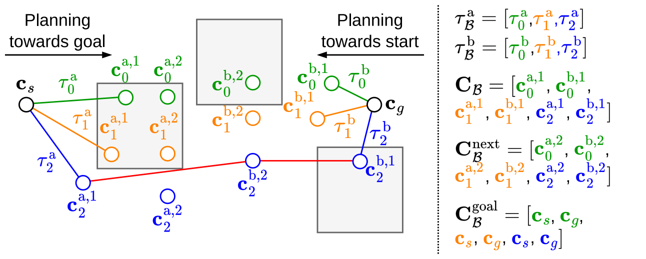
Planning towards goal
Planning towards start
$\mathbf{c}_s$
$\mathbf{c}_g$
τ_0^{a}
τ_1^{a}
τ_2^{a}
τ_0^{b}
τ_1^{b}
τ_2^{b}
$\mathbf{c}_0^{\mathrm{a},1}$
$\mathbf{c}_0^{\mathrm{a},2}$
$\mathbf{c}_1^{\mathrm{a},1}$
$\mathbf{c}_1^{\mathrm{a},2}$
$\mathbf{c}_2^{\mathrm{a},1}$
$\mathbf{c}_2^{\mathrm{a},2}$
$\mathbf{c}_0^{\mathrm{b},1}$
$\mathbf{c}_0^{\mathrm{b},2}$
$\mathbf{c}_1^{\mathrm{b},1}$
$\mathbf{c}_1^{\mathrm{b},2}$
$\mathbf{c}_2^{\mathrm{b},1}$
$\mathbf{c}_2^{\mathrm{b},2}$
$\tau_{\mathcal{B}}^{\mathrm{a}} = [\tau_0^{\mathrm{a}}, \tau_1^{\mathrm{a}}, \tau_2^{\mathrm{a}}]$
$\tau_{\mathcal{B}}^{\mathrm{b}} = [\tau_0^{\mathrm{b}}, \tau_1^{\mathrm{b}}, \tau_2^{\mathrm{b}}]$
$\mathbf{C}_{\mathcal{B}} = [\mathbf{c}_0^{\mathrm{a},1}, \mathbf{c}_0^{\mathrm{b},1}, \mathbf{c}_1^{\mathrm{a},1}, \mathbf{c}_1^{\mathrm{b},1}, \mathbf{c}_2^{\mathrm{a},1}, \mathbf{c}_2^{\mathrm{b},1}]$
$\mathbf{C}_{\mathcal{B}}^{\mathrm{next}} = [\mathbf{c}_0^{\mathrm{a},2}, \mathbf{c}_0^{\mathrm{b},2}, \mathbf{c}_1^{\mathrm{a},2}, \mathbf{c}_1^{\mathrm{b},2}, \mathbf{c}_2^{\mathrm{a},2}, \mathbf{c}_2^{\mathrm{b},2}]$
$\mathbf{C}_{\mathcal{B}}^{\mathrm{goal}} = [\mathbf{c}_s, \mathbf{c}_g, \mathbf{c}_s, \mathbf{c}_g, \mathbf{c}_s, \mathbf{c}_g]$

Fig. 5. Batch planning in NeuralPlannerEx algorithm (batch size $B = 3$, iteration $i = 2$). The algorithm expands three forward-backward path pairs $\tau_{\mathcal{B}}^{\mathrm{a}}, \tau_{\mathcal{B}}^{\mathrm{b}}$, and tries to connect each path pair. The third pair $(\tau_2^{\mathrm{a}}, \tau_2^{\mathrm{b}})$ is connected to form a path from start to goal $\tau = \{\mathbf{c}_s, \mathbf{c}_2^{\mathrm{a},1}, \mathbf{c}_2^{\mathrm{b},2}, \mathbf{c}_2^{\mathrm{b},1}, \mathbf{c}_g\}$.

NeuralPlanner. It keeps track of the forward-backward path pairs $(\tau_{\mathcal{B}}^{\mathrm{a}}, \tau_{\mathcal{B}}^{\mathrm{b}})$, which are initialized with start-goal points (lines 4-5), as well as path lengths $\ell_{\mathcal{B}}$ (initialized with all ones, line 6), path endpoints $\mathbf{C}_{\mathcal{B}}$, and corresponding destination points $\mathbf{C}_{\mathcal{B}}^{\mathrm{goal}}$ (lines 2-3).

In each iteration i, PNet takes $(\phi(\mathcal{P}), \mathbf{C}_{\mathcal{B}}, \mathbf{C}_{\mathcal{B}}^{\mathrm{goal}})$ as input and computes the next waypoints $\mathbf{C}_{\mathcal{B}}^{\mathrm{next}} = [\mathbf{c}_0^{\mathrm{a},i+1}, \mathbf{c}_0^{\mathrm{b},i+1}, \ldots, \mathbf{c}_{B-1}^{\mathrm{a},i+1}, \mathbf{c}_{B-1}^{\mathrm{b},i+1}]$ for a batch of paths within one inference step (line 8), resulting in a total batch size of $2B$. Note that $\mathbf{c}_j^{\mathrm{a},i+1}, \mathbf{c}_j^{\mathrm{b},i+1}$ denote $i+1$-th waypoints in j-th forward-backward paths ($j \in [0, B)$). Then, for each sample j, the algorithm tries to connect a path pair $\tau_j^{\mathrm{a}}, \tau_j^{\mathrm{b}}$, by checking whether any of the three lines connecting $(\mathbf{c}_j^{\mathrm{a},i+1}, \mathbf{c}_j^{\mathrm{b},i})$, $(\mathbf{c}_j^{\mathrm{a},i}, \mathbf{c}_j^{\mathrm{b},i+1})$, and $(\mathbf{c}_j^{\mathrm{a},i+1}, \mathbf{c}_j^{\mathrm{b},i+1})$ is obstacle-free and hence passable (lines 9-17). If this check passes, τ_j^{a} and τ_j^{b} are concatenated and the result $\tau = \left[\mathbf{c}_s, \mathbf{c}_s^{\mathrm{a},1}, \ldots, \mathbf{c}_s^{\mathrm{b},1}, \mathbf{c}_g\right]$ is returned to the caller (line 20, blue path in Fig. 5); otherwise, the algorithm updates the current endpoints $\mathbf{C}_{\mathcal{B}}$ with the new ones $\mathbf{C}_{\mathcal{B}}^{\mathrm{next}}$ and proceeds to the next iteration. $\mathbf{C}_{\mathcal{B}}^{\mathrm{next}}$ is also used to update the paths $\tau_{\mathcal{B}}^{\mathrm{a}}, \tau_{\mathcal{B}}^{\mathrm{b}}$ accordingly (lines 18-19). It fails if no solution is found after the maximum number of iterations I.

NeuralPlannerEx is more likely to find a solution compared to NeuralPlanner, as it creates B candidate paths for a given task. This allows the replanning process (Alg. 2, lines 7-12), which repetitively calls NeuralPlannerEx, to complete in a less number of trials, leading to higher success rates as confirmed in the evaluation (Section VI-C). To further improve success rates, P3Net runs the initial coarse planning for $I_{\mathrm{Init}} \geq 1$ times (Alg. 2, lines 2-5), as opposed to MPNet which immediately fails when a feasible path is not obtained in the first attempt (Alg. 1, lines 2, 3).

2) Refinement Step: The paths generated by MPNet may not be optimal, since it returns a first found path in an initial coarse planning or a replanning phase. As highlighted in Alg. 2, lines 13-18, the refinement phase is added at the end of P3Net to gradually improve the quality of output paths (Fig. 5). For a fixed number of iterations I_{Refine}, it computes a new collision-free path τ_{new} based on the current solution τ_{best} (with a cost of c_{best}) using Refine algorithm (Alg. 3, lines 23-32), and accepts τ_{new} as a new solution if it lowers the cost ($c_{\mathrm{new}} < c_{\mathrm{best}}$). Same as the replanning phase, Refine also relies on NeuralPlannerEx at its core. It takes the collision-free path τ as an input and builds a new path τ_{new} as follows: for every edge $(\mathbf{c}_i, \mathbf{c}_{i+1}) \in \tau$, it plans a path $\tau_{i,i+1}$ using NeuralPlannerEx (line 26) and connects $\mathbf{c}_i, \mathbf{c}_{i+1}$ with $\tau_{i,i+1}$ if it is collision-free (line 29). Note that MPNet can be viewed as a special case of P3Net with $(B, I_{\mathrm{Init}}, I_{\mathrm{Refine}}) = (1, 1, 0)$.

B. Lightweight Encoding and Planning Networks

Instead of E, PNet, P3Net uses a lightweight encoder with a PointNet [11] backbone (**ENetLite**) to extract permutation-invariant features, in conjunction with a downsized planning network (**PNetLite**) for better parameter efficiency and faster inference, as described in the following[2].

1) ENetLite: PointNet-Based Encoding Network: As shown in Fig. 6 (top left), MPNet uses a simple encoder architecture (ENet) stacking four FC layers. It directly processes raw point coordinates, and hence costly preprocessing such as normal estimation or clustering is not required. ENet2D takes a point cloud $\mathcal{P}$ with $N = 1400$ points, flattens it into a 2,800D vector, and produces a $F = 28$D feature vector $\phi(\mathcal{P})$ in a single forward pass. Similarly, ENet3D extracts a $F = 60$D feature vector $\phi(\mathcal{P})$ from a 3D point cloud of size $N = 2000$ with a series of FC layers[3].

In spite of its simplicity, ENet has the following major drawbacks; (1) the number of input points is fixed to $N = 1400$ or $N = 2000$ regardless of the complexity of planning environments, (2) the number of parameters grows linearly with the number of points, and more importantly, (3) the output feature is affected by the input orderings. This means ENet produces a different feature if any of the two points are swapped; since the input still represents exactly the same point set, the result should remain unchanged.

