Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code

Dolores Miao[1(✉)], Ignacio Laguna[2], Cindy Rubio-González[1]

[1] University of California, Davis, Davis, CA 95616, USA
{wjmiao,crubio}@ucdavis.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
ilaguna@llnl.gov

Abstract. As the demand for developing and porting numerical applications to heterogeneous computing platforms increases, such programs may exhibit *numerical inconsistencies* caused by architectural differences and aggressive compiler optimizations. These numerical inconsistencies can negatively impact reproducibility and debugging. This paper presents Ciel, designed to identify the root cause of compiler-induced numerical inconsistencies in heterogeneous programs. Ciel uses a floating-point precision enhancement strategy, guided by a recursive bisection search algorithm with increasing search granularity, to identify the *program expressions* that induce numerical inconsistencies due to compiler optimizations. Ciel achieves 99.4% precision in isolating numerical inconsistencies in both CPU and GPU programs, including 330 synthetic GPU programs, benchmark applications like NAS Parallel Benchmarks and Rodinia, and real-world scientific applications such as CLOUDSC, a cloud microphysics parameterization mini-app for the ECMWF IFS. Furthermore, when compared with the state of the art, which only isolates *lines* of code in CPU programs, Ciel runs 24.5% fewer searches for statement isolation, and produces more precise results for 84.9% of the programs. Finally, manual inspection of hundreds of compiler-induced numerical inconsistencies in heterogeneous programs reveals common characteristics.

1 Introduction

Heterogeneous computing uses different processing cores, such as CPUs and graphics processing units (GPUs), to run programs with maximized performance [8]. Software engineers from various fields use GPUs to form heterogeneous architectures and accelerate large-scale parallel computations. General-purpose computing on GPUs (GPGPUs) has become the go-to choice for physics simulations, digital signal processing, machine learning, and climate research.

Compiler optimizations are often the first method software engineers consider when optimizing programs. Aggressive optimization options are also often invoked to push program performance as much as possible. Additionally, switching or upgrading compilers in the middle of a project is also a frequent industrial practice. Unfortunately, such modifications on a project global scale can have a negative impact on software reliability, particularly on floating-point arithmetic which could result in local errors that may propagate to the final program's

output. In cases found in the literature [5, 18, 21], it has required significant effort and domain knowledge to isolate and fix these issues.

Numerical Reproducibility Challenges. Given the large number of hardware architectures, compilers and host environments involved in executing heterogeneous programs, maintaining numerical consistency and reproducibility is equally important to their pure CPU counterparts. Most hardware devices and compilers follow the IEEE 754-2008 standard [1], but offer optimization options, such as `-ffast-math` in Clang, that further push computational performance at the cost of strict IEEE 754-2008 compliance. Such non-compliant optimizations can yield different computation results—*numerical inconsistencies* —between CPU- and GPU-computed results, or for CPU- or GPU-only computations optimized at different levels. These inconsistencies often result in numerical correctness bugs, some of which are reported in widely adopted numerical libraries [15]. Many applications, when ported to GPU platforms, struggle to find a balance between performance speedup and avoiding *compiler-induced numerical variability* impacting the precision of the results [21]. Such impact has already been acknowledged by the floating-point research community concerned with ensuring numerical accuracy on heterogeneous computing systems [17].

Simply disabling compiler optimizations, or increasing precision uniformly across an application, may solve compiler-induced variability, but they are not practical solutions. Instead, developers strive to find the root cause of these compiler-induced inconsistencies and manually fix them to reduce their impact without disabling compiler optimizations. Currently, identifying the root cause of such issues in heterogeneous programs is a manual effort, requires domain knowledge, and is a time-consuming task.

Main Contributions. We present CIEL (which stands for Compiler-induced Inconsistency Expression Locator), the first tool that automatically isolates numerical inconsistencies in heterogeneous programs at the expression level. Prior work [18, 28] has proposed automated approaches to isolate such inconsistencies in pure CPU programs. FLiT [28] works at the function level, while pLiner [18] isolates lines of code that cause inconsistencies, but neither targets GPU code nor isolates at expression level, which further reduces developer workload.

In numerical program error analysis, replacing floating-point operations with higher precision variants is widely employed [7, 16, 30] to more accurately approximate the results of operations in infinite precision. Higher precision operations have a smaller ulp (unit in last place) error, and exceptions such as subnormal numbers and infinity are much less likely to be triggered. It is shown in [18] that compiler-induced inconsistencies can be minimized by enhancing precision. CIEL operates on the same assumption that compilers will produce enhanced precision binary instructions when specific source code regions are in enhanced precision.

Compared to the state-of-the-art [18] for CPUs where each code block at each level is treated individually, our approach traverses the abstract syntax tree (AST) of each function and performs a bisection search for all *adjacent* sibling code blocks, maintaining the adjacency relationship between them; during precision enhancement, adjacent code blocks are either combined into a single

code region or have variable checkpoints where redundant type conversions are removed. Furthermore, Ciel isolates code down to the expression level rather than the statement level (line level in [18]). Since the program statements causing the compiler-induced inconsistencies may include many floating-point operations involving different operators, variables, constants, or function calls, isolating at the expression level provides a more precise insight into the inconsistencies, pointing users directly to their root cause and potential fix.

In particular, to adapt to features and limitations on GPU platforms, such as the lack of floating-point arithmetic beyond double precision or the built-in vector arithmetic, Ciel supports extended precision libraries, and can transform built-in vector arithmetic to enhanced precision. Ciel detects code written for different target platforms (CPU or GPU code) and automatically transforms them according to platform specifications, e.g., platform-specific language constructs and data types. Ciel provides a solid foundation for extending support to other platforms, such as OpenMP or OpenCL [3], as long as they are supported by Clang. *To the best of our knowledge,* Ciel *is the only tool capable of isolating code regions in heterogeneous computing programs that, combined with compiler optimizations, produce inconsistent numerical results.*

We evaluate Ciel on a set of heterogeneous programs, including 330 synthetic GPU programs, and on GPU programs from the NAS [6] and Rodinia [10] benchmarks. Ciel achieves a precision of 99.4% in isolating compiler-induced inconsistencies in these programs. Moreover, Ciel finds the root cause of a real-world compiler-induced inconsistency in the C version of ECMWF Cloud Physics mini-app CLOUDSC [14] in only 7 minutes. The root cause of the inconsistency has been confirmed by ECMWF domain experts. Finally, compared to pLiner [18], the state of the art in isolating *lines* of code in CPU programs, Ciel performs 24.5% fewer searches for statement isolation, and produces more precise isolation results for 84.9% of the pLiner's CPU benchmarks.

