An Interval Compiler for Sound Floating-Point Computations

Joao Rivera
Computer Science
ETH Zurich, Switzerland
hectorr@inf.ethz.ch

Franz Franchetti
Electrical and Computer Engineering
Carnegie Mellon University, USA
franzf@ece.cmu.edu

Markus Püschel
Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

***Abstract*—Floating-point arithmetic is widely used by software developers but is unsound, i.e., there is no guarantee on the accuracy obtained, which can be imperative in safety-critical applications. We present IGen, a source-to-source compiler that translates a given C function using floating-point into an equivalent sound C function that uses interval arithmetic. IGen supports Intel SIMD intrinsics in the input function using a specially designed code generator and can produce SIMD-optimized output. To mitigate a possible loss of accuracy due to the increase of interval sizes, IGen can compile to double-double precision, again SIMD-optimized. Finally, IGen implements an accuracy optimization for the common reduction pattern. We benchmark our compiler on high-performance code in the domain of linear algebra and signal processing. The results show that the generated code delivers sound double precision results at high performance. In particular, we observe speed-ups of up to 9.8 when compared to commonly used interval libraries using double precision. When compiling to double-double, our compiler delivers intervals that keep error accumulation small enough to compute results with at most one bit of error in double precision, i.e., certified double precision results.**

***Index Terms*—floating-point arithmetic, source-to-source compiler, interval arithmetic, certified accuracy.**

I. Introduction

Floating-point data types are a natural choice for software developers when implementing numerical computations. They are widely used in computer science and engineering, including in cyber-physical systems that interact with the physical world. Even relatively small embedded processors nowadays provide floating-point units. For example, the recent Helium Vector Extension [1] by ARM targets low-power microcontrollers and supports vectorized single and half precision floating-point operations.

Unfortunately, floating-point numbers do not exactly represent real values and thus can suffer from roundoff errors. Their non-intuitive nature can make it hard to predict when and how these errors accumulate to the extent that they invalidate the final result. In the worst case this can have catastrophic effects [2]. Thus, it is important to provide guarantees (i.e, sound floating-point arithmetic) and do so fast for any given input. The need for floating-point soundness extends to different domains. For example, the work in [3] highlights the need of soundness for the stability of control systems. Similarly, [4] provides guarantees for a collision-avoidance monitor using sound floating-point arithmetic in its implementation. In abstract domains for verification, sound floating-point arithmetic could replace rational arithmetic to gain speed [5], [6]. For the robustness analysis of neural networks, sound floating-point computations are essential [7], [8]. Another direction is mixed-precision tuning in which tools estimate error bounds on floating-point computations to select a suitable precision [9], [10].

One common technique to guarantee soundness is static code analysis based on abstract interpretation using intervals or more complex polyhedra [11]. Many tools have emerged to guarantee soundness by estimating roundoff errors using static analysis [12]–[17]. However, since the input is not known, the bounds become too lose. A more practical approach is thus to rewrite the code to account for ranges, either from scratch or using libraries [18]–[21]. The cheapest solution is interval arithmetic [22], which treats all values as intervals that are guaranteed to contain the true value. However, the manual effort can be considerable, the resulting code can become significantly slower than the original, and, most importantly, too imprecise. Namely, intervals approximate conservatively and thus the bounds can quickly become too loose to be helpful. Thus, the challenges are accuracy (keeping intervals small), fast execution, and automation.

Contributions. In this paper, we address this problem with IGen, a source-to-source compiler for interval arithmetic. IGen takes as input a C function doing floating-point computations, possibly using SIMD intrinsics, and outputs a C function, possibly optimized using intrinsics, that implements the same function soundly using intervals.

We make the following contributions:

- The design and implementation of IGen, supporting a significant subset of C.
- Support for a large subset of Intel AVX intrinsics in the input using a generator that produces C code from their XML specification as provided by Intel. In addition, support for AVX intrinsics in the output function.
- Support for intervals with endpoints in double-double precision. Using this precision can keep error accumulation small enough to compute certified results in double precision.
- An algorithmic accuracy transformation for the common reduction pattern in linear-algebra-type computations and beyond.

- A thorough evaluation of execution time and accuracy on a variety of benchmarks.

II. Background: Interval Arithmetic

Interval arithmetic [22] is a classical tool for sound computation with floating-point values. Instead of representing a real value by a float, one uses an interval with floating-point boundaries that contains the real value. Computations on reals are then done by operations on intervals that preserve soundness, i.e., the result of any interval operation is again guaranteed to contain the real result.

Formally, an interval $\widehat{a} = [a_{\text{inf}}, a_{\text{sup}}]$ is defined as the set of real numbers between and including its endpoints, which are floating-point values:

$$[a_{\text{inf}}, a_{\text{sup}}] = \{x \in \mathbb{R} \mid a_{\text{inf}} \leq x \leq a_{\text{sup}}\},$$

where $\mathbb{R}$ is the set of all real numbers. a_{inf} and a_{sup} are the lower and upper endpoints respectively. Operations on intervals are done using the endpoints. For example, the addition and multiplication of intervals is computed as:

$$\begin{aligned}
\widehat{a} + \widehat{b} &= [\mathrm{RD}(a_{\text{inf}} + b_{\text{inf}}), \mathrm{RU}(a_{\text{sup}} + b_{\text{sup}})], \\
\widehat{a} \cdot \widehat{b} &= [\min\big(\mathrm{RD}(a_{\text{inf}} \cdot b_{\text{inf}}), \mathrm{RD}(a_{\text{inf}} \cdot b_{\text{sup}}), \\
&\qquad \mathrm{RD}(a_{\text{sup}} \cdot b_{\text{inf}}), \mathrm{RD}(a_{\text{sup}} \cdot b_{\text{sup}})\big), \\
&\quad \max\big(\mathrm{RU}(a_{\text{inf}} \cdot b_{\text{inf}}), \mathrm{RU}(a_{\text{inf}} \cdot b_{\text{sup}}), \\
&\qquad \mathrm{RU}(a_{\text{sup}} \cdot b_{\text{inf}}), \mathrm{RU}(a_{\text{sup}} \cdot b_{\text{sup}})\big)],
\end{aligned}$$

where $\mathrm{RD}(x)$ is the downward rounding function and $\mathrm{RU}(x)$ is the upward rounding function. The use of these rounding modes ensures soundness. Since changing the rounding mode is usually expensive on processors, one uses the identity $\mathrm{RD}(-x) = -\mathrm{RU}(x)$ to use only one rounding mode. For example, the interval addition then takes the form

$$\widehat{a} + \widehat{b} = [-\mathrm{RU}((-a_{\text{inf}}) - b_{\text{inf}}), \mathrm{RU}(a_{\text{sup}} + b_{\text{sup}})]\,.$$

Another common trick avoids the negation operation of the lower endpoints by keeping them negated by default [23]. This way interval addition only requires two additions and interval multiplication eight multiplications and six comparisons (max and min require 3 comparisons each).

III. IGen Overview

In this paper we present IGen, a source-to-source compiler that translates a numerical C program performing floating-point computations to an equivalent sound numerical C program using interval arithmetic. By sound we mean that the resulting intervals are guaranteed to contain the result of the original program if it were performed using real arithmetic.

Fig. 1 shows the high-level architecture of IGen. The input is a C function, possibly using SIMD intrinsics, performing floating-point computations and a target precision for the interval endpoints. This target precision can be either single or double precision (the default) or the higher double-double precision [24], which we explain in Section VI. The output is an equivalent C function performing the same computation soundly using interval arithmetic, optionally optimized with SIMD

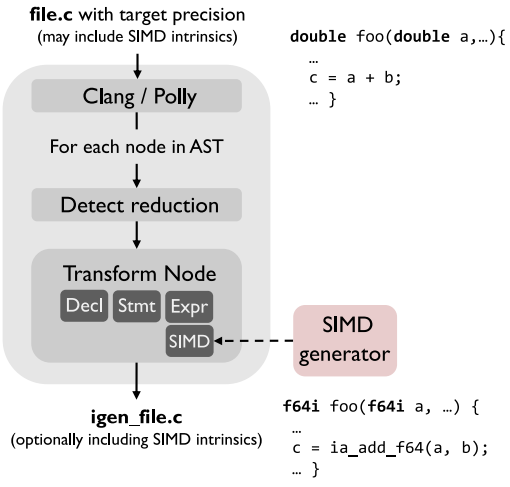

Fig. 1. High-level architecture of IGen.

intrinsics. In the first step, IGen uses the Clang LibTooling library [25] to construct the abstract syntax tree (AST) of the computation. The AST is then traversed and, for each node, IGen produces an equivalent set of interval operations, possibly in increased precision. The interval operations are provided by an interval library that we developed as part of IGen. This library supports all the basic operations, and elementary functions in all precisions and is also optimized for SIMD instructions.

