Ciel: Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code

Dolores Miao
PhD student
UC Davis

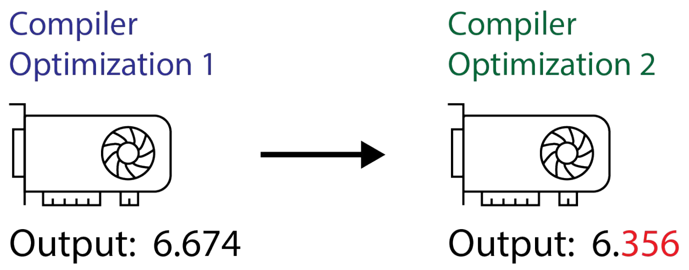
Ignacio Laguna
Computer Scientist
LLNL

Cindy Rubio-González
Associate Professor
UC Davis

UC DAVIS UNIVERSITY OF CALIFORNIA
Lawrence Livermore National Laboratory

Compiler-induced Numerical Inconsistencies

- Inconsistencies caused simply by compiling the same program in different compilers and/or command line options are called **compiler-induced numerical inconsistencies**.
- Compiler-induced numerical consistencies are common in HPC and can significantly impact programming productivity
- Numerical inconsistencies occur when:
  - Codes are compiled with a new version of the compiler
  - Optimizations are used to optimize code for specific platforms
- As HPC codes are ported to new heterogeneous platforms, it is crucial to reduce the amount of time programmers spend debugging such issues



Compiler-induced Numerical Inconsistencies

- Example 1: example from NMSE 3.3.4/FPBench[1]

  pow((x + 1.0), (1.0 / 3.0)) - pow(x, (1.0 / 3.0)); where x = 8291454011552366.0

Command line	Platform	Results
`nvcc -O0`	**CUDA**	**0**
`nvcc –O3 –use_fast_math`	**CUDA**	**0**
`gcc -O0`	**x64**	**2.9103830456733704e-11**
`gcc –O3 –ffast-math`	**x64**	**-5.8207660913467407e-11**

- pow(x, 1/3) having slightly different results when x is very large, resulting in flipped sign

1. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis NSV'16: N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock

Compiler-induced Numerical Inconsistencies

- Motivational example 2: BT.S from NAS Parallel Benchmarks, CUDA version (NPB-GPU[1])
- Result: even smaller inconsistencies can build up, even across compilers.

Command line	Runtime(s)	Maximum Relative Error
`nvcc -O0`	0.104	**6.98176E-13**
`nvcc –O3 –use_fast_math`	0.052	9.73738E-13
`clang -O0`	0.349	8.32928E-13
`clang –O3 –ffast-math`	0.059	**3.50905E-12**

- Real-world Impact
  - CESM: one supported platform failed[2] CESM-ECT verification because of FMA

1. de Araujo, G.A., Griebler, D., Danelutto, M., Fernandes, L.G.: Efficient NAS parallel benchmark kernels with CUDA. In: PDP, pp. 9–16, IEEE (2020)
2. Ahn, D.H., et al.: Keeping science on keel when software moves. Commun. ACM 64(2), 66–74 (2021)

Cause of Compiler-induced Inconsistencies

Compiler-induced Numerical Inconsistencies

- Different compiler implementations
  - Architecture-specific fast approximation
  - Different expression reassociation rules
  - Undefined behavior (e.g., division by zero)
  - etc.

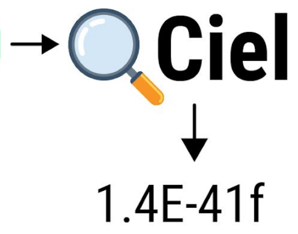
- Aggressive optimization flags
  - Flush to zero/denormals as zero
  - Constant division -> multiplication
  - Fused Multiply-Add
  - Other implied, hidden optimizations

- Just disable optimizations/use higher precision, duh!
  - But it is temporary and slow
- Best if we can find out the cause automatically

