P3Net: PointNet-based Path Planning on FPGA

Keisuke Sugiura and Hiroki Matsutani
Dept. of ICS, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
EMail: {sugiura,matutani}@arc.ics.keio.ac.jp

***Abstract*—Path planning is of crucial importance for autonomous mobile robots, and comes with a wide range of real-world applications including transportation, surveillance, and rescue. Currently, its high computational complexity is a major bottleneck for the application on such resource-limited robots. As a promising and effective solution to tackle this issue, in this paper, we propose a novel learning-based method for 2D/3D path planning, P3Net (PointNet-based Path Planning Network), along with its resource-efficient implementation targeting Xilinx ZCU104 boards. Our proposal is built upon two improvements to the recently proposed MPNet: we use a parameter-efficient PointNet-based encoder network to extract high-fidelity obstacle features from a point cloud, in conjunction with a lightweight planning network to iteratively plan a path. Experimental results using 2D/3D datasets demonstrate that our FPGA-based P3Net performs significantly better than MPNet and even comparable to the state-of-the-art sampling-based methods such as BIT*. P3Net is able to plan near-optimal paths 6.24x-9.34x faster than MPNet, and eventually improves the success rate by up to 24.45%, while reducing the parameter size by 5.43x–32.32x. This enables the subsecond real-time performance in many cases and opens up a new research direction for the edge-based efficient path planning.**

***Index Terms*—Path Planning, PointNet, Point Cloud, FPGA**

I. Introduction

Path planning is a crucial component for realizing autonomous robots, with a plethora of methods proposed in the literature. Its objective is to plan a collision-free and shortest possible path connecting a given start and goal position (see Figs. 1-2). The low-cost implementation on resource-limited edge devices would allow small mobile robots (e.g., drones) to perform path planning on its own, greatly expanding the application of such robots in fields like transportation [1], rescue [2], and surveillance [3]. Sampling-based methods such as PRM [4] and RRT [5] are widely used in practice. RRT explores the environment by placing randomly sampled nodes in an obstacle-free space and incrementally expanding a tree; a number of RRT variants [6]–[10] have been proposed to improve search efficiency and convergence speed. Despite steady improvements, they still rely heavily on heuristics, which may be invalid if the environment condition (e.g., obstacle configuration) is distinct from those expected by the methods, or require careful parameter tuning. These methods inevitably face a tradeoff between the computational effort and the quality of generated paths [11]. More fundamentally, due to the limited computing capabilities of mobile robots, it usually takes tens of seconds to obtain solutions in complex scenarios (Fig. 1 (bottom)). Considering that robots run other necessary tasks (e.g., mapping and localization) simultaneously, they should ideally plan feasible paths in subseconds to prevent delays. To circumvent these issues, the recent trend is to apply deep learning techniques and develop an efficient method [12]–[14], that ideally work in a variety of environments without relying on any heuristics.

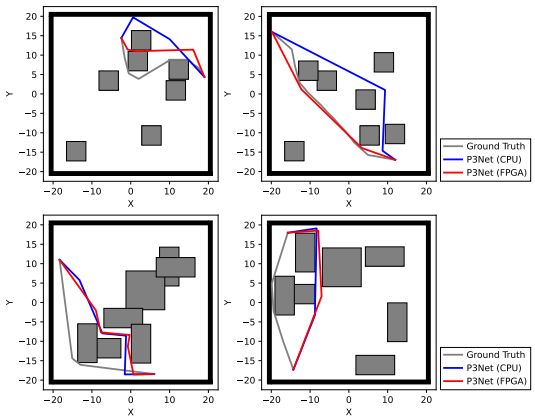

Fig. 1. Results on the 2D datasets (top: Simple2D, bottom: Complex2D).

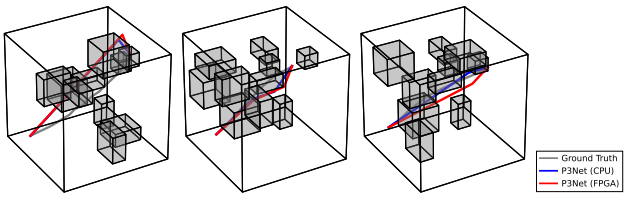

Fig. 2. Results on the Complex3D dataset.

MPNet [11] is such a learning-based method designed with generality in mind. It solves planning problems with various state-space dimensions and constraints (point-mass, rigid-body, arms, etc.) as shown in Figs. 1-2, while requiring less computational effort than conventional methods. MPNet offers a high degree of parallelism, making it more amenable to hardware acceleration. Considering these features, MPNet is suitable for the low-cost implementation targeting edge devices. MPNet uses two DNNs to (1) encode obstacle shapes from point clouds and (2) iteratively plan a feasible path. In spite of the simple structure, they have the following shortcomings. The encoder does not consider the unordered and unstructured nature of point clouds, which degrades the quality of extracted features, and the planning network has low parameter efficiency. As a result, MPNet lacks stability and struggles to obtain a solution in difficult problem settings as shown in Fig. 1 (bottom).

In this paper, we make two improvements to MPNet in order to address the above issues, and propose a novel learning-based method for 2D and 3D path planning, **P3Net** (PointNet-based Path Planning Network). We use a PointNet [15]-based encoder to obtain high-fidelity point cloud features, in conjunction with a lightweight planning network. PointNet is a simple but powerful architecture consisting of fully-connected

layers, and is the current mainstream for point cloud-based tasks, e.g., classification [16], segmentation [17], and registration [18]. We then present a resource-efficient P3Net implementation based on FPGA SoCs, consisting of a custom FPGA-based accelerator integrating these DNNs. According to the computational flow and characteristics of DNNs, we conduct a set of design optimizations for the accelerator. Experimental results demonstrate that P3Net achieves better success rates with fewer parameters and computational cost compared to MPNet. We also confirm that P3Net is faster than the state-of-the-art sampling-based planners and still produces the comparable results.

The remainder of this paper is organized as follows: Section I-A briefly reviews related works, and Section II presents the preliminaries. Our proposal is described in Section III and its implementation details are given in Section IV. Experimental results are shown in Section V. Section VI concludes the paper.

A. Related Works

1) Learning-based path planning: Learning-based methods apply deep learning techniques, e.g., value iteration [19], LSTM [20], [21], and Transformer [22], to plan paths in a discrete grid space. They take grid map images to encode the given planning tasks; point clouds are more flexible and space-efficient, since they can represent obstacles of arbitrary shape, and do not contain information about obstacle-free regions. Also, we opt to use simpler and lightweight DNNs, as we put more focus on the resource-efficient implementation for low-cost edge devices. Strudel *et al.* [12] combine the reinforcement learning (RL) and PointNet encoding. The performance of RL is severely affected by random seeds and reward design, which easily incur training instabilities.

Aside from these end-to-end approaches, there also exist a line of work for the hybrid approach. WPN [23] incorporates LSTM-based waypoint generation into A*, whereas Yonetani *et al.* [14] introduce a differentiable A*. Several studies aim to improve the exploration efficiency of RRT by generating informed samples in a learned latent space [24]–[26]. P3Net is an extension of MPNet and hence inherits its advantages; P3Net can be jointly used with any RRT-based method.

2) Acceleration of path planning: A number of studies have been reported on the hardware acceleration of PRM-based [27]–[31] and RRT-based [32]–[35] methods. Some works consider the GPU acceleration or distributed implementation of RRT [36]–[38]. The sampling-based methods require intricate strategy for parallelization due to their inherently sequential nature. Compared to that, P3Net is comprised of easily parallelizable DNN computations, and does not involve nearest neighbor search or complex data structures (e.g., K-d trees), substantially facilitating the hardware implementation. Owing to the efficient informed sampling, P3Net requires less collision checks, which constitute a bottleneck in conventional methods. To our knowledge, this paper is the first to consider the FPGA-based implementation of path planning, harnessing the advantages of learning-based approach.

