Compilation and optimizations for variable precision floating-Point arithmetic : from language and libraries to code generation

Tiago Trevisan Jost

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Tiago TREVISAN JOST

Thèse dirigée par **Frédéric PETROT**, Professeur, Grenoble INP/Ensimag, Université Grenoble Alpes
et codirigée par **Albert COHEN**, Ingénieur HDR, Google
et **Christian FABRE**, ingénieur-chercheur, CEA LIST Grenoble
préparée au sein du **Laboratoire d'Intégration des Systèmes et des Technologies / CEA LIST Grenoble** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Compilation et optimisations pour l'arithmétique à virgule flottante en précision variable : du langage et des bibliothèques à la génération de code

Compilation and optimizations for Variable Precision Floating-Point Arithmetic: From Language and Libraries to Code Generation

Thèse soutenue publiquement le **02/07/2021**
devant le jury composé de :

Monsieur FREDERIC PETROT
PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES, Directeur de Thèse
Monsieur PHILIPPE CLAUSS
PROFESSEUR DES UNIVERSITES, UNIVERSITE STRASBOURG, Rapporteur
Monsieur ERVEN ROHOU
DIRECTEUR DE RECHERCHE, CNRS BRETAGNE et PAYS DE LA LOIRE, Rapporteur
Monsieur SEBASTIAN HACK
PROFESSEUR, Universität des Saarlandes, Examinateur
Monsieur DAVID MONNIAUX
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Président
Madame NATHALIE REVOL
CHARGE DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONEALPES, Examinatrice
Monsieur ALBERT COHEN
INGENIEUR HDR, Google France, Co-directeur de thèse
Monsieur CHRISTIAN FABRE
INGENIEUR DE RECHERCHE, CEA-List, Invité et Co-Encadrant de Thèse

Abstract

Floating-Point (FP) units in processors are generally limited to supporting a subset of formats defined by the IEEE 754 standard, along with a few target-specific ones (X86 with an 80-bit FP format, and PowerPC performing 128-bit FP arithmetic). As a result, high-efficiency languages and optimizing compilers for high-performance computing are also limited by the FP types supported by these units. However, the pursuit of efficiency and stability on applications has led researchers to investigate a finer control of exponent and fraction bits for finding the right balance between accurate results and execution time and/or energy consumed. For example, numerical computations often involve iterative solvers where the residual error is a function of the input data, or where dynamically adaptive precision can accelerate convergence. Numerical analysts have to resort to explicit conversions and multi-versioning, resulting in code bloat and making the intent of the program even less clear. Little attention in languages and compilers has been given to formats that disrupt the traditional FP arithmetics with runtime capabilities and allow the exploration of multiple configurations, a paradigm recently referred to as variable precision computing. This thesis proposes to overcome the limiting language and compiler support for traditional FP formats with novel FP arithmetic with runtime capabilities, showing the intersection between compiler technology and variable precision arithmetic. 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. We provide support for 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. We demonstrate the productivity benefits of our intuitive programming model and its ability to leverage an existing compiler framework. Experiments on two high-performance benchmark suites yield strong speedups for both our software and hardware targets. We also show interesting insights on the use of variable precision computing in linear algebra kernels.

Résumé

Les unités de calcul à virgule flottante (FP) prennent en charge un sous-ensemble de formats définis par la norme IEEE 754, ainsi que quelques formats qui leur sont spécifiques (le format de 80 bits sur de l'architecture x86, et le format 128 bit propriétaire des PowerPC). De fait, les langages et les compilateurs optimisants utilisés en calcul intensif sont limités par les formats supportés sur les machines cibles. Cependant, la recherche de l'efficacité et de la stabilité des applications a conduit les numériciens à explorer d'autres tailles pour les exposants et les parties fractionnaires afin de trouver un bon équilibre entre la précision des résultats, le temps d'exécution et l'énergie consommée. C'est le cas pour les calculs numériques qui font appel à des solveurs itératifs dont l'erreur résiduelle est une fonction des données d'entrée, ou ceux pour lesquels une précision adaptable dynamiquement peut accélérer la convergence. Les numériciens doivent recourir à des conversions explicites et prévoir plusieurs versions du code, ce qui entraîne un accroissement de la taille de ce dernier au détriment de sa lisibilité Peu d'attention a été accordée au support d'autre formats flottants dans les langages et à leur compilation, ainsi qu'à leurs conséquences sur le processus d'analyse numérique. Le calcul en précision variable est un paradigme récent qui propose de faire varier les formats à l'exécution et d'en analyser les effets. Les travaux que nous présentons visent à surmonter les limites actuelles des langages et de leur compilation en y ajoutant le support aux formats à précision variable, et en abordant certains des problèmes que ces formats font apparaître à la jonction de la compilation et de l'arithmétique à précision variable. Pour cela nous proposons une extension du système de types du langage C permettant de représenter de manière générique les formats flottants et leurs opérations, aussi bien en précision statique que dynamique. Nous avons mis en œuvre un flot de compilation qui implémente ce système de type jusqu'à la génération de code vers des jeux d'instructions ou des bibliothèques supportant de tels formats. Notre solution basée sur LLVM a démontré son efficacité en tirant parti des puissantes optimisations de LLVM, notamment l'optimisation de nids de boucles par Polly. Nous proposons un support pour deux générateurs de code : un premier pour le jeu d'instruction d'un coprocesseur arithmétique à précision variable, et un deuxième ciblant la bibliothèque MPFR de virgule flottante en multi précision. Ce support démontre les avantages de productivité de notre modèle de programmation intuitif et sa capacité à tirer parti d'une chaîne de compilation existante. Les expérimentations réalisées sur deux suites de référence en calcul à haute performance ont permis d'obtenir de fortes accélérations aussi bien pour nos cibles logicielles que matérielles. Nous présentons également des résultats intéressants sur l'utilisation de la précision variable pour des noyaux d'algèbre linéaire.

Acknownledgement

First, I would like to the jury for having accepted to be part of my defense and for all insightful comments, and questions. I would also like to thank Christian and Albert for giving me the opportunity to work on this subject since the beginning, and Frédéric for having accepted to get on board in the last year of my PhD. All of you have made these last three years much easier with your availability, insights, feedbacks, and enormous help whenever I needed. I would like to thank Yves, my unofficial math-related advisor, who had the patience to explain many math concepts that I would, otherwise, have struggled with during this period. I extend my gratitude to Vincent, Diego and Alexandre who welcomed me in their laboratories, and all my amazing co-workers from LIALP, LSTA and the whole CEA for their kindness and for their support. Special thanks to Andrea whose work have played an important role in the experiment section of this manuscript. Many thanks to all my friends here in France, in Brazil, and around the world for helping to relax in times of stress, specially close to paper deadlines.

None of this would have been possible without the caring and loving support of my whole family. The awesome energy you have sent me from Brazil surely played a role on the output of this work. Thank you for believing in me. Love you guys! And last but not least, this work is dedicated to Jana, my amazing wife, who were always there for me and has agreed to embark with me in this journey. I can never thank you enough for all your love and support. Love you so much!

Contents

List of Figures

List of Tables

List of Listings

Chapter 1: Introduction

The dearth of compatible floating-point (FP) formats among companies in the early 1980s has significantly held back a plethora of applications, from signal and image processing to neural networks and numerical analysis, that leverage real numbers in their computation. The standardization of FP representations in the 1985 [45] was an important instrument to increase the productivity and the usage of real numbers in a variety of research fields. In the same direction, the progress of Very Large-Scale Integration (VLSI) technology has also contributed to allowing the generalization of hardware units for floating-point arithmetic.

The possibility of packing more transistors in the same die, predicted by Moore's Law [89], has played an important role in the integration of multiple types of FP units in computer systems, for instance, scalar and single instruction, multiple data (SIMD) in single and multicore processors, and single instruction, multiple threads (SIMT) in graphics processing units (GPU). The availability of hardware FP units is not only the rule nowadays, but they are also paramount to the performance of numerical applications. However, none of these major advancements by the hardware industry would have been possible without the effort and robustness of computer systems and, especially, compiler technology.

Compilers play an important role and have long leveraged efficient usage of FP units. Compiler optimizations for FP operations target representations supported by the hardware: at best, 16, 32, 64, and 128 bits IEEE formats, and perhaps some target-specific formats (X86 FP80 and POWER9 128 bits). Although these representations are still well-suited for the majority of applications, there is a need to rethink FP arithmetics as to improve performance, energy and/or accuracy. The reasoning is twofold:

(1) Exploring the trade-off between output quality, and accuracy has already motivated the adaptation of standard FP formats in applications. The main idea is to reduce precision and exponent bits in use in the FP format in an attempt to trade quality by energy and/or performance. By controlling the FP formats at a fine-grained granularity, this trade-off has energy-saving and performance-increase potentials.

(2) A wide range of applications show optimal performance for FP representations that cannot be represented with standard formats. On the one hand, Google's `bfloat16` [1] has attracted a lot of interest for training and inference in neural networks [72] due to having a higher exponent range than the IEEE 16-bit format. On the other hand, linear solvers, n-body problems [50], and other applications in mathematics and physics [14] have shown to benefit from higher-than-standard representations since (a) they may not converge with fewer bits of precision or (b) they can converge faster with higher precision [64]. These applications may suffer from cancellation and accumulative errors when numbers cannot be precisely represented using the standard formats.

Motivation (1) has gained a lot of research attention in the last years through approximate computing (AC) [88, 125]. AC aims to trade accuracy for energy savings and performance with the assumption that a loss of accuracy in output results is irrelevant and can be tolerated.

The latter, especially for higher-than-standard representations, has not been explored to its extent. The pursuit of efficiency and stability in the aforementioned domains has led researchers to reexamine the use of standard formats. Considering there is no unique precision value that fits all targeted applications, variable precision (VP) computing has been used as an exploration tool to search for the most suitable solution for each.

This paradigm change has also led to the emergence of new FP formats and representations. New paradigms and representations must still rely on solid and robust languages and compiler infrastructure in order to ease the exploration of further techniques and solutions. A gap between hardware designs for VP computing and their programming models still exists, as well as what the role of compilers is. This motivates the work of this thesis.

1.1 Contributions

The main contributions of this thesis are:

(1) A multi-format, multi-representation language and compiler support for FP representations that are suitable for VP Computing. Among the sub-contributions related to it are:

   (a) 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 the overall memory footprint. These may be known statically or only at runtime,

   (b) an IR embedding of the runtime and compile-time aspects of generic FP types. Thanks to a tight integration within a state-of-the-art compiler infrastructure, this embedding allows benefiting from most existing compiler optimizations supporting high-performance numerical computing,

   (c) two code generators implemented specifically to take advantage of VP Computing.

(2) A study demonstrating the exploration of VP in high-performance computing (HPC) applications, which includes:

   (a) Basic Linear Algebra Subprograms (BLAS) libraries aimed at helping exploration of VP in applications,

   (b) A study that shows interesting insights to precision exploration in different variants of the Conjugate Gradient algorithm, an iterative method for solving linear systems.

1.2 Outline

This thesis is organized as follows:

Chapter 2 will present the main ideas within the literature and challenges that give basis to this thesis. We will cover aspects related to compiler and language support for novel FP representations, and how precision variation is being neglected on the software/hardware integration stack.

Chapter 3 will discuss the main state-of-the-art concepts related to this thesis, illustrating hardware and software aspects and how they co-related.

The first main contribution of this thesis, the `vpfloat` C-based language extension and type system, is described in Chapter 4, covering all aspects and details of the syntax, and semantics, along with library implementations.

Chapter 5 goes through the compiler integration requirements that are necessary to give support for our type system. It also presents the design and implementation of our code generators and target-specific passes.

Chapter 6 focus on the experimental setup and main results that give basis to the contributions of this thesis.

Finally, chapter 7 summarizes the main contribution of this work and shows the main directions envisioned as part of future work.

Chapter 2: Problem Statement

Contents

2.1 Introduction

Computation techniques for real numbers is still an active field of research. Finding new computer representations for real numbers that are optimized for an application remains challenging. In this chapter, we present the main ideas that form the basis for the research performed through the course of this thesis.

2.2 Precision versus Accuracy

Precision and accuracy are related concepts, although it is a mistake to think they are equivalent. The accuracy of a value relates to the proximity between the measurement of a value and its true value. In the context of the representation of real numbers, it is often expressed as the difference between the result of the computation and its exact result. On the other hand, precision makes reference to the current representation in use, and it is often expressed as a number of bits or digits.

Table 2.1 illustrates the difference between accuracy, here calculated as the norm function between measured and exact values of the output vector or matrix, and precision, represented by 24, 53, 128, and 512 bits. One can notice that the accuracy is an application- and data-dependent constraint, while precision is only subject to the value one chooses to adopt. The choice of

precision generally leads to high accuracy, however, *rounding error* and *cancellation* inherent to finite-sized representation may also influence output accuracy which can sometimes lead to high-precision representations to have less accurate results.

Table 2.1: Residual error for some Polybench [98] applications. It illustrates the difference between accuracy, here calculated through the *residual error* of each application, and precision, represented by 24, 53, 128, and 512 bits. The *residual error* is calculated as the *norm* function between measured and exact values of the output vector or matrix.

		Dataset				
		Mini	Small	Medium	Large	Xlarge
`gemm`	24 bits*	1.5e-5	2.1e-4	4.1e-3	2.3e-1	1.45e0
	53 bits*	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`	24 bits*	6.7e-07	1.1e-04	3.1e-02	4.4e+01	998.4
	53 bits*	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`	24 bits*	5.8e-5	5.6e-3	2.1e-1	41.02	5.7e+02
	53 bits*	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`	24 bits*	28	71	220	616	868
	53 bits*	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

*Correspond to IEEE 32 and IEEE 64 formats, respectively.

Without any need for a rounding operation, computers can only represent a small subset of values (integers and few rational numbers, for example) in a finite number of bits. Predominantly, values cannot be precisely represented with a finite number of bits, and a rounding operation is needed to bound the value to a fixed (and finite) representation. The accumulation of rounding operations may cause the propagation of rounding errors, also known as round-off errors, that can compromise the result of the application. The choice of precision used during the computation can play an important role in binding the error to an expectable value. Table 2.1 shows how round-off errors influence the accuracy of multiple applications from the Polybench suite.

Cancellation is another property from finite-sized representations that can impact the quality of results. It occurs when we subtract two values that are very close, but different. If the difference is too small to be represented with the precision of the numeric format, the result becomes zero. It is usually overcome by analyzing its sources of issues and re-implementing the algorithm [53].

A simple look at Table 2.1 shows that some algorithms are more susceptible to errors than others, and lower-precision implementations have, in general, lower accuracy. Some kernels are also actually numerically unstable for 24, and 53 precision bits, 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.

While most modern processors have hardware support for variants with *24 bits* and *53 bits*, *128-bit*[1] and *512-bit* variants are more cumbersome. High precision is only supported through software libraries MPFR [49], GMP [54], 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. Multiple-precision floating-point arithmetic, as provided by MPFR and GMP to explore different precision levels, is difficult to write and maintain, and more than the performance gap of a software FP implementation, the productivity gap makes this approach inaccessible to potential users.

In that regard, the following questions may be asked: How can languages and compilers be used to accelerate and improve the productivity of multi-precision FP libraries? Can one improve the integration between compilers and these libraries to take better advantage of classical compiler optimizations?

2.3 Floating-Point Representation

Floating point is the most common way to represent real numbers in computer systems. They are written in the form of:

$$(-1)^s \times 1.m \times 2^e \tag{2.1}$$

where s is a single bit specifying the sign of the number, $1.m$ denotes the mantissa part, also known as precision or fraction part, and e represents the exponent of the number.

2.3.1 IEEE Formats

Prior to the standardization of floating-point representations in 1985[2], companies had their own proprietary FP formats, with specific rules and format layout. For instance, IBM System/360 [66] introduced the Hexadecimal floating point (called HFP), Microsoft used the Microsoft Binary Format (MBF) for its BASIC [73] language products, and even the U.S. Air Force defined a formal specification of an ISA that included floating-point capabilities [108]. The IEEE 754 standard technical document served as an important instrument to conform floating-point formats to specific properties, and offer compatibility and portability for long-time use. Since then, the standard was widely adopted for representing real numbers in computer systems.

It defines a set of rules for rounding, exception handling, and operations in FP, along with different encoding formats for FP arithmetics with representation ranging between 16 and 128 bits (see Figure 2.1). Although equation 2.1 generalizes how FP numbers are calculated, one must notice that format-specific features such as, *Infinity*, *Not-a-Number (NaN)*, *subnormals*, and even *biasing* cannot be expressed through the formula. Instead, the IEEE 754 Standard for floating-point arithmetic helps to address them individually.

The Intel 8087 [95], introduced in 1980, was the FP coprocessor for the Intel 8086 line of microprocessors that is historically seen as the pioneer of the IEEE 754 standard. Although the coprocessor did not implement it in all its details, it gave the basis for the standard specification. Subsequently, all major processor manufacturers have started to adopt IEEE formats in the design of FP units in order to leverage compatibility across multiple computing systems.

[1] Notice that *128 bits* of precision does not correspondent to the IEEE FP128 format, which has 113 bits of precision

[2] The IEEE Standard for Floating-Point Arithmetic (IEEE 754) was established in 1985 [45], and revised in 2008 [112] and 2019 [67])

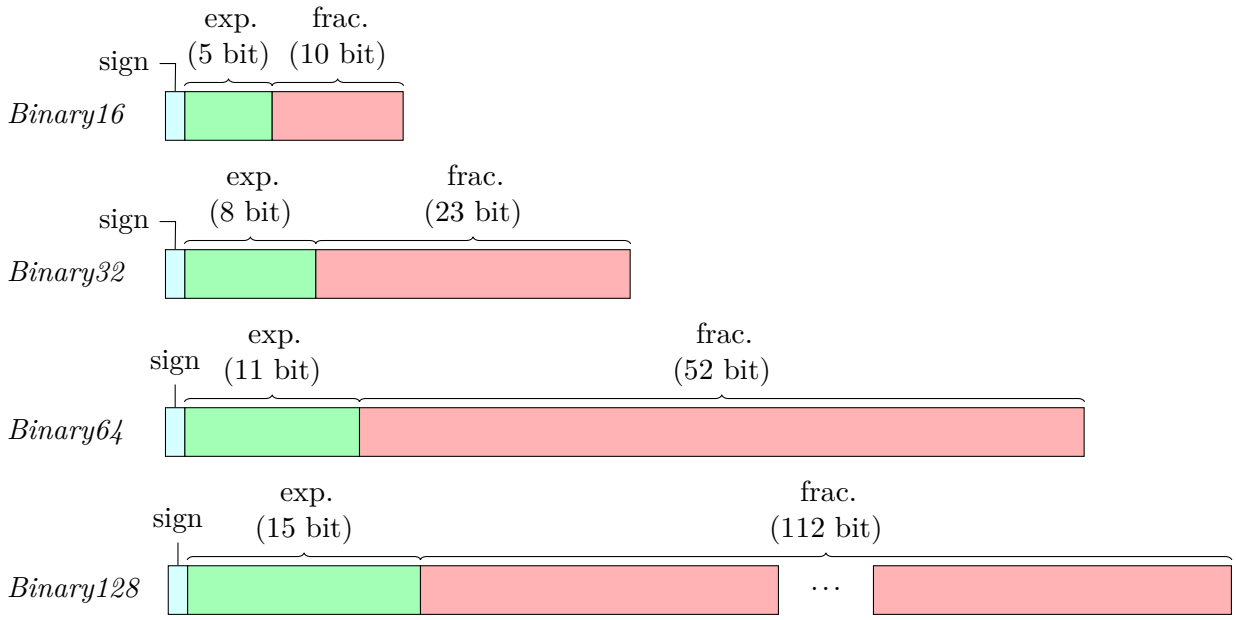


Figure 2.1: IEEE Formats (officially known as *binary16*, *binary32*, *binary64*, and *binary128* as defined by the standard.

Programming languages and compilers have long contributed to support these formats in order to ease the utilization of FP-capable hardware. In C-based languages, for example, types `float` and `double` are typically used for *binary32* and *binary64* formats. Support for *binary128* is provided through `__float128` *type* specifier, while *binary16* has only recently been added to GCC [113] and LLVM [76] compilers.

Although these formats are sufficient for most applications, many works have shown the benefit of using different representations:

(1) `bfloat16` prevailed over IEEE's *binary16* for neural network applications due to its 8-bit exponent size that offers a wider dynamic range and allows IEEE32 to be truncated directly.

(2) IEEE's course-grained format selection hinders the ability to fine-tune the number of exponent and mantissa bits actually needed for a computation. Additionally, one hypothetical application may produce accurate output with a format that has the same number of mantissa bits as *binary32*, and the same number of exponent bits as *binary64*. Exploring new configurations and formats are still limited to library-dependent solutions, thus, there is no downstream compiler support from these libraries.

(3) High precision has shown their importance on many scientific domains [13]. X86 FP80[3] and PowerPC Double-Double[4] formats were proposed as non-standard alternatives for applications that require more accuracy. In fact, even the IEEE committee has considered the growing interest in formats with larger encoding. The IEEE 754-2008 Standard shows how encodings for formats with footprints larger than or equal to 128 bits can be specified. No format definition is defined per se, but the specification and requirements necessary for a format to be considered IEEE beyond 128-bit width are provided. In spite of that, no

[3] X86 FP80 has a sign bit, 15 exponent bits, and 64 mantissa bits with no hidden bit.

[4] PowerPC double-double uses pairs of `double`(*binary64*) to represent 128-bit numbers, with a sign bit, 11 exponent bits and 106 mantissa bits.

further discussion is given, and support for any type beyond 128 bits of a footprint is only achieved with multiple-precision libraries.

Alternatively, an important research venue in the past years lies on rethinking the FP arithmetic in order to compensate for the IEEE's deficiencies (cancellation, rounding). Two alternative floating-point formats, which can provide finer-grained control on the numeric precision and accuracy are UNUM [57] and Posit [58], and are described in the following sections.

2.3.2 UNUM

The Universal NUMber (UNUM) format is a variable precision format proposed in 2015 to overcome some of the rounding-related issues of IEEE formats [57]. It is a self-descriptive FP format with 6 subfields: the sign *s*, the exponent *e*, the fraction *f* (like in IEEE 754) and three descriptor fields: *u*, *es-1* and *fs-1* (see in Fig. 2.2). Variable-length fields *es-1* and *fs-1* encode the number of bits contained in the exponent *e* and fraction *f*, respectively. Thanks to the "uncertainty" bit *u*, the format can also be used for interval arithmetic with values being represented as a bounded pair of two UNUM numbers, an interval. The only sizing limitation of a UNUM is given by the maximum length of *es-1* and *fs-1* fields, known as UNUM environment *(ess, fss)*. Thus, UNUM encoding is characterized as having a variable precision footprint.

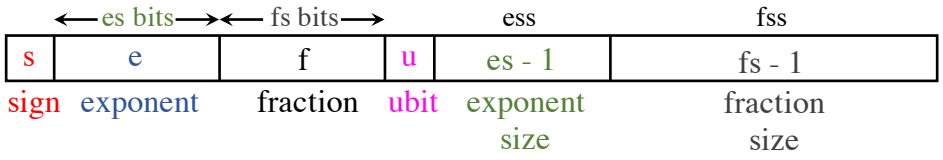


Figure 2.2: The Universal NUMber (UNUM) Format

Hardware accelerators [21, 52] were proposed to facilitate performance comparison between this new format and the IEEE standard. Its amount of flexibility has shown to (1) incur a higher hardware implementation cost when compared to traditional FP units; (2) demand extra, and more complex memory management due to the variable-length capabilities.
Applications that require higher-than-standard precision representations can, nonetheless, still benefit from its use to design solutions with smaller residual error [22].

2.3.3 Posit

Posit [58] was proposed as a simpler, more hardware-friendly alternative to the UNUM format. It uses a fixed-size encoding scheme but still enables variable-length exponent and mantissa fields through tapered accuracy. Along the usual *sign*, *exponent*, and *fraction* fields, posits specifies the *regime bits* fields to allow changes in the size of the exponent field.

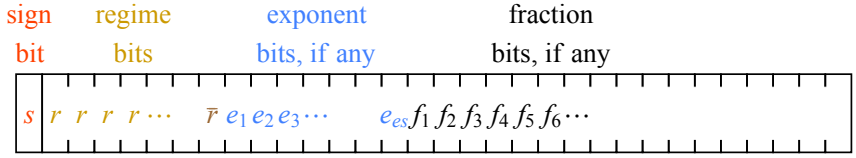


Figure 2.3: The Posit Format

Although UNUM was first proposed as a replacement for IEEE, its hard design cost and complexity has shortly been overruled by this hypothesis. Posit, however, has shown to be a better competitor than UNUM for the IEEE standard. De Dinechin et al. [42] shows uses cases for the posit system in machine learning, some Monte Carlo methods, and graphics rendering. In other situations, posit formats present large degradations of accuracy than the IEEE counterparts. Regardless, authors also assert that, for a new floating-point to displace the current specification, tools, like compilers, must exist and can explore all properties and features of a new format. Yet, compiler support for Posit types are scarce and have not shown to be openly included in any mainstream infrastructure.

2.3.4 New FP Formats from a Compiler's Point of View

The full exploration of IEEE formats was only possible through the integration between hardware and the software stack, and compilers, capable of harnessing all the power of FP units. Having defined formats and proposed their arithmetics do not guarantee their utilization unless powerful tooling is also available. Effective ways to use programming languages are needed to drive novel FP formats. Additionally, the integration of new FP formats with an optimizing compilation flow is paramount for improving their productivity, and making use of FP units [23, 52, 69, 115] that implement them.

Compilers must take into consideration format-specific attributes, and how they can efficiently generate code for formats with different requirements. One must also verify how types are impacted by classical compiler optimizations, as well as the need for new format-specific ones. For instance, UNUM's variable size is a challenge for memory management of data types in compilers, and must not be overlooked.

Recent works evaluated the potential of UNUM and Posit formats in scientific computing [22, 65] as well as machine learning [28, 70]. However, the lack of an integrated compilation flow still hinders the comparison of numerical benchmarks across formats, precision control schemes, and hardware/software implementations. Languages, types, and code generation strategies integrated with state-of-the-art compilers could enable a more thorough investigation of the impact of new formats across the hardware/software stack.

This leads to the following question: How can one extend languages and provide compiler support for new formats taking into consideration their singular properties?

2.4 Variable Precision as a New Paradigm for FP Arithmetic

The rise of new formats has also contributed to a further investigation of real numbers and FP representations in real-life applications. Variable precision (VP) computing is emerging as an alternative computing paradigm for the utilization of real numbers in computer systems. It differentiates from mixed precision [12] by having a finer granularity that exceeds the scope of IEEE formats. VP also offers characteristics similar to arbitrary-precision computing [49, 87] as it is mostly used to address high-precision applications. However, while the latter targets a platform for high-precision representations, VP embeds not only this platform but also a programming scheme to explore formats, precision values, and ultimately, trade-offs between accuracy/precision, and performance/energy. In other words, variable precision encapsulates and generalizes the ideas of mixed precision with finer granularity, and arbitrary-precision computing. In this work, we focus on the exploration of variable precision in high-precision scenarios, although no restrictions are imposed for low precision.

This section walks the reader into the main concepts of VP. Then, it overviews the main challenges imposed to compiler and language designers for the provision of a full-fledged infrastructure for VP exploration.