Tools on Compiler-induced Inconsistencies

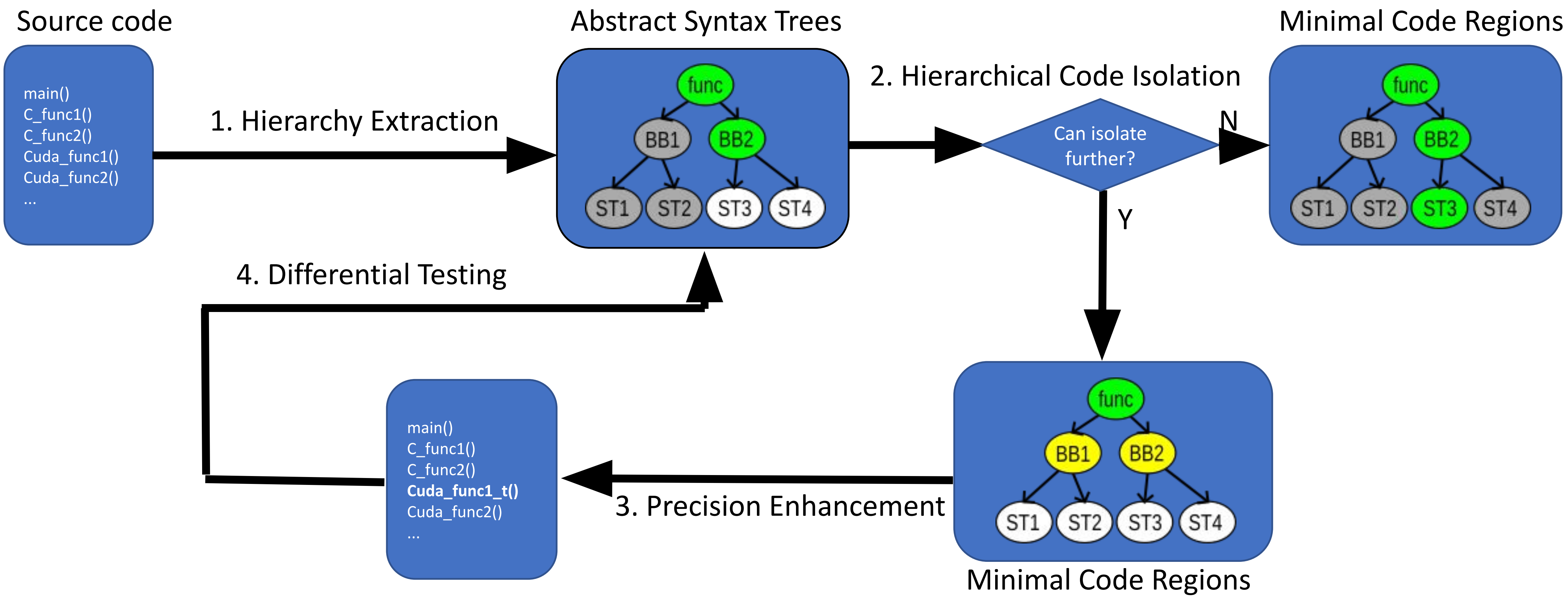
- FLiT[1]: tries out different compiler/optimization flag combinations to find which function in a program has inconsistencies
- pLiner[2]: isolates the source code line(s) that cause inconsistencies in CPU programs
- CIGEN[3]: generating inputs to trigger high inconsistency error
- Ciel: isolates source **expressions** that cause inconsistencies in CPU**/GPU** programs

```
void compute (/* var args */){
  for (int i = 0; i < n; ++i) {
    comp = x - 1.6f;
    float t = 1.4697E36f;
    comp += t + 1.4E-41f;
    if (comp < sinhf(y)) {
      comp = tanf(z);
    }
  }
  printf("%.17g", comp);
}
```