The support of SIMD intrinsics in the input function is handled by a generator that automatically produces the interval implementation of (a large subset of) SIMD intrinsics from its XML specification. IGen then transforms each intrinsic to its equivalent interval implementation.

Finally, IGen performs an algorithmic accuracy transformation. Using Polly [26], it detects the reduction pattern, which is particularly sensitive to error accumulation, and replaces each reduction by an input-cognizant algorithm as explained in Section VI.

Section IV explains the design and implementation of the compiler in detail. Section V details the implementation of the generator to support SIMD intrinsics in the input. In Section VI, we explain the methods used to increase precision and to improve the accuracy of reductions.

IV. Source-to-Source Interval Compiler

In this section, we describe our interval compiler IGen in detail. For the needed interval arithmetic in the output function, we implemented our own library which is described first. Then we explain the compilation process and the implementation of custom language extensions in our compiler.

A. Interval Library

Our interval library is intended to be fast, precise, and tightly coupled with our compiler. As explained in Section II, we use only upward-rounding and represent an interval $[a, b]$ as the pair $(-a, b)$ for efficiency. The library does not fully comply with the relatively new IEEE standard for interval

TABLE I
SUPPORTED OPERATIONS IN OUR INTERVAL LIBRARY.

Type	Operations
Arithmetic	Add, Sub, Mul, Div, Neg
Comparison	$<, \leq, >, \geq, =, \neq$
Elementary	Sqrt, Abs, Floor, Ceil, Exp, Log, Trigonometric
Other	Casts, Three-valued logic, Auxiliary functions

arithmetic [27] since we do not provide support for decorations, nor address the standard's accuracy requirements for double-double. Decorations are tags attached to an interval to provide additional information about how that interval was derived.

Basic interval types and operations. We use standard floating-point types to store the endpoints, i.e., the library supports single precision and double precision interval data types named `f32i` and `f64i`, respectively. In addition, we also support an interval data type for double-double precision named `ddi`, explained later in Section VI. The library implements all basic arithmetic operations including addition, multiplication, division, along with comparison operations and elementary functions. Table I shows a more comprehensive list of the supported operations.

Vectorized intervals. Using intrinsics, we also provide a vectorized implementation of the basic operations for double precision by storing the intervals in SSE registers similar to the work in [28]. Note that a double precision interval can fit exactly in a `_m128d` vector type. Vectorized implementations for arithmetic operations are implemented accordingly. For example, an interval addition can now be computed with a single SIMD instruction.

Elementary functions. We use the CRlibm [29] library as the building block to support interval elementary functions (see Table I). CRlibm guarantees that the result of an elementary function is computed as if done in infinite precision and then rounded, i.e., at most 1 bit of precision is lost. The interval implementation for monotonic elementary functions, e.g., sqrt and log, is straightforward, by applying the corresponding CRlibm elementary function to each endpoint. For non-monotonic functions such as sine, we first divide the function into monotonic sections and determine in which sections the endpoints are located. Then, we apply the corresponding CRlibm function to the endpoints.

Handling NaN and infinity. Endpoints of intervals can become NaNs or infinity and it is important to handle these cases correctly and soundly. This situation means that we have lost information or possibly the operation performed was invalid. In particular, any interval containing at least one NaN means that an invalid operation was performed and thus, the floating-point represented by the interval could be anything, including a NaN. For example, $\texttt{sqrt}([-1, 1]) = [\text{NaN}, 1]$. Further, the interval $[-\infty, \infty]$ means the represented number could be any floating-point except a NaN, and $[\infty, \infty]$ means that the value is larger than the maximum representable floating-point. Finally, $[1, \infty]$ represents any value greater or equal to one.

Example input function:

```
1: double foo(double a, double b) {
2:   double c;              // Decl
3:   c = a + b + 0.1;       // Expr
4:
5:   if (c > a) {           // Stmt
6:     c = a * c;
7:   }
8:   return c;
9: }
```

IGen output function:

```
1: #include "igen_lib.h"
2:
3: f64i foo(f64i a, f64i b) {
4:   f64i c;
5:   f64i t1 = ia_add_f64(a, b);
6:   f64i t2 = ia_set_f64(0.099999999999999992,
                          0.100000000000000005);
7:   c = ia_add_f64(t1, t2);
8:
9:   tbool t4 = ia_cmpgt_f64(c, a);
10:   if (ia_cvt2bool_tb(t4)) { //It may signal exception
11:     c = ia_mul_f64(a, c);
12:   }
13:   return c;
14: }
```

Fig. 2. Interval transformation of function `foo`.

We extensively tested the basic operations and elementary functions of our library to verify that these cases are handled soundly. In the test cases, we randomly tested combinations of NaNs, infinity, Zero and other special inputs such as denormals in the endpoints of intervals. We validated the result against MPFI [21], which is a multi-precision library for interval arithmetic.

B. Interval Compiler

We now explain the compilation process shown in Fig. 1 in detail. After generating the AST using Clang, IGen visits the top nodes to perform the necessary transformations. Clang specifies three types of basic nodes: declarations (`Decl`), statements (`Stmt`), and expressions (`Expr`)[1]. `Decl` nodes declare variables, functions, fields, etc. `Expr` nodes represent binary or unary operations, function calls, constants, and others. Finally, `Stmt` nodes are used for loops, if-else statements, and other special statements such as return and attributes (used for pragmas). Figure 2 shows a function `foo` where the node type is added as a comment.

Declarations. When a `Decl` node is visited, IGen generates code for the declaration promoting floating-point types to intervals as specified in Table II. Note that SIMD types are also transformed to an equivalent interval representation (explained in Section V). Lines 3–4 of Fig. 2 (bottom) shows the promotion of the function `foo` and the variable `c` to interval types. Since double precision is commonly implemented efficiently in modern computers, we convert single precision floats to double precision intervals by default but double-double (DD) is also supported. Other types, such as pointers and arrays, that

[1]`Expr` nodes are also statements in the Clang's AST but we handle them separately.

TABLE II
SUPPORTED TRANSFORMATIONS FROM FLOATING-POINT AND SIMD TYPES TO INTERVAL TYPES AND ITS VECTOR IMPLEMENTATION.

FP type	Interval Type	
	Double	DD
`float`, `double`	`f64i`	`ddi`
`_m128d`	`m256di_1`	`ddi_2`
`_m128`, `_m256d`	`m256di_2`	`ddi_4`
`_m256`	`m256di_4`	`ddi_8`

Interval	Vector implement.
`f64i`	$1\times$`_m128d`
`ddi`	$1\times$`_m256d`
`ddi_`k	$k\times$`_m256d`
`m256di_`k	$k\times$`_m256d`

are not floating-point types but refer to one of them are also transformed accordingly to refer to interval representation.

Expressions. IGen transforms `Expr` nodes to equivalent interval expressions. Internally, all `Expr` nodes are transformed to an object `igenExpr` that stores its generated interval representation and additional attributes, such as type, and whether it represents a constant. Lines 5–7 of Fig. 2 show the transformation of the expression `c = a + b + 0.1`. As expected, two interval additions are generated. Further, the constant term is lifted to an interval. In addition, IGen detects elementary functions by checking the name and signature of function calls which are also `Expr` nodes. When the function matches one of the supported elementary functions, the compiler replaces it with its equivalent interval function. For example, `sin(x)` is converted to `ia_sin_f64(x)`.

Interval constants. Constants are converted to intervals in a conservative way to guarantee soundness. There are two cases: 1) Integer constants are converted to exact intervals. For example, `1.0` is converted to `[1, 1]`. 2) Non-integer constants are enclosed by an interval whose endpoints are the two neighbouring floating-point values of the constant. Thus, a constant x that is not representable as a floating-point value, e.g. `0.1`, is converted to an interval of length ulp(x), where ulp(x) is the unit in the last place defined as the gap between the two adjacent floating-point numbers enclosing x. Further, a constant x that has an exact floating-point representation, e.g. `0.5`, converts to an interval of length 2ulp(x) with center in x. IGen also supports constant folding on intervals. Thus, expressions such as `2.0 + 0.1` are directly transformed to a single interval constant by the compiler.

