Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning

Yutong Wang
University of California, Davis
United States of America
ytwwang@ucdavis.edu

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

ABSTRACT
A mixed-precision program is a floating-point program that utilizes different precisions for different operations, providing the opportunity of balancing the trade-off between accuracy and performance. Precision tuning aims to find a mixed-precision version of a program that improves its performance while maintaining a given accuracy. Unfortunately, existing precision tuning approaches are either limited to small-scale programs, or suffer from efficiency issues. In this paper, we propose FPLEARNER, a novel approach that addresses these limitations. Our insight is to leverage a Machine Learning based technique, Graph Neural Networks, to learn the representation of mixed-precision programs to predict their performance and accuracy. Such prediction models can then be used to accelerate the process of dynamic precision tuning by reducing the number of program runs. We create a dataset of mixed-precision programs from five diverse HPC applications for training our models, which achieve 96.34% F1 score in performance prediction and 97.03% F1 score in accuracy prediction. FPLEARNER improves the time efficiency of two dynamic precision tuners, PRECIMONIOUS and HhFPTUNER, by an average of 25.54% and up to 61.07% while achieving precision tuning results of comparable or better quality.

1 INTRODUCTION
With the advancement of artificial intelligence techniques and supercomputer performance, numerical software with extensive use of floating-point (FP) arithmetic has become increasingly prevalent, accompanied by a rapid escalation in power consumption. Unfortunately, designing compute-intensive applications that are both reliable and energy-efficient remains a significant challenge in recent years [5]. The reason is that when working with FP arithmetic, determining the appropriate FP precision is crucial. Although high precision guarantees program accuracy and reliability, it may also compromise efficiency and result in unnecessary energy consumption. For example, on most modern processors, utilizing single precision formats can be at least twice as fast as the performance of double precision formats [3]. A trade-off between accuracy and performance is often achieved by mixed precision, i.e., performing different operations in different precisions.

Automated precision tuning is regarded as a promising direction for finding mixed-precision programs that achieve the best trade-off between performance and accuracy [13]. Precision tuning entails replacing the original precision assigned to FP variables in numerical programs with lower precision in a manner that ensures accuracy standards are maintained. However, it is non-trivial to reason about mixed precision due to the higher potential for numerical errors arising from minor changes in the precision of FP variables. This characteristic presents difficulties in various domains such as Deep Neural Networks acceleration [10], compiler optimizations in FP programs [24, 46], and CUDA programs acceleration [30].

Existing automated precision tuners mainly use either static analysis or dynamic search-based approaches. Although static approaches [14, 16, 58] are generally sound and do not require executing programs with input data, they are restricted to FP expressions or small programs and unable to tune large codes with conditionals and loops, thus have not been utilized for High Performance Computing (HPC) workloads [45]. On the other hand, dynamic search-based approaches [25, 37, 53, 54] have been applied to larger-scale numerical programs but require running numerous mixed-precision program versions to determine the effect of mixed precision in program performance and accuracy. Thus, dynamic approaches are time-intensive and face the challenge of an exponential search space of mixed-precision programs. Furthermore, the overall time required by dynamic approaches is based on the program’s execution time. As a result, performing dynamic analysis on larger HPC programs becomes progressively more challenging and time-consuming, as they necessitate longer execution times. As far as we are aware, all search-based precision tuners suffer from scalability issues when applied to large HPC programs.

CCS CONCEPTS
• Computing methodologies → Neural networks; • Software and its engineering → Software performance; Software reliability.

KEYWORDS
program representation, Graph Neural Networks, floating point, mixed precision, numerical software, program optimization, precision tuning

ACM Reference Format:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE ’24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
https://doi.org/10.1145/3597503.3623338
In this paper, we present FPLearner, a Machine Learning (ML) based approach to learn the representation of floating-point mixed-precision programs for predicting their performance and computation accuracy. FPLearner is designed to improve the efficiency of existing dynamic precision tuners, while ensuring the quality of the proposed solutions. Our insight is straightforward: reducing the number of program runs required during the search by automatically predicting “promising” mixed-precision programs, i.e., programs that are likely to exhibit performance speedup while satisfying the specified accuracy constraint.

We are inspired by recent work in vulnerability detection [9, 70], type inference [63], and bug detection [17], among others, which investigate the potential of Graph Neural Networks (GNNs) [55] in program representation. However, predicting performance and accuracy of mixed-precision programs remains challenging due to several factors. First, existing methods have not been applied to represent mixed-precision programs which contain numerous arithmetic operations. We propose a novel GNN-based approach to learn features from a customized graph representation, named Precision Interaction Graph (PIG), which is designed to represent mixed-precision programs by modeling interactions of precision among FP variables across the program. Second, mixed-precision programs involve long-range dependencies among FP variables. To overcome the challenge, we innovatively deploy a Gated Graph Neural Network (GGNN) architecture [40] to capture long dependencies among FP operations in such programs, while also effectively learning information from various relations in the graph.

Since there is no existing dataset for the purpose of making inferences on mixed-precision programs, we build a dataset with 1228 mixed-precision programs from five representative HPC applications. Each sample has a performance label and an accuracy label, indicating whether the program has speedup and is within error threshold, respectively. Our experimental evaluation shows that our models are effective at accurately predicting both execution performance (96.34% F1 score) and computation accuracy (97.03% F1 score), outperforming other baseline methods. Additionally, we integrate our models to existing precision tuners and evaluate it on four case studies. The results show that our models improve the efficiency of precision tuners by an average of 25.54% and up to four times. The precision tuners typically start by creating a search space that includes all variables and function calls requiring precision tuning. The precision tuners then proceed to conduct a search with the aim of identifying an optimal mixed-precision program. The optimal solution is defined as the mixed-precision program variant that delivers the greatest performance speedup while keeping the computation error below a predetermined threshold. In reality, finding the “best” solution is not feasible, and precision tuners settle on a local minimum.

Despite their potential benefits, dynamic precision tuners face significant scalability challenges. For instance, they must execute every candidate mixed-precision program encountered during the search at least once to determine if it is faster than the original program and meets the given error threshold. This is particularly problematic when the target program has a long runtime, as it becomes infeasible to explore a large number of mixed-precision program variants due to the considerable time cost involved.