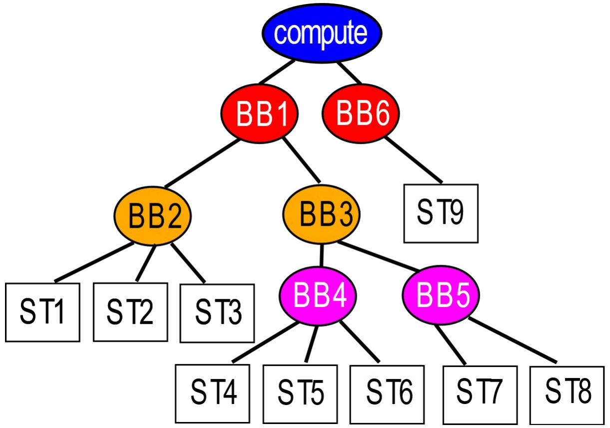


1. Sawaya, et al.: FLiT: Cross-platform floating-point result-consistency tester and workload. In: IISWC 2017, p. 229
2. Guo, et al.: pLiner: isolating lines of floating-point code for compiler-induced variability. In: SC 2020, p. 49
3. Miao, et al.: Input Range Generation for Compiler-Induced Numerical Inconsistencies. In: ICS 2024, p. 201

Ciel (**C**ompiler-induced **I**nconsistency **E**xpression **L**ocator)



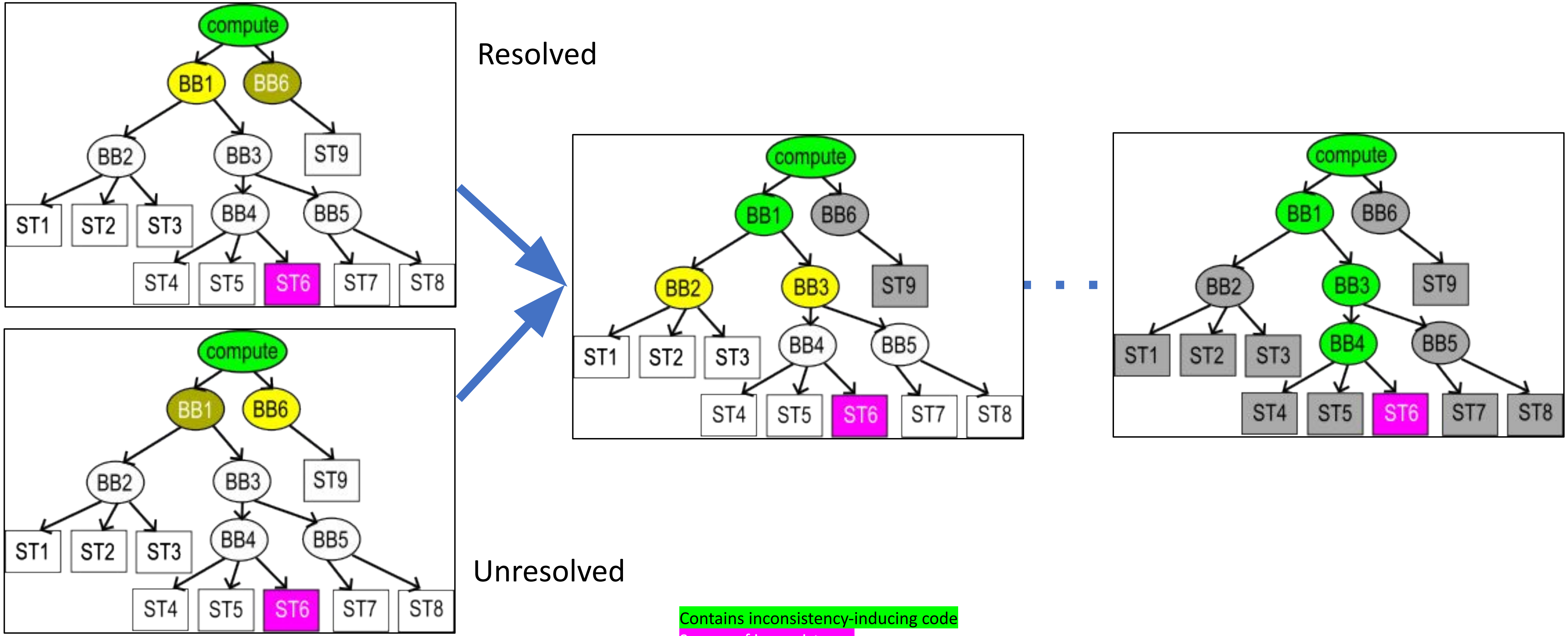
```c
void compute(/*var args*/) {
  for (int i=0; i<n; ++i) {          // ST1-3
    comp = x-1.6f;                    // ST4
    float t = 1.4697E36f;             // ST5
    comp += t+1.4E-41f;               // ST6
    if (comp < sinhf(y)) {            // ST7
      comp = tanf(z);                 // ST8
    }
  }
  printf("%.17g\n", comp);            // ST9
}
```