2.4.1 Problem with Precision Cherry-picking: Numerical Stability and Numerical Accuracy

Most numerical algorithms and techniques for linear algebra are built upon continuous mathematics. The Newton-Raphson's method (from equation [127]), Euler's number calculation, and other mathematical formulas expressed through the Mathematics's *limit of a sequence* term in *convergence* state have no exact representations in computer systems. Instead, their discretization through finite representations adds a layer of complexity to the equation as only approximate results can be computed.

There is no guarantee that those methods are fully compatible with discrete mathematics. In particular, the chosen representation may preclude the convergence of the numerical algorithm, an issue in the heart of **numerical stability**. Iterative methods for solving linear systems, which will be discussed herein, are good examples of techniques where numerical stability may not always be reached. The Gramschmidt method, denoted *gram* in Table 2.1, is an example of a numerically unstable algorithm. In other cases, an algorithm may be convergent but deviates from the expected value due to low **numerical accuracy**. Table 2.1 also shows that applications *3mm* and *covar* have low numerical accuracy for large and extra-large data sets when using IEEE 32 format and, thus, may produce unsatisfactory results for further utilization.

2.4.1.1 Quantifying Errors in Floating Points

Numerical analysis techniques can be employed to remedy issues of stability and accuracy in applications. The idea is centered on analyzing the main sources of errors through *backward error* or *forward error* analysis [64]. *Backward error* is the input error, commonly known as Δx, associated with the approximate solution to a problem. Forward error is the distance between the exact solution of a problem and the produced value. Fig. 2.4a illustrates these relations.

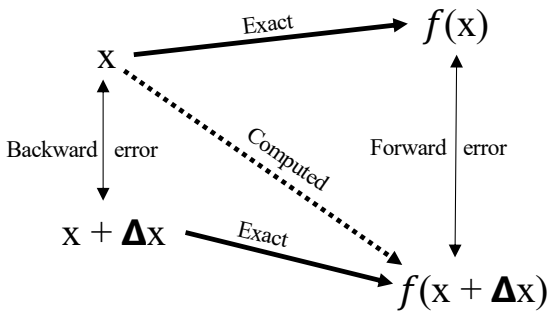
Higham [64] shows that the relationship between backward and forward error is given by:

$$forward\ error \approx conditioning \times backward\ error \tag{2.2}$$

where conditioning represents how a small change in the input reflects in the output. A system is said to be *ill-conditioned* when a small change in the input results in a large change in the output.

The use of *backward* and *forward* error analysis assess the nature of the accuracy and stability problem of applications. Table 2.4b illustrates how they aid numerical analysts in the design of more stable algorithms. For instance, an algorithm with small forward error and large backward error is not sensitive to accuracy, so there is no need to reduce its backward error. On the other hand, a problem is considered ill-conditioned when it has a small backward error and a large forward error. In this case, a new implementation is unlikely to improve stability, and methods to reduce its conditioning factor should be preferred (we will cover this topic in another chapter).

Additionally, an algorithm with large forward and backward errors can potentially improve accuracy with a new implementation that minimizes the backward error. It is likely that the forward error is strongly connected to the backward error of the application. However, not only is it nontrivial to devise a new algorithm for a certain problem, but it may also be difficult to precisely analyze their sources of uncertainties even with the availability of tools to perform it.



(a) Backward and forward errors [64]

Forward Error	Backward Error	Analysis
Small	Small	Problem has a good implementation
Small	Large	Not sensitive to accuracy
Large	Large	May increase accuracy with new implementation
Large	Small	Ill-conditioned problem. Unlikely to improve only with new implementation

(b) Backward and forward error analysis: what could we do?

Figure 2.4: Relationship between backward and forward errors.

2.4.1.2 Augmenting Precision to Remedy Stability

As an easier-employable alternative, numerical stability and accuracy issues can also be solved by augmenting the computation precision in use[5]. This procedure is equivalent to lowering the quantification step, formally known as *unit in the last place (ulp)* or u, between two representable FP values [91]. Although u is directly connected to numerical stability and accuracy, it is often more practical to make use of the relative error ϵ (given by $\frac{f(x+\Delta x)-f(x)}{f(x)}$ or $\frac{\Delta y}{y}$) so as to estimate how numerically stable an algorithm or a solution to this algorithm is.

In the general case, working with a smaller *ulp* automatically leads to better accuracy and stability (although Higham 2002 [64] shows that is not always the case). A question we may want to ask is: which value of u to use if we want to guarantee a minimum relative error ϵ? Or reformulating the question to a computer scientist: which precision can we use to guarantee stability and accuracy?

Considering there is no unique precision value that fits all targeted applications, variable precision (VP) computing can be used as an exploration tool to search for the most suitable solution for the stability of kernels or applications. Equally important, different representations or formats can take advantage of this paradigm, i.e., there might not be a right solution for a single problem. Input data that generate large numbers benefit from wide-range FP formats, while others may not have this same requirement.

Linear algebra algorithms have the potential to take advantage of precision alternation and hint at being an interesting investigation venue for VP approaches and techniques [16].

[5]Numerical analysis and augmenting precision are not mutually exclusive methods of stability/accuracy resolution. They can potentially be used in combination. However, this subject will not be discussed herein.

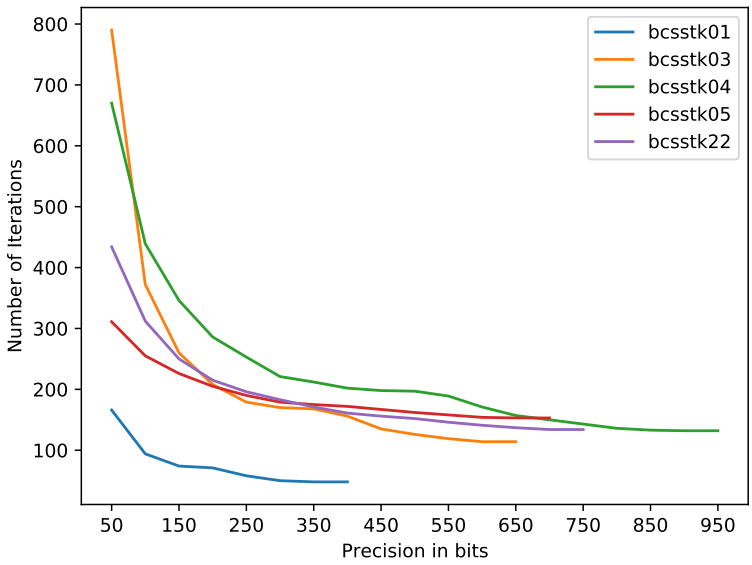


Figure 2.5: When applied on different MatrixMarket [24, 86] matrices, the number of iterations for the conjugate gradient (CG) algorithm decreases when precision augments. This experiment aims to show the usage of variable precision in a real-life application.

2.4.2 Linear Algebra from the Variable Precision Perspective

Linear algebra kernels are the actual working engine underneath many scientific applications. They are widely used in most modern software for physics, molecular chemistry, structural engineering, and many other scientific fields. The most representative kernel is the linear solver, which computes the solution vector x for a linear system given by $Ax = b$. Each solver of this linear system is a delicate trade-off between computing complexity, memory occupancy and numerical stability. Within the last decades, considerable research in that area has provided hundreds of interchangeable libraries which diversely address those three criteria [43]. In some cases, the choice of the appropriate method is left to the scientist. In other cases, he/she may even be left with trial experimentation for selecting the most appropriate method.

Among the algorithms to solve this ubiquitous problem in scientific computing, direct solvers such as Gaussian elimination or Cholesky propose to find the exact solution of a linear system using a finite number of steps/operations, while iterative linear solver methods aim at finding an approximate solution to the problem that stays within a threshold limit. Due to the growth in the size of the linear systems to be solved, the latter have gained importance, and they are now often preferred in many applications due to their low memory occupancy: typically with an $\mathcal{O}(N)$ memory cost rather than $\mathcal{O}(N^3)$ for direct methods.

This improvement comes at the expense of numerical instability. For example, these methods tend to accumulate more round-off errors than their direct counterparts. There are compensation techniques for restoring stability, such as preconditioning, or reorthogonalization, but they may be impractical for the memory cost of computational complexity. Increasing the precision of arithmetic computations can be used as a powerful alternative to address this problem. The impact on the computation is twofold: (1) to accelerate the convergence of the iterative algorithm, and (2) to avoid the need for complex compensations techniques.

The exploration of variable precision through high-precision representations becomes important as applications are not only dependent on the algorithm itself. Input data can also

impact its accuracy and final results. Figure 2.5 shows an example of the conjugate gradient (CG) algorithm, an iterative solver for linear systems, executed for different MatrixMarket [24, 86] matrices. One can observe that input data, and precision impact the number of iterations needed for the algorithm to converge. There is no exclusive value of precision that fits all cases in a common application since the input data also influences the number of iterations needed for convergence. Therefore, the investigation of variable precision is still limited by the availability of hardware, and high-performance libraries for numerical analysts, but perhaps above all, languages and compilers.

2.5 Languages and Data types

Previous sections of this chapter summarize different issues regarding stability, precision, FP representations, and their relation with programming languages, and/or compilers. Section 2.2 describes precision and accuracy, and how it can be cumbersome to verify this relation with state-of-the-art tools. Section 2.3 illustrates some of the FP formats available nowadays and their use, while Section 2.4 covers aspects of numerical stability, and how precision variation could potentially be an asset for linear algebra kernels. It also comes downs to how programmers or numerical analysts can develop algorithms to minimize computational errors in the output, and how programming languages, data types, and compilers can provide the properties needed for this exploration.

A largely relevant reason why languages have not explored VP yet is the recent realization of the potential of variable precision in many fields [16]. Admittedly, researchers have always been keen on precision control. However, the rise of new formats has given further motivation on the matter. With the state-of-the-art apparatuses, varying precision in applications can be achieved in different ways:

(1) Mixed precision [12] offers little flexibility for precision variation with the great advantage of compiler and hardware support for traditional IEEE formats. This greatly increases user productivity with a programming model that is simple and efficient. But it is still limited to the lack of support for other representations, such as UNUM and Posit.

(2) Multi-precision software libraries, such as MPFR [49] and GMP [54], and MPFun [15], yield to higher performance cost than mixed precision, but offer a larger flexibility with a bit-wise mantissa configuration. Due to the programming model imposed by these libraries written in C and Fortran, higher-level programming languages (Julia, Python and C++, among others) provide wrappers to ease their use, with an additional cost in performance.

(3) Other representations have been explored mainly through libraries [82, 83, 84]. There are software implementations for UNUM, and Posit formats, but the lack of integration with a compiler hinders the exploration and performance improvement of these formats in real-life applications.

David Bailey, a well-known mathematician and computer scientist, states, in a recent publication [16], that existing software facilities for variable precision computations are rather difficult to use, particularly for large, computationally demanding applications. Existing languages have no direct syntax and semantics for the variation and dynamic precision. Programmers must rely on high-level language (HLL) abstractions to implement data structures capable of this handling. The composite data type `struct` can be used in C-based languages for that purpose. In objected-oriented languages like C++, Rust and Java, one may implement them as class abstractions. Even dynamically typed languages, such as Julia and Python, would rely on high-level abstractions to express the syntax requirements for variable precision. Programming

languages have yet to offer extensions to handle variable precision capabilities by default. And yet, none of the aforementioned abstractions has a syntax that could benefit from hardware support.

Along with the language support, it is paramount that a specific type (or a type system) be able to drive all properties that come along. Programming languages are not normally equipped with types that allow this flexibility, i.e., variation of the precision in any simple fashion. A concise semantics must allow simple reusability, which is difficult to express using dynamic typing of any kind.

Equal importance should be given to the runtime requirements for dynamic precision code. A feature that clearly favors HLL abstractions is precision genericity, i.e., users can hide the precision value in data structures so that it is only evaluated at runtime. This allows one to write code that is precision-agnostic. By looking back at programming languages and their types, one may notice that this requirement are only possible through the abstractions mentioned previously. In those languages, there are no data types that are able to express this requirement, including how to properly manage the memory of types that are not inherently constant-sized.

Additionally, even the implementation of a type system with these requirements in a lower-lever language would greatly benefit HL languages. Python and Julia are inherently dependent on lower-level languages through code binding. As example, TensorFlow [1] is implemented in C++ and mostly used by Python users. Thus, even HL languages require low-level abstraction for the generation of highly optimized code and libraries.

2.6 Compilers and Optimizations

Compilers and optimizations also play a central role as an interface between programming languages and powerful hardware architecture. Performance improvements in VP computing, and novel FP representations, pass through the integration with an optimized, state-of-the-art compiler infrastructure. IEEE standard formats have long been supported by industry-standard compilers, like the GNU C Compiler [113], and LLVM [76]. Few FP representations or types, however, have seen their support integrated into standard compiler toolchains. Akkaş et al.[5] proposed intrinsic compiler support for intervals in GCC in order to accelerate the execution of interval arithmetic algorithms.

Compilers, similarly to programming languages, struggle with alternative FP formats, missing on the essential, data-flow, control-flow and algebraic optimizations available for IEEE compatible arithmetic. They also miss opportunities to leverage hardware implementations for alternative FP arithmetic. As consequence, the exploration of variable precision, closely associated with these formats, becomes difficult. The numerical analyst is left with the choice among HL abstractions that cannot deliver the expected flexibility and performance, as explained in the previous section.

There has not been any investigation of type systems for variable precision exploration that includes proper compiler specialization. The challenge and implementation complexity are twofold:

(1) At runtime, compilers have similar constraints as programming languages: a variable precision model should allow algorithms to be written in a precision-agnostic model, which implicates in a novel type system not compatible with current architectures. It also compels such a type system to be, in some way, dynamic. This implies a more complex memory allocation strategy, as type declarations may not always infer variables with fixed (and

[5]Except for *binary16* which was only introduced by the IEEE committee in 2008. Compilers nowadays can support `half` type for *binary16*, and Google's `bfloat` data type, both 16-bit representations.

constant) sizes. While this is not a novelty to many languages, its integration with a compiler has not been shown yet.

(2) At compile time, implementations of these type systems must make the most of what is already inside the infrastructure. The myriad of optimizations available should be reused, or revised so that new types can still profit from them.

- Multi-precision FP libraries offer the flexibility to handle many variations of real numbers. However, they introduce performance overheads due to the lack of compiler support. This prevents optimizations easily explored by traditional IEEE formats, namely, constant propagation, loop nest optimizations, inlining opportunities, and many more. Compiler integration and compatibility to classical optimizations can potentially harness the power of these libraries, and eliminate some of these overheads. Practically, this integration imposes an implementation challenge to properly typify their properties. This has yet to be proven in a well-established compiler toolchain.
- As new FP formats emerge, hardware implementations were proposed to drive their arithmetics. Equally important is their interaction with compilers and classical optimizations. New optimizations may also be devised to enable compiler integration, and classical ones may need to be enhanced to handle different scenarios. For example, constant propagation, instruction selection, and register allocation should take into consideration not only the format itself but also the characteristics of the hardware and architecture in use.
- The handling of precision-agnostic code should be largely integrated with classical optimizations so that types with these properties can also be boosted in performance. Additionally, code generators in compilers should grant these types an interface to specialized hardware units, enabling the acceleration of formats with variable footprint.

All these requirements are far-fetched from today's compilers. Variable precision, although can be studied without much of the help of a compiler, would greatly benefit from it. The implementation of compilers that support dynamic type systems is also significant for HL languages. This specialization grants these languages a low-level substrate to which they can be bound, and thus enables a tighter integration with a downstream compilation flow.

Furthermore, there is a complementary aspect of the importance of language and compiler integration: memory utilization. The memory wall [123], speed disparity between outside-chip memory and processor, has long affected the performance of computer systems. From a variable precision perspective that can deeply rely on multi-precision FP libraries, the urge to optimize memory usage is imminent and paramount to performance. The use of high-precision formats incurs an additional memory overhead imposed by high-level data structures used in declarations. For instance, MPFR uses a 32-byte[6] `struct` to represent values, an increase of 16× when compared to IEEE 128-bit format[7]. The lack of specific register allocation support may also augment the use of temporary values and, by consequence, significantly increase memory usage. Language and compiler integration can contribute to mitigating the effect of memory usage on high-precision formats, and this is, in fact, one of the main long-term goals and benefits of this work.

[6] In a 64-bit machine like *x86-64*, where sizes of pointers and integers types are 8 bytes.

[7] The value is calculated without considering the mantissa field. Thus, $32/2 = 16$, since 32 bytes are needed for MPFR, and 2 bytes (1 sign bit and 15 exponent bits) are required for IEEE 128-bit format.

2.7 Conclusion

This chapter presented the main motivations for this work and put into perspective the issues vis-à-vis compiler and language support for alternative formats, high-precision representations, and at last, variable precision computing. Although this thesis does not aim to prove the full potential of variable precision computing, but rather show the means to its exploration, we briefly show that linear algebra is a likely candidate to benefit from it. Better compiler and language integration are needed since the software abstraction still highly depend on library implementations, which are difficult to use, prone to errors, and inefficient if not used wisely.

Hence, this thesis will try to answer questions in two categories:

(1) **Languages, Compilers and Data types**

- How can languages and compilers be used to accelerate and improve the productivity of multi-precision FP libraries? Can one improve the integration between compilers and these libraries to take better advantage of classical compiler optimizations?
- How can one extend languages and provide compiler support for new formats taking into consideration their singular properties?
- What are the compiler and optimizations requirements to support an FP type system with runtime capabilities? How can compilers provide proper memory management for these types?

(2) **Variable Precision Computing**

- Can variable precision serve as an interesting exploration paradigm in the context of numerical algorithms?

The following chapters of this document are intended to answer the questions set out above.

Chapter 3: Programming Languages, Paradigms for FP Computation and exploration Tools

Contents

This chapter will discuss the main state-of-the-art concepts and work that give basis to this thesis. It will be divided into two main categories: (1) computing paradigms that cover the main ideas and approaches for the efficient use of real numbers, and (2) hardware and software solutions within the context of new representations, along with libraries and techniques that contribute to widening the exploration tooling. The former relates to the thesis since it overviews how different paradigms handle precision control, its flexibility, and exploration potential, important to justify our design choices, and implementation decisions of following chapters; while the latter summarizes the main contributions to the exploration of new representations, and how they can potentially be used in a broader context.

3.1 Computing paradigms for floating-point arithmetics

Having representations for real numbers available by themselves is often not enough to provide the functionality or performance/energy needed by the user. In other words, assigning `float`, `double`, or `__float128` types in applications (pseudo)randomly may offer meaningful results, but applications might not reach the expected performance and/or energy; or even worse, applications may execute fast, but results are out of bounds and do not reach expectation. Essentially, users are confronted with trade-offs between accuracy and performance/energy. Besides, indiscriminate use of large-footprint types can have a negative impact on memory usage, and as consequence, performance.

Although computing paradigms for FP arithmetics extends to many approaches (exact computing [87, 126], interval arithmetics [90, 100] and other alternative methods [9, 68], we

will focus on two principal paradigms that lay the foundation of variable precision computing: mixed precision and arbitrary precision.

3.1.1 Mixed precision computing

The mixed precision computing paradigm [12] involves the combination of IEEE FP types in order to provide sufficient output accuracy and compatible hardware support. The paradigm main focus is to find the right balance between IEEE types and numerical stability, prioritizing reduced-precision formats (*binary16* and *binary32*) over higher-precision ones (*binary64*) and harnessing an increased computer power on applications. Its main benefit lies in the performance boost and energy reduction it can achieve, considering that 16-bit and 32-bit operations can execute more than 2× faster than their 64-bit counterparts, and reduce data movement due to small data types.

It has been widely explored in linear solver methods for high-performance computing (HPC) in conjunction with the *iterative refinement* process, a long-standing technique to improve the accuracy of a computed solution by applying a correction factor to the final solution. The approach is based on Newton-Raphson's method [127]:

$$x_{n+1} = x_n - f(x_n)/f'(x_n) \tag{3.1}$$

which enables lower-precision solutions to be refined to the same accuracy as higher precision.

Baboulin et al. [12] present an approach to mixing 32-bit and 64-bit floating-point arithmetic and iterative refinement in direct and iterative methods for solving a linear system of equations. It argues that the reduction of data movement to/from the memory compensates the refinement process, and makes iterative refinement a valuable approach for linear solvers. The combination of these two techniques yields performance results that are comparable to a full single (32 bits) precision solver while still delivering the same accuracy as the double (64-bit) precision implementation.

The reduction of data movement in memory, computations cost in low precision, and suitable accuracy are the principal reasons why mixed precision and iterative refinement are being employed together. The literature is extensive on the exploration of these techniques together. Carson and Higham [29] accelerate GMRES (an iterative solver for linear systems) by iterative refinement employing three levels of precision (16, 32, and 64 bits). Adding another additional level of precision reduces communication with the memory even more, and improves performance even further. GPUs have also been used to explore these techniques [2, 59, 60, 61], confirming their potentials and benefits.

Nevertheless, a major drawback of these approaches stems from the lack of flexibility vis-à-vis FP types in use. They only work with a limiting number of types, at best `binary16`, `bfloat`, `binary32`, and `binary64`, and, thus, hinders the exploration of other representations. Lee et al. [79] goes further by controlling and adapting precision dynamically in FPGAs, which can achieve up to 2–3× speedups over other mixed precision techniques.

3.1.2 Arbitrary precision

Arbitrary precision extends the idea of FP operations to formats that go beyond the scope of the IEEE standard. Unlike mixed precision, it may offer gradual and bit-level customization of exponent and mantissa values, which makes arbitrary precision an excellent tool to explore the impact of precision in applications. In order to offer such flexibility, arbitrary precision toolchains have the downside of being much slower than mixed precision solutions, as they have to resort to software implementation rather than taking advantage of hardware-specific FP units.

```
typedef struct  {
  int  _mpfr_prec;
  int  _mpfr_sign;
  int  _mpfr_exp;
  int  *_mpfr_d;
} __mpfr_struct, *mpfr_ptr, mpfr_t[1];
```

Listing 3.1: MPFR variable type as defined in [49]

```
1  void mat_mult(mpfr_t *matResult, mpfr_t *matA, mpfr_t *matB,
2                unsigned dim1, unsigned dim2, unsigned dim3,
3                unsigned precision) {
4
5      mpfr_t sum;
6      mpfr_init2(sum, precision);
7      mpfr_t tmp;
8      mpfr_init2(tmp, precision);
9      mpfr_set_d(sum, 0.0, MPFR_RNDN);
10
11     for (unsigned i = 0; i < dim1; ++i) {
12         for (unsigned j = 0; j < dim3; ++j) {
13             for (unsigned k = 0; k < dim2; ++k) {
14                 mpfr_mul(tmp, matA[i*dim2 + k], matB[k*dim3 + j], MPFR_RNDN)
       ;
15                 mpfr_add(sum, sum, tmp, MPFR_RNDN);
16             }
17             mpfr_set(matResult[i*dim3 + j], sum, MPFR_RNDN);
18             mpfr_set_d(sum, 0.0, MPFR_RNDN);
19         }
20     }
21     mpfr_clear(sum);
22     mpfr_clear(temp);
23 }
```

Listing 3.2: Usage of the MPFR library in a matrix multiplication example

Ada [78], Fortran (MPFUN [15]), C (MPFR [49] and GMP [54]), and Cuda (Campary [71]) are some of the programming languages for which arbitrary precision libraries were implemented.

3.1.2.1 MPFR Multi-precision library

MPFR [49] is the state-of-the-art multi-precision (or arbitrary-precision) library for floating point computations, allowing bit-wise control of precision values. It was created as an effort to provide floating-point operations with as much precision as necessary, without needing hardware support. Although it makes use of GMP internally, precision values of MPFR variables have the *exact* number of bits requested by the user, which is not true to its counterpart. It uses a regular base-2 notation in the form of $\pm 1 \times mantissa \times 2^{exponent}$, where the value of *mantissa* is only limited by the amount of memory in the system, and *exponent* takes a type according to the machine word (32 or 64 bits), which gives enough dynamic range to represent essentially any real number needed [26].

An MPFR object is of type `__mpfr_struct`, as illustrated in Listing 3.1. Its API follows 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),

```
void mat_mult(unsigned prec, mpfr_float *matResult,
              mpfr_float *matA, mpfr_float *matB,
              unsigned dim1, unsigned dim2, unsigned dim3) {

  mpfr_float::default_precision(prec);
  mpfr_float tmp;
  for (unsigned i = 0; i < dim1, ++i) {
    tmp = 0.0;
    for (unsigned j = 0; j < dim3; ++j)
      for (unsigned k = 0; k < dim2; ++k)
        tmp = matA[i*dimi2 + k] * matB[k*dim3 +j];
      matResult [i*dim3 + j] = tmp;
  }
}
```

Listing 3.3: Usage of the C++ Boost library for Multi-precision in a matrix multiplication example

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 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 3.2 shows the code for a simple matrix multiplication implemented in MPFR. One can notice how verbose MPFR coding is, even for simple algorithms. MPFR's programming model, although simple to use and efficient, is error-prone to memory management, as programmers are in charge of allocating and freeing resources.

Due to its popularity, MPFR has been widely used in high-level language abstractions, such as in the C++ Boost library for multi-precision arithmetic [85], MFPI [101] and CGAL [48], as well as in dynamic languages like Julia [19] and Python [117]. They provide high-level features (through classes and dynamic type systems) that abstract allocations and deallocations away from programmers.

3.1.2.2 C++ Boost for Multi-precision

As an example, listing 3.3 illustrates the implementation of the same matrix multiplication algorithm using the Boost Multi-precision library. Except for the function `default::precision` that sets up the current precision, the compiler handles object allocations and deallocations, and the creation of intermediate temporaries, similar to an ordinary integer or floating-point variable declaration. However, these abstractions draw an additional overhead to the library that can compromise its performance even more.

Compilers have supported and are able to optimize floating-point types (IEEE types in general) for a long time. However, the lack of compatibility between compilers and library-defined FP representations, like MPFR, is still a major challenge. Better integration with compilers would allow faster execution, as well as improve the productivity of these libraries.

3.1.2.3 Dynamic-typed Languages: a Julia example

As an alternative to the high-level abstraction provided through classes, dynamic-typed languages like Julia are more versatile, using a flexible type system that leaves type evaluation until runtime.

```
function mat_mult(matResult, matA, matB, dim1, dim2, dim3)
    for i = 1:dim1
        tmp = zero(typeof(matResult[1, 1]))
        for j = 1:dim3
            for k = 1:dim2
                tmp = matA[i,k] * mat[k, j]
            end
            matResult[i,j] = tmp
        end
    end
end
```

Listing 3.4: Implementation of matrix multiplication example in the Julia language: its dynamic type system hides the types of variables until runtime evaluation

This means that programmers are not obliged to specify the types of variables, and instead, types are only evaluated and check at runtime. Listing 3.4 exemplifies the power of Julia's type system. One may notice that no information type has been given, and the type for each of the variables will only be known at runtime.

As such, Julia's type system has a property that greatly benefits the exploration of new types, and representations: *precision genericity*. Sections 2.5 and 2.6 in Chapter 2 shows that variable precision computing can leverage the use of precision-agnostic code, and hence, enabling a type system that resembles Julia's is highly demanding and beneficial.

3.2 Exploration Tools (Hardware and Software)

Variable precision computing and its applicabilities intersect with different research topics and approaches. Alternative FP formats widen the design space exploration on appropriate representations for real numbers. Many researchers have proposed frameworks and tools to explore the impact of different representations in real-life applications. Complementary, the study of precision-awareness approaches have highly contributed to make accuracy a mainstream constraint in many fields, and it collaborates to give significance to variable precision computing. Additionally, it is paramount that new libraries and hardware architectures drive the variable precision paradigm to compete with existing models.

