VP Float: First Class Treatment for Variable Precision Floating Point Arithmetic

Tiago Trevisan Jost, Yves Durand, Christian Fabre, Albert Cohen, Frédéric Pétrot

Seamless Compiler Integration of Variable Precision Floating-Point Arithmetic

Tiago Trevisan Jost, Yves Durand, Christian Fabre
Univ. Grenoble Alpes
CEA LIST
Grenoble, France
tiago.trevisanjost@cea.fr, yves.durand@cea.fr
christian.fabre1@cea.fr

Albert Cohen
Google
Paris, France
albertcohen@google.com

Frédéric Pétrot
Univ. Grenoble Alpes, CNRS
Grenoble INP, TIMA
Grenoble, France
frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—Floating-Point (FP) units in processors are generally limited to supporting a subset of formats defined by the IEEE 754 standard. As a result, high-efficiency languages and optimizing compilers for high-performance computing only support IEEE standard types and applications needing higher precision involve cumbersome memory management and calls to external libraries, resulting in code bloat and making the intent of the program unclear. We present an extension of the C type system that can represent generic FP operations and formats, supporting both static precision and dynamically variable precision. We design and implement a compilation flow bridging the abstraction gap between this type system and low-level FP instructions or software libraries. The effectiveness of our solution is demonstrated through an LLVM-based implementation, leveraging aggressive optimizations in LLVM including the Polly loop nest optimizer, which targets two backend code generators: one for the ISA of a variable precision FP arithmetic coprocessor, and one for the MPFR multi-precision floating-point library. Our optimizing compilation flow targeting MPFR outperforms the Boost programming interface for the MPFR library by a factor of $1.80\times$ and $1.67\times$ in sequential execution of the PolyBench and RAJAPerf suites, respectively, and by a factor of $7.62\times$ on an 8-core (and 16-thread) machine for RAJAPerf in OpenMP.**

***Index Terms*—Floating-point arithmetic, compiler optimization, LLVM, MPFR, UNUM.**

I. Introduction

The standardization of the IEEE 754 floating-point (FP) format in 1985 [1] and progress with VLSI ever since led to generalized usage of hardware computing units for FP. These units are essential to obtain good performance from numerical applications, as long as compilers use them efficiently. However, an increasing number of applications are better served by more suitable FP formats. Linear solvers, *n*-body problems [2], and other applications in mathematics and physics [3] have shown to benefit from higher-than-standard representations since (1) they may not converge with fewer bits of precision or (2) they can converge faster with higher precision [4], [5].

A finer control of exponent and precision sizes allows for finding the right trade-off between the accuracy of the results and the time taken and/or energy consumed for the computations. Table I illustrates how precision impacts the results of some benchmarks selected from the PolyBench suite [6], a popular benchmark for evaluating compiler optimizations on numerical computations. One notices gains in resulting accuracy

TABLE I
Residual error for some PolyBench applications

		Mini Dataset	Small Dataset	Medium Dataset	Large Dataset	Xlarge Dataset
gemm	IEEE 32	1.5e-5	2.1e-4	4.1e-3	2.3e-1	1.45e0
	IEEE 64	3.1e-14	4.0e-13	7.7e-12	4.33e-10	2.69e-9
	128 bits*	< 1e-600	< 1e-600	< 1e-600	1.49e-34	2.6e-33
	512 bits*	< 1e-600	< 1e-600	< 1e-600	< 1e-600	< 1e-600
3mm	IEEE 32	6.7e-07	1.1e-04	3.1e-02	4.4e+01	998.4
	IEEE 64	1.3e-15	2.1e-13	5.8e-11	8.2e-08	1.8e-06
	128 bits*	3.5e-38	5.6e-36	1.5e-33	2.13e-30	4.8e-29
	512 bits*	< 1e-600	< 1e-600	< 1e-600	< 1e-600	< 1e-600
covar	IEEE 32	5.8e-5	5.6e-3	2.1e-1	41.02	5.7e+02
	IEEE 64	1.2e-13	2.5e-12	2.37e-10	7.2e-8	1.0e-06
	128 bits*	3.2e-36	6.6e-35	6.3e-33	1.9e-30	2.6e-29
	512 bits*	9.1e-152	1.8e-150	1.6e-148	4.8e-146	6.7e-145
gram	IEEE 32	28	71	220	616	868
	IEEE 64	9.1	76	231	584	849
	128 bits*	1.1e-21	7.0e-21	3.5e-20	1.7e-19	3.6e-6
	512 bits*	4.6e-137	2.1e-136	7.5e-136	1.1e-134	1.3e-121

*bits of mantissa. IEEE 32 has 24 bits of mantissa. IEEE 64 has 53.

significantly higher than the number of extra bits of precision in the intermediate computation. Some kernels are also actually numerically unstable for the 32 and 64 IEEE data types, even with small datasets, while higher precision reaches stability (e.g. *gramschmidt*). If one strives for accuracy, it is paramount that higher-than-standard precision be adopted. Practically, high precision is only supported through software libraries [7], [8], and high-efficiency languages in high-performance computing do not provide any higher level abstraction. This leads to tedious, error-prone and library-dependent implementations involving explicit memory management. Multi-precision code is difficult to write and maintain, and more than the performance gap of a software FP implementation, the productivity gap makes this approach unaccessible to potential users.

We address this challenge through a generic type system and language extension of standard C for variable precision FP formats. We provide an intuitive syntax for this type system, suiting the expressiveness needs of numerical analysis while enabling highly efficient in-place execution, stack allocation and compiler optimizations expected for the C language. This is made possible by extending an industry-standard compiler intermediate representation (IR), allowing classical compiler optimizations to operate on multi-precision FP types. We enable multiple FP formats to coexist in a single numerical kernel, with full procedural abstraction and application binary

interface (ABI) compatibility. For each format that supports it, we also enable computations over multiple variables of different, possibly dynamically varying, precision and memory footprints. This provides the ability to explore multiple numerical configurations within a single program run, without code duplication or even without recompilation, as well as dynamically varying precision to adjust at runtime to the propagation of error or numerical instabilities. We demonstrate our approach with two backend code generators targeting (1) a coprocessor implementing native UNUM instructions [9] and (2) the MPFR [7] reference library for arbitrary precision FP arithmetic. In summary, our contributions are the following:

1) a C type-system extension for declaring FP numbers of arbitrary representation and size. This generic class of FP types has attributes, such as the size of mantissa or exponent, or the size of the field encoding the mantissa or exponent, and overall memory footprint. These may be known statically or only at runtime,
2) an IR embedding of the runtime and compile time aspects of generic FP types. Thanks to a tight integration within the LLVM infrastructure [10], this embedding allows to benefit from most existing compiler optimizations supporting high-performance numerical computing,
3) while most of this is transparent to optimization passes, some key optimizations are modified and extended to offer notable performance benefits compared to a higher-level abstraction in C++ with no specific compiler support [11],
4) support for multi-threaded and parallel programming in synergy with LLVM's code generator and runtime for OpenMP [12],
5) two backend code generators: (1) a backend for the UNUM instruction set architecture (ISA) proposed by [9]. The runtime-configurable attributes of our proposed generic types makes it a good fit for the UNUM format. All optimization passes, including the lower level register allocation and instruction selection, operate on variable precision UNUM values the same way as on primitive IEEE data types. (2) a backend for the MPFR library [7] with a transformation pass that lowers operations on values of the generic type system into MPFR calls. The pass automatically manages the allocation of MPFR objects, and enables classical optimizations to operate on MPFR calls as if they were plain IEEE arithmetic and performs in-place operation and stack allocation that would not be accessible to a numerical analyst with little experience in low level memory management.

The paper is organized as follows. Section II presents background material on higher precision FP arithmetic in numerical applications. Section III describes the proposed compilation flow for generic FP computations, covering aspects of the type system, IR, and target-specific backends, as well as their integration in the LLVM infrastructure and the related optimizations. Section IV evaluates the compilation flow on a number of applications, and shows how we are able to leverage optimization passes already included in the compiler. Section V discusses existing language and compiler support for variable precision arithmetic. Finally Section VI summarizes our proposal and findings.