Statements. `Stmt` nodes are mainly used to transform loops and branches. In most cases, the structure of a statement is not changed during transformation, only its child `Expr` nodes are transformed, so loops are straightforward. `if`-statements become interesting if the condition compares floats. For example, the interval comparison $[0, 2] < [1, 3]$ cannot be evaluated to true or false and hence leads to an unknown state. To handle this case, we implemented a special data type `tbool` (see lines 9–10 in Fig. 2) that represents a boolean with three possible states (true, false, or unknown), and that is evaluated in the `if` condition. IGen provides two options to handle the unknown state in a branch: 1) an exception is signaled when detected (the default) or 2) an extra branch is generated to handle this case as explained next.

Example input function:

```
double read_sensor(double:0.125 a) {
  double c = 5.0 + 0.25t;
  return a + c;
}
```

IGen output function:

```
#include "igen_lib.h"
f64i read_sensor(double a) {
  f64i _a = ia_set_tol_f64(a, 0.125);// _a = a +- 0.125
  f64i  c = ia_set_f64(4.75,5.25);    //  c = [4.75,5.25]
  f64i t1 = ia_add_f64(_a, c);
  return t1;
}
```

Fig. 3. Language extensions to support known tolerances.

Unknown-state in if-else statements. When the branch to take in a `if-else` statement is unknown, IGen also supports the alternative to generate code that computes both branches and joins the resulting intervals at the end. We did not implement this approach in the case that arrays of intervals or integer variables are modified in the body, thus it is not enabled by default. In our experiments, this case does not occur.

Limitations. Operations performing bit-level manipulation on floating-point variables are not supported by our compiler. Further, casts from floating-point to integer are not supported since we do not implement intervals on integers. Dynamic memory allocations can be dangerous when not done properly. For example, `double* a = malloc(8)` could incorrectly be transformed to `f64i* a = malloc(8)`. IGen shows a warning when malloc is used. Finally, since we target performance, a natural choice is to start by targeting C code as input and generate C code as output. However, further C++ support is feasible and possible future work.

C. Language Extensions

There are cases where it is necessary to express the known imprecision of variables and constants. For example, in cyber-physical systems the input may come from a sensor with known resolution, or a constant may be derived empirically up to a known error bound. This situation cannot be inferred from the code but has to be available when transforming to a sound interval. To support this situation, we implemented two custom C language extensions for use by the programmer. First, input parameters of a function can be annotated with a colon symbol followed by a constant literal expressing its tolerance error. For example, `double:0.25` is a double precision float whose value is precise with an error margin of 0.25. Second, a floating-point constant can be annotated with a postfix `t` to represent an interval around zero. For example, the constant `5 + 0.25t` represents the interval $[4, 75, 5.25]$. Fig. 3 shows an example and an associated output of IGen.

V. AUTOMATIC SUPPORT OF SIMD INTRINSICS

IGen provides support for SIMD SSE and AVX intrinsics in the input function and, by default, the interval types in the output use SIMD types. Also by default, single precision intrinsics are transformed to equivalent double precision interval

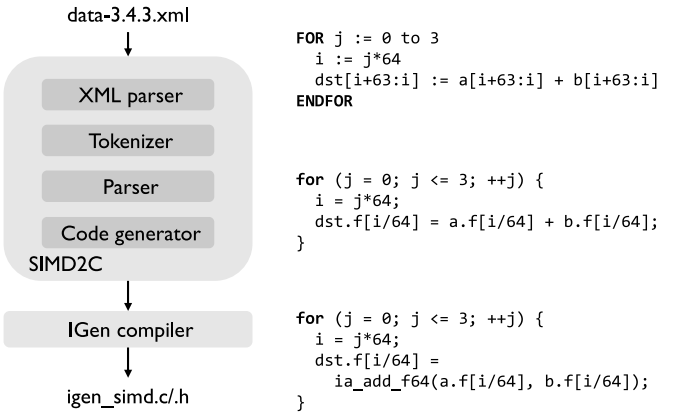

Fig. 4. Generating interval version of SIMD intrinsics from vendor's specification.

intrinsics. Further, even scalar (non-SIMD) input can be transformed to SIMD code by pairing the endpoints in a SSE register, thus, implementing 2-way vectorization.

The first step towards supporting SIMD intrinsics in the input programs is to implement equivalent sound operations in C, which can then be processed further by IGen. There are currently over 6,000 Intel SIMD intrinsics of which around 40% perform floating-point computations. Thus, implementing these by hand is out of question. Instead, we implemented a code generator that produces these implementations automatically from their XML specification provided by Intel [30], using an approach inspired by [31]. The work in [31], however, generated embedded domain-specific languages to express intrinsics inside managed languages and did not generate C code from the specification. Figure 4 shows a high level overview of the generator, which is explained in detail next.

XML parser. The Intel Intrinsics Guide [30] provides an XML file containing the specification of each available intrinsic. Figure 5 (top) shows an example for `_mm256_add_pd`. We extract relevant information for each intrinsic such as its name, return type, parameter list, and the operation that it performs. The operation is specified using a C-like pseudo-language. We only consider intrinsics that perform floating-point operations.

Tokenizer and parser. Our generator produces the AST for each intrinsic based on the information extracted from its XML specification. To do this, we first defined the associated grammar for the Intel pseudo-language, which we used to implement the parser.

Code generator. After parsing, we traverse the AST and generate C code. For each SIMD vector type we define a custom union data type that encapsulates the vector with an array of floats and integers. Lines 1–5 of the generated code in Fig. 5 show an example of this data structure. Using it, we can directly access the vector's elements. During the generation of a SIMD intrinsic, all vector input parameters are transformed to this custom type as shown in line 8 of the example. Further, all used intermediate variables are declared with its inferred type (line 9). Most of the statements in the Intel pseudo-language are similar to C and thus can be directly transformed.

XML specification:

```
<intrinsic rettype='__m256d' name='_mm256_add_pd'>
    <type>Floating Point</type>
    <CPUID>AVX</CPUID>
    <category>Arithmetic</category>
    <parameter varname='a' type='__m256d'/>
    <parameter varname='b' type='__m256d'/>
    <description>
      Add packed double-precision (64-bit)
      floating-point elements in "a" and "b",
      and store the results in "dst".</description>
    <operation>
      FOR j := 0 to 3
        i := j*64
        dst[i+63:i] := a[i+63:i] + b[i+63:i]
      ENDFOR
      dst[MAX:256] := 0
    </operation>
    <instruction name='vaddpd' form='ymm, ymm, ymm'/>
    <header>immintrin.h</header>
</intrinsic>
```

Generated C code:

```
 1: typedef union {
 2:   __m256d   v;
 3:   uint64_t i[4];
 4:   double   f[4];
 5: } vec256d;
 6:
 7: __m256d _c_mm256_add_pd(__m256d _a, __m256d _b) {
 8:   vec256d dst, a = {.v = _a}, b = {.v = _b};
 9:   int i, j;
10:   for (j = 0; j <= 3; ++j) {
11:     i = j * 64;
12:     dst.f[i/64] = a.f[i/64] + b.f[i/64];
13:   }
14:   return dst.v;
15: }
```

Fig. 5. XML specification of the `_mm256_add_pd` intrinsic and its generated C implementation. The notation `v[a:b]` in the XML specification means that the bits `a`...`b` in a vector `v` are accessed.

In contrast to C, however, it is common in the Intel pseudo-language to access parts of a vector using the corresponding range of bits. For example, accessing the j-th element of a `__m256d` vector `v` is expressed as `v[i+63:i]` where `i = 64 * j` (see the operation in Fig. 5). To support this, we first derive symbolically the number of bits accessed in a vector. If the number of bits is the same as the size of one of its elements, we access the vector's element using either the floating-point array or integer array depending on the operation to perform. This is shown in line 12 of the example, where each bit-access expression is transformed to an access of a floating-point array, e.g., `a.f[i/64]`. It is also possible that more than one element of a vector is accessed (see for example `_mm256_load_pd` in [30]). Note that the integer array is useful when accessing a subset of bits of a vector's element, or when performing bit-wise operations.

We also took care of small differences in the semantics between C and the Intel pseudo-language. For example, the pseudo-language allows testing multiple variables for equality using the expression `a == b == c`, which is not the proper way to do it in C.

