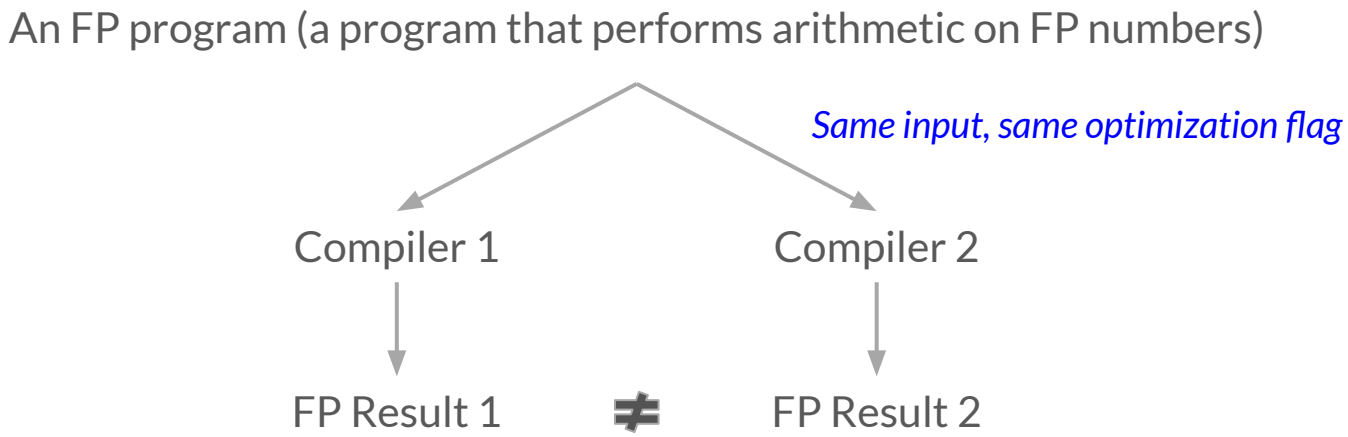
LLM4FP: LLM-Based Program Generation for Triggering Floating-Point Inconsistencies Across Compilers

Yutong Wang (UC Davis)
Cindy Rubio-González (UC Davis)

Correctness @ SC25
St. Louis, MO, USA, 11.17.2025

Floating-Point (FP) Inconsistencies in HPC



Such FP inconsistencies have been observed across major compilers like GCC, Clang, and NVCC with results differing by typically 1–3 digits on average in our LLM4FP experiments.

Challenges in Numerical Correctness

- Numerical bugs in widely used libraries
- Difficulty of ensuring numerical accuracy and reproducibility across heterogeneous computing systems

↓

Need a way to systematically uncover them before they affect real applications.

↓

Existing work: **Varity**[1] based on random program generation

[1] Ignacio Laguna. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. IPDPS 2020.

Challenges in Numerical Correctness

- Numerical bugs in widely used libraries
- Difficulty of ensuring numerical accuracy and reproducibility across heterogeneous computing systems

↓

Need a way to systematically uncover them before they affect real applications

↓

Can **LLMs** help us automate this process?

LLM4FP: *the **first** LLM-based framework to generate **FP** programs in C language to trigger numerical inconsistencies across compilers and/or optimization levels.*

Existing Tools on FP Inconsistencies

Cross-platform testing on existing programs (FLit[1])

Localizing inconsistencies in existing programs (pLiner[2], Ciel[3])

Program generation for inconsistency discovery (Varity[4])