In the remainder of this chapter, we focus on giving the reader an overview of the state-of-the-art in all these topics. We start by presenting some software for alternative FP formats, including UNUM and Posit tooling. After, we cover methods to enable precision awareness and the detection of numerical errors. An introduction to BLAS and LAPACK is later presented. We conclude by showing hardware implementations used to improve the numerical stability and numerical error in applications, and the compiler or software support proposed for the usage of each unit.

3.2.1 Software for Alternative FP Formats

State-of-the-art compilers like GCC and LLVM have started to include support for a few non-standardized FP formats, from Google's *bfloat16* and Intel's *extended precision* to POWER9 `double-double`. Although proprietary compilers for Posit formats already exists, no much is known about the hardware, ISA or the compiler toolchain available. Instead, many work focus on providing infrastructure for the exploration of alternative formats.

Tagliavini et al. [114] propose FlexFloat, a software library designed to aid the development of applications with fine-grained precision configuration. Unlike MPFR, FlexFloat enables bit-width control of both mantissa and exponent fields, reducing the FP emulation time. It achieves a significant performance speedup in comparison to other FP emulation libraries because it leverages native type support for `float` and `double` types. It later performs a user-transparent sanity check to guarantee that calculated values are representable in the specified type. FlexFloat has no support for high-precision representations, and although it could be extended, it still cannot be used with the same flexibility as MPFR.

Anderson et al. [6] also take a similar approach as FlexFloat by proposing a set of non-standard byte-aligned FP types refer to as *flytes*, that still leverages native IEEE types. The proposed scheme reduces the cost of data transfer volume and storage space requirements by using reduced-precision representations in memory, while it still enables acceleration through existing hardware vector units of general-purpose processors. However, it still suffers from overheads imposed by format conversions, since operations are still highly dependent on IEEE formats. Similar to the previous work, it lacks support for high-precision representations which limits its utilization scope.

There are also work that focus on the implementation of format-specific software libraries. SoftPosit [84] is a software implementation of posit types that conforms to the specification as defined in [58]. It borrows many ideas from the original IEEE-complaint Berkeley SoftFloat library [62], and has been used to compare the accuracy of posit to IEEE types [32]. Software implementation for the UNUM formats had also been proposed [83], but it is certainly less popular than SoftPosit due to the significant complexity of the UNUM representation.

Other researchers have focused on investigating the arithmetic design space of hardware units with alternative numerical representations. Johnson [70] uses tapered encodings from the posit format, logarithmic number system (LNS) [9, 74], and kulisch accumulator [75] to design an accelerator that is effective for convolutional neural networks (CNNs). The work finds inspiration in previous research and shows momentum to explore alternative representation and repurpose ideas.

Lindstrom et al. [81, 82] propose a modular framework for representing real numbers that generalize posit, IEEE and other floating-point number systems. Similarly, Omtzigt et al. [93, 94] present a high-performance number systems library for the exploration of custom number systems, from tapered types to arbitrary floating points, with support for reproducible arithmetic computation in concurrent environments. Additionally, all the recent publications w.r.t. custom formats, their applicability, and hardware implementations show interesting opportunities for extending the role of compilers to novel numerical systems.

Nevertheless, none of the aforementioned work has all properties needed for variable precision exploration. Anderson et al., Tagliavini et al. [6, 114] lack support for high-precision representations, and others [82, 94, 114] provide no hardware support. There is no solution that combines a language scheme for multiple representations, compiler integration capable of software or hardware compatibility.

3.2.2 Precision-Awareness, Auto-Tuning, and Numerical Error Detection

The variety of numerical errors in floating-point formats has often led developers to choose data types that deliver sufficient accuracy but poor performance. For instance, the performance difference between using `float` and `double` can be up to 2× in favor of the former. Precision-aware schemes can be used to detect and debug numerical errors in applications, and auto-tuning techniques propose to find the right compromise between enough accuracy and speed. We overview some of the main schemes and techniques for precision awareness and auto-tuning.

Ansel et al. [7] propose language extensions to expose accuracy choices to the user, allowing the user to incorporate trade-offs between time and accuracy directly at code level. Once accuracy information is set by the programmer in the PetaBricks programming language [8], the compiler and auto-tuner employ a genetic algorithm approach to search valid candidates that meet the desired accuracy. The scheme uses a source-to-source compiler from the PetaBricks language to C++, so its limitation is directly influenced by this integration. Its auto-tuning can be further improved if C++ data structures or types were to allow a fine-grained accuracy configuration. Moreover, this work has widely known as a viable software approach for approximate computing [125].

Darulova et al. [40] presents a programming model, specification language, and compilation algorithm that guarantees to meet the desired precision with respect to real numbers.

Precimonious [102] is a dynamic program analysis tool aimed to assist programmers to tune the precision of FP applications. It relies on user-input configuration files in order to specify which variables in code can be submitted for auto-tuning analysis. The algorithm iteratively searches for suitable sets of types within the specified files and variables that are satisfiable within an error threshold. Precimonious uses a more modern design since it is built on top of LLVM, so it can potentially be used for a wider range of programming languages in different architectures.

Herbie and Herbgrind [96, 105] are two related tools to help developers identify and address root causes of numerical instability in applications. Many works have proposed to use shadow program execution to measure numerical errors in applications. Rubio-Gonzales et al. [103] minimizes Precimonious's analysis time through multiple shadow executions. Chowdhary et al. [33] employ shadow execution to detect and debug errors in posit formats by relying on high-precision values. And more recently, NSan [38] has been proposed as a new sanitizer for locating and debugging floating-point numerical issues, fully integrated into the LLVM sanitizer infrastructure.

All of the work described in the above paragraphs are great examples of how compiler support for alternative formats can be beneficial. The limiting number of FP types natively has a negative impact on the interaction of these work with the compiler.

3.2.3 Software for Scientific Computing Exploration

During the decades before the 2000s, the very active research in computational physics, and others fields, drove the development of standard and efficient FP libraries. Two of the most important software technologies put forward were the design and specification of the Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK).

3.2.3.1 Basic Linear Algebra Subprograms (BLAS)

Mixed precision techniques, as described in 3.1.1, took great advantage of the Basic Linear Algebra Subprograms (BLAS), a set of standardized linear algebra routines that are intended to be reliable, fast, and portable. It specifies the arithmetic computational routines which perform common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. It is further structured in three categories, according to their complexities.

Level 1 defines a set of linear algebra functions that operate on vectors only. These functions share the form:

$$y \leftarrow \alpha x + y \tag{3.2}$$

where α is a scalar and x, y are vectors

Level 2 functions are matrix-vector operations, such as multiple variations of the matrix-vector product operation. These functions share the form:

$$y \leftarrow \alpha Ax + \beta y \tag{3.3}$$

where A is a Matrix, α, β are scalars and x, y are vectors

Level 3 functions are intended for matrix-matrix operations. Typically, these functions perform $\mathcal{O}(N^3)$ operations on $\mathcal{O}(N^2)$ data, therefore the algorithm structure, i.e. its interaction scheme between FPU and memory, is decisive for performance, but it is still very specific to the platform it runs on. Most of these functions share the form:

$$y \leftarrow \alpha A \times B + \beta C \tag{3.4}$$

where A, B, C are matrices, α, β are scalars.

The original BLAS implementation ([77]) is the reference for the arithmetic operations involving vectors and matrices. As a sequel of the original work, several variants have been developed: OpenBLAS [124], Intel Matrix Kernel Library (MKL) [120], and ATLAS [122]. All of these libraries are tailored for standard precision (and support multiple degrees of parallelism), and no high-precision specification of any kind is given.

The design of XBLAS [80] is the first attempt on the design of a representative subset of BLAS routines that internally operate in extended precision (80- or 128-bit representations). Internal extra digits can aid the reduction of accumulated round-off errors and cancellations and can justify the use of slower operations. External operands and function signatures are still similar to classical BLAS routines and are easily adaptable to the proposed library. While many functions, such as *dot products*, *matrix-vector products*, *matrix-matrix multiplications*, can be improved with internal extra precision, others struggle to make them worth using. Particularly, scaling matrix or vector by a constant, vector additions, and computing norms of vectors and matrices have shown no benefit of internal operating digits. From the perspective of adjusting precision dynamically, however, XBLAS still hinders exploration by disallowing multi-precision computation.

MPACK [92] goes further on the development of a true arbitrary/higher-precision linear algebra (MBLAS). The library supports many multi-precision libraries, like MPFR [49], GMP [54], QD [17], as well as IEEE 754 *binary128*, and uses similar function signatures as the traditional BLAS implementations. MPACK is the project that perhaps best captures a multi-precision BLAS, with the significant drawback of neither having full hardware support nor a friendly interface to interact with compiler optimizations.

The BLAS-like Library Instantiation Software (BLIS) [118] framework was proposed as a new infrastructure for rapid instantiation of BLAS functionality. The framework shows that it is possible to express all level-2 and level-3 BLAS operations in terms of a few simple kernels, and thus, it accelerates the instantiation of BLAS-like operations. By isolating BLIS operations to few kernels, BLIS may aid those who wish to auto-tune operations for high performance.

In the previous chapter, we show VP computing as a use case in the context of linear algebra. Hence, a VP-specific BLAS library implementation would also contribute to the study of the trade-off between precision and other constraints (energy, execution time, number of iterations, etc.) in many fields. While this is already achieved with MBLAS from MPACK, its use of high-level structures means there are no support in compilers, or hardware specialization, which translates to significant performance overheads in comparison to other hardware-compatible BLAS libraries.

3.2.3.2 Linear Algebra Package (LAPACK)

The Linear Algebra Package (LAPACK) library is an effort to design and implement high-performance routines for solving linear systems and eigenvalue problems and finding least-square solutions of systems of equations. It also provides matrix factorization algorithms (LU, Cholesky, QR, SVD, etc) that are needed by other routines in the library. LAPACK routines are implemented by relying on BLAS. Block algorithm techniques [44] are employed to improve the locality of matrices and accelerate the code executed.

Other relevant projects associated to LAPACK are the Matrix Algebra on GPU and Multicore Architectures (MAGMA) [116], a LAPACK library for heterogeneous/hybrid architecture, ScaLAPACK [20] that targets distributed-memory platforms, and the Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [27]. MPACK also implements an arbitrary/high-precision library for LAPACK kernels, MLAPACK. However, the library also suffers from the same issues encountered in MBLAS, i.e, lack of hardware support, and no compiler integration.

3.2.4 Characteristics of the Hardware Implementation of Variable Precision FP Units

Compilers and languages can certainly work as enablers to expand the use of new computer architectures. As is the case of variable precision computing, there needs to be coordination between the software, i.e, compilers and languages, and hardware targets. One must take into account the specificities of the hardware in order to augment compiler support. The following section presents three interesting work proposed within the context of variable precision arithmetics. We then highlight their main aspects, and the software interface proposed to enable their usage.

3.2.4.1 Round-off Error Minimization through Long Accumulators

Kulisch et al. [75] proposes a rather orthodox way of error-free calculations for FP arithmetic through long accumulations, known as *kulisch accumulator*. A *kulisch accumulator* is a fixed-point register wide enough to minimize the impact of round-off errors of FP arithmetic operations. It aims to be used in long chains of arithmetic operations, such as *dot products*, in order to avoid values to be rounded when stored back to memory.

Two major issues can be spotted in *kulisch accumulators*: (1) an accumulator can operate only in a single chain of arithmetic operations at a time. Multiple accumulators would have to be added for parallel use. (2) automatic detection of long accumulators by the compiler can be implemented. However, it prevents the user from evaluating which set of operations need additional accuracy, and hence, an unnecessary overhead is introduced. Still, the simple design of a *kulisch accumulator* also inspired further exploration of these long accumulators in fused operations of the posit arithmetic [58], which means that it can be used for multiple formats, and integrated with different approaches.

3.2.4.2 A Family of Variable Precision, Interval Arithmetic Processors

Schulte et al. [107] presents a family of variable precision, interval arithmetic processors for improved accuracy and reduced accumulation of errors. Contrary to previous architecture that uses fixed-point arithmetic and relies on an internal accumulator to minimize round-off errors, this work takes an orthogonal approach of using FP arithmetic, and a register file to allow multiple operations at a time.

The operating mode of these processors depends essentially on the data type declarations provided by the user in the program. Table 3.1 shows some of the supported data types. The software interface is provided by class declarations of each type in a similar fashion as in the

Table 3.1: Some of the data types supported in Schulte et al. [107]

Data Type	Description
`vp_float`	Variable-precision floating point number
`vp_vector`	Variable-precision floating point vector
`vp_matrix`	Variable-precision floating point matrix
`vp_interval`	Variable-precision interval
`vp_ivector`	Variable-precision interval vector
`vp_imatrix`	Variable-precision interval matrix

VPI software package [46], and provides support for arithmetic instructions and operations in scalar types, as well as intervals. Although this solution is able to add specialized hardware support for this family of processors, a major drawback lies in the lack of compiler integration to enable classical optimizations for these types. C++ objects don't provide the necessary low-level abstraction for driving these new types to optimizations.

3.2.4.3 Scalar Multiple-precision UNUM RISC-V Floating-point Accelerator (SMURF)

Scalar Multiple-precision UNUM RISCV-V Floating-point Accelerator (SMURF) [23] is a coprocessor built up on top of a RISC-V [121] Rocketchip architecture [11] that operates in a variable precision fashion. As UNUM is a self-descriptive format, FP values in memory are store using this format, while arithmetics operations execute in a regular base-2 scientific notation form.

An important concept introduced with this coprocessor is the idea of the *Maximum Byte Budget* (MBB), a value that is configured through control status registers and which limits the size of the number stored in memory. The main idea behind MBB is to reduce the memory footprint and memory pressure when dealing with high-precision numbers while maintaining outputs within the same order of magnitude. Additionally, Working G-Layer Precision (WGP) is the nomenclature adopted to the precision used in the coprocessor.

The coprocessor uses a scratchpad as a register file of 32 interval registers. By keeping values within the register file, we are able to significantly reduce memory pressure. Each of these interval registers has two endpoints divided in the header, containing sign, exponent, flags (NaN, ∞, zero, ...), and mantissa which is divided into 64-bit chunks of up to 512 bits (with a total of 8 chunks).

A RISC-V ISA extension, depicted in figure 3.2, was proposed in order to make use of the coprocessor. The supported operations (⑫-⑭) are comparisons, addition, subtraction, multiplication, interval midpoint (GGUESS) and interval radius (GRADIUS). Other operations (e.g. division) are implemented in software. The ISA has three main features: (i) It supports a set of instructions to control status registers for internal operation precision and MBB (mentioned in the previous section),(①-④), etc.; (ii) It supports internal registers copies and on-the-fly conversion among IEEE and gbound formats (⑤-⑪); (iii) and it also has a dedicated Load and Store Unit (LSU) with compatible instructions which handle misaligned memory accesses (⑮-⑱).

The ISA extension proposed to help the integration between software and hardware. However, no compiler specification is given, which prevents complex code execution. Language, compiler and runtime support for the features presented in this work could further improve its use in applications. Moreover, by using the values of MBB and WGP as knobs for the size of variables

Table 3.2: Coprocessor's Instruction Set Architecture

	31 25	24 20	19 15	14	13	12	11 7	6 0
⓪	func7	rs2	rs1	xd	xs1	xs2	rd	opcode
	7	5	5	1	1	1	5	7
①	`susr`	unused	Xs1	0	1	0	unused	CUST
②	`lusr`	unused	unused	1	0	0	Xd	CUST
③	`smbb/swgp/sdue/ssue`	unused	Xs1	0	1	0	unused	CUST
④	`lmbb/lwgp/ldue/lsue`	unused	unused	1	0	0	Xd	CUST
⑤	`mov_g2g`	unused	gRs1	0	0	0	gRd	CUST
⑥	`movll/movlr`	unused	gRs1	0	0	0	gRd	CUST
⑦	`movrl/movrr`	unused	gRs1	0	0	0	gRd	CUST
⑧	`mov_x2g`	#imm5	Xs1	0	1	0	gRd	CUST
⑨	`mov_g2x`	#imm5	gRs2	1	0	0	Xd	CUST
⑩	`mov_d2g/mov_f2g`	#imm5	Xs1	0	1	0	gRd	CUST
⑪	`mov_g2d/mov_g2f`	#imm5	gRs2	1	0	0	Xd	CUST
⑫	`fcvt.x.g/fcvt.g.x`	unused	Xs1	0	1	0	gRd	CUST
⑬	`fcvt.f.g/fcvt.g.f`	unused	Xs1	0	1	0	gRd	CUST
⑭	`fcvt.d.g/fcvt.g.d`	unused	Xs1	0	1	0	gRd	CUST
⑮	`gcmp`	gRs2	gRs1	1	0	0	Xd	CUST
⑯	`gadd/gsub/gmul`	gRs2	gRs1	0	0	0	gRd	CUST
⑰	`gguess/gradius`	unused	gRs1	0	0	0	gRd	CUST
⑱	`lgu/ldub`	unused	Xs1	0	1	0	gRd	CUST
⑲	`stul/stub`	gRs2	Xs1	0	1	0	unused	CUST
⑳	`lgu_next/ldub_next`	gRs2	Xs1	1	1	0	Xd	CUST
㉑	`stul_next/stub_next`	gRs2	Xs1	1	1	0	Xd	CUST

in memory and precision in operations, respectively, the architecture enables a fair degree of variable precision computing.

3.2.4.4 Other (UNUM or Posit) accelerators

Along with the three main hardware units for variable precision computing, several researchers have shown the use of UNUM or Posit accelerators to facilitate comparison across formats. Tiwari [115] proposes a posit-enabled RISC-V core as the first-level replacement for IEEE units. It proposes to reinterpret `float` or `double` types as posit computing, avoiding specific native compiler support for these types. Other examples of accelerators for alternative formats can be found in [52, 69]. The main hurdle for their adoption still relates to the required software stack support: compiler, and language integration, along with scientific computing libraries like those presented in 3.2.3 for fast execution.

3.3 Conclusion

This chapter presented the main state-of-the-art contributions that give basis to the work of this thesis. From one side, we described different computing paradigms for efficiently use of real

numbers, presenting specific libraries and work that have been developed within this context. In the second part, we focused our attention on the principal hardware and software work that covers the exploration of new representations. These previous works present some caveats: while mixed precision is fundamentally supported by compiler and hardware, it lacks the flexibility of arbitrary precision libraries. Other previous works are either standalone solutions to handle precision with no hardware and compiler support or hardware accelerators without compiler support. There is a lack of synergy between software and computer paradigms, and the hardware layer, as well as a flexibility concern that has been poorly addressed so far. This will be the main focus of the following chapters.

Chapter 4: Language and Type System Specifications for Variable Precision FP Arithmetic

Contents

Albeit the IEEE 754 standard have met immense success across diverse areas, some applications require alternative representation to approximate real numbers. Linear algebra solvers, as briefly discussed in 2.4.2, can potentially benefit from high-precision representations. In *direct solvers* algorithms, such as Cholesky and Gaussian elimination, they can reduce the residual error [29, 64] of the output. Besides, they can be even more beneficial for *iterative methods*, preferred for large-sized problems. Increasing the precision of the intermediate residual vector is an effective mean to reduce the number of iterations needed in an *iterative solver*. Chapter 6 will show and discuss the benefits of using high-precision formats with these algorithms.

While a variable precision algorithm may be seen superficially similar to a typical model[1], the estimation and configuration of the precision (in this case, the fractional part) value for part or all of the arithmetic operations is done at the algorithmic level. This estimation is generally complex to compute and may lead to unnecessary over-provisioning of fractional bits. Thus, a

[1]The programming model for C++ Boost Multi-precision is fundamentally similar to an IEEE-enabled implementation, except by the line that sets the precision in use.

practical alternative is to adapt precision dynamically, i.e., instead of computing the necessary precision *a priori*, the algorithm is modified to use an outer loop to systematically check the result for accuracy at predefined points, as depicted in Algorithm 1. If the residual is above a predefined threshold, or if convergence is too slow, the solver increases its internal precision and resumes computation. One should also notice a similar procedure can also be taken to dynamic adjustment of exponent bits.

Algorithme 1: Simplified variable precision algorithm: function is precision-agnostic, i.e., it is not restricted to a single precision value.

```
output = function(precision)
repeat
    if convergence is slow then
        increase speed
    end
    precision = precision + speed
    output = function(precision)
until output accuracy is greater than threshold;
```

Hence, a strong requirement for the software engineering, tools, and more importantly, compiler and languages supporting this variability is:

(1) the language used for variable precision development must be performance-oriented;

(2) the kernel source code must be unique, therefore the programming style must be agnostic of the underlying precision of its variable precision data, like described section 2.5;

(3) the required precision and exponent value depend on the conditioning of data: it may be defined at kernel initiation time or dynamically and gradually increased in the case of adaptive methods;

(4) a variable-precision implementation should be as similar as possible to its original reference algorithm 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 model able to meet these requirements makes variable precision an extension (and a generalization) of mixed precision (3.1.1) and allows a deeper space exploration of precision, exponent, and in general FP representations. In the remainder of this chapter, we will focus on the description of a type system and language definition that target the requirements specified above.

4.1 Syntax

As shown in previous chapters, programming languages struggle with IEEE-alternative formats for FP arithmetic, and as consequence, with the exploration of variable precision arithmetic. The user is left with no choice but to rely on (1) high-level managed languages like Julia [19] or Python [117] whose abstractions and type systems provides a high productivity variable-precision interface but fails to deliver competitive performance, (2) an efficient language like C with reduced flexibility due to manual memory management and calls to specific software libraries

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

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

type:
  unum | mpfr | posit | custom_ieee | ...

exp-info:
  integer-literal | identifier

prec-info:
  integer-literal | identifier

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

Listing 4.1: Backus normal form (BNF) like notation for the `vpfloat` language extension

such as MPFR [49] or GMP [54], or (3) sacrificing much precision control by relying on a mixed-precision paradigm [12] based on IEEE-compatible formats. In any case, significant overheads are encountered: from (1) and (2) performance-wise, and from (3) in terms of flexibility.

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 programming languages in general. Julia, Python and class-based objects in C++ (like Boost) offer underlying data structures to store the value of precision used. MPFR uses a C `struct` to reach the same effect. Nevertheless, FP formats with runtime capabilities cannot be achieved with the semantics of primitive data types. Types `float`, `double`, `long double`, and every other primitive FP type have predefined exponent and mantissa values, permitting the user only to select values statically in a course-grained fashion.

As a way to improve the state of the art on languages and types for variable precision exploration, we propose an extended type system for C-based languages capable of manipulating FP operations with different representations and formats. It provides first-class support to programming with variable precision FP arithmetic, enabling hardware support when available and offering the flexibility of numerical libraries that can operate with multiple precisions. It differs from the current C syntax by introducing a new parametrized type for multiple representations named `vpfloat` that borrows the syntax of C++ `template`.

The new primitive type `vpfloat` is parameterized with attributes to control a given FP implementation, such as its specific format, exponent, precision and/or storage size information. The syntax aims at providing a generic way for different formats to coexist within the same keyword, while also enabling the addition of new formats or representations as they are proposed. Listing 4.1 shows the syntax rules of the `vpfloat` language extension in a Backus Normal Form (BNF) like notation.

Every declaration must provide a *type* attribute which (1) specifies its representation in memory; and (2) if and how many subsequent attributes are needed, along with which information they carry. Attributes are specified in the following order: *type*, *exponent*, *precision*, and *size*. One should notice that, with the exception of *type*, attributes can all be defined with integral constant literals or identifiers. This specific property demonstrates one of the main advantages of `vpfloat` over their counterparts: types are not obliged to be declared to retain unmodified constant exponent and mantissa values. Allowing FP attribute information to be declared

Table 4.1: Comparison of the `vpfloat` type system and FP types, and data structures found in the literature.

Type	Exponent (in bits)	Mantissa or precision (in bits)	Compiler-integrated	Hardware-enabled	High-precision support	Multi-format
`half`	5	11	Yes	Yes	No	No
`bfloat16`	8	8	Yes	Yes	No	No
`float`	8	24	Yes	Yes	No	No
`double`	11	53	Yes	Yes	No	No
`quad`	15	113	Yes	No	Yes	No
`double quad`	19	237	No	No	Yes	No
`FlexFloat`	Variable	Variable	No	Yes	No	No
`Flytes`	Variable	Variable	Yes	Yes	No	No
`__mpfr_struct`	Variable	Variable	No	No	Yes	Yes*
`Boost Multi-prec.`	Variable	Variable	No	No	Yes	Yes*
`vpfloat<...>`	Variable	Variable	Yes	Yes	Yes	Yes+

* Multi-format libraries can be implemented by relying on it
\+ New Formats can be added

with *identifiers*, our language extension supports constant-size and dynamically-sized types, which is always a technical hurdle in unmanaged languages like C, its associated Intermediate Representations (IRs) and Application Binary Interfaces (ABIs).

Table 4.1 illustrates how our type system contrasts with the most common FP types, and data structures found in the literature when considering different constraints, such as compiler integration, hardware capabilities, multi-format support, among others. Our solution is the only one able to check all boxes. It combines the hardware and compiler support provided by primitive types like `float` and `double`, with the multi-level flexibility of MPFR and the Boost Multi-precision library. In the following sections and chapters, we explain how these types are implemented, their semantics and integration with an industry-level compiler infrastructure.

4.2 Semantics

After each declaration has been syntactically analyzed, it is during semantic analysis that the compiler checks declarations to guarantee that FP attributes respect the semantics rules required by the specific type. While some types may hold meaning through their *type* attribute alone, others may require up to three additional attributes to provide meaningful information to the type declaration.

As a practical example, one may use our type system substrate to implement support for the `bfloat16` format. Variables for this format only need to specify the *type* attribute, as information about exponent and precision is embedded within the format's name. Other declarations, such as `unum` and `mpfr`, are parametrisable and require information about the exponent and mantissa, and an optional attribute that may hold the size of the variable's memory footprint. For each different type, attributes shall be interpreted in different ways.

```
typedef struct  {
  int  _mpfr_prec;
  int  _mpfr_sign;
  int  _mpfr_exp;
  int  *_mpfr_d;
} __mpfr_struct, *mpfr_ptr, mpfr_t[1];
```

Listing 4.2: MPFR variable type as defined in [49].

```
1  void axpy100(int N,
2               vpfloat<mpfr, 16, 100> alpha,
3               vpfloat<mpfr, 16, 100> *X,
4               vpfloat<mpfr, 16, 100> *Y) {
5      for (unsigned i = 0; i < N; ++i)
6          Y[i] = alpha * X[i] + Y[i];
7  }
8
9  void axpy256(int N,
10              vpfloat<mpfr, 16, 256> alpha,
11              vpfloat<mpfr, 16, 256> *X,
12              vpfloat<mpfr, 16, 256> *Y) {
13     for (unsigned i = 0; i < N; ++i)
14         Y[i] = alpha * X[i] + Y[i];
15 }
16
17 void vaxpy(unsigned prec, int N,
18            vpfloat<mpfr, 16, prec> alpha,
19            vpfloat<mpfr, 16, prec> *X,
20            vpfloat<mpfr, 16, prec> *Y) {
21     for (unsigned i = 0; i < N; ++i)
22         Y[i] = alpha * X[i] + Y[i];
23 }
```

Listing 4.3: `axpy` benchmark with `vpfloat<mpfr, ...>` type.