P3Net avoids these drawbacks by using PointNet [11] as an encoder backbone, referred to as **ENetLite**. PointNet is specifically designed for point cloud processing, and Fig. 6 (top center) presents its architecture. It is still a fully-connected network and directly operates on raw point clouds. ENetLite2D first extracts $F = 252$D individual features $\{\psi(\mathbf{p}_0), \ldots, \psi(\mathbf{p}_{N-1})\}$ for each point using five building blocks. It then computes a 252D global feature $\phi(\mathcal{P}) = \max(\psi(\mathbf{p}_0), \ldots, \psi(\mathbf{p}_{N-1}))$ by aggregating these pointwise features via max-pooling. ENetLite3D has the same structure except the first and the last building blocks; it extracts $F = 250$D features from 3D point clouds[4]. As seen above, ENetLite can adapt to different point dimensions by simply replacing the topmost FC layer, without compromising the overall computational cost.

Compared to ENet, ENetLite can handle point clouds of any size, and the number of parameters is independent from the

[2]To distinguish the models for 2D/3D planning, we suffix them with **-2D**/**-3D** when necessary. A fully-connected (FC) layer with m input and n output channels is denoted as $\mathrm{FC}(m, n)$, a 1D batch normalization with n channels as $\mathrm{BN}(n)$, a 1D max-pooling with window size n as $\mathrm{MaxPool}(n)$, and a dropout with a rate $p \in [0, 1)$ as $\mathrm{Dropout}(p)$, respectively.

[3]ENet3D is denoted as $\mathrm{FC}(6000, 784) \to \mathrm{ReLU} \to \mathrm{FC}(512) \to \mathrm{ReLU} \to \mathrm{FC}(256) \to \mathrm{ReLU} \to \mathrm{FC}(60)$.

[4]ENetLite2D is written as $\mathrm{BE}(2, 64, 64, 64, 128, 252)$, where $\mathrm{BE}(m, n) = \mathrm{FC}(m, n) \to \mathrm{BN}(n) \to \mathrm{ReLU}$ is a basic building block that maps mD point features into an nD space. In ENetLite3D, the first and the last building blocks are replaced with $\mathrm{BE}(3, 64)$ and $\mathrm{BE}(250)$.

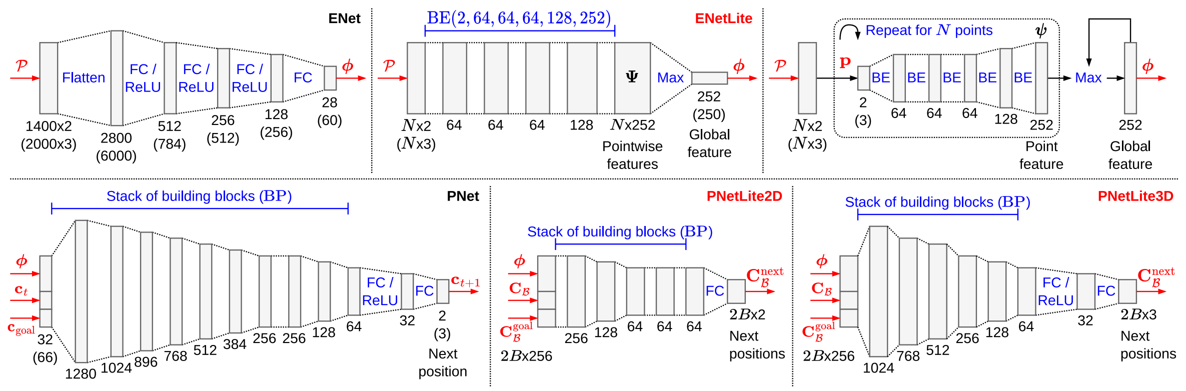

Fig. 6. Encoding and planning networks for MPNet and P3Net (top right: sequential feature extraction in Section V-A). BE is a shorthand for a building block consisting of a fully-connected layer, batch normalization, and ReLU activation. BP is a shorthand for a building block consisting of a fully-connected layer, ReLU activation, and a dropout with a probability of 0.5.

input size. ENetLite2D(3D) provides informative features with 9x (28/252) and 4.17x (60/250) more dimensions, while requiring 31.73x (1.60M/0.05M) and 104.47x (5.25M/0.05M) less parameters than ENet2D(3D). PointNet processes each point in a point cloud sequentially and thus avoids random accesses. In addition, as ENetLite involves only pointwise operations and a symmetric pooling function, its output $\phi(\mathcal{P})$ is invariant to the permutation of input points, leading to better training efficiency and robustness.

2) PNetLite: Lightweight Planning Network: The original PNet is formed by a set of building blocks, as shown in Fig. 6 (bottom left). It takes a concatenated input $[\phi(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\text{goal}}]$ consisting of an obstacle feature $\phi(\mathcal{P})$ passed from ENet, a current position $\mathbf{c}_t$, and a destination $\mathbf{c}_{\text{goal}}$, and computes the next position $\mathbf{c}_{t+1}$ which is one step closer to $\mathbf{c}_{\text{goal}}$. Notably, PNet2D/3D have the same set of hidden layers; the only difference is in the leading and trailing layers. PNet2D uses $\text{BP}(32, 1280)$ and $\text{FC}(2)$ to produce 2D coordinates from $F + 4 = 32$D inputs, whereas PNet3D uses $\text{BP}(66, 1280)$ and $\text{FC}(3)$ to handle $F + 6 = 66$D inputs and 3D outputs[5].

Such design has a problem of low parameter efficiency especially in the 2D case; PNet2D will contain redundant layers which do not contribute to the successful planning and only increase the inference time. The network architecture can be adjusted to the number of state dimensions and the complexity of planning tasks. In addition, as discussed in Section IV-B1, PointNet encoder provides permutation-invariant features that better capture the planning environment. Assuming that MPNet uses a larger PNet in order to compensate for the lack of robustness and geometric information in ENet-extracted features, it is reasonable to expect that PointNet allows the use of more shallower networks for path planning.

From the above considerations, P3Net employs planning networks with fewer building blocks, **PNetLite**. PNetLite2D (Fig. 6, bottom center) is composed of six building blocks[6] to compute a 2D position from a $F + 4 = 256$D input. As described in Section IV-A1, NeuralPlannerEx creates B pairs of forward-backward paths; PNetLite computes a batch of next positions $\mathbf{C}_{\mathcal{B}}^{\text{next}} \in \mathbb{R}^{2B \times 2}$ from an input matrix $[\phi(\mathcal{P}), \mathbf{C}_{\mathcal{B}}^{\text{next}}, \mathbf{C}_{\mathcal{B}}^{\text{goal}}] \in \mathbb{R}^{2B \times 256}$. PNetLite3D is obtained by removing a few blocks from PNet3D and replacing the first block with $\text{BP}(256, 1024)$ to process $F + 6 = 256$D inputs, as depicted in Fig. 6 (bottom right). PNetLite{2D, 3D} has 32.58x (3.76M/0.12M) and 2.35x (3.80M/1.62M) less parameters than PNet{2D, 3D}; {E, P}NetLite together achieves 32.32x and 5.43x parameter reduction in the 2D and 3D case. These two networks are jointly trained in an end-to-end supervised manner (Sections VI-A–VI-B).

V. Implementation

This section details the design and implementation of **P3NetCore**, a custom IP core for P3Net. It has three modules, namely, (1) **Encoder**, (2) **NeuralPlanner**, and (3) **CollisionChecker**, which cover most of the P3Net (Alg. 2) except the evaluation of path costs (lines 13, 17). The details are provided in the supplementary material.

A. Encoder Module

While the number of parameters is reduced, ENetLite takes a longer inference time than ENet (Table I, top), as it extracts local features for each point, which requires N forward passes. **Encoder** is to accelerate the inference.

Since the operations for each point is independent except the last max-pooling, the module computes a point feature $\psi(\mathbf{p}_i)$ one-by-one[7] and updates the global feature by taking a maximum $\phi(\mathcal{P}) \leftarrow \max(\phi(\mathcal{P}), \psi(\mathbf{p}_i))$ (Fig. 6, top right). In this way, **Encoder** can handle point clouds of any size, while the buffer size is constant regardless of N. **Encoder** consists of three kinds of submodules: $\text{FC}(m, n)$, $\text{BN-ReLU}(n)$, and Max. $\text{FC}(m, n)$ involves a matrix-vector product, and Max updates the feature by $\max(\phi(\mathcal{P}), \psi)$. $\text{BN-ReLU}(n)$ couples the batch normalization and ReLU. FC and BN-ReLU are parallelized by partially unrolling the loop and partitioning the

[5] $\text{BP}(m, n) = \text{FC}(m, n) \rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.5)$.
[6] PNetLite2D is denoted as $\text{BP}(256, 256, 128, 64, 64, 64, 2)$.

[7] It is possible to compute multiple point features in parallel, while the current design already gives satisfactory performance (Table I, top).

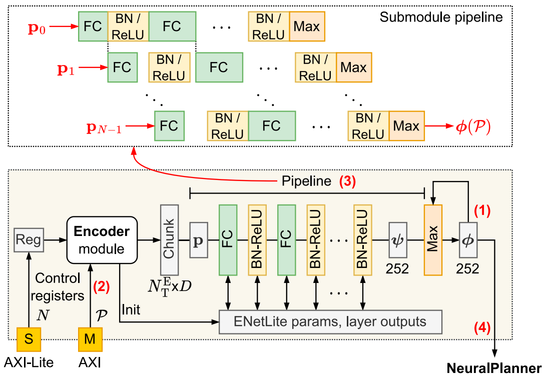

Fig. 7. Block diagram of **Encoder**.

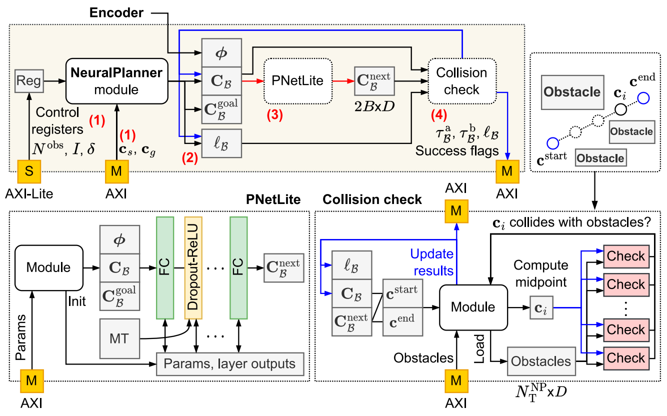

Fig. 8. Block diagram of **NeuralPlanner**.