An Example of Precision Tuning. We present a motivating example of precision tuning on LULESH version 2.0 [33], a proxy application developed at Lawrence Livermore National Laboratory. LULESH discretely approximates the hydrodynamics equations by dividing the spatial problem domain into a collection of volumetric elements defined by a mesh.

Search Space. We first define a search space which considers 365 FP variables declared in the program. The initial type of each FP variable is double. With the precision candidate set \{float, double\}, the size of the search space is \(2^{365}\). The approximate average runtime of the original LULESH program on our machine is 18 seconds. If we assume each mixed-precision program version of LULESH also takes around 18 seconds, then evaluating all possible mixed-precision programs would take \(2^{365} \times 18\) seconds, approximately equaling to \(3.76 \times 10^{157}\) hours. Even if we parallelize this task, the search space remains excessively vast, leading to significant computational resource consumption.

Search-based Precision Tuning. Since exploring the whole search space for the global optimum is overly expensive, we adopt a state-of-the-art dynamic precision tuner [54], leading us to a local minimum. The precision tuner narrows down the scope to 2564 mixed-precision programs by applying a heuristic search. Each of mixed-precision programs requires to be run at least once to observe its runtime and computation accuracy. If we assume there is no overhead other than running the programs, and each mixed-precision program takes an average 18-second runtime, then the tuning process would take 2564 \times 18\) seconds, which equals to 13 hours. This is a large amount of time compared to the running time of the
target program, and limits the scalability of existing search-based precision tuners.

**Demand for Performance and Accuracy Prediction.** To optimize the search process, our insight is to make predictions about the performance and accuracy of mixed-precision programs to reduce the total number of program runs required by precision tuning. Specifically, if we can accurately predict the two key factors: (i) whether a mixed-precision program is faster than the original program, and (ii) whether its computation error falls within a given error threshold, then precision tuners can avoid program runs, resulting in significant time savings. This motivates the need for predicting the performance and accuracy of mixed-precision programs.

The following section describes how FPLearner represents mixed-precision programs and uses an ML architecture to train models that predict performance and accuracy. We also describe the integration of FPLearner models into existing precision tuners.

## 3 TECHNICAL APPROACH

Our goal is to train models that predict if a mixed-precision version of a given initial FP program (i) achieves performance speedup with respect to the initial program, and (ii) produces a result within a predefined error threshold. Figure 1 shows a high-level description of FPLearner, which includes two stages: program representation of mixed-precision programs, and graph learning using GGNNS [40]. This section discusses our approach in more detail along with a use case scenario of our models in precision tuning.

### 3.1 Program Representation

In the first stage, FPLearner analyzes a mixed-precision program and extracts the necessary information to build a graph representation, named Precision Interaction Graph (PIG). Representing programs is challenging due to the abundance of structural information contained within them, which cannot be effectively captured by conventional text-based representations. To address this, graph-based methods are employed to represent programs. However, accurately representing FP programs is challenging because of their mixed-precision nature. Numerical arithmetic operations in such programs, even with minor precision changes, e.g., converting a variable from double to float, may significantly impact performance and accuracy. Inspired by this, we prioritize the program semantics concerning FP arithmetic operations, where precision interactions among FP variables occur, during the graph construction process. We leverage the graph structure to model precision interactions in FP programs, leading to a more effective program representation for reasoning about the use of FP mixed precision.

To achieve this, FPLearner utilizes the Abstract Syntax Tree (AST) as the backbone of a PIG and extracts FP arithmetic related features of the nodes in the AST to obtain their initial representation (Section 3.1.1). Furthermore, FPLearner constructs four additional kinds of edges from the graph backbone, each of which emphasizes different aspects of the target programs (Section 3.1.2). The final PIG serves as input to the second stage of FPLearner for graph-level prediction tasks (Section 3.2).

#### 3.1.1 Graph Backbone and its Node Representation

FPLearner starts by constructing the AST of the mixed-precision program, whose nodes and edges serve as the foundation for the PIG. An AST is an ordered tree where inner nodes represent operators and leaf nodes represent operands [64]. Each statement or predicate in the program is mapped to an operator in the graph. A sample mixed-precision program is shown in Figure 2a as well as its graph representation in Figure 2b. In this program, each assignment statement (lines 2, 3, 5) is represented by an assignment operator “=” in the graph; the predicate on line 4 is represented by a comparison operator “>=”, and the function call on line 6 is represented by “CALL”. In addition to statements and predicates, an inner node in the graph can also represent an arithmetic operator such as “+” on line 5, or a function call such as the mathematical library call “sqrt” on line 2.

Leaf nodes represent identifiers and constants, which in numerical programs are often of type floating point. To differentiate mixed-precision versions of an FP program, it is useful to represent leaf nodes using their precision. For example, in the mixed-precision program from Figure 2a, every identifier and constant is in either double or single precision. To reflect this, we use “D” to represent type double and “F” to represent type float in the corresponding graph. The scope of our work centers around two precisions, double and single. However, it can be effortlessly adapted to accommodate additional precisions.

Unlike most of the existing node embedding methods for program representation, we do not directly use source code to represent nodes. Instead, we create the initial node representation using three node features that are most relevant to FP characteristics: the node’s type, its precision (if applicable), and the name of the operator (if applicable). The type refers to the kind of program construct a node represents, such as variable, constant, or control structures.
This feature provides the fundamental structural information of the program. The node’s precision applies to (i) FP identifiers, (ii) FP constants, (iii) arithmetic operators, and (iv) FP mathematical functions that have implementations in different precisions. Consider again the example from Figure 2. The precision of the FP identifier a on line 4 is double, the type of FP constants such as 1.1, 2.0 and 1.3952 is double if no suffix “f” is used. The precision of the node “a” is double—the type of the operand with the highest precision. Finally, the type of the FP mathematical function “sqrt” is double, while the type of “sqrt f” (the single-precision implementation of “sqrt”) would be float. Regarding operator names, the name attribute is extracted when a node represents (i) an assignment statement, (ii) a predicate, (iii) an arithmetic operator (e.g., “*”), or (iv) a mathematical library function call (e.g., “sqrt”). Our insight for extracting FP precision and operator names is to learn the semantics from FP arithmetics between different precisions.

After node feature extraction, we use word2vec [47] to encode each feature and concatenate the three encodings together into a fixed-length vector to initialize the node representation. Note that we do not use features such as variable names. FP programs may follow different naming conventions, and we have found that FP programs in particular lack using descriptive variable names.