In summary, the contributions of this paper are as follows:

- An approach for isolating minimal code regions that cause compiler-induced numerical inconsistencies in heterogeneous programs. Our approach uses a more efficient bisection search compared to the state of the art for CPU programs that operates on simplified ASTs and provides a finer search granularity at the expression level (Sections 3.1 and 3.2).
- A precision enhancement strategy that more accurately reflects the resolvability of inconsistencies under precision enhancement, and addresses challenges specific to transforming heterogeneous code to higher precision (Section 3.3).
- An implementation of our approach in the tool Ciel, and an evaluation that shows (1) efficacy at isolating inconsistencies in a large and diverse set of heterogeneous programs: 330 synthetic GPU programs, NAS and Rodinia GPU benchmarks, and the real-world mini-app CLOUDSC (Section 4.1), and (2) higher precision and efficiency in comparison with the state of the art in isolating numerical inconsistencies in CPU programs (Section 4.2).
- A manual inspection of the isolated code that causes compiler-induced inconsistencies, which reveals common characteristics (Section 4.1).

2 Examples of Compiler-Induced Inconsistencies

Table 1: Inconsistencies in BT.S.

Compiler Options	Runtime	Error
nvcc -O0	0.104s	6.98176E-13
nvcc -O3 -use_fast_math	0.052s	9.73738E-13
clang -O0	0.349s	8.32928E-13
clang -O3 -ffast-math	0.059s	3.50905E-12

Compilers for CPU and GPU code, such as Clang [4] and nvcc [2], offer various levels of optimization flags from `-O0` to `-O3`. With higher optimization levels, program performance is improved, sometimes significantly, but at the cost of potentially generating non-compliant IEEE 754-2008 floating-point code. There are optimization flags that explicitly violate the IEEE 754-2008 standard, but in cases where precision is of less concern, they offer good speedups. For example, consider the CUDA version of the BT NAS program with input class S. Table 1 shows the program runtime and the maximum relative error for each compiler and optimization flag combination. Using `-O3 -use_fast_math` with nvcc yields 100% speedup compared to `-O0`, but at the cost of the error being 39% larger. Performance and error with Clang is generally worse, with the largest error in `clang -O3 -ffast-math` being 403% larger than `nvcc -O0`.

In real-world applications, such compiler-induced numerical inconsistencies occur frequently. They can happen when migrating software to other hardware/-software platforms, switching applications to a new compiler, or just using more aggressive optimization flags for compilation. These inconsistencies may cause major software failures that take tremendous amount of effort to identify and resolve. A documented case [18, 21] in the Laghos (LAGrangian High-Order Solver) application [9] is observed when ported to the Lawrence Livermore National Laboratory's Sierra system using the IBM xlc compiler. This triggered an inconsistency in the energy computed by the application under `xlc -O3` but not with `xlc -O2`. In another documented case in [5], the Community Earth System Model (CESM) failed its verification using the CESM-ECT quality assurance framework when it was ported to the Mira machine at Argonne National Laboratory. Both took from weeks to months for scientists and engineers to identify the source code that caused such failures.

Issues like the above are bound to occur when real-world scientific applications are written or ported to new platforms. *Automatically resolving such issues without extensive domain knowledge would save a massive amount of time and increase programming productivity.*

3 Technical Approach

Problem Statement. Given *heterogeneous programs* with known compiler-induced numerical inconsistencies, a practical problem for software developers is how to isolate the *expressions* that cause such inconsistencies in a precise and efficient manner. CIEL is designed with the goal of tackling this problem. Specifically, CIEL takes as input a program P and its associated input, which under compilers $C_1, C_2, ..., C_n$ and optimization flags $O_{i1}, O_{i2}, ..., O_{ik}$ for each

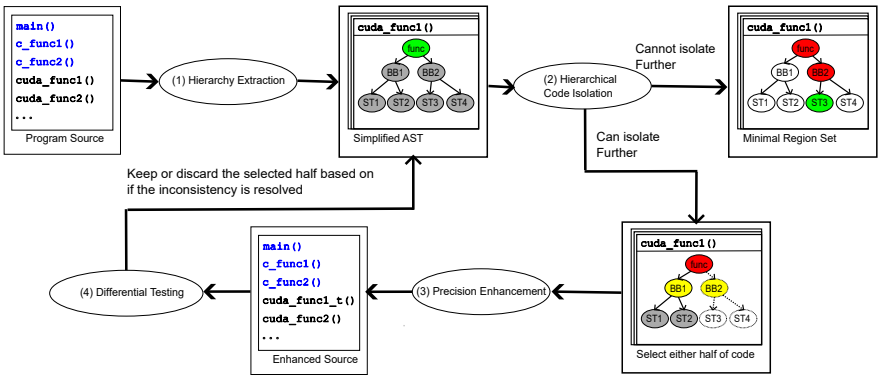


Fig. 1: The workflow of CIEL.

compiler C_i, produces inconsistent results. CIEL outputs the *minimal region* R_m in m searches, which means that by generating program variants $P'_1, P'_2, ..., P'_m$ it isolates the root cause of the inconsistency to a code region as narrow as possible. Below we present definitions that will be used throughout the rest of the paper.

Definition 1. *The output of program P given a specific error threshold ϵ under compiler C_i and optimization flags O_{ij} is written as $f(P, \epsilon, C_i, O_{ij})$.*

Definition 2. *Compiler-induced inconsistencies occur if there are two sets of compiler/optimization flag combinations C_i, O_{ij} and C_k, O_{kl}, where $f(P, \epsilon, C_i, O_{ij}) \neq f(P, \epsilon, C_k, O_{kl})$. When any two sets of combinations have the same output, the inconsistencies are considered to be resolved.*

Definition 3. *A region set R of a program P is defined as a set of regions in P, each of which is a straight-line code fragment with an entry point and one or more exit points.*

Definition 4. *A region set R_m is minimal if (a) the inconsistency is resolved when code in R_m is executed in higher precision, and (b) either R_m consists of only one expression, or leaving any expression in R_m in lower precision would result in unresolved inconsistencies.*

CIEL's Workflow. The overall workflow of CIEL is illustrated in Figure 1. To find the minimal region that causes the numerical inconsistency, CIEL performs *hierarchical bisection search* on the source code—first between functions, then between code regions in the suspected functions. Each iteration increases the search granularity. The search algorithm identifies regions suspected of causing compiler-induced inconsistencies. For each region $R_1, R_2, ..., R_m$, CIEL then creates a mutated variant of the program $P'_1, P'_2, ..., P'_m$ for which code in the corresponding region is in enhanced precision. Whether the variant resolves these inconsistencies is then used to guide the further, narrower isolation of source code

that triggers inconsistencies. The isolation process ends when region R_m satisfies the conditions of a minimal region. The modules in CIEL are described below:

(1) *Hierarchy Extraction* traverses the AST of the functions under analysis, extracting information relevant to floating-point operations, and generating a simplified AST for these functions. This is the entry point of the analysis.

