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 HiFPTuner, by an average of 25.54% and up to 61.07% while achieving precision tuning results of comparable or better quality.

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:
Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623338

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623338

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.

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 61.07% in time cost while generating a mixed-precision program of comparable or better quality.

In summary, our paper makes the following contributions:

- We design a novel graph representation to model precision interactions in mixed-precision programs (Section 3.1).
- We deploy a GNN architecture highly suitable for learning features from the graph representation of mixed-precision programs (Section 3.2), and describe how the models can be integrated into existing precision tuners (Section 3.3).
- We construct training datasets of mixed-precision programs from five diverse HPC applications (Section 4.1).
- We present an evaluation that compares FPLearner models to popular baselines and measures our design choices in program representation. Furthermore, we demonstrate the benefits of integrating our prediction models into state-of-the-art precision tuners (Section 4).

2 MOTIVATION

This section describes dynamic precision tuning, and provides an example to emphasize the demand for predicting performance and accuracy of mixed-precision programs.

Dynamic Precision Tuning. Given a target FP program, the dynamic FP precision tuning process seeks to find a lower-precision variant of the program, often a mixed-precision program, that improves performance while adhering to specified computation accuracy constraints. The majority of existing precision tuners [5, 23, 25, 38, 53, 54] rely on a search-based approach with a trial-and-fail paradigm. 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 3.76×10^{107} 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 2564×18 seconds, which equals to 13 hours. This is a large amount of time compared to the running time of the

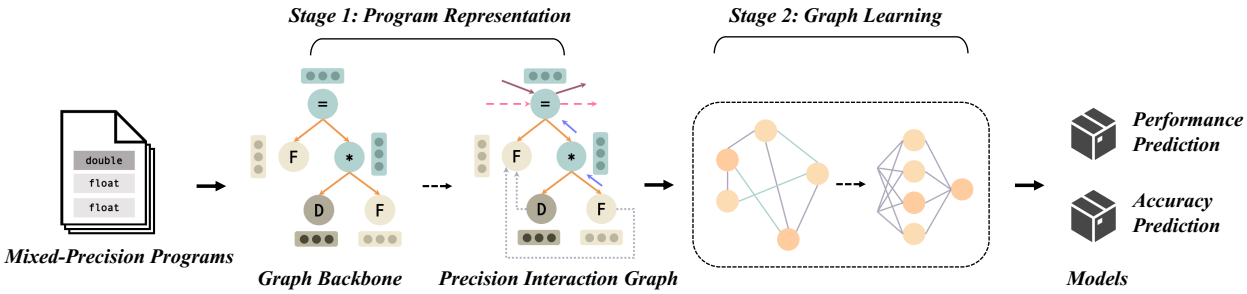


Figure 1: Overview of FPLearner.

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).

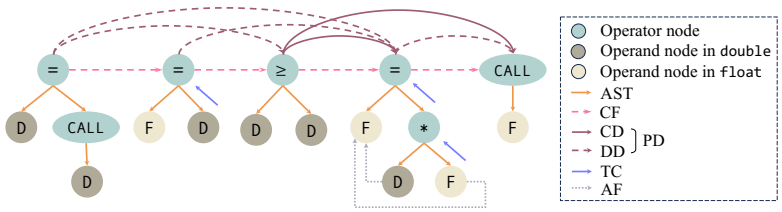
3.1.1 Graph Backbone and its Node Representation. FPLearner starts by constructing the AST of the mixed-precision program, whose nodes and edges serve as the foundation for the PIG. An AST is an ordered tree where inner nodes represent *operators* and leaf nodes represent *operands* [64]. Each statement 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 mathematic 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.

```
1  void foo() {
2    double a = sqrt(1.1);
3    float b = 2.0;
4    if (a >= 1.3952) {
5      float c = a * b;
6      update(c);
7    }
8  }
```

(a) A sample mixed-precision program.



(b) A sample Precision Interaction Graph (PIG).

Figure 2: A code sample and its PIG.