II. Preliminaries

This section presents a brief description of MPNet, which is summarized in Alg. 1 and Figs. 3-4 (refer to [11] for more details). Here we assume a point-mass robot moving in a 2D or 3D environment, with its position denoted as $\mathbf{c} \in \mathbb{R}^D$ ($D = 2, 3$). MPNet aims to find a feasible path τ connecting a start and a goal position $\mathbf{c}_{\text{start}}, \mathbf{c}_{\text{goal}}$. As shown in Fig. 3 (right), MPNet takes as input $\mathbf{c}_{\text{start}}$, $\mathbf{c}_{\text{goal}}$, and a point cloud $\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$ representing obstacles in the environment. It utilizes two DNNs, **ENet** and **PNet**, for obstacle encoding and planning (Fig. 3 (left)). First, using ENet, the planner obtains a MD feature embedding $\phi(\mathcal{P}) \in \mathbb{R}^M$ of the point cloud (line 1). Then, it performs an iterative and bidirectional planning using PNet as follows (line 2, lines 12-24).

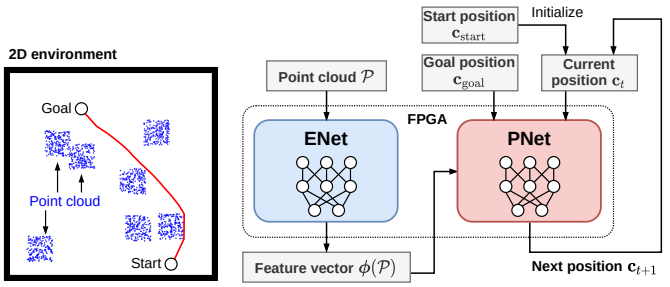

Fig. 3. Overview of the MPNet algorithm. As shown in left, MPNet takes as input a start position $\mathbf{c}_{\text{start}}$, a goal position $\mathbf{c}_{\text{goal}}$, and a point cloud $\mathcal{P}$ representing obstacles in the environment (blue points). MPNet uses two DNNs, **ENet** and **PNet**, for feature extraction and planning. First, the planner computes a feature embedding $\phi(\mathcal{P})$ using ENet. The planner then computes a path by repetitively calling PNet. Given $\mathbf{c}_{\text{goal}}$, $\phi(\mathcal{P})$, and a current position $\mathbf{c}_t$, PNet computes the next position $\mathbf{c}_{t+1}$ which is one step closer to the goal.

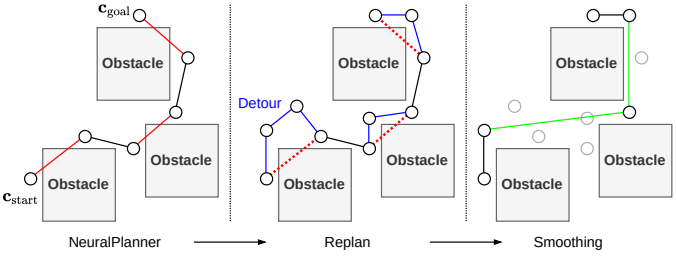

Fig. 4. MPNet-based path planning. MPNet mainly consists of three steps. NeuralPlanner (left) computes a near-optimal path connecting a start and a goal position using ENet and PNet. Replan (center) creates detours (blue lines) as necessary to avoid the obstacles and obtain a feasible, collision-free path. Smoothing (right) removes redundant waypoints (semi-transparent points) to smoothen a path and reduce a cost (green lines).

Given a start position $\mathbf{c}_{\text{start}}$, the planner incrementally expands a path $\tau^{\text{a}} = \{\mathbf{c}_{\text{start}}, \ldots, \mathbf{c}^{\text{a}}_{\text{end}}\}$ towards the goal using PNet (Fig. 4 (left), lines 16-17). PNet takes as input $\phi(\mathcal{P})$, $\mathbf{c}_{\text{goal}}$, and a current position $\mathbf{c}^{\text{a}}_{\text{end}}$ to compute a next position $\mathbf{c}^{\text{a}}_{\text{new}}$ which is one step closer to the goal (Fig. 3 (right), line 16). At the same time, the planner expands a secondary path $\tau^{\text{b}} = \{\mathbf{c}_{\text{goal}}, \ldots, \mathbf{c}^{\text{b}}_{\text{end}}\}$ from goal to start (lines 19-20). In this case, PNet computes a next position $\mathbf{c}^{\text{b}}_{\text{new}}$ from an input tuple $[\phi(\mathcal{P}), \mathbf{c}^{\text{b}}_{\text{end}}, \mathbf{c}_{\text{start}}]$ to get closer to the start position (line 19). At this point, there may be obstacles on the line between adjacent waypoints $\mathbf{c}^{\text{a}}_{\text{end}}, \mathbf{c}^{\text{a}}_{\text{new}}$ ($\mathbf{c}^{\text{b}}_{\text{end}}, \mathbf{c}^{\text{b}}_{\text{new}}$), which is addressed later by creating a detour. The planner returns a path $\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}$ between $\mathbf{c}_{\text{start}} = \mathbf{c}_0$ and $\mathbf{c}_{\text{goal}} = \mathbf{c}_T$ by connecting two paths τ^{a} and τ^{b} if possible, i.e., there is no obstacle between two path endpoints $\mathbf{c}^{\text{a}}_{\text{end}}, \mathbf{c}^{\text{b}}_{\text{end}}$ (lines 22-23). Starting from $\mathbf{c}_{\text{start}}$ and $\mathbf{c}_{\text{goal}}$, the planner expands two paths $\tau^{\text{a}}, \tau^{\text{b}}$ alternatively (lines 15, 18), and attempts to connect them; this process is repeated I_{NP}

times at maximum (line 14). If two paths cannot be connected after I_{NP} iterations, path planning between $\mathbf{c}_{\mathrm{start}}$ and $\mathbf{c}_{\mathrm{goal}}$ is considered unsuccessful (line 24). This process is denoted as $\tau \leftarrow \mathrm{NeuralPlanner}(\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}}, \boldsymbol{\phi}(\mathcal{P}), I_{\mathrm{NP}})$ (line 2).

If the above path τ is collision-free, the planner returns τ as a final solution after performing a **smoothing** process, which removes redundant waypoints in τ to shorten and smoothen the path length (lines 3-5). Given three waypoints $\mathbf{c}_i, \mathbf{c}_j, \mathbf{c}_k \in \tau$ $(i < j < k)$, the intermediate one $\mathbf{c}_j$ is pruned if $\mathbf{c}_i$ and $\mathbf{c}_k$ can be directly connected by a straight line (see Fig. 4 (right)). Otherwise, the planner moves on to the **replan** process (line 7, lines 25-37). For all pairs of adjacent waypoints $\mathbf{c}_i, \mathbf{c}_{i+1} \in \tau$ which are not connectable, it attempts to create a new path (detour) $\tau_{i,i+1} = \{\mathbf{c}_i, \mathbf{c}_i^{(1)}, \mathbf{c}_i^{(2)}, \ldots, \mathbf{c}_{i+1}\}$ between $\mathbf{c}_i, \mathbf{c}_{i+1}$ to circumvent the obstacle by calling $\mathrm{NeuralPlanner}(\mathbf{c}_i, \mathbf{c}_{i+1}, \boldsymbol{\phi}(\mathcal{P}), I_{\mathrm{RNP}})$ (Fig. 4 (center), line 32). The maximum number of iterations is set to I_{RNP} instead of I_{NP} in this case. Since the planner is now creating a shorter path, I_{RNP} is usually set smaller than I_{NP}. The new path τ' is obtained by inserting new waypoints $\mathbf{c}_i^{(1)}, \mathbf{c}_i^{(2)}, \ldots$ into τ (line 34). We refer to this process as $\tau' \leftarrow \mathrm{Replan}(\tau, \boldsymbol{\phi}(\mathcal{P}))$ (line 7). Note that PNet has a stochastic behavior due to Dropout layers, i.e., PNet returns different results on multiple runs for the same input, so do NeuralPlanner and Replan. PNet is interpreted as an efficient informed sampling process; it generates next positions $\mathbf{c}_{\mathrm{new}}$ from a promising region in the environment which may contain an optimal path.