II. Background

While the IEEE 754 standard met immense success across diverse areas, some applications require alternative representations to approximate real numbers. The prototypical example would be a linear algebra solver, computing a vector x solution of the matrix equation $Ax = b$. Among the algorithms to solve this ubiquitous problem in scientific computing, *direct solvers* such as Gaussian elimination or Cholesky benefit from extended precision arithmetic to reduce the residual error [4], [5]. More generally, higher precision arithmetic greatly simplifies the solution of ill-conditioned problems, making it possible to retain the well-known ''classical'' algorithms and avoiding the need for costly workarounds such as ad-hoc preconditioners. Extended precision is even more beneficial for iterative methods, which are preferred for large problems. It compensates the accumulation of roundoff error during iterations, which may slow down or even hinder convergence. As a result, increasing the precision of the intermediate residual vector is an effective means to reduce the number of iterations. It enables the solver to address *larger problems* than with lower precision arithmetic [4], [5].

This strategy is called ''mixed precision'' [13] when it uses `double`-precision FP numbers within a computational scheme dominated by single-precision `float`. However, when the required precision exceeds the possibilities of the `double` type, the numerical analyst may be left with no choice but to resort to a higher-precision approximation of real vectors. Of course, this approximation must still be parsimonious in computing and memory.

As a result, while a variable-precision algorithm may be superficially very similar to its standard model, it is also responsible for estimating and configuring its precision (the length of the fractional part) for part or all of the arithmetic operations, resulting from the problem characteristics such as its conditioning and required accuracy. This estimation is generally complex to compute, and may lead to unnecessary over-provisioning of fractional bits. A practical alternative is to adapt precision dynamically: instead of computing the necessary precision *a priori*, the modified kernel uses an outer loop to systematically check the result for accuracy at predefined points. If the residual is above a predefined threshold, or if convergence is too slow, the solver increases its internal precision and resumes the computation. This approach, called ''transprecision'', has recently motivated research initiatives [14]--[16] and [17]. Taking again the example of direct or iterative solvers, this adds strong requirements on the software engineering and tool flow supporting variable or transprecision computing:

- the kernel source code must be unique, therefore the programming style must be agnostic of the underlying precision of its variable precision data,

- the required precision depends on the conditioning of data: it may be defined at kernel initiation time or dynamically and gradually increased in the case of adaptive methods.

One may add a third, practical adoption requirement:

- a variable-precision implementation should be as similar as possible to its original reference model in C with standard IEEE arithmetic; in particular, extended precision should be used only when necessary, which implies that applications may smoothly transition between legacy support libraries (e.g., double-precision BLAS) and extended or adaptive precision solutions when required.

A programming environment meeting these requirements effectively makes variable precision an extension of mixed precision. This is the general motivation for our work.

III. Compilation Flow for Generic FP Operations

Programming languages struggle with alternative FP formats, missing on the essential data-flow, control-flow and algebraic optimizations available for IEEE-compatible arithmetic, and also missing opportunities to leverage hardware implementations for alternative FP arithmetic. The numerical analyst is left with the choice between a high-level managed language like Julia [18] whose abstractions and type systems provides a high-productivity variable-precision interface but fails to deliver competitive performance, and using an efficient language like C with hand-optimized memory management and calls to a software library such as MPFR [7], or sacrificing much precision control by relying on a mixed-precision paradigm based on IEEE-compatible formats.

We propose a type system extension for the C language and for the IRs of C compilers to provide first-class support to programming with mixed-precision FP arithmetic. It enables hardware support when available and offers the flexibility of numerical libraries that can operate with multiple precisions. The remainder of this section reviews the main aspects of the language and IR extensions, their syntax and semantics, and two backends demonstrating the retargetability and genericity of the approach.

A. Language Extension

Approximate representations of real numbers in which the sizes of mantissa and exponent may vary according to the user's needs require runtime capabilities that are not easily expressed with the semantics of the C primitive data types. Therefore, we propose an extended type system capable of manipulating FP operations with different representations and formats, along with memory management and calling conventions to support dynamic variations of precision and memory footprint compatible with typical ABIs. Our type system captures FP types with constant attributes, i.e., those known at compile time, as well as attributes known only at runtime. It differs from the C standard through the introduction of a parameterized type for multiple representations named `vpfloat`, borrowing the syntax of C++ `template`.

1) Syntax: The new primitive type `vpfloat` is parameterized with attributes to control a given FP implementation, such as its specific format, exponent, precision and/or size. The syntax aims at providing a generic way for different formats to coexist within the same generic type, with the possibility of adding new formats or representations as they are proposed:

```
vpfloat-declaration:
  vpfloat '<' vpfloat-attributes '>' declaration

vpfloat-attributes:
  type ',' exp-info ',' prec-info ',' size-info

type:
  unum | mpfr | posit | bfloat16 | ...

exp-info:
  integer-literal | identifier

prec-info:
  integer-literal | identifier

size-info:
  integer-literal | identifier
```

Every declaration must provide a *type* attribute which defines if subsequent attributes are needed and which information they carry. Attributes are specified in the following order: *type*, *exponent*, *precision*, and *size*. With the exception of *type*, `vpfloat` attributes can all be defined with integral constant literals or identifiers. This generic type supports constant-size and dynamically-sized types, which is always a technical hurdle in unmanaged languages like C and its associated IRs and ABIs.

2) Semantics: During semantic analysis, our compiler checks for declarations to guarantee that attributes are well-formed and respect the semantics required by the specific type attribute. We designed and implemented a full compilation flow supporting *unum* and *mpfr* types, with attributes interpreted accordingly.

```
typedef struct  {
  int  _mpfr_prec;
  int  _mpfr_sign;
  int  _mpfr_exp;
  int  *_mpfr_d;
} __mpfr_struct, *mpfr_ptr;
```

Listing 1. MPFR variable type as defined in [7]

Variables declared as *mpfr* hold the number of bits of exponent and mantissa in the second and third fields of the declaration, respectively. These values are used later to set up MPFR objects created during our MPFR backend transformation pass (see Section III-C1). An MPFR object is of type `__mpfr_struct` (see Listing 1). The MPFR API is particularly well suited to low-level code generation as the functions globally follow the pattern `mpfr_op(dest, src1[, src2, ...,][rounding mode])`, in which the dest and src parameters are `mpfr_ptr`. `Op` can be a basic operation ($+, -, \times, \div$), a fused operation (fma, fms), or one of many possible mathematical functions ($\sqrt[n]{}, \cos, \sin, \log, ...$). The parameters may have different exponent and precision sizes, and the destination parameter can be identical to a source parameter. They need (destination included) to be allocated

```
void axpy_mpfrconst(int N,
                    vpfloat<mpfr, 16, 256> alpha,
                    vpfloat<mpfr, 16, 256> *X,
                    vpfloat<mpfr, 16, 256> *Y) {
    for (unsigned i = 0; i < N; ++i)
        Y[i] = alpha * X[i] + Y[i];
}

void axpy_mpfr(unsigned prec, int N,
               vpfloat<mpfr, 16, prec> alpha,
               vpfloat<mpfr, 16, prec> *X,
               vpfloat<mpfr, 16, prec> *Y) {
    for (unsigned i = 0; i < N; ++i)
        Y[i] = alpha * X[i] + Y[i];
}

void axpy_unumconst(int N,
                    vpfloat<unum, 4, 6, 8> alpha,
                    vpfloat<unum, 4, 6, 8> *X,
                    vpfloat<unum, 4, 6, 8> *Y) {
  for (unsigned i = 0; i < N; ++i)
    Y[i] = alpha * X[i] + Y[i];
}

void gemm_unum(unsigned prec, int M, int N,
               double *A,
               vpfloat<unum, 4, prec> alpha,
               vpfloat<unum, 4, prec> *X,
               vpfloat<unum, 4, prec> beta,
               vpfloat<unum, 4, prec> *Y) {
  for (unsigned i = 0; i < M; ++i) {
    // From III.A.5 "Dynamically-sized Types":
    // alphaAX's dynamic size is computed by
    // __sizeof_vpfloat(4, prec).
    vpfloat<unum, 4, prec> alphaAX = 0.0;
    for (unsigned j = 0; j < N; ++j)
      alphaAX += A[i*N + j] * X[j];
    Y[i] = alpha * alphaAX;
    // Free memory of alphaAX back to stack.
  }
}
```

Listing 2. Sample BLAS functions reimplemented with *mpfr* and *unum* types

and have their precision defined with `mpfr_init` before being used and freed with `mpfr_clear` once useless. These functions have a high performance penalty and should thus be called wisely. The value of an MPFR variable is set using `mpfr_set`, which is useful in particular for spilling variables.