1. Hierarchy Extraction

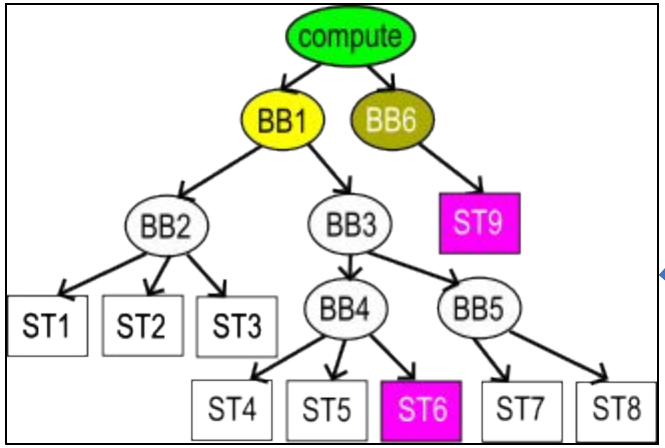
- Simplified AST nodes
  - Statement node
  - Basic block node
    - Normal basic block
    - Conditional basic block
    - Loop basic block
  - Function node
- AST hierarchy
  - Sibling/adjacent nodes

2. Hierarchical Bisection Search - Best Case

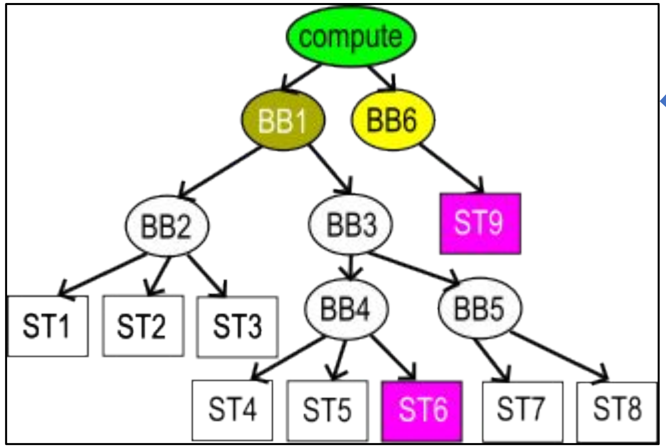


Contains inconsistency-inducing code
Source of inconsistency
Currently under consideration as possible source of inconsistency
Excluded temporarily for bisection search
Discarded from bisection search

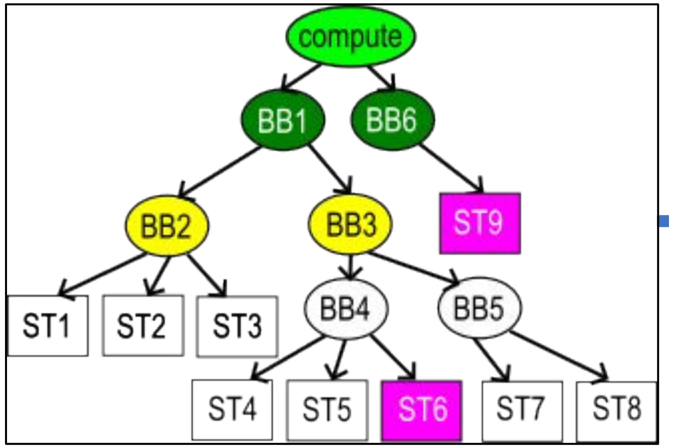
2. Hierarchical Bisection Search - Worst Case