4.3 A multi-format type system

As proof of the power and flexibility of our type system, we designed and implemented a full compilation flow supporting the two representations: `mpfr` and `unum`. Even though the reasoning to categorize our type system according to formats is valid, it is also possible to distinguish types in terms of other constraints, such as allocation schemes, and attribute groups. Viewing them from the perspective of allocation schemes, *constant-size types* have a fixed and always-constant allocation size, while *dynamically-sized* ones have a memory footprint which in most cases can only be evaluated at runtime. When classifying types according to their attribute groups, a type can have constant or runtime attributes which means it was declared with *integer-literal* or *identifier*, respectively, as depicted in Listing 4.1. As we progress into the details of our type system, examples are presented to walk the reader into these different categories.

4.3.1 MPFR

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 5.3.1 in chapter 5). As

```
void mat_mult(unsigned prec, vpfloat<mpfr, 32, prec> *matResult,
              vpfloat<mpfr, 32, prec> *matA,
              vpfloat<mpfr, 32, prec> *matB,
              unsigned dim1, unsigned dim2, unsigned dim3) {

  vpfloat<mpfr, 32, prec> tmp;
  for (unsigned i = 0; i < dim1, ++i) {
    tmp = 0.0;
    for (unsigned j = 0; j < dim3; ++j)
      for (unsigned k = 0; k < dim2; ++k)
        tmp = matA[i*dimi2 + k] * matB[k*dim3 +j];
      matResult [i*dim3 + j] = tmp;
  }
}
```

Listing 4.4: Usage of the `vpfloat<mpfr, ...>` in a matrix multiplication example.

presented in 3.1.2.1, an MPFR object is of type `__mpfr_struct` (see Listing 3.1, reproduced in Listing 4.2). MPFR uses an assembly-like programming model that requires operands (destination included) to be pre-allocated and have their precision defined before being used and freed once no longer needed. Moreover, there are no restrictions on a maximum exponent or mantissa used for each declaration.

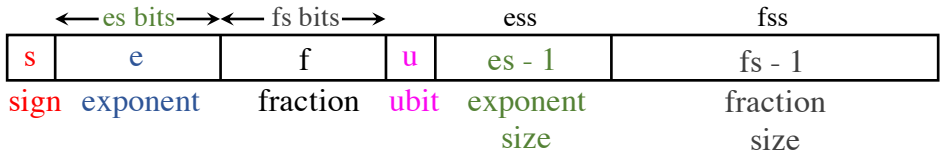
Listing 4.3 shows examples of *AXPY*, a level-1 BLAS [77] routine for vector multiplication, for different MPFR types. Functions `axpy100` and `axpy256` implement *AXPY* with constant-size MPFR type of 100, and 256 bits of mantissa, respectively. Function `vaxpy` show the flexibility of our language extension, illustrating the support for types with runtime attributes 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.

At heart, `vpfloat<mpfr, ...>` types work as a thin C-compatible wrapper for the MPFR library in the fashion as other heavier higher-level ones (like in Julia, Python or C++ Boost for Multi-precision). However, because they are fully integrated at compiler level, more optimization opportunities can be found, thus improving performance. Listing 4.4 shows the `vpfloat<mpfr, ...>` type implementation of the matrix multiplication algorithm found in Chapter 3 for MPFR and Boost for Multi-precision. One may argue that our approach requires some information to be duplicated, particularly, type attributes (`<mpfr, 16, prec>`). While this is true, it guarantees that underlying compiler layers until code generation hold the correct information for each variable type, and thus, justifying this duplication. Furthermore, we can partially resolve duplications with `typedef` declarations: they work for constant types but are rather difficult to express with types that rely on runtime FP attributes.

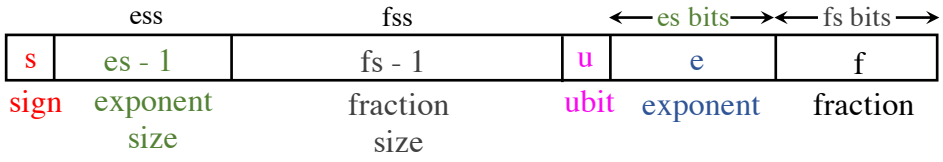
4.3.2 UNUM

While attributes *exp-info* and *prec-info* for MPFR types are enough to be read as *exponent* and *mantissa/precision*, respectively, the UNUM format requires rethinking on how those attributes must be interpreted. As defined in the format specification (see Fig. 4.1a), meta-data information *ess* and *fss* not only act as fields bounding the number of bits of exponent and precision, but also contribute to defining the size of a UNUM value [57]. One should notice that the language syntax does not specify *exp-info* and *prec-info* as the number of bits of exponent and precision, respectively, but rather attributes that relate to them. The UNUM format derives

this information from *ess* and *fss*, which means that the second field of a UNUM declaration holds the *size of exponent* and the third field holds the *size of mantissa*.



(a) Gustafson's Universal NUMber (UNUM) Format



(b) Universal NUMber (UNUM) Format as defined by Bocco et al. [21, 23].

Figure 4.1: Difference between Gustafson and Bocco et al. [21, 23] UNUM formats.

We provide a backend code generator targeting a simplified version of the instruction set architecture (ISA) proposed by Bocco et al. [21, 23]. It implements support for UNUM format in memory, for both scalar and interval-based operations, although the latter has not been considered in our work. Bocco also proposes a modification to UNUM, depicted in Fig. 4.1b, which changes the order of the header fields for a simplification in the decode stage of the coprocessor. 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. Since *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 bytes and is in perfect conformity with the concept of the architecture's Memory Byte Budget (MBB). MBB was primarily design to reduce the growth of UNUM numbers due to *ess* and *fss* fields. The first 6 examples in table 4.2 show that variables declared with no MBB practically double of size for an *fss* increase of one bit. MBB, and consequently, the *size-info* field for UNUM types not only improve and correct that but allow many more format declarations, as depicted in table 4.2.

Declarations with no *size-info* convey that sizes are calculated according to the values of *ess* and *fss*. Particularly, a variable has $(2+2^{ess}+2^{fss}+7)/8$ bytes, where 2 comes from *sign* and *utag* fields from the UNUM format, *ess* and *fss* are specified in the declaration, and 7 and 8 are used to round the values to a multiple of 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 4.2 illustrates different UNUM representations, showing the corresponding values of exponent, mantissa and total size, they may assume. Owing to the variable-length nature of UNUM, the number of mantissa bits directly relates to the number of exponent bits needed for the represented number. For instance, `vpfloat<unum,3,6,6>` may have its precision truncated to 29 bits, *if and only if* the represented number requires 8 exponent bits. Otherwise, more mantissa bits are used. This level of flexibility not only complexifies hardware design but is often misunderstood by users.

The ability of our language extension to handle the UNUM flexibility illustrates its generic nature and runtime capabilities. Listing 4.5 shows the implementation of three functions: (1)

```
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 axpy_UnumDyn(int N, int fss,
                  vpfloat<unum, 4, fss, 18> alpha,
                  vpfloat<unum, 4, fss, 18> *X,
                  vpfloat<unum, 4, fss, 18> *Y) {
  for (unsigned i = 0; i < N; ++i)
    Y[i] = alpha * X[i] + Y[i];
}

void vgemv(unsigned fss, int M, int N,
           vpfloat<unum, 4, fss> alpha,
           double *A,
           vpfloat<unum, 4, fss> *X,
           vpfloat<unum, 4, fss> beta,
           vpfloat<unum, 4, fss> *Y) {
  for (unsigned i = 0; i < M; ++i) {
    // Calls vpfloat_allocation_size(4, fss)
    // to enforce type consistency, see
    // Section III.A.5 "Dynamically-sized Types"
    vpfloat<unum, 4, fss> alphaAX = 0.0;
    for (unsigned j = 0; j < N; ++j)
      alphaAX += A[i*N + j] * X[j];
    Y[i] = alpha * alphaAX;
    // Free stack for 'alphaAX' here
  }
}
```

Listing 4.5: Comparing naïve implementations of AXPY with a constant-size type (`axpy_UnumConst`), a constant-size type with runtime attribute (`axpy_UnumDyn`), and GEMV with a dynamically-sized type (`vgemv`).

AXPY (`axpy_UnumConst`) implemented using a constant-size UNUM type, (2) a second AXPY function (`axpy_UnumDyn`) with constant-size type and a runtime attribute, and (3) the General Matrix Vector Multiplication (GEMV) from BLAS implemented with a dynamically-sized type. One can notice that even constant-size types may not be entirely constant. They can still hold attributes that can be analyzed at runtime, as illustrated in function `axpy_UnumDyn`.

4.3.3 Alternatives Formats

Variable precision computing does not boil down only to the dynamic adjustment of precision. It also encapsulates the tuning of exponent bits and, more importantly, the use of alternative representations. Our language extension offers a general infrastructure that enables multiple FP formats to coexist. The main motivation for the template-like `vpfloat` syntax is to offer a common ground to specify different FP formats.

Figure 4.2 summarizes the multi-format characteristic of our language extension. Two potentially interesting additions are *posit* [58] and a customize IEEE-like format with a bit-wise exponent and mantissa capabilities. Posit has already been proved to be a valuable alternative [42] to IEEE formats in some scenarios, but it is still very far from replacing them entirely.

Table 4.2: 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,3,4>`	8	16	5
`vpfloat<unum,3,5>`	8	32	9
`vpfloat<unum,3,6>`	8	64	11
`vpfloat<unum,3,7>`	8	128	19
`vpfloat<unum,3,8>`	8	256	35
`vpfloat<unum,3,9>`	8	512	67
`vpfloat<unum,3,6,6>`	8	29	6
`vpfloat<unum,4,6,12>`	16	64	12
`vpfloat<unum,3,8,60>`	8	256	60
`vpfloat<unum,4,9,20>`	16	129	20

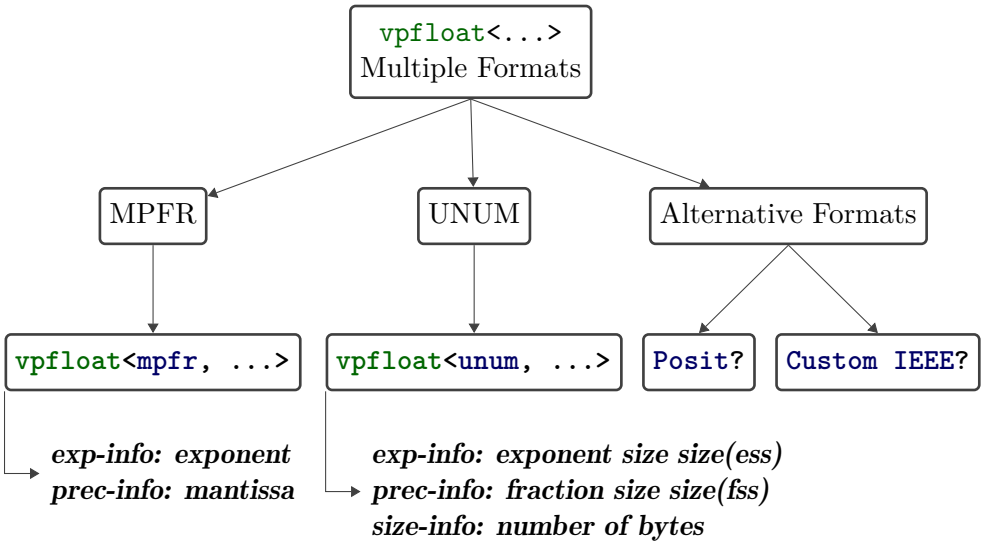


Figure 4.2: Summary of `vpfloat` multi-format schemes.

4.4 Memory allocation schemes

Two distinct allocation schemes are necessary to accommodate the flexibility of our language extension. Viewing from perspective of how memory must be handled, we can divide them in two categories: *constant-size* and *dynamically-sized* types.

As the name suggests, *constant-size* types are declarations with fixed (and constant) memory footprint. They can be further subdivided according to their attribute types and must satisfy one of the three criteria: (1) its variable size can be derived from *exp-info* and *prec-info* attributes known at compile time; (2) variable is declared by specifying all attributes (including *size-info*) known at compile time; or (3) *size-info* attribute is known at compile time. Variables that fulfil criteria (1) or (2) are said to have constant types, while those that meet the requirement of criterion (3) have constant size but with some variability.

Figure 4.3 shows how types are divided according to these criteria. We survey the characteristics and differences among these three groups (*constant types*, *constant-size types* with runtime attribute, and *dynamically-sized types*), highlighting the differences between MPFR and UNUM types if they exist.

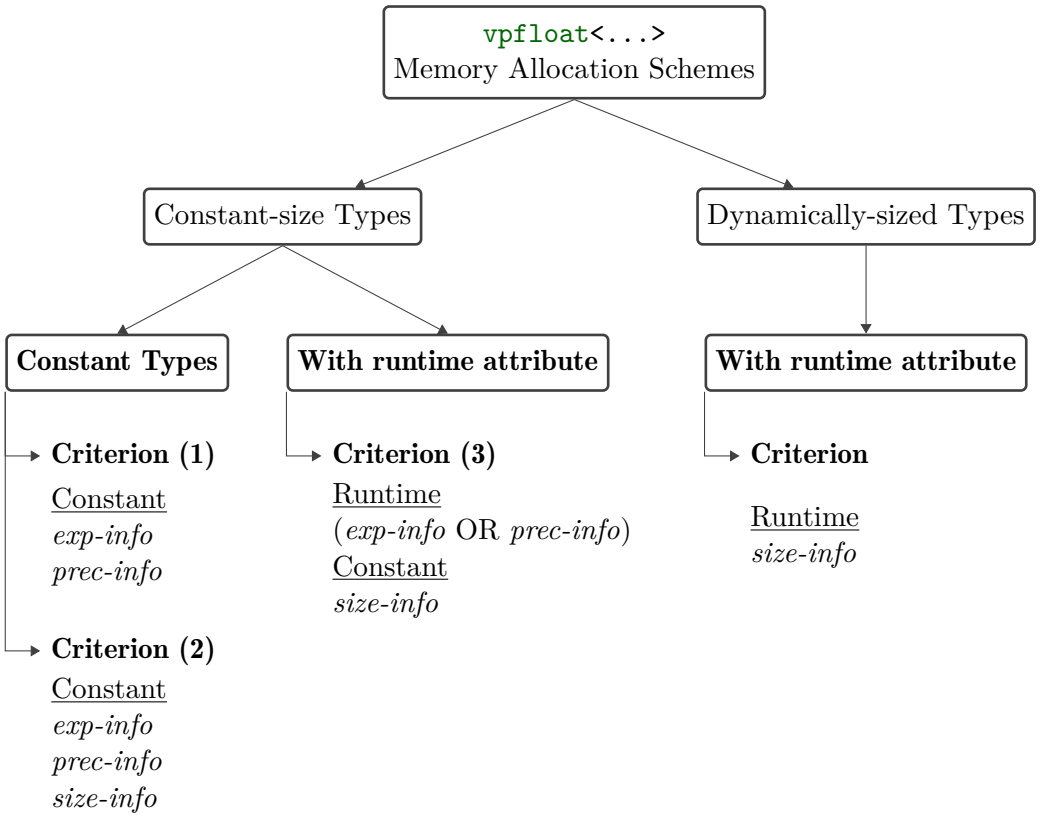


Figure 4.3: Memory Allocation Schemes for Constant-Size and Dynamically-Sized Types

4.4.1 Constant Types

We allow the declaration of constant 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 primitive type variable in C.

4.4.1.1 Representing constants

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 in chunks of 64 bits such that the last chunk always contains the value

Table 4.3: Floating-point literal 1.3 represented with different types

`vpfloat<unum, ...>` or `vpfloat<mpfr, ...>`	Representation of 1.3 (hexadecimal)
`vpfloat<unum,3,6,6>`	0xV001FE999999A
`vpfloat<unum,3,6,8>`	0xV001FE9999999999A
`vpfloat<unum,4,9,20>`	0xV9999999999999A9999
	9999999999990001FFFE
`vpfloat<unum,4,9>`	0xVCCCCCCCCCCCCCCCDCCCCCCCCCCC
	CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
	CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
	CCCCCCCCCCCCCCCCCCCCCCCC4CCCC
	CCCCCCCCCCCCCCC0000FFFF
`vpfloat<mpfr,8,48>`	0xY0FF4CCCCCCCCCD
`vpfloat<mpfr,16,100>`	0xYCCCCCCCCCCCCCCC
	D0FFFF4CCCCCCCC

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 contain the rest of the mantissa. Values are biased according to the maximum exponent value, similar to the IEEE formats.

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 transformation pass, we added a dedicated finalization pass instead to properly set up all UNUM constant literals.

4.4.2 Constant-Size Types with Runtime-Decidable Attributes

Constant-size types can still offer some variability if *exp-info* or *prec-info* attributes are only known at runtime. Its sole shared characteristic with constant types, as described above, comes from the constant memory footprint, which, from a compiler perspective, eases considerably the calculation of addresses for multi-dimensional types, like arrays. Constant creation, runtime verification of attributes, and type conversions are done in the fashion as for dynamically-sized types, which are described below.

4.4.3 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 and is one of the main requirements of variable precision computing exploration: to allow users to programmatically explore multiple configurations of exponent and mantissa in a single run. 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). The runtime aspect of dynamically-sized types in regards to memory management

```
1  int f(int n) {
2
3    int total = 0;
4    for (int i = 1; i < n; ++i) {
5
6      int j;       // Allocation at the beginning
7                   // of the function
8      int arr[i]; // Stack allocation here
9      arr[0] = 0;
10
11     for (j = 1; j < i; ++j)
12       arr[j] = f2(j) + arr[j - 1];
13
14     total = arr[i - 1];
15     // Free stack for 'arr' here
16   }
17   return total;
18 }
```

Listing 4.6: Variable Length Array (VLA) example.

is akin to how Variable Length Arrays (VLA), a feature from the C Standard, are handled by compilers.

A VLA is an array declaration where its number of elements is not known at compile, and is, therefore, evaluated at runtime. The code snippet in Listing 4.6 shows the use of a VLA for an array with n elements. The compiler is not able to infer the size of the array at the time of compilation, for that reason, it generates code that evaluates it 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 the scope of the declaration. In the example, the array declaration at line 8 is alive until the end of the loop at line 15, which means that *arr* is dead (or does not exist) outside of the loop. VLAs are stack-allocated, and their memory locations are only valid within their declaration scope. While VLAs are allocated *on-demand* according to their scope lifetime, compilers usually stack-allocated variables with constant sizes at the beginning of the function. In the example, *arr* allocates stack space within the loop scope while variable j (at line 6) is allocated, if needed, at the beginning of the function. Of course, some optimizations attempt to promote variables to registers and, therefore avoid stack allocation for constant-size types; this is an important category of optimization we would like to leverage in this work (e.g. when targeting the UNUM ISA).

Like in VLAs, dynamically-sized `vpfloat` types can only be declared as local variables and function parameters, and their life cycles also follow those of VLAs. Hence, they are stack-allocated within their declaration scopes since it is not possible to guarantee that all values used as attributes exist at the beginning of the function. 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.

4.4.3.1 Runtime verification

The C Standard, for instance, provides no specification that guarantees VLA arrays are declared with a positive size expression. In practice, the user is free to pass a negative value as the size of a VLA array, leading to undefined behavior, according to the standard. Adopting the same behavior for `vpfloat` declarations eases 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,

```
1  void vpfloat_mpfr_checker(int attributeValue, mpfr_t op) {
2
3    if (op->_mpfr_prec != attributeValue) {
4      fprintf(stderr, "Error: Precision values do not match.\n"
5              "Error: Passed %d to an object with %d precision bits.\n"
6              "Error: Impossible to continue!\n",
7              attributeValue, (int)op->_mpfr_prec);
8      exit(EXIT_FAILURE);
9    }
10 }
```

Listing 4.7: Runtime checker implementation example of `vpfloat<mpfr, ...>` types.

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. Listing 4.7 implements a runtime verifier for the `vpfloat<mpfr, ...>` that checks if MPFR object's precision holds the correct value, i.e., `attributeValue`. One should also notice that all `vpfloat<mpfr, ...>` have already been converted to MPFR, which is why no more reference for these types are presented. Our compiler 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. These calls, although not directly imposed by the language, but rather compiler-generated, are important to ensure correctness from the code, or language, perspective.

4.4.3.2 Function `__sizeof_vpfloat`

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. The function can be generated under two situations: (1) as a replacement to the user-specified `sizeof` function, or (2) when a dynamically-sized type needs memory allocation.

In Listing 4.5, 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. Furthermore, a *mem2reg*[2] optimization can eliminate this allocation inside the loop, since there is no loop dependency in the declaration.

Listing 4.8 shows an example implementation of `__sizeof_vpfloat` function. For `vpfloat<mpfr, ...>`, the only type verification needed concerns the *precision* attribute, which must be a positive value. These types always return the value of `sizeof(mpfr)`, as they will be lowered to MPFR at a later compilation stage. In `vpfloat<unum, ...>` types, values *ess*, *fss*, and *size* are checked according to target ISA. That is, *ess* must have a value between 1 and 4 (lines 19-22), *fss* between 1 and 9 (lines 25-28), and *size* between 1 and 68 (lines 33-36). Additionally, values are guaranteed to reserve bits for the mantissa (lines 39-43) and size is either already specified

[2] *mem2reg* promotes memory references to registers, avoiding the need for memory allocation.

```
int __sizeof_vpfloat(int type, int exp, int prec, int size) {

  int hasError = 0;

  if (type == 1) {
    // MPFR type == 1
    // Only need to check prec > 0
    if (prec <= 0) {
      fprintf(stderr, "precision must be a positive value.\n");
      hasError = 1;
    }
    // And return sizeof(mpfr_t)
    size = sizeof(mpfr_t);

  } else if (type == 0) {
    // UNUM type == 0

    // ess should be between 1 and 4
    if (exp < 1 || exp > 4) {
      fprintf(stderr, "ESS out of bounds (1 <= exp <= 4).\n");
      hasError = 1;
    }

    // fss should be between 1 and 9
    if (prec < 1 || prec > 9) {
      fprintf(stderr, "FSS out of bounds (1 <= fss <= 9).\n");
      hasError = 1;
    }

    if (size) {

      // size should be between 1 and 68
      if (size < 1 || size > 68) {
        fprintf(stderr, "SIZE out of bounds (1 <= size <= 68).\n");
        hasError = 1;
      }

      // Checks for mantissa bits
      if ((size << 3) <= (2 + exp + prec + (1 << exp))) {
        fprintf(stderr, "Format has no mantissa bits. "
                "SIZE(=%d) too small.\n", size);
        hasError = 1;
      }

    } else
      size = (2 + exp + prec + (1 << exp) + (1 << prec) + 7) / 8;

  } else
    fprintf(stderr, "ERROR: Unknown type.\n");

  if (hasError)
    exit(EXIT_FAILURE);

  return size;
}
```

Listing 4.8: `__sizeof_vpfloat` implementation example for `vpfloat<unum, ...>` and `vpfloat <mpfr, ...>` types.

```
1  void example_dynamic_type(unsigned p) {
2
3    vpfloat<mpfr, 16, 256> a;
4    vpfloat<mpfr, 16, 256>  X[10], Y[10];
5    // here initialize a, X and Y here
6
7    vpfloat<mpfr, 16, p> a_dyn;
8    vpfloat<mpfr, 16, p> X_dyn[10], Y_dyn[10];
9    // initialize a_dyn, X_dyn and Y_dyn here
10
11   vaxpy(100, 10, a, X, Y);  // ERROR
12   vaxpy(256, 10, a, X, Y);  // OK
13
14   vaxpy(256, 10, a_dyn,  X_dyn, Y_dyn); // OK if p == 256
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<unum, 4, fss> // OK
21   ex_dyn_type_ret(unsigned fss, vpfloat <unum, 4, fss> a) {
22   a = 1.3;
23   return a;
24 }
25
26 vpfloat<unum, 4, fss> // ERROR
27   ex_dyn_type_ret_error(unsigned p, vpfloat <unum, 4, p> a) {
28   a = 1.3;
29   return a;
30 }
```

Listing 4.9: Uses of dynamically-sized types in function calls and returns.

or calculated accordingly (line 46).

4.4.3.3 Function Parameter and Return

Declaring dynamically-sized types as a local variable is not enough to provide a robust language extension. It is also important to integrate them to function calls through argument passing and parameters, so that is possible to bind function call arguments to function parameters. Listings 4.3, 4.5, and 4.9 show that programmers can also make use of these 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 (runtime) attributes are bound to formal arguments so that (dependent) types from different functions can be passed through function calls. Function `vaxpy` uses parameter `prec` as a bound attribute for *precision* in Listing 4.3. Parameters `alpha`, `X`, `beta`, `Y` in Listing 4.5 have been bound to parameter `fss`, while local variables `a_dyn`, `X_dyn`, and `Y_dyn` in Listing 4.9 have made use of parameter `p` as a runtime attribute.

Function `example_dynamic_type` in Listing 4.9 also shows examples of how these 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 as shown in Listing 4.4.3.1. A compile-time error is raised at line 11, since values of `a`, `X`, and `Y` were created with a constant value of 256, instead of 100. Lines 15 and 18 show examples of how runtime verifications can guarantee correctness between attributes and `vpfloat` declaration in function calls. In line 15, a runtime error is reported back to the user if `p` is not equal to 256, while in line 18 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 delays the creation of the function's return type until all function arguments have been 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.

4.4.3.4 Constants

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 and cast it at runtime to the dynamically-sized type in use. 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. UNUM's maximum configuration comes directly from the target ISA specification that accepts maximum values of *ess* and *fss* as 4 and 9, respectively. The maximum configuration for MPFR was chosen to correspond to the number of bytes of `sizeof(__mpfr_struct)`, that is, 32 bytes.

Additionally, we also enable binding constant values to dynamically-sized types in function calls. For these cases, the compiler is unable to properly infer bindings between runtime arguments and the constant, with function parameters. Hence, users are required to properly cast the specified constant, which means binding the constant accordingly in the function call. Listing 4.10 uses two examples of how function `vaxpy` from Listing 4.3 is being called with `alpha` parameter equals to 1. One can observe that only the first call succeeds as the proper binding for the constant is provided. Furthermore, since variables `x` and `y` in the function have also been declared as `vpfloat<mpfr, 16, precision>`, the same type for `1.0` is required.

4.5 Type Comparison, Casting and Conversion

Types are only considered to be equal if they hold the exact same attributes. Our type system only implements subtyping or implicit conversion in two situations: (1) when doing plain variable assignments, which means we can guarantee that truncation or extension of a source type to its destination can be done safely; (2) and between constant types, when it is possible to guarantee a correct conversion. For instance, constants whose types differ by one attribute. In these cases,

```
1  void someFunction() {
2
3    // Some code
4
5    // OKAY
6    vaxpy(precision, n, (vpfloat<mpfr, 16, precision>)1.0, x, y);
7
8    // ERROR
9    vaxpy(precision, n, (vpfloat<mpfr, 15, precision>)1.0, x, y);
10
11   // Some code
12 }
```

Listing 4.10: Uses of dynamically-sized types in function call and return.

we follow the C language Standard which favors higher-ranked formats, that is, those with larger exponent, mantissa, or size.