Listing 2 shows examples of the *axpy*, a level-1 BLAS [19] routine for vector multiplication, for different *mpfr* types. Function `axpy_mpfrconst` implements *axpy* with constant-size *mpfr* type of 256 bits of mantissa, and `axpy_mpfr` illustrates the support for dynamically-sized types, as the number of mantissa bits (the precision) is not known at compile-time. In this function, the number of mantissa bits will be the one provided by the caller, and a simple analysis of the dynamic attributes is implemented to finalize the type-checking of function calls at runtime.

The UNUM format, on the other hand, requires rethinking the semantics of attributes. As defined in the format specification, meta-data information *ess* and *fss* not only act as parameters bounding the number of bits of exponent and precision, but also contribute to defining the size of a UNUM value [20]. It should also be highlighted that the language syntax does not specify the number of bits of exponent and precision, respectively *exp-info* and *prec-info*: the UNUM format derives this information from *ess* and *fss*, i.e., the second field of a *unum* declaration holds the *size of exponent* and the third field holds the *size of mantissa*.

We provide a backend to generate code targeting a simplified version (interval arithmetic instructions are left aside) of the ISA of Bocco et al. [9], [21]. Our frontend semantically analyzes *unum* types and follows the requirements of the target ISA. In particular, values of *ess* and *fss* range between 1 and 4, and 1 and 9, respectively, which allows exponents between 1 and 16 bits and mantissas between 1 and 512 bits. Considering that *ess* and *fss* produce exponent and mantissa values that grow exponentially, *unum* types may be declared with an optional *size-info* attribute that holds the maximum number of bytes used to represent the number. This attribute value must be in the range 1..68.

Declarations with no *size-info* convey that sizes are calculated according to the values of *ess* and *fss* as $(2 + 2^{ess} + 2^{fss} + 7)/8$. The presence of *size-info* implicates the truncation of a declaration to a maximum of *size* bytes. Since the mantissa is the last field specified in the format, the *size* attribute may truncate bits of the mantissa. The formula used to calculate the number of mantissa bits in this case is given by $\min(2^{fss}, size * 8 - (2 + 2^{ess} + 2^{fss}))$. Table II illustrates different UNUM representations, showing the corresponding values of exponent, mantissa and total size.

TABLE II
SAMPLE UNUM DECLARATIONS AND THEIR RESPECTIVE EXPONENT, MANTISSA, AND SIZE VALUES

`vpfloat<unum,ess,fss>` or `vpfloat<unum,ess,fss,size>`	exponent (in bits)	precision (in bits)	size (in bytes)
`vpfloat<unum,3,6>`	8	64	11
`vpfloat<unum,3,6,6>`	8	29	6
`vpfloat<unum,3,8,60>`	8	256	60
`vpfloat<unum,4,9,20>`	16	129	20
`vpfloat<unum,4,9>`	16	512	68

The ability of our language extension to handle the flexibility of UNUM illustrates its generic nature and runtime capabilities. Listing 2 also shows the implementation of two functions: (1) *axpy* implemented using a constant-size *unum* type, and (2) the General Matrix Multiplication (GEMM) from BLAS implemented with a dynamically-sized type. There are no restrictions on using dynamically-sized types for *mpfr* or *unum* representations as long as compatible backends are provided. More specifically, backends are responsible for implementing the runtime capabilities, either through library calls (as is the case of `vpfloat<mpfr, ...>`), or through a compatible ISA (as for `vpfloat<unum, ...>` in our examples).

3) Type Comparison, Casting and Conversion: Types are only considered equal if they hold the exact same attributes. Also, our type system does not implement any form of sub-typing or implicit conversion, except for plain variable assignments. Implicit conversions over more general expressions would be too ambiguous when determining the types of intermediate values. If types are not equal, it is the user's

responsibility to insert the appropriate cast or type conversion. Casting exposes the underlying array of bytes implementing a given format, and vice versa.

Our toolchain enables the configuration of type sizes at byte granularity, in other words, we support non-power of two FP types sizes. In Section IV, experiments were executed with `vpfloat<unum,..>` of 25 and 67 bytes of size. There are many other properties of the language and the IR that only apply to constant- or dynamically-sized types. In the next sections, we survey their characteristics and differences, highlighting the differences between *mpfr* and *unum* types if they exist.

4) Constant-Size Types: Our language extension allows the declaration of constant-size types in the same fashion as standard primitives types. As the name indicates, constant-type variables are declared by only specifying attributes known at compile-time. They can be declared as global, local variables, and arguments, just like any constant-size variable in C.

Variables of a constant-size type can be initialized providing a FP literal. A *v* suffix is used to denote a literal of `vpfloat<unum, ...>` types, and a *y* suffix is used for `vpfloat<mpfr, ...>` types. They can also be initialized with a IEEE 754 FP literal, i.e., using `float` and `double` FP literals, however an implicit conversion is performed by the compiler in those cases which may incur loss of precision through rounding.

Table III represents the FP literal 1.3 for different `vpfloat` *unum* and *mpfr* types. Representations are in hexadecimal, with the *V* prefix for UNUM types and *Y* for MPFR types. Each format shows a different representation for the closest approximation of the same value. Values are displayed chunks of 64 bits such that the last chunk always contains the value of the sign bit and other fields, with the mantissa being the last field of the format. If the representation exceeds 64 bits, the remaining chunks contains the rest of the mantissa. Values are biased according to the maximum exponent value, similar to the IEEE formats.[1]

TABLE III
FLOATING-POINT (FP) LITERAL 1.3 REPRESENTED IN DIFFERENT TYPES

`vpfloat<unum, ...>` or `vpfloat<mpfr, ...>`	Representation of 1.3 (hexadecimal)
`vpfloat<unum,3,6,6>`	`0xV001FE999999A`
`vpfloat<unum,4,9,20>`	`0xV999999999999999A9999 9999999999990001FFFE`
`vpfloat<mpfr,8,48>`	`0xY0FF4CCCCCCCCCCD`
`vpfloat<mpfr,8,64>`	`0xY4CCCCCCCCCCCCCCD0FF`
`vpfloat<mpfr,16,100>`	`0xYCCCCCCCCCCCCCCC D0FFFF4CCCCCCCC`

5) Dynamically-Sized Types: One challenging feature of variable precision FP formats is the need to declare types whose memory footprint is not known until runtime evaluation. Such functionality is an important aspect of our extension, since it allows users to programmatically explore multiple configuration of exponent and mantissa in a single run.

The runtime memory management of dynamically-sized types is done in the same way as for Variable Length Arrays (VLA) in C [22], through ad-hoc generated code. A VLA is an array declaration where its number of elements is not known at compile time, and is, therefore, evaluated at runtime. VLAs shall only be declared as local variables and function parameters, and their lifetime extends from the declaration of the object until the program leaves their declaration scope. The compiler generates code that evaluates the size of the array at runtime and allocates it on the stack. Likewise, dynamic `vpfloat` types can only be declared as local variables and function parameters, and their life cycle follows the one of VLAs. Just as for VLAs, since it is not possible to guarantee that all attributes of a dynamically-sized `vpfloat` are know at the beginning of the function, they are stack-allocated within their declaration scopes.

Additionally, this aspect of the language adds extra complexity to dynamically-sized types as dynamic values for the attributes are not assured to respect the limits for each specific type. For instance, the C standard defines the allocation of a VLA with a negative size as *undefined behavior*. Adopting the same behavior for `vpfloat` declarations ease the role of the compiler as no verification is needed at runtime in order to ensure the programmer uses a valid expression. On the other hand, no guarantees would be given that the executed code will perform the correct computation if runtime attributes have not been checked for consistency.