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 "∗" 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 "`sqrtf`" (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

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 $v \in \{1, 2, ..., |V|\}$, and an edge set E where an edge is defined as $e = (v, v') \in V \times V$. 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 GGNN architecture, is determined by the subsequent recurrence:

$$h_v^{(0)} = x_v, v \in \{1, 2, \ldots, |V|\} \tag{1}$$

$$m_{v,q}^{(l+1)} = \sum_{v' \in N_q(v)} \Theta_q \cdot h_{v'}^{(l)} \tag{2}$$

$$h_v^{(l+1)} = \text{GRU}\left(h_v^{(l)}, \sum_{q=1}^{|R|} m_{v,q}^{(l+1)}\right) \tag{3}$$

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 Θ_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 $v \in V$.

Table 1: Notations and Explanations.

Notation	Explanation
x_v	Initial feature vector of node v
l	Hidden layer $l \in \{1, 2, ..., L\}$
h_v	Hidden state vector of node v
q	Edge type $q \in \{1, 2, ..., \|R\|\}$
$N_q(v)$	Neighbors of node v for its outgoing edges in terms of q
Θ_q	Learnable parameters in terms of edge type and direction
$m_{v,q}$	Message vector for node v in terms of edge type q
N	The number of total data examples in the training set
p_i	Mixed-precision program, $i \in \{1, 2, ..., N\}$
y_i	Class label of the corresponding program p_i

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

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

Once the GGNN 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 $Prob(p_i)$ of program p_i. The output value $Prob(p_i)$ will then be used to calculate the Binary Cross Entropy (BCE) loss:

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log\,\text{Prob}\,(p_i) + (1 - y_i) log(1 - \text{Prob}\,(p_i)) \tag{5}$$

where $y_i = \{0, 1\}$ and $Prob(p_i)$ 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 GGNN 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 `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 FPLearner 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.

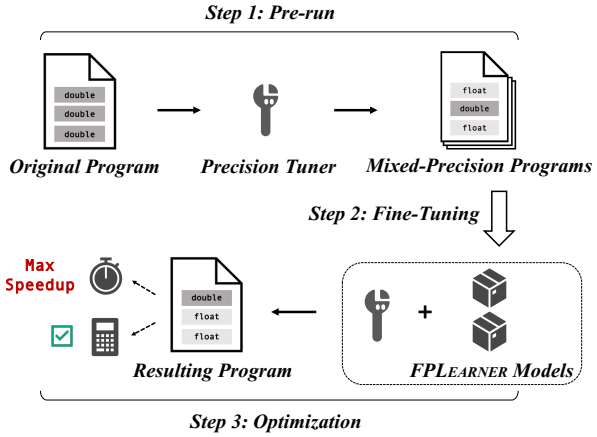


Figure 3: Dynamic Precision Tuning with FPLearner.

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 FPLearner 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 FPLearner 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++: Blackscholes [6], CFD [11], Hotspot [11], HPCCG [29], and LavaMD [11]. These programs are part of HPC-MixPBench [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].

[1]For example, work on vulnerability detection [70] reports graph node size no larger than 500 when representing functions from large C projects such as the Linux kernel.

Table 2: Mixed-Precision Program Dataset Overview.

Application	LOC	#Progrs/ #Graphs	Graph Size	
			Avg. #Nodes	Avg. #Edges
Blackscholes	287	760	2237	5215
CFD	648	1798	4713	18809
Hotspot	302	504	1254	7581
HPCCG	287	552	2676	6121
LavaMD	288	348	2416	6122
Datasets				
Performance	-	628	3195	11487
Accuracy	-	600	3191	11597

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 PIG 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.

Response to RQ1: Benefitting from the graph representation PIG and the NN architecture GGNN, our approach proves to be effective in accurately predicting performance (96.34% F1 score) and accuracy (97.03% F1 score) of mixed-precision programs, which outperforms other baseline methods.

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.

Response to RQ2: Our ablation study shows that each type of edge provides a distinct context for learning the FP precision interactions, and thus improves the effectiveness of the graph representation for mixed-precision programs.

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

Approach	Performance Prediction				Accuracy Prediction			
	A (%)	P (%)	R (%)	F1 (%)	A (%)	P (%)	R (%)	F1 (%)
LSTM	70.05	69.28	69.49	68.45	71.09	71.85	71.22	68.23
BiLSTM	80.31	86.20	77.90	81.84	84.38	84.92	85.59	85.25
RGCN	85.16	88.01	85.26	86.62	82.81	84.33	84.03	84.18
GGNN	**96.09**	**96.72**	**95.96**	**96.34**	**96.88**	**97.24**	**96.82**	**97.03**

Table 4: Impact of Distinct Edges. A: Accuracy, P: Precision, R: Recall, F1: F1 score. CF: Control Flow, PD: Program Dependence, TC: TypeCasting, AF: AssignedFrom.