(2) *Hierarchical Code Isolation* performs a hierarchical bisection search on the simplified AST to generate regions for subsequent precision enhancement.

(3) *Precision Enhancement* increases the precision of the code regions identified by hierarchical code isolation. The output is the transformed source code with the floating-point operations in specific code regions written in higher precision.

(4) *Differential Testing* compiles and runs the transformed program with specified compilers and optimization flags in parallel. The output of these combinations of compilers and flags is compared to determine if the compiler-induced inconsistencies are resolved.

The rest of the section describes modules 1-3 in more detail.

3.1 Hierarchy Extraction

The hierarchy extraction module traverses the program AST and extracts source code hierarchy information for each function in the form of a *simplified AST*. The simplified AST acts as a data exchange format between modules, and contains additional data specific to CIEL that includes whether a node should be enhanced in precision (enabled/disabled), and the list of all floating-point operations such as reads, writes, declarations, function calls, and constants in every statement under a node. The simplified AST classifies statement structure of a function into five node categories:

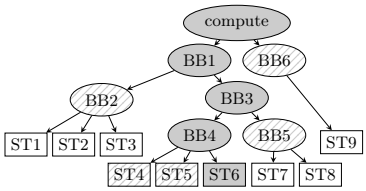
1. Each statement that ends with a semicolon (declaration, expression, and *return*/*break*) is a **statement node** on the simplified AST. Each statement node also contains its expression AST hierarchy.
2. A set of statements with only one entry point and one exit point is grouped as a **basic block (BB) node**.
3. For a selection statement such as *if-else* or *switch-case* statement, one BB node is assigned to each branch; and then a **conditional block node** is assigned for the whole selection statement as a code block.
4. For a loop statement such as a *for* or *do-while* statement, one BB node is assigned to the condition, another to the loop body, and then a **loop block node** is assigned for the whole loop statement as a code block.
5. A **function node** is assigned to the whole function.

The relationship between a sample program and its simplified AST representation is shown in Figure 2. Each expression statement (for-loop header in Line 2; if condition in Line 6; statements in Lines 3, 4, 5, 7, 10) in Subfigure 2a has its own statement node, which is organized into block nodes. The corresponding simplified AST is shown in Subfigure 2b.[3]

[3] Statements and blocks with no floating-point operations are recorded but excluded from precision enhancement.

```
1 void compute(/*var args*/){
2   for(int i=0; i<n; ++i) { //ST1-3
3     comp = x-1.6f; //ST4
4     float t = +1.4697E36f; //ST5
5     comp += t+1.4E-41f; //ST6
6     if (comp < sinhf(y)) { //ST7
7       comp = tanf(z); //ST8
8     }
9   }
10  printf("%.17g",comp); //ST9
11 }
```

(a) Sample function `compute`. Variables `comp`, `x`, `y` and `z` are function arguments of type `float`.



(b) Simplified AST of `compute`. The inconsistency is in ST6. Dark filled nodes are hierarchically isolated. Stripe filled nodes are considered in the bisection search at each level of hierarchy, but not isolated. White nodes are not considered during bisection search.

Fig. 2: Sample function and its simplified AST.

3.2 Hierarchical Code Isolation

When hierarchy extraction is complete, the simplified ASTs for all functions are output to the hierarchical code isolation module. As code isolation progresses, it marks nodes on the simplified ASTs as enabled or disabled depending on whether the node is still in consideration as a potential cause of inconsistencies.

Bisection search is the basis for the approach, followed by a 1-minimal check [31]. Bisection has shown to be an effective search strategy in the context of code isolation in CPU programs [18, 28]. CIEL bases its search algorithm on the same idea of partitioning the program into functions, code blocks, and statements, but improves on how the hierarchical search is performed to reduce search time and improve precision. In particular, CIEL proposes a refined hierarchical region isolation with the explicit goal of improving isolation accuracy by reducing unnecessary type conversions when enhancing precision. Furthermore, unlike previous work, CIEL explores expression-level granularity during the search.

CIEL isolates a minimal region of code amongst a set of code regions by recursively bisecting suspicious code regions into two halves and verifying if enhancing either half resolves the inconsistency (Line 1 in Algorithm 1). Hierarchical search first sets the whole program in enhanced precision (Line 17 in Algorithm 1), then finds the minimal region in increasing granularity, following two stages:

1. **Function Isolation.** During the function isolation stage (Line 16 in Algorithm 1), bisection search is performed at the function level, and the result is a minimal set of functions that cause the inconsistencies.
2. **Hierarchical Region Isolation.** For each function isolated in the first stage, during the hierarchical region isolation (Line 11 in Algorithm 1), bisection search is performed at increasingly granular levels, from code block level to statement level, and ultimately to expression level.

CIEL traverses the simplified AST of each function isolated in the function isolation stage, and isolates child nodes of the current node(s) that cause the compiler-induced inconsistency: from the child nodes of the function node, to

Algorithm 1: Hierarchical Code Isolation.

```
1  Function BisectionSearch(regions) :
2      if regions.size() > 1 then
3          regions_1, regions_2 = ArraySplit(regions, 2);
4          if HasResolvedInHighPrecision(regions_1) then
5              BisectionSearch(regions_1);
6          else if HasResolvedInHighPrecision(regions_2) then
7              BisectionSearch(regions_2);
8          else
9              BisectionSearch(regions_1);
10             BisectionSearch(regions_2);
11 Function RegionIsolation(regions) :
12     BisectionSearch(regions);
13     foreach region in regions do
14         if region.hasSubBlocks() && region.inHighPrecision() then
15             RegionIsolation(region.getSubBlocks());
16 Function FuncIsolation(Funcs ):
17     Funcs.setHighPrecision();
18     BisectionSearch(Funcs);
19     minFuncs = Funcs.getHighPrecisionFuncs();
20     foreach func in minFuncs do
21         RegionIsolation(func.getBlocks());
```

child nodes of BB nodes, to all isolated statement nodes, until within the smallest subexpression in a statement node, e.g., a variable, constant, or function call. A difference of this code isolation method, compared to prior work, is that these child nodes are *continuous* blocks or statements, which are split into two continuous sets of code blocks or statements. For example, n continuous code blocks are split into the first $\lfloor n/2 \rfloor$ blocks and the remaining $n - \lfloor n/2 \rfloor$ blocks. Combined with the *region merge pass* (Section 3.3), all blocks are merged into as few continuous code regions as possible, reducing redundant type conversions. Section 4.2 shows that by removing redundant type conversions, the transformed program can more accurately and efficiently reflect the resolvability of compiler-induced numerical inconsistencies under precision enhancement.