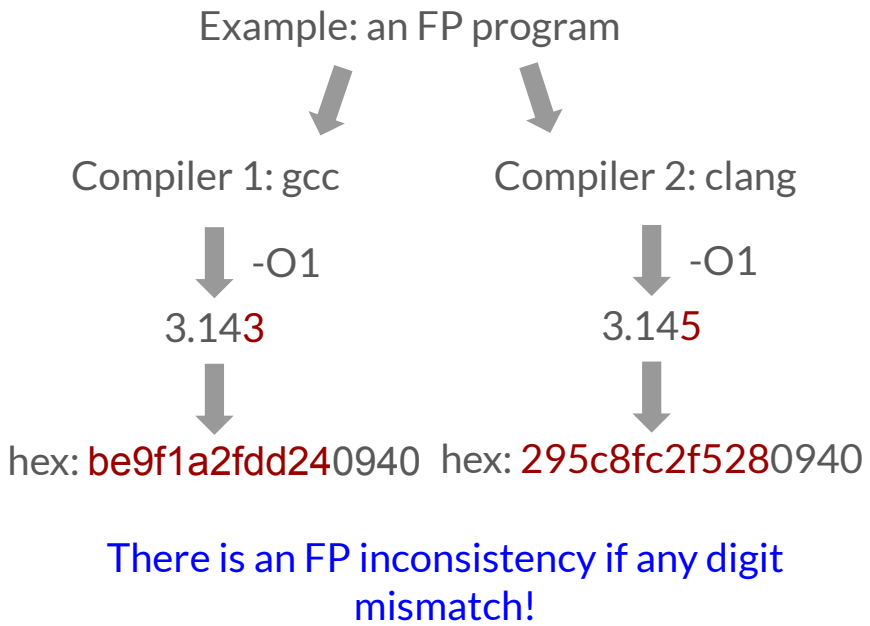
LLM4FP

Input generation for existing programs (S3FP[5], CIV[6], CIGEN[7])

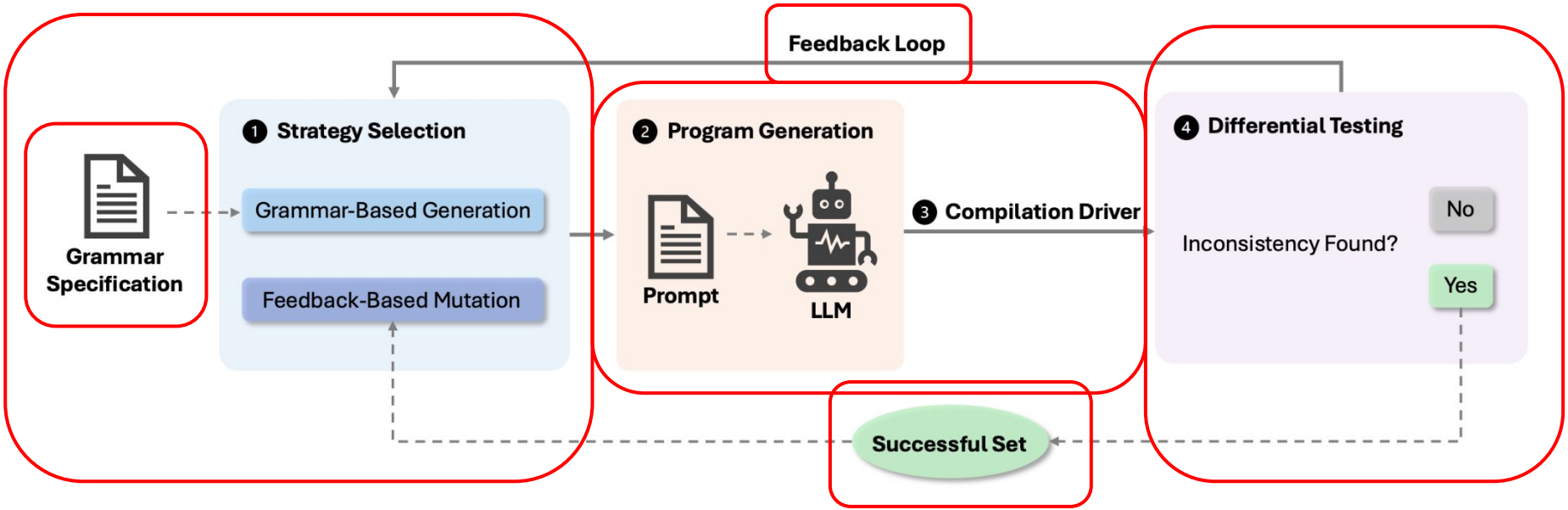
[1] Geoffrey Sawaya et al. FLiT: Cross-platform floating-point result-consistency tester and workload. IISWC 2017.
[2] Hui Guo et al. pLiner: isolating lines of floating-point code for compiler-induced variability. SC 2020.
[3] Dolores Miao. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. ISC 2023.
[4] Ignacio Laguna. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. IPDPS 2020.
[5] Wei-Fan Chiang et al. Efficient search for inputs causing high floating-point errors. PPoPP 2014.
[6] Hengbiao Yu et al. Efficient Generation of Floating-Point Inputs for Compiler-Induced Variability. SANER 2023.
[7] Dolores Miao et al. Input Range Generation for Compiler-Induced Numerical Inconsistencies. ICS 2024.