Edges	Performance Prediction				Accuracy Prediction			
	A (%)	P (%)	R (%)	F1 (%)	A (%)	P (%)	R (%)	F1 (%)
No AST	89.84	91.66	89.84	90.74	89.06	90.21	90.16	90.19
No CF	83.54	86.87	83.59	85.19	88.28	89.75	88.84	89.29
No PD	85.16	87.23	85.26	86.23	89.84	91.80	81.88	81.87
No TC	89.06	91.05	89.22	90.12	92.19	93.35	92.73	93.04
No AF	90.63	92.82	90.54	91.66	91.41	92.78	91.69	92.23
All types	**96.09**	**96.72**	**95.96**	**96.34**	**96.88**	**97.24**	**96.82**	**97.03**

Table 5: Statistics of Benchmarks used as Case Studies.

Benchmark	LOC	Graph Size		Execution Time
		#Nodes	#Edges	
CG	903	2564	11241	1.38s
MG	1228	6299	28354	1.25s
LULESH	3144	12512	44226	18.56s
LBM	1086	6500	32046	269.58s

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,

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

Benchmark	Precision Tuner	Setting	#Programs	#Runs	%Runs	Final Speedup	Training Cost	Search Cost	Total Cost
CG	Precimonious (P)	Vanilla P	532	532	100.00%	25.50%	-	01:20:27	01:20:27
		P + FPLearner	464	198	42.67%	**26.56%**	00:05:40	00:51:03	**00:56:43** (↓ 29.50%)
		P + FPLearner w/o FT	455	197	43.30%	21.96%	00:33:20	00:49:32	01:22:52
	HiFPTuner (H)	Vanilla H	388	388	100.00%	27.47%	-	00:52:01	00:52:01
		H + FPLearner	414	156	37.68%	**27.62%**	00:06:05	00:33:29	**00:39:34** (↓ 23.93%)
		H + FPLearner w/o FT	433	178	41.11%	23.81%	00:31:19	00:43:25	01:14:44
MG	Precimonious (P)	Vanilla P	570	570	100.00%	14.69%	-	01:13:47	01:13:47
		P + FPLearner	356	214	60.11%	**15.02%**	00:07:28	01:05:44	**01:13:12** (Comparable)
		P + FPLearner w/o FT	482	246	51.04%	13.98%	00:29:52	01:20:18	01:50:10
	HiFPTuner (H)	Vanilla H	438	438	100.00%	14.07%	-	00:57:32	00:57:32
		H + FPLearner	317	149	47.00%	**14.54%**	00:07:36	00:40:13	**00:47:49** (↓ 17.18%)
		H + FPLearner w/o FT	370	202	54.59%	14.39%	00:29:59	00:45:25	01:15:24
LULESH	Precimonious (P)	Vanilla P	2564	2564	100.00%	18.13%	-	20:02:34	20:02:34
		P + FPLearner	2428	960	39.54%	**21.31%**	01:13:04	15:30:16	**16:43:20** (↓ 18.75%)
		P + FPLearner w/o FT	2850	1281	44.95%	19.47%	02:42:24	18:48:37	21:31:01
	HiFPTuner (H)	Vanilla H	994	994	100.00%	**23.73%**	-	05:15:05	05:15:05
		H + FPLearner	937	424	45.25%	23.33%	01:04:32	02:56:38	**04:01:10** (↓ 23.49%)
		H + FPLearner w/o FT	731	349	47.74%	20.41%	02:22:56	02:17:09	04:40:05
LBM	Precimonious (P)	Vanilla P	316	316	100.00%	18.42%	-	20:51:12	20:51:12
		P + FPLearner	223	132	59.19%	**21.35%**	00:08:16	07:58:41	**08:06:57** (↓ 61.07%)
		P + FPLearner w/o FT	486	256	52.67%	14.32%	00:51:40	14:01:17	14:52:57
	HiFPTuner (H)	Vanilla H	217	217	100.00%	17.38%	-	09:54:45	09:54:45
		H + FPLearner	222	144	64.86%	**20.06%**	00:09:12	06:44:56	**06:54:08** (↓ 30.37%)
		H + FPLearner w/o FT	322	140	43.48%	17.14%	00:57:30	06:55:04	07:52:34

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 LBM. To measure program speedup, we execute each mixed-precision program of CG and MG ten times and report their average. We notice that LULESH and LBM are less sensitive to performance noise given their larger runtime. Thus, we only report the average of five runs for LULESH, and one run for LBM.