Furthermore, given how statements in loops could accumulate errors that could exacerbate compiler-induced inconsistencies, our bisection search prioritizes loop structures. Thus, loop BBs at the current level of the AST hierarchy are isolated first. If inconsistencies are resolved then the search is narrowed down to the identified loop BBs; otherwise the search proceeds normally.

We use the sample function from Subfigure 2a to illustrate hierarchical region isolation within a function. The statement that causes the compiler-induced inconsistency is in Line 5 (`ST6` in Subfigure 2b). The algorithm first searches in the loop BBs at the top level, between `BB1` and `BB6`. The inconsistency is resolved with `BB1` in enhanced precision, thus `BB1` is isolated and further split into `BB2` and `BB3`. `BB3` is isolated next, which is then split into `BB4` and `BB5`, with `BB4` then isolated and split into statements `ST4`, `ST5` and `ST6`, from which the constant expression $1.4E-41f$ in `ST6` is found to be the root cause of the inconsistency.

3.3 Source-to-Source Precision Enhancement

The precision enhancement module takes as input the marked simplified AST from the hierarchical code isolation module, and produces a transformed program where all floating-point operations in an enabled code region, whether it is a whole function or a continuous code segment, are in enhanced precision. Source-to-source program transformation allows the resulting programs to be successfully compiled by the same compilers that trigger the original inconsistencies. Furthermore, a source-level transformation, in contrast to IR or assembly level, is not affected by aggressive optimization passes such as instruction reordering which would obscure and obfuscate the boundaries between source code statements during binary generation.

Ciel detects and classifies CUDA host and device functions according to language-specific modifiers in the AST, such as the `__global__` and `__device__` modifiers in the function signature, and transforms code accordingly.

In terms of enhancing precision for CUDA kernels, while CPU programming platforms generally natively support floating-point types beyond double precision, GPU platforms do not. Thus we design Ciel to support precision enhancement with custom extended precision floating-point types that support operator overloading and math functions. Some examples of extended precision libraries include CAMPARY [19] and CUMP [25], but only GPUprec [24] fits the above criteria for integration. GPUprec only requires modest effort to be integrated with Ciel. We use its quadruple precision type to perform precision enhancement because it offers the most support for math functions.

Ideally, all code executed is available to Ciel when isolating code within a code region. However, external functions whose code is not available may be called within a code region. Even though it would not be possible to isolate individual expressions within such external functions, isolating the function call site itself may still be helpful in isolating numerical inconsistencies. In cases where inconsistencies exist in external functions, and an enhanced precision version of the same function is available, replacing the original function calls with calls to their corresponding enhanced-precision functions is expected to resolve the inconsistencies. Thus, for functions called within enhanced code regions, Ciel automatically replaces those given in a customizable *replacement function list*, most of which are math library functions, with an enhanced precision version. In the example in Subfigure 2a, `sinhf` and `tanf` would be replaced with `sinh` and `tan`, respectively. On the other hand, precision enhancement of variables and constants consists of two stages: region and expression transformation, with targeted strategies for different categories of variables.

Stage 1: Region Transformation. This stage enhances the precision of floating-point operations in a specific code region including scalar variables, built-in vectors and constants. Region transformation consists of three passes: region merge, variable categorization, and code transformation.

Pass 1: Region Merge. This pass merges all basic blocks and statements to be enhanced into as few continuous code regions as possible. Compared to prior work, this pass is added specifically as an improvement in removing unnecessary type conversions in precision enhanced code. If two adjacent code blocks on the same level of the AST hierarchy are to be enhanced, then these blocks are merged into one single block. For example, `ST5` and `ST6` in Subfigure 2b are adjacent and on the same level in the AST, thus they are merged into one block. If two adjacent code blocks that are on different levels of the AST hierarchy are to be enhanced, we insert a *variable checkpoint* between them so that redundant type conversions can be detected and removed during the Code Transformation pass. For example, `ST6` and `BB5` in Subfigure 2b are adjacent but on different levels in the AST, a variable checkpoint is inserted here so that there would be no redundant type conversions in between for variables such as `comp`.

Pass 2: Variable Categorization. This pass iterates through all variable uses in a code region, and categorizes scalar and built-in vector floating-point variables[4] into four groups. Our variable categorization algorithm is based on [18] which, in essence, separates *read-after-write* variables in a code region that require allocating temporary storage from variables that just require casting when referenced. We then categorize the *read-after-write* variables into two groups based on whether the variable declaration is inside (*reviseVars*) or outside (*replaceVars*) the code region since they require different transformation strategies. Finally we group the variables that only require casting when referenced by checking if they are only read (*rdVars*) or only written (*wrVars*). The last category (*wrVars*) was added in Ciel to implement precision enhancement in GPU programs with custom extended precision floating-point types described later in this subsection.

Pass 3: Code Transformation. This pass transforms variables according to their categorization:

T1 For *reviseVars*, the declarations of these variables (originally inside the region) are replaced with a temporary variable in higher precision; any reference to this variable inside the code region is replaced with its corresponding temporary variable; the declaration of the original variable is moved prior to the region's exit points, and initialized with the temporary variable.
T2 For *replaceVars*, a temporary variable declaration is inserted at the entry of the region, initialized with the value of the original variable (declared outside the region); any reference to this variable inside the code region is replaced with the temporary variable; the value of the temporary variable is assigned back to the original variable at region exit points.
T3 For *rdVars*, any reference to the variable inside the code region is explicitly upcast to higher precision.
T4 For *wrVars*, any assignment to the variable inside the code region is explicitly downcast to lower precision (reads may occur only outside the region).

[4] Pointers and array references are not categorized; their dereferences are directly cast.

Lastly, type conversion statements from/back to original precision are inserted at the entry/exit point(s) of a code region. Note that calls to functions that are not included in the replacement function list are treated as special exit points of the code region, and their arguments are cast to original precision prior to the function call. Additionally, if the exit/entry of a code region is a *variable checkpoint*, CIEL finds all the variables shared between the two code regions, and simply assigns the enhanced-precision replacement variable in the first region to the one in the second region. By doing so, CIEL prevents redundant type conversions between these two code regions.

Custom extended-precision floating-point types present unique challenges compared to built-in floating-point types. C++ allows implicit conversions among floating-point types, even when such conversions incur precision loss, such as from `double` to `float`. However, such implicit conversions are not possible for custom floating-point types. For example (assuming the type name is `dd_real`):

```
dd_real a = 1.0; dd_real b = a + 2.0;
```

There is ambiguity in `a + 2.0`, which can either be interpreted as an addition of two `dd_real` values or two `double` values before assigning the result to `b`. For such code to pass compilation, CIEL inserts explicit casts back to original precision in value assignments, function arguments, and other possible situations. These explicit casts require CIEL to categorize variables that are only written in a code region, hence a new category, *wrVars*, was added.