What is an FP inconsistency?

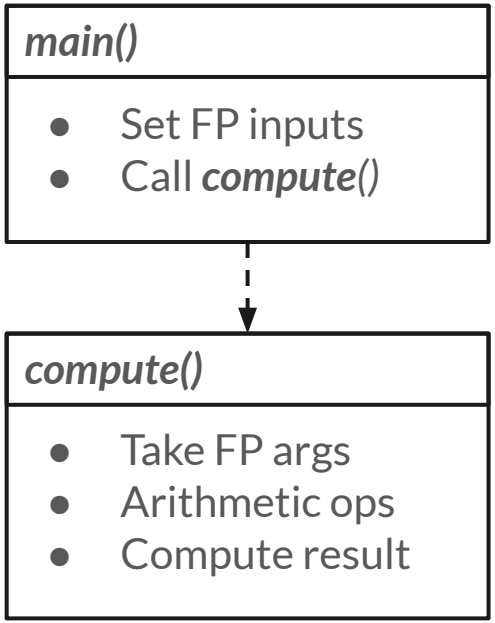
A floating-point inconsistency occurs when the **same program**, compiled with **different compilers** under the **same input and optimization level**, produces **different numerical results**.



LLM4FP Overview



High-Level FP Program Structure



- Each generated program contains two functions: ***main*** and ***compute***.
- ***main*** prepares FP inputs.
- ***compute*** performs arithmetic operations and prints out a scalar result.
- This consistent structure simplifies differential testing.

Internal Structure of *compute()*

Grammar that formally define the structure:

1) Different FP Types
2) Arithmetic Expressions
3) Loops
4) Conditions
5) Variables

```
<function> ::= "void" "compute"
"(" <param-list> ")" "{" <block> "}"

<param-list> ::= <param-declaration>
| <param-list> "," <param-declaration>

<param-declaration> ::= "int" <id>
| <fp-type> <id>
| <fp-type> "*" <id>

<assignment> ::= "comp" <assign-op> <expression> ";"
| <fp-type> <id> <assign-op>
<expression> ";"

<expression> ::= <term>
| "(" <expression> ")"
| <expression> <op> <expression>

<term> = <identifier> | <fp-numeral>

<block> ::= {<assignment>}+
| <if-block> <block>
| <for-loop-block> <block>

<if-block> ::= "if" "(" <bool-expression> ")"
"{" <block> "}"

<for-loop-block> ::= "for" "(" <loop-header> ")"
"{" <block> "}"

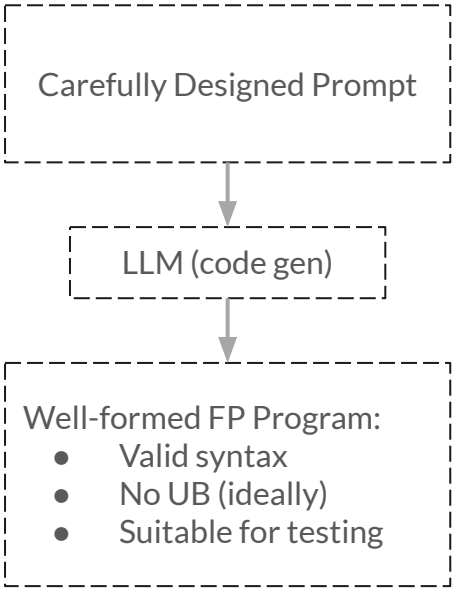
<bool-expression> ::= <id> <bool-op> <expression>

<loop-header> ::= "int" <id> ";" <id> "<" <int-numeral>
    ";" "++" <id>
```

Grammar Specification of compute() from Varity[1]

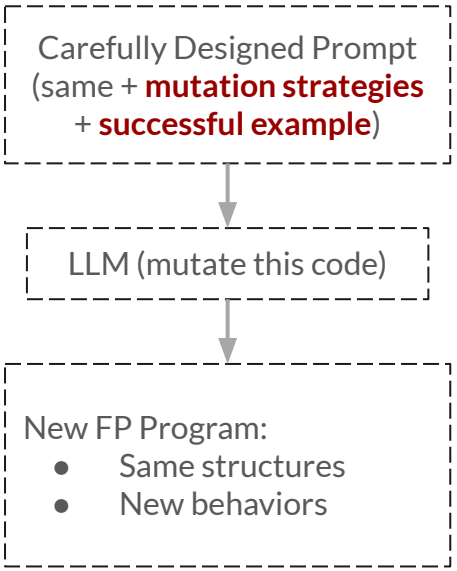
[1] Ignacio Laguna. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. IPDPS 2020.