If τ' is collision-free, the planner returns τ' as a final solution (lines 8-10); otherwise, the planner calls $\mathrm{Replan}(\tau, \boldsymbol{\phi}(\mathcal{P}))$ again. If a collision-free path τ' is still not obtained after I_{RE} times execution of Replan, then the path planning is considered unsuccessful (line 11). Owing to the stochasticity of Replan, different candidate paths are generated by multiple Replan attempts until a feasible one is found; this trial-and-error approach leads to the increased chance of avoiding obstacles and hence the better success rate. Note that the maximum iterations $I_{\mathrm{NP}}, I_{\mathrm{RNP}}, I_{\mathrm{RE}}$ are hyperparameters. The network structure of ENet and PNet, and our proposal are presented in the next section.

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: Feasible path $\tau' = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}$ $(\mathbf{c}_0 = \mathbf{c}_{\mathrm{start}}, \mathbf{c}_T = \mathbf{c}_{\mathrm{goal}})$
1: Compute point cloud feature: $\boldsymbol{\phi}(\mathcal{P}) \leftarrow \mathrm{ENet}(\mathcal{P})$
2: $\tau \leftarrow \mathrm{NeuralPlanner}(\mathbf{c}_{\mathrm{start}}, \mathbf{c}_{\mathrm{goal}}, \boldsymbol{\phi}(\mathcal{P}), I_{\mathrm{NP}})$
3: $\tau' \leftarrow \mathrm{Smoothing}(\tau)$
4: **if** $\tau' \neq \varnothing$ and τ' is collision-free **then**
5: **return** τ'
6: **for** $i = 0, \ldots, I_{\mathrm{RE}} - 1$ **do**
7: $\tau' \leftarrow \mathrm{Replan}(\tau, \boldsymbol{\phi}(\mathcal{P}))$
8: $\tau' \leftarrow \mathrm{Smoothing}(\tau)$
9: **if** $\tau' \neq \varnothing$ and τ' is collision-free **then**
10: **return** τ'
11: **return** $\varnothing$ ▷ Failure
12: **function** $\mathrm{NeuralPlanner}(\mathbf{c}_s, \mathbf{c}_g, \boldsymbol{\phi}(\mathcal{P}), I)$
13: $\tau^{\mathrm{a}} \leftarrow \{\mathbf{c}_s\}$, $\tau^{\mathrm{b}} \leftarrow \{\mathbf{c}_g\}$, $r = 0$
14: **for** $i = 0, \ldots, I - 1$ **do**
15: **if** $r = 0$ **then**
16: Compute next position: $\mathbf{c}_{\mathrm{new}}^{\mathrm{a}} \leftarrow \mathrm{PNet}(\boldsymbol{\phi}(\mathcal{P}), \mathbf{c}_{\mathrm{end}}^{\mathrm{a}}, \mathbf{c}_g)$
17: $\tau^{\mathrm{a}} \leftarrow \tau^{\mathrm{a}} \cup \{\mathbf{c}_{\mathrm{new}}^{\mathrm{a}}\}$, $r = 1$
18: **else if** $r = 1$ **then**
19: Compute next position: $\mathbf{c}_{\mathrm{new}}^{\mathrm{b}} \leftarrow \mathrm{PNet}(\boldsymbol{\phi}(\mathcal{P}), \mathbf{c}_{\mathrm{end}}^{\mathrm{b}}, \mathbf{c}_s)$
20: $\tau^{\mathrm{b}} \leftarrow \tau^{\mathrm{b}} \cup \{\mathbf{c}_{\mathrm{new}}^{\mathrm{b}}\}$, $r = 0$
21: **if** τ^{a} and τ^{b} are connectable **then**
22: $\tau \leftarrow \mathrm{Concatenate}(\tau^{\mathrm{a}}, \mathrm{Reverse}(\tau^{\mathrm{b}}))$
23: **return** τ
24: **return** $\varnothing$ ▷ Failure
25: **function** $\mathrm{Replan}(\tau = \{\mathbf{c}_0, \ldots, \mathbf{c}_T\}, \boldsymbol{\phi}(\mathcal{P}))$
26: $\tau_{\mathrm{new}} \leftarrow \varnothing$
27: **for** $i = 0, \ldots, T - 1$ **do**
28: **if** $\mathbf{c}_i$ and $\mathbf{c}_{i+1}$ are connectable **then**
29: $\tau_{\mathrm{new}} \leftarrow \tau_{\mathrm{new}} \cup \{\mathbf{c}_i, \mathbf{c}_{i+1}\}$
30: **else**
31: $\tau_{i,i+1} \leftarrow \mathrm{NeuralPlanner}(\mathbf{c}_i, \mathbf{c}_{i+1}, \boldsymbol{\phi}(\mathcal{P}), I_{\mathrm{RNP}})$
32: ▷ Compute a detour $\tau_{i,i+1} = \{\mathbf{c}_i, \mathbf{c}_i^{(1)}, \mathbf{c}_i^{(2)}, \ldots, \mathbf{c}_{i+1}\}$
33: **if** $\tau_{i,i+1} \neq \varnothing$ **then**
34: $\tau_{\mathrm{new}} \leftarrow \tau_{\mathrm{new}} \cup \tau_{i,i+1}$
35: **else**
36: **return** $\varnothing$ ▷ Failure
37: **return** τ_{new}

III. Method

In this section, we propose (1) an improved version of MPNet models, **P3Net**, and (2) its FPGA-based implementation.

A. P3Net: improved MPNet models

MPNet employs two DNNs, **ENet** and **PNet**, for feature extraction and planning (Fig. 3 (right)). We suffix the names with **2D/3D** to indicate DNNs for 2D/3D path planning when necessary. For the sake of brevity, a fully-connected (FC) layer of m inputs and n outputs is denoted as $\mathrm{FC}(m, n)$, a batch normalization for n-channel inputs/outputs as $\mathrm{BatchNorm}(n)$, a max-pooling layer with window size n as $\mathrm{MaxPool}(n)$, and a dropout with rate $p \in [0, 1)$ as $\mathrm{Dropout}(p)$, respectively. We use a PointNet [15]-based ENet and a more compact version of PNet, together called **P3Net**, as described below.

1) PointENet (PointNet-based ENet): As illustrated in Fig. 5 (left), the original ENet is a stack of FC layers, each followed by a ReLU activation. ENet2D takes as input a 2D point cloud $\mathcal{P} \in \mathbb{R}^{1400\times 2}$ containing 1400 points, which is flattened into a 2800D vector before fed to the layers. The output is a 28D feature vector, denoted as $\boldsymbol{\phi}(\mathcal{P}) \in \mathbb{R}^{28}$. Similarly, ENet3D contains a series of FC layers and computes a 60D feature from a 3D point cloud of size 2000, i.e., $\mathrm{FC}(6000, 784) \rightarrow \mathrm{ReLU} \rightarrow \mathrm{FC}(784, 512) \rightarrow \mathrm{ReLU} \rightarrow \mathrm{FC}(512, 256) \rightarrow \mathrm{ReLU} \rightarrow \mathrm{FC}(256, 60)$. Such encoder design suffers from the following two limitations. It lacks the flexibility as the number of points is fixed and cannot be adjusted according to the complexity of the planning tasks. More importantly, the output depends on the order of input points, which is undesirable for point cloud encoders. The output should remain the same even when input points are randomly swapped, since the input still represents a point cloud of the exact same shape.