We consider three settings in our experiments:

(1) *Vanilla Precision Tuner:* 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 + FPLearner 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.

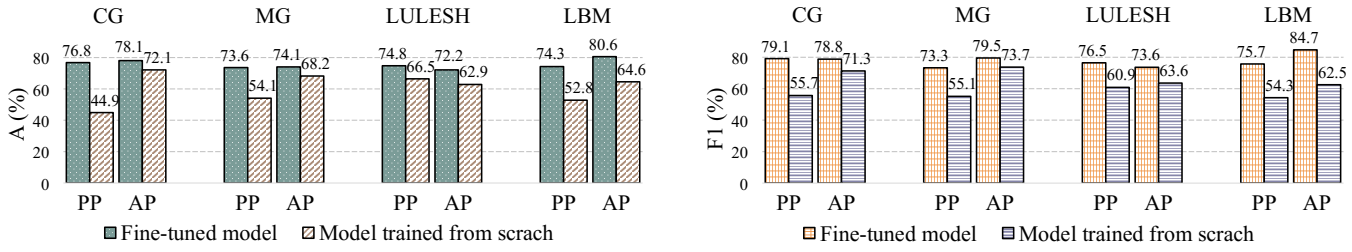
We compare the settings with respect to:

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



(a) A: Accuracy. **(b) F1: F1 score.**

Figure 4: Fine-tuning vs. Training from Scratch. PP: Performance Prediction, AP: Accuracy Prediction.

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 *#Runs* by *#Programs* 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-MixPBench, 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 FPLearner 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 [53] uses shadow execution to prune its search space. HIFPTuner [25] extends Precimonious to improve the search efficiency via hierarchy construction. Gathering dynamic program behavior as feedback, Promise [23] uses Discrete Stochastic Arithmetic, while PyFloT [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 misusage by learning from a syntax tree with data-flow information. Dinella et al. [17] use GGNNs to detect and fix bugs in JavaScript programs. TehraniJamsaz et al. [59] leverage code region graphs to learn intermediate representations for NUMA/prefetcher optimizations. Several works [9, 62, 70] target vulnerability detection and their graph representations typically contain control-flow and data-flow information. In contrast, FPLearner is the first to predict both performance and accuracy of numerical software that uses mixed precision. Moreover, we proposed a distinct graph representation, PIG, specialized for such programs by modeling precision interactions among FP variables across the program.

A large body of work has applied ML to perform various SE tasks such as program repair [41, 44], functional code clone detection [18], defect prediction [12], patch correctness prediction [60], name-based bug detection [51], and type inference [28]. Our work fills the gap by utilizing an ML-based method to learn features from mixed-precision programs in the numerical software domain.

6 CONCLUSION

We presented FPLearner, a novel approach for predicting the performance and accuracy of mixed-precision programs. We proposed an effective graph representation PIG for mixed-precision programs, and utilized GNNs, an advanced ML technique, to learn features from their graph representation. By incorporating our prediction models into the dynamic precision-tuning process, we are able to save time that would otherwise be spent on running programs. Our evaluation demonstrated that FPLearner models produce highly accurate predictions and significantly enhance the efficiency of precision tuners. Through the creation of a diverse dataset containing 1228 mixed-precision programs from five HPC applications, our models achieved a 96.34% F1 score in performance prediction and a 97.03% F1 score in accuracy prediction. Moreover, FPLearner substantially improved time efficiency in two dynamic precision tuners, Precimonious and HiFPTuner, boasting an average enhancement of 25.54% and reaching up to 61.07%, all while maintaining precision tuning results of comparable or superior quality. Our code, documentation and experimental data are publicly available at https://github.com/ucd-plse/FPLearner.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under award CCF-1750983, and the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under awards DE-SC0020286 and DE-SC0022182.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 91–105. https://doi.org/10.1145/3385412.3385997

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[3] Marc Baboulin, Alfredo Buttari, Jack J. Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. 2009. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.* 180, 12 (2009), 2526–2533. https://doi.org/10.1016/j.cpc.2008.11.005

[4] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. The NAS Parallel Benchmarks 2.0.