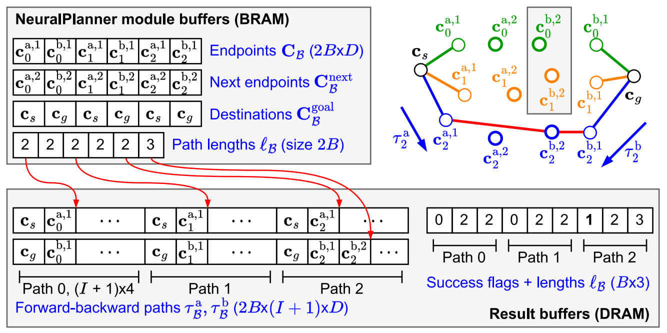
NeuralPlanner module buffers (BRAM)
Endpoints $\mathbf{C}_\mathcal{B}$ ($2BxD$)
Next endpoints $\mathbf{C}_\mathcal{B}^{\text{next}}$
Destinations $\mathbf{C}_\mathcal{B}^{\text{goal}}$
Path lengths $\ell_\mathcal{B}$ (size $2B$)
Path 0, $(I+1)$x4
Path 1
Path 2
Forward-backward paths $\tau_\mathcal{B}^{\text{a}}, \tau_\mathcal{B}^{\text{b}}$ ($2B$x$(I+1)$xD)
Success flags + lengths $\ell_\mathcal{B}$ (Bx3)
Result buffers (DRAM)

Fig. 9. Buffers for **NeuralPlanner** ($B = 3$). At each iteration, NeuralPlanner updates the path endpoints $\mathbf{C}_\mathcal{B}, \mathbf{C}_\mathcal{B}^{\text{next}}$, lengths $\ell_\mathcal{B}$, as well as the DRAM buffers for forward-backward paths $\tau_\mathcal{B}^{\text{a}} = [\tau_0^{\text{a}}, \tau_1^{\text{a}}, \tau_2^{\text{a}}], \tau_\mathcal{B}^{\text{b}} = [\tau_0^{\text{b}}, \tau_1^{\text{b}}, \tau_2^{\text{b}}]$. The path lengths stored on-chip are used as pointers to the DRAM path buffers (red arrows). In iteration $i = 2$, the third path pair $\tau_2^{\text{a}}, \tau_2^{\text{b}}$ is found to be connectable. The module writes the success flags and path lengths $\ell_\mathcal{B}$ to the DRAM buffer and completes the task.

relevant buffers. Besides, a dataflow optimization is applied for the fully-pipelined execution of these submodules (Fig. 7, top).

Fig. 7 (bottom) shows the block diagram. Upon a request from the host CPU, it first (1) initializes the FD output feature $\phi(\mathcal{P})$ with zeros. Then, the input $\mathcal{P}$ is processed in tiles of size N_{T}^{E} as follows: the module (2) transfers a tile from DRAM to a point buffer of size $(N_{\text{T}}^{\text{E}}, D)$, and (3) the submodule pipeline consumes points in this buffer one-by-one to update the output. After repeating this process for all $\lceil N/N_{\text{T}}^{\text{E}} \rceil$ tiles, (4) the output $\phi(\mathcal{P})$ is written to the on-chip buffer of size $(2B, F + 2D)$ for later use in **NeuralPlanner**. All parameters and layer outputs are stored on-chip, which avoids DRAM accesses during computation.

B. NeuralPlanner Module

As apparent in Algs. 1–2, the bidirectional neural planning (NeuralPlannerEx) is at the core of P3Net and thus has a significant impact on the overall performance. In contrast to the previous work [34], which only implements PNet inference on the custom IP core, **NeuralPlanner** covers the entire NeuralPlannerEx algorithm (Alg. 3).

1) PNetLite Inference and Collision Checking: For PNetLite inference, the module contains two types of submodules: $\text{FC}(m, n)$ and $\text{Dropout-ReLU}(p)$ which fuses ReLU and dropout into a single pipelined loop. $\text{FC}(m, n)$ exploits the fine-grained (data-level) parallelism as described in Section V-A, while $\text{Dropout-ReLU}(p)$ $(p = 0.5)$ is implemented with Mersenne-Twister (MT). Note that in the 3D case, parameters for the second building block $\text{BP}(1024, 768)$ are kept on the DRAM due to limited on-chip memory, which necessitates a single sweep (burst transfer) of weight and bias parameters during inference.

In addition to PNetLite, the module deals with collision checking. It adopts a simple approach based on the discretization [10], [31]: the line between $\mathbf{c}^{\text{start}}, \mathbf{c}^{\text{end}}$ is split into segments with a predefined interval δ (Fig. 8, right top), producing equally spaced midpoints $\mathbf{c}_0, \ldots, \mathbf{c}_M$[8]. If any midpoint collides with any obstacle, the line is in collision. To simplify the implementation, the module assumes that each obstacle is rectangular and represented as a bounding box with minimum and maximum corner points $\mathbf{c}_{i,\min}^{\text{obs}}, \mathbf{c}_{i,\max}^{\text{obs}}$. The module contains a total of $P_{\text{Chk}}^{\text{NP}} \cdot P_{\text{ChkP}}^{\text{NP}}$ Check submodules to test $P_{\text{ChkP}}^{\text{NP}}$ midpoints against $P_{\text{Chk}}^{\text{NP}}$ obstacles in parallel (Fig. 8, right bottom). The interval δ should be set small enough to test a line $(\mathbf{c}^{\text{start}}, \mathbf{c}^{\text{end}})$ with a finer resolution and ensure reliable collision avoidance. The false negatives, i.e., lines being misclassified as collision-free, would cause the planner to produce an unfeasible path, and a robot may collide with obstacles as a result.

2) Processing Flow of NeuralPlanner Module: As shown in Figs. 8–9, the module consists of submodules and several buffers to perform the bidirectional planning. The module first (1) reads the task configurations such as start-goal points $\mathbf{c}_s, \mathbf{c}_g$ from DRAM, as well as algorithmic parameters (e.g., the maximum iterations I) from control registers. It then (2) initializes BRAM buffers (Fig. 9, top) for the current endpoints, destinations, next waypoints $\mathbf{C}_\mathcal{B}, \mathbf{C}_\mathcal{B}^{\text{goal}}, \mathbf{C}_\mathcal{B}^{\text{next}} \in \mathbb{R}^{2B \times D}$ $(D = 2, 3)$, and path lengths $\ell_\mathcal{B} \in \mathbb{N}^{2B}$, as well as the result buffers on DRAM (Alg. 3, lines 2-6). These result buffers store the entire forward-backward paths $\tau_\mathcal{B}^{\text{a}}, \tau_\mathcal{B}^{\text{b}}$ along with their lengths and success flags in a format depicted in Fig. 9 (bottom).

The module proceeds to alternate between PNetLite inference (line 8) and collision checking (lines 9-17); it (3) sets

[8] $\mathbf{c}_i = \mathbf{c}^{\text{start}} + (i/M)\boldsymbol{\Delta}$, $\boldsymbol{\Delta} = \mathbf{c}^{\text{end}} - \mathbf{c}^{\text{start}}$, $M = \|\boldsymbol{\Delta}\| / \delta$.

PNetLite inputs $\mathbf{C}_{\mathcal{B}}, \mathbf{C}_{\mathcal{B}}^{\text{goal}}$ and computes $\mathbf{C}_{\mathcal{B}}^{\text{next}}$ (Fig. 8, left bottom). Using the current and next endpoints $\mathbf{C}_{\mathcal{B}}, \mathbf{C}_{\mathcal{B}}^{\text{next}}$, the module (4) attempts to connect each path pair $\tau_j^{\text{a}}, \tau_j^{\text{b}}$ $(j \in [1, B))$ (Fig. 5). If the path connection succeeds, it transfers the success flag and path length $(\ell_j^{\text{a}}, \ell_j^{\text{b}} \in \ell_{\mathcal{B}})$ to the DRAM result buffer and completes the task. The module appends new waypoints $\mathbf{C}_{\mathcal{B}}^{\text{next}}$ to the DRAM result buffers, increments path lengths $\ell_{\mathcal{B}}$ accordingly, and replaces $\mathbf{C}_{\mathcal{B}}$ with $\mathbf{C}_{\mathcal{B}}^{\text{next}}$ for the next iteration. The collision check of a line segment is cast as checks on the discrete midpoints $\{\mathbf{c}_i\}$. The module reads a tile of bounding boxes $\{(\mathbf{c}_{i,\min}^{\text{obs}}, \mathbf{c}_{i,\max}^{\text{obs}})\}$ into an on-chip obstacle buffer of size $(N_{\text{T}}^{\text{NP}}, 2, D)$, and tests $P_{\text{ChkP}}^{\text{NP}}$ midpoints with $P_{\text{Chk}}^{\text{NP}}$ obstacles in parallel using an array of Check submodules (Fig. 8, right bottom).

C. CollisionChecker Module

Since collision checking is performed throughout P3Net, it is implemented in a dedicated **CollisionChecker** module for further speedup. It checks whether the path $\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}$ is in collision by repeating the process described in Section V-B1 for each edge $(\mathbf{c}_i, \mathbf{c}_{i+1})$. Similar to **NeuralPlanner**, it contains $P_{\text{Chk}}^{\text{CC}} \cdot P_{\text{ChkP}}^{\text{CC}}$ Check submodules to test $P_{\text{ChkP}}^{\text{CC}}$ midpoints on the edge $(\mathbf{c}_i, \mathbf{c}_{i+1})$ against $P_{\text{Chk}}^{\text{CC}}$ obstacles in parallel. The obstacle bounding boxes are transferred from DRAM to an on-chip buffer of size $(N_{\text{T}}^{\text{CC}}, 2, D)$ as necessary. After completion, the result is written to the control register (1 if τ collides with any obstacle).