3.1.2 Edge Construction. To obtain more comprehensive information beyond the capabilities of an AST, FPLEARNER creates four additional types of edges using the graph backbone established in Section 3.1.1. This expanded graph structure, as illustrated in Figure 2b, is what we refer to as the PIG: TypeCasting, AssignedFrom, Control Flow, and Program Dependence (including Data Dependence and Control Dependence) edges.

TypeCasting Edge (TC). Type casting refers to both, explicit castings included in the program, and implicit castings added by compilers. In mixed-precision programs, type casting typically involves automatic type conversion between different precisions, such as converting from double to float or vice versa. When performing FP arithmetic operations, the precision of the result is the maximum of the precisions of the operands. In Figure 2’s sample program, line 5 uses a multiplication operator where one operand variable a is in double precision while the other operand variable b is in single precision. Thus, the multiplication is performed in double precision, which requires variable b to be cast to double. This is illustrated in Figure 2b by the TypeCasting edge from the node “F” to the node “*”. Additionally, when assigning FP values, the precision of the right-hand-side expression must match the precision of the target variable on the left-hand side. Thus, on line 5, the precision of the multiplication result is double, and must be cast to float before being stored in variable c. Consequently, another casting edge exists from the node “*” to the node “=”. FPLEARNER constructs the TypeCasting edges based on our observation that excessive type castings can have adverse effects on program performance, and even potentially lead to an increase in computation errors. For instance, if the result of an arithmetic operation in double precision is assigned to a single-precision variable, the result must be rounded to fit the single-precision format, causing not only additional processing time but also a loss of precision that would introduce errors in the calculation. As our approach aims to infer program performance and computation accuracy, these TypeCasting edges provide relevant information to learn patterns in precision interactions.

AssignedFrom Edge (AF). AssignedFrom edges denote that the values of right-hand-side variables are used to compute the value of the left-hand-side variable in an arithmetic assignment. In other words, these edges capture dependencies within assignment statements. For example, the values of variable a and b on line 5 of the sample program are used to compute the value of variable c. As shown in Figure 2b, this assignment statement leads to two AssignedFrom edges within PIG. One edge originates from the node that represents variable a and links to the node that represents variable c, while the other edge connects the node that represents variable b to the node that represents variable c. The use of AssignedFrom edges is motivated by a prior study [25] that leverages variable dependence in FP arithmetic assignments to model programs. This approach is based on the assumption that highly dependent variables are more likely to be assigned the same precision. The addition of AssignedFrom edges to our PIG improves its effectiveness by highlighting the precision interactions resulting from FP arithmetic assignments within the program.

Control Flow (CF) and Program Dependence (PD) Edges. To build the PIG across a wider range of contexts, we employ two classic program analysis techniques: control flow analysis and program dependence analysis. Control flow edges, as shown in Figure 2b, capture the execution order of FP arithmetic statements and alternative paths that are determined by conditional statements, such as the if statement on line 4 of the program. Program dependence edges [19] reflect dependencies among statements and predicates. Data dependence (DD) edges are a type of program dependence edge that is created by calculating reaching definitions for each edge.
statement and predicate. This type of edge captures the influence of one FP variable on another across different locations within the program. Control dependence (CD) edges, on the other hand, correspond to the influence of predicates on the values of FP variables.

3.2 Graph Learning

To make inferences on program performance and computation accuracy, the second stage of FPLEARNER uses a GNN architecture to learn features on the input graph PIG. GNNs are deep learning (DL) based methods specialized for the graph domain. We train separate models for the performance and accuracy prediction respectively. In this section, we will first introduce the graph definition and notations that will be used (Section 3.2.1), subsequently describe the two parts of our graph learning architecture: the propagation model (Section 3.2.2) and the output model (Section 3.2.3). And next, we will discuss the novelty of our approach (Section 3.2.4) and the insights behind our design choices (Section 3.2.5).

3.2.1 Graph Definition and Notations. We formulate the PIG as a multi-relational graph, which is a type of information network defined in [56]. As shown in Figure 2b, our multi-relational PIG has five types of edges (AST, TC, AF, CF and PD), as well as one type of node, for the sake of simplicity. Nodes are distinguished based on their features. The graph is denoted as a directed graph \( G = (V,E) \), consisting of a node set \( V \) where a node is defined as \( v \in \{1,2,\ldots,|V|\} \), and an edge set \( E \) where an edge is defined as \( e = (u,v') \in V \times V \). The graph is also associated with an edge type mapping function \( \phi : E \rightarrow R \), where \( R \) denotes the set of edge types and in our case, \( |R| = 5 \). The rest of the notations that will be used in the following sections are shown and explained in Table 1.

3.2.2 Propagation Model. The propagation model, which constitutes the initial segment of the GNN architecture, is determined by the subsequent recurrence:

\[
\begin{align*}
    h_v^{(0)} &= x_v, v \in \{1,2,\ldots,|V|\} \quad (1) \\
    m_{vq}^{(l+1)} &= \sum_{v' \in N_q(v)} \Theta_q \cdot h_{v'}^{(l)} \quad (2) \\
    h_v^{(l+1)} &= \text{GRU} \left( h_v^{(l)} \sum_{q=1}^{R} m_{vq}^{(l+1)} \right) \quad (3)
\end{align*}
\]

In the first step, represented by Equation (1), for node \( v \) in the node set \( V \), the initial representation vector \( x_v \) is assigned to the first component of the node \( v \)'s hidden state, which is denoted as \( h_v^{(0)} \). As discussed in Section 3.1.1, each node’s initial embedding has a fixed length. The second step, represented by Equation (2), passes information between node \( v \) and all adjacent nodes in its neighborhood \( N_q(v) \), with learnable parameters \( \Theta_q \) that depend on the edge type and direction, and aggregates this information using a summation operator. The third step, represented by Equation (3), involves the Gated Recurrent Unit (GRU) update for \( h_v^{(l+1)} \), which incorporates the summation aggregation of each edge type’s neighborhood information and the information from the previous step \( h_v^{(l)} \). By following this recurrence, the final representation vector for each node in the last layer \( L \) is obtained and denoted as \( h_v^{(L)} \) for node \( v \in V \).