[5] Dorra Ben Khalifa and Matthieu Martel. 2023. Everything you Need to Know About Reduced Mixed Precision Computation in Numerical Programs. (2023). https://hal.science/hal-03978176 preprint.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, Andreas Moshovos, David Tarditi, and Kunle Olukotun (Eds.). ACM, 72–81. https://doi.org/10.1145/1454115.1454128

[7] Hugo Brunie, Costin Iancu, Khaled Z. Ibrahim, Philip Brisk, and Brandon Cook. 2020. Tuning floating-point precision using dynamic program information and temporal locality. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 50. https://doi.org/10.1109/SC41405.2020.00054

[8] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar (Eds.). ACM, 41–42. https://doi.org/10.1145/3185768.3185771

[9] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1456–1468. https://doi.org/10.1145/3510003.3510219

[10] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. 2020. Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead. *IEEE Access* 8 (2020), 225134–225180. https://doi.org/10.1109/ACCESS.2020.3039858

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[12] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, Republic of Korea, May 23-29, 2020*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM. https://doi.org/10.1145/1122445.1122456

[13] Stefano Cherubin and Giovanni Agosta. 2021. Tools for Reduced Precision Computation: A Survey. *ACM Comput. Surv.* 53, 2 (2021), 33:1–33:35. https://doi.org/10.1145/3381039

[14] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 300–315. http://dl.acm.org/citation.cfm?id=3009846

[15] Savelie Cornegruta, Robert Bakewell, Samuel Withey, and Giovanni Montana. 2016. Modelling Radiological Language with Bidirectional Long Short-Term Memory Networks. In *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis, Louhi@EMNLP 2016, Austin, TX, USA, November 5, 2016*, Cyril Grouin, Thierry Hamon, Aurélie Névéol, and Pierre Zweigenbaum (Eds.). Association for Computational Linguistics, 17–27. https://doi.org/10.18653/v1/W16-6103

[16] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, Chris Gill, Bruno Sinopoli, Xue Liu, and Paulo Tabuada (Eds.). IEEE Computer Society / ACM, 208–219. https://doi.org/10.1109/ICCPS.2018.00028

[17] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[18] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 516–527. https://doi.org/10.1145/3395363.3397362

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[20] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[21] Qichuan Geng, Zhong Zhou, and Xiaochun Cao. 2018. Survey of recent progress in semantic image segmentation with CNNs. *Sci. China Inf. Sci.* 61, 5 (2018), 051101:1–051101:18. https://doi.org/10.1007/s11432-017-9189-6

[22] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 580–587. https://doi.org/10.1109/CVPR.2014.81

[23] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. 2019. Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. *J. Comput. Sci.* 36 (2019). https://doi.org/10.1016/j.jocs.2019.07.004

[24] Hui Guo, Ignacio Laguna, and Cindy Rubio-González. 2020. pLiner: isolating lines of floating-point code for compiler-induced variability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 49. https://doi.org/10.1109/SC41405.2020.00053

[25] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 333–343. https://doi.org/10.1145/3213846.3213862

[26] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 3–14. https://doi.org/10.1109/ICSE.2017.9

[27] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogério Schmidt Feris. 2019. SpotTune: Transfer Learning Through Adaptive Fine-Tuning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 4805–4814. https://doi.org/10.1109/CVPR.2019.00494

[28] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 152–162. https://doi.org/10.1145/3236024.3236051

[29] Michael A. Heroux. 2007. HPCCG Solver Package, Version 00. https://www.osti.gov//servlets/purl/1230960

[30] Nhut-Minh Ho, Himeshi De Silva, and Weng-Fai Wong. 2021. GRAM: A Framework for Dynamically Mixing Precisions in GPU Applications. *ACM Trans. Archit. Code Optim.* 18, 2 (2021), 19:1–19:24. https://doi.org/10.1145/3441830