D. Design Space Exploration for P3NetCore

Following [37], we conduct a design space exploration to find a parameter set that gives the best performance under FPGA resource and bandwidth constraints. Due to page limit, we briefly describe the performance model for each module, which is defined as:

$$\text{Perf} = \min\left(N_{\text{op}}/L \cdot f, \text{AI} \cdot \text{BW}\right) \ \text{(ops/s)}, \tag{1}$$

where an arithmetic intensity $\text{AI} = N_{\text{op}}/D_{\text{all}}$ is a ratio of the number of operations (OPs) N_{op} to the bytes D_{all} transferred from/to DRAM. L, f, BW denote the latency (cycles), clock frequency (Hz), and the maximum bandwidth (bytes/s), respectively[9]. Here we consider the usages of BRAM, URAM, and DSP, which are the limiting resources for our design (Table II). BRAM usage is modeled as [38]:

$$\begin{aligned} &R_{\text{B}}(s, w, \text{PF}) \\ &\quad = \text{PF} \cdot \lceil s/(\text{PF} \cdot \lceil w/36 \rceil \cdot (18 \cdot 10^3)) \rceil \cdot \lceil w/36 \rceil, \end{aligned} \tag{2}$$

where s, w, PF are the required buffer size (bits), bit-width, and partition factor[10]. URAM model is similar to Eq. 2. DSP consumption mainly comes from the MAC operations in layer submodules (Sections V-A and V-B1)[11].

For **Encoder**, N_{op}, L, and D_{all} are modeled as:

$$\begin{aligned} N_{\text{op}}^{\text{E}} &= N \cdot \left(\textstyle\sum_i N_{\text{op}}^{\text{E},i}\right), \ D_{\text{all}}^{\text{E}} = N \cdot (128/8) \\ L^{\text{E}} &= \left\lceil \frac{N}{N_{\text{T}}^{\text{E}}} \right\rceil \cdot \left(L^{\text{E},\max}\left(N_{\text{T}}^{\text{E}} - 1\right) + \textstyle\sum_i L^{\text{E},i} + L_{\text{P}}\left(N_{\text{T}}^{\text{E}}\right)\right), \end{aligned} \tag{3}$$

where $N_{\text{op}}^{\text{E},i}/L^{\text{E},i}$ denotes the OPs/latency for the i-th pipeline stage (Fig. 7, top)[12]. $D_{\text{all}}^{\text{E}}$ is straightforward because only the input points are transferred. $L_{\text{P}}(n) = Dn + L_{\text{Req}}$ $(D = 2, 3)$ models the latency for transferring n points from DRAM, and L_{Req} is a latency for the read request. L^{E} is determined by the longest stage $L^{\text{E},\max} = \max_i(L^{\text{E},i})$. To reduce design space, only an unroll factor for the longest pipeline stage, P^{E}, is considered, and ones for the other stages are adjusted accordingly to balance the latency; thus, **Encoder** has two design parameters: $N_{\text{T}}^{\text{E}}, P^{\text{E}}$.

The performance model of **NeuralPlanner** is follows:

$$\begin{aligned} N_{\text{op}}^{\text{NP}} &= \bar{I}\left(N_{\text{op}}^{\text{NP,Net}} + N_{\text{op}}^{\text{NP,Chk}}\right), \ N_{\text{op}}^{\text{NP,Net}} = \textstyle\sum_i N_{\text{op}}^{\text{NP},i} \\ N_{\text{op}}^{\text{NP,Chk}} &= 3BN_{\text{op}}^{\text{Chk}}(\bar{d}, N^{\text{obs}}) \\ L^{\text{NP}} &= \bar{I}\left(L^{\text{NP,Net}} + L^{\text{NP,Chk}}\right), \ L^{\text{NP,Net}} = \textstyle\sum_i L^{\text{NP},i} \\ L^{\text{NP,Chk}} &= B\left\lceil \frac{N^{\text{obs}}}{N_{\text{T}}^{\text{NP}}} \right\rceil \cdot \left(L_{\text{O}}(N_{\text{T}}^{\text{NP}})\right. \\ &\qquad \left. + 3L^{\text{Chk}}\left(\bar{d}, N_{\text{T}}^{\text{NP}}, P_{\text{Chk}}^{\text{NP}}, P_{\text{ChkP}}^{\text{NP}}\right)\right) \\ D_{\text{all}}^{\text{NP}} &= (2B + 2\bar{I}BN^{\text{obs}} + 2\bar{I}B + B) \cdot (128/8), \end{aligned} \tag{4}$$

where $N^{\text{obs}}, \bar{I}$ denote the number of obstacles and an empirical average of the iterations. $N_{\text{op}}^{\text{NP},\{\text{Net,Chk}\}}/L^{\text{NP},\{\text{Net,Chk}\}}$ denotes the OPs/latency for the PNetLite inference and collision checking. $N_{\text{op}}^{\text{NP,Net}}/L^{\text{NP,Net}}$ is computed by summing up the OPs/latency in each layer (Fig. 8, bottom left)[13]. Similar to **Encoder**, only an unroll factor $P_{\text{Net}}^{\text{NP}}$ for the largest FC layer is considered, and ones for the other layers are adjusted accordingly. $N_{\text{op}}^{\text{Chk}}(d, n)/L^{\text{Chk}}(d, n, P_0, P_1)$ denotes the OPs/latency to check the collision between n obstacles and an edge of length d, with an interval of δ (Section V-B1). They are modeled as $(d/\delta) \cdot 2Dn$ and $(d/(\delta \cdot P_1))(L_0 + L_1 \cdot \lceil n/P_0 \rceil)$, where d/δ is the number of midpoints on the edge, P_0, P_1 the number of Check submodules, L_0 the latency to compute the next midpoint, and L_1 the latency to check each midpoint. Since d depends on the input, its empirical average $\bar{d}$ is used instead for performance modeling. $L_{\text{O}}(n) = 2Dn + L_{\text{Req}}$ is a latency

[9]Since P3NetCore is connected to DRAM via a single 128-bit AXI port (Fig. 8), we set to $f = 200$MHz and $\text{BW} = f \cdot 128$bit $= 3.2$GB/s, assuming that the 128-bit data arrives at every clock cycle.

[10]A single BRAM block can store up to 18Kb of data and the maximum bit-width is 36.

[11]For instance, FC submodule in **Encoder** and **NeuralPlanner** consumes $2P$ and $2BP$ DSPs, respectively, given an unroll factor P.

[12]For instance, the module $\text{FC}(m, n)$ involves $n(2m+1)$ OPs and consumes around mn/P cycles, given that the loop over the output dimension n is unrolled with a factor of P.

[13]For instance, $\text{FC}(m, n)$ involves $2Bn(2m+1)$ OPs and takes around mn/P cycles, where P is an unroll factor on the dimension n (the loop on the batch dimension is always fully unrolled). In the 3D case, weight parameters for the second FC layer are stored on DRAM; it takes $n(m/4)$ cycles to read weight parameters from DDR (four values per cycle) and compute outputs.

for transferring n obstacles from DRAM. $D_{\text{all}}^{\text{NP}}$ consists of the terms for start-goal positions, obstacles, paths, and planning results (Fig. 9); in the 3D case, it includes the parameter size for the second FC layer as well (≈ 3.15MB). Similar to Eq. 4, **CollisionChecker** is modeled as follows:

$$\begin{aligned} N_{\text{op}}^{\text{CC}} &= N_{\text{op}}^{\text{Chk}}(\bar{\tau}, \delta, N^{\text{obs}}) \\ D_{\text{all}}^{\text{CC}} &= \left(2N^{\text{obs}} + \left\lceil \frac{N^{\text{obs}}}{N_{\text{T}}^{\text{CC}}} \right\rceil \bar{T}\right) \cdot (128/8) \\ L^{\text{CC}} &= \left\lceil \frac{N^{\text{obs}}}{N_{\text{T}}^{\text{CC}}} \right\rceil \cdot \left(L_{\text{O}}\left(N_{\text{T}}^{\text{CC}}\right)\right. \\ &\quad \left. + L^{\text{Chk}}\left(\bar{\tau}, N_{\text{T}}^{\text{CC}}, P_{\text{Chk}}^{\text{CC}}, P_{\text{ChkP}}^{\text{CC}}\right)\right), \end{aligned} \tag{5}$$

where $\bar{T}$ and $\bar{\tau}$ denote the empirical average length and distance of the input path τ (Section V-C).

For **Encoder**, $N_{\text{op}}^{\text{E}} \approx N \cdot 100$Kops and AI^{E} reaches 6.25Kops/bytes (6.23Kops/bytes) in the 2D (3D) case, independent of N. From Eq. 3, the bandwidth during transferring points is estimated to be 1.26GB/s (3D: 1.06GB/s). The AI of **NeuralPlanner** increases with a batch size; $N_{\text{op}}^{\text{NP}}$ and AI^{NP} are 1.12–8.95Mops and 1.18–5.25Kops/bytes (3D: 13.53–108.26Mops and 4.30–34.36ops/bytes) for $B = [1, 8]$. The higher AI in the 2D case is because all network parameters are stored on-chip. In **CollisionChecker**, $N_{\text{op}}^{\text{CC}}$ and AI^{CC} are 85.74Kops and 178.23ops/bytes (3D: 179.88Kops and 373.16ops/bytes). The effective bandwidth is around 1.04–1.26GB/s (3D: 0.79–0.90GB/s) when transferring obstacle and path information. While it is lower than the theoretical maximum BW, it can be easily improved by using more available AXI ports. For these three modules, we observe that the first term in Eq. 1 is up to 545x, 1302x, and 179x (3D: 1078x, 5.26x, and 373x) lower than the second term due to the small amount of data transfer, meaning that they are compute-bound. The performance is limited by the amount of onboard computing resources rather than the memory bandwidth. For this reason, we focus on minimizing the weighted latency L_{total} under FPGA resource constraints by conducting an exhaustive search:

$$L_{\text{total}} = \bar{N}_{\text{call}}^{\text{E}} L^{\text{E}} + \bar{N}_{\text{call}}^{\text{NP}} L^{\text{NP}} + \bar{N}_{\text{call}}^{\text{CC}} L^{\text{CC}}, \tag{6}$$

where $\bar{N}_{\text{call}}^{\{\text{E,NP,CC}\}}$ is an average number of calls to the respective module during planning. These empirical values are obtained by running P3Net with P3Net2D/3D dataset (Section VI-B). For instance, we use the design point $(N_{\text{T}}^{\text{E}}, P^{\text{E}}, P_{\text{Net}}^{\text{NP}}, P_{\text{Chk}}^{\text{NP}}, P_{\text{ChkP}}^{\text{NP}}, P_{\text{Chk}}^{\text{CC}}, P_{\text{ChkP}}^{\text{CC}}) = (128, 64, 16, 4, 8, 8, 8)$ (3D: $(2048, 64, 8, 8, 8, 2, 8)$) for ZCU104 ($B = 4$). More details are provided in the supplementary material.