Generation of interval intrinsics. The final step in the generation process is to use IGen to translate the C implementation of the intrinsics to an interval implementation. The only addition to the compiler is the support of bit-wise

logical operations on intervals. This is used for example in the intrinsic `_mm256_and_pd` where a bit-wise logical AND is applied to two `_m256d` vectors. These operations are performed endpoint-wise for each interval in the vector. In general, bit-wise operations are only safe to use with intervals if one of the intervals is a mask with either all bits sets to one or to zero. Since, this is a common use case for these operations on floating-point vectors, we decided to support them.

Limitations. Similar to scalar code, any operation performing bit level manipulation or conversion from floating-point to integer is not supported. In addition, the compiler currently lacks support for intrinsics that modify internal registers, e.g. `_mm256_testz_pd`. However, supporting this type of intrinsics is feasible with some additional effort. Finally, there are intrinsics using functions which operations are not defined and, thus, had to be implemented manually, e.g. `Convert_FP64_To_FP32` in `_mm256_cvtps_pd`.

Optimized implementations. The automatic approach presented so far generates implementations that closely resemble the XML pseudo code. Since this often leads to inefficient code, we also implemented a small set of very common intrinsics by hand, directly using intrinsics. IGen detects these by checking name and signature and inserts the hand-optimized implementation for the output. Otherwise, it uses the automatically generated interval implementation.

VI. Improving Accuracy

Interval arithmetic can suffer from too loose overapproximation since only bounds are maintained and relationships between variables are ignored (unlike, for example, in the considerably more expensive affine arithmetic [32]). To mitigate this there are two possible avenues: 1) increasing the precision of the endpoints beyond the standard double precision, and 2) program transformations that improve the accuracy of the computations. IGen provides examples of both as explained next.

A. Increasing the Precision with Double-Double

Double-double arithmetic is an efficient software technique to increase precision on processors with native support for double precision [24]. A double-double number a is an unevaluated sum of two double precision numbers, a_h and a_ℓ,

$$a = a_h + a_\ell, \quad \text{such that} \quad a_h = \mathrm{RN}(a), \tag{1}$$

where $\mathrm{RN}(x)$ is the round-to-nearest function as defined in IEEE 754. This property guarantees that the significands of a_h and a_ℓ do not overlap [33]. Thus, double-double provides a precision of at least 106 bits except for values very close to zero. In comparison, quad precision provides 113 bits of precision. As an example, the double-double that best approximates π is $\pi_h = 3.141592653589793116$ and $\pi_\ell = 1.1030637736600981 1247 \times 2^{-53}$ [34]. However, the range of double-double is the same as in double precision since the exponent's size does not change (11 bits).

Interval double-double arithmetic using AVX. IGen provides double-double intervals as compilation target for both single and double precision input functions, possibly using intrinsics. The exact default promotion of floating-point types is shown in Table II. IGen support for double-double requires an extension of our interval library (Section IV-A).

DD_Add(x_h, x_ℓ, y_h, y_ℓ):
(s_h,s_ℓ) ← TwoSum(x_h, y_h)
(t_h,t_ℓ) ← TwoSum(x_ℓ, y_ℓ)
$c \leftarrow \mathrm{RU}(s_\ell + t_h)$
(v_h,v_ℓ) ← FastTwoSum(s_h, c)
$w \leftarrow \mathrm{RU}(t_\ell + v_h)$
(z_h,z_ℓ) ← FastTwoSum(v_ℓ, w)
return (z_h,z_ℓ)

TwoSum(a, b):
$s \leftarrow \mathrm{RU}(a + b)$
$a' \leftarrow \mathrm{RU}(s - b)$
$b' \leftarrow \mathrm{RU}(s - a)$
$\delta_a \leftarrow \mathrm{RU}(a - a')$
$\delta_b \leftarrow \mathrm{RU}(b - b')$
$e \leftarrow \mathrm{RU}(\delta_a + \delta_b)$
return (s,e)

Fig. 6. Double-double addition (left) and TwoSum (right).

TABLE III
Number of Intrinsics in Double-Double Interval Operations.

Operation	Flops	SIMD intrinsics		
		Arithmetic	Shuffles	Total
Addition	40	14	3	17
Multiplication	114	27	29	56
Division	158	48	37	85

An interval with double-double precision is represented using four double precision numbers (two doubles for each endpoint) and thus it is a natural fit for AVX vector registers, which means `_m256d` is the base type of our double-double interval type `ddi`. For the vectorized implementation of addition, subtraction, multiplication, and division, we used the most accurate algorithms available in the literature [35]. Figure 6 shows as example the algorithm for the addition of two double-double numbers. The functions TwoSum and FastTwoSum compute both the result and the roundoff error of adding two double precision numbers.

Table III shows the number of floating-point operations (flops) for each double-double interval operation and the number of intrinsics in its vectorized implementation. As can be seen, the cost of double-double intervals is rather high; a single interval addition already requires 40 flops. Thus, it is important to provide optimized implementations. As of now, we do not support elementary functions in double-double precision. Further, double-double arithmetic is normally done using round-to-nearest. Thus, the challenge is to implement sound double-double intervals with upward-rounding.

Soundness guarantee. The definition of double-double in (1) slightly changes when using upward-rounding. Using a rounding mode other than to-nearest slightly degrades the accuracy of double-double computations [36]. However, it enables sound interval operations.

Lemma 1. *Let x, y be two double-double numbers and z be their sum computed using algorithm DD_Add (Fig. 6). When using upward-rounding, z is an upper bound of the exact sum $x+y$, i.e., $z_h + z_\ell \geq x_h + x_\ell + y_h + y_\ell$. Similarly, z is a lower bound of the exact sum when using downward-rounding.*

Proof. The work in [36] proved that computing TwoSum and FastTwoSum using upward-rounding yields an upper bound of

the exact result. Thus, lines 2, 3, 5 and 7 yield the inequalities $s_h + s_\ell \geq x_h + y_h$, $t_h + t_\ell \geq x_\ell + y_\ell$, $v_h + v_\ell \geq s_h + c$, $z_h + z_\ell \geq v_\ell + w$. Further, it is clear that $c \geq s_\ell + t_h$ and $w \geq t_\ell + v_h$ (lines 4 and 6). Reordering and simplifying these inequalities yields $z_h + z_\ell \geq x_h + x_\ell + y_h + y_\ell$ which concludes the proof for the upper bound. The proof for the lower bound follows analogously. □

From the lemma we conclude that the algorithm for double-double addition is suitable for interval arithmetic. Analogously, we can prove soundness for the multiplication and division of two double-double numbers.

B. Accuracy Transformation: Reduction

Another avenue for improving accuracy are code transformations. As an example, we focus on the transformation of reductions, and specifically summations, a common pattern in linear algebra computations and beyond that is particularly sensitive to error accumulation. A common example is the ubiquitous scalar product of two vectors.

Our approach consists of two steps: 1) detecting reductions, and 2) replacing the computation with a more accurate method. We explain the details.

Reduction detection. We detect reductions in the source code in two steps. We first use Polly [26] to detect reductions at the LLVM-IR level and extract relevant information. Then, we search for the node performing the reduction in the AST.

Polly is a loop optimizer and part of the LLVM compiler framework that uses the polyhedral model to analyze and optimize programs. Polly implements a reduction detection mechanism [37] to exploit possible parallelism but that we use to improve accuracy. Figure 7 shows example code and relevant information of the analysis performed by Polly. As can be seen, Polly specifies the reduction type + (summation) and reduction dependence $(i_0, i_1) \rightarrow (i_0, i_1 + 1)$ for the matrix-vector multiplication function in the example. In the example, the dependence indicates that there is a loop carried self-dependence due to a reduction in the inner loop. Further, we can also extract the location in the source file (line and column) of the instruction where Polly detects the reduction by enabling debug information when generating the LLVM-IR. When enabling code transformations for accuracy in our compiler, IGen calls Polly and extracts the location and dependence information of all detected reductions.

We extended the clang compiler with OpenMP-like pragmas to make it possible for programmers to specify the loops where the reduction transformation should be enabled and to specify the variable or array to reduce. When a loop is detected with this pragma, IGen searches for the `Expr` node inside the loop performing the reduction using the location information extracted from Polly. In the `mvm` example of Fig. 7, the variable to reduce is `y` (specified in line 2) and the detected reduction expression is in line 5. Once the reduction is detected, IGen proceeds to perform the necessary code transformations.