We choose to err on the side of correctness, and compliance with the underlying numerical libraries when relying on them (such as MPFR), and implement runtime verification functions to ensure that all parameters and the size of each declaration respect the boundaries defined by the representation. Each dynamically-sized type declaration generates a call to `__sizeof_vpfloat`, a function from our runtime library that checks for consistency of attributes and returns the number of bytes needed for the specific type. Generating a call to this function ensures all attributes are well-defined and respect boundaries for the type. In Listing 2, variable `alphaAX` of function *gemm_unum* is allocated and freed in every iteration of the loop, with its allocation size given by calling `__sizeof_vpfloat`. A better solution would be to declare the variable outside of the loop, that way only one call to the function is required. However, the purpose of the example is not to show the optimal solution but to illustrate when our runtime library checks types and how memory management occurs. A call to the `__sizeof_vpfloat` function is generated for each **`sizeof`** computation of a dynamically-sized type, allowing to obtain the size of the type at runtime and providing support for memory allocation in general. Additionally, our compiler also generates verification calls for `vpfloat` parameters passed through a function call in order to guarantee that values passed as attributes still hold the same value upon creation.

[1]The 0s shown in UNUM formats are reserved for *ess* and *fss* values which are only properly set later in the compilation flow. Indeed, constants are created everywhere in the compilation flow, and these fields depend on the evaluation context. This behavior is specific to the UNUM format. Since it would be too intrusive to modify every LLVM pass, we added a dedicated finalization pass instead to properly set up all UNUM constant literals.

```
1  void example_dynamic_type(unsigned p) {
2
3    vpfloat<mpfr, 16, 200> a, X[10], Y[10];
4    // here initialize a, X and Y here
5
6    vpfloat<mpfr, 16, p> a_dyn;
7    vpfloat<mpfr, 16, p> X_dyn[10], Y_dyn[10];
8    // initialize a_dyn, X_dyn and Y_dyn here
9
10   vaxpy(100, 10, a, X, Y);  // ERROR
11   vaxpy(200, 10, a, X, Y);  // OK
12
13   // OK if p == 200
14   vaxpy(200, 10, a_dyn,  X_dyn, Y_dyn);
15   vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // OK
16   ++p;
17   vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // ERROR
18 }
19
20 vpfloat<mpfr, 16, prec> // OK
21   example_dyn_type_return (unsigned prec) {
22   vpfloat <mpfr, 16, prec> a = 1.3;
23   return a;
24 }
25
26 vpfloat<mpfr, 16, prec> // ERROR
27   example_dyn_type_return_error (unsigned p) {
28   vpfloat <mpfr, 16, p> a = 1.3;
29   return a;
30 }
```

Listing 3. Uses of dynamically-sized types in function call and return

Listings 2 shows that programmers can also make use of dynamically-sized types as function parameters, as long as attributes are known declarations for the specific context. In that case, a valid runtime attribute may come from a global integer variable declaration or a previously declared parameter, as shown in the examples. Our compiler parses and analyzes the given attributes in order to ensure that known attributes are being used. It is important to highlight that dynamically-sized types are only valid within the scope in which they were created. In other words, functions do not share dynamically-sized types, but dynamic size attributes are bound to formal arguments so that (dependent) types from different functions can be passed through function calls.

Function `example_dynamic_type` in Listing 3 shows examples of how types interact in a function call. Similarly to VLAs, the compiler ensures that each type attribute in a formal argument of the callee depends on attributes properly bound in the declaration. Any inconsistency found by the compiler is reported back to the user through our compile-time and runtime checks. A compile-time error is raised at line 10, since values of `a`, `X`, and `Y` were created with a constant value of 200, instead of 100. Lines 14 and 17 show examples of how runtime verifications can guarantee correctness between attributes and `vpfloat` declaration in function calls. In line 14, a runtime error is reported back to the user if `p` is not equal to 200, while in line 17 an error is raised since the value of `p` has changed.

Dynamically-sized types can also be declared as return types and their semantics are similar to function arguments. Our language allows attributes of return types to be bound to function arguments, even though arguments are not yet available when parsing the return type. Our compiler checks and builds a function's return type after all arguments are processed, and semantic analysis verifies that attributes given in a declaration exist and can be used to build a return type. While a parameter requires attributes to be declared previous to its declaration, this is not the case for return types. For example, `example_dyn_type_return` shows how to use dynamically-sized types with a function argument as an attribute, and `example_dyn_type_return_error` is caught by syntax analysis since `prec` is not declared in that context.

FP literal support is also provided. Constant-size types have the advantage of having a fixed memory footprint, and thus constant values can be represented as soon as values of the exponent and mantissa are known. Dynamically-sized types pose another challenge as attributes are only known at runtime, which makes it impossible to represent a constant value according to its statically unknown attributes. We handle them by creating a fixed size representation of the constant in a maximum configuration at compile time,[2] and cast it at runtime to the dynamically-sized type in use.

B. Intermediate Representation (IR)

We provide an extension to the LLVM IR Type System that enables `vpfloat` FP types in the intermediate code. Similar to the language-level type system, `vpfloat` IR types also keep the information about attributes, so that expressiveness is maintained even in a lower level of the compilation process. Attributes are declared as `Value`[3] objects, since attributes can be represented by an instruction, a constant, or function parameter.

Our deep integration with LLVM allows `vpfloat` types to interact with optimizations available in the infrastructure. The following paragraphs cover the modifications necessary in the toolchain in order to add `vpfloat` types compatibility.

- One important aspect of our design is that types and attributes are not linked through a def-use relation. In other words, a `vpfloat` type cannot be obtained by traversing the def-use chain of an attribute. This approach was not considered due to requiring modification of many key components of the compiler. Instead, we keep a list of all objects being used as types attributes. If an object is replaced by a new one, our type system makes sure to update any type that uses it. An object deletion, on the other hand, can potentially invalidate types. The compiler makes sure `vpfloat` attributes are not deleted by adding a mark through an intrinsic call. Although this may have a negative impact on the generated code, we ensure that `vpfloat` types are not invalid. Notice that constant values do not require tracking, since they will never change,
- *Loop Idiom Recognition* is an optimization that transforms simple loops into a non-loop form. Two of its optimizations are the generation of `memset` calls to initialize

[2]For *unum* types, the maximum configuration is 16 bits of exponent and 512 bits of mantissa; for *mpfr* types the maximum configuration is 16 bits of exponent and 240 bits of mantissa.

[3]`Value` is an LLVM base class to define arguments, instructions, constants, etc., in the IR.

objects in an array and `memcpy` calls to copy objects from a location to another. We have modified this pass to take into consideration dynamically-sized types where sizes cannot be known in compile time. If a dynamically-sized type is found, the compiler uses the `__sizeof_vpfloat` function to calculate the size of the type in use. Due to the requirements of *mpfr* types (see Section III-C1), this optimization can only be enabled for *unum* types,

- *Inline Expansion* replaces a function call site by the body of the function. Dynamically-Sized types require additional work during inline expansion as they are only considered valid inside a function. We have expanded the pass to include support for dynamically-sized types. Values with dynamically-sized types have their types changed (or mutated) in order to comply to the current function where they are being used.

C. Backends

We designed and implemented two backend code generators that both support dynamically-sized `vpfloat` variables to evaluate the effectiveness of our language extensions.

1) MPFR Library: The MPFR code generator takes the form of a middle-end transformation pass that lowers the `vpfloat<mpfr, ...>` type into MPFR references. It runs at a late stage of the middle-end LLVM compiler to guarantee that the main optimizations, if enabled, have already been executed. Although the pass is used for MPFR code generation, we made it generic enough to handle any type expressible with `vpfloat`.