To address these limitations, we adopt PointNet [15] as a backbone for ENets: we refer to PointNet-based ENet as **PointENet2D/3D**. PointENet extracts a feature $\boldsymbol{\psi}(\mathbf{p}_i)$ for each point $\mathbf{p}_i \in \mathcal{P}$ and then aggregates these pointwise features to obtain a global feature $\boldsymbol{\phi}(\mathcal{P})$ by max-pooling. As shown in Fig. 5 (center), PointENet2D extracts 252D feature vectors and is represented as $\mathrm{BE}(2, 64) \rightarrow \mathrm{BE}(64, 64) \rightarrow \mathrm{BE}(64, 64) \rightarrow \mathrm{BE}(64, 128) \rightarrow \mathrm{BE}(128, 252) \rightarrow \mathrm{MaxPool}(N)$. N is the number of points and $\mathrm{BE}(m, n)$ is a basic building block consisting of three layers, i.e., $\mathrm{FC}(m, n) \rightarrow \mathrm{BatchNorm}(n) \rightarrow \mathrm{ReLU}$. Unlike the original ENets, the number of parameters

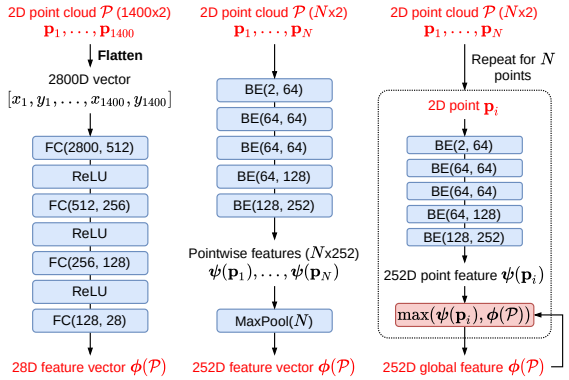

Fig. 5. ENet for 2D path planning (left: original **ENet2D**, center: **PointENet2D**, right: computation flow of the FPGA-based **PointENet2D**)

is independent from the number of input points, leading to the improved space-efficiency. PointENet3D has the same network structure as PointENet2D, except the input and output layers are replaced with $\mathrm{FC}(3, 64)$ and $\mathrm{FC}(128, 250)$ in order to extract 250D global features from 3D point clouds.

By its design, PointENet is able to process point clouds of any size, and is also invariant to the permutation of points, as it uses the symmetric max-pooling function. Another benefit is that it obtains more detailed feature representation with substantially fewer parameters than ENets. In 2D and 3D cases, the number of parameters is reduced by 31.73x (1.60M to 0.05M) and 104.47x (5.25M to 0.05M), while at the same time increasing the output feature dimensions by 9x (28 to 252) and 4.17x (60 to 250), respectively.

2) SPNet (shrinked PNet): The original PNets are constructed from a set of building blocks, denoted as $\mathrm{BP}(m, n) = \mathrm{FC}(m, n) \rightarrow \mathrm{ReLU} \rightarrow \mathrm{Dropout}(0.5)$ (Fig. 6 (left)). PNets take as input a concatenated input $[\phi(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\mathrm{goal}}]$, and compute a next position $\mathbf{c}_{t+1}$. PNet2D/3D share the same hidden layers; the only difference is in the first and last FC layers: PNet2D uses $\mathrm{FC}(32, 1280)$ and $\mathrm{FC}(32, 2)$ to take $28+2+2 = 32$D inputs and compute 2D positions, whereas PNet3D uses $\mathrm{FC}(66, 1280)$ and $\mathrm{FC}(32, 3)$ to take $60+3+3 = 66$D inputs and compute 3D positions.

As discussed above, PointENet provides obstacle features which are useful for path planning, as they are robust to the input permutations and better capture the geometric structure of the environment. It leads to an expectation that shallower PNet is sufficient for path planning. In addition, it is not reasonable to use the same set of hidden layers for both 2D/3D problems. Especially, PNet2D has low parameter efficiency, i.e., PNet2D is unnecessarily large for the complexity of the problem. We hence employ PNet models with fewer building blocks, referred to as **SPNet2D/3D**. As shown in Fig. 6 (center), SPNet2D is a stack of six building blocks; it takes a $252 + 2 + 2 = 256$D input and generates a 2D position. SPNet3D is constructed by removing some superfluous layers from PNet3D and its structure is shown in Fig. 6 (right). SPNet2D/3D have 32.58x (3.76M to 0.12M) and 2.35x (3.80M to 1.62M) fewer parameters than PNet2D/3D; by combining the results from PointENet, we finally achieve 32.32x and 5.43x parameter reduction in 2D/3D cases, respectively.

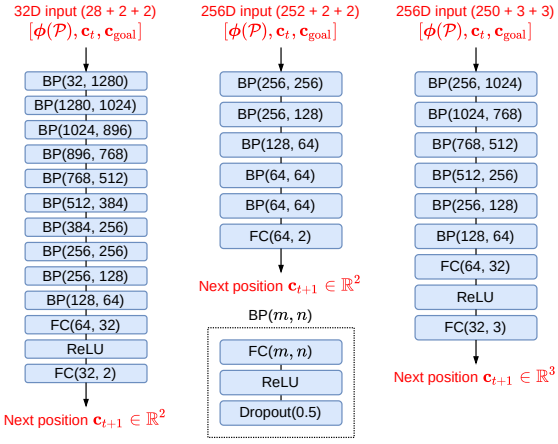

Fig. 6. PNet for 2D/3D path planning (left: original **PNet2D**, center: **SPNet2D**, right: **SPNet3D**)

B. FPGA acceleration of PointENet model

The feature extraction using PointENet requires a longer time than ENet (see Table III), since it extracts an individual feature for every point; we aim to address this by the FPGA-based acceleration. The typical software implementation of PointENet would compute individual features for all points at once, producing a matrix $\mathbf{\Psi} = \{\psi(\mathbf{p}_1), \ldots, \psi(\mathbf{p}_N)\}$ of size $N \times 252$ (or $N \times 250$ in the 3D case), and then apply max-pooling to obtain a 252D (250D) global feature $\phi(\mathcal{P}) = \mathrm{MaxPool}(\mathbf{\Psi})$. In this case, each building block involves a series of matrix operations, i.e., $\mathrm{BE}(m, n) = \mathrm{ReLU}(\mathrm{BatchNorm}(\mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^\top))$, where $\mathbf{W} \in \mathbb{R}^{n \times m}$ and $\mathbf{b} \in \mathbb{R}^n$ are weight and bias parameters of FC layer, $\mathbf{1} \in \mathbb{R}^N$ is a vector of all ones, and $\mathbf{X} \in \mathbb{R}^{N \times m}$ is an input matrix stacking mD features for all N points. While matrix operations are highly efficient and hardware-amenable, this approach is not suitable for FPGA implementation, as it requires large buffers for storing intermediate results and features for all N points, resulting in higher memory consumption. Also, the maximum number of input points is constrained by the total amount of memory resources, limiting the scalability. To circumvent this issue, we take another approach for extracting point cloud features as follows.

As depicted in Fig. 5 (right), our PointENet implementation computes an individual feature $\psi(\mathbf{p}_i)$ every time a new point $\mathbf{p}_i$ arrives, and then computes the element-wise maximum between $\phi(\mathcal{P})$ and $\psi(\mathbf{p}_i)$ to update the output feature $\phi(\mathcal{P})$. This process is repeated for all N points. The max-pooling operation is replaced by N times execution of element-wise maximum $\forall j \ \phi(\mathcal{P})_j \leftarrow \max(\phi(\mathcal{P})_j, \psi(\mathbf{p}_i)_j)$. In this way, the memory consumption is minimized, as we need to store intermediate results for only a single point $\mathbf{p}_i$. PointENet design is optimized by exploiting both data-level and task-level parallelism. In FC layers, matrix and vector operations are parallelized by partially unrolling the loop over output dimension and partitioning the buffers. The batch normalization and ReLU activation are also optimized by calculating multiple output elements at once. In addition, we employ a dataflow optimization to pipeline the entire design (see Fig. 7); each layer is viewed as one stage of the pipeline, meaning that feature extraction for multiple different points is performed

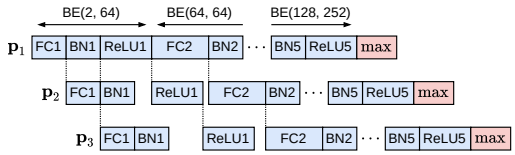

Fig. 7. Dataflow optimization of PointENet2D.

in parallel, which effectively reduces the overall latency with only a few additional resources.