3.2.3 Output Model. Equation (4) defines the output model, which is the second component of the GNN architecture.

\[
\text{Prob} (p_i) = \text{Sigmoid} \left( \text{MLP} \left( \frac{1}{|V|} \sum_{v \in V} h_v^{(L)} \right) \right) \quad (4)
\]

Once the GNN architecture has propagated information through \( L \) layers, the representation vector \( h_v^{(L)} \) of each node \( v \in V \) is averaged globally to obtain a vector that represents the entire graph. This vector is then fed into a Multi-Layer Perceptron (MLP) that is enveloped by a Sigmoid activation layer to generate the output value \( \text{Prob}(p_i) \) of program \( p_i \). The output value \( \text{Prob}(p_i) \) will then be used to calculate the Binary Cross Entropy (BCE) loss:

\[
L = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot \log \text{Prob} (p_i) + (1 - y_i) \log (1 - \text{Prob} (p_i)) \quad (5)
\]

where \( y_i = \{0,1\} \) and \( \text{Prob}(p_i) \) represent the likelihood that program \( p_i \) belongs to the class label 1 in the binary graph classification task. The training goal is to minimize the BCE loss on all labeled programs, which is experimentally confirmed as the most simple and effective loss function in our implementation.

It has been shown that the GNN architecture can capture long-range interactions [40, 70]. This is suitable to our domain where long-range interactions between mixed-precision values are often observed in numerical programs. For instance, in the sample program depicted in Figure 2a, the variable \( a \) is used twice on lines 4 and 5, following its assignment on line 2. The return value from the function call \( \text{sqrt} \) and its precision have an impact on the subsequent usages of \( a \). However, even for such a small program, the graph nodes representing \( a \) on lines 2 and 5 are not sufficiently close to facilitate learning. Real-world mixed-precision programs are often significantly longer and more complex than the sample program. Therefore, to draw accurate inferences on the execution performance and computation accuracy of mixed-precision programs in practical applications, our architecture is a vital necessity.

3.2.4 Novelty of Our Approach. Diverse GNN architectures have shown revolutionary performances in software engineering [69]. FPLEARNER showcases the novelty of adopting a GNN architecture, GGNN, that benefits us to learn features from compute-intensive numerical programs. This is a significant departure from previous
research, as we address the unique challenges posed by mixed-precision programs. The first challenge is that FP operations usually exist throughout the entire program, and a single FP variable may be used in a far-off arithmetic operation within the program. Representing such programs requires graphs one order of magnitude larger than those reported in prior work [9, 17, 70]. Larger graphs require propagating information over longer ranges in the graph. GGNNs have the advantage of capturing long-range dependencies within the graph, which assists us in tackling this challenge effectively. The second challenge is that different types of relations describing distinct and rich contexts in mixed-precision programs should be captured. For instance, TypeCasting edges can capture the context of precision castings on FP variables, while AssignedFrom edges capture dependencies between variables in an FP assignment statement. The GGNN architecture allows to learn features from a multi-relational graph with distinct meanings of connectivity from various edge types.

3.2.5 Why Binary Classification. Our approach simplifies the task of predicting performance and accuracy by treating it as a binary classification problem. This simplification naturally fits our precision tuning use case, where the focus is on determining whether a program meets the required standards rather than precise values, especially for accuracy checking.

Alternatively, modeling the prediction as a regression problem poses challenges [65]. For example, our preliminary studies show that accuracy and performance values often have an imbalanced and skewed distribution. Rare and extreme values, including program errors that can reach infinity, are frequently encountered. Additionally, missing data in certain target regions makes generalization difficult across the entire supported range. Therefore we leverage the binary classification approach to mitigate the impact of such data distribution. However, tackling the challenges within the regression problem is a future direction that can provide more precise and comprehensive information for the tuning process.

3.3 Using the Models in Precision Tuners

We specifically focus on dynamic precision tuning [5, 23, 25, 38, 53, 54], to showcase a use case scenario of the FP Learner models. For a description of the typical workflow of dynamic precision tuners, please refer to Section 2.

The workflow for using the models in precision tuners is shown in Figure 3. Our goal is to utilize the models to aid precision tuning of any FP program, especially those not included in the initial training process. Therefore, it is necessary to fine tune the models prior to their use to learn features of mixed-precision programs from unseen applications more effectively. This motivates the need for three main steps, as described below.

**Step 1: Pre-run Stage.** This stage relates to collecting data to fine tune the models for a new target application. During this stage, we leverage the precision tuner to produce an initial set of mixed-precision programs. In other words, the precision tuner runs the search on the target application for a short amount of time to gather initial mixed-precision programs. The programs are executed to determine their performance and computation accuracy, thus obtaining the ground truth. These mixed-precision programs, along with their respective performance and accuracy labels, constitute the fine-tuning dataset for the subsequent fine-tuning stage.

**Step 2: Fine-tuning Stage.** The pre-trained performance prediction and accuracy inference networks are fine tuned on the target application’s dataset, which has limited data because of the time and cost associated with program execution for dataset construction in the pre-run stage. Note that running the precision tuner for a longer period of time would defeat the purpose of having models to predict performance and accuracy. As a result of the limited data, a challenge in this stage is that for either performance or accuracy inference task, the binary class label distribution varies on distinct target applications. In case of an imbalanced label distribution, we apply a widely accepted and straightforward technique known as random oversampling, which entails the random repetition of minority instances to balance the class distribution, and has been proven useful when working with limited data [48].

A standard fine-tuning technique [66] is adopted to copy layers from the propagation model of the pre-trained prediction networks to the target network. After that, the output model is initialized randomly and trained on the target dataset. This method of fine tuning has been shown effective in training a large target network without the risk of overfitting, particularly when the target dataset is much smaller in size than the base dataset [22, 27, 66].

**Step 3: Optimization Stage.** In the optimization stage, the precision tuner benefits from the use of two fine-tuned networks that improve the efficiency of the remaining search. This stage starts by continuing the search from the pre-run stage. Every candidate mixed-precision program in the search path is evaluated using two models: the performance prediction model to determine if it has a runtime speedup compared to the original program, and the accuracy inference model to determine if its computation results are within a given error threshold. The search process aims to identify the program with the highest speedup, and to achieve this, only programs that are classified as “promising”—with both a speedup and within the error threshold—are executed to verify the prediction, and most importantly, to obtain the actual speedup. If a program
is predicted to fail to meet the speedup or error threshold criteria, it is not executed. This allows the search process to continue without being burdened by mixed-precision programs that are less likely to meet the performance and accuracy requirements. This methodology results in a more efficient precision tuning process.