Other 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. Listing 4.11 illustrates many of the implicit and explicit situations discussed above. In line 7, the compiler cannot implicitly cast one of the operands of the addition to the other, so the programmer must provide the correct cast for one of the two operands, as in line 10. From line 12 to 25, one can see that the compiler can still offer implicit conversions when no ambiguity is assured. The intermediate type for `b + d`(line 24) can be implicitly casted to type of `d`, due to having two more bits in the *fss* attribute. In line 25, compiler can safely cast the intermediate of `b + e` to the type of `b` as it has one more bit in the *ess* attribute. Nonetheless, ambiguity can still be found in constant types when multiple attributes differ (line 18). In those cases, an explicit cast must be used. After these conversions take place, the compiler can proceed to implicitly cast the intermediate type to the destination type. In lines 24, and 25, for example, a compiler-generated conversion from `vpfloat<unum, 4, 8>` to `vpfloat<unum, 4, 7>`, and from `vpfloat<unum, 4, 6>` to `vpfloat<unum, 4, 7>` are generated.

4.6 Language Extension Limitations

Our extension gives significant flexibility for users to declared FP types with dynamic attributes. Not only we enable plain `vpfloat` declarations, but also their use as underlying types to construct *pointers*, *arrays*, or even *vectors*[3]. However, one current limitation of our language extension lies on the use of types with runtime attributes (from sections 4.4.2 and 4.4.3) within some of the compound types in C-based languages, namely, `struct` and `class`.

From a language perspective, we have not found a concrete syntax that satisfies the same requirements of plain declarations. We could potentially declare a primitive `vpfloat` type inside a `struct` that uses an internal variable as runtime attribute. Although that would be syntactically possible, it leads to an addled situation: `vpfloat`'s allocation size relies on a value initialization inside the struct, which usually happens after `struct` allocation. Limiting the declarations to pointer-based types would help to circumvent this issue. In this case, we could simple restrict pointer `vpfloat` declarations inside these compound types, since their

[3]From the LLVM perspective, there is a difference between *array* and *vector*, as the latter is lowered to *vector*-enabled instructions.

```
void example_conversions(unsigned fss, unsigned fss2) {

  vpfloat <unum, 4, fss> a_fss = 1.3v;
  vpfloat <unum, 4, fss2> a_fss2 = 1.3v;

  // ERROR: invalid operands to binary expression
  vpfloat<unum, 4, 7> add_a_error = a_fss + a_fss2;

  // OK
  vpfloat<unum, 4, 7> add_a = a_fss + (vpfloat<unum, 4,fss>)a_fss2;

  vpfloat<unum, 4, 6> b = 1.3v;
  vpfloat<unum, 3, 7> c = 1.1v;
  vpfloat<unum, 4, 8> d = 1.1v;
  vpfloat<unum, 3, 6> e = 1.1v;

  // ERROR: invalid operands to binary expression
  vpfloat<unum, 4, 7> add_bc_error = b + c;

  // OK
  vpfloat<unum, 4, 7> add_bc_okay = (vpfloat<unum, 3,7>)b + c;

  // OK
  vpfloat<unum, 4, 7> add_bd_okay = b + d;
  vpfloat<unum, 4, 7> add_be_okay = b + e;
}
```

Listing 4.11: Examples of implicit and explicit conversions between `vpfloat<...>` types

allocations can be done after type creation. There have been proposal submissions [41, 111] to the committee to add support for classes with runtime size in C++, showing there is still interest in the community to offer this level of flexibility. Our variable (parametric) floating-point types as presented herein may also help justifying their inclusion in the language. Nonetheless, this is considered part of a future work due to its complexity in compiler instrumentation.

4.7 Libraries for Variable Precision

The language description, its syntax, and semantics presented in this chapter are sufficient to allow users to implement algorithms and libraries that enable FP variation. Having compatibility with two representations (MPFR, UNUM) with language and compiler integration, extends the use of variable precision for algorithmic evaluation. One may want to explore the impact of high-precision representations within linear solvers and singular value algorithms, while others may utilize it to implement physics and engineering applications.

As Basic Linear Algebra Subprogram (BLAS) has been successfully used in linear algebra and other fields to produce fast and portable software (see Section 3.2.3.1), we make use of our proposed multi-format language extension and designed a representative subset of BLAS routines to show the interest of our approach. The selected subset of routines is enough to implement different variations of linear algebra algorithms such as the Conjugate Gradient and the Singular Value Decomposition. In the following sections, we show two versions of BLAS: *mpfrBLAS*, a BLAS library based on `vpfloat<mpfr, ...>` data types, and *unumBLAS* that is implemented with `vpfloat<unum,...>` types. Although they require few changes, the design of these libraries aspires to follow similar models as standard implementations. In particular, our libraries:

(1) implement different flavors for the same algorithm. The goal is to minimize the number of variables in extended (high-) precision footprints. These representations should be used only when necessary and providing multiple implementations aid application designers to choose the right function with the right balance between accuracy and performance. For instance, a variable-precision implementation of CG can bear to have the matrix stored as `double` and its extended-precision representation is not needed.

(2) try to use the same function signatures as standard BLAS implementations. That way, applications may smoothly transition between legacy support libraries (e.g., double-precision BLAS) and extended or adaptive precision solutions when required. Similar to xBLAS [80], our libraries use resembling function signatures to those of legacy code for rapid adaptation.

4.7.1 mpfrBLAS: A `vpfloat<mpfr, ...>` BLAS library

The mpfrBLAS is a `vpfloat<mpfr, ...>`-based BLAS implementation that targets the exploration of variable precision in, but not limited to, high-precision scenarios. No restrictions are imposed on employing it for lower precision, but it has the drawback over other implementations by underlying the use of MPFR. It was implemented to enable execution in multi-threaded environments through the use of OpenMP directives.

The complete list of implemented routines is presented below, divided according to the BLAS level to which they belong. The naming strategy used for them follows the type declarations for matrix and vector variables. For example, a *DOT* function where `y` is a `vpfloat` and `x` is a `double` is named as `vpm_dot_vd`. String "vpm_" is an indicator of a `vpfloat<mpfr, ...>` BLAS routine, while 'vd' indicates that `y` and `x` have `vpfloat` and `double` types, respectively.

4.7.1.1 Level 1: Vector-to-vector operations

– COPY: Copy a vector to a new vector.

$$y_i = x_i$$

```
void vpm_copy(int precision, int n,
              vpfloat<mpfr, 16, precision> *x, int incx,
              vpfloat<mpfr, 16, precision> *y, int incy)

void vpm_copy_2p(int precision1, int precision2, int n,
                 vpfloat<mpfr, 16, precision1> *x, int incx,
                 vpfloat<mpfr, 16, precision2> *y, int incy)
```

Two *COPY* functions are provided: one where vectors `X` and `Y` have the same *precision* value, and a more general implementation where *precision* may be different between vectors. As an example, one may want to increase the precision of a vector for a better estimation of the accuracy of the computed value.

– SCAL: Multiplies each vector element by a constant.

$$x = \alpha x$$

```
void vpm_scal(unsigned precision, int n,
              vpfloat<mpfr, 16, precision> alpha,
```

```
                vpfloat<mpfr, 16, precision> *X, int incx)
```

– DOT: Computes the DOT product between two `vpfloat< mpfr,...>` vectors.

$$r = \sum x_i + y_i$$

```
vpfloat<mpfr, 16, precision>
    vpm_dot_vd(int precision, int n, double *X, int incx,
               vpfloat<mpfr, 16, precision> *Y, int incy)


vpfloat<mpfr, 16, precision>
    vpm_dot_vv(int precision, int n,
               vpfloat<mpfr, 16, precision> *X, int incx,
               vpfloat<mpfr, 16, precision> *Y, int incy)
```

In some cases, it is important to perform accumulation in a high-precision container to minimize the effects of round-off errors. The kulisch accumulator [75] is based on this concept and it is also the reason why `vdot_vd` has been implemented.

– AXPY: Computes vector accumulation.

$$y_i = \sum x_i + y_i$$

```
void vpm_axpy_vdd(unsigned precision, int n, double alpha,
                  double *X, int incx,
                  vpfloat<mpfr, 16, precision> *Y, int incy)

void vpm_axpy_vvv(unsigned precision, int n,
                  vpfloat<mpfr, 16, precision> alpha,
                  vpfloat<mpfr, 16, precision> *X, int incx,
                  vpfloat<mpfr, 16, precision> *Y, int incy)
```

`vpm_axpy*` functions are simple implementations of vector accumulation, which is widely used in linear algebra. More interestingly, `vpm_axpy_vdd` also works as a *copy* function from `double` vector to a `vpfloat` one.

4.7.1.2 Level 2: Matrix-vector operations

– GEMV: General matrix-vector product.

$$y = \alpha Ax + \beta y$$

```
void vpm_gemv_vdd(unsigned precision,
                  enum CBLAS_ORDER order,
                  enum CBLAS_TRANSPOSE trans,
                  int m, int n,
                  vpfloat<mpfr, 16, precision> alpha,
```

```
                    double *A, int k,
                    double *X, int incx,
                    vpfloat<mpfr, 16, precision> beta,
                    vpfloat<mpfr, 16, precision> *Y, int incy)

void vpm_gemv_vdv(unsigned precision,
                    enum CBLAS_ORDER order,
                    enum CBLAS_TRANSPOSE trans,
                    int m, int n,
                    vpfloat<mpfr, 16, precision> alpha,
                    double *A, int lda,
                    vpfloat<mpfr, 16, precision> *X, int incx,
                    vpfloat<mpfr, 16, precision> beta,
                    vpfloat<mpfr, 16, precision> *Y, int incy)

void vpm_gemv_vvv(unsigned precision,
                    enum CBLAS_ORDER order,
                    enum CBLAS_TRANSPOSE trans,
                    int m, int n,
                    vpfloat<mpfr, 16, precision> alpha,
                    vpfloat<mpfr, 16, precision> *A, int lda,
                    vpfloat<mpfr, 16, precision> *X, int incx,
                    vpfloat<mpfr, 16, precision> beta,
                    vpfloat<mpfr, 16, precision> *Y, int incy)
  int t1, t2, t3, t4;
  int lb, ub, lbp, ubp, lb2, ub2;
  register int lbv, ubv;

  if ((m >= 1) && (n >= 1)) {
    lbp=0;
    ubp=floord(m-1,32);
#pragma omp parallel for private(lbv,ubv,t2,t3,t4)
    for (t1=lbp;t1<=ubp;t1++) {
      for (t2=0;t2<=floord(n-1,32);t2++) {
        for (t3=32*t1;t3<=(min(m-1,32*t1+31))-7;t3+=8) {
          for (t4=32*t2;t4<=min(n-1,32*t2+31);t4++) {
            Y[t3*incy] = alpha * A[t3*lda + t4] * X[t4*incx]
                              + beta * Y[t3*incy];
            Y[(t3+1)*incy] = alpha*A[((t3+1)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+1)*incy];
            Y[(t3+2)*incy] = alpha*A[((t3+2)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+2)*incy];
            Y[(t3+3)*incy] = alpha*A[((t3+3)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+3)*incy];
            Y[(t3+4)*incy] = alpha*A[((t3+4)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+4)*incy];
            Y[(t3+5)*incy] = alpha*A[((t3+5)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+5)*incy];
            Y[(t3+6)*incy] = alpha*A[((t3+6)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+6)*incy];
            Y[(t3+7)*incy] = alpha*A[((t3+7)*lda)+t4]*X[t4*incx]
                             + beta*Y[(t3+7)*incy];
          }
        }
        for (;t3<=min(m-1,32*t1+31);t3++) {
          for (t4=32*t2;t4<=min(n-1,32*t2+31);t4++) {
            Y[t3] = alpha * A[t3*lda + t4] * X[t4*incx]
                              + beta * Y[t3*incy];
          }
        }
      }
```

```
    }
  }
}
```

Above, we show an implementation example for `vpm_gemv_vvv`, automatically generated by Pluto [25], an automatic parallelizer and locality optimizer. The only modifications done concern the inclusion of variables `lda`, `incx`, and `incy`, requirements for BLAS functions. We will further discuss support for Loop Nest optimizer within the context of our types (in Chapter 5), however, Pluto's capabilities go beyond the support for types. That is, it optimizes the data locality by merely analyzing patterns of access, without requiring to have previous knowledge about the footprint of the type. This example intent is to show that state-of-the-art optimizers can be used to design high-performance libraries for `vpfloat` types. Future implementations of our libraries will focus on these specializations, which require substantial work to guarantee performance improvement in most cases.

– SYMV: Sparse matrix vector product.

$$y = \alpha Ax + \beta y$$

```
void vpm_sparse_gemv_vdd(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         double *A, int *rowInd, int *colInd,
                         double *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)

void vpm_sparse_gemv_vdv(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         double *A, int *rowInd, int *colInd,
                         vpfloat<mpfr, 16, precision> *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)

void vpm_sparse_gemv_vvv(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         vpfloat<mpfr, 16, precision> *A,
                         int *rowInd, int *colInd,
                         vpfloat<mpfr, 16, precision> *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)
```

– Sparse MV: Sparse matrix vector product.

$$y = \alpha Ax + \beta y$$

```
void vpm_sparse_symv_vdd(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         double *A, int *rowInd, int *colInd,
                         double *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)
```

```
void vpm_sparse_symv_vdv(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         double *A, int *rowInd, int *colInd,
                         vpfloat<mpfr, 16, precision> *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)
```

```
void vpm_sparse_symv_vvv(unsigned precision, int m, int n,
                         vpfloat<mpfr, 16, precision> alpha,
                         vpfloat<mpfr, 16, precision> *A,
                         int *rowInd, int *colInd,
                         vpfloat<mpfr, 16, precision> *X,
                         vpfloat<mpfr, 16, precision> beta,
                         vpfloat<mpfr, 16, precision> *Y)
```

We have implemented different variations of the matrix-vector multiplications in order to handle different scenarios: dense, sparse and symmetric sparse matrices. Those with a *sparse* prefix are used for sparse matrices represented in the CSR (Compressed Row Storage) format.

4.7.1.3 Level 3: Matrix-matrix operations

$$C = \alpha AB + \beta C$$

```
void vpm_gemm(unsigned precision, enum CBLAS_ORDER order,
              enum CBLAS_TRANSPOSE transA,
              enum CBLAS_TRANSPOSE transB, int m, int n, int k,
              vpfloat<mpfr, 16, precision> alpha,
              vpfloat<mpfr, 16, precision> *A, int lda,
              vpfloat<mpfr, 16, precision> *B, int ldb,
              vpfloat<mpfr, 16, precision> beta,
              vpfloat<mpfr, 16, precision> *C, int ldc)
```

GEMM has, as have other routines, been implemented so that it has a signature close to BLAS original one. The sole difference is the presence of a `precision` variable as the first function parameter. The remaining ones conform to the legacy code signature [4], easing portability. Furthermore, both transpose and non-transpose matrices are supported by our implementation.

Our *mpfrBLAS* is composed of 17 routines, ranging from simple vector-vector operations (Level 1) to a matrix product (Level 3). Even though there are many functions missing, these functions already allow us to implement some linear algebra algorithms, which will be discussed in Chapter 6. Moreover, we should clarify that the *exponent* field, defined as a constant `16`, has no real effect on MPFR types. Its actual size value is predefined by the underlying `mpfr_t struct`.

[4] https://developer.apple.com/documentation/accelerate/1513282-cblas_dgemm

4.7.2 unumBLAS: A `vpfloat<unum, ...>` BLAS library

We also propose a subset of BLAS routines for `vpfloat<unum, ...>` types. Due to the flexibility of our extension, many variations of `vpfloat<unum,...>`-enabled libraries can be created. This one, in particular, has been implemented considering all FP attributes being expressed as variables, but other implementations may consider different constraints and use *constant* value for the declaration of any FP attribute. Function names are preceded by prefix "vpu_", while 'v' indicates `vpfloat<unum, ...>` types, 'd' is used for `double`.

4.7.2.1 Level 1: Vector-to-vector operations

– COPY: Copy a vector to a new vector.

$$y_i = x_i$$

```
void vpu_copy(int ess, int fss, int size, int n,
              vpfloat<unum, ess, fss, size> *x, int incx,
              vpfloat<unum, ess, fss, size> *y, int incy);
```

– SCAL: Multiplies each vector element by a constant.

$$x = \alpha x$$

```
void vpu_scal(int ess, int fss, int size, int n,
              vpfloat<unum, ess, fss, size> alpha,
              vpfloat<unum, ess, fss, size> *X, int incx);
```

– DOT: Computes the DOT product between two `vpfloat< mpfr,...>` vectors.

$$r = \sum x_i + y_i$$

```
vpfloat<unum, ess, fss, size>
    vpu_dot_vd(int ess, int fss, int size, int n,
               double *X, int incx,
               vpfloat<unum, ess, fss, size> *Y, int incy);

vpfloat<unum, ess, fss, size>
    vpu_dot_vv(int ess, int fss, int size, int n,
               vpfloat<unum, ess, fss, size> *X, int incx,
               vpfloat<unum, ess, fss, size> *Y, int incy);
```

– AXPY: Computes vector accumulation.

$$y_i = \sum x_i + y_i$$

```
void vpu_axpy_vdd(int ess, int fss, int size, int n,
                  double alpha, double *X, int incx,
                  vpfloat<unum, ess, fss, size> *Y, int incy);
```

```
void vpu_axpy_vvv(int ess, int fss, int size, int n,
                  vpfloat<unum, ess, fss, size> alpha,
                  vpfloat<unum, ess, fss, size> *X, int incx,
                  vpfloat<unum, ess, fss, size> *Y, int incy);
```

4.7.2.2 Level 2: Matrix-vector operations

– GEMV: General matrix-vector product.

$$y = \alpha Ax + \beta y$$

```
void vpu_gemv_vdd(int ess, int fss, int size,
                  enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE trans,
                  int m, int n, vpfloat<unum, ess, fss, size> alpha,
                  double *A, int lda, double *X, int incx,
                  vpfloat<unum, ess, fss, size> beta,
                  vpfloat<unum, ess, fss, size> *Y, int incy);

void vpu_gemv_vvd(int ess, int fss, int size,
                  enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE trans,
                  int m, int n, vpfloat<unum, ess, fss, size> alpha,
                  double *A, int lda,
                  vpfloat<unum, ess, fss, size> *X, int incx,
                  vpfloat<unum, ess, fss, size> beta,
                  vpfloat<unum, ess, fss, size> *Y, int incy);

void vpu_gemv_vvv(int ess, int fss, int size,
                  enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE trans,
                  int m, int n, vpfloat<unum, ess, fss, size> alpha,
                  vpfloat<unum, ess, fss, size> *A, int lda,
                  vpfloat<unum, ess, fss, size> *X, int incx,
                  vpfloat<unum, ess, fss, size> beta,
                  vpfloat<unum, ess, fss, size> *Y, int incy);
```

– Sparse MV: Sparse matrix vector product.

$$y = \alpha Ax + \beta y$$

```
void vpu_sparse_gemv_vdd(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         double *A, int *rowInd, int *colInd,
                         double *X, vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);

void vpu_sparse_gemv_vvd(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         double *A, int *rowInd, int *colInd,
                         vpfloat<unum, ess, fss, size> *X,
                         vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);
```

```
void vpu_sparse_gemv_vvv(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         vpfloat<unum, ess, fss, size> *A,
                         int *rowInd, int *colInd,
                         vpfloat<unum, ess, fss, size> *X,
                         vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);
```

– SYMV Sparse: Sparse Symmetric matrix vector product.

$$y = \alpha Ax + \beta y$$

```
void vpu_sparse_symv_vdd(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         double *A, int *rowInd, int *colInd,
                         double *X, vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);
```

```
void vpu_sparse_symv_vvd(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         double *A, int *rowInd, int *colInd,
                         vpfloat<unum, ess, fss, size> *X,
                         vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);

void vpu_sparse_symv_vvv(int ess, int fss, int size, int m, int n,
                         vpfloat<unum, ess, fss, size> alpha,
                         vpfloat<unum, ess, fss, size> *A,
                         int *rowInd, int *colInd,
                         vpfloat<unum, ess, fss, size> *X,
                         vpfloat<unum, ess, fss, size> beta,
                         vpfloat<unum, ess, fss, size> *Y);
```

4.7.2.3 Level 3: Matrix-matrix operations

$$C = \alpha AB + \beta C$$

```
void vpu_gemm(int ess, int fss, int size, enum CBLAS_ORDER order,
              enum CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB,
              int m, int n, int k, vpfloat<unum, ess, fss, size> alpha,
              vpfloat<unum, ess, fss, size> *A, int lda,
              vpfloat<unum, ess, fss, size> *B, int ldb,
              vpfloat<unum, ess, fss, size> beta,
              vpfloat<unum, ess, fss, size> *C, int ldc);
```

unumBLAS routines are very similar to the ones defined in *mpfrBLAS*. However, one may notice that UNUM types have broader adaptability. MPFR types are declared with two FP

attributes, one of which has no effect on the representation. Types created as `vpfloat<unum, ...>` are more flexible with up to three attributes to control.

4.8 Conclusion

This chapter introduced and presented all aspects of the C language extension and type system for variable precision FP computing that we propose. When comparing to state-the-art solutions, we hope to have proven that our language extension has similar capabilities and equivalent programming model as traditional formats (like programming with IEEE formats), with better language integration than high-level structures. This enables further integration with a compiler toolchain, which will be introduced in the next chapter. We also allow multiple formats to coexist within the same keyword, which can further improve the exploration of alternative formats and can accelerate the integration with industry-level compiler infrastructures.

Along with the extension, we also provide implementations for a subset of BLAS routines that uses our types. They allow developers to rapidly instantiate code for testing multiple formats and variable precision. Furthermore, the main feature of these libraries is enabling programmatically driven experimentations in an attribute agnostic fashion for a given representation. That is, we may single-run applications in multiple precision configurations without requiring code recompilation. Some of the experimental results from Chapter 6 were obtained this way.

Chapter 5: Compiler Integration for Variable Precision FP Formats

Contents

We will demonstrate the value and realism of our language extension through its integration into an industry-level compiler. Our solution was implemented on top of the LLVM [76] infrastructure, since it provides a modular compilation flow, a powerful Intermediate Representation (IR), and is widely used both in academia and industry. This chapter focuses on detailing the compilation flow needed to support the `vpfloat` type system. We will cover the aspects of the proposed LLVM IR extension for variable precision FP arithmetic, and how it interacts with the compilation flow and optimizations. Later we describe the backend code generators we implemented in order to consume `vpfloat<mpfr, ...>` and `vpfloat<unum, ...>` types from our language extension.

5.1 Frontend

We have implemented support for the `vpfloat` language extension and type system in LLVM's frontend for C/C++ (clang). Our frontend supports both representations MPFR and UNUM as defined in the Language chapter (4). Aside from implementing the specification of our language, semantic and syntax analysis, the only work needed by this stage is the generation of LLVM Intermediate Representation (LLVM IR).

5.2 Intermediate Representation (IR)

LLVM uses a target-independent intermediate language for a common infrastructure where code optimizations and transformations can be applied. The LLVM IR defines a set of assembly-like instructions that operate on values and types, and which are later used by the code generators of different architectures. More specifically, LLVM defines a set of types to represent different

```
1  class VPFloatType : public Type {
2
3  public:
4
5  enum VPTypeID {
6    UNUMTyID = 0,      ///<  0: Unum Type
7    MPFRTyID           ///<  1: MPFR Type
8  };
9
10 private:
11
12   // Code ...
13   ConstantInt *ConfigTy;
14   Value *ExponentInfo;
15   Value *PrecisionInfo;
16   Value *SizeInfo;
17   // Code ...
```

Listing 5.1: Partial implementation of `vpfloat` types in the LLVM IR.

data structures, from simple primitive types like integers and floating points, to vectors, arrays and structures.

As such, we provide an extension to the LLVM IR Type System that enables `vpfloat` FP types in intermediate code. Similar to our language-level type system, `vpfloat` IR types also keep the information on attributes internally, so that expressiveness is maintained across the compilation process. We essentially translate the functionalities from language level to intermediate, that is, even closer to target code generators.

5.2.1 VPFloat Types

A new class for `vpfloat` was implemented to provide LLVM with a native representation of variable precision FP types. Attributes are defined as `Value`[1] objects since they can be represented by instructions, constants, function parameters, or global variables. In Listing 5.1, we show a snippet of the implementation of `VPFloatType` class for UNUM and MPFR types, here represented by constants *0* and *1*, respectively. With the exception of `ConfigTy` declared as an integer constant (line 13), all other attributes can assume values of different `Value`-inherited classes (lines 14 to 16), and help to show the flexibility of our type system.

The addition of a new type system, although somewhat intrusive from a maintenance perspective, allows us to have a tighter integration with LLVM. This favors its use not only on novel optimizations we may wish to implement but also qualifies these new types to be used in classical ones. Constant propagation, stack-to-register promotion, and common subexpression elimination are some of the optimizations that work out-of-the-box in our type system without any necessary modification.

Additionally, the human-readable representation of our IR `vpfloat` types resembles that of the language extension. Listing 5.2 shows the the IR code of the `ex_dyn_type_ret` function from Listing 4.9. UNUM VPFloats in the IR are specified as *0*, while MPFR use a constant of *1* for its format. This non-optimized (`-O0` compilation flag) version also illustrates the dynamic aspect of our types. A `call` to function `__sizeof_vpfloat` in line 10 is used to get the number of bytes needed by the allocation instruction `alloca` (line 13). We also make use of a pair of

[1] `Value` is one of the most important LLVM IR classes because it is the base class for all values computed in a program.

```
1  ; Function Attrs: noinline nounwind optnone
2  define vpfloat<0, 4, %fss>
3      @ex_dyn_type_ret(i32 signext %fss, vpfloat<0, 4, %fss> %a) #0 {
4  entry:
5    call void @llvm.vpfloat.runtimeattr.mark.i32(i32 %fss)
6    %fss.addr = alloca i32, align 4
7    %saved_stack = alloca i8*, align 8
8    store i32 %fss, i32* %fss.addr, align 4
9    %0 = load i32, i32* %fss.addr, align 4
10   %1 = call i64 @__sizeof_vpfloat(i32 0, i32 4, i32 %0, i64 0)
11   %2 = call i8* @llvm.stacksave()
12   store i8* %2, i8** %saved_stack, align 8
13   %vla = alloca i8, i64 %1, align 8
14   %a.addr = bitcast i8* %vla to vpfloat<0, 4, %fss>*
15   store vpfloat<0, 4, %fss> %a, vpfloat<0, 4, %fss>* %a.addr, align 8
16   store vpfloat<0, 4, %fss> fpext (double 1.300000e+00 to vpfloat<0, 4, ↩
       %fss>), vpfloat<0, 4, %fss>* %a.addr, align 8
17   %3 = load vpfloat<0, 4, %fss>, vpfloat<0, 4, %fss>* %a.addr, align 8
18   %4 = load i8*, i8** %saved_stack, align 8
19   call void @llvm.stackrestore(i8* %4)
20   ret vpfloat<0, 4, %fss> %3
21 }
22
23 declare extern_weak i64 @__sizeof_vpfloat(i32, i32, i32, i64)
```

Listing 5.2: IR code of the `ex_dyn_type_ret` function from Listing 4.9 types in the LLVM IR.

instructions `store` and `load` to control the address pointer of the stack. Lines 11-12 and 18-19 are used to record and recover the status of the stack, respectively, so that it can be dynamically allocated. To the best of our knowledge, this is the first work that proposes these levels of interaction between types and values in the program for FP arithmetic. LLVM types are usually self-sufficient, i.e., free from the interference of other values in the program (except for VLAs). This flexibility comes at the cost of evaluating their compatibility with the significant number of optimizations LLVM offers.