C. FPGA acceleration of SPNet model

Since SPNet is used in every iteration in the NeuralPlanner algorithm (Section II), we also implement SPNet on the FPGA to improve the entire performance. For FC and ReLU activation layers, we leverage the data-level parallelism to optimize the matrix and vector operations, as in the previous section. As described in Section II, a notable characteristic of PNet/SPNet is its stochastic behavior, which originates from the dropout layers. For a nD input $\mathbf{x} = [x_1, \ldots, x_n]$, dropout of probability p is usually implemented by generating i.i.d. random samples $r_1, \ldots, r_n$ from a uniform distribution over $[0, 1]$ and then setting x_i to zero if $r_i < p$. Our design of $\mathrm{Dropout}(0.5)$ follows the same approach; we generate a 32-bit pseudorandom unsigned integer r_i for each input element x_i and replace x_i with zero if r_i is less than 2^{16}, which is performed in a pipelined manner. A 32-bit linear feedback shift register (LFSR) with feedback polynomial $r^{32}+r^{22}+r^2+r+1$ is used as a pseudorandom generator. All dropout layers share the same LFSR, and before starting the SPNet computation, its internal state is initialized with a 32-bit seed generated on the CPU side, in order to avoid dropping elements in the same positions. We discuss the implementation details of our FPGA-based P3Net in the next section.

IV. Implementation

A. Board-level implementation of MPNet

As shown in Fig. 8, our board-level implementation of P3Net targets Xilinx Zynq UltraScale+ MPSoC devices, which is divided into two parts: Processing System (PS) and Programmable Logic (PL). Our proposed DNNs are packaged in a single AXI-compatible custom IP core, referred to as **P3NetCore**. We developed two versions of P3NetCore for 2D/3D path planning, referred to as P3NetCore2D/3D, which include 2D/3D versions of the proposed models, respectively.

P3NetCore along with one AXI DMA controller (DMAC) and two AXI interconnects is implemented on the PL side. DMAC is connected to the High-Performance (HP0) port via interconnect to enable high-speed data transfer between PS and PL. When trigerred from the PS side, DMAC reads data from the DDR memory and transfers them to the slave port of P3NetCore using AXI4-Stream protocol (red arrows in Fig. 8). DMAC also retrieves the outputs from P3NetCore and writes them back to the DDR. P3NetCore and DMAC both have an AXI4-Lite slave port, which is connected to the High-Performance Master (HPM0) port, to allow access to their control registers from PS (blue arrows in Fig. 8). P3NetCore provides a set of control and status bits (e.g., start and ready). DMAC also has registers to specify the starting physical address and length of the DDR buffer to be transferred. PS runs the main part of MPNet algorithm and calls P3NetCore by configuring those registers via memory-mapped I/O.

B. Implementation of P3NetCore

Fig. 9 presents an overview of the data and control flows among modules inside P3NetCore2D. P3NetCore consists of a main module and two inference modules (white rounded squares in Fig. 9). It also provides four operation modes in total, i.e., **InitENet**, **InitPNet**, **RunENet**, and **RunPNet**. Main module orchestrates the overall process; when a start bit is asserted by PS, it receives an operation mode from DMAC. **InitENet/InitPNet** are for initializing PointENet/SPNet parameters, respectively: main module receives model parameters from DMAC and stores them to the dedicated on-chip buffer (Mode 1 and 2, Fig. 9). In **InitPNet** mode, a 32-bit seed is also transferred to initialize the LFSR pseudorandom generator for dropout layers (see Section III-C).

When **RunENet** or **RunPNet** mode is selected, main module delegates PointENet and SPNet inference tasks to the two dedicated modules (Mode 3 and 4, Fig. 9). Each inference module consists of a chain of submodules, which correspond to the structure of DNNs, and on-chip buffers for storing network parameters and layer input/outputs. In **RunENet** mode, PointENet module (upper part of Fig. 9) receives a 2D/3D point one-by-one from DMAC and processes it in a pipelined manner to compute a global feature $\boldsymbol{\phi}(\mathcal{P})$, as described in Section III-B. PointENet module contains three types of submodules for FC layers, batch normalization coupled with ReLU activation (BN-ReLU), and max-pooling (Max), respectively. Similarly, in **RunPNet** mode, SPNet module (lower part of Fig. 9) receives a concatenated input $[\boldsymbol{\phi}(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\mathrm{goal}}]$, and computes a next position $\mathbf{c}_{t+1}$ using a set of submodules. The output from both inference modules is transferred back to the PS via AXI4-Stream master interface.

Thanks to the compact PointENet/SPNet2D models, P3NetCore2D stores all model parameters on the on-chip BRAM buffers, leading to the minimized memory access latency. P3NetCore3D stores most parameters on BRAM buffers, except ones for the first three FC layers in SPNet3D, which are stored on the DDR due to the limited memory resources. As a result, P3NetCore3D has three additional AXI master ports connected to the HP0 port (see Fig. 8); FC modules for these three layers only have small BRAM buffers for storing a subset of the weight and bias parameters, which are read from the DDR using a burst transfer as needed. Considering that SPNet performs a regression task, i.e., it should compute the precise position $\mathbf{c}_{t+1}$, we use a 32-bit fixed-point format comprised of 16-bit integer part and 16-bit fraction part, for model parameters and layer input/outputs.

We developed P3NetCore2D/3D (Fig. 8) using Vitis HLS 2020.2, and run synthesis and place-and-route of the block design (Fig. 9) using Vivado 2020.2. We used Xilinx ZCU104 Evaluation Kit as a target SoC, which integrates an FPGA fabric (XCZU7EV-2FFVC1156), a quad-core ARM Cortex-A53 CPU running at 1.2GHz, and 2GB of DDR3 DRAM. For running the software implementation, we used Pynq Linux v2.7 based on Ubuntu 20.04. The operation frequency of P3NetCore is set to 100MHz.

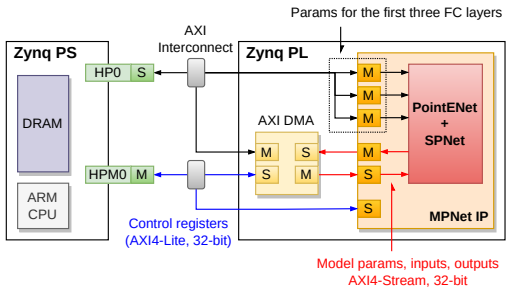

Fig. 8. Board-level implementation of P3Net3D.

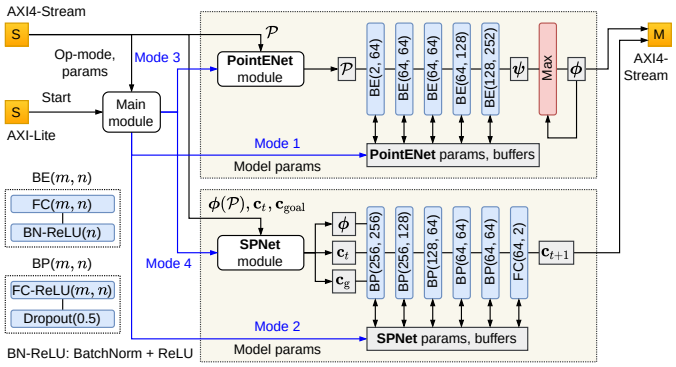

Fig. 9. Block diagram of P3NetCore2D.

C. Details for the software implementation and training

For the definition of original MPNet models, we used the source code from the paper authors [11] as a reference; we implemented all the other necessary parts such as training, testing, and MPNet-based path planner from scratch using Python 3.8.2 and PyTorch 1.10.2. We then added a new P3Net-based path planner using the CPU- and FPGA-based implementation of the proposed DNNs. Note that we compiled and built PyTorch from source, enabling ARM Neon intrinsics to fully take advantage of the multicore processor. We took the implementation of the famous sampling-based methods, i.e., RRT* [7], Informed-RRT* [8], and BIT* [9] from the GitHub repository [39], and adapted it into our codebase. For a fair performance comparison, we modified it to use FLANN library [40] for faster nearest-neighbor search.

We followed the offline batch training strategy as presented in [11], and jointly trained two models in an end-to-end fashion as follows. We first computed an embedding of the point cloud $\phi(\mathcal{P})$ using ENet (PointENet), and then estimated the next position $\widehat{\mathbf{c}}_{t+1}$ from a concatenated input $[\phi(\mathcal{P}), \mathbf{c}_t, \mathbf{c}_{\text{goal}}]$ using PNet (SPNet). Then, we computed a L2 loss $\|\mathbf{c}_{t+1} - \widehat{\mathbf{c}}_{t+1}\|^2$ using a ground-truth next position $\mathbf{c}_{t+1}$, and its gradient was back-propagated through two models. We used Adam optimizer with default parameters in PyTorch. The batch size is set to 1024, 8192, and 128 when training MPNet3D, MPNet2D, and P3Net, respectively. We set the number of epochs to 200 and 50 when training MPNet and P3Net.