4 EVALUATION

The goal of this evaluation is to answer the following questions:

RQ 1. How effective is our approach in predicting performance and computation accuracy of mixed-precision programs?
RQ 2. How effective is each type of edge in PIG to represent mixed-precision programs?
RQ 3. How useful are our PPLERNA models when integrated into existing dynamic precision tuners?
RQ 4. How effectively can the parameters in our pre-trained models be transferred to new programs?

4.1 Datasets for Model Training

To evaluate the effectiveness of PPLERNA we must create a large dataset of mixed-precision programs for which both performance and accuracy are known. To the best of our knowledge, we are the first to create such a dataset.

We create a dataset of mixed-precision programs based on five large representative HPC applications written in C/C++: Blacksholes [6], CFD [11], Hotspot [11], HPCCG [29], and LavaMD [11]. These programs are part of HPC-MixPBenck [49], a benchmark suite for mixed-precision analysis. We excluded small kernels and applications for which we could not find mixed-precision versions that outperform the original programs. The precisions used in the mixed-precision programs are double and single precision.

The dataset must include (1) acceptable mixed-precision programs that are faster than the original program and meet the error threshold, and (2) unacceptable programs that are slower than the original program or fail to satisfy the error threshold. Finding acceptable programs is challenging as randomly assigning lower precision often leads to unacceptable programs. Instead, we leverage the precision tuner PRECIMONIOUS [54], which systematically searches for suitable precision configurations while adhering to performance and accuracy constraints. We collect all explored mixed-precision programs and label them based on speedup and error threshold compliance. This process uses representative inputs provided by the benchmarks, which achieve a 92% code coverage on average.

Table 2 presents an overview of the dataset. Our focus on real-world HPC applications with intensive FP operations results in relatively large graph sizes compared to previous work in program representation.1 The class label distribution in the set of mixed-precision programs is imbalanced. To address this, we randomly select 628 programs for a balanced dataset in performance prediction and another 600 for accuracy prediction, both including samples from all applications. As shown in the rest of this section, the prediction models trained on these datasets prove effective.

We use the code analysis platform joern [64] to extract nodes and edges from the AST and to compute the CF and PD edges of PIG. The TC edges are inferred based on FP arithmetic operators, assignments, and function call arguments, considering implicit type conversions across different precisions. Additionally, we incorporate AF edges following the approach outlined in [1]. The PIG edges are directed. While TC and AF edges are typically much less common than other types, we amplify their impact by adding corresponding inverse edges which is a common practice [2, 9].

4.2 RQ1: Model Performance

4.2.1 Baselines. To the best of our knowledge, we are the first to propose a technique to predict the performance and accuracy of mixed-precision HPC applications. We compare the performance of our GGNN approach with three DL-based baselines that we implement. The first two treat source code as natural language, while the third uses a graph representation of the program as input.

Our text-based baselines are a native LSTM [31] (the most commonly used DL technique for code analysis [57]), and a Bidirectional LSTM (BiLSTM) architecture [15] inspired by [43]. BiLSTMs, which prove superior than unidirectional RNNs [26, 31] and CNNs [21, 39] according to recent studies [42, 70], are suitable for our purpose as they consider both forward and backward directions, capturing the influence of earlier and later statements on FP variables.

The third baseline is a Relational Graph Neural Network (RGCN) architecture [56]. Recent works [9, 59, 71] have shown that RGCNs are more effective for multi-relational data compared to other GNNs like Graph Convolutional Networks (GCNs) [35] and Graph Attention Networks (GATs) [61]. RGCNs extend the commonly used GCNs and are well-suited for our use case as they can learn transformations specific to relations, adapting based on the type and direction of an edge in PIG.

4.2.2 Implementation and Training Details. We use PyTorch [50] and PyTorch Geometric [20] to implement our approach and baselines. Our models are trained on two Nvidia RTX A6000 GPUs (48GB memory per GPU) using Ubuntu 20.04 and CUDA 11.7.

The datasets of mixed-precision programs are randomly divided into three parts: 70% for training, 10% for validation, and the remaining 20% for testing. The batch size is set to 16 and we shuffle the training dataset for each epoch during training. We set the training epochs as 500, and use the early stopping manner [52] with the patience set to 30 epochs to reduce overfitting on the training dataset and improve the generalization of our neural networks. We use the Adam optimizer [34] with learning rate 0.0001, weight decay 0.001
and L2 regularization to avoid overfitting. The dimension of the vector representation of each token in our vocabulary is set to 100. In our GGNN, we set the dimension of hidden states as 100, and the number of time steps as 3.

4.2.3 Evaluation Metrics. We use four evaluation metrics to measure the effectiveness of our prediction models: accuracy (A), precision (P), recall (R) and F1 score (F1). In either the performance prediction or the accuracy inference tasks, we calculate the metrics for each label and then report their unweighted mean.

4.2.4 Experimental Results. As shown in Table 3, our approach, which utilizes GGNNs to learn the graph representation of mixed-precision programs, outperforms the other DL-based baseline methods in all four evaluation metrics. For instance, in terms of F1 score, our approach achieves 96.34% on the performance prediction task, which is a 27.89% improvement over LSTM, a 14.50% improvement over BiLSTM and a 9.72% improvement over RGCN. Additionally, our approach’s F1 score achieves as high as 97.03% on the accuracy prediction task, resulting in a 28.80%, 11.78% and 12.85% gain compared to LSTM, BiLSTM and RGCN, respectively. We hypothesize the reasons for our approach to surpass others. Firstly, we find that PIG provides a more effective program representation for mixed-precision programs by modeling inner precision interactions. Secondly, the GGNN architecture, as opposed to LSTM and BiLSTM, can learn heterogeneous relationships within a graph and benefit from a wider range of contextual information to capture program features. Finally, compared to RGCN, the GRU mechanism in GGNNs allows for deeper exploration and the capture of longer-range dependencies in the graph.