5.2.2 Function Declarations

LLVM IR has a particular way of representing function declarations that omits variables names for function parameters. As an example, `__sizeof_vpfloat` from Listing 5.2 (line 23) is printed as a function declaration, because it was either inserted by the compiler automatically or came from an included header file. One should notice that only parameter types are included, and not their names, simply because they are not needed.

Declaration-wise, parameters' names are irrelevant. Although this works well for standard LLVM IR code and also conforms to the C specification (as in line 3 of Listing 5.3a), our type system cannot afford the use of the same approach. Types with runtime-decidable attributes may have their declarations dependent on function parameters, which can lead these types to be invalid. As values are used as binding operands for `vpfloat` types, it is important to make sure that function declarations have `vpfloat` types with proper attributes.

We propose to address this issue by using negative values as a way of expressing binding relations between parameters and `vpfloat` types. Since negative values have no meaning for floating-point format parameters whatsoever, we can make use of them to specify these relations. We depict some examples of this approach in Listing 5.3: with Listing 5.3a showing the C-level code for different function declarations, and Listing 5.3b having their corresponding IR

```c
double example1_double_declaration(double a);

double example2_double_declaration(double);

vpfloat<unum, 4, 6> example_const_declaration(vpfloat <unum, 4, 6> a);

vpfloat<unum, 4, fss>
  example1_declaration(unsigned fss, vpfloat <unum, 4, fss> a);

vpfloat<unum, ess_fss, ess_fss, size>
  example2_declaration(unsigned ess_fss, unsigned size,
                       vpfloat <unum, ess_fss, ess_fss, size> a);

vpfloat<unum, ess, fss, size>
  example3_declaration(unsigned ess, unsigned fss, unsigned size,
                       vpfloat <unum, ess, fss, size> a);
```

(a) C-level function declarations for `vpfloat` types with runtime-evaluated attributes.

```
declare double @example1_double_declaration(double) #2

declare double @example2_double_declaration(double) #2

declare vpfloat<0, 4, 6> @example_const_declaration(vpfloat<0, 4, 6>) #2

declare vpfloat<0, 4, -1> @example1_declaration(i32 signext, vpfloat<0, 4,↩
    -1>) #2

declare vpfloat<0, -1, -1, -2> @example2_declaration(i32 signext, i32 ↩
   signext, vpfloat<0, -1, -1, -2>) #2

declare vpfloat<0, -1, -2, -3> @example3_declaration(i32 signext, i32 ↩
   signext, i32 signext, vpfloat<0, -1, -2, -3>) #2
```

(b) IR-level function declarations for `vpfloat` types with runtime-evaluated attributes.

Listing 5.3: Function declaration examples for `vpfloat` types with runtime-evaluated attributes in C headers and LLVM IR.

function declarations. The first 3 function declarations illustrate the current LLVM IR support for constant types, where parameter names are not shown and types require no additional (value-related) information. The remaining three examples display binding relations between *integer* parameters and `vpfloat` type declarations. Negative value *-1* binds the attribute to the first function parameter, *-2* to the second, and so on. This allows semantic analysis to check that function calls match the function signature for declarations.

5.2.3 Interaction with Classical Optimizations

A new IR type system allows `vpfloat` types to interact and benefit from classical optimizations available in the infrastructure. While many optimizations are out-of-the-box compatible with `vpfloat`, some require modifications for proper support. The following sections cover the necessary changes in toolchain and optimizations.

5.2.3.1 Type-value Relation

Standard compilers usually use specific data structures to track the relation between a value definition and its uses [4]. A *Use-Definition (use-def) chain* consists of a *use* (U) of a variable and all the *definitions* that can reach U directly without any other intermediate definition. It usually expresses the assignment of a value to a variable. Its counterpart, *Definition-Use (def-use) chain* relates the definition (D) of a variable with all its uses that are reachable from D without any intervening definition. These concepts and structures are prerequisites for applying many compiler optimizations and transformations, such as constant propagation, register allocation, and common subexpression elimination.

LLVM also makes use of these concepts to construct relations between value definitions and uses throughout the compilation flow. Although LLVM's *def-use chain* and *use-def chain* implementations allows one only to track chains between `Value` objects, `vpfloat` types also have *def-use* relations with attribute `Value` objects. Hence, the current LLVM implementation is incompatible with our type extension, and a `vpfloat` type cannot be obtained by traversing the def-use chain of an attribute. Due to the required modifications of many key components of the compiler for making the current *def-use* implementation compatible with our type system, we have chosen to keep a separated list of `Value` objects being used as types attributes. One should notice that constant values do not require tracking, since they will never change. Once the list is constructed, two operations are possible:

(1) Update: optimizations may optimize values in the program that require the update of references from our tracking list. If an object is replaced by a new one, our type system makes sure to update any type that uses the old value to the new reference.

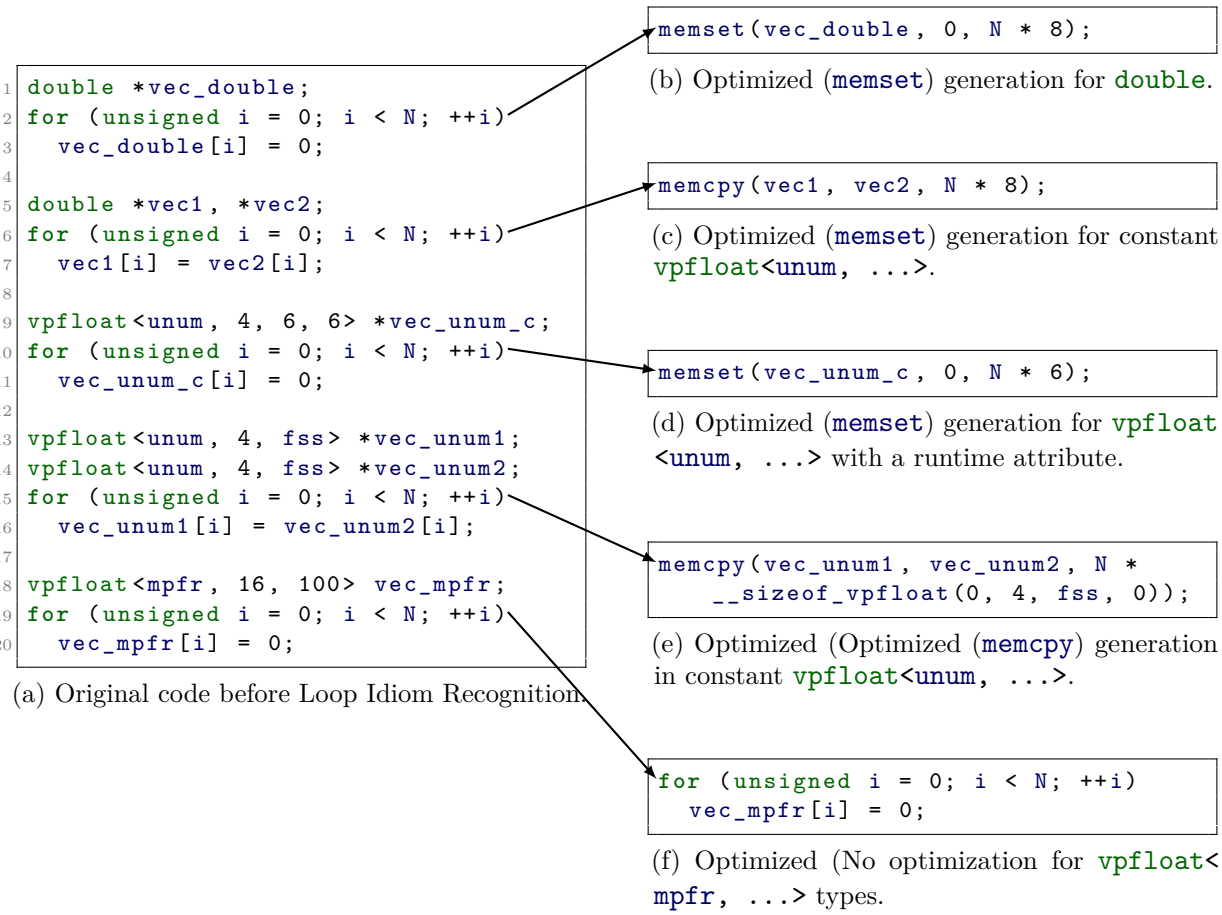
(2) Deletion: Because LLVM classical optimizations have no knowledge about our tracking system, it is possible they wind up realizing that values being used as `vpfloat` attributes are no longer needed in the computation and, thus, can be deleted. However, an object deletion can invalidate types. Our compiler makes sure `vpfloat` attributes are never deleted by adding a mark through an intrinsic call. Because they are inherently seen as having side effects, adding intrinsic calls to mark values avoids their deletion. Although this may have a negative impact on the generated code, we ensure that `vpfloat` types are not invalidated.

5.2.3.2 Loop Idiom Recognition

Another optimization change for `vpfloat` compatibility is implemented for the Loop Idiom Recognizer. This pass implements a simple, yet efficient transformation that replaces a simple loop structure into a non-loop form. Two of the implemented optimizations consist of detecting loop structures that can be replaced by `memset` and `memcpy` calls: the former is used to detect object initialization in an array and the latter copies an object from one location to another.

These transformations can greatly improve application performance and are depicted in Listing 5.4a. Constant `vpfloat` types are implemented in the same fashion as traditional primitive types. Lines 1-10 and Listings 5.4b-5.4d exemplify how the pass generates `memset` and `memcpy` calls to initialize array values for constant types `double` and a 6-byte `unum`.

We have modified this pass to take into consideration dynamically-sized types where sizes cannot be known at 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. Listing 5.4e shows how the pass optimizes a vector-copy *for-loop* by generating a `memcpy` call instead. Because the pass does not know the size of `vec_unum1` and `vec_unum2` a priori, it generates a call to `__sizeof_vpfloat`. Additionally, due to the definition of the base MPFR data type which



```
1  double *vec_double;
2  for (unsigned i = 0; i < N; ++i)
3    vec_double[i] = 0;
4
5  double *vec1, *vec2;
6  for (unsigned i = 0; i < N; ++i)
7    vec1[i] = vec2[i];
8
9  vpfloat<unum, 4, 6, 6> *vec_unum_c;
10 for (unsigned i = 0; i < N; ++i)
11   vec_unum_c[i] = 0;
12
13 vpfloat<unum, 4, fss> *vec_unum1;
14 vpfloat<unum, 4, fss> *vec_unum2;
15 for (unsigned i = 0; i < N; ++i)
16   vec_unum1[i] = vec_unum2[i];
17
18 vpfloat<mpfr, 16, 100> vec_mpfr;
19 for (unsigned i = 0; i < N; ++i)
20   vec_mpfr[i] = 0;
```

(a) Original code before Loop Idiom Recognition.

```
memset(vec_double, 0, N * 8);
```

(b) Optimized (`memset`) generation for `double`.

```
memcpy(vec1, vec2, N * 8);
```

(c) Optimized (`memset`) generation for constant `vpfloat<unum, ...>`.

```
memset(vec_unum_c, 0, N * 6);
```

(d) Optimized (`memset`) generation for `vpfloat<unum, ...>` with a runtime attribute.

```
memcpy(vec_unum1, vec_unum2, N *
    __sizeof_vpfloat(0, 4, fss, 0));
```

(e) Optimized (Optimized (`memcpy`) generation in constant `vpfloat<unum, ...>`.

```
for (unsigned i = 0; i < N; ++i)
  vec_mpfr[i] = 0;
```

(f) Optimized (No optimization for `vpfloat<mpfr, ...>` types.

Listing 5.4: Loop idiom recognition pass for different types: `double`(5.4b, 5.4c), `vpfloat<unum, ...>`(5.4d, 5.4e), and `vpfloat<mpfr, ...>`(5.4f).

includes a pointer variable in the `struct` (see in Listing 4.2), this pass has been disabled for the `vpfloat <mpfr, ...>` types. Listing 5.4f shows that no optimization is performed for the `vpfloat <mpfr, 16, 100>` data type.

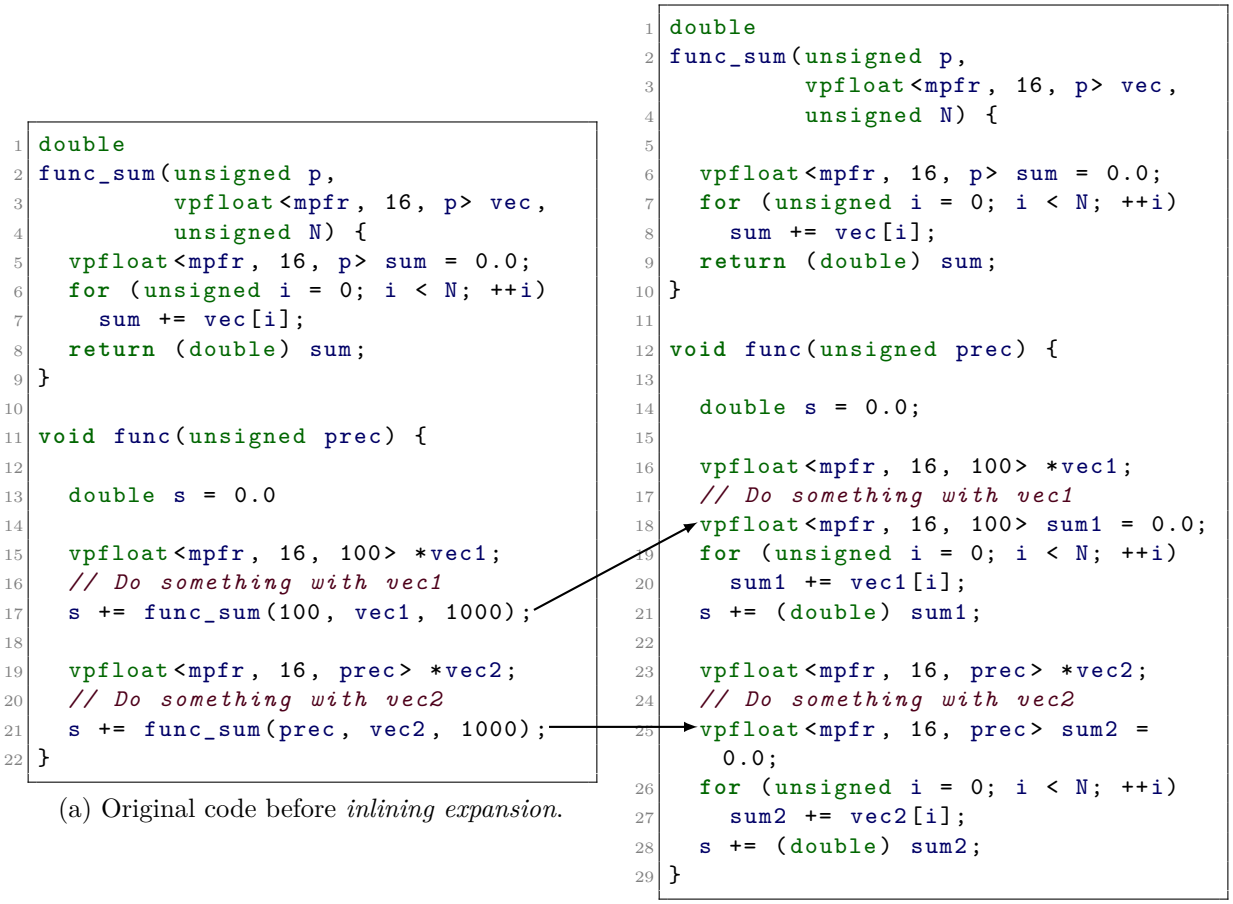
5.2.3.3 Inlining

Inlining Expansion replaces a function call site by the body of the function and is a relevant optimization to improve code performance. Some inlining heuristics can lead to speedups of more than 50% in execution time [10]. While constant types have out-of-the-box support, types with runtime attributes require additional work during inline expansion as they are only considered valid inside a function. This issue boils down to the *def-use chains* from the previous section (see 5.2.3.1).

We have expanded the pass to include support for these types. Values with dynamically-sized types have their types changed (or mutated) in order to comply with the current function where they are being used in. We illustrate two examples of inlining expansions that may be generated:

(1) Constant-type generation mutates the original type to a constant one, thus replacing a function-specific type by a global[2] counterpart. The call to function `func_sum` (Lines

[2]Types such as `float`, `double`, integers, and constant `vpfloat` are considered global because they have a unique representation for the whole program unit.



```
1  double
2  func_sum(unsigned p,
3           vpfloat<mpfr, 16, p> vec,
4           unsigned N) {
5    vpfloat<mpfr, 16, p> sum = 0.0;
6    for (unsigned i = 0; i < N; ++i)
7      sum += vec[i];
8    return (double) sum;
9  }
10
11 void func(unsigned prec) {
12
13   double s = 0.0
14
15   vpfloat<mpfr, 16, 100> *vec1;
16   // Do something with vec1
17   s += func_sum(100, vec1, 1000);
18
19   vpfloat<mpfr, 16, prec> *vec2;
20   // Do something with vec2
21   s += func_sum(prec, vec2, 1000);
22 }
```

(a) Original code before *inlining expansion*.

```
1  double
2  func_sum(unsigned p,
3           vpfloat<mpfr, 16, p> vec,
4           unsigned N) {
5
6    vpfloat<mpfr, 16, p> sum = 0.0;
7    for (unsigned i = 0; i < N; ++i)
8      sum += vec[i];
9    return (double) sum;
10 }
11
12 void func(unsigned prec) {
13
14   double s = 0.0;
15
16   vpfloat<mpfr, 16, 100> *vec1;
17   // Do something with vec1
18   vpfloat<mpfr, 16, 100> sum1 = 0.0;
19   for (unsigned i = 0; i < N; ++i)
20     sum1 += vec1[i];
21   s += (double) sum1;
22
23   vpfloat<mpfr, 16, prec> *vec2;
24   // Do something with vec2
25   vpfloat<mpfr, 16, prec> sum2 =
       0.0;
26   for (unsigned i = 0; i < N; ++i)
27     sum2 += vec2[i];
28   s += (double) sum2;
29 }
```

(b) C representation of the inlining optimization.

Listing 5.5: Inlining expansion examples for types with runtime attributes.

15-17) from Listing 5.5a is inline expanded to the code shown in lines 7-10 from 5.5b, which mutates `vpfloat<mpfr, 16, p>` to `vpfloat<mpfr, 16, 100>`.

(2) The callee `vpfloat` with runtime attributes needs to mutate to a caller one with the same characteristics. Our implementation ensures that caller attributes are used, and no reference to the callee function remains to invalidate values of the new type. Function call to `func_sum`(Lines 19-21) has been inlined, and had values mutate from `vpfloat<mpfr, 16, p>`, in function `func_sum`, to `vpfloat<mpfr, 16, p>`, in function `func`.

In both cases, we must also guarantee that constants, return types, and any call within the inlined function (and its type) are mutated.

5.2.3.4 Lifetime Marker Optimization

LLVM generates a pair of intrinsics, `llvm.lifetime.start` [3] and `llvm.lifetime.end` [4], to mark the lifetime of variables in the program. The main goal is to provide passes with hints on the lifetime of variables for optimization opportunities. More precisely, the compiler implements a stack coloring optimization that makes use of lifetime markers to represent the lifetime of stack slots. It then attempts to reduce the used stack space by merging disjoint stack slots. We must

[3] https://llvm.org/docs/LangRef.html#llvm-lifetime-end-intrinsic
[4] https://llvm.org/docs/LangRef.html#llvm-lifetime-start-intrinsic

follow the same reasoning from loop idiom recognition and disable, or prevent, this optimization to run for MPFR based types. Because code generation will transform them into MPFR *struct* types, we are not able to guarantee proper memory management when stack locations between objects overlap. The pointer variable `_mpfr_d` makes the `mpfr_t` struct non trivially-copyable. Removing *lifetime* intrinsics for MPFR-derived types resolves this issue. Although one may argue that removing lifetime markers may incur an extra performance overhead, the stack coloring optimization has shown negligible impact on performance in our experiments. Their removal asserts that deletions are executed for all objects.

5.2.3.5 OpenMP Multithread Programming

Improvements in performance have much to do with how much parallelism can be explored in computer applications. Having first-class support for primitive types in the compiler enables their use within parallel programming schemes like OpenMP [39]. Looking at both language and compiler perspectives, to the best of our knowledge, this is the first work that shows MPFR and UNUM types integrated within language and compiler, with multi-thread execution capabilities. Previous works are OpenMP-compatible through high-level abstractions and miss optimization opportunities.

5.2.3.6 Loop nest Optimizations

Our LLVM type extension is also able to leverage advanced loop nest optimizers. Polly [55], LLVM's loop nest optimizer, can also be used to optimize `vpfloat` memory accesses. Any limitation of the polyhedral model in `vpfloat` is mostly given by the optimizer. Constant-sized types, described in sections 4.4.1 and 4.4.2, are fully supported by Polly. Any potential performance slowdown is caused by suboptimal heuristic tuning, a well-known challenge with loop nest optimizers in general.

In the case of dynamically-sized types, the limitation stems from Polly's lack of support for performing loop tiling with runtime loop bounds, i.e., the application of loop tiling where bounds are not known at compile time. This requires heuristics to generate loops with a dynamic (runtime decidable) behavior, and full automatic support for dynamically-sized types are mostly dependent upon adding this functionality to Polly. Although we can trick the optimizer to assume these types to have constant sizes, it is likely to generate suboptimal heuristics with poor performance. Nevertheless, we can still explore polyhedral techniques at code-level with other optimizers [25, 119], as shown in 4.7.1. Still, our experimental results 6.1 will show the benefits of Polly for constant-sized types.

5.2.3.7 Vectorization

Vectorization has become one of the most essential techniques in today's systems. Although our backends are not yet prepared to handle *vector* instructions, our LLVM IR type system has been implemented so that it does not hinder the use of vectorization. In other words, vectorization of `vpfloat` types is possible if compatible code generators are provided. Perhaps more importantly, because of our extensible language and compiler extensions, developers can, more easily, define new types that can potentially make use of vectorization strategies.

In Listing 5.6, we show an example *axpy* function implemented with an MPFR-derived type (Listing 5.6a) and its vectorized IR code for the loop body (Listing 5.6b). Code was compiled with *-O3* flag, which enables vectorization, and *-mllvm -force-vector-width=128* option to force vector size of 128 elements. There is no IR restriction for vectorization of `vpfloat` types with

```
void vec_axpy(unsigned precision, int n,
              vpfloat<mpfr, 16, precision> alpha,
              vpfloat<mpfr, 16, precision> *X, int incx,
              vpfloat<mpfr, 16, precision> *Y, int incy) {
  for (unsigned i = 0; i < n; ++i) {
    Y[i] = alpha * X[i] + Y[i];
  }
}
```

(a) C code to be vectorized.

```
vector.body:
  %index = phi i64 [ 0, %vector.ph.new ], [ %index.next.1, %vector.body ]
  %niter = phi i64 [ %unroll_iter, %vector.ph.new ], [ %niter.nsub.1, %vector.body ]
  %9 = getelementptr inbounds vpfloat<1, 16, %precision>, vpfloat<1, 16, %precision>* %X, i64 %index
  %10 = bitcast vpfloat<1, 16, %precision>* %9 to <128 x vpfloat<1, 16, %precision>>*
  %wide.load = load <128 x vpfloat<1, 16, %precision>>, <128 x vpfloat<1, 16, %precision>>* %10
  %11 = fmul <128 x vpfloat<1, 16, %precision>> %wide.load, %broadcast.splat
  %12 = getelementptr inbounds vpfloat<1, 16, %precision>, vpfloat<1, 16, %precision>* %Y, i64 %index
  %13 = bitcast vpfloat<1, 16, %precision>* %12 to <128 x vpfloat<1, 16, %precision>>*
  %wide.load18 = load <128 x vpfloat<1, 16, %precision>>, <128 x vpfloat<1, 16, %precision>>* %13
  %14 = fadd <128 x vpfloat<1, 16, %precision>> %11, %wide.load18
  store <128 x vpfloat<1, 16, %precision>> %14, <128 x vpfloat<1, 16, %precision>>* %13
  %index.next = or i64 %index, 128
  %15 = getelementptr inbounds vpfloat<1, 16, %precision>, vpfloat<1, 16, %precision>* %X, i64 %index.next
  %16 = bitcast vpfloat<1, 16, %precision>* %15 to <128 x vpfloat<1, 16, %precision>>*
  %wide.load.1 = load <128 x vpfloat<1, 16, %precision>>, <128 x vpfloat<1, 16, %precision>>* %16
  %17 = fmul <128 x vpfloat<1, 16, %precision>> %wide.load.1, %broadcast.splat
  %18 = getelementptr inbounds vpfloat<1, 16, %precision>, vpfloat<1, 16, %precision>* %Y, i64 %index.next
  %19 = bitcast vpfloat<1, 16, %precision>* %18 to <128 x vpfloat<1, 16, %precision>>*
  %wide.load18.1 = load <128 x vpfloat<1, 16, %precision>>, <128 x vpfloat<1, 16, %precision>>* %19
  %20 = fadd <128 x vpfloat<1, 16, %precision>> %17, %wide.load18.1
  store <128 x vpfloat<1, 16, %precision>> %20, <128 x vpfloat<1, 16, %precision>>* %19
  %index.next.1 = add i64 %index, 256
  %niter.nsub.1 = add i64 %niter, -2
  %niter.ncmp.1 = icmp eq i64 %niter.nsub.1, 0
  br i1 %niter.ncmp.1, label %middle.block.unr-lcssa, label %vector.body
```

(b) Vectorized IR code for the loop body.

Listing 5.6: Vectorization of `vpfloat` types.

current LLVM vectorizers. Rather than that, code generators are not yet designed to handle them and are an important exploration venue to consider in the future.

5.3 Code Generators

From the application perspective, an IR type extension like `vpfloat` is not worthwhile unless a compatible backend for each type is provided to drive their capabilities. In order to evaluate the effectiveness of this integration, we designed and implemented two backend code generators that support each `vpfloat` representation.

Backends were selected to fit the requirements of our extension. In both cases, we chose those that effectively offer the flexibility we provide. That is, (1) support for variations of precision, exponent attributes with static and dynamic features. The MPFR backend consists of lowering the `vpfloat<mpfr, ...>` to generate MPFR [49] code, hence, showing the integration with a software target. Our second backend makes use of the `vpfloat<unum, ...>` representation in order to partially implement the RISC-V ISA [21, 23] extension for UNUM variable precision arithmetic of [57], thus targeting a hardware solution. The remaining sections of this chapter detail the aspects relative to each backend, showing code transformations, and target-specific optimizations implemented for each backend.

5.3.1 Software Target: MPFR

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`. This means that adding backend support for a software target can easily be achieved by implementing a few functions. Our transformation only requires one to write callbacks to set up the external library information, such as, the base data type used, names of function that implement each operation, allocation strategy and routines, among others. In fact, one could easily use the MPFR backend as a GMP [54] backend by providing the appropriate callback routines. Or even more, a software backend for the `vpfloat<unum, ...>` backend could also be derived in the same way.

In its essence, 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 4.2) that must be allocated and initialized before first use. This characteristic imposes the first hurdle for MPFR code generation in an unmanaged language like C. While wrappers for higher-level languages like C++, Python and Julia, can hide allocations and deallocations away from users through their language abstractions, C has no automatic support for memory management.

We thus provide similar functionality to enable automatic memory management of MPFR objects by monitoring LLVM IR `alloca` instructions and their enclosing scope. This is possible because `vpfloat` variables are typed as first-class scalar values, and are modeled as stack-allocated in upstream passes. 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. 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.