Code transformations. There are four components that are generated to improve the accuracy of reductions. First, an accumulator is declared (see line 2 in Fig. 7 bottom) where the elements of the reduction will be accumulated with high accuracy. Second, the accumulator is initialized and the first element is added (see line 4). Third, IGen extracts the term to accumulate, e.g. `A[i*500+j]*x[j]` in the example, and its interval transformation is added to the accumulator (lines 7). Finally, the elements in the accumulator are reduced and the result is assigned to the reduction variable (line 9). IGen uses the information on the reduction dependence provided by Polly to detect in what level in a nested loop the accumulator should be initialized and perform its final reduction. In the example, the initialization and final reduction are done before and after the innermost loop, respectively.

Matrix-vector multiplication:

```
1: void mvm(double* A, double* x, double* y) {
2: #pragma igen reduce y
3:   for (int i = 0; i < 100; i++)
4:     for (int j = 0; j < 500; j++)
5:       y[i] = y[i] + A[i*500+j]*x[j]; // Reduction
6: }
```

Polly Analysis: .

```
function 'mvm': (Max Loop Depth: 2)
  ...
  Reduction dependences [Reduction Type: +]:
    Stmt3[i0, i1] -> Stmt3[i0, 1 + i1] : 0 <= i0 <= 98
  ...                                    and 0 <= i1 <= 499
  Printing analysis:
     for (int c0 = 0; c0 <= 99; c0 += 1)
       for (int c1 = 0; c1 <= 499; c1 += 1)
         Stmt3(c0, c1);
```

IGen Transformation: .

```
1:void mvm(f64i* A, f64i* x, f64i* y) {
2:  acc_f64 acc1;                          // Declaration acc
3:  for (int i = 0; i < 100; i++) {
4:    isum_init_f64(&acc1, y[i]);          // Initialization
5:    for (int j = 0; j < 500; j++) {
6:      f64i t1 = ia_mul_f64(A[i*500+j], x[j]);
7:      isum_accumulate_f64(&acc1, t1);// Accumulate term
8:    }
9:    y[i] = isum_reduce_f64(&acc1);    // Final reduction
10: }
11:}
```

Fig. 7. From top to bottom: 1) Example code implementing matrix-vector multiplication. 2) Relevant information when executing `opt` on the LLVM-IR of `mvm` function with flags `-analyze -polly-dependences -polly-ast`. 3) IGen code improving accuracy of reduction.

Summation method. When targeting double precision intervals, IGen simply uses an accumulator with double-double precision to improve accuracy. When targeting double-double intervals, the cost for a higher precision accumulator would be too high. Thus, as a different approach, we use an accurate summation method similar to [38], [39]. More precisely, in this case the accumulator consists of two double precision arrays of size $n = 4096$; one for each endpoint. Given an interval to accumulate, the following is done for each of its double-double endpoints:

1) The double-double value is split into its higher and lower double precision terms. For example, $t_h = 2.0$ and $t_\ell = 1.0 \times 2^{-53}$ for a double-double endpoint.

2) Each double precision term t is inserted into an empty position in the array. The position p in the array is determined by the exponent field e and least-significant bit b of t using the formula $p = 2 \cdot e + b$. For example, the term $t_h = 2.0$ (with $e_h = 1024$ and $b_h = 0$) is inserted to position $p = 2048$.
3) If the array is not empty in the specified position, the elements are added together in double precision and the result is inserted to its corresponding new position in the array. For our example, assuming that the array already contains the value $A[p] = 3.0$ at position $p = 2048$, the new term to accumulate is $t_{new} = t_h + A[p] = 5.0$. Afterwards, the array field is cleared, $A[p] \leftarrow 0$, and we repeat the process for t_{new} as the new term to accumulate in the array.

Since the result of adding two normal floating-point numbers with the same exponent and same least-significant bit is exact assuming no overflow occurs, this method eliminates roundoff errors when adding elements to the accumulator. Once all elements are added, a final reduction is done by summing up its elements in double-double precision.

VII. Evaluation

In this section, we evaluate the performance and accuracy of IGen on four applications from the domains of signal processing, linear algebra and machine learning. Further, we evaluate the improvement in accuracy of the transformation for reductions. Finally, we compare IGen with the (manual) use of affine arithmetic.

Benchmarks. Table IV shows the computations used in the evaluation. The initial non-interval code is automatically generated by high-performance generators, except for the *ffnn* benchmark which is a loop-based implementation of a fully-connected feedforward neural network with nine hidden layers and n neurons per layer. The network is already trained to distinguish handwritten digits using the MNIST dataset [40]. For each computation, there is a scalar and a SIMD-optimized non-interval implementation available. We use IGen to generate equivalent interval code for both using scalar and SIMD-optimized output with double and double-double precision. We compare against manually using three common interval libraries: Boost v16.5 [18], Gaol v4.2 [20], and Filib++ v3.0 [19]. Gaol is the only library with SIMD optimizations for scalar interval operations using SSE registers to store double precision intervals. Only the scalar code of the benchmarks is manually implemented with the libraries since they only support interval implementations of scalar operations. All libraries are configured with their fastest sound configuration (i.e., the rounding mode is not constantly changed as explained in Section II).

Experimental setup. We perform the tests on an Intel Xeon E-2176M CPU (Coffee Lake microarchitecture) running at 2.7 GHz, under Ubuntu 18.04 with Linux kernel v4.15. Turbo Boost is disabled. All tests are compiled using gcc 7.5 with flags -O3 -march=host. To evaluate performance, every measurement was repeated 30 times on different inputs, and the median of the runtime is taken. All tests are run with warm cache. The inputs for all benchmarks are random intervals except for *ffnn* where the MNIST dataset [40] is used. Each input interval has a length of 1 ulp. In particular, for double-double precision, the length of an input interval is ulp(x_ℓ), where x_ℓ is the lower term of a random double-double number x. We measure the number of correct bits by subtracting the loss of accuracy from the number of bits used by the given precision (i.e., 53 and 106 bits for double and double-double precision respectively). The loss of accuracy is defined as the base-2 logarithm of the number of double precision floating-point values contained in an interval. Intuitively, the accuracy of an interval is the number of most-significant bits shared by the mantissas of the endpoints assuming same exponent[2].

TABLE IV
Benchmarks.

Label	Description	Base implementation
fft	Fast Fourier transform	Spiral [41]
potrf	Cholesky decomposition	SLinGen [42]
gemm	Matrix-matrix multiplication	ATLAS [43]
ffnn	Feedforward neural network	Straightforward

A. Performance and Accuracy

Plot navigation. Fig. 8 shows the performance results of using IGen and the libraries. The problem size is on the x-axis and the interval operations (iops) per cycle on the y-axis. Here, an interval multiplication and an addition count as one operation each. The inputs of the *gemm* and *potrf* benchmarks are square $n \times n$ matrices and the input for the *fft* benchmark is a vector with n complex values. The problem size in the *ffnn* benchmark represents the number of neurons per layer.

For each benchmark, IGen generates interval code with the following configurations:

- IGen-ss: scalar code generated from scalar input code.
- IGen-sv: vectorized code (SSE) from scalar input code.
- IGen-vv: vectorized code (AVX) from vectorized code.
- IGen-sv-dd: as IGen-sv but with double-double output.
- IGen-vv-dd: as IGen-vv but with double-double output.

Speedup. The results in Fig. 8 show that IGen-vv and IGen-sv are the fastest configurations across all the benchmarks. Further, already IGen-ss is around $2\times$ faster than the libraries in the *ffnn* and *gemm* benchmarks and has similar performance in the other two cases. In particular, IGen-vv benefits from the use of SIMD intrinsics in the input, which IGen converts to efficient interval operations. IGen-vv and IGen-sv are up to $9.8\times$ and $4.4\times$ faster than using the best library.

Note that the implementation of interval multiplication in libraries specialize to the sign of the operands. This seems to make them particularly sensitive to branch misprediction, leading to low performance in some benchmarks (e.g., in

[2]We consider the maximum accuracy of a double precision interval to be 53 bits and it is achieved when both endpoints are the same. In this case, the loss is zero.