Table 3: Our Approach vs. Baselines. A: Accuracy, P: Precision, R: Recall, F1: F1 score.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Performance Prediction</th>
<th>Accuracy Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>A (%)</td>
<td>P (%)</td>
</tr>
<tr>
<td></td>
<td>70.05</td>
<td>69.28</td>
</tr>
<tr>
<td>BiLSTM</td>
<td>80.31</td>
<td>86.20</td>
</tr>
<tr>
<td>RGCN</td>
<td>85.16</td>
<td>88.01</td>
</tr>
<tr>
<td>GGNN</td>
<td>96.09</td>
<td>96.72</td>
</tr>
</tbody>
</table>

4.3 RQ2: Edge Ablation Study

To answer RQ2, we conduct an ablation study that investigates the influence of each type of edge used in PIG by selectively excluding one type at a time from the entire graph. This study allows us to isolate and observe the specific contribution of each individual edge type. The results are shown in Table 4. Compared to using all types of edges, excluding any one type of edge decreases the accuracy score by 5.46%–12.55% for performance prediction, and 4.69%–8.60% for accuracy prediction. The individual contributions of each edge type to the overall results are considered notable in comparison to earlier studies with edge analysis [1, 2, 70]. Although TypeCasting and AssignedFrom edges occur less frequently than other edge types, they still make a similar contribution to an average accuracy gain of 5.66%. Overall, this ablation study confirms that our models benefit from interactions among all edge types.


<table>
<thead>
<tr>
<th>Edges</th>
<th>Performance Prediction</th>
<th>Accuracy Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A (%)</td>
<td>P (%)</td>
</tr>
<tr>
<td>No AST</td>
<td>89.84</td>
<td>91.66</td>
</tr>
<tr>
<td>No CF</td>
<td>83.54</td>
<td>86.87</td>
</tr>
<tr>
<td>No PD</td>
<td>85.16</td>
<td>87.23</td>
</tr>
<tr>
<td>No TC</td>
<td>89.06</td>
<td>91.05</td>
</tr>
<tr>
<td>No AF</td>
<td>90.03</td>
<td>92.82</td>
</tr>
<tr>
<td>All types</td>
<td>96.09</td>
<td>96.72</td>
</tr>
</tbody>
</table>

Table 5: Statistics of Benchmarks used as Case Studies.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Graph Size</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>903</td>
<td>2564</td>
<td>11241</td>
</tr>
<tr>
<td>MG</td>
<td>1228</td>
<td>6299</td>
<td>28354</td>
</tr>
<tr>
<td>LULESH</td>
<td>3144</td>
<td>12512</td>
<td>44226</td>
</tr>
<tr>
<td>LBM</td>
<td>1086</td>
<td>6500</td>
<td>32046</td>
</tr>
</tbody>
</table>

4.4 RQ3: Case Studies

4.4.1 Experimental Setup. We present four case studies to explore the usefulness of our FPLearner models in a real-world scenario, namely FP dynamic precision tuning. We consider CG and MG from the NAS C Parallel Benchmarks version 3.0 (NPB) [4], LULESH version 2.0 [33] from LLNL, and LBM from the SPEC CPU 2017 Benchmarks [8]. Table 5 lists their sizes in lines of code (LOC), graph size, and average runtime. These programs are commonly used in precision tuning evaluation and represent the largest reported in the existing literature. Notably, the number of FP variables in LULESH, i.e., 365, is considerably larger compared to others, resulting in a significantly larger search space for precision tuning. Additionally, LBM exhibits a significantly longer execution time, emphasizing the need to minimize program runs for reducing time cost.

The choice of program inputs and error thresholds for each program can vary across different usage scenarios. A more experienced user might be more selective on the program inputs and the error thresholds to use [54]. For CG and MG, we use the provided input Class A. For LULESH, we use the default program size 30 × 30 for each spatial problem domain. And for LBM, we follow the standard reference workload to run the program. These representative inputs achieve a 85% code coverage on average. For CG, MG, and LULESH,
we set the computation error threshold to be $10^{-4}$, while for LBM we use $10^{-7}$, for which a larger speedup is found when using a smaller (more restrictive) error threshold.

We evaluate our models on two dynamic precision tuners: Precimonious [54] and HiFPTuner [25], which we refer to as Vanilla Precision Tuners. Precimonious utilizes delta debugging [67], which has been recognized as the most effective search strategy in recent precision tuning studies [16, 49]. Besides, Precimonious has served as the one and only dynamic tuning baseline for many of the latest state-of-the-art precision tuners [23, 25, 36, 53]. More recent state-of-the-art tuners that apply a trial-and-error paradigm include Blame [53], HiFPTuner [25], Promise [23], PyFLOT [7], and AMPT-GA [36]. HiFPTuner is selected over Blame because it is more recent. Promise and PyFLOT require additional runtime information that makes them unsuitable for our evaluation. Finally, while conceptually AMPT-GA could benefit from our models, it is designed for CUDA programs, and it is not publicly available.

During the pre-run stage (Section 3.3), we run the Vanilla Precision Tuners to collect the initial fine-tuning datasets: the first 100 mixed-precision programs for CG and MG, the first 500 mixed-precision programs for LULESH, and the first 80 mixed-precision programs for CG and MG, the first 500 mixed-precision programs for LULESH, and the first 70 mixed-precision programs for LBM.

We consider three settings in our experiments:

1. **Vanilla Precision Tuners**: the original precision tuner that executes every candidate mixed-precision program explored during the search to evaluate its performance and accuracy with respect to the given error threshold.

2. **Precision Tuner + FPLearner**: the precision tuner enhanced with our ML models. Specifically, the precision tuner’s search is guided by the models’ predictions. However, only “promising” mixed-precision programs are executed, i.e., those programs predicted by the models to be both faster than the original program and to produce a result within the given error threshold. Note that “promising” programs must still be run because the goal is to find the program with the highest speedup. As a result, we not only verify the predictions but also obtain the actual speedup when our models make correct decisions.

3. **Precision Tuner w/o FT**: same process as (2) except that the models employed are trained from scratch on the target programs instead of applying model fine tuning.