The pass traverses functions in the compilation unit (or module) searching for `vpfloat<mpfr, ...>` types and recreate them as MPFR objects. Lowering to MPFR calls and references involves the following transformations:

1) MPFR represents its objects by a C struct (see Listing 1) that must be allocated and initialized before first use. Of course, MPFR C++ wrappers already exist to abstract these allocations and deallocations away from programmers. We provide a similar functionality by monitoring LLVM IR `alloca` instructions and their enclosing scope.[4] This enables fully transparent creation and deletion of MPFR objects. In addition, any optimization pass reducing the number of live variables will translate into more efficient memory management after lowering to MPFR. The pass is also in charge of generating proper object initialization, translating constant and dynamically-sized types to the appropriate MPFR configurations and calls. In particular, the size of the exponent and mantissa are set up during initialization. Our pass detects single and multi-dimensional arrays and structs of variable-precision values, generating the appropriate calls to allocate multiple MPFR objects if needed. Moreover, it supports the creation and deletion of MPFR objects through dynamic memory allocation (`malloc`, `new`, etc.), and transparently manages objects created with these functions,
2) Arithmetic IR instructions *fadd, fsub, fmul, fdiv* are converted to *mpfr_{add,sub,mul,div}* or any of their derivative functions (*mpfr_{add,sub,mul,div}_{si,ui,d}*). Comparisons, negation, and conversions all have corresponding functions in the MPFR library. Store instructions are converted to *mpfr_set* or any of its derivative functions (*mpfr_set_{si,ui,d}*). We try to leverage MPFR functions specialized for the case where one or more operand is a primitive data type, e.g. `double`, `unsigned`,
3) Functions with `vpfloat` MPFR arguments are cloned and re-created as MPFR objects. The pass respects the C standard for argument passing, such as, *pass by value*, *pass by reference*, etc. Return types are handled through LLVM's *StructRet* attribute and returning the value as the first argument of the function,
4) Load instructions, Φ Nodes, dereferencing (element-indexing) instructions, and constant values of `vpfloat` arguments are all rewritten to use the MPFR struct type.
5) C++ imposes particular challenges for MPFR code generation due to some object-oriented features, such as VTables, lambda functions, and classes. Our code generator supports all these features for constant size types, while dynamically-sized types support is part of a future work. Compound types (class, struct, and function types, etc.) that make use of `vpfloat<mpfr, ...>` and its compound-type variances (pointers, arrays, etc.) are reconstructed as MPFR `struct` types. VTables are all updated to the newly recreated references so that the C++ polymorphism feature is supported. Although *Loop Idiom Recognition* is disabled for `vpfloat<mpfr, ...>`, the compiler can still make use of memory-related functions (`memcpy`, `memmove`, etc.) through the C++ standard library, or when capturing lambda functions by value. Our code generator detects these functions and generates an additional set of functions that provides full support for them. Essentially, we are able to guarantee that the pointer to the mantissa field is not overwritten, only its content is copied/moved accordingly,
6) OpenMP support is included almost out-of-the-box. The only special treatment lies on handling `omp atomic` directives, which generates a call to `atomic_compare_exchange` that implements an atomic compare-and-swap.[5] Since MPFR objects cannot be atomically modified with a single IR intrinsic or instruction (they use a library call) our code generator enforces atomicity by inserting a critical section and calling our implementation of `compare_and_exchange`. The critical section uses a dedicated mutex, properly nested to avoid interference with any other synchronization,
7) Eventually, the MPFR code generation pass attempts to optimize the number of dynamically created MPFR objects by reusing old references if it is guaranteed that their values will no longer be needed. Notice that

[4]Since `vpfloat` variables are typed as first-class scalar values, they are modeled as stack-allocated in upstream passes.

[5]https://en.cppreference.com/w/c/atomic/atomic_compare_exchange

the pass operates on SSA form, making this step differ from a traditional copy elimination and coalescing, both implemented in target-specific backend compilers. Instead, we follow a backward traversal of use-def-chains to identify MPFR objects that may be shared across variable renaming of invariant values, and across convergent paths with mutually exclusive live intervals.

In summary, the pass rewrites all `vpfloat<mpfr, ...>` operands by replacing with MPFR objects and the appropriate initialization. Unlike higher level MPFR abstractions such as the C++ Boost library for multi-precision arithmetic [11], we are able to leverage the compiler toolchain and its existing optimizations, with MPFR objects only being created at the end of the middle-end compilation flow.

2) UNUM-Based ISA: Our second backend makes use of the `vpfloat<unum, ...>` representation in order to partially implement the RISC-V ISA [23] extension for UNUM variable precision arithmetic of [9]. This ISA extension supports generic FP instructions with precision ranging from 8 to 512 bits. Since the UNUM format [20] is used to represent values stored in memory, loads and stores must be parameterized according to the variable size and positioning of the UNUM fields in the highly flexible format. Two control registers hold the *ess* and *fss* fields of the UNUM formats, defining the number of bytes read and written into memory. The ISA also defines concepts of WGP (Working G-layer precision) and MBB (Memory Byte Budget), which are, respectively, the precision used in computation and the maximum number of bytes read and written during load and store operations.

We designed and implemented two passes to properly handle the generation of generic FP operations with the UNUM ISA:

1) FP configuration: the first pass analyzes functions in the call graph and configures values of *ess*, *fss*, *WGP* and *MBB* as to convey the high level type information. The pass keeps track of values that come in and go out of basic blocks. By analyzing the control flow graph, it guarantees that values are being properly assigned. If any change is needed when entering a basic block, a new instruction is added.
2) Array address computation: the second pass is applicable only to dynamically-sized types and aims at providing proper array addresses. Since LLVM provides no support for dynamically-sized types, additional care is needed to compute the addresses of values whose sizes are only known at runtime. The `__sizeof_vpfloat` function allows to perform this task. The pass traverses every function searching for *GetElementPtr* instructions. These instructions are replaced by the appropriate low level address computation, accumulating over the number of elements and the dynamic size of every element.

IV. Experimental Results

We conducted a series of experiments to compare our solution to state-of-the-art approaches.

A. MPFR vpfloat vs. Boost for Multi-Precision

The first experiment consists of a comparison of our MPFR type `vpfloat<mpfr, ...>` with the Boost library for multi-precision. Both approaches rely on the MPFR library and execute code with identical precision. The difference resides in how optimizations can improve performance on applications when lowering to MPFR calls takes place at a late optimization stage, as is the case for our solution.

We compiled the Polybench suite version 4.1 [6] at optimization level -O3, enabling and disabling Polly's polyhedral loop nest optimizations [24] for both `vpfloat<mpfr, ...>` and Boost. The execution time reference for each application is the best of both (with and without Polly). We also compiled the RAJAPerf suite [25] at optimization level -O3, with 6 different variants: three that explore sequential execution, and three with OpenMP Support. Tests were conducted on a system with two Intel Xeon E5-2637v3 with 128GB of RAM, for a total of 8 cores and 16 hardware threads.

Figure 1 shows the speedup for each benchmark using Boost as the baseline in (1) the Polybench suite, and (2) RAJAPerf. We observe speedups over most of the test suite. Jacobi 1D and 2D have similar performance to Boost at the lower precision settings. The only slowdowns are for *adi* and *deriche* in Polybench, at lower precisions only, and some RAJAPerf variants. These results are due to the complexity of the array access patterns in the stencil kernel, hitting limitations of the MPFR lowering pass in reusing MPFR objects over invariant or mutually exclusive values. In other cases, the measures show that a late MPFR lowering dramatically improves performance, especially on computationally intensive kernels benefiting from greater cache locality and a proportionally more significant decrease of MPFR memory management overhead.

Interestingly, RAJAPerf shows that our solution scales much better than Boost in a multi-threaded environment. Hardware counter measurements indicate that speedups in the 7–9× range with OpenMP stem from the reduction of memory accesses and cache misses, with up to 90× reduction in last-level cache misses. The Boost implementation often converts compute-bound kernels into memory-bound ones as memory transactions exceed the off-chip bandwidth. Deep integration with the compiler and its optimizations and the reuse of old MPFR objects contribute to reduce the memory pressure and allow our solution to scale much better in multi-threaded environments.

Overall, results show an average performance speedup of 1.80× for the Intel Xeon processor when comparing `vpfloat<mpfr, ...>` to the Boost library for MPFR in Polybench. For RAJAPerf, we observed average speedups of 1.74×, 1.61×, and 1.65×, for sequential execution variants (Base_Seq, Lambda_Seq, and RAJA_Seq, respectively), and 7.98×, 7.16×, and 7.72× for the OpenMP variants (Base_OpenMP, Lambda_OpenMP, and RAJA_OpenMP).