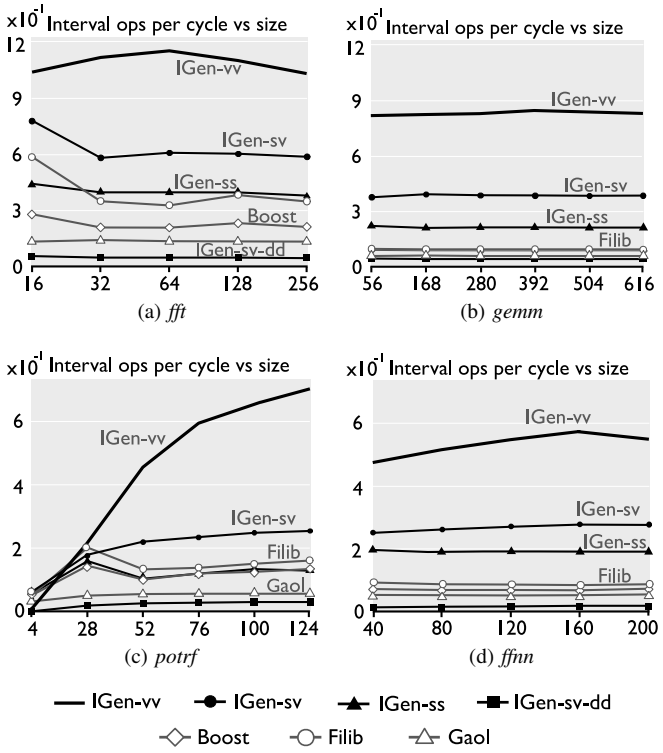

Fig. 8. Interval operations per cycle of IGen vs libraries.

Fig. 8d). On the other hand, their performance may improve in certain cases, for example Filib outperforms IGen-ss in the *potrf* benchmark. In addition, note that Gaol is a precompiled library. Thus, the compiler may be unable to inline the interval operations, preventing further optimizations. This may explain its lower performance.

Penalty of interval code. Table V shows the slowdown of different configurations of IGen as the ratio between the runtime of the non-interval input in double precision and interval output programs in double and double-double precision. All configurations generate vectorized interval code. The interval code with double precision is between 2.3×–13× slower. When using double-double, the code is 19×–280× slower. Note that the slowdown for double-double is significantly higher when using vectorized code as input (vv) compared to scalar code (sv). The runtime is actually similar for both, but the input program of the former is highly optimized. The reason is that we rely on the slower but automatic approach discussed in Section V to support SIMD intrinsics with double-double. Manually optimizing the double-double intrinsics to take the SIMD structure in the input into account will likely improve runtime and it is a subject of future work.

Efficiency. Fig 9a shows the real performance of IGen-vv and its non-interval input program. Since we do not use FMAs in the implementation, the theoretical peak performance of the interval code in the CPU is 8 flops per cycle (non-interval code can achieve higher performance). Thus, Fig 9a shows that double precision intervals achieve an efficiency of 53%–85%

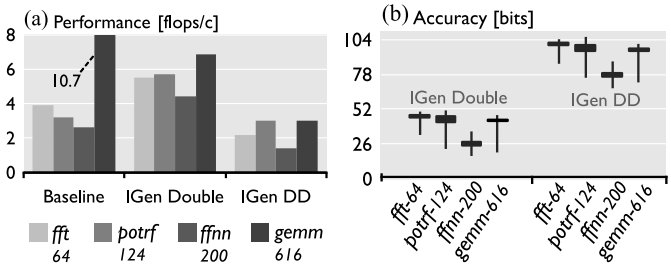

Fig. 9. (a) Real performance in flops per cycle of IGen-vv and its non-interval input program (baseline). (b) Accuracy of IGen in double and double-double precision.

TABLE V
SLOWDOWN OF DIFFERENT CONFIGURATIONS OF IGEN.

Name	IGen Dbl		IGen DD	
	sv	vv	sv	vv
fft-64	2.3	3.4	29	106
potrf-124	2.4	4.5	19	83
ffnn-200	6.1	4.8	99	147
gemm-616	5.6	13	62	279

of CPU utilization. This efficiency is in most cases higher than the non-interval baseline due to the increase in floating-point operations and thus higher operational intensity. The double-double implementation is less efficient since it is not fully optimized for SIMD intrinsics in the input. In addition, the implementation requires many shuffle operations (see Table III).

Certified accuracy. Fig. 9b shows the accuracy of IGen when using double and double-double precision. As can be seen, IGen certifies more than 17 bits in all the benchmarks when using double precision and more than 68 bits when using double-double. Further, using double-double improves the accuracy of all benchmarks by roughly 51 bits. Note that there is no noticeable difference in accuracy between IGen and the libraries when using double precision.

Certified double precision result. In case the inputs are exact double precision values or double-double values with small errors, using double-double can keep error accumulation small enough to compute certified double precision results, i.e., results with at most one bit of error in double precision. For example, the accuracy of all benchmarks in Fig. 9b with double-double precision is at least 68 bits, which is more than enough to round the resulting intervals to the nearest certified double precision value.

B. Accuracy of Reductions

We evaluate the improvement in accuracy of our reduction transformation approach using a double-loop matrix-vector multiplication (mvm) computation $y = Ax + y$, where x, y are of length n and m, respectively, and A is $m \times n$. IGen generates interval versions of the program in double and double-double precision with and without reduction transformations. We fixed $m = 10$ and vary n in the experiments. The matrix and vectors are initialized with values whose magnitudes are

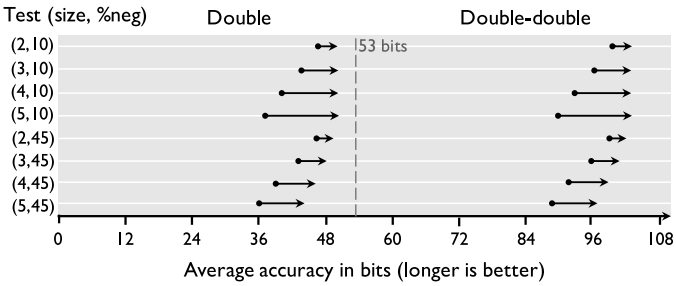

Fig. 10. Each arrow shows an improvement in accuracy of the mvm program. Test (s, p) means an input vector of size $n = 10^s$, where $p\%$ of the inputs are negative numbers. Arrows start from the accuracy of the interval program without reduction improvement and point to the generated accuracy with reduction improvement.

drawn randomly from the set of double precision numbers. We show two set of experiments: one where 10% of the input values are negative and another where 45% are negative[3].

Accuracy improvement. Fig. 10 shows the gain in accuracy when using reduction transformations in the mvm benchmark. The results show an improvement between 3 and 13 bits depending on the size of the reduction. Using the reduction transformation keeps the accuracy high and roughly constant for different sizes when testing with 10% of negative numbers in the input. On the other hand, the code without reduction transformation degrades when increasing the input size.

Balancing the amount of negative and positive numbers in the reduction increases the relative error in the final result. This explains the lower accuracy for the last four cases in Fig. 10. However, we still get an improvement of 8 bits for the largest test when transforming reductions.

Runtime. Without reduction transformation, the interval code runs 1.6× and 14.2× slower than the non-interval input when using double and double-double precision, respectively. With reduction transformations, the code runs 8.2× and 77.6× slower, respectively. The slowdown is independent of the size n.

ATLAS gemm. The previous *gemm* benchmark also contains reductions in the innermost loop of its matrix multiplication kernel. Unfortunately, Polly does not detect these reductions since ATLAS unrolls the loop. By manually converting to a loop, IGen is able to apply the reduction transformation and improves the accuracy by 3.5 bits on average at the price of a 3× slowdown for $n = 616$.

C. Comparison with Affine Arithmetic

Dependency problem. The dependency problem [22] is well known in interval arithmetic. It arises when variables become correlated during computation. Since interval arithmetic does not maintain relationships between variables, overapproximation may occur. This can be mitigated using affine arithmetic [32] which preserves the linear correlation between variables.

Benchmark. We compare IGen against the use of affine arithmetic on the Henon map, which is known to produce

[3]We did not choose 50% to avoid that the result of the reduction is approximately zero in expectation.

```
double henon_map(double x, double y, int iterations) {
  double a = 1.05;
  double b = 0.3;
  for (int i = 0; i < iterations; i++) {
    double xi = x;
    double yi = y;
    x = 1 - a*xi*xi + yi;
    y = b*xi;
  }
  return x;
}
```

Fig. 11. Input code for the Henon map.

significant losses in the accuracy of intervals due to dependency [44]. For affine arithmetic we use the YalAA v0.92 library [45] . The Henon map is an iterative algorithm defined as $x_{i+1} = 1 - ax_i^2 + y_i$ and $y_{i+1} = bx_i$. We use $a = 1.05$ and $b = 0.3$ in our experiments, and the initial condition $x_0 = y_0 = 0$. Figure 11 shows the input scalar code for the Henon map. In addition, we evaluate the use of affine arithmetic in our *fft* benchmark from Table IV[4]. In both benchmarks, we use the IGen-sv configuration to generate SIMD-optimized interval code from the scalar input.