E. Implementation Details

We used Xilinx Vitis HLS 2022.1 to develop the IP core, and Vivado 2022.1 for synthesis and place-and-route. Two variants of P3NetCore were created for 2D and 3D planning. The board-level implementation is depicted in the supplementary material. Xilinx ZCU104 is used as a target device, and the clock frequency is set to 200MHz. To preserve the precision of PNetLite outputs (i.e., waypoint coordinates), **Encoder** and **NeuralPlanner** employ 32-bit (16.16) fixed-point format for layer outputs, and 24-bit (8.16) fixed-point for model parameters. Besides, collision checking is performed using 32-bit floating-point, taking into account that the interval δ (Section V-B1) is set sufficiently small to prevent false negatives. Following the previous work [34], we create another IP core for P3Net, P3NetCore-NN, that only implements the {E, P} NetLite inference.

VI. Evaluation

This section evaluates the proposed P3Net to demonstrate the improvements on success rate, speed, quality of solutions, and power efficiency. For convenience, P3Net accelerated by the proposed IP core is referred to as **P3Net-FPGA**.

A. Experimental Setup

MPNet and the famous sampling-based methods, RRT* [7], Informed-RRT* (IRRT*) [8], BIT* [9], and ABIT* [12] were used as a baseline. All methods including P3Net were implemented in Python using NumPy and PyTorch. The implementation of RRT*, Informed-RRT*, and BIT* is based on the open-source code [39][14]. We implemented an ABIT* planner following the algorithm in the paper [12]. For MPNet, we used the code from the authors [10] as a reference; MPNet path planner and the other necessary codes for training and testing were rewritten from scratch. The experiments were conducted on a workstation (with Intel Xeon W-2235 (3.8GHz) and Nvidia GeForce RTX 3090), Nvidia Jetson {Nano, Xavier NX}, and Xilinx ZCU104 (see the supplementary material). Following the MPNet paper [10], we trained models in an end-to-end supervised manner, which is detailed in the supplementary material as well. The maximum number of iterations I in NeuralPlanner(Ex) (Algs. 1, 3) is set to 50[15]. RRT* and Informed-RRT* were configured with a maximum step size of 1.0 and a goal bias of 0.05. BIT* and ABIT* were executed with a batch size of 100 and Euclidean distance heuristic. The collision checking interval δ is set to 0.01.

B. Path Planning Datasets

For evaluation, we used a publicly-available dataset for 2D/3D path planning provided by the MPNet authors [10], referred to as **MPNet2D/3D**[16]. It is split into one training and two testing sets (**Seen**, **Unseen**); the former contains 100 workspaces, each of which has a point cloud $\mathcal{P}$ representing obstacles, and 4000 planning tasks with randomly generated start-goal points and their respective ground-truth paths. **Seen** set contains the same 100 workspaces as the training set, but

[14]For a fair performance comparison, we replaced a brute-force linear search with an efficient K-D tree-based search using Nanoflann library (v1.4.3) [40]. In BIT* planner, we used a priority queue to pick the next edge to process, which avoids searching the entire edge queue, as mentioned in the original paper [9].

[15]We find that the average is around 2. We thus set the maximum of I to 50 throughout the evaluation and consider the other algorithmic parameters, e.g., I_{Replan}, I_{Refine}, etc.

[16]Originally called Simple2D and Complex3D in the MPNet paper.

with each having 200 new planning tasks; **Unseen** set contains ten new workspaces not observed during training, each of which has 2000 planning tasks.

Each workspace is a square (or cube) of size 40 containing randomly-placed seven square obstacles of size 5 (or ten cuboid obstacles with a side length of 5 and 10). Note that, trivial tasks in the testing sets are excluded, where start-goal pairs can be connected by straight lines and obstacle avoidance is not required. We only used the first 20/200 tasks for each workspace in **Seen/Unseen** dataset. As a result, the total number of planning tasks is 945/892 and 740/756 in MPNet2D (Seen/Unseen) and MPNet3D (Seen/Unseen) datasets, respectively. Both MPNet and P3Net models were trained with MPNet2D/3D training sets.

In addition, we generated **P3Net2D/3D** dataset for testing (Figs. 1–2), which contains 100 workspaces, with 20 planning tasks for each. Compared to MPNet2D/3D, the number of obstacles is doubled to simulate more challenging tasks.

C. Planning Success Rate

First, the tradeoff between planning success rate and the average computation time per task is evaluated on the workstation with GPU acceleration.

1) Comparison of Encoding and Planning Networks: MPNet is executed under three combinations of models, i.e., {E, P}Net (original), ENetLite-PNet (**ELite**), and {E, P}NetLite (**EPLite**), with a varying number of replan iterations $I_{\text{Replan}} = \{10, 20, 50, 100\}$. The refinement step is not performed in P3Net ($I_{\text{Refine}} = 0$). The results on MPNet and P3Net test datasets are shown in Fig. 10 (left).

In the 2D case (1st/3rd row), replacing {E, P}Net with {E, P}NetLite yields substantially higher success rate and even faster computation time, while reducing the parameters by 32.32x (Section IV-B2). For $I_{\text{Replan}} = 10$, MPNet-ELite is 15.58% (69.17/84.75%) and 23.75% (45.15/68.90%) more successful than MPNet-original on MPNet-Unseen and P3Net datasets. PNetLite further improves the success rate by 3.48% (84.75/88.23%) and 4.45% (68.90/73.35%). MPNet-EPLite is 1.40x (0.067/0.048s) and 1.12x (0.131/0.117s) faster than MPNet-original, indicating that the proposed models help MPNet algorithm to quickly find a solution in a less number of replan attempts. This empirically validates the discussion in Section IV-B1 that the shallower PNet is sufficient since the PointNet encoder provides permutation-invariant features. Considering the success rate improvements (19.06/28.20%) in these two datasets, the proposed models offer greater performance advantages in more difficult problem settings.

In the 3D case (2nd/4th row, left), MPNet-EPLite gives the similar success rate as MPNet-original, while achieving 5.43x parameter reduction (Section IV-B2). ENetLite improves the success rate by 1.46% (89.82/91.27%) and 3.70% (79.35/83.05%), whereas PNetLite slightly lowers it by 0.40% (91.27/90.87%) and 3.95% (83.05/79.10%) on MPNet-Unseen and P3Net datasets. This performance loss is compensated by the P3Net planner. Comparing the results from MPNet- Seen / Unseen datasets (dashed/solid lines), the difference in the

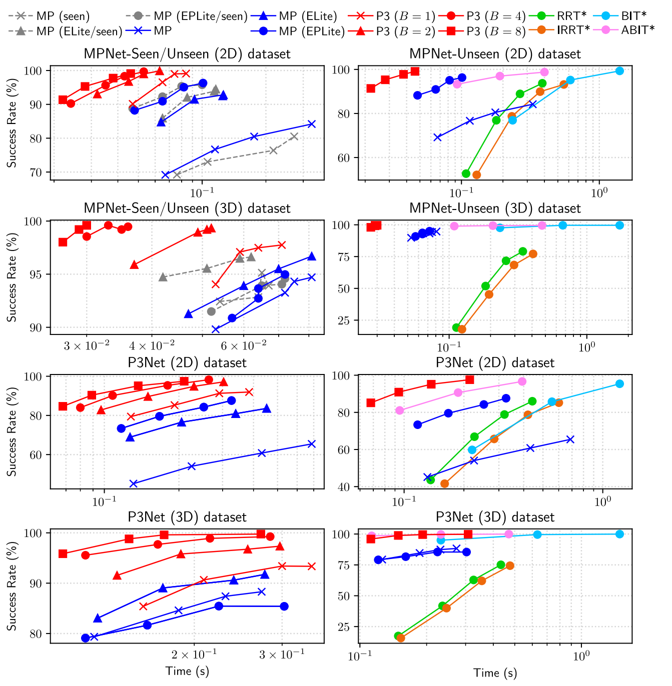

Fig. 10. Success rate and computation time tradeoffs (measured on the workstation with GPU acceleration). Left: comparison of MPNet and P3Net. Right: comparison of MPNet, P3Net, and sampling-based methods. Upper left is better.

success rate is at most 3.46%, which confirms that our proposed models generalize to workspaces that are not observed during training. We compare the P3Net success rate under varying feature dimensions F in ENetLite. The result is presented in the supplementary material.

2) Comparison of P3Net With MPNet: Fig. 10 (left) highlights the advantage of P3Net over MPNet: P3Net success rate gradually improves with increasing I_{Replan} and reaches nearly 100%.[17] While MPNet shows a similar trend, P3Net consistently outperforms MPNet[18]. In the 2D case (1st/3rd row), compared to MPNet-EPLite, P3Net ($B = 8$) is 2.80% (96.30/99.10%) and 9.85% (87.60/97.45%) more successful and is 2.20x (0.101/0.046s) and 1.55x (0.326/0.210s) faster on MPNet-Unseen and P3Net datasets ($I_{\text{Replan}} = 100$). While MPNet shows a noticeable drop in success rate (96.30/87.69%) when tested on P3Net dataset, P3Net only shows a 1.65% drop (99.10/97.45%) and maintains the high success rate in a more challenging dataset. In the 3D case (2nd/4th row), it is 2.91% (96.69/99.60%) and 8.00% (91.75/99.75%) better, and runs 2.70x (0.081/0.030s) and 1.02x (0.277/0.272s) faster.

[17]In NeuralPlanner(Ex), path waypoints are sampled from a learned distribution; since it is defined on a high-dimensional latent feature space $\phi(\mathcal{P})$, the theoretical analysis of P3Net (e.g., how to determine the optimal value of I_{Replan}) is challenging and thus is out of scope of this work. While it is not guaranteed that the success rate reaches one as $I_{\text{Replan}} \to \infty$, Fig. 10 shows a promising result. One simple approach to guarantee this property is to employ a classical planner (e.g., RRT*) when NeuralPlanner fails to produce a detour in the replanning process (Alg. 1, line 29).