Grammar-Based Program Generation



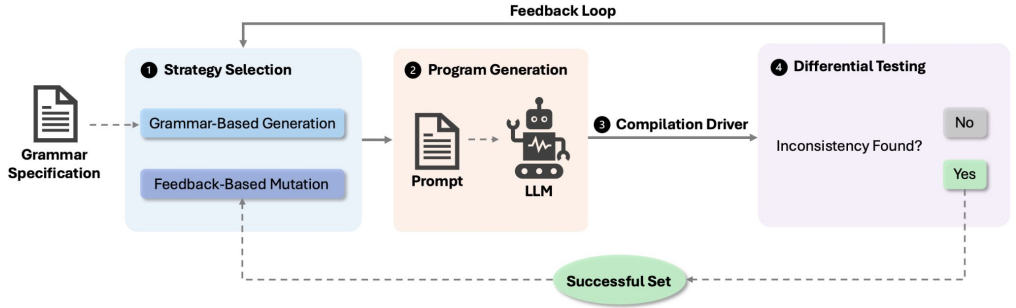
- **Goal**: Generate new and valid floating-point programs from scratch.
- **Prompt design includes**:
  1. **Precision control**: single / double.
  2. **Program structure**: high-level + internal grammar.
  3. **Coding rules**: use only stdio.h, stdlib.h, math.h; avoid undefined behaviors (UB), e.g., initialize all variables.
  4. **Output constraint**: plain C code only, no comments or explanations.

Feedback-Based Program Mutation



- **Goal**: Generate new and diverse FP programs by mutating previously successful examples.
- **Prompt design includes**:
  1. **Precision control**: same
  2. **Program structure**: same
  3. **Coding rules**: same
  4. **Mutation strategies**:
     i. reorder / deeply nest expressions
     ii. change numeric constants
     iii. insert loops / conditionals
     iv. swap math functions (sin, exp, etc.)
  5. **Example input**: a previously successful program
  6. **Output constraint**: same

Feedback Loop



- Once an FP inconsistency is detected, we add the generated program into **successful set**.
- The program example from the successful set is **randomly selected** by the feedback-based mutation strategy.
- A **probability parameter** controls whether the system performs generation-based generation (**0.3**) or feedback-based mutation (**0.7**).

Evaluation Setup

- **Compilers**: gcc-9.4, clang-12.0, and nvcc-12.3
- **Optimization levels**: O0_nofma, O0, O1, O2, O3, O3_fastmath
- **Program generation budget**: 1000 programs
  - 1,000 programs ×3 compilers ×6 op levels = 18,000 program runs
- **Model**: GPT-4 (checkpoint: gpt-4.1-2025-04-14)

RQ1: Effectiveness of LLM4FP vs. Baselines

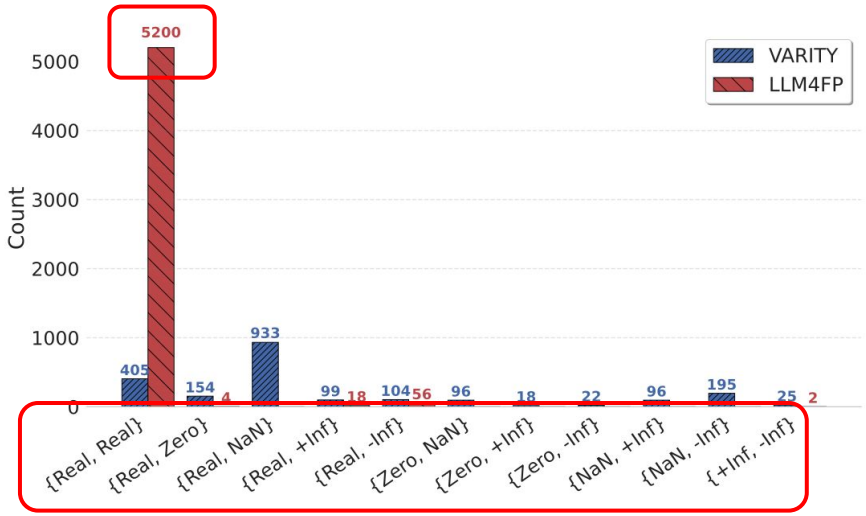
- **Random Testing**: Randomly generates FP programs with grammar specification.
- **Direct-Prompt**: LLM directly generates FP programs from a plain instruction.
- **Grammar-Guided**: LLM is guided by a grammar specification to generate programs.
- **LLM4FP**: Grammar-guided LLM with feedback loop to generate diverse FP programs.