Table VI shows the certified accuracy in bits and the slowdown for different number of iterations in the Henon map and different input sizes for the *fft* benchmark. The slowdown is the ratio between the runtime of the interval or affine configuration and the non-interval input in double precision. The accuracy is determined as the average of the minimum number of certified bits across 100 runs of the same configuration.

Accuracy. In the Henon map, the accuracy of IGen-generated code using double and double-double precision decreases in every iteration whereas the accuracy in affine arithmetic stays roughly constant. In contrast to double precision, double-double still preserves correct bits even after 170 iterations. For the *fft* benchmark, the accuracy achieved by double precision intervals and affine arithmetic is roughly the same and double-double outperforms both. We conclude that affine arithmetic is considerable more accurate than intervals in computations with strong dependencies but it achieves similar accuracy than double precision when dependencies are not an issue.

Slowdown. Table VI also shows that affine arithmetic is considerable more expensive than both, double and double-double interval. In particular, affine arithmetic is 2–3 orders of magnitude slower than double-double intervals on the two benchmarks.

VIII. Related Work

In this section, we review the existing related work.

Interval libraries. Most of the available interval arithmetic libraries [18]–[20] are implemented in C++ overloading basic operations with equivalent interval operations. Thus, making

[4]We were unable to evaluate the accuracy of affine arithmetic on the other benchmarks in Table IV since it did not pass the time limit of 1 hour that we imposed for the completion of each benchmark.

TABLE VI
CERTIFIED ACCURACY AND SLOWDOWN OF HENON MAP AND FFT BENCHMARKS.

Henon Map						
	Accuracy [bits]			Slowdown		
Iterations	`f64i`	`ddi`	Aff.	`f64i`	`ddi`	Aff.
10	44	96	44	2.8	14.1	795
50	24	76	45	2.9	14.8	4.4K
90	4	57	45	2.9	14.7	7.6K
130	0	37	44	2.8	14.8	10.6K
170	0	17	44	2.8	14.8	13.7K
FFT						
	Accuracy [bits]			Slowdown		
Size	`f64i`	`ddi`	Aff.	`f64i`	`ddi`	Aff.
16	43	96	44	2.3	32.6	3.4K
32	41	94	42	2.3	28.1	3.8K
64	39	92	40	2.3	28.9	4.5K
128	38	91	38	2.4	29.8	12.1K
256	36	89	36	2.4	29.7	28.4K

them easy to use. They are also relatively efficient since the endpoints of intervals are implemented using single or double precision floating-point. In addition, other libraries [21], [46] implement intervals with arbitrary large precision but are computationally very expensive. Finally, there are efficient libraries [33], [47] implementing double-double arithmetic; however, they do not provide interval operations. Only the work in [48] supports intervals in double-double precision but the implementation is not vectorized.

Although very useful, there are some disadvantage of using libraries to guarantee soundness instead of the compiler that we propose. First, it requires manual adaption by the user. Second, constants may be unsound. Third, they do not support SIMD intrinsics. Finally, interval libraries cannot automatically improve accuracy. It is up to the user of the library to handle these cases manually.

Automatic roundoff error analysis. There are many tools that statically analyze programs to derive worst-case roundoff errors. These tools normally use abstract interpretation using intervals or affine arithmetic to compute the ranges of variables as well as its error bounds [12]–[15]. Other tools are [16], [17] which formulate finding bounds of roundoff errors as an optimization problem. The former uses semi-definite programming whereas the latter uses symbolic Taylor expansions to approximate floating-point expressions and obtains error bounds using global optimization.

Since these tools have to consider all possible inputs, they sometimes provide bounds that are too conservative to be useful, especially when dealing with conditional branches and loops [13], [49]. We believe that some of the techniques in these tools can be applied to our interval compiler. For example, we may support affine arithmetic or a more expressive abstract domain to reduce interval overestimation in programs where the dependency problem occurs.

Program transformations for accuracy. One of the first approaches to improve or guarantee accuracy is mixed-precision tuning, which selectively changes the precision of floating-point variables to meet a certain error specification. Tools implementing this approach [9], [50]–[55] normally use roundoff error analysis to lead the search for a suitable precision.

In the field of code rewriting for accuracy, Salsa [56] and Herbie [57] have emerged. After identifying an expression in a program, Salsa improves its accuracy by building a set of equivalent expressions and choosing the one with the lowest error. Herbie, on the other hand, uses a guided search to select suitable transformations from a predefined database. IGen can benefit directly from these tools to improve the accuracy of the intervals. However, we focused on the transformation of reductions, which is a common pattern in linear algebra.

IX. CONCLUSIONS

In this paper, we presented an automatic approach to provide sound guarantees on the result of floating-point computations. We showed the design and implementation of IGen, a source-to-source compiler that automatically transforms a C program with floating-point computations to an equivalent and sound program using interval arithmetic. Our compiler supports Intel SIMD intrinsics in the input which are generated automatically from the vendor XML specification and enables fast updates in the future. When using double precision, our benchmarks demonstrate that IGen generates sound and efficient code which is $2.3\times - 12\times$ slower compared to the original unsound program, and it is in general faster than using interval libraries. Further, we showed that increasing the precision of intervals to double-double can keep error accumulation small enough to compute certified results with at most one bit of error in double precision in our set of benchmarks. The use of double-double, however, leads to a slowdown of one to two orders of magnitude compared to the original program, especially when transforming high-performance code. Finally, we showed that we can recover up to 13 bits lost due to rounding errors using a transformation technique to improve the accuracy of the common reduction pattern in linear-algebra-like computations.

ARTIFACT APPENDIX

A. Abstract

Our artifact provides the source code of IGen interval compiler, benchmarks to evaluate its performance and accuracy, and scripts for reproducing main experiments. The artifact is in the form of a virtual machine running Ubuntu 18.04 which provides all required dependencies.

More specifically, our artifact consists of:

1) Full source code of IGen interval compiler and library.
2) Source code for all of our benchmarks.
3) Scripts to setup and automate running the benchmarks saving the results in CSV files.
4) Scripts to generate graphs from the CSV files.

In addition, the artifact also contains the source code of the LLVM Project 11.0 with custom modifications to enable both, language extensions and specific pragmas used by IGen.

B. Artifact Check-List (Meta-Information)

- **Program:** IGen compiler with benchmarks (full source code).
- **Compilation:** We have included a script that builds IGen and associated benchmarks using GCC 7.5.
- **Binary:** A modified version of Clang 11.0 is precompiled to allow language extensions used by our interval compiler.
- **Run-time environment:** An Ubuntu-based virtual machine with all necessary software dependencies.
- **Hardware:** x86 machine supporting AVX2.
- **Execution:** We provide scripts to set up, run and plot the benchmarks in this paper. A more detailed description of how to use them is included in README.
- **Output:** Running the scripts yields CSV files containing performance numbers and accuracy of the benchmarks. Optionally, the performance plots in Fig. 8 can also be generated.
- **How much disk space required (approximately)?:** 20 GB to support the virtual machine.
- **How much time is needed to prepare workflow?:** Immediately available after importing the virtual machine in VirtualBox.
- **How much time is needed to complete experiments?:** Approximately 1 hour to setup, compile and run all benchmarks.
- **Publicly available?:** Yes.
- **Code license:** BSD 3-Clause License

C. Description

How delivered. This artifact is provided as a virtual machine and available at
`https://doi.org/10.5281/zenodo.4283110`.

Hardware dependencies. x86 machine with AVX2 support. We recommend testing on an Intel Skylake or similar microarchitecture to get comparable results with the ones presented in this paper.

Software dependencies. All software dependencies have been pre-installed in the provided virtual machine. We tested the artifact in VirtualBox 6.1.0. Details on the dependencies pre-installed in the virtual machine can be found in the `README` distributed with the artifact.