Another challenge when enhancing the precision of CUDA kernels is built-in floating-point vector classes. These classes provide vertex and matrix calculation in 2 to 4 dimensions and are widely used. CIEL supports transforming built-in vector type operations to enhanced precision, including type conversions for function arguments passed by reference or by dereferencing. This requires creating temporary variables that are live only during the function call. For this purpose, we implemented converter template class instances as anonymous variables in function arguments. Upon construction, they accept a reference or a pointer of the source variable, convert it to the target type, and provide a reference or a pointer in the target type to the function calls. When the function call is finished, destructors for these converter classes are invoked, and we assign the return value of these references/pointers back to the original variable.

Stage 2: Expression Transformation. This transformation is only applied when the code has been successfully isolated at the statement level. Specified subexpressions in each isolated statement are converted to enhanced precision. CIEL traverses the AST starting from the subexpression node, cast all variable reads and constants from subexpressions to enhanced precision, and the whole subexpression is explicitly converted back to original precision. For example, the subexpression `b*2.0f` in expression `a = b*2.0f+c` would be transformed to `(float)((double)b*2.0)` in enhanced precision.

4 Experimental Evaluation

This experimental evaluation answers the following research questions:

RQ1 How effective is Ciel at isolating compiler-induced numerical inconsistencies in heterogeneous programs?

RQ2 How does Ciel compare with the state of the art in isolating compiler-induced numerical inconsistencies in CPU programs?

4.1 RQ1: Numerical Inconsistencies in Heterogeneous Programs

Benchmarks. We collected a total of 339 compiler-induced inconsistencies: 330 inconsistencies observed in floating-point synthetic GPU programs, 5 inconsistencies triggered in NAS Parallel Benchmarks for GPU (NPB-GPU) [6], 3 inconsistencies triggered in the CUDA version of the Rodinia Benchmark suite for heterogeneous computing [10], and a real-world inconsistency found in the C version of the ECMWF Cloud Physics mini-app CLOUDSC [14].

The synthetic GPU programs were generated with Varity [21], a framework that randomly generates small programs written in CUDA C along with an input for which a numerical inconsistency is observed when using `nvcc -O3 -fastmath` in comparison to `nvcc -O0`. These programs use single-precision floating-point arithmetic, various C syntax mechanisms such as `for-loop` and `if` statements, and calls to external math functions.

The CUDA NAS Parallel Benchmarks demonstrate the ability of Ciel to isolate compiler-induced floating-point inconsistencies in programs originally written for CPU architectures and ported to GPUs. These programs use double precision, which means the extended precision capabilities of Ciel are used. On the other hand, the Rodinia programs are originally written as heterogeneous applications for which single and double precision implementations are available. Therefore, we use the version in single precision.

Finally, CLOUDSC is a standalone mini-app of the ECMWF cloud microphysics parameterization, which tests the CLOUDSC cloud microphysics scheme of the ECMWF Integrated Forecasting System (IFS). We choose CLOUDSC as a candidate to demonstrate the efficacy of Ciel in finding compiler-induced inconsistencies in real-world applications, and show its capability of adapting to other software platforms and languages supported by Clang.

Experimental Environment. We use a PC with octa-core Intel(R) i7-11800H processors, and NVIDIA RTX 3070 GPU with 5120 CUDA cores, running Ubuntu 20.04 LTS. We use Clang version 14.0.6 to perform source-to-source transformation, which supports CUDA SDK versions up to 11.1 with Compute Capability up to 8.6. Clang 14.0.6 and nvcc 11.1 are also the compiler versions we use to compile transformed GPU programs. For CPU programs, we use gcc 9.4.0. Our methodology is independent of GPU models as long as they have the same Compute Capability. For all compilers, we considered two sets of compiler flags: `-O0`, and `-O3` with fastmath.

Table 2: Numerical inconsistencies in NAS, Rodinia and CLOUDSC programs.

Benchmark	Program	LOC	Input	Epsilon	Compilation Command
NPB-GPU	BT	5062	S	3.0e-12	`clang -O3 -ffast-math`
NPB-GPU	CG	1868	S	1.1e-15	`clang -O3 -ffast-math`
NPB-GPU	CG	1868	W	4.0e-16	`clang -O3 -ffast-math`
NPB-GPU	LU	4437	S	1.9e-12	`nvcc -O3 -use_fast_math`
NPB-GPU	MG	2349	W	2.9e-14	`clang -O3 -ffast-math`
Rodinia	LUD	717	256	1.2e-5	`nvcc -O0`
Rodinia	CFD	647	097K	7.2e-2	`nvcc -O0`
Rodinia	CFD	647	193K	1.9e-1	`nvcc -O0` & `-O3 -ffast-math`
N/A	CLOUDSC	2593	N/A	1.0e-11	`gcc -O3 -ffast-math`

Methodology for Triggering Numerical Inconsistencies. The compiler-induced numerical inconsistencies in NAS, Rodinia and CLOUDSC were previously unknown, and were discovered through testing. Specifically, we rely on verification routines that compare the relative errors in output values to an epsilon value ϵ to determine whether the results meet accuracy constraints. A compiler-induced inconsistency exists if a program passes its verification routines for some compiler settings but not for others.

For six of the NAS programs (BT, CG, FT, LU, MP and SP) and Rodinia LUD, we utilize existing verification routines where results are either compared to precalculated ground truth embedded in the program source code, or in the case of Rodinia LUD, the resulting two matrices are multiplied and then compared against the original matrix. For the Rodinia CFD Solver, we calculate the total density energy (TDE) as specified in [22] and compare it to the reference TDE value precalculated by running the double-precision version of CFD Solver compiled with `nvcc -O0`. For CLOUDSC, we compare relative errors for the main variables at the end of program execution against ground truth precalculated by running the original cloud scheme from IFS in FORTRAN.

We follow an existing methodology to *trigger* numerical inconsistencies, first introduced in [18]. Specifically, we set the epsilon value ϵ between the minimum and maximum errors observed amongst all compiler/optimization flag combinations for a given program, and maximize the ϵ value such that the program passes its verification routines only for some compiler settings but not for others. Table 2 lists the inputs, epsilon values, and compiler commands used to trigger each of the 9 numerical inconsistencies reported for these programs.