[31] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[32] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 328–339. https://doi.org/10.18653/v1/P18-1031

[33] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[34] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[35] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[36] Pradeep V. Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. 2019. AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, Rudolf Eigenmann, Chen Ding, and Sally A. McKee (Eds.). ACM, 160–170. https://doi.org/10.1145/3330345.3330360

[37] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. 2013. Automatically adapting programs for mixed-precision floating-point computation. In *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, Allen D. Malony, Mario Nemirovsky, and Samuel P. Midkiff (Eds.). ACM, 369–378. https://doi.org/10.1145/2464996.2465018

[38] Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. 2013. Dynamic floating-point cancellation detection. *Parallel Comput.* 39, 3 (2013), 146–155. https://doi.org/10.1016/j.parco.2012.08.002

[39] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[40] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05493

[41] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 602–614. https://doi.org/10.1145/3377811.3380345

[42] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[43] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[44] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[45] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 48:1–48:13. http://dl.acm.org/citation.cfm?id=3291720

[46] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2023. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. In *High Performance Computing - 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21-25, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13948)*, Abhinav Bhatele, Jeff R. Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer, 381–401. https://doi.org/10.1007/978-3-031-32041-5_20

[47] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[48] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. 2020. Machine learning with oversampling and undersampling techniques: overview study and experimental results. In *2020 11th international conference on information and communication systems (ICICS)*. IEEE, 243–248.

[49] Konstantinos Parasyris, Ignacio Laguna, Harshitha Menon, Markus Schordan, Daniel Osei-Kuffuor, Giorgis Georgakoudis, Michael O. Lam, and Tristan Vanderbruggen. 2020. HPC-MixPBench: An HPC Benchmark Suite for Mixed-Precision Analysis. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*. IEEE, 25–36. https://doi.org/10.1109/IISWC50251.2020.00012

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[51] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[52] Lutz Prechelt. 2012. Early Stopping - But When? In *Neural Networks: Tricks of the Trade - Second Edition*, Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller (Eds.). Lecture Notes in Computer Science, Vol. 7700. Springer, 53–67. https://doi.org/10.1007/978-3-642-35289-8_5

[53] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 1074–1085. https://doi.org/10.1145/2884781.2884850

[54] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 27:1–27:12. https://doi.org/10.1145/2503210.2503296

[55] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[56] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10843)*, Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam (Eds.). Springer, 593–607. https://doi.org/10.1007/978-3-319-93417-4_38

[57] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. 2021. A Survey on Machine Learning Techniques for Source Code Analysis. *CoRR* abs/2110.09610 (2021). arXiv:2110.09610 https://arxiv.org/abs/2110.09610

[58] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2019. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019), 2:1–2:39.

[59] Ali TehraniJamsaz, Mihail Popov, Akash Dutta, Emmanuelle Saillard, and Ali Jannesari. 2022. Learning Intermediate Representations using Graph Neural Networks for NUMA and Prefetchers Optimization. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 1206–1216. https://doi.org/10.1109/IPDPS53621.2022.00120

[60] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 981–992. https://doi.org/10.1145/3324884.3416532

[61] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[62] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[63] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Hkx6hANtwH

[64] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 590–604. https://doi.org/10.1109/SP.2014.44

[65] Yuzhe Yang, Kaiwen Zha, Ying-Cong Chen, Hao Wang, and Dina Katabi. 2021. Delving into Deep Imbalanced Regression. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11842–11851. http://proceedings.mlr.press/v139/yang21m.html

[66] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 3320–3328. https://proceedings.neurips.cc/paper/2014/hash/375c71349b295fbe2dcdca9206f20a06-Abstract.html

[67] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[68] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-Bert: Only Attention is Needed for Learning Graph Representations. *CoRR* abs/2001.05140 (2020). arXiv:2001.05140 https://arxiv.org/abs/2001.05140

[69] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

[70] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

[71] Shichao Zhu, Chuan Zhou, Shirui Pan, Xingquan Zhu, and Bin Wang. 2019. Relation Structure-Aware Heterogeneous Graph Neural Network. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, Jianyong Wang, Kyuseok Shim, and Xindong Wu (Eds.). IEEE, 1534–1539. https://doi.org/10.1109/ICDM.2019.00203