RQ1: Effectiveness of LLM4FP vs. Baselines

Approach	Incons. Rate	# Incons.	Time Cost	CodeBLEU
VARITY	11.93%	2,147	**00:30:42**	0.3581
DIRECT-PROMPT	14.23%	2,562	04:16:37	0.3561
GRAMMAR-GUIDED	16.47%	2,964	05:56:13	0.3442
LLM4FP	**29.33%**	**5,280**	05:37:42	**0.2788**

- LLM4FP achieves the highest inconsistency rate, about 2.5× random testing.
- LLM4FP also shows the lowest CodeBLEU score, indicating highest diversity.
- Higher time cost, but significant gain in inconsistency detection.
- Similar time cost as the other two LLM approaches, but much more inconsistencies.

RQ2: Characterizing FP Inconsistencies



- The majority (98.48%) of inconsistencies of LLM4FP are {Real, Real} differences.
- Extreme outcomes (NaN/±Inf) are rare in LLM4FP.
- Compared to random testing, the focus shifts from crash-like cases to realistic subtle numerical divergence.

RQ3: Inconsistency Rates Across Compilers and Opt. Levels

	Varity			LLM4FP		
Level	gcc, clang	gcc, nvcc	clang, nvcc	gcc, clang	gcc, nvcc	clang, nvcc
O0_nofma	0.03%	0.67%	0.66%	0.48%	1.77%	1.75%
O0	0.03%	0.69%	0.68%	0.48%	2.12%	2.09%
O1	0.03%	0.70%	0.70%	0.46%	2.19%	1.98%
O2	0.03%	0.70%	0.70%	0.39%	2.17%	1.98%
O3	0.03%	0.70%	0.70%	0.38%	2.12%	1.95%
O3_fastmath	**1.02%**	**1.80%**	**2.06%**	**1.55%**	**2.64%**	**2.82%**
Total	1.17%	5.26%	**5.50%**	3.74%	**13.01%**	12.57%

- LLM4FP > random testing under every compiler pair and optimization level.
- O3_fastmath amplifies differences across all compiler pairs.
- Host–Device pairs dominate.
- Demonstrates LLM4FP's ability to uncover broader, realistic cross-compiler divergences.

RQ4: Comparing to O0 without FMA

	VARITY			LLM4FP		
Level	gcc	clang	nvcc	gcc	clang	nvcc
O0	–	–	0.03%	–	–	0.51%
O1	0.02%	0.04%	0.03%	0.36%	0.75%	0.51%
O2	0.02%	0.04%	0.03%	0.41%	0.77%	0.51%
O3	0.01%	0.04%	0.03%	0.38%	0.78%	0.51%
O3_fastmath	**1.61%**	**2.07%**	**0.04%**	**1.76%**	**2.30%**	**0.52%**
Total	1.66%	**2.19%**	0.16%	2.91%	**4.60%**	2.56%

- LLM4FP finds far more intra-compiler inconsistencies.
- Most occur at O3_fastmath, but LLM4FP also finds differences at moderate levels (O1–O3).
- clang dominates for btoh.

Limitations

- **Higher runtime cost**: LLM API latency increases total generation time; improved effectiveness partly offsets this.
- **Possible undefined behaviors**: not all can be avoided, causing potential false positives.
- **Prompt sensitivity**: effectiveness depends on well-crafted prompts and mutation rules.
- **Interpreting inconsistencies**: some are valid compiler differences, not real bugs.

LLM4FP Takeaways

Code and data: https://github.com/ucd-plse/LLM4FP

- Combines grammar guidance and a feedback loop for valid, diverse, and effective FP program generation that expose numerical inconsistencies.
- Achieves 2.5× higher inconsistency rate and greater diversity than existing approaches.
- Reveals subtle, realistic numerical disagreements, not just extreme cases.

A step toward automated, LLM-driven validation of numerical correctness.