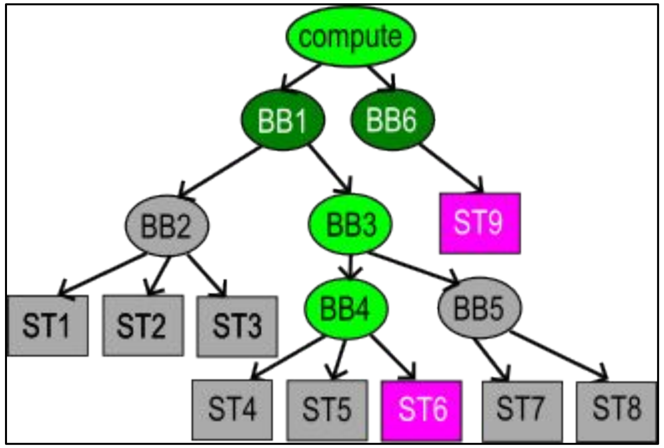


Unresolved



Unresolved





Both branches may contain inconsistency-inducing code
Contains inconsistency-inducing code
Source of inconsistency
Currently under consideration as possible source of inconsistency
Excluded temporarily for bisection search
Discarded from bisection search

3. Precision Enhancement

- Increase floating-point operation precision in code regions
- Infinite precision is best, but the second best is enhanced precision
  - Avoid conditions that cause inconsistencies
  - Minimize rounding error caused by inconsistencies
- Can enhance precision for a single statement/expression

3. Precision Enhancement - Code Transformation

- Notice all math functions are replaced with enhanced precision too
- Perform boundary check, remove redundant type conversions

```
void compute(/*var args*/) {
  for (int i=0; i<n; ++i) {     // ST1-3
    comp = x-1.6f;              // ST4
    float t = 1.4697E36f;       // ST5
    comp += t+1.4E-41f;         // ST6
    if (comp < sinhf(y)) {      // ST7
      comp = tanf(z);           // ST8
    }
  }
  printf("%.17g\n", comp);      // ST9
}
```

```
void compute(/*var args*/) {
  for (int i=0; i<n; ++i) {                 // ST1-3
    comp = x-1.6f;                          // ST4
    double comp_d1 = comp;                  // ST5'
    double t_d1 = 1.4697E36;                // ST5
    comp_d1 += t_d1+1.4E-41;                // ST6
    if (comp < sinh(y)) {                   // ST7
      comp = tan(z);                        // ST8
    }
    comp = comp_d2; t = t_d2;               // ST8'
  }
  printf("%.17g\n", comp);                  // ST9
}
```

3. Precision Enhancement - Expression Transformation

- Expression isolation
  - Useful for long, complex statements
  - All operation in the expression are in enhanced precision
  - The expression itself is cast back to original precision

```
a = b * 2.0f + c    →    a = (float)((double)b * 2.0) + c
```

Adapting to Heterogeneous Code

- Use custom double-double types (GPUprec[1]) for CUDA kernels
  - Use explicit conversion to avoid implicit conversion ambiguity

```
dd var1, var2;
var2 = var1 + 2.0; // (dd)((double)var1 + 2.0)?
                   // var1 + (dd)2.0?
```

```
template <typename Src, typename Dst >
class RefConverter2 {
public:
    FUNCTION_DECL RefConverter2(Src& dv) {
        this->dv = &dv;
        init_vec2(v, dv);
    }
    FUNCTION_DECL RefConverter2(Src* pdv) {
        this->dv = pdv;
        init_vec2(v, (*pdv));
    }
    FUNCTION_DECL ~RefConverter2() {
        init_vec2((*dv), v);
    }
    Dst v;
    Src* dv;
    FUNCTION_DECL Dst& ref() { return v; }
    FUNCTION_DECL Dst* ptr() { return &v; }
};
```