B. UNUM vpfloat with Hardware Support vs. MPFR vpfloat

We also demonstrate the effectiveness of our type extension and its integration with LLVM on a hardware implementation of the UNUM format. To the best our knowledge, until

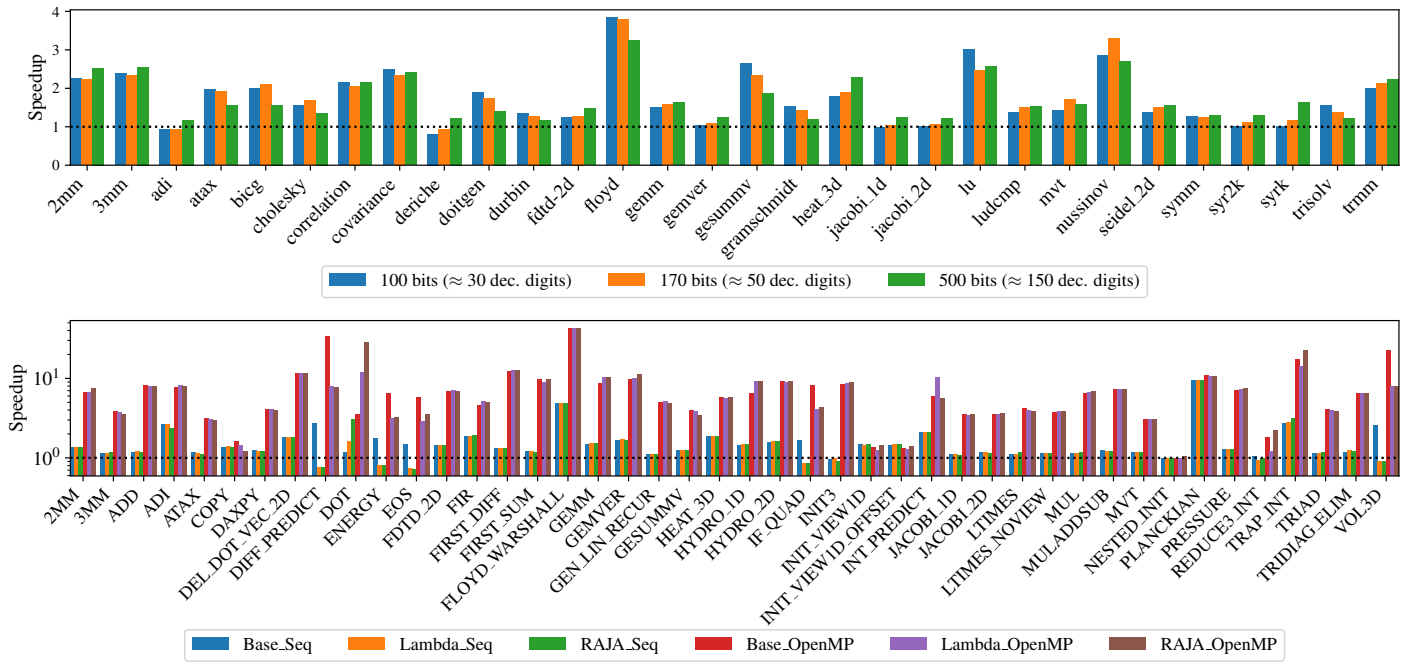

Fig. 1. Speedup of `vpfloat<mpfr, ...>` over the Boost library for multi-precision both targeting MPFR library calls for (1) the Polybench benchmark suite, and (2) the RAJAPerf benchmark suite.

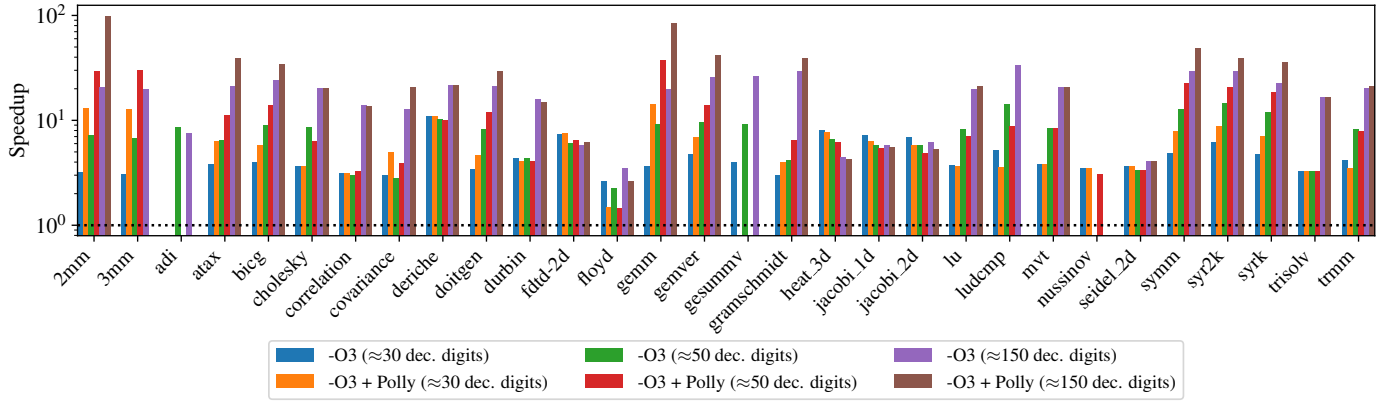

Fig. 2. Speedup of `vpfloat<unum, ...>` over `vpfloat<mpfr, ...>` on the PolyBench suite

now, UNUM's functionality could only be evaluated with software libraries since no hardware implementation supported a software stack capable of running representative benchmarks. Owing to the better performance observed in Section IV-A when compared to Boost, we used our `vpfloat<mpfr, ...>` implementation as the baseline for comparison with the UNUM coprocessor.

Our target platform consists of an FPGA implementation of a RISC-V Rocket processor [26] connected to the UNUM coprocessor of [9]. All benchmarks including baseline MPFR implementations have been compiled to the RISC-V ISA. Indeed, our MPFR backend is target independent: i.e., applications with `vpfloat<mpfr, ...>` types can potentially be executed on any LLVM-compatible platform with MPFR support.

Figure 2 shows the speedup of applications normalized to the baseline MPFR performance (note the logarithmic scale). Unfortunately we hit hardware bugs when executing some benchmarks: *gesummv* and *adi* failed to run when compiled with Polly and 3 more benchmarks failed at the highest precision with Polly (*3mm*, *ludcmp*, *nussinov*). This is due to an issue in the coprocessor memory subsystem.We notice that Polly is able to significantly improve performance for many applications in the test suite. This is solid validation of the robustness of our design and implementation, given the complexity of polyhedral compilation methods and their sensitivity to efficient memory management. It further validates the benefits of making variable precision FP arithmetic transparent to upstream optimization passes. Notably, *gemm*, *2mm* and *3mm* show speedup of more than 20× over the baseline, benefiting from cache and register reuse through polyhedral loop optimization with downstream loop unrolling and scalar promotion. Average speedups at the highest precision (150 digits) are 18.03× and 27.58× for -O3, and -O3 + Polly, respectively. The rare slowdowns with Polly are caused by suboptimal heuristic tuning, a well known challenge with loop nest optimizations in general.

C. Dynamically-Sized Types

Dynamically-sized types are a great exploration tool for precision-awareness in numerical applications. They are also essential to adaptive or variable precision computing. Although high-level languages like Julia [18] or Python [27] provide dynamic type systems amenable to this kind of research, our type system has the great advantage of enabling C-level performance, as well as supporting both hardware and software targets.

To demonstrate how dynamically-sized types can be beneficial, we implemented the Conjugate Gradient (CG) method [28], an iterative solver for linear systems, using our `vpfloat<mpfr, ...>` type. CG is a good example to illustrate the influence of arbitrary precision in an application's output, since the number of iterations needed for the algorithm to converge depends on the chosen precision. The pseudo-code for the algorithm is given by Algorithm 1, where line 6 shows how error controls the number of iterations.