### 4.4.2 Evaluation Metrics

We compare the settings with respect to:

*Definition 4.1. (Program Quality). A mixed-precision program $P_1$ is better than a program $P_2$ if $P_1$ achieves a larger performance speedup that meets the accuracy requirement than $P_2$.***

*Definition 4.2. (Search Effectiveness). A precision tuner $T_1$ is more effective than the precision tuner $T_2$ if the final mixed-precision program generated by $T_1$ is better than that found by $T_2$.***

*Definition 4.3. (Search Efficiency). A precision tuner $T_1$ is more efficient than the precision tuner $T_2$ if $T_1$ generates an equivalent or a better mixed-precision program than $T_2$ with fewer program runs.***

Our ideal goal is to discover a mixed-precision program that achieves a speedup equivalent to that found by the Vanilla Precision Tuner in fewer runs. However, small runtime variation or mispredictions may lead to different search paths, which ultimately may result in different local minima being found. These variations are

### Table 6: Case Studies. Time Cost Represented in hh:mm:ss. FT: Fine-tuning.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Precision Tuner Setting</th>
<th>#Programs</th>
<th>#Runs</th>
<th>%Runs</th>
<th>Final Speedup</th>
<th>Training Cost</th>
<th>Search Cost</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Vanilla P</td>
<td>532</td>
<td>532</td>
<td>100.00%</td>
<td>25.50%</td>
<td>-</td>
<td>01:20:27</td>
<td>01:20:27</td>
</tr>
<tr>
<td></td>
<td>P + FPLearner w/o FT</td>
<td>464</td>
<td>198</td>
<td>42.67%</td>
<td>26.56%</td>
<td>00:05:40</td>
<td>00:51:03</td>
<td>00:56:43*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>455</td>
<td>197</td>
<td>43.30%</td>
<td>21.96%</td>
<td>00:33:20</td>
<td>00:49:32</td>
<td>01:22:52</td>
</tr>
<tr>
<td></td>
<td>HiFPTuner (H)</td>
<td>388</td>
<td>388</td>
<td>100.00%</td>
<td>27.47%</td>
<td>-</td>
<td>00:52:01</td>
<td>00:52:01</td>
</tr>
<tr>
<td></td>
<td>H + FPLearner w/o FT</td>
<td>414</td>
<td>156</td>
<td>37.68%</td>
<td>27.62%</td>
<td>00:06:05</td>
<td>00:33:29</td>
<td>00:39:54*</td>
</tr>
<tr>
<td></td>
<td>H + FPLearner</td>
<td>433</td>
<td>178</td>
<td>41.11%</td>
<td>23.81%</td>
<td>00:31:19</td>
<td>00:43:25</td>
<td>01:14:44</td>
</tr>
<tr>
<td>MG</td>
<td>Vanilla P</td>
<td>570</td>
<td>570</td>
<td>100.00%</td>
<td>14.69%</td>
<td>-</td>
<td>01:13:47</td>
<td>01:13:47</td>
</tr>
<tr>
<td></td>
<td>P + FPLearner w/o FT</td>
<td>356</td>
<td>214</td>
<td>60.11%</td>
<td>15.02%</td>
<td>00:07:28</td>
<td>01:05:44</td>
<td>01:13:12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>482</td>
<td>246</td>
<td>51.04%</td>
<td>13.98%</td>
<td>00:29:52</td>
<td>01:20:18</td>
<td>01:50:10</td>
</tr>
<tr>
<td></td>
<td>HiFPTuner (H)</td>
<td>438</td>
<td>438</td>
<td>100.00%</td>
<td>14.07%</td>
<td>-</td>
<td>00:57:32</td>
<td>00:57:32</td>
</tr>
<tr>
<td></td>
<td>H + FPLearner w/o FT</td>
<td>317</td>
<td>149</td>
<td>47.00%</td>
<td>14.54%</td>
<td>00:07:36</td>
<td>00:40:13</td>
<td>00:47:49*</td>
</tr>
<tr>
<td>LULESH</td>
<td>Vanilla P</td>
<td>2564</td>
<td>2564</td>
<td>100.00%</td>
<td>18.13%</td>
<td>-</td>
<td>20:02:34</td>
<td>20:02:34</td>
</tr>
<tr>
<td></td>
<td>P + FPLearner w/o FT</td>
<td>2428</td>
<td>960</td>
<td>39.54%</td>
<td>21.31%</td>
<td>01:13:04</td>
<td>15:36:16</td>
<td>16:43:20*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2859</td>
<td>1281</td>
<td>44.95%</td>
<td>19.47%</td>
<td>02:42:24</td>
<td>18:48:57</td>
<td>21:31:81</td>
</tr>
<tr>
<td></td>
<td>HiFPTuner (H)</td>
<td>994</td>
<td>394</td>
<td>100.00%</td>
<td>23.73%</td>
<td>-</td>
<td>05:15:03</td>
<td>05:15:03</td>
</tr>
<tr>
<td></td>
<td>Vanilla H</td>
<td>957</td>
<td>424</td>
<td>45.25%</td>
<td>23.33%</td>
<td>01:04:32</td>
<td>02:56:38</td>
<td>04:01:10</td>
</tr>
<tr>
<td></td>
<td>H + FPLearner w/o FT</td>
<td>731</td>
<td>349</td>
<td>47.74%</td>
<td>20.41%</td>
<td>02:22:56</td>
<td>02:17:09</td>
<td>04:40:05</td>
</tr>
<tr>
<td>LBM</td>
<td>Vanilla P</td>
<td>316</td>
<td>316</td>
<td>100.00%</td>
<td>18.42%</td>
<td>-</td>
<td>20:51:12</td>
<td>20:51:12</td>
</tr>
<tr>
<td></td>
<td>P + FPLearner w/o FT</td>
<td>223</td>
<td>132</td>
<td>59.19%</td>
<td>21.35%</td>
<td>00:08:16</td>
<td>07:58:41</td>
<td>08:06:57*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>486</td>
<td>256</td>
<td>52.67%</td>
<td>14.32%</td>
<td>00:51:40</td>
<td>14:01:17</td>
<td>14:52:57</td>
</tr>
<tr>
<td></td>
<td>HiFPTuner (H)</td>
<td>217</td>
<td>217</td>
<td>100.00%</td>
<td>17.38%</td>
<td>-</td>
<td>09:54:45</td>
<td>09:54:45</td>
</tr>
<tr>
<td></td>
<td>H + FPLearner w/o FT</td>
<td>222</td>
<td>144</td>
<td>64.86%</td>
<td>20.06%</td>
<td>00:09:12</td>
<td>06:44:56</td>
<td>06:54:08*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>322</td>
<td>140</td>
<td>43.48%</td>
<td>17.14%</td>
<td>00:57:30</td>
<td>06:55:04</td>
<td>07:52:34</td>
</tr>
</tbody>
</table>
acceptable as long as the resulting program has a speedup comparable to that reported by the Vanilla Precision Tuner. Here we define “comparable” as having a value difference of less than 0.5%.