Below, we show examples of the allocation strategies adopted as described in the aforementioned paragraphs, namely, for `alloca` instructions, multi-dimensional arrays, and dynamic memory allocation, respectively. On the left-hand side, we depict a snippet of a `vpfloat` code as written by the programmer, while the right-hand side illustrates a C-level representation generated by our transformation pass. Second and third examples, more specifically, show library function calls to allocate and free MPFR objects in multi-dimensional arrays.

```
{
vpfloat <mpfr, 16, 200> val200;
// Do something with val200
} // Leaves scope, thus, frees
    val200.
```

```
{
mpfr_t val200;
mpfr_init(val200, 200);
// Do something with val1200
mpfr_clear(val1200); // frees val200
    before leaving
} // Leaves scope
```

```
{
vpfloat <mpfr, 16, 200> vp_mat
    [10][20];
// Do something with vp_mat
} // Leaves scope, thus, frees
    vp_mat.
```

```
{
mpfr_t vp_mat[10][20];
vp_malloc_2d(vp_mat, 200, 10, 20);
// Do something with vp_mat
vp_free_2d(vp_mat); // frees vp_mat
    before leaving
} // Leaves scope
```

```
{
vpfloat <mpfr, 16, 200> *vp_vec =
    malloc(20000 * sizeof(vpfloat <
    mpfr, 16, 200>));
// Do something with vp_vec
free(vp_vec);
} // Leaves scope, thus, frees
    vp_vec.
```

```
{
mpfr_t *vp_vec = malloc(20000 *
    sizeof(mpfr_t));
vp_malloc_1d(vp_vec, 200, 1000);
// Do something with vp_vec
vp_free_1d(vp_vec); // frees vp_vec
    before leaving
free(vp_vec);
} // Leaves scope
```

(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. These *op-to-op* conversions gives us opportunities for a first optimization: we try to leverage MPFR functions specialized for the case where one or more operands can be re-written with a regular primitive data type, e.g. `double`, `unsigned`, `float`, etc, without precision lost. The following example shows some of the conversions and optimizations performed. Notice that values `1.0` and `0.5` can be represented as `int` and `double` even though they were previously defined as `vpfloat`. This simple approach is used to generate more specialized functions to accelerate computation since they are likely to operate in objects with a smaller memory footprint.

```
void func(vpfloat<mpfr, 16, 1000> val1, vpfloat<mpfr, 16, 1000> val2,
          vpfloat<mpfr, 16, 1000> *res){

  *res = -val1;
  *res -= val2;
  *res += -3.0v;
  *res = 0.5v/(*res);
  // Continue...
```

```
void func(mpfr_t val1, mpfr_t val2, mpfr_t res){

  mpfr_neg(res, val1, MPFR_RNDN);
  mpfr_sub(res, res, val2, MPFR_RNDN);
  mpfr_add_si(res, val1, -3.0, MPFR_RNDN);
  mpfr_d_div(res, 0.5, val1, MPFR_RNDN);
  // Continue...
```

(3) Functions with `vpfloat` MPFR arguments are cloned and re-built as MPFR objects. Although previous examples show that references and value copies are passed similarly in function calls, the pass respects the C standard for argument passing, such as, *pass by value*, *pass by reference*. They follow the same behavior as primitive scalar floating-point types in regards to multi-dimensional arrays, pointers as function parameters. The compiler also makes sure that passed-by-value arguments are only affected in the callee function, while changes in the passed-by-reference values are also seen within the caller. Additionally, return types are handled through LLVM's *StructRet* attribute, being returned as the first argument of the function.

```
vpfloat<mpfr, 16, 1000>
    vdot(int n, vpfloat<mpfr, 16, 1000> *X,
         vpfloat<mpfr, 16, precision> *Y);
```

```
void vdot(mpfr_t ret_result, int n, mpfr_t X, mpfr_t Y);
```

(4) Load instructions, Φ Nodes, dereferencing (element-indexing) instructions, and constant values of `vpfloat` arguments are all rewritten to use the MPFR `struct` type. Store instructions are converted to `mpfr_set` or any of its derivative functions (`mpfr_set_{si, ui,d}`) for performance purposes.

(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 types. The lack of support for types with runtime attributes within compound types (class, struct, and function types, etc.), as described in the *Language Extension Limitations* section 4.6 from the previous chapter, propagates this limitation to the backend.

Compound types 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.

In the example above, our pass creates a new `VPFloatClass` with `mpfr_t` types and recreates all objects with this new type. Although LLVM provides a `mutateType` method, it is strongly advised to recreated objects instead. Creating new compound types involves

```
class KernelBase {
public:
  virtual void updateChecksum(VariantID vid) = 0;
};

class VPFloatClass :  public KernelBase {
public:
  void updateChecksum(VariantID vid);
private:
  vpfloat<mpfr, 16, 1000> x;
  vpfloat<mpfr, 16, 1000>* vec;
};
```

```
class VPFloatClass :  public KernelBase {
public:
  void updateChecksum(VariantID vid);
private:
  mpfr_t x;
  mpfr_t vec;
};
```

a recursive operation to remake them in a bottom-up order due to type dependencies: `type1` may declare an object of `type2` that has `vpfloat<mpfr, ...>` inside. Not only do we need to recreate `type2`, but `type1` as well, and the associated structures that they use.

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. From the language perspective, `vpfloat` types are considered to have the same semantics as regular FP types. That is, they are seen as trivially copyable types. On the one hand, this facilitates IR code generation from a C++ level since it does not require implementation of *copy* and *move constructors*. On the other hand, it also adds extra complexity to the generation of MPFR code, which, due to the pointer variable in the `struct`, is not trivially copyable. To circumvent this issue, our code generator detects memory-related functions and generates three additional functions (`prepare_memcpy`, `clean_memcpy`, `vp_memcpy`) to 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. We illustrate this transformation in the example below:

Function `prepare_memcpy` performs a simple `memcpy` to save the values of `X` which are overwritten by the original `memcpy` operation. After `memcpy` is executed, the pass calls `vp_memcpy` to restore the correct values of the object and make a copy to their right locations. Lastly, `clean_memcpy` frees up the temporary memory allocated by `prepare_memcpy`. These functions were implemented to handle the manipulation of `vpfloat` types in two scenarios: (1) when pure multi-dimensional arrays are copied/moved between locations, like in the example; (2) or when copies and moves are introduced due to lambda functions and C++ standard library. For instance, in the case of having `vpfloat` declarations within a class object.

(6) Section 5.2.3.5 briefly states that having support for primitive types facilitates the use of parallelization through programming models. Support for OpenMP is included almost out-of-the-box for MPFR-derived types. The only special treatment lies on handling `omp atomic` directives, which generates a call to `atomic_compare_exchange`, a function

```
void mem_func_example(int n, vpfloat<mpfr, 16, 1000> *X,
                      vpfloat<mpfr, 16, 1000> *Y) {
    memcpy(X, Y, sizeof(vpfloat<mpfr, 16, 100>)*n);
    // Some code
}
```

```
void mem_func_example(int n, mpfr_t X, mpfr_t Y) {

    // Prepare memcpy
    void **tmp;
    prepare_memcpy(tmp, X, sizeof(mpfr_t)*n);

    // memcpy executed, pointer is overwritten
    memcpy(X, Y, sizeof(mpfr_t)*n);

    // Restore correct values overwritten by memcpy and clenup
    vp_memcpy(X, tmp, &X, &Y, 1, 32, sizeof(mpfr_t)*n);
    clean_memcpy(tmp);

    // Some code
}
```

that implements an atomic compare-and-swap.[5] MPFR-derived types are not automatically handled by OpenMP in scenarios where objects need to be atomically modified with a single IR or instruction (they use a library call). This occurs because, as mentioned herein repeatedly, the pointer variable in the MPFR `struct` prevents atomic updates. 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. Function `vp_atomic_compare_exchange` has an equivalent implementation to those of traditional `atomic_compare_exchange` functions, but targets MPFR objects. We present below the use of `omp atomic` directive to calculate the value of π and pseudo-codes generated by the compiler from the `pragma` directive. The code on the left follows the classical compilation flow and generates a call to `atomic_compare_exchange` to atomically manipulated π in `double`. On the right, π is calculated using `vpfloat<mpfr, 16, 1000>`, and our transformation adds a critical section in order to properly update the object value with `vp_atomic_compare_exchange`. Moreover, we also present a possible implementation for `vp_atomic_compare_exchange`.

(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 the pass operates on Static Single Assignment (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 them with MPFR objects and the appropriate initialization. Unlike higher-level MPFR abstractions such as the C++ Boost library for multi-precision arithmetic [85], we are able to leverage the compiler toolchain and its existing optimizations (constant folding, inlining, common subexpression

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

```
void prepare_memcpy(char **temp_object, char *src, unsigned size) {
  *temp_object = (char*)malloc(size);
  memcpy(*temp_object, src, size);
}

void clean_memcpy(char **temp_object) {
  free(*temp_object);
}

void vp_memcpy(mpfr_t* dst, void* src, char *initial_address,
               unsigned num_elems, unsigned elem_size, unsigned size) {

  unsigned num_objects = ceil((double)size/(double)(elem_size*num_elems));
  unsigned offset = (unsigned)((char*)dst - initial_address);

  // Do nothing in the case offset is larger than size.
  // It means the compiler could not verify at compilation time
  // that a memcpy is not needed
  if (offset > size || size < 32)
    return;
  else if (num_objects == 0) {
    printf("Number of objects cannot be Zero.\n");
    exit(0);
  } else if (elem_size == 32)
    size = 32;

  for (unsigned i = 0; i < num_objects; ++i) {
    for (unsigned j = 0; j < num_elems; ++j) {
      mpfr_t *srcval = (mpfr_t*)((char*)src + offset + (i*size));
      mpfr_t *dstval = (mpfr_t*)((char*)dst + (i*size));
      // Copy the value of srcval to dstval and change
      // the pointer to that value.
      mpfr_set(srcval[j], dstval[j], roundingMode);
      dstval[j]->_mpfr_d = srcval[j]->_mpfr_d;
    }
  }
}
```

```
#pragma omp parallel for
for (i = 0; i < 10000; ++i) {
  double x=(i+0.5)*dx;
  #pragma omp atomic
  pi += (dx / (1.0 + x * x));
}
```

```
#pragma omp parallel for
for (i = 0; i < 10000; ++i) {
  vpfloat<mpfr,16,500> x=(i+0.5)*dx;
  #pragma omp atomic
  pi += (dx / (1.0 + x * x));
}
```

```
// Coded in double
pi_local += (dx / (1.0 + x * x));
calls atomic_compare_exchange to
    compare/update pi and pi_local
```

```
// Coded in vpfloat<mpfr,16,500>
pi_local += (dx / (1.0 + x * x));
start critical section
calls vp_atomic_compare_exchange to
    compare/update pi and pi_local
end critical section
```

```
bool vp_atomic_compare_exchange(unsigned size, void *ptr,
                                void *expected, void *desired,
                                int success_order, int failure_order) {
  bool ret;
  if (mpfr_cmp(*((mpfr_t*)ptr), *((mpfr_t*)expected)) == 0) {
    mpfr_set(*((mpfr_t*)ptr), *((mpfr_t*)desired), roundingMode);
    ret = true;
  } else {
    mpfr_set(*((mpfr_t*)expected), *((mpfr_t*)ptr), roundingMode);
    ret =  false;
  }
  return ret;
}
```

Table 5.1: Instructions supported by the UNUM Backend.

Instruction group	Instruction group
susr	fcvt.x.g/fcvt.g.x
lusr	fcvt.f.g/fcvt.g.f
smbb/swgp/sdue/ssue	fcvt.d.g/fcvt.g.d
lmbb/lwgp/ldue/lsue	gcmp/gadd/gsub/gmul
~~movll/movlr~~	~~gguess/gradius~~
~~movrl/movrr~~	lgu/lgu.s/~~ldub/ldub.s~~
mov_x2g/mov_g2x	stu/stu.s/~~stub/stub.s~~
mov_d2g/mov_f2g	~~lgu_next/ldub_next~~
mov_g2d/mov_g2f	~~stul_next/stub_next~~

elimination, among others), with MPFR objects only being created at the end of the middle-end compilation flow. Furthermore, since this backend is actually implemented as a middle-end pass, it is usable for all target ISAs with respective LLVM backends. In fact, because our pass is target-independent, we have collected results for this target in two distinct systems: a RISC-V FPGA-based processor, as well as X86 systems.

5.3.2 Hardware Target: UNUM

Our second backend makes use of the `vpfloat<unum, ...>` representation in order to partially implement the RISC-V ISA [121] extension for UNUM variable precision arithmetic of [23]. Table 5.1 shows the instructions supported by our backend. We exclude all instructions related to interval arithmetic, since our types are only used for UNUM scalar representations, and are not intended for interval endpoints.

The ABI specification for the ISA extension is similar to the standard RISC-V FP ABI. Table 5.2 lists the coprocessor registers and respective roles in the defined calling convention. Register naming uses the same convention as the FP registers in RISC-V. However, in lieu of using `f` as a prefix, the letter `g` is used to denote coprocessor registers. The calling convention for argument and return values also complies with the RISC-V FP ABI, as well as the registers preserved across function calls.

5.3.2.1 Compiler-Controlled Status Registers

The compiler must work together with the proposed ISA to coordinate and control multiple parameters inside the coprocessor. This ISA extension supports generic FP instructions with precision ranging from 64 to 512 bits, controlled at a 64-bit granularity. Since the UNUM format [57] 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, controlling the maximum values of *ess* and *fss* defined by the coprocessor, also known as UNUM environment *(ess,fss)*. For instance, a UNUM environment (4, 8) indicates maximum *ess* and *fss* values of 4, 8, respectively. These values are sufficient to represent numbers with at most 16 bits of exponent, and 256 bits of mantissa. 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.

According to the ISA specification, the compiler is in charge of controlling six control registers. Table 5.3 gives a description of each control register with the minimum and maximum values they can assume. The ISA specifies two UNUM environments that can be used simultaneously by operating with different memory instructions. Instructions *lgu* and *stu* use values of the default UNUM environment, while *lgu.s* and *stu.s* consider *ess* and *fss* values of the second UNUM environment. In the following section 5.3.2.2, it will be shown that having two UNUM environments help the compiler handling constants for types with runtime attributes. Values *wgp* and *mbb*, on the other hand, are have no duplication. That is, the compiler may need to alternate these values in order to allow multiple configurations simultaneously.

Additionally, we have designed and implemented two additional passes to properly handle the generation of generic FP operations with the UNUM ISA: the first targets the configuration of status registers in the coprocessor, and the second addresses memory accesses for dynamically-sized types in multi-dimensional arrays.

5.3.2.2 FP Configuration Pass

The FP configuration pass consists on analyzing functions in the call graph and properly configures the *status registers* (*ess*, *fss*, *WGP*, *MBB*) as to convey to high level type information. The pass runs in the middle-end phase since it is the lowest level to retain information about type configuration. Types in the backend are lightweight and cannot fully represent FP attributes of `vpfloat`. An IR pass also enables it to be target-independent and can potentially be used for architectures other than RISC-V.

The pass keeps track of values that come in and go out of basic blocks. By analyzing the control flow graph (CFG), it guarantees that values are being properly assigned. If any change is

Table 5.2: ABI Convention for the VP registers

Register	ABI Name	Description	Saver
g0-7	gt0-7	Temporaries	Caller
g8-9	gs0-1	Saved Registers	Callee
g10-11	ga0-1	Arguments/return values	Caller
g12-17	ga2-7	Arguments	Caller
g18-27	gs2-11	Saved Registers	Callee
g28-31	gt8-11	Temporaries	Caller

Table 5.3: Control registers inside the UNUM Coprocessor.

Category	Name	(min,max) Allowed	Description
Default UNUM Environment (DUE)	due.ess	(1,4)	Tells the coprocessor the maximum number of bits in the *ess* field for load and store (*lgu* and *stu*)
	due.fss	(1,9)	Tells the coprocessor the maximum number of bits in the *fss* field for load and store (*lgu* and *stu*)
Second UNUM Environment (SUE)	sue.ess	(1,4)	Tells the coprocessor the maximum number of bits in the *ess* field for load and store (*lgu.s* and *stu.s*)
	sue.fss	(1,9)	Tells the coprocessor the maximum number of bits in the *fss* field for load and store (*lgu.s* and *stu.s*)
Precision	wgp	(0,7)	Number of bits used for arithmetic operations in the FPU unit
Memory Byte Budget	mbb	(1,68)	Number of bits used in load and store instructions

needed when entering a basic block, a new instruction is added. We make use of variations of the *susr* instruction to configure each register separately: instructions *susr.ess*, *susr.fss* manipulate *ess* and *fss* values of the default UNUM environment, and *susr.wgp* and *susr.mbb* are used to configure *WGP*, and *MBB*.

Although the coprocessor has two UNUM environments that could be used to alternate the used values of *ess* and *fss*, we reserve the second one to handle two situations:

(1) Load constants in dynamically-sized types: section 4.4.3.4 explains that constants for these types are generated with a maximum configuration. For UNUM types, that corresponds to the maximum values of *ess* and *fss* supported by the coprocessor. Therefore, assigning a constant to a dynamically-sized variable implies the use of *lgu.s* in lieu of the *lgu* instruction[6].

(2) Register spilling may be necessary when the number of live variables during the execution of the program is greater than the number of available registers. We use the second environment to spill the register as a maximum configuration. Even though this slows down spilling and filling operations during register allocation, it does not require the register allocator to have any previous knowledge of the FP configuration of variables been spilled. Moreover, since there may be a significant difference between the spilling size of a variable and its actual memory size, techniques to avoid spilling [97] can play an important role to reduce the memory impact on high-precision representations.

5.3.2.3 Array Address Calculation Pass

Array address calculation 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

[6]Instructions *lgu* and *lgu.s* load UNUM variables from memory. They take into consideration values *ess* and *fss* of the default and second UNUM environment, respectively, as well as MBB.

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.

5.4 Conclusion

This chapter presents the modifications and requirements needed to extend the support of our `vpfloat` type system substrate to an industry-level compiler infrastructure. We showed many aspects that justify why this integration improves the state-of-the-art. We showed which optimizations may and may not run in the compilation flow, and code modifications needed for many of the language aspects, such as runtime capabilities and type compatibility. We also described how `vpfloat<unum, ...>` and `vpfloat<mpfr, ...>` go from the LLVM IR type system to their code generators: the former through a RISC-V ISA extension and the latter by relying on the open-source MPFR library.

Chapter 6: Experimental results

Contents

In the previous chapters, we have described the whole compilation flow required to provide variable precision FP arithmetic. Language extension, type system and compiler support have been proposed as an intent to better explore different types, precision and exponent configurations, and at last variable precision computing. This chapter will focus on the experimental results that give basis to the contributions of this thesis.

We start by presenting and hopefully convincing the reader on, the benefits of language and compiler integration of a language extension for multiple FP formats. The second segment of this section consists of illustrating the use of variable precision within linear algebra kernels. By making use of the *mpfrBLAS* library described in section 4.7.1, we provide implementations to different variants of the Conjugate Gradient (CG) and show interesting insights that may encourage the exploration of variable precision computing.

6.1 The Benefits of Language and Compiler Integration

This section is mainly divided into two parts. The first focuses on comparing our extension, type system and compiler solution to a state-of-the-art approach with equivalent functionality with the intent of showing the benefits of compiler integration. The last part presents experimental results to demonstrate the advantage of having hardware support for high-precision formats.

6.1.1 MPFR `vpfloat` vs. Boost Multi-precision

This section shows a comparison of our MPFR type `vpfloat<mpfr, ...>` with the Boost library for multi-precision [85]. Both approaches rely on the MPFR library and execute code with equivalent precision values. The goal is to demonstrate that high-precision FP emulation libraries can further benefit from language and compiler integration. Because compilers cannot detect FP representations within high-level languages abstractions, optimizations opportunities are missed. While traditional IEEE formats are handled through different compilation strategies at a later stage of the compilation process[1], our type system mimics the behavior of standard

[1] In LLVM, for instance, the choice between an FP-capable ISA and software implementation is only done at the backend stage, when building Direct-Acyclic Graphs (DAGs).

Table 6.1: Machine configurations used for experiments.

	Processor Model	# Procs.	# Cores # Threads	Frequency	Caches			RAM
					L1	L2	L3	
M1	Intel Xeon E5-2637v3	2	8 Cores 16 Threads	3.50 GHz	4 x 32KB D-Cache 4 x 32KB I-Cache	4 x 256KB	15MB	128GB
M2	Intel Xeon Gold 5220	2	36 Cores 72 Threads	2.20 GHz	18 x 32KB D-caches 18 x 32KB I-Caches	18 x 1 MB	24.75MB	96GB

(compiler-compatible) types by relying on a middle-end pass but still leveraging much of the compilation flow and optimizations. Experiments were conducted using two different systems: a dual-processor Intel Xeon E5-2637v3 machine with 128GB of RAM, and a dual-processor Intel Xeon Gold 5220 machine with 96 GB of RAM, labeled `M1` and `M2`, respectively, and depicted in Table 6.1.

6.1.1.1 Polybench

The Polybench benchmark suite [98] is a collection of 30 kernels aimed to explore the impact of compiler optimizations in a variety of areas. It is widely used to measure the benefits of applying polyhedral optimization techniques in a compiler and contains code for many computation-intensive algorithms in fields of linear algebra, stencil, and data-mining.

Two strategies were used to Polybench version 4.1: (1) first, we compiled Polybench for both baseline and `vpfloat<mpfr, ...>` types using optimization level -O3; (2) in the second scenario, we use optimization level -O1 for `vpfloat<mpfr, ...>` for reasons that will be explained later, while Boost continued to be compiled with -O3. In both scenarios, we enable and disable Polly's polyhedral loop nest optimizations [55], and the best execution time reference, with and without Polly, is taken for each application. Each application was compiled for three different precision configurations: 100 bits (≈30 decimal digits), 170 bits (≈50 decimal digits), and 500 bits (≈150 decimal digits).

Figure 6.2 shows the speedup for each benchmark using Boost as the baseline and when compiling both solutions with optimization level -O3. Figure 6.3 depicts the values of speedups considering the same baseline but with `vpfloat<mpfr, ...>` applications being compiled with optimization level -O1. We observe speedups over most of the test suite for all precision configurations in the two machines where experiments were executed. A few kernels have shown similar performance to Boost: *jacobi-1d* and *jacobi-2d* at the lower precision settings, and *doitgen*, *durbin* and *mvt* at highest precision when executed in the `M2`. The only slowdowns occur on some specific cases: *adi* and *deriche* at lower precisions only. 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.

The reason why we chose to plot results for *-O1* optimization level is that we observed many benchmarks exhibit better performance with this optimization flag rather than with *-O3*. In

fact, the overall speedup of *-O1* has shown to be superior to *-O3*. This was originally a surprise, but can be explained as follows: as MPFR requires memory allocation for every object in use, suboptimal performance at *-O3* stems from the higher number of allocations generated. Because the compiler runs many aggressive optimizations (inlining, loop unrolling, aggressive code motion, just to name a few), many more objects need allocation and incur an extra execution time in the application. In fact, finding optimization sequences that are likely to yield better performance is challenging. However, recent work [109, 128] on the exploration of optimization space could potentially help us investigate better optimization heuristics in the presence of multi-precision arithmetic, and is a relevant topic for future work.

We have also decided to run experiments in two different machines in order to show that speedups are not inherent to specific system configuration. Experimental results are similar in the two configurations, and differences are, in the most part, within an acceptable tolerance range. Kernels *floyd*, and *lu* show significantly better speedups in `M2`, because Boost versions show bad usage of the cache with an increase in the number of cache misses, and thus, impacting execution time. A following experiment will demonstrate that cache misses in Boost have an even greater impact when running applications with multiple threads of execution.

Overall, results show an average performance speedup (`vpfloat<mpfr, ...>` vs. Boost library for MPFR) of 1.80× and 1.86× for `M1` and `M2`, respectively in *-O3*. Additionally, it should be pointed out that these results have been corroborated by peers through an Artifact Evaluation (AE)[2] submitted to a conference, and *functional* and *reproduced* badges had been given. For *-O1*, speedups of 1.88× and 1.91× were observed in the two machines.

Impact of the Loop nest optimizer

One of the major advantages of our solution, as we have been highlighting throughout this document, relates to compiler integration and the possibility of using optimizations out-of-the-box. This is particularly true when considering Polly Loop nest Optimizer. Even without modifications to support `vpfloat`, Polly is able to find optimal array access patterns in the Polybench suite compiled for `vpfloat`. To illustrate our argument, figure 6.1 shows the speedup of *-O3 + Polly* vs. *-O3*. Although many kernels have seen slowdowns when compiled with Polly, an overall speedup of 1.18× is observed. It shows that deeper compiler integration helps to accelerate FP emulation libraries. On the other hand, Polly is unable to optimize Boost array accesses. The optimizer gives up trying to find patterns that suit the complexity of the library.

Table 6.2: Compilation time for Polybench with different optimization levels and types.

Type	Opt. Flags	Compilation time (in seconds)	Speedup over Boost
`vpfloat<mpfr, ...>`	-O3	13.02	8.91
	-O3 + Polly	91.79	1.27
	-O1	8.81	13.17
	-O1 + Polly	69.87	1.68
Boost	-O3	116.09	1.00
	-O3 + Polly	117.32	1.00

[2] https://www.acm.org/publications/policies/artifact-review-badging

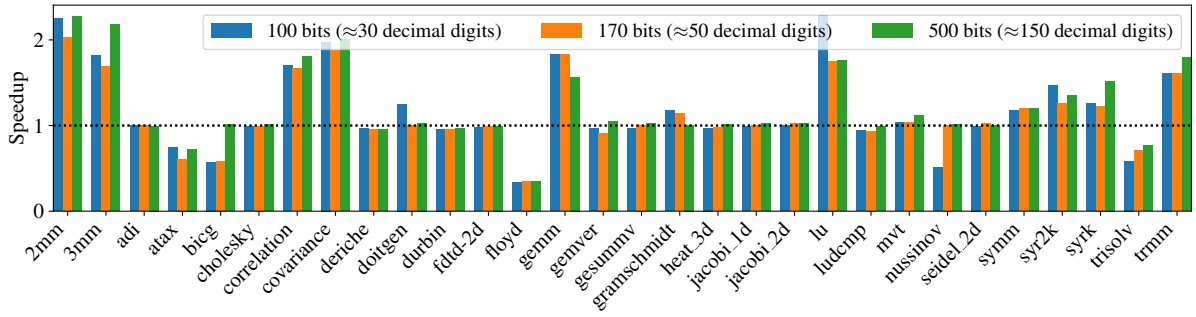


Figure 6.1: Speedup of Polly's loop nest optimizer in Polybench compiled for `vpfloat<mpfr, ...` types.

Compilation time

Another important metric to consider is the time needed to compile each application. Because the middle-end pass implemented to transform `vpfloat<mpfr, ...>` to MPFR references incurs an extra compilation time to code generation, it is important to measure its impact in compilation time. Table 6.2 shows significant speedups in terms of compilation time for the Polybench applications. In fact, Boost high compilation times stem from C++ template deduction, highly used in the Boost Multi-precision library. The difference in compilation times between *-O3* and *-O3 + Polly* for `vpfloat<mpfr, ...>` types shows that the optimizer is, in fact, supported by these types. Similar compilation times between *-O3* and *-O3 + Polly* for Boost shows the contrary.