[18]Note that the success rate of P3Net ($B = 1$) surpasses that of MPNet-EPLite, since the former performs more collision checks to connect a pair of forward-backward paths in each iteration (Alg. 1, line 21 and Alg. 3, lines 9-17).

Notably, increasing the batch size B improves both success rate and speed, which clearly indicates the effectiveness of the batch planning strategy (Section IV-A1). On P3Net2D dataset (3rd row), P3Net with $B, I_{\mathrm{Replan}} = 8, 10$ is 5.25% more successful and 1.88x faster than with $B = 1$ (79.40/84.65%, 0.128/0.068s). The number of initial planning attempts I_{Init} affects the performance as well; increasing it from 1 to 5 yields a 2.85% better success rate on P3Net2D dataset ($I_{\mathrm{Replan}} = 10$). We confirm that P3Net-FPGA and P3Net achieve similar success rates. More evaluations are presented in the supplementary material.

3) Comparison With Sampling-Based Methods: Fig. 10 (right) plots the results from sampling-based methods. The number of iterations is set to $\{200, 300, 400, 500\}$ for RRT*/IRRT*, and $\{50, 100, 200\}$ for BIT*/ABIT*. Sampling-based methods exhibit a higher success rate with increasing iterations; they are more likely to find a solution as they place more random nodes inside a workspace and build a denser tree. While MPNet-GPU is as fast as the sampling-based methods and {E, P}NetLite leads to higher success rates, it is still less successful than ABIT*. Compared to that, P3Net achieves comparable or even better success rates than ABIT* by planning multiple paths simultaneously (i.e., setting $B = 8$), and outperforms the other methods. On P3Net2D dataset (Fig. 10 (right), 3rd row), MPNet-original and MPNet-EPLite ($I_{\mathrm{Replan}} = 100$) plan in 0.687/0.326s on average and are 65.45/87.60% successful. P3Net ($B, I_{\mathrm{Replan}} = 8, 100$) improves the result to 0.214s and 97.60%, making it 1.84/5.69x faster and 0.95/2.20% better than ABIT*/BIT* (200 iterations). This demonstrates that both the batch planning strategy (Section IV-A1) and model architecture improvement (Section IV-B) contribute to better performance and hence are equally important.

D. Computation Time

Fig. 11 visualizes the distribution of computation time measured on various platforms. The sampling-based methods were run on the CPU. On Nvidia Jetson, MPNet and P3Net were executed with GPU. WS {CPU, GPU} refers to the workstation with and without GPU acceleration. On the basis of results from Section VI-C, hyperparameters of the planners were selected to achieve similar success rates.

P3Net-FPGA is faster than the other planners on ZCU104 and Jetson in most cases, and its median computation time is below 0.1s. In the 2D case (Fig. 11, left), it even outperforms sampling-based methods on the WS CPU, and is comparable to MPNet/P3Net on the WS GPU. The performance advantage of P3Net-FPGA comes from that the entire planning algorithm is implemented on the dedicated IP core, which (i) effectively speeds up both {E, P}Net inference and collision checking and (ii) reduces the data transfer overhead. In the 3D case (Fig. 11, right), while MPNet on the WS-GPU looks faster than P3Net-FPGA, it only solves easy planning tasks that require less replan trials, and gives more than 10% lower success rate. We observe a significant reduction of the variance in computation time. On the ZCU104 and P3Net dataset, MPNet solves a task in 7.623±19.987s on average, which is improved to 4.651±8.909s and 0.093±0.092s in P3Net-{CPU, FPGA}.

1) Path Planning Speedup: Fig. 12 shows the performance gain of P3Net-FPGA over the other planners. We compared the sum of execution times for successful planning tasks. In the 2D case, P3Net-FPGA is the fastest among the methods considered.

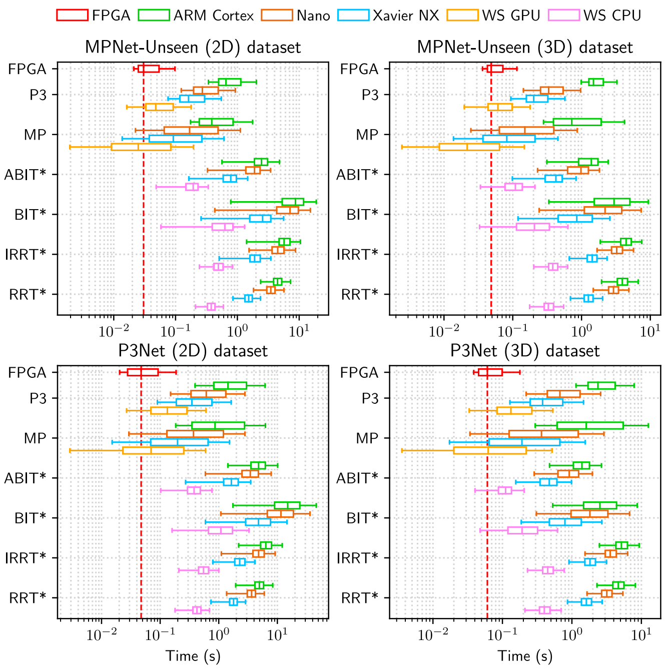

Fig. 11. Computation time distribution. Hyperparameters are as follows. P3Net2D: $(B, I_{\mathrm{Init}}, I_{\mathrm{Replan}}, I_{\mathrm{Refine}}) = (4, 5, 50, 5)$, P3Net3D: $(B, I_{\mathrm{Init}}, I_{\mathrm{Replan}}, I_{\mathrm{Refine}}) = (4, 5, 20, 5)$, MPNet2D/3D: {E, P}NetLite, $I_{\mathrm{Replan}} = 100$, ABIT*/BIT* (2D): 100 (200) iterations for MPNet (P3Net) dataset, ABIT*/BIT* (3D): 50 iterations, IRRT*/RRT* (2D/3D): 500 iterations. The red dashed line is a median time of P3Net-FPGA.

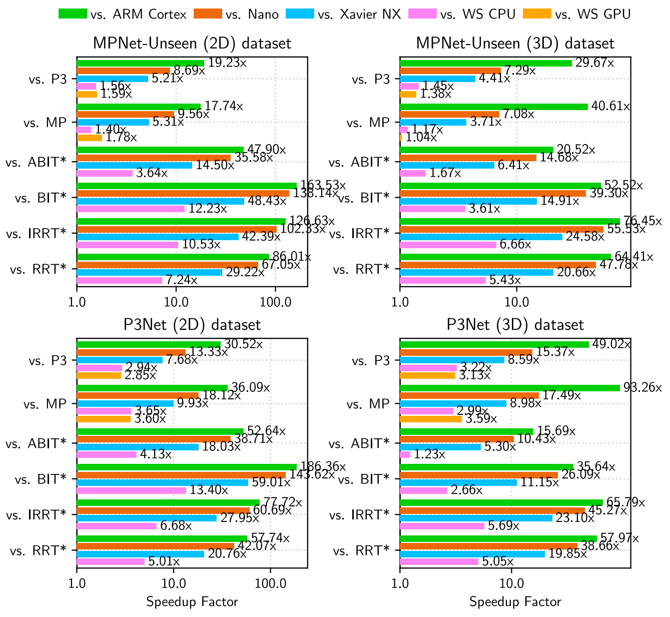

Fig. 12. Average speedup factors. Hyperparameter settings are the same as in Fig. 11.

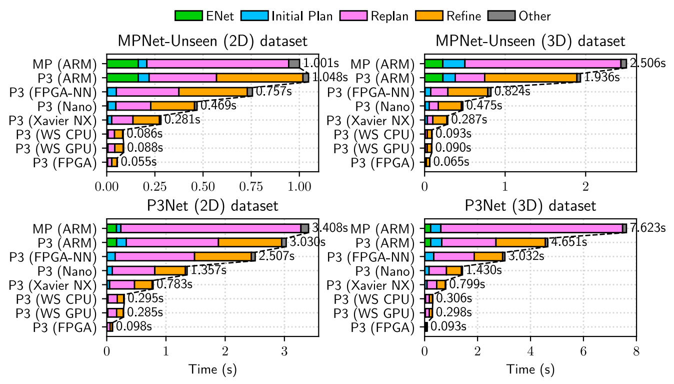

Fig. 13. Computation time breakdown. Hyperparameter settings are the same as in Fig. 11.

On P3Net dataset (bottom left), it achieves 30.52–186.36x, 13.33–143.62x, 7.68–59.01x, and 2.85–13.40x speedups over the ARM Cortex CPU, Jetson Nano, Jetson Xavier NX, and a workstation, respectively. Offloading the entire planning algorithm to the dedicated IP core eliminates unnecessary data transfers and brings more performance benefits than using highend CPUs. P3Net-FPGA completes more planning tasks in a shorter period of time than MPNet (e.g., 8.05% higher success rate and 3.60x speedup than MPNet on WS GPU), despite that it also performs an extra refinement phase ($I_{\mathrm{Refine}} = 5$). Additionally, while P3Net-FPGA shows a 1.49x speedup over MPNet (WS GPU) in terms of median time (Fig. 11, bottom left), the total execution time is reduced by 3.60x (Fig. 12, bottom left). This implies P3Net solves challenging tasks much faster than MPNet, thanks to the improved algorithm and model architecture. P3Net offers performance advantages in a more challenging dataset, considering that it runs 17.74x/36.09x faster than MPNet-CPU on ZCU104 for MPNet/P3Net dataset. P3Net-FPGA mostly outperforms the other planners in the 3D case as well. On P3Net3D dataset (bottom right), it provides 15.69–93.26x, 10.43–45.27x, 5.30–23.10x, and 1.23–5.69x speedups compared to the ARM Cortex CPU, Jetson Nano, Jetson Xavier NX, and a workstation, respectively.