- Anonymous converters for derived data types

```
float2 f2;
...
func(f2);
```

→

```
float2 f2;
...
double2 f2_d = Convert2(f2);
...
func(RefConverter2(f2_d).ref());
...
```

1. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: DaMoN, pp. 19–26, ACM (2010)

Ciel Mini-Demo

- Try it on your x64 computer if you have Docker installed
  - Demo is run in WSL2 with Ubuntu 22.04
  - Sample program is CPU only, so no need to have an NVIDIA GPU
- Code repository: https://github.com/LLNL/Ciel
- Run docker container from code repo

```
# docker pull ucdavisplse/ciel:latest
# docker run -it -v [ciel directory]:/root/ciel/ --name ciel ucdavisplse/ciel:latest
```

- Setup runtime environment

```
# cd /root/ciel
# source setup.sh
# source driver/setup_cc.sh
# ./build_single_plugin.sh
```

Ciel Mini-Demo

- Sample program in `samples/mini_sample` directory
- Sample outputs under different compiler optimizations

```
# make OPT_LEVEL=0 FASTMATH=0
# make run
>>> 1.9024985824408881e-318
# make OPT_LEVEL=3 FASTMATH=1
# make run
>>> -1.1051e-186
```

- Run Ciel to isolate the expression that cause this inconsistency

```
# python3 ~/Ciel/driver/run_test.py
...
>>> offending subexpression: DeclRefExpr, exp
>>> offending text: ./test.cpp:12:13, ./test.cpp:12:13
```

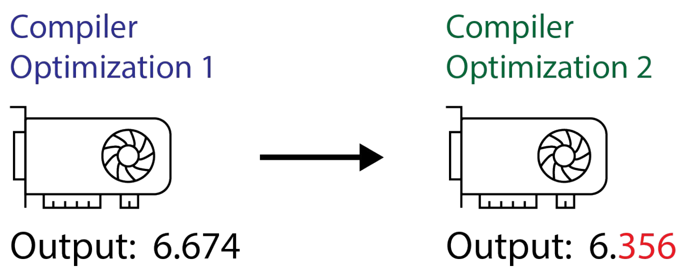
- Source of inconsistency: `comp <= `**`exp`**`(log10(var_1 / var_2)) var_1 = 0.0`

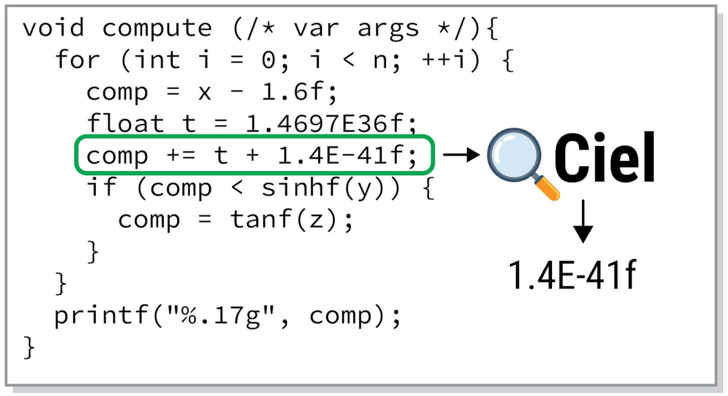
Next session

- How to setup Ciel for your program
- A look into inconsistencies and how they may occur

Ciel – Brief Review

- Heterogeneous programs may exhibit numerical inconsistencies caused by compiler differences and aggressive compiler optimizations
- Our tool, Ciel, isolates these inconsistencies with floating-point precision enhancement and a novel recursive bisection search algorithm
- Next demo: setting up Ciel for your programs





Ciel demo - Setup

- Code repository: https://github.com/LLNL/Ciel
- Run docker container from code repo, with GPU passthrough (NVIDIA GPU sm_61 or later required)

```
# docker pull ucdavisplse/ciel:latest
# docker run –gpus all -it -v [ciel directory]:/root/ciel/ --name ciel ucdavisplse/ciel:latest
```