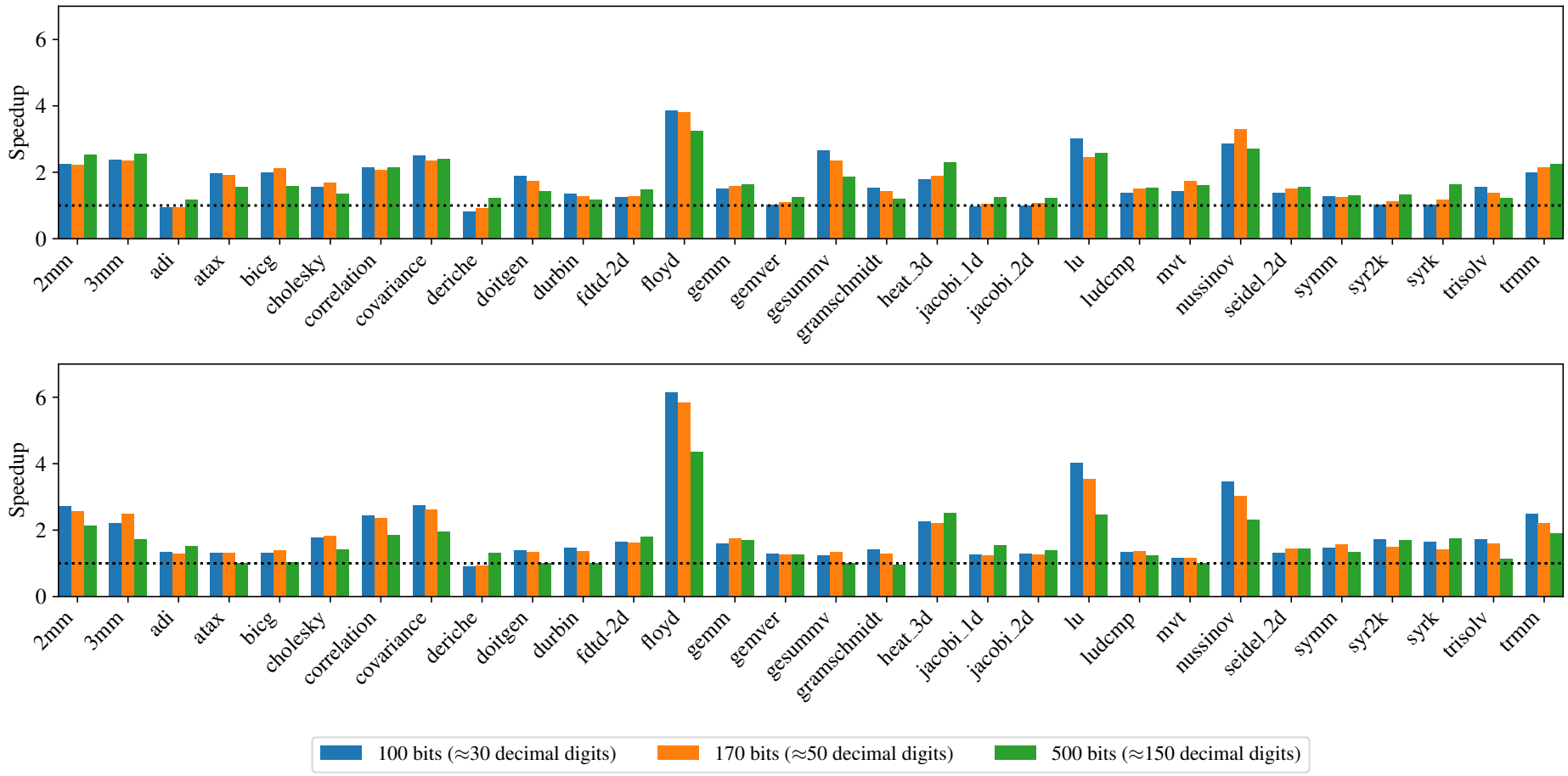


Figure 6.2: Speedup of `vpfloat<mpfr, ...>` over the Boost library for multi-precision for the Polybench benchmark suite, and compiled with optimization level -O3. The execution time reference taken are the best between compilations with and without Polly. Results are shown for two different machines: an Intel Xeon E5-2637v3 with 128GB of RAM (`M1`), and an Intel Xeon Gold 5220 with 96 GB of RAM (`M2`), respectively. Y-axes are shown with the same limits to ease comparisons between results in the two machines.

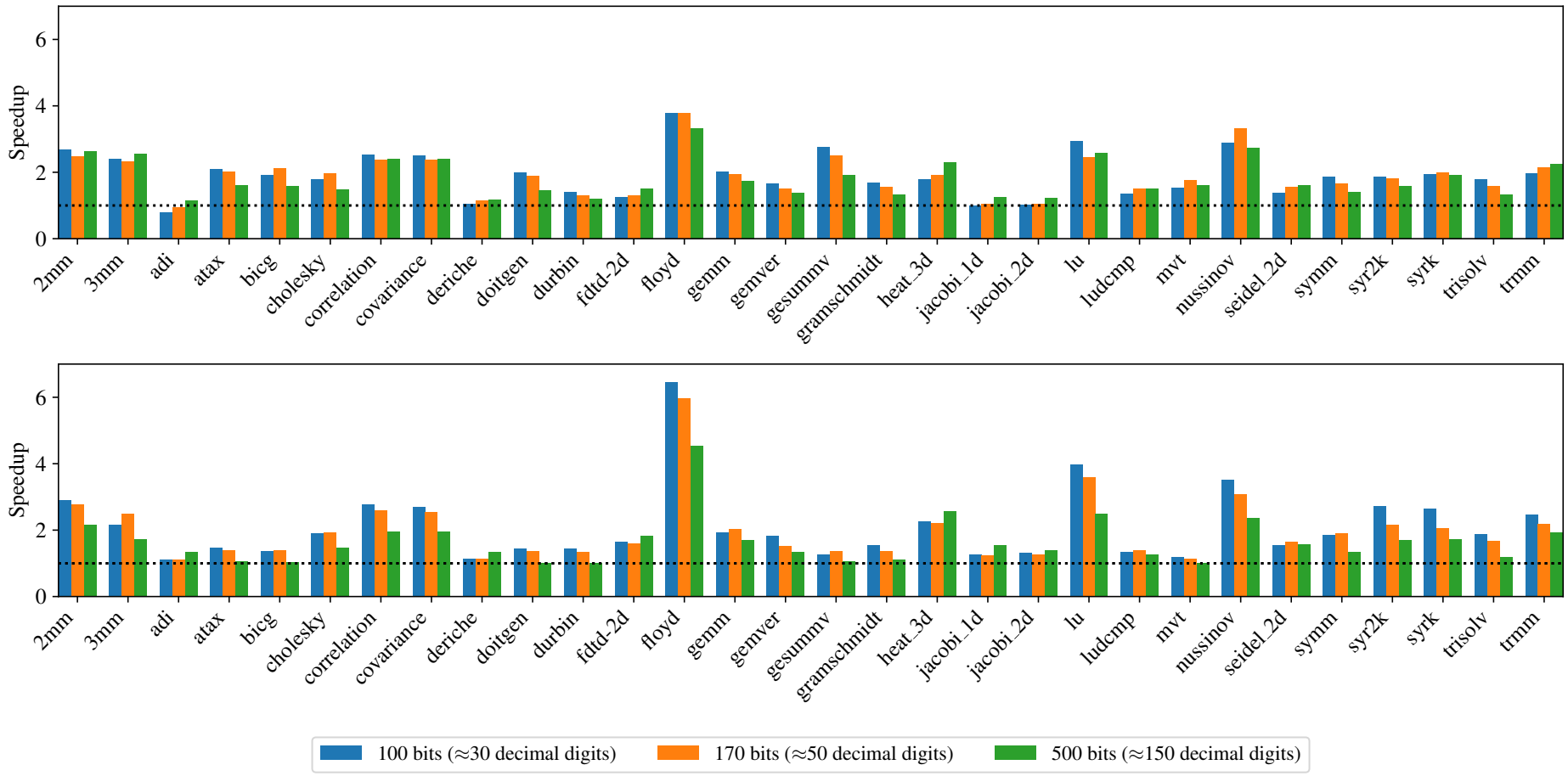


Figure 6.3: Speedup of `vpfloat<mpfr, ...>` over the Boost library for multi-precision for the Polybench benchmark suite. `vpfloat<mpfr, ...>` applications were compiled with optimization level -O1 and Boost with -O3. The execution time reference taken are the best between compilations with and without Polly. Results are shown for two different machines: an Intel Xeon E5-2637v3 with 128GB of RAM (`M1`), and an Intel Xeon Gold 5220 with 96 GB of RAM (`M2`), respectively. Y-axes are shown with the same limit to ease comparisons between results in the two machines.

6.1.1.2 RAJAPerf

The RAJAPerf benchmark suite [99] is a collection of loop-based computational kernels commonly found in HPC applications. Kernels are written such that they can be implemented in different variants: sequential (single-threaded) execution, and parallel programming models like OpenMP, and CUDA. Examples of applications are the traditional *AXPY* and *DOT* product from BLAS libraries, Polybench kernels like *2mm*, *3mm* and *adi*, vector additions and multiplications, among many others. Table 6.3 shows the list of the 43 RAJAPerf applications used in this experiment and classified according to their groups.

We compiled the suite at optimization level *-O3* for our `vpfloat<mpfr, ...>` type and compare it with the Boost library for Multi-precision, both relying on the MPFR library with equivalent precision ($\approx$ 30 decimal digits). The suite is implemented in C++, and makes use of high-level abstractions such as Lambda functions, polymorphism, dynamic dispatching, and thus, highly motivated the support of such features in our type system. All applications are compiled for sequential and parallel execution. By leveraging the integration with OpenMP, described in Section 5.3.1, we also collect results for three OpenMP variants along with the three sequential variants for a total of 6 variants of each kernel, and a total of 258 variants.

Figure 6.4 shows the speedup of `vpfloat<mpfr, ...>` over Boost. Y-axes are depicted in different log scales in order to show the speedups achieved by OpenMP variants in each machine. As was the case for Polybench, we observe speedups over most of the test suite, and slowdowns are associate with additional copies and object allocations that are needed in these specific cases. Measurements also demonstrate that a late MPFR lowering pass significantly improves performance for the majority of cases. More interestingly, our solution scales much better than Boost when using OpenMP in both systems. Hardware counter measurements, collected in `M1`, indicate that speedups in the 7–9$\times$ range stem from the reduction of memory accesses and cache misses, with up to 90$\times$ 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. Table 6.4 shows averages speedups in `M1` and `M2` for all RAJAPerf variants. We notice that systems with many more processors/threads lead to

Table 6.3: List of RAJAPerf applications classified according to their groups.

Group	Num. of apps.	Applications
apps	7	del_dot_vec_2d, energy, fir, ltimes, ltimes_noview, pressure, vol3d
basic	9	daxpy, if_quad, int3, init_view1d, init_view1d_offset, mataddsubb, nested_init, reduced3_int, trap_int
lcals	10	diff_predict, eos, first_diff, first_sum, gen_lin_recur, hydro_1d, hydro_2d, int_predict, planckian, triad_elim
polybench	12	2mm, 3mm, adi, atax, fdtd_2d, floyd_warshall, gemm, gemver, gesummv, jacobi_1d, jacobi_2d, mvt
stream	5	add, copy, dot, mul, trial

Table 6.4: Average speedups for RAJA in machines `M1` and `M2` (from Table 6.1).

	Sequential			OpenMP		
	Base	Lambda	RAJA	Base	Lambda	RAJA
`M1`	1.74	1.61	1.65	7.98	7.16	7.72
`M2`	1.90	1.77	1.86	32.94	32.57	31.67

better overall speedups, especially in OpenMP variants. This shows an important property of our solution: scalability. Because our solution scales better than the baseline, better speedups are seen in more powerful systems. Finally, these results have also been corroborated by peers through an Artifact Evaluation (AE).

Compilation time

Similar to Polybench, we also estimate the compilation time needed for RAJAPerf in the two scenarios. Once more, our solution has shown to outperform the baseline in that metric. RAJAPerf with `vpfloat<mpfr, ...>` was compiled in 313 seconds, while the Boost version required an average of 1054 seconds. In summary, with a compilation time speedup of 3.37× we are able to produce code that can be more than 32× faster than the baseline.

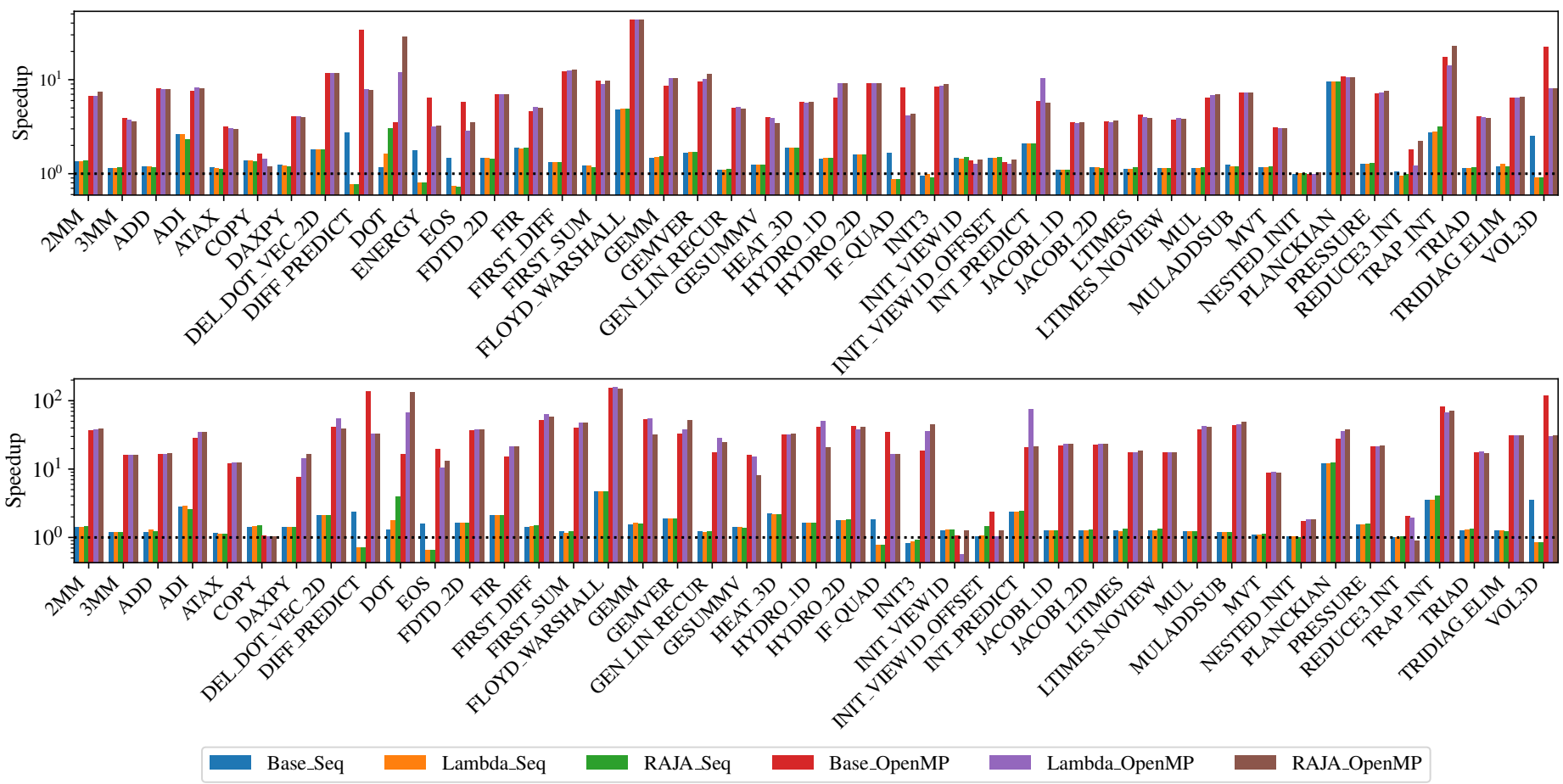


Figure 6.4: Speedup of `vpfloat<mpfr, ...>` over the Boost library for multi-precision for the RAJAPerf benchmark suite, both compiled with *-O3* optimization level.

6.1.2 Hardware(UNUM `vpfloat`) vs. Software (MPFR `vpfloat`)

We also demonstrate the effectiveness of our type extension and its integration with LLVM on a hardware implementation of the UNUM format. The main purpose of this experiment is to show the benefits of hardware support for high-precision representations. To the best of our knowledge, until 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 6.1.1.1 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 [11] connected to the UNUM coprocessor of [23]. All benchmarks including baseline MPFR implementations have been compiled to the RISC-V ISA. As explained in section 5.3.1, our MPFR backend is target independent, and hence, applications with `vpfloat<mpfr, ...>` types can potentially be executed on any LLVM-compatible platform with MPFR support.

We compiled Polybench for both baseline (`vpfloat<mpfr, ..>`), and `vpfloat<unum, ...>` type using two strategies: *-O3*, and *-O3 + polly*. Each application was compiled using three precision configurations: 100 bits (≈30 decimal digits), 170 bits (≈50 decimal digits), and 500 bits (≈150 decimal digits). The execution time for the baseline is obtained considering the best time reference, with and without Polly. The main purpose of this experiment is to compare the best result for the `vpfloat<mpfr, ...>` type with two compilation procedures for the UNUM type (with and without Polly).

Figure 6.5 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. We notice a larger speedup gap in `vpfloat<unum, ...>` (with and without Polly) than in `vpfloat<mpfr, ...>` types. `vpfloat<unum, ...>` kernels compiled with Polly have an overall speedup of 1.44× over its -O3, and this difference over `vpfloat<mpfr, ...>`(1.18×) is explained as follows: because a UNUM type has an in-place layout, the array access patterns generated by Polly better match such characteristic. Although MPFR types are accelerated by the optimizer, the heuristics available do not work as well as for UNUM types, and optimizing them for MPFR types is an interesting venue to explore as future work.

It further validates the benefits of making variable precision FP arithmetic transparent to upstream optimization passes. Notably, *gemm*, *2mm* and *3mm* show speedups 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. As expected, having support for high-precision computation lead to significant performance gains over software targets. These results show great benefits of having compiler (and hardware) support for high-precision representations, especially now with the rise of the dark silicon era [47].

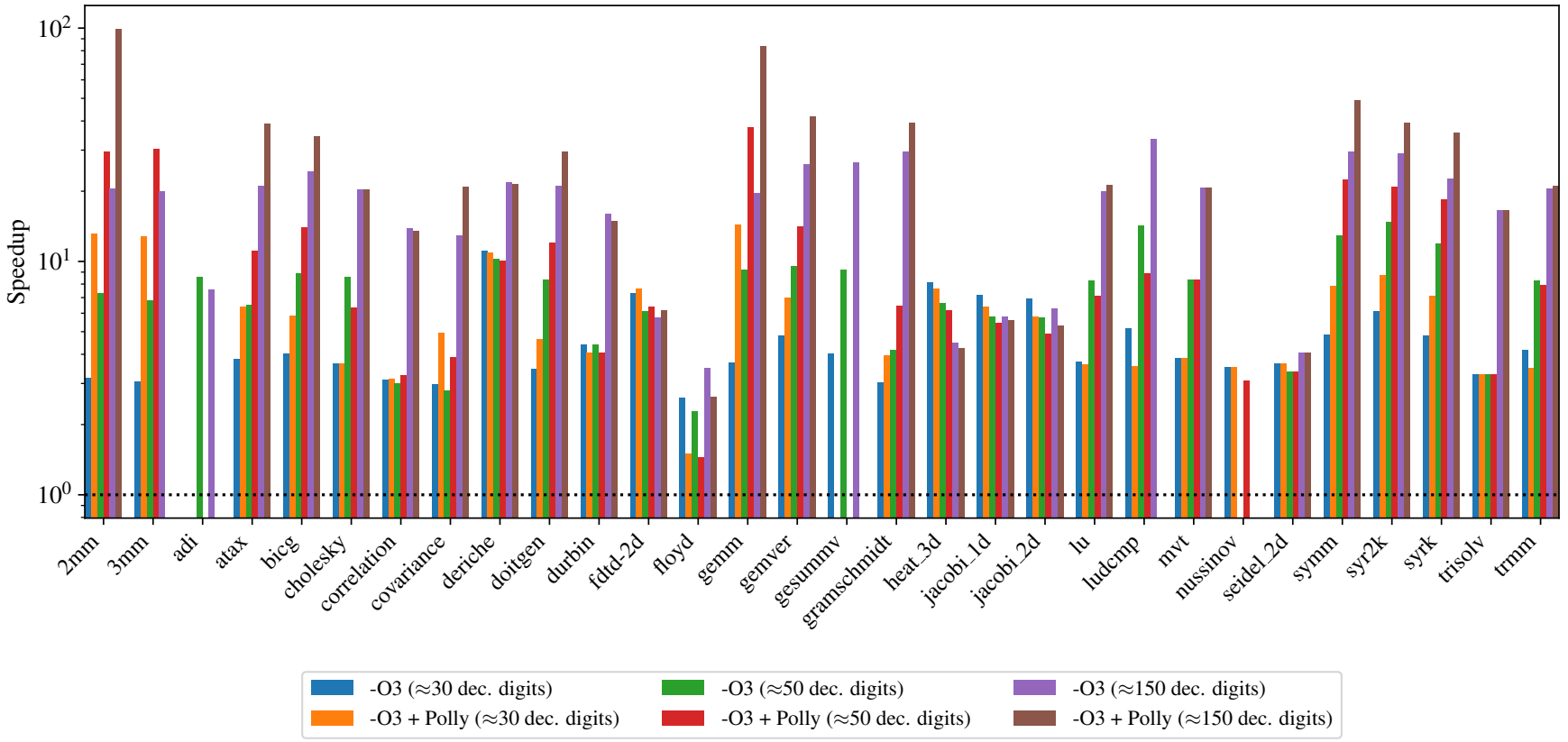


Figure 6.5: Speedup of `vpfloat<unum, ...>` over `vpfloat<mpfr, ...>` on the PolyBench suite

6.2 Linear Algebra Kernels

The first segment of this chapter showed that supporting a substrate of variable precision computing in a language and compiler enable significant acceleration when comparing with state-of-the-art solutions. These experiments demonstrate the potential of our proposal and the benefits of a compiler-integrated solution. The remaining of this chapter is dedicated to illustrating the use of variable precision in the context of Linear Algebra and is used to show interesting insights to encourage the exploration of variable precision computing in this area.

The experiments conducted in section 6.1 all made use of constant MPFR and UNUM types, as there was no need for dynamic type manipulation of any sort. In this section, we concentrate our attention on showing potential use cases of types with runtime-decidable attributes, which are not only a great exploration tool for precision-awareness in numerical applications but also essential to adaptive or variable precision computing. Although high-level languages like Julia [19] or Python [117] 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. We conclude this chapter by experimenting with multiple variants of the Conjugate Gradient (CG) linear solver method.

Conjugate Gradient (CG)

To demonstrate how our type system can be beneficial to the exploration of variable precision, we used our `vpfloat<mpfr, ...>` to implement multiple variants of the Conjugate Gradient method. The CG algorithm solves linear systems in the form of $Ax = b$ when matrix A is Symmetric Positive Definite (SPD), a common case for the resolution of partial differential equations (PDEs). It is classified as an iterative method, a mathematical procedure that uses an initial value to generate improvements of the solution with successive operations. Contrary to direct methods that have a finite (predefined) number of operations, iterative methods are based on a convergence state to determine the number of operations to execute. The method is exact in theory, but the successive roundoff errors slow down or even destroy convergence. The projective iterative method, typically exemplified by CG, has gained importance due to a low memory occupancy, typically $\mathcal{O}(N)$ rather than $\mathcal{O}(N^3)$ for direct methods. 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. In the remaining sections that cover conjugate gradient, we go through a series of requirements, properties and descriptions for CG algorithms. Four CG variants are implemented: original Hestenes and Stiefel [63] CG, preconditioned CG, pipelined CG, and BiCG. We wrap up the CG section by showing experiments and a discussion over them.

Stopping condition

A common characteristic of CG algorithms is the stopping condition required by the iterative process. We adopt the common choice of using the value of r_k, called *recursive residual*, as a *threshold*, usually called *tolerance*, for the stopping condition. When r_k is under the *threshold*, the algorithm leaves the scope and stops the execution, and the result is kept within a predictable error margin. CG algorithms have a complex behavior regarding the mathematical relation between tolerance and attainable accuracy. This involves in-depth analysis which is well beyond the scope of this thesis.

```
for (unsigned prec = 50; prec <= 2000; prec += 50) {
        cg_algo(prec, x, A, b)
}
```

Listing 6.1: Calling a precision-generic implementation of CG

Precision-generic implementation

An important property we aimed to preserve in our type system is the ability to write code that is unique so that no duplication is needed to handle the underlying precision in use. In other words, our main focus was to design a type system that is precision-generic, i.e., where code is not attained to a single configuration. CG algorithms implemented with `vpfloat<mpfr, ...>` follow this requirement and, therefore, are precision-agnostic: the core CG iteration takes a precision parameter, and every run of the function can make use of a different precision value. This allows us within a single run of the application, without recompilation, to programmatically drive experiments with multiple precision configurations. Listing 6.1 shows how one would call `cg_algo`, implemented as precision-agnostic. We iterate over a loop where precision, defined as `prec`, ranges from 50 to 2000 with a step of 50. Finally, translating a CG algorithm to `vpfloat` is straightforward: (1) there is no need to extend the precision of the matrix A, as it is read-only; (2) only working arrays have to be declared as `vpfloat`.

Original CG: Hestenes and Stiefel algorithm

The pseudo-code for the algorithm is given by Algorithm 2, where line 6 shows how r_k controls the number of iterations.

Algorithme 2: conjugate gradient: original Hestenes and Stiefel algorithm [63]

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**

One may notice how every operation can in fact be expressed as a BLAS function. The main motivation of implementing BLAS libraries for `vpfloat` types is to enable the exploration of these linear algebra algorithms. Listing 6.2 shows the implementation of the CG algorithm [63] with our *mpfrBLAS* library(from Section 4.7.1). Two stopping conditions can be seen: (1) at line 21 when the value of `rs_next` is below a threshold; (2) at line 4 when the maximum number of iterations has been reached and the algorithm is unlikely to converge. We only replaced: (1) vector-scalar products by `vpm_axpy_vvv`, (2) matrix-vector products by `vpm_sparse_gemv_vdv`, (3) dot products by `vpm_dot_vv`, and (4) products of a vector by a scalar by `vpm_scal`. Because these functions have rather similar function signatures (with the exception from the `precision`) to standard BLAS, it is fairly simple to retarget this algorithm for other library instantiations.

```
//      rs = rk'*rk
vpfloat<mpfr, 16, precision> rs = vpm_dot_vv(precision, n, r_k, 1, r_k, 1);

for (unsigned num_iter = 1; num_iter < m * 1000; ++num_iter) {

  vpm_sparse_gemv_vdv(precision, m, n, (vpfloat<mpfr, 16, precision>)1.0,
                      A, rowInd, colInd, p_k,
                      (vpfloat<mpfr, 16, precision>)0.0, Ap_k);
  alpha = rs / vpm_dot_vv(precision, n, p_k, 1, Ap_k, 1);

  // x = x + alpha*p
  vpm_axpy_vvv(precision, n, alpha, p_k, 1, x_k, 1);

  // r = r - alpha *Ap
  vpm_axpy_vvv(precision, n, -alpha, Ap_