2) Computation Time Breakdown: The computation time breakdown of MPNet/P3Net is summarized in Fig. 13. FPGA-NN refers to P3Net with P3NetCore-NN (i.e., only the inference is accelerated). P3NetCore effectively reduces the execution time of all three phases. On P3Net2D dataset (bottom left), the replanning phase took 3.041s and accounted for 89.24% of the entire execution time in MPNet, which is almost halved to 1.557s (51.37%) in P3Net (ARM), and is brought down to 0.052s (53.53%) in P3Net-FPGA. The saved time can be used to perform the additional refinement step and improve the quality of solutions. As seen from the results of WS {CPU, GPU}, GPU acceleration of the DNN inference only slightly improves the overall performance; since MPNet/P3Net alternates between collision checks on CPU and PNet inference on GPU, the frequent data transfer undermines the performance gain. Similarly, P3Net (FPGA-NN) is only 1.21/1.53x faster than P3Net (ARM) on P3Net2D/3D dataset, as collision checking runs slower on the host CPU. Only offloading the collision checking

TABLE I
LATENCY FOR INFERENCE (TOP) AND COLLISION CHECKING (BOTTOM) ON ZCU104

	2D			3D		
Model	N	CPU (ms)	**IP** (ms)	N	CPU (ms)	**IP** (ms)
ENet	1400	43.25	–	2000	146.40	–
ENetLite	1400	163.70	3.40	2000	229.85	4.67
	2048	242.05	4.87	4096	478.06	9.31
	4096	477.47	9.52	8192	950.91	18.40
Model	B	CPU (ms)	**IP** (ms)	B	CPU (ms)	**IP** (ms)
PNet	1	102.77	–	1	103.22	–
	2	107.79	–	2	108.31	–
	4	130.41	–	4	130.68	–
PNetLite	1	3.56	0.319	1	44.48	1.74
	2	4.02	0.321	2	46.85	1.78
	4	4.72	0.392	4	56.63	1.82
d	N^{obs}	CPU (ms)	**IP** (ms)	N^{obs}	CPU (ms)	**IP** (ms)
5.0	16	3.02	0.261	16	3.29	0.262
	32	5.23	0.267	32	5.75	0.287
	64	9.65	0.262	64	10.68	0.318
	128	18.49	0.285	128	20.59	0.358
20.0	16	10.12	0.264	16	11.22	0.311
	32	18.97	0.265	32	21.18	0.336
	64	36.91	0.274	64	41.09	0.402
	128	72.86	0.275	128	80.78	0.474

part onto FPGA would lead to 7.02/19.47x longer execution time than P3Net-FPGA, which highlights the advantage of our design choices.

3) Speedup of Inference and Collision Checking: Table I (top) lists the inference time of {E, P}Net and {E, P}NetLite, measured on the ZCU104 with and without the P3NetCore. The data is the average of 50 runs. As mentioned in Section V-A, ENetLite basically involves N times of forward passes to compute pointwise features, which increases the inference time by 3.78/1.57x compared to ENet2D/3D ($N = 1400, 2000$). P3NetCore accelerates the inference by 48.15/49.22x and as a result achieves 12.72/31.35x faster feature extraction than ENet. P3NetCore consistently attains more than 45x speedup, owing to the combination of inter-layer pipelining (Fig. 7 (top)) and parallelization within each layer. The inference time increases proportionally to N, which corresponds to the $\mathcal{O}(N)$ complexity of ENetLite, and P3NetCore yields a better performance gain with a larger N (48.15/50.15x for $N = 1400, 4096$, 2D). PNetLite has a 26.81/2.32x shorter inference time than PNet2D/3D, mainly due to the 32.58/2.35x parameter reduction (Section IV-B2). It does not grow linearly with the batch size B, since the computation is always parallelized along the batch dimension. The batch planning strategy in P3Net thus effectively improves success rates without incurring a significant additional overhead. With P3NetCore, PNetLite is sped up by 11.16–12.52/25.56–31.12x, resulting in an overall speedup of 322.16–335.79/59.32–71.80x over PNet.

Table I (bottom) presents the computation time of collision checking on the ARM Cortex CPU and P3NetCore. We conducted experiments with four random workspaces of size 40 containing different numbers of obstacles N^{obs}, and 50 random start-goal pairs with a fixed distance of $d = 5.0, 20.0$ for each workspace. P3NetCore gives a larger speedup with the increased problem complexity (larger d or N^{obs}). It runs

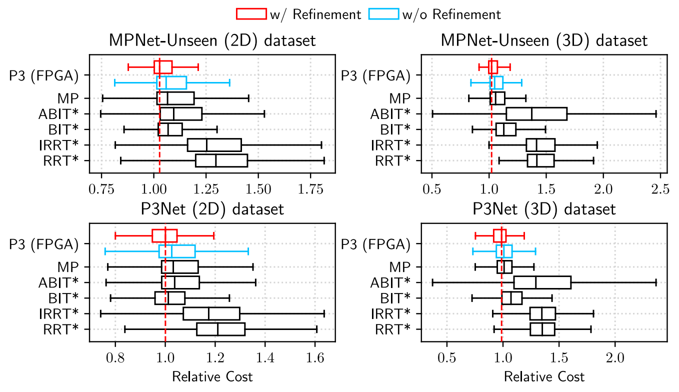

Fig. 14. Distribution of the relative path cost.

2D/3D collision checking 11.57–64.88/12.56–57.51x faster for $d = 5.0$, and 38.33–264.95/36.08–170.42x for $d = 20.0$. The parallelization with an array of submodules (Section V-B2) contributes to the two orders of magnitude faster execution. The latency does not increase linearly with N^{obs}, as P3NetCore terminates the checks as soon as any of midpoints between start-goal points is found to be in collision, and such early-exit is more likely to occur in a workspace with more obstacles.

E. Path Cost

This subsection evaluates the quality of solutions returned from P3Net in comparison with the other planners. The relative path cost is used as a quality measure; it is computed by dividing a length of the output path by that of the ground-truth available in the dataset[19]. Fig. 14 shows the results on MPNet/P3Net datasets ($I_{\text{Refine}} = 5$).

On P3Net2D dataset, P3Net with/without refinement achieves the median cost of 1.001/1.026, which is comparable to the sampling-based methods (1.038, 1.012, 1.174, and 1.210 in ABIT*, BIT*, IRRT*, and RRT*). The median is further improved by increasing the number of refine steps (1.001, 0.996, 0.989 for $I_{\text{Refine}} = 5, 10, 20$). In the replanning phase (Alg. 3, lines 23–32), the current path τ is replaced by a new one τ_{new} when it has a lower cost, meaning that the cost monotonically decreases and the path at least converges to a sub-optimal solution as $I_{\text{Refine}} \to \infty$. Similar to [16], the uniform distribution may also be used in conjunction with PNet in the sampling process (Alg. 3, line 8) to avoid sub-optimal solutions. The result confirms that the median cost is close to one without such a heuristic, and thus the learned distribution of PNetLite allows to sample a waypoint from a promising region and build a close-to-optimal path. In the supplementary material, the evolution of the path cost is presented in comparison of IRRT*. Figs. 1–2 show examples of the output paths obtained from MPNet and P3Net-FPGA on P3Net dataset.

[19]Since ground-truth paths were obtained by running sampling-based planners with a large number of iterations, the relative cost may become less than one when a planner finds a better path with a smaller cost than the ground-truth.

TABLE II
FPGA Resource Consumption (%) of P3NetCore on ZCU104

	B	BRAM	URAM	DSP	FF	LUT
2D	1	75.48	–	34.26	13.64	50.51
	2	97.92	–	49.31	14.20	46.85
	4	97.92	–	49.77	13.71	44.61
		(95.51)	–	(46.64)	–	–
	8	100.00	–	50.75	13.56	45.49
3D	1	60.10	89.58	42.36	15.04	46.02
	2	51.12	89.58	33.91	15.37	48.98
	4	69.71	89.58	46.88	15.87	52.93
		(62.18)	(83.33)	(42.13)	–	–
	8	99.20	89.58	65.45	16.29	48.91

F. FPGA Resource Consumption

Table II summarizes the FPGA resource consumption of P3NetCore for various batch sizes. The design parameters are obtained by the design space exploration as described in Section V-D. The BRAM usage reaches nearly 100%, which is mainly due to the on-chip buffers for model parameters and layer outputs. The exploration allows to automatically find a design point that fully utilize the FPGA resource and maximize the performance, without repeating the time-consuming synthesis and place-and-route.

In the 2D case, {E, P}NetLite fits within the BRAM thanks to the parameter reduction and memory-efficient sequential feature extraction (Section V-A). Plus, P3NetCore consumes less than half of the onboard DSP, FF, and LUT resources, while it consists of a collision checker and two DNNs. The 48.4% of BRAM is consumed for parameters, and 42.3% for layer outputs. The other data (point clouds, obstacles, and path information) are stored on DRAM and transferred on-chip as necessary; it consumes only 2.7% of BRAM, which is independent of N or N^{obs}. Obstacles are simply represented as bounding boxes, and point cloud is a memory-efficient format compared to dense voxels, as it does not contain the information about obstacle-free regions and is free from the curse of dimensionality. In the 3D case, the 56.4% of URAM is used to store PNetLite parameters (except the second FC layer). The estimates are presented in Table II (with parentheses) for $B = 4$; our resource model estimates the BRAM and DSP usages within around 7% error.

G. Power Efficiency

Finally, we compare the power consumption of P3Net-FPGA with that of P3Net and ABIT*. The details are provided in the supplementary material. Table III presents the results. In the 2D (3D) case, P3Net-FPGA consumes 124.51x, 147.80x, and 448.47x (39.64x, 45.60x, and 145.48x) less power than ABIT*, P3Net-CPU, and P3Net-GPU on the workstation. It achieves 4.40–5.38/4.58–6.10x (1.36–1.77/1.55–2.07x) power savings over ABIT*/P3Net on Nvidia Jetson. While P3Net-FPGA consumes slightly more power than P3Net in the 3D case, this indicates that the power consumption of the P3NetCore itself is at most 0.318W $(0.809 - 0.491)$. Combined with the results from Fig. 12 (bottom), P3Net-FPGA offers

TABLE III
COMPARISON OF THE POWER CONSUMPTION (W)

Machine	2D				3D			
	ABIT*	P3Net			ABIT*	P3Net		
	CPU	CPU	+GPU	**+IP**	CPU	CPU	+GPU	**+IP**
ZCU104	0.480	0.461	–	0.255	0.480	0.491	–	0.809
Nano	1.373	–	1.556	–	1.434	–	1.678	–
Xavier	1.123	–	1.168	–	1.097	–	1.250	–
WS	31.75	37.69	114.36*	–	32.07	36.89	117.69*	–

* 114.36W: 29.72 + 84.64 (CPU/GPU); 117.69W: 32.51 + 85.17 (CPU/GPU).