V. Evaluation

We evaluate P3Net in terms of execution time, success rate, path cost, and resource utilization, and compare the results against MPNet and the three famous sampling-based methods, i.e., RRT*, Informed-RRT* (IRRT*), and BIT*.

A. Path planning datasets

We used two datasets for 2D/3D path planning, **Simple2D** and **Complex3D**, with each divided into one training and two test sets (**seen/unseen**). Both are provided by the authors of MPNet [11]. Their training sets contain 100 workspaces with different obstacle configurations. Each has 4000 planning problems, consisting of randomly generated start/goal pairs and their corresponding ground-truth paths. The **seen** sets contain the same 100 workspaces as the training sets, with each having 200 problems, whereas the **unseen** sets consist of ten workspaces with each having 2000 problems, which are not observed during training. Fig. 3 (left) visualizes an example of the problem in Simple2D dataset. Additionally, we generated **Complex2D** dataset containing 100 workspaces with more obstacles than Simple2D (Fig. 1 (bottom)). Note that we only used the first 5 and 50 problems for each workspace in seen and unseen sets, respectively, and excluded trivial problems whose solutions were straight lines connecting start and goal positions. As a result, the total number of problems in Simple2D, Complex2D, and Complex3D were 229/240 (unseen/seen), 409, and 84/78 (unseen/seen), respectively.

B. Success rate

The success rates of 2D/3D path planning under various settings are listed in Tables I-II (U/S denote unseen/seen datasets). The sampling-based methods (a1–a12, c1–c12) show higher success rates with larger number of iterations I as expected due to their asymptotic optimality, i.e., they are more likely to find a solution as they place more random nodes inside a workspace and build a denser tree. BIT* achieves the best success rates with smaller I among the other sampling-based methods. As seen in a13–a14 and a16–a17, P3Net2D achieves 24.0% and 22.7% higher success rates with 32.32x fewer parameters than MPNet2D when the number of replan attempts I_{RE} is set to 10 and 50 (see Alg. 1), respectively. Notably, P3Net with $I_{\text{RE}} = 10$ (a16) already gives better result than MPNet with more replan attempts (a14). In the 3D case (c13–c14, c16–c17), P3Net3D shows slightly better results than MPNet3D with 5.43x fewer parameters. P3Net generally outperforms RRT* and Informed-RRT*, and gives results comparable to BIT* in a shorter period of time as demonstrated in Section V-C. We also confirm that the FPGA-based P3Net (a19–a21, c19–c21) maintains high success rates as that of the CPU counterpart (a16–a18, c16–c18), despite the use of a simpler pseudorandom generator (Section III-C) and arithmetic errors introduced by the fixed-point format. The results obtained from unseen/seen datasets (e.g., a17–a18) demonstrate that the proposed models generalize well to a variety of environments that are not observed during training.

We also consider the combination of PointENet and the original PNet (P3Net-a, P3Net-c) to validate the effectiveness of the proposed models. As hypothesized in Section III-A, the PointNet-based feature extraction substantially improves the success rate (a13–a14 and a22–a23) and allows to use more shallower PNet models for trajectory planning without compromising the success rate (a22–a23 and a16–a17). We obtain the similar results in the 3D case as seen in c13–c14, c22–c23, and c16–c17. P3Net was 74.08% successful

($I_{\mathrm{RE}} = 50$) in Complex2D dataset; it outperformed MPNet (38.14%) by a large margin in a more difficult problem setting.

TABLE I
SUCCESS RATES ON SIMPLE2D DATASET

#	U/S	Method	I	%	#	U/S	Method	I_{RE}	FPGA	%
a1	U	RRT*	50	3.06	a13	U	MPNet	10		59.83
a2	U	RRT*	100	15.28	a14	U	MPNet	50		69.00
a3	U	RRT*	250	66.81	a15	S	MPNet	50		66.67
a4	U	RRT*	500	91.27	a16	U	P3Net	10		83.84
a5	U	IRRT*	50	2.62	a17	U	P3Net	50		91.70
a6	U	IRRT*	100	16.16	a18	S	P3Net	50		92.50
a7	U	IRRT*	250	63.76	a19	U	P3Net	10	✓	81.86
a8	U	IRRT*	500	90.83	a20	U	P3Net	50	✓	**93.45**
a9	U	BIT*	50	74.67	a21	S	P3Net	50	✓	96.25
a10	U	BIT*	100	96.07	a22	U	P3Net-a	10		78.17
a11	U	BIT*	250	99.13	a23	U	P3Net-a	50		89.08
a12	U	BIT*	500	**99.56**	a24	S	P3Net-a	50		91.67

TABLE II
SUCCESS RATES ON COMPLEX3D DATASET

#	U/S	Method	I	%	#	U/S	Method	I_{RE}	FPGA	%
c1	U	RRT*	50	0.0	c13	U	MPNet	10		95.24
c2	U	RRT*	100	2.38	c14	U	MPNet	50		96.43
c3	U	RRT*	250	38.10	c15	S	MPNet	50		98.72
c4	U	RRT*	500	77.38	c16	U	P3Net	10		91.67
c5	U	IRRT*	50	1.19	c17	U	P3Net	50		97.62
c6	U	IRRT*	100	3.57	c18	S	P3Net	50		100.00
c7	U	IRRT*	250	32.14	c19	U	P3Net	10	✓	86.90
c8	U	IRRT*	500	85.71	c20	U	P3Net	50	✓	**98.81**
c9	U	BIT*	50	**100.00**	c21	S	P3Net	50	✓	98.72
c10	U	BIT*	100	100.00	c22	U	P3Net-c	10		96.43
c11	U	BIT*	250	100.00	c23	U	P3Net-c	50		97.62
c12	U	BIT*	500	100.00	c24	S	P3Net-c	50		98.72

C. Execution time

Table III presents the inference times of MPNet and P3Net models (averaged over 10 runs). Since PointENet2D/3D involve N times forward propagation of FC layers to compute individual features for all N points (see Section III-B), they have 3.70x/1.53x longer inference times than ENet2D/3D, respectively. The FPGA-based implementation reduces the inference time by 25.53x/25.74x, leading to 6.91x/16.78x faster feature extraction than ENet2D/3D. SPNet2D/3D require 30.23x/2.33x less inference times than PNet2D/3D, thanks to the shallower network structure. P3NetCore further accelerates the computation by 5.52x/3.59x, which contributes to the overall speedup of 167.0x/8.36x.

TABLE III
INFERENCE TIMES OF MPNET AND P3NET MODELS

	ENet	PNet	CPU		FPGA	
			PointENet	SPNet	PointENet	SPNet
2D	43.58ms	101.87ms	161.09ms	3.37ms	6.31ms	0.61ms
3D	147.13ms	101.95ms	225.77ms	43.81ms	8.77ms	12.20ms

Table IV summarizes the distributions of execution times on Simple2D and Complex3D datasets. Note that we chose path planners from Tables I-II that produced the similar success rates, and took only the successful planning problems into consideration. For a fair performance comparison, we included the data transfer overhead between PS and PL. As apparent, the FPGA-based P3Net (a20, c20) is the fastest in terms of mean and median values, making P3Net competitive to the state-of-the-art sampling-based methods, while it requires longer execution times in some challenging cases. Notably, P3Net outperforms MPNet (a14, c14) on all metrics, indicating that P3Net greatly improves the stability of the algorithm by extracting more informative features and efficiently estimating paths. In Complex2D dataset, the execution time of MPNet and FPGA-based P3Net were $31.309\mathrm{s} \pm 49.994\mathrm{s}$ and $2.336\mathrm{s} \pm 3.242\mathrm{s}$ on average, respectively; the performance gain increased in more difficult problems.