4.4.3 Experimental Results. Table 6 shows the results for the four case studies. #Programs is the number of candidate mixed-precision programs explored during the search, while #Runs indicates the number of programs actually executed. #Runs is the result from dividing #Programs by #Runs in the search. Final Speedup refers to the performance speedup achieved by the final mixed-precision program recommended by the precision tuner. Training Cost for FPLearner is the time taken for model fine tuning, whereas for FPLearner w/o FT is the time required for training the models from scratch. In addition, Search Cost refers to the time cost of the full precision tuning process. Note that #Programs, #Runs and Search Cost include the pre-run stage, during which the original precision tuners are used to gather the initial set of mixed-precision programs, and the optimization stage, in which our models are used during the search to predict performance and accuracy, as described in Section 3.3. Lastly, Total Cost is the overall time, which is composed of Training Cost and Search Cost.

Here we compare Vanilla Precision Tuners with Precision Tuner + FPLearner. We will further explore the comparison with Precision Tuner + FPLearner w/o FT in Section 4.5. Across all case studies, FPLearner achieves a 35.14%–62.32% reduction in program runs compared to the total number of programs in the search. When compared to Vanilla Precision Tuners, using FPLearner successfully reduces program runs by 57.34%–65.98%. Total time cost reductions are observed in all cases (17.18%–61.07%) except for MG, for which using FPLearner achieves comparable time cost with Vanilla Precimonious. The most significant cost reduction is observed in LBM, which has the longest running time. This serves as evidence that FPLearner is particularly well-suited for programs with relatively large runtime. Finally, compared to Vanilla Precision Tuners, using FPLearner yields comparable or slightly superior results in terms of final program speedup, which is expected as predictions are not meant to deliberately make different search choices. This confirms the effectiveness of our predictions.

Response to RQ3: Our models improve the time efficiency of precision tuners by an average of 25.54% and up to 61.07% while generating mixed-precision programs of comparable or better quality, proving useful in both efficiency and effectiveness.

4.5 RQ4: Model Parameter Transferability

4.5.1 Experimental Setup. We measure the effectiveness of the model parameter transferability in two settings: (1) FPLearner, i.e., fine-tuning the pre-trained performance and accuracy prediction models on the dataset of the target benchmark, and (2) FPLearner w/o FT, i.e., training the same NN architectures from scratch on the same target dataset. We compare these two settings in terms of Model Performance, Search Effectiveness, and Search Efficiency. We do not consider the scenario in which our pre-trained models are used to make predictions on new benchmarks without fine-tuning. We find that in all such cases the models are not capable of generating reliable predictions.

4.5.2 Training and Testing Details. For FPLearner, we use the same fine-tuning methodology as in prior work [27, 32, 68] to avoid overfitting when the dataset size of target benchmarks is limited, by selecting a small number of epochs (less than 10) for training. We found 8 epochs to be a good default for fine-tuning our models on all four target benchmarks, resulting in validation accuracy exceeding 80%. However, we observe that training FPLearner w/o FT for the same duration of epochs proves insufficient. Specifically, after 8 epochs, the models trained from scratch tend to classify the majority of data examples into a single class, leading to a low validation accuracy. For a fair comparison, we continue to train FPLearner w/o FT for a maximum number of 50 epochs with early stopping [52], and terminate the training when its validation accuracy is equal to or larger than that of the fine-tuned models. We use the same set of unseen programs when testing the generalizability of both FPLearner and FPLearner w/o FT. Specifically, for each benchmark, we report the performance of the models on the set of mixed-precision programs that are explored during the optimization stage of the Precimonious + FPLearner setting.

4.5.3 Experimental Results and Discussion. Figure 4 demonstrates that fine-tuning our pre-trained models on all target programs yields a substantial improvement in model performance of up to 31.9% when compared to training from scratch. This finding proves the transferability of the knowledge learned from existing programs to new programs. The superiority in Model Performance is reflected in the fine-tuned FPLearner achieving better Search Effectiveness as shown in Table 6. At the same time, FPLearner w/o FT requires 3.83× training cost on average than our fine-tuned FPLearner. Based on these experimental results, we conclude that leveraging...
the fine-tuning technique is beneficial in compensating for the lack of sufficient training data in the target benchmarks.

Response to RQ4: Transferring parameters from pre-trained models to new programs can significantly save time and prove to be more effective than training without any prior knowledge.

4.6 Threats to Validity
Our primary external threat is the extent to which our results can be generalized. We address this by (1) training our models on programs from HPC-MixP Bench, a representative benchmark for mixed-precision analysis in HPC workloads, (2) employing a fine-tuning technique for adapting to new numerical benchmarks that proves its effectiveness, and (3) conducting case studies in benchmarks widely-used in precision tuning, and the largest reported in the literature. Secondly, our evaluation focuses on double and single precision in C/C++ languages, but FP LEARNER can be extended to support other precisions and languages. Finally, there is an internal threat in selecting suitable baselines for predicting performance and accuracy of mixed-precision programs as no existing work addresses our research goal. We carefully selected representative neural network architectures from [31], [43] and [56] based on their potential in learning mixed-precision program features.

5 RELATED WORK
A substantial portion of precision tuners relies on dynamic analysis. PRECIMONIOUS [54] is a search-based precision tuner that uses the delta-debugging algorithm to explore mixed-precision programs. BLAME ANALYSIS [35] uses shadow execution to prune its search space. HIPPTUNER [25] extends PRECIMONIOUS to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.

To capture code structure, there have been an increasing number of research work that utilizes GNNs to learn graph representations of programs. Allamanis et al. [2] predict variable names and miss-usage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, PROMISE [23] uses Discrete Stochastic Arithmetic, while PyFlO T [7] uses call stack information and temporal locality. AMPT-GA [36] performs precision optimization for GPU kernels in a genetic algorithm-based search. All of the above face scalability limitations given the exponential nature of the search space. ADAPT [45] provides a precision sensitivity analysis as a guide for precision tuning, but it still relies on program execution. Different from all the above work, we are the first to utilize a DL-based approach to replace program execution with a model prediction to improve the efficiency of precision tuning.