Evaluation Results. We find that Ciel is effective at isolating code responsible for the numerical inconsistencies in 337 out of 339 instances (99.4%). In terms of isolation granularity, Ciel isolates at expression level in 318 out of 339 instances (93.8%), while the rest of the inconsistencies are isolated at line, block, or function level. Below we describe the results per benchmark.

Synthetic GPU Programs. Ciel isolates all inconsistencies: individual expressions in 310 cases, a code block in 18, and a function in the remaining 2. We manually examined the source code, inputs, outputs, and in some cases the assembly code

Table 3: Categorization of inconsistencies found in synthetic GPU programs.

Categories	# Programs	Percentage	Sample Code
Subnormal Arithmetic	125	37.9%	`+1.8922E-42f + var_3`
`Inf` or `NaN` Arithmetic	53	16.0%	`+1.3797E-35f / -0.0f`
Math Functions	41	12.4%	`sinf(+1.0195E25f)`
Rounding Errors	18	5.5%	`-16458 / 1.67329e-16`
Program Inputs	164	49.7%	N/A
Print Statements	11	3.3%	N/A

of each of program. Our inspection revealed that CIEL correctly isolated 328 out of 330 (99.4%) inconsistencies while only 2 (0.6%) were false positives.

We identified six categories of *true* compiler-induced numerical inconsistencies isolated by CIEL. Table 3 lists these categories, the number of occurrences, and sample code. Note that an inconsistency may belong to multiple categories.

The first four categories are purely related to floating-point operations. *Subnormal arithmetic* indicates that subnormal numbers are involved in the floating-point operations. *Math functions* are often involved in which extreme values may be computed differently depending on the implementations. For example, nvcc compiles `sinf()` as a single fast approximation instruction instead of a full function. Also in some cases, `Inf` or `NaN` values are involved, which are not strictly IEEE 754-2008 compliant under fast math. We also found that the results of some operations differ under different optimization flags due to *rounding errors*.

The last two categories are related to the setup of the benchmark programs themselves. We observed cases where resolving compiler-induced inconsistencies also required enhancing the precision of their *program inputs*. And lastly, we found a few instances for which the final line of code where the computation result is *printed* byte by byte is the cause of the inconsistency. This is because the result of the computation is subnormal when converted from enhanced precision.

As for false positives, we found two cases where CIEL isolates statements that have no effect on the computation. Specifically, a variable is assigned a value that is immediately overwritten by another value. These assignment statements are located inside a loop. When the precision of the entire loop is enhanced, the inconsistency is resolved; but if only the precision of the statements after the initial assignment is enhanced, the inconsistency persists because of type conversions inserted by CIEL at the end of the region inside the loop.

Overall, CIEL took a total of 3 hours and 2 minutes to analyze all 330 programs, and 33 seconds per program on average.

NAS and Rodinia. CIEL isolated all 8 numerical inconsistencies in NAS and Rodinia programs. Table 4 shows the results for each program for both statement and expression level isolation. For BT.S, LU.S, MG.W, CIEL isolates variable expression(s) in one statement that causes the compiler-induced inconsistency. In CFD 097K, CIEL isolates a function call with a variable parameter. For CG and CFD 193K, CIEL isolates 2 variables across 2 to 3 statements as the cause of the inconsistencies. In all cases above, the isolated expressions are inside deeply

Table 4: NPB-GPU and Rodinia Experiment Results. Time is given in mm:ss.

	Statement Level				Expression Level		
Program	Isolated Function	Line(s)	# Cfgs	Time	Exp.	# Cfgs	Time
BT.S	`exact_solution`	1874-1886	10	1:23	`zeta`	20	2:10
CG.S	`sparse`	1710,1722	18	1:23	`size,shift`	24	1:52
CG.W	`sparse`	1710,1713,1765	19	1:34	`size,scale`	28	2:24
LU.S	`ssor_gpu_kernel_2`	4023	8	1:03	`tmp`	11	1:15
MG.W	`rprj3_gpu_kernel`	2045-2050	14	1:16	`x2,y2`	34	3:02
CFD 097K	`cuda_compute_step_factor`	283	14	6:01	`sqrtf`, `speed_sqd`	26	10:10
CFD 193K	`compute_speed_sqd`	252 257	10	7:29	`velocity`, `speed_sqd`	40	22:11
LUD 256	`lud_internal`	—	17	1:16	—	—	—

nested loops, so even a slight offset can be accumulated into a larger inconsistency that exceeds the error threshold. The only exception is the LUD program where only a function, `lud_internal` is isolated. Upon inspection, the reason seems to be that a variable `sum` is read and written throughout the function, affecting the whole matrix, and any type conversion would cause the inconsistency to persist.

Ciel isolated each inconsistency within 22 minutes, used less than 20 searches (configurations) for statement isolation, and used no more than 40 searches for expression isolation. About 1%-5% of run time is used on code transformation.

CLOUDSC. Ciel isolated a constant expression `(float)0.4` as the cause of the inconsistency. After looking further into the code repository [13] and reporting the issue to ECMWF scientists, we confirmed Ciel's result. It turns out ECMWF scientists had meant to temporarily introduce a bug during testing with the type casting but had forgotten to remove it; Ciel correctly suggests increasing the precision of that same argument to resolve the inconsistency. Ciel took 7 minutes to isolate the inconsistency, from which 8% is spent on program transformation.

Answer to RQ1: Ciel isolated 337 out of 339 inconsistencies in minutes with a precision of 99.4%, which included 328 synthetic GPU programs, NAS and Rodinia programs, and the mini-app CLOUDSC. In 318 cases (93.8%), Ciel isolated expressions. Manual inspection revealed inconsistency characteristics, such as the involvement of `Inf`, `NaN`, or subnormal numbers in arithmetic.

4.2 RQ2: Comparison with the State of the Art

Baseline. We compare Ciel to pLiner [18], to the best of our knowledge, the only tool available to isolate inconsistencies at the statement level in CPU programs.

Benchmarks. Due to pLiner capabilities, this evaluation is limited to CPU programs. We adopt benchmarks from the publicly available pLiner repository (SHA ef94b40)[5] originally used to evaluate pLiner, which include 50 floating-point synthetic CPU programs on Intel CPU platforms, and 3 programs from the C

[5] https://github.com/LLNL/pLiner/commit/ef94b40

Table 5: NPB CPU Experiment Results. Time is in minutes:seconds.

	CIEL Statement Level				pLiner Statement Level				CIEL Expression		
Prog.	Function	Line(s)	#Cfgs	T_{line}	Function	Line(s)	#Cfgs	T_{line}	Exp.	#Cfgs	T_{exp}
CG.B	`sparse`	814,819,876	19	16:23	`sparse`	—	7	3:53	—	—	—
SP.A	`tzetar`	65,69	16	6:56	`y_solve`	68	25	7:50	`r4,t2`	23	9:36
SP.B	`exact_solution`	44-47	9	17:28	`exact_solution`	44-47	17	24:51	`zeta`	19	34:57

version of the NAS Parallel Benchmark: CG.B, SP.A, and SP.B. We use the same compiler, optimization flags, and error thresholds as pLiner in our evaluation.