- Setup runtime environment

```
# cd /root/ciel
# source setup.sh
# source driver/setup_cc.sh
# ./build_single_plugin.sh
```

Ciel demo – 1. Rodinia CFD

- Sample program: `experiments_nas_rodinia/rodinia_cfd_097K`
- Setup threshold to verify results
  - `experiments_nas_rodinia/rodinia_cfd_097K/euler3d.cu` line 186 - 195
- Run Ciel to isolate the expression that cause this inconsistency

```
# python3 ~/Ciel/driver/run_test.py
...
>>> offending subexpression: DeclRefExpr, sqrtf
>>> offending text: ./euler3d.cu:283:54, ./euler3d.cu:283:54
>>> offending subexpression: DeclRefExpr, speed_sqd
>>> offending text: ./euler3d.cu:283:60, ./euler3d.cu:283:60
```

- Log files
  - workspace/config_*: transformed source code, etc.
  - *.out: log files for Clang plugins, outputs for building and running programs

Ciel demo - How to Setup Ciel for Your Program

- Makefile for Ciel
  - Ciel always calls Makefile
  - `cc` Macro: compiler parameter
  - `make`: Default compilation
    - `OPT_LEVEL` macro: 0, 1, 2, 3; corresponds to options from –O0 to –O3
    - `FASTMATH` macro: 0, 1; corresponds with –ffast-math or equivalent options
    - For example, for CUDA, `make OPT_LEVEL=3 FASTMATH=1` => `–O3 –use_fast_math`
  - `make run`: Run the program with designated arguments
  - `make clean`: Cleanup
  - Clang plugins related:
    - `make mk_workspace:` create workspace directories for Ciel
    - `make extract_hierarchy:` hierarchy extraction
    - `make enhance_precision:` precision enhancement
    - `make print_results:` print results

Ciel demo - How to Setup Ciel for Your Program

- Customizing Ciel behavior with setup.ini
  - `UseExtendedPrecision`: enable/disable extended precision library GPUPrec
  - `BlackList`: functions not included in search
  - `SubExpressionIsolation`: enable/disable isolating sub-expressions inside an expression ended with semicolon
  - `CompilerList`: a list of compilers to be tested, separated by colons
- Customizing Ciel itself (driver/search.py)
  - `class Compiler(Enum):` an enumeration of available compilers
  - `class OptLevel(Enum):` an enumeration of possible macro combinations

Ciel demo - – 2. CLOUDSC

- Non-Makefile sample program: `experiments_ecmwf/`
- This project use a combination of bash scripts + CMake
  - Create a separate Makefile for Ciel to call
  - In CMake files, append different optimization flags according to environment variables

```
if(DEFINED ENV{FASTMATH})
    if($ENV{FASTMATH} STREQUAL "1")
        message(STATUS "fast math enabled")
        set(ECBUILD_CXX_FLAGS
"${ECBUILD_CXX_FLAGS} -ffast-math")
    endif()
else()
    message(STATUS "fast math disabled")
endif()
```

Isolated Inconsistency-Inducing Code from Ciel

- Sample 1: Rodinia CFD
  - input file: fvcorr.domn.097K has inconsistency
  - We used Ciel to isolate an expression in cuda_compute_step_factor()
    - `step_factors[i] = float(0.5f) / (sqrtf(areas[i]) * (sqrtf(speed_sqd) + speed_of_sound));`
- Sample 2: CLOUDSC
  - ECMWF cloud microphysics parameterization mini-app
  - We used the tool to isolate leftover debug code that cause inconsistency when compiled with –O3 –ffast-math:
    - `zfallcorr = pow(yrecldp->rdensref/zrho[jl-1], (float)0.4);`

Thank you!

Ciel is still in development.

Code repo: https://github.com/LLNL/Ciel

Contact: Dolores Miao (wjmiao@ucdavis.edu)

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PRES-849103), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under awards DE-SC0022182 and DE-SC0020286, and the National Science Foundation under award CCF-1750983.