TABLE IV
EXECUTION TIMES ON 2D AND 3D DATASETS (IN SECONDS)

#	Mean & Std	Med	Max	#	Mean & Std	Med	Max
a4	6.003 ± 1.027	5.847	11.102	c4	3.343 ± 0.334	3.292	4.424
a8	8.201 ± 3.433	7.194	34.210	c8	3.966 ± 0.636	3.916	6.738
a10	2.289 ± 0.586	2.343	**5.088**	c9	0.351 ± 0.240	0.319	**2.327**
a14	19.843 ± 37.901	1.946	217.244	c14	1.253 ± 2.424	0.459	18.824
a17	1.841 ± 2.848	0.718	20.530	c17	0.999 ± 1.611	0.407	9.994
a20	$\mathbf{1.128 \pm 1.995}$	**0.414**	16.255	c20	$\mathbf{0.212 \pm 0.418}$	**0.064**	3.024

Table V shows the speedup factors obtained using P3Net on Simple2D and Complex3D datasets. We only considered the successful cases as above. P3Net gives the median speedup of 3.04x/1.20x and the average speedup of 22.22x/1.92x over MPNet in 2D/3D cases (a14/a17, c14/c17), respectively. Our FPGA-based implementation further improves the median performance by 1.97x/7.72x and the average by 3.38x/12.37x (a17/a20, c17/c20). Combining these results yields the median speedup of 6.24x/9.34x and average speedup of 49.46x/14.09x. Note that the larger difference between the mean and median speedup factors implies that P3Net plans paths significantly faster on some challenging problems, which also supports the advantage of the proposed models. The FPGA-based P3Net (a20, c20) is faster than all the sampling-based methods, highlighting the efficiency of learning-based approach.

TABLE V
SPEEDUP FACTORS ON 2D AND 3D DATASETS

	Mean & Std	Med		Mean & Std	Med
a4/a20	21.242 ± 19.819	14.151	c4/c20	69.018 ± 54.418	56.008
a8/a20	33.590 ± 48.183	17.109	c8/c20	81.495 ± 66.791	68.871
a10/a20	8.189 ± 7.992	5.361	c9/c20	6.533 ± 6.054	4.053
a14/a17	22.220 ± 50.949	3.042	c14/c17	1.918 ± 2.510	1.197
a17/a20	3.384 ± 4.934	1.970	c17/c20	12.368 ± 19.717	7.723
a14/a20	49.462 ± 133.183	6.241	c14/c20	14.090 ± 18.743	9.340

The distributions of execution times are visualized in Fig. 10. Though the histogram of the CPU-based P3Net (blue) is more spread out than BIT* (cyan), it has more weight on the far left side, clearly indicating the performance advantage of P3Net. Comparing CPU- and FPGA-based P3Net (blue and red), we observe that the distribution has shifted to the left (noticeable in the 3D case). Fig. 11 shows typical cases of the execution time breakdowns. In MPNet2D (a14), the inference of ENet and PNet (red + green) accounts 80.74% of the execution time, which is reduced to only 3.93% in the FPGA-based P3Net2D (a20). In MPNet3D (c14) and FPGA-based P3Net3D (c20), 98.79% and 95.15% of the execution time is spent on the inference respectively; though the inference still occupies a large portion of the entire workload, P3Net effectively improves the performance as seen in Fig. 11 (right).

D. Path cost

Fig. 12 plots relative costs of the output paths. We computed a relative cost as a length of the estimated path relative to one of the near-optimal path available in the dataset. The paths produced by sampling-based methods tend to be jagged and

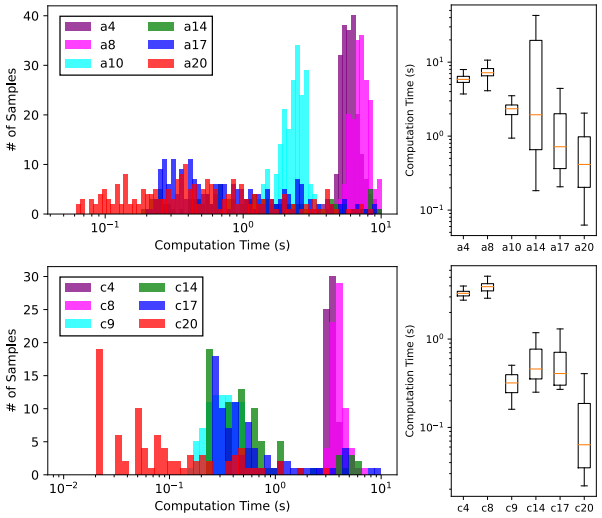

Fig. 10. Execution times (top: Simple2D, bottom: Complex3D). a4/c4: RRT*, a8/c8: Informed-RRT*, a10/c10: BIT*, a14/c14: MPNet, a17/c17: P3Net (CPU), a20/c20: P3Net (FPGA).

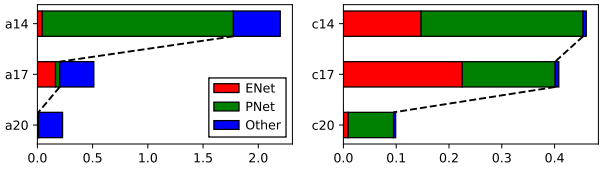

Fig. 11. Execution time breakdown (left: Simple2D, right: Complex3D). a14/c14: MPNet, a17/c17: P3Net (CPU), a20/c20: P3Net (FPGA).

not optimal, since they connect nodes in a tree which are randomly placed during the exploration process. For a fair comparison, we also perform the smoothing (see Section II) in sampling-based methods. As seen in Fig. 12, P3Net (e.g., a17) is able to produce near-optimal paths in both 2D and 3D cases. We also confirm that the FPGA implementation maintains the quality of solutions. In Fig. 12, MPNet2D (a14) seems to give a smaller cost than P3Net2D (a17, a20), since MPNet2D only solves problems for which optimal solutions are easily obtained, and we only consider those successful cases to compute cost. As described in Section V-B, P3Net2D (a17) outperforms MPNet2D by a large margin in terms of the 22.7% higher success rate.

In Figs. 1-2, we show examples of paths obtained by CPU- and FPGA-based P3Net planners (blue and red lines) along with the ground-truth ones (gray lines). MPNet failed to plan paths in the problems shown in Fig. 1 (bottom), while P3Net found near-optimal paths that pass through narrow passages.

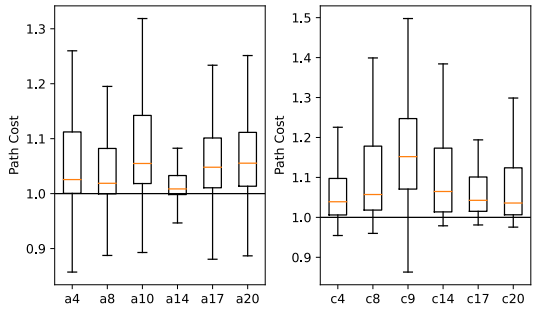

Fig. 12. Distribution of path cost (left: Simple2D, right: Complex3D).

E. FPGA resource utilization

Table VI summarizes the FPGA resource utilization of P3NetCore. The on-chip buffers for model parameters and layer input/outputs (Fig. 9) occupy a large portion of on-chip BRAM and URAM resources, in exchange for the improved access latency. In addition, P3NetCore consumes more than half of the onboard DSP blocks to parallelize the matrix-vector operations in FC layers. Since the utilization of FF and LUT is low, we could also implement other parts of the algorithm, e.g., collision checking, on the FPGA fabric to further improve the performance. Note that our compact network design allows to store a large part of parameters on the on-chip buffers and benefit from the low access latency; the implementation of original MPNet would require more DRAM buffers and thus suffer from the data transfer overhead.