Evaluation Results. CIEL achieves more precise isolation results than pLiner for 84.9% of the programs. When isolating at the same statement level as pLiner, CIEL is 24.5% more efficient in terms of number of searches. The rest of this section describes the results per benchmark.

Synthetic CPU Programs. In 42 out of 50 programs, CIEL successfully isolates code at the statement level, and subsequently at the expression level. In 36 of these programs, CIEL isolates the same statement (line) as pLiner. In the remaining 6 cases, CIEL isolates at the statement level while pLiner can only isolate at code block or function level. CIEL explores 29.7% fewer configurations to achieve this result. On average, CIEL explores 5.2 configurations for statement isolation compared to 7.4 configurations explored by pLiner. Expression level isolation incurs in exploring additional configurations: 16.5 on average.

For the remaining 8 programs, there are two cases in which CIEL isolates a smaller code block than pLiner. There are four programs for which neither CIEL nor pLiner can resolve the inconsistencies by using precision enhancement. Lastly, there are two programs for which we were not able to reproduce the numerical inconsistencies. Note that in these cases, pLiner still proceeded with the search while CIEL immediately detected the absence of an inconsistency.

NAS CPU Benchmarks. Results for the NAS CPU benchmark are shown in Table 5. In CG.B, CIEL isolates three statements in function `sparse`, which has 227 lines of code, while pLiner can only isolate the whole function. pLiner stopped after it failed to resolve the inconsistency even when all basic blocks in `sparse` are in enhanced precision; CIEL prevents this by avoiding unnecessary type conversions between basic blocks. In SP.B, CIEL first isolates the same statement as pLiner in function `exact_solution`, and then further isolates a variable. Finally in SP.A, CIEL and pLiner isolate different functions (`tzetar` vs. `y_solve`) due to exploring different areas of the search tree. We confirmed that precision enhancement of either function resolves the inconsistency. If we were to limit the search in CIEL to only explore function `y_solve`, then CIEL would isolate the same statement as pLiner. Ultimately, CIEL isolates two variables.

In terms of efficiency, CIEL uses fewer configurations than pLiner to isolate inconsistencies in SP.A (16 vs. 25) and SP.B (9 vs. 17) at the statement level. CIEL uses more configurations for CG.B (19 vs. 7), but it isolates the same function with only 4 configurations, and isolates at a finer granularity. Expression isolation requires an additional 7 and 10 configurations for SP.A and SP.B, respectively.

Answer to RQ2: Ciel shows comparable or superior results in isolating the inconsistencies in 44 out of 48 (92%) of synthetic CPU programs and the NAS programs. Overall, Ciel isolates inconsistencies at the same level of granularity than pLiner but with higher efficiency, or at a finer level of granularity with an additional cost, in particular in the case of expression isolation.

4.3 Threats to Validity

While our evaluation set of programs is large and diverse, our results may not generalize to all applications. Also, compiler-induced numerical inconsistencies are input dependent, thus it is possible that other inputs could trigger additional inconsistencies in the same code regions, or elsewhere. Complementary use of dynamic analysis or code coverage information may be useful. Ciel does not handle non-deterministic applications, but some of such programs could still be analyzed by removing certain sources of non-determinism for testing purposes [27].

Ciel's implementation only handles a subset of C/C++ and CUDA platform constructs. Features such as anonymous functions or the `auto` keyword, introduced in C++11, are not currently supported. Handling some of these features may require a new approach in simplified AST generation and code isolation. Nevertheless, given how Ciel can differentiate between host and device code, it could be adapted to any platform supported by the Clang compiler frontend, such as OpenMP and OpenCL [3].

Code isolation may be further impacted by special floating-point values such as ± 0.0, `Inf`, and `NaN`. The processing of these values, if consistent across precisions but inconsistent between different optimization flags, may become a blind spot for precision enhancement. The choice of extended precision library may also impact search results and efficacy. GPUprec, for example, has known issues with math functions when it should return `NaN` but returns `zero` instead, which could affect isolation results. Finally, Ciel requires source code when enhancing precision. However, if the source code for a function is not available, Ciel may still isolate the call site if an enhanced precision variant of the function exists.

5 Related Work

Detecting and Isolating Numerical Errors. pLiner [18] isolates known compiler-induced numerical inconsistencies in C/C++ CPU programs at the line level. pLiner's approach also includes hierarchical code isolation and precision enhancement as a method to isolate inconsistencies. Unlike pLiner, Ciel works on heterogeneous programs, which pose unique challenges when isolating numerical inconsistencies, as described in Section 3.3. Furthermore, Ciel isolates inconsistencies to the expression level rather than lines. FLiT [28] generates and runs custom-made tests under different optimization levels to *trigger* compiler-induced numerical inconsistencies, which are then isolated at the function level only. Compared to Ciel, FLiT does not employ precision enhancement for inconsistency isolation, and focuses on CPU programs.

There are also tools that automatically detect or isolate specific categories of numerical errors but not compiler-induced inconsistencies. FPChecker [20] is a tool that automatically detects floating-point exceptions in GPU applications, which also uses Clang to transform CUDA code, but it does so at the IR level. While FPChecker operates on GPU programs, it does not isolate compiler-induced numerical inconsistencies. FPChecker also inspired other tools, such as Predoo [33] in the field of precision testing for Deep Learning (DL) operators. On the other hand, PFPSanitizer [12] detects numerical errors by performing shadow execution with higher precision in parallel. Shadow execution with precision enhancement is also employed by Herbgrind [26] and FPDebug [7] with the goal of finding floating-point precision errors.

Testing Compilers and Numerical Code. Ciel transforms source code in small increments and tests whether numerical inconsistencies are resolved. Ciel is inspired by prior work on compiler mutation testing. Le et al. [23] introduce equivalent modulo inputs (EMI) which mutates programs on unexecuted paths to expose compiler bugs that incorrectly execute these paths. ClassFuzz [11] uses EMI by mutating Java classfiles on predefined mutation operators, and send them to various JVM implementations for differential testing. Zhu and Zaidman [34] propose new mutator operations alongside conventional ones to expose bugs in GPU programs, but their work does not involve floating-point arithmetic. HeteroFuzz [32] introduces a multi-pronged fuzzing approach to detect platform-dependent divergence in heterogeneous programs running on FPGAs, using techniques including dynamic probabilistic mutations to reduce the long latency between invocations to hardware simulators. Overall, none of the above tools focus on exposing or isolating compiler-induced numerical inconsistencies.