79.40–208.43/35.18–81.34x (7.19–18.49/13.27–31.88x) power efficiency than ABIT*/P3Net on ARM Cortex CPU and Nvidia Jetson in the 2D (3D) case. The power efficiency reaches 514.23x, 434.54x, and 1278.14x (48.76x, 146.83x, and 455.34x) when compared with ABIT*, P3Net-CPU, and P3Net-GPU on the workstation.

VII. CONCLUSION

In this paper, we have presented a new learning-based path planning method, P3Net. P3Net aims to address the limitations of the recently-proposed MPNet by introducing two algorithmic improvements: it (1) plans multiple paths in parallel for computational efficiency and higher success rate, and (2) iteratively refines the solution. In addition, P3Net (3) employs hardware-amenable lightweight DNNs with 32.32/5.43x less parameters to extract robust features and sample waypoints from a promising region. We designed P3NetCore, a custom IP core incorporating neural path planning and collision checking, to realize a planner on a resource-limited edge device that finds a path in $\sim$0.1s while consuming $\sim$1W. P3NetCore was implemented on the Xilinx ZCU104 board and integrated into the P3Net path planner.

Evaluation results successfully demonstrated that P3Net achieves a significantly better tradeoff between computational cost and success rate than MPNet and the state-of-the-art sampling-based methods. In the 2D (3D) case, P3Net-FPGA obtained 30.52–186.36x and 7.68–143.62x (15.69–93.26x and 5.30–45.27x) average speedups over ARM Cortex CPU and Nvidia Jetson, and its performance was even comparable to the workstation. P3Net-FPGA was up to 514.23x and 1278.14x (48.76x and 455.34x) more power efficient than P3Net and ABIT* on the Nvidia Jetson and workstation, showcasing that FPGA SoC is a promising solution for efficient path planning. P3Net converged fast to close-to-optimal solutions in most cases.

P3Net is currently evaluated on the simulated static 2D/3D environments, and the robot is modeled as a point-mass. As a future work, we plan to extend P3Net to more complex settings, e.g., multi-robot problems, dynamic environments, and higher state dimensions. P3NetCore employs a standard fixed-point format for DNN inference; while it already provides satisfactory performance improvements, low-precision formats and model compression techniques (e.g., pruning, low-rank factorization) could be used to further improve the resource efficiency and speed.

REFERENCES

[1] J. Pak, J. Kim, Y. Park, and H. I. Son, "Field evaluation of path-planning algorithms for autonomous mobile robot in smart farms," *IEEE Access*, vol. 10, no. 1, pp. 60253–60266, Jun. 2022.

[2] H. Lee, H. Kim, and H. J. Kim, "Planning and control for collision-free cooperative aerial transportation," *IEEE Trans. Autom. Sci. Eng.*, vol. 15, no. 1, pp. 189–201, Jan. 2018.

[3] T. Dang, F. Mascarich, S. Khattak, C. Papachristos, and K. Alexis, "Graph-based path planning for autonomous robotic exploration in subterranean environments," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Nov. 2019, pp. 3105–3112.

[4] R. Liu, J. Yang, Y. Chen, and W. Zhao, "eSLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform," in *Proc. Annu. Des. Automat. Conf.* (DAC), Jun. 2019, pp. 1–6.

[5] P. Gu, Z. Meng, and P. Zhou, "Real-time visual inertial odometry with a resource-efficient Harris Corner detection accelerator on FPGA platform," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2022, pp. 10542–10548.

[6] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Iowa State Univ., Ames, IA, USA, Tech. Rep. 98–11, Oct. 1998.

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

[8] 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 *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2014, pp. 2997–3004.

[9] 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 *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2015, pp. 3067–3074.

[10] A. H. Qureshi, Y. Miao, A. Simeonov, and M. C. Yip, "Motion planning networks: Bridging the gap between learning-based and classical motion planners," *IEEE Trans. Robot.*, vol. 37, no. 1, pp. 48–66, Feb. 2021.

[11] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 652–660.

[12] M. P. Strub and J. D. Gammell, "Advanced BIT* (ABIT*): Sampling-based planning with advanced graph-search techniques," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2020, pp. 130–136.

[13] M. P. Strub and J. D. Gammell, "Adaptively informed trees (AIT*): Fast asymptotically optimal path planning through adaptive heuristics," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2020, pp. 3191–3198.

[14] B. Ichter, J. Harrison, and M. Pavone, "Learning sampling distributions for robot motion planning," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2018, pp. 7087–7094.

[15] B. Ichter and M. Pavone, "Robot motion planning in learned latent spaces," *IEEE Robot. Autom. Lett.*, vol. 4, no. 3, pp. 2407–2414, Jul. 2019.

[16] J. Wang, W. Chi, C. Li, C. Wang, and M. Q.-H. Meng, "Neural RRT*: Learning-based optimal path planning," *IEEE Trans. Autom. Sci. Eng.*, vol. 17, no. 4, pp. 1748–1758, Oct. 2020.

[17] R. Yonetani, T. Taniai, M. Barekatain, M. Nishimura, and A. Kanezaki, "Path planning using neural A* search," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 12029–12039.

[18] A.-I. Toma et al., "Waypoint planning networks," in *Proc. IEEE Conf. Robot. Vis. (CRV)*, May 2021, pp. 87–94.

[19] M. Inoue, T. Yamashita, and T. Nishida, "Robot path planning by LSTM network under changing environment," in *Proc. Adv. Comput. Commun. Comput. Sci.*, Aug. 2018, pp. 317–329.

[20] M. J. Bency, A. H. Qureshi, and M. C. Yip, "Neural path planning: Fixed time, Near-optimal path generation via oracle imitation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Nov. 2019, pp. 3965–3972.

[21] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, Jun. 2017, pp. 1527–1533.

[22] D. S. Chaplot, D. Pathak, and J. Malik, "Differentiable spatial planning using transformers," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 1484–1495.

[23] Y. Kim, D. Shin, J. Lee, Y. Lee, and H.-J. Yoo, "A 0.55V 1.1mW artificial-intelligence processor with PVT compensation for micro

robots," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Jan. 2016, pp. 258–259.

[24] A. Kosuge and T. Oshima, "A 1200Ã—1200 8-edges/vertex FPGA-based motion-planning accelerator for dual-arm-robot manipulation systems," in *Proc. IEEE Symp. VLSI* Circuits, Jun. 2020, pp. 1–2.

[25] L. R. Everson, S. S. Sapatnekar, and C. H. Kim, "A time-based intra-memory computing graph processor featuring A* wavefront expansion and 2-D gradient control," *IEEE J. Solid-State Circuits*, vol. 56, no. 7, pp. 2281–2290, Jul. 2021.

[26] C. Yu, Y. Su, J. Lee, K. Chai, and B. Kim, "A 32x32 time-domain wavefront computing accelerator for path planning and scientific simulations," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2021, pp. 1–2.

[27] G. S. Malik, K. Gupta, K. M. Krishna, and S. R. Chowdhury, "FPGA based combinatorial architecture for parallelizing RRT," in *Proc. Eur. Conf. Mobile Robots (ECMR)*, Sep. 2015, pp. 1–6.

[28] S. Xiao, A. Postula, and N. Bergmann, "Optimal random sampling based path planning on FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Log. Appl. (FPL)*, Sep. 2016, pp. 1–2.

[29] S. Xiao, N. Bergmann, and A. Postula, "Parallel RRT* architecture design for motion planning," in *Proc. IEEE Int. Conf. Field Programmable Log. Appl. (FPL)*, Sep. 2017, pp. 1–4.

[30] C. Chung and C.-H. Yang, "A 1.5-μJ/task path-planning processor for 2-D/3-D autonomous navigation of microrobots," *IEEE J. Solid-State Circuits*, vol. 56, no. 1, pp. 112–122, Jan. 2021.

[31] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. IEEE/RSJ Conf. Intell. Robots Syst. (IROS)*, Sep. 2011, pp. 3513–3518.

[32] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing RRT on large-scale distributed-memory architectures," *IEEE Trans. Robot.*, vol. 29, no. 2, pp. 571–579, Apr. 2013.

[33] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed RRT for motion planning," in *Proc. IEEE Int. Conf. Robot. Automat. (ICRA)*, May 2013, pp. 5088–5095.

[34] K. Sugiura and H. Matsutani, "P3Net: PointNet-based path planning on FPGA," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2022, pp. 1–9.

[35] L. Huang, X. Zang, Y. Gong, C. Deng, J. Yi, and B. Yuan, "IMG-SMP: Algorithm and hardware co-design for real-time energy-efficient neural motion planning," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, Jun. 2022, pp. 373–377.

[36] L. Huang, X. Zang, Y. Gong, and B. Yuan, "Hardware architecture of graph neural network-enabled motion planner (Invited Paper)," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, Oct. 2022, pp. 1–7.

[37] S. Ribes, P. Trancoso, I. Sourdis, and C.-S. Bouganis, "Mapping multiple LSTM models on FPGAs," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2020, pp. 1–9.

[38] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for HLS," in *Proc. Annu. Des. Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[39] H. Zhou. "PathPlanning (GitHub repository)." GitHub. Accessed: Apr. 22, 2023. [Online]. Available: https://github.com/zhm-real/PathPlanning

[40] J. L. Blanco and P. K. Rai. "Nanoflann: A C++ header-only fork of FLANN, a library for nearest neighbor (NN) with KD-trees." GitHub. Accessed: Jun. 19, 2022. [Online]. Available: https://github.com/jlblancoc/nanoflann

Keisuke Sugiura received the B.E. and M.E. degrees from Keio University, Yokohama, Japan, in 2020 and 2022, respectively. He is currently working toward the Ph.D. degree with the Graduate School of Science and Technology, Keio University. His research interest includes the computer architecture and robotics.

Hiroki Matsutani (Member, IEEE) received the B.A., M.E., and Ph.D. degrees from Keio University, Yokohama, Japan, in 2004, 2006, and 2008, respectively. He is currently a Professor with the Department of Information and Computer Science, Keio University. His research interest includes the areas of computer architecture.