TABLE VI
FPGA RESOURCE UTILIZATION OF P3NETCORE

		BRAM	URAM	DSP	FF	LUT
	Available	312	96	1728	460800	230400
2D	Used	198.5	80	888	33875	69333
	Utilization %	63.62	83.33	51.39	7.35	30.09
3D	Used	298	72	920	39140	80603
	Utilization %	95.51	75.00	53.24	8.49	34.98

F. Power consumption

According to the post-place-and-route reports by Vivado 2020.2, the power consumption of CPU and our P3NetCore2D/3D are 2.212W, 1.223W, and 1.764W, respectively. While the entire power consumption increases by 55.3% and 79.7% when we employ P3NetCore2D/3D, the average speedups of 3.38x and 12.37x (Table V) lead to the 2.18x and 6.88x improvements in energy efficiency, respectively.

VI. CONCLUSION

In this paper, we proposed P3Net, a novel learning-based 2D/3D path planning method, as an improvement to MPNet. P3Net leverages two types of DNNs: a PointNet-based point cloud encoder and a lightweight planning network by informed sampling, to better capture the obstacle information and quickly find a collision-free path, while at the same time substantially reducing the parameter sizes and computational costs. We then presented an FPGA-based custom accelerator, P3NetCore, and implemented it on the Xilinx ZCU104 board, in order to fully exploit the parallelism in DNN computations and realize an efficient learning-based path planning on resource-limited devices. Experimental results demonstrate that P3Net outperforms MPNet in terms of success rate, execution time, and algorithm stability. In the 2D/3D cases, the FPGA-based P3Net achieves 24.45%/2.38% higher success rates than the original MPNet with 32.32x/5.43x fewer parameters and 6.24x/9.34x faster computation time, leading to the 2.18x/6.88x better energy efficiency. We also confirm that P3Net generalizes well to the unseen environments and is able to produce near-optimal paths faster than the state-of-the-art sampling-based methods, highlighting the advantages of learning-based approaches.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number JP22J21699.

REFERENCES

[1] H. Lee, H. Kim, and H. J. Kim, "Planning and Control for Collision-Free Cooperative Aerial Transportation," *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 1, pp. 189–201, Jan. 2018.

[2] M. Brunner, B. Brüggemann, and D. Schulz, "Motion Planning for Actively Reconfigurable Mobile Robots in Search and Rescue Scenarios," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Nov. 2012, pp. 1–6.

[3] A. Wallar, E. Plaku, and D. A. Sofge, "Reactive Motion Planning for Unmanned Aerial Surveillance of Risk-Sensitive Areas," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 3, pp. 969–980, Jul. 2015.

[4] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[5] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Iowa State University, Technical Report No. 98–11, Tech. Rep., Oct. 1998.

[6] J. James J. Kuffner and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Apr. 2000, pp. 995–1001.

[7] S. Karaman and E. Frazzoli, "Sampling-Based Algorithms for Optimal Motion Planning," *International Journal of Robotics Research*, 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 *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2014, pp. 2997–3004.

[9] ——, "Batch Informed Trees (BIT*): Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 3067–3074.

[10] M. P. Strub and J. D. Gammell, "Adaptively Informed Trees (AIT*): Fast Asymptotically Optimal Path Planning through Adaptive Heuristics," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 3191–3198.

[11] 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 Transactions on Robotics*, vol. 37, no. 1, pp. 48–66, Feb. 2021.

[12] R. Strudel, R. G. Pinel, J. Carpentier, J.-P. Laumond, I. Laptev, and C. Schmid, "Learning Obstacle Representations for Neural Motion Planning," in *Proceedings of the Conference on Robot Learning (CoRL)*, Nov. 2020, pp. 355–364.

[13] C. Yu and S. Gao, "Reducing Collision Checking for Sampling-Based Motion Planning Using Graph Neural Networks," in *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, Dec. 2021, pp. 4274–4289.

[14] R. Yonetani, T. Taniai, M. Barekatain, M. Nishimura, and A. Kanezaki, "Path Planning using Neural A* Search," in *Proceedings of the International Conference on Machine Learning (ICML)*, Jul. 2021, pp. 12 029–12 039.

[15] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 652–660.

[16] X. Ma, C. Qin, H. You, H. Ran, and Y. Fu, "Rethinking Network Design and Local Geometry in Point Cloud: A Simple Residual MLP Framework," arXiv preprint arXiv:2202.07123, Feb. 2022.

[17] W. Wang, R. Yu, Q. Huang, and U. Neumann, "SGPN: Similarity Group Proposal Network for 3D Point Cloud Instance Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018, pp. 2569–2678.

[18] Y. Aoki, H. Goforth, R. A. Srivatsan, and S. Lucey, "PointNetLK: Robust & Efficient Point Cloud Registration Using PointNet," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019, pp. 7163–7172.

[19] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel, "Value Iteration Networks," in *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, Dec. 2016, pp. 2154–2162.

[20] L. Lee, E. Parisotto, D. S. Chaplot, E. Xing, and R. Salakhutdinov, "Gated Path Planning Networks," in *Proceedings of the International Conference on Machine Learning (ICML)*, Jul. 2018, pp. 2947–2955.

[21] M. Inoue, T. Yamashita, and T. Nishida, "Robot Path Planning by LSTM Network Under Changing Environment," in *Advances in Computer Communication and Computational Sciences*, Aug. 2018, pp. 317–329.

[22] D. S. Chaplot, D. Pathak, and J. Malik, "Differentiable Spatial Planning using Transformers," in *Proceedings of the International Conference on Machine Learning (ICML)*, Jul. 2021, pp. 1484–1495.

[23] A.-I. Toma, H. A. Jaafar, H.-Y. Hsueh, S. James, D. Lenton, R. Clark, and S. Saeedi, "Waypoint Planning Networks," in *Proceedings of the IEEE Conference on Robotics and Vision (CRV)*, May 2021, pp. 87–94.

[24] B. Ichter, J. Harrison, and M. Pavone, "Learning Sampling Distributions for Robot Motion Planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 7087–7094.

[25] B. Ichter and M. Pavone, "Robot Motion Planning in Learned Latent Spaces," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 2407–2414, Jul. 2019.

[26] J. Wang, W. Chi, C. Li, C. Wang, and M. Q.-H. Meng, "Neural RRT*: Learning-Based Optimal Path Planning," *IEEE Transactions on Automation Sciences and Engineering*, vol. 17, no. 4, pp. 1748–1758, Oct. 2020.

[27] N. Atay and B. Bayazit, "A Motion Planning Processor on Reconfigurable Hardware," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2006, pp. 125–132.

[28] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The Microarchitecture of a Real-Time Robot Motion Planning Accelerator," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[29] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A Programmable Architecture for Robot Motion Planning Acceleration," in *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2019, pp. 185–188.

[30] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-P: A Scalable Accelerator for Robot Motion Planning in a Dynamic Environment," in *Proceedings of the ACM/ESDA/IEEE Design Automation Conference (DAC)*, Jun. 2018, pp. 1–6.

[31] Y. Yang, X. Chen, and Y. Han, "Dadu-CD: Fast and Efficient Processing-in-Memory Accelerator for Collision Detection," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, Jul. 2020, pp. 1–6.

[32] G. S. Malik, K. Gupta, K. M. Krishna, and S. R. Chowdhury, "FPGA based Combinatorial Architecture for Parallelizing RRT," in *Proceedings of the European Conference on Mobile Robots (ECMR)*, Sep. 2015, pp. 1–6.

[33] S. Xiao, A. Postula, and N. Bergmann, "Optimal Random Sampling Based Path Planning on FPGAs," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Application (FPL)*, Sep. 2016, pp. 1–2.

[34] S. Xiao, N. Bergmann, and A. Postula, "Parallel RRT* Architecture Design for Motion Planning," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[35] C. Chung and C.-H. Yang, "A 1.5-μJ/Task Path-Planning Processor for 2-D/3-D Autonomous Navigation of Microrobots," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 1, pp. 112–122, Jan. 2021.

[36] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively Parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems (IROS)*, Sep. 2011, pp. 3513–3518.

[37] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing RRT on Large-Scale Distributed-Memory Architectures," *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 571–579, Apr. 2013.

[38] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A Scalable Distributed RRT for Motion Planning," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2013, pp. 5088–5095.

[39] H. Zhou, "PathPlanning (GitHub repository)," https://github.com/zhm-real/PathPlanning, 2020.

[40] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," in *Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP)*, Feb. 2009, pp. 331–340.