Algorithm 1 conjugate gradient: original Hestenes and Stiefel algorithm [28]

1: $p_0 := r_0 := b - Ax_0$
2: **while** iteration count not exceeded **do**
3: $\quad \alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
4: $\quad x_{k+1} := x_k + \alpha_k p_k$
5: $\quad r_{k+1} := r_k - \alpha_k A p_k$
6: $\quad$ **if** $\| r_{k+1} \| \leq tol$ **then** break
7: $\quad \beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
8: $\quad p_{k+1} := r_{k+1} + \beta_k p_k$
9: $\quad k++$
10: **end while**

Listing 4 shows the headers of the functions created using our language extension, and needed to implement the CG algorithm. We only replace: (1) vector-scalar products by `vaxpy`, (2) matrix-vector products by `vgemv`, (3) dot products by `vdot`, and (4) products of a vector by a scalar by `vscal`. Moreover, our implementation is precision-generic: the core CG iteration takes a precision parameter, and every run of the function can make use of a different precision value. Hence a single run of the application, *without recompilation*, enables programatically-driven experimentations with multiple precision configurations.

Figure 3 shows the impact of precision on a standard matrix taken from the Matrix Market suite [29], [30]. Our experiments confirm a well-established result: the higher the precision, the fewer iterations it takes to converge. Although it may initially sound paradoxical, this demonstrates that higher precision may actually reduce the execution time of a numerical application.

The impact of precision on the number of iterations, due to accumulation of rounding errors [31], is not linear for CG, and the rapid drop in execution time stems from faster convergence. Increasing precision beyond 700 bits, the speedup plateau for this example, (slowly) negatively impacts performance, although the number of iterations continues to reduce. These findings enable us to further investigate the impact of precision

```
1  void vaxpy(unsigned precision, int n,
2             vpfloat<mpfr, 16, precision> alpha,
3             vpfloat<mpfr, 16, precision> *X,
4             int incx,
5             vpfloat<mpfr, 16, precision> *Y,
6             int incy);
7
8  void vgemv(unsigned precision, int m, int n,
9             vpfloat<mpfr, 16, precision> alpha,
10            double *A, int *rowInd, int *colInd,
11            vpfloat<mpfr, 16, precision> *X,
12            vpfloat<mpfr, 16, precision> beta,
13            vpfloat<mpfr, 16, precision> *Y);
14
15 vpfloat<mpfr, 16, precision>
16     vdot(int precision, int n,
17          vpfloat<mpfr, 16, precision> *X,
18          int incx,
19          vpfloat<mpfr, 16, precision> *Y,
20          int incy);
21
22 void vscal(unsigned precision, int n,
23            vpfloat<mpfr, 16, precision> alpha,
24            vpfloat<mpfr, 16, precision> *X,
25            int incx);
```

Listing 4. Function headers for a variable-precision BLAS library used to implement algorithm 1

in iterative methods. In addition, our experiments once again confirm that our `vpfloat` implementation outperforms the Boost library for multi-precision in this benchmark by a factor of $1.51\times$, and the Julia version by more than $9\times$.

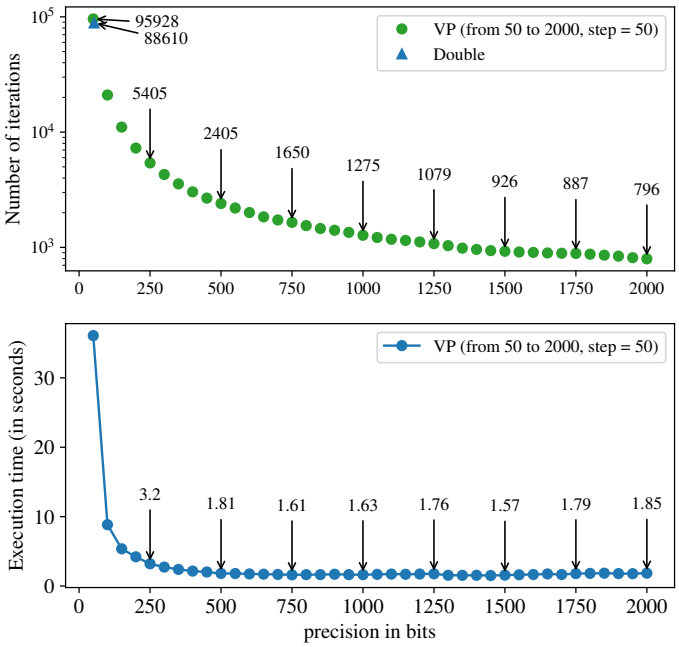

Fig. 3. Conjugate Gradient (CG) iterations and precision on the *bcsstk20* matrix

V. Related Work

Mixed precision algorithms [13] have been highly successful in saving energy and accelerating scientific codes, and more recently, machine learning models [32], [33]. Obviously, reducing precision without caution may harm numerical stability

and magnify error propagation: this motivates analyses and heuristics to determine a suitable FP representation for all input, output and temporary values in a numerical application [34]--[36]. However, as discussed in Section II, the immaturity of software and systems for non-IEEE formats hinders the exploration of the most suitable representation beyond the standard IEEE 16–128 bits formats.

This partly motivated the design and implementation of the MPFR library [7], addressing the bottom layers of the software infrastructure. It is highly optimized and portable for higher precision FP arithmetic. It underlines numerous research and applications, from numerical analysis to computational geometry. At the higher layers of the stack, multi-precision abstractions have been adopted in C++ [11], Python [27] and Julia [18], following much earlier language and compiler support for Ada [37]. The former is the natural baseline for our performance comparisons: it offers a similar abstraction and productivity level as our `vpfloat` generic type but lacks first-class compiler support for downstream optimization and backend code generation. The latter three provide near seamless transition across format and precision, but lack optimizing compiler support for their multi-precision abstractions (Python and Julia also suffer from runtime memory management overhead and dynamic typing).

UNUM [20] and Posit [38] are two variable precision formats proposed to tackle limitations of IEEE standard. They provide variable-length exponent and mantissa fields: (1) UNUM through two extra fields *ess* and *fss*, the exponent size and mantissa size, respectively, and (2) Posit through *regime bits* providing tapered accuracy which changes the size of exponent field. Recent work evaluated the potential of these alternative formats in scientific computing [39], [40] as well as machine learning [41], [42]. Hardware accelerators and FP coprocessors [9], [43]--[45] have also been proposed to facilitate performance comparison across formats. Several studies specifically addressed the comparison of these alternative representations with the IEEE standard [46]--[49], but they all lack of an integrated flow to compare numerical benchmarks across formats, precision control schemes, and hardware/software implementations.

Tiwari et al. [43] proposed to address the issue by interpreting `float` or `double` types as Posit computing. Bocco et al. [40] introduced a new data type as to more transparently integrate IEEE and non-IEEE format computations at the ISA level. Flytes [50] enabled the use of lower precision formats for reducing the cost of data transfer volume and storage space requirements. To the best of our knowledge, no previous work tackled the integration with an optimizing compilation flow, taking into consideration format-specific attributes, and how the compiler can efficiently generate code for formats that may not have statically fixed size. Unlike flytes which converts non-IEEE types early on into standard types, we provide IR support for multiple representations, without depending on IEEE types internally. This has never been achieved before, not even on statically sized FP types. Our solution also provides comparable or better expressiveness than other approaches that rely on high-level abstractions, like the C++ Boost library or the Julia language, with hardware support compatibility when it exists. Moreover, support for larger-than-standard FP types is a novelty not easily found in state-of-the-art compilers nowadays (except from POWER9 `double-double` (128 bit), and x86 FP80 formats). Finally, Ansel et al. [51] provides numerical analysts an exploration tool for tuning the precision of non-IEEE types. It also helps to motivate our work on multi-precision FP types, and we provide a more thorough experimentation platform, fully-integrated with an industry-standard compiler.

VI. Conclusion and Future Work

We propose a type system, an embedding into the compiler's intermediate representation, and lowering and backend code generation strategies to provide both high-productivity and high-performance with variable precision FP formats. Our extension supports FP arithmetic of arbitrary representation and precision, whose precision and exponent can be configured at compilation time and runtime.

We implemented this extension into LLVM to benefit from all optimizations available on its intermediate representation. We also modified specific optimizations such as *Loop Idiom Recognition* and *Inlining* to handle types whose memory footprint can vary at runtime. We developed two code generators: a backend for the MPFR multi-precision FP library which lowers operations on our abstract data type to MPFR calls and manages the allocation of MPFR objects, and a backend for the ISA of a UNUM coprocessor.