D. Installation

Import and access the virtual machine in VirtualBox[5]. The login credentials are the following:

username: cgo2021
password: igen@cgo21

Since all dependencies have been pre-installed, the system should be ready once accessing the virtual machine.

E. Experiment Workflow

Once in the virtual machine, open the terminal and navigate to the `benchmarks` directory:

```
$ cd artifact/benchmarks
```

There is a script named `run_benchmarks.py` which sets up, builds and runs all benchmarks. You can run it as follows:

```
$ python3 run_benchmarks.py -all
```

To run individual benchmarks consult the `--help` option.

[5]More information on importing virtual machine in VirtualBox can be found at `https://docs.oracle.com/cd/E26217_01/E26796/html/qs-import-vm.html`

F. Evaluation and Expected Result

After the experiments finish running, the generated CSV files with the results are saved in `artifact/benchmarks/results` directory. There is one folder for each benchmark (e.g. `gemm`, `ffnn`, `potrf` etc.). For comparison, the numbers presented in the paper are also included in `results/paper`.

The results used to generate Figures 8, 9a, 9b and Table V are saved in the following files respectively:

- `<benchmark>/interval_perf.csv`
- `<benchmark>/real_perf_<size>.csv`
- `<benchmark>/accuracy_<size>.csv`
- `<benchmark>/overhead_<size>.csv`

To generate the performance graphs in Fig. 8 use the following command:

```
$ python3 run_benchmarks.py -plot
```

The graphs will be saved as PDF files in the `results` folder.

G. Notes

We recommend disabling Intel Turbo Boost and Hyper Threading technologies in the host machine to avoid the effects of frequency scaling and resource sharing on the measurements. These technologies can be disabled in the BIOS settings of the machines that have BIOS firmware.

H. Methodology

Information regarding submission, reviewing and badging methodology can be found at the following sites:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] ARM. (2019) Arm technologies: Helium. https://www.arm.com/why-arm/technologies/helium. [Online; accessed 4-August-2020].

[2] D. N. Arnold. (2000) The patriot missile failure. http://www-users.math.umn.edu/~arnold//disasters/patriot.html.

[3] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *Formal Methods for Industrial Critical Systems (FMICS)*, 2009, pp. 53–69.

[4] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso, "High-Assurance SPIRAL: End-to-end guarantees for robot and car control," *IEEE Control Systems Magazine*, vol. 37, no. 2, pp. 82–103, 2017.

[5] D. Cattaruzza, A. Abate, P. Schrammel, and D. Kroening, "Sound numerical computations in abstract acceleration," in *Numerical Software Verification (NSV)*, 2017, pp. 38–60.

[6] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008, pp. 3–18.

[7] K. Jia and M. Rinard, "Exploiting verified neural networks via floating point numerical error," *CoRR*, vol. abs/2003.03021, 2020.

[8] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 3, 2019.

[9] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[10] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - framework for analysis and optimization of numerical programs (tool paper)," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018, pp. 270–287.

[11] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Symposium on Principles of Programming Languages (POPL)*, 1978, pp. 84–96.

[12] E. Goubault and S. Putot, "Static analysis of finite precision computations," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011, pp. 232–247.

[13] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 39, no. 2, 2017.

[14] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 1, 2010.

[15] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A unified coq framework for verifying c programs with floating-point computations," in *Certified Programs and Proofs (CPP)*, 2016, pp. 15–26.

[16] V. Magron, G. Constantinides, and A. Donaldson, "Certified roundoff error bounds using semidefinite programming," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, 2017.

[17] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, 2018.

[18] H. Brönnimann, G. Melquiond, and S. Pion, "The design of the Boost interval arithmetic library," *Theoretical Computer Science*, vol. 351, no. 1, pp. 111–118, 2006.

[19] M. Lerch, G. Tischler, J. W. V. Gudenberg, W. Hofschuster, and W. Krämer, "FILIB++, a Fast Interval Library Supporting Containment Computations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 2, pp. 299–324, 2006.

[20] F. Goualard, "Gaol 4.2.0: Not just another interval arithmetic library," https://sourceforge.net/projects/gaol, 2015.

[21] N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the mpfi library," *Reliable Computing*, vol. 11, no. 4, pp. 275–290, 2005.

[22] R. E. Moore, "Interval analysis," *Prentice-Hall*, 1966.

[23] F. Goualard, "Towards Good C++ Interval Libraries: Tricks AND Traits," 2000. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00430568

[24] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[25] Clang. (2020) Clang libtooling. Available at https://clang.llvm.org/docs/LibTooling.html, version 11.0.0.

[26] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[27] "IEEE Standard for Interval Arithmetic," *IEEE Std 1788-2015*, pp. 1–97, 2015.

[28] F. Goualard, "Fast and Correct SIMD Algorithms for Interval Arithmetic," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[29] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Lauter, and J.-M. Muller, "CR-LIBM A library of correctly rounded elementary functions in double-precision," Research Report, 2006. [Online]. Available: https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804

[30] Intel, "Intel Intrinsics Guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2012, [Online; accessed 4-August-2020].

[31] A. Stojanov, I. Toskov, T. Rompf, and M. Püschel, "SIMD Intrinsics on Managed Language Runtimes," in *International Symposium on Code Generation and Optimization (CGO)*, 2018, pp. 2–15.

[32] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, 2004.

[33] Y. Hida, S. Li, and D. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *IEEE Symposium on Computer Arithmetic (ARITH16)*, 2001, pp. 155–162.

[35] M. Joldes, J.-M. Muller, and V. Popescu, "Tight and rigorous error bounds for basic building blocks of double-word arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, 2017.

[34] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser Basel, 2018.

[36] S. Graillat and F. Jézéquel, "Tight interval inclusions with compensated algorithms," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1774–1783, 2019.

[37] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's polyhedral scheduling in the presence of reductions," in *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2015.

[38] M. A. Malcolm, "On accurate floating-point summation," *Communications of the ACM*, vol. 14, no. 11, pp. 731–736, 1971.

[39] J. Demmel and Y. Hida, "Fast and accurate floating point summation with application to computational geometry," in *Numerical Algorithms*, 2002, pp. 101–112.

[40] Y. LeCun, C. Cortes, and C. Burges. (2010) Mnist handwritten digit database. Available: http://yann.lecun.com/exdb/mnist, 2010.

[41] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[42] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel, "Program generation for small-scale linear algebra applications," in *International Symposium on Code Generation and Optimization (CGO*, 2018, pp. 327–339.

[43] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *ACM/IEEE Conference on Supercomputing (SC)*, 1998, pp. 38–38.

[44] M. Kashiwagi, "An algorithm to reduce the number of dummy variables in affine arithmetic." in *In SCAN conference*, 2012.

[45] S. Kiel, "Yalaa: Yet another library for affine arithmetic," *Reliable Computing*, vol. 16, pp. 114–129, 2012.

[46] F. Johansson, "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic," *IEEE Transactions on Computers*, vol. 66, pp. 1281–1292, 2017.

[47] K. Briggs. (1998) The doubledouble library. Available at http://www.boutell.com/fracster-src/doubledouble/doubledouble.html.

[48] M. Kashiwagi. kv - a c++ library for verified numerical computation. Available at http://verifiedby.me/kv/index-e.html.

[49] E. Goubault and S. Putot, "Robustness analysis of finite precision implementations," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013, pp. 50–57.

[50] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *International Conference on Cyber-Physical Systems (ICCPS)*, 2018, p. 208–219.

[51] S. Graillat, F. Jézéquel, R. Picot, F. cois Févotte, and B. Lathuiliére, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *Journal of Computational Science*, vol. 36, p. 101017, 2019.

[52] J. Vignes, "Discrete stochastic arithmetic for validating results of numerical software," *Numerical Algorithms*, vol. 37, no. 1, pp. 377–390, 2004.

[53] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamariundefined, "Rigorous floating-point mixed-precision tuning," in *Symposium on Principles of Programming Languages (POPL)*, 2017, pp. 300–315.

[54] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *International Conference on Supercomputing (ICS)*, 2013, pp. 369–378.

[55] N. Damouche and M. Martel, "Mixed precision tuning with Salsa," in *International Joint Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*, 2018, p. 185–194.

[56] N. Damouche, M. Martel, and A. Chapoutot, "Improving the numerical accuracy of programs by automatic transformation," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 4, pp. 427–448, 2017.

[57] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Conference on Programming Language Design and Implementation (PLDI)*, 2015, pp. 1–11.