Ciel performs differential testing to check whether compiler-induced inconsistencies exist by providing the same input to a series of programs compiled from the same source code but with various compilers and optimization flags. Differential testing has been applied before to numerical programs. FPDiff [29] performs differential testing between automatically identified *synonymous functions* across various numerical libraries to identify inconsistencies between the results from these functions under certain inputs. Unlike Ciel, FPDiff tests *different* implementations of a given function, and it does not consider different compilers or optimization flags.

6 Conclusion

With scientific code ported or developed on GPUs, compiler-induced numerical inconsistencies can arise at various stages of development. Unfortunately, automatic tools to isolate such problems are nonexistent, which harms productivity in GPU computing. In this paper, we demonstrate a practical method to identify the root cause of such inconsistencies in heterogeneous code. We implemented our approach in the tool Ciel based on the effective bisection search algorithm, and improved over the state of the art for CPU programs in both efficiency and accuracy. Most importantly, Ciel addresses a number of challenges to handle

heterogeneous code. Our evaluation on synthetic GPU programs, GPU benchmarks, and real world mini-app shows the effectiveness of Ciel at isolating inconsistencies in heterogeneous code with a precision of 99.4%. Our code and experimental data are publicly available at https://github.com/LLNL/Ciel/.

Acknowledgments

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-CONF-846081), 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.

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (2008), https://doi.org/10.1109/IEEESTD.2008.4610935
2. CUDA LLVM compiler (Oct 2018), URL https://developer.nvidia.com/cuda-llvm-compiler
3. Clang 14.0.0 documentation (2022), URL https://releases.llvm.org/14.0.0/tools/clang/docs/ReleaseNotes.html
4. Compiling CUDA with Clang (2022), URL https://releases.llvm.org/14.0.0/docs/CompileCudaWithLLVM.html
5. Ahn, D.H., et al.: Keeping science on keel when software moves. Commun. ACM **64**(2), 66–74 (2021)
6. 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)
7. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: PLDI, pp. 453–462, ACM (2012)
8. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. **18**(1), 1–33 (2010)
9. CEED: CEED/Laghos: High-Order Lagrangian Hydrodynamics Miniapp (2017), URL https://github.com/CEED/Laghos
10. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: IISWC, pp. 44–54, IEEE Computer Society (2009)
11. Chen, Y., Su, T., Sun, C., Su, Z., Zhao, J.: Coverage-directed differential testing of JVM implementations. In: PLDI, pp. 85–99, ACM (2016)
12. Chowdhary, S., Nagarakatte, S.: Parallel shadow execution to accelerate the debugging of numerical errors. In: ESEC/SIGSOFT FSE, pp. 615–626, ACM (2021)
13. ECMWF: CLOUDSC-V3: Re-create the single-exponent bug in the c variant (2019), URL https://github.com/ecmwf-ifs/dwarf-p-cloudsc/commit/d88c0c8f8d1effd5bd395cb71657629fb242f661
14. ECMWF: Standalone mini-app of the ECMWF cloud microphysics parameterization (2022), URL https://github.com/ecmwf-ifs/dwarf-p-cloudsc
15. Franco, A.D., Guo, H., Rubio-González, C.: A comprehensive study of real-world numerical bug characteristics. In: ASE, pp. 509–519, IEEE Computer Society (2017)
16. Fu, Z., Bai, Z., Su, Z.: Automated backward error analysis for numerical code. In: OOPSLA, pp. 639–654, ACM (2015)

17. Gopalakrishnan, G., Laguna, I., Li, A., Panchekha, P., Rubio-González, C., Tatlock, Z.: Guarding numerics amidst rising heterogeneity. In: Correctness@SC, pp. 9–15, IEEE (2021)
18. Guo, H., Laguna, I., Rubio-González, C.: pLiner: isolating lines of floating-point code for compiler-induced variability. In: SC, p. 49, IEEE/ACM (2020)
19. Joldes, M., Muller, J., Popescu, V., Tucker, W.: CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In: ICMS, Lecture Notes in Computer Science, vol. 9725, pp. 232–240, Springer (2016)
20. Laguna, I.: FPChecker: Detecting floating-point exceptions in GPU applications. In: ASE, pp. 1126–1129, IEEE (2019)
21. Laguna, I.: Varity: Quantifying floating-point variations in HPC systems through randomized testing. In: IPDPS, pp. 622–633, IEEE (2020)
22. Laguna, I., Wood, P.C., Singh, R., Bagchi, S.: GPUMixer: Performance-driven floating-point tuning for GPU scientific applications. In: ISC, Lecture Notes in Computer Science, vol. 11501, pp. 227–246, Springer (2019)
23. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI, pp. 216–226, ACM (2014)
24. Lu, M., He, B., Luo, Q.: Supporting extended precision on graphics processors. In: DaMoN, pp. 19–26, ACM (2010)
25. Nakayama, T., Takahashi, D.: Implementation of multiple-precision floating-point arithmetic library for GPU computing. In: PDCS, pp. 343–349 (2011)
26. Sanchez-Stern, A., Panchekha, P., Lerner, S., Tatlock, Z.: Finding root causes of floating point error. In: PLDI, pp. 256–269, ACM (2018)
27. Sato, K., Ahn, D.H., Laguna, I., Lee, G.L., Schulz, M.: Clock delta compression for scalable order-replay of non-deterministic parallel applications. In: SC, pp. 62:1–62:12, ACM (2015)
28. Sawaya, G., Bentley, M., Briggs, I., Gopalakrishnan, G., Ahn, D.H.: FLiT: Cross-platform floating-point result-consistency tester and workload. In: IISWC, pp. 229–238, IEEE Computer Society (2017)
29. Vanover, J., Deng, X., Rubio-González, C.: Discovering discrepancies in numerical libraries. In: ISSTA, pp. 488–501, ACM (2020)
30. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient automated repair of high floating-point errors in numerical libraries. In: POPL, pp. 56:1–56:29, ACM (2019)
31. Zeller, A.: Yesterday, my program worked. today, it does not. why? ACM SIGSOFT Software engineering notes **24**(6), 253–267 (1999)
32. Zhang, Q., Wang, J., Kim, M.: HeteroFuzz: fuzz testing to detect platform dependent divergence for heterogeneous applications. In: ESEC/SIGSOFT FSE, pp. 242–254, ACM (2021)
33. Zhang, X., et al.: Predoo: precision testing of deep learning operators. In: ISSTA, pp. 400–412, ACM (2021)
34. Zhu, Q., Zaidman, A.: Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to GPU programs. In: ICST, pp. 209–219, IEEE (2020)