We demonstrate the productivity benefits of our intuitive programming model and its ability to leverage an existing optimizing compiler framework. Experiments on PolyBench and RAJAPerf suites yield strong speedups for both software and hardware targets.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough and valuable comments. This work was partially funded by the French *Agence nationale de la recherche (ANR)* for project IMPRENUM (Improving Predictability of Numerical Computations) under grant n° ANR-18-CE46-0011.

References

[1] D. Zuras *et al.*, *IEEE standard for floating-point arithmetic*, ser. IEEE Std 754, 2008.

[2] A. M. Frolov and D. H. Bailey, ''Highly accurate evaluation of the few-body auxiliary functions and four-body integrals,'' *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 37, no. 4, pp. 955--955, Feb. 2004.

[3] D. H. Bailey and J. M. Borwein, ''High-precision arithmetic in mathematical physics,'' *Mathematics*, vol. 3, no. 2, pp. 337--367, 2015.

[4] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1996.

[5] E. Carson and N. Higham, ''Accelerating the solution of linear systems by iterative refinement in three precisions,'' *SIAM Journal on Scientific Computing*, vol. 40, pp. A817--A847, Jan. 2018.

[6] L.-N. Pouchet *et al.*, ''PolyBench: The polyhedral benchmark suite,'' 2012, http://www.cs.ucla.edu/pouchet/software/polybench.

[7] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, ''MPFR: A multiple-precision binary floating-point library with correct rounding,'' *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007.

[8] T. Granlund, ''GNU multiple precision arithmetic library 6.1.2,'' Dec. 2016, https://gmplib.org/.

[9] A. Bocco, Y. Durand, and F. De Dinechin, ''SMURF: Scalar multiple-precision unum risc-v floating-point accelerator for scientific computing,'' in *Proceedings of the Conference for Next Generation Arithmetic*, 2019, pp. 1--8.

[10] C. Lattner and V. Adve, ''LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,'' in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75--86.

[11] J. Maddock and C. Kormanyos, *Boost.Multiprecision*, 2020, https://www.boost.org/doc/libs/1_74_0/libs/multiprecision/doc/html/index.html.

[12] L. Dagum and R. Menon, ''Openmp: an industry standard api for shared-memory programming,'' *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46--55, 1998.

[13] M. Baboulin, A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, ''Accelerating scientific computations with mixed precision algorithms,'' *Computer Physics Communications*, vol. 180, pp. 2526--2533, 2009.

[14] J. K. Lee, ''Air: Adaptive dynamic precision iterative refinement,'' Ph.D. dissertation, University of Tennessee, 2012.

[15] J. K. Lee, H. Vandierendonck, M. Arif, G. D. Peterson, and D. S. Nikolopoulos, ''Energy-efficient iterative refinement using dynamic precision,'' *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 722--735, 2018.

[16] H. Anzt, J. Dongarra, G. Flegar, N. Higham, and E. S. Quintana-Orti, ''Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers: Adaptive precision in block-jacobi preconditioning for iterative solvers,'' *Concurrency and Computation: Practice and Experience*, vol. 31, p. e4460, Mar. 2018.

[17] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, ''The transprecision computing paradigm: Concept, design, and applications,'' in *Design, Automation Test in Europe Conference & Exhibition*, 2018, pp. 1105--1110.

[18] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, ''Julia: A fast dynamic language for technical computing,'' *arXiv preprint arXiv:1209.5145*, 2012.

[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, ''Basic linear algebra subprograms for fortran usage,'' *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308--323, 1979.

[20] J. L. Gustafson, *The end of error: UNUM computing*, 2017.

[21] A. Bocco, Y. Durand, and F. de Dinechin, ''Dynamic precision numerics using a variable-precision UNUM type I HW coprocessor,'' in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 104--107.

[22] ISO/IEC JTC 1/SC 22, *Information technology -- Programming languages -- C*, Geneva, Switzerland, Jul. 2018, cancels and replaces ISO/IEC 9899:2011. [Online]. Available: https://www.iso.org/standard/74528.html

[23] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, ''The RISC-V instruction set manual, volume i: User-level ISA,'' *CS Division, EECE Department, University of California, Berkeley*, 2014.

[24] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, ''Polly -- polyhedral optimization in LLVM,'' in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011, pp. 1--6.

[25] ''Rajaperf.'' [Online]. Available: https://github.com/LLNL/RAJAPerf

[26] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, ''The Rocket Chip Generator,'' EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[27] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[28] M. R. Hestenes *et al.*, ''Methods of conjugate gradients for solving linear systems,'' *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409--436, 1952.

[29] ''Matrix market repository,'' https://math.nist.gov/MatrixMarket/, 2007.

[30] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, ''Matrix market: A web resource for test matrix collections,'' in *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*. Chapman & Hall, Ltd., 1997, pp. 125--137.

[31] E. Carson and Z. Strakoš, ''On the cost of iterative computations,'' *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2166, p. 20190050, Mar. 2020. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2019.0050

[32] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, ''Deep learning with limited numerical precision,'' in *Proceedings of the 32nd International Conference on Machine Learning*. JMLR.org, 2015, pp. 1737--1746.

[33] Z. Li, Y. Ma, C. Vajiac, and Y. Zhang, ''Exploration of Numerical Precision in Deep Neural Networks,'' *arXiv e-prints*, May 2018.

[34] 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 *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1--12.

[35] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, ''Rigorous floating-point mixed-precision tuning,'' *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 300--315, 2017.

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

[37] R. P. Leavitt, ''Adjustable precision floating point arithmetic in Ada,'' *ACM SIGAda Ada Letters*, vol. VII, pp. 63--78, 1987.

[38] J. L. Gustafson and I. Yonemoto, ''Beating floating point at its own game: Posit arithmetic,'' *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71--86, 2017.

[39] J. Hou, Y. Zhu, S. Du, and S. Song, ''Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA,'' *Journal of Signal Processing Systems*, vol. 91, no. 10, pp. 1137--1148, 2019.

[40] A. Bocco, T. T. Jost, A. Cohen, F. de Dinechin, Y. Durand, and C. Fabre, ''Byte-aware floating-point operations through a UNUM computing unit,'' in *IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2019, pp. 323--328.

[41] J. Johnson, ''Rethinking floating point for deep learning,'' *arXiv preprint arXiv:1811.01721*, 2018.

[42] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, ''Deep positron: A deep neural network using the Posit number system,'' in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2019, pp. 1421--1426.

[43] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, ''PERI: A posit enabled RISC-V core,'' *arXiv preprint arXiv:1908.01466*, 2019.

[44] M. K. Jaiswal and H. K.-H. So, ''Pacogen: A hardware posit arithmetic core generator,'' *IEEE access*, vol. 7, pp. 74 586--74 601, 2019.

[45] F. Glaser, S. Mach, A. Rahimi, F. K. Gürkaynak, Q. Huang, and L. Benini, ''An 826 MOPS, 210uW/MHz UNUM ALU in 65 nm,'' in *IEEE International Symposium on Circuits and Systems*. IEEE, 2018, pp. 1--5.

[46] F. De Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, ''Posits: the good, the bad and the ugly,'' in *Proceedings of the Conference for Next Generation Arithmetic*, 2019, pp. 1--10.

[47] P. Lindstrom, S. Lloyd, and J. Hittinger, ''Universal coding of the reals: alternatives to IEEE floating point,'' in *Proceedings of the Conference for Next Generation Arithmetic*, 2018, pp. 1--14.

[48] S. W. Chien, I. B. Peng, and S. Markidis, ''Posit NPB: Assessing the precision improvement in HPC scientific applications,'' *arXiv preprint arXiv:1907.05917*, 2019.

[49] J. Hou, Y. Zhu, Y. Shen, M. Li, Q. Wu, and H. Wu, ''Enhancing precision and bandwidth in cloud computing: Implementation of a novel floating-point format on FPGA,'' in *IEEE 4th International Conference on Cyber Security and Cloud Computing*, 2017, pp. 310--315.

[50] A. Anderson and D. Gregg, ''Vectorization of multibyte floating point data formats,'' in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, pp. 363--372.

[51] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, ''Language and compiler support for auto-tuning variable-accuracy algorithms,'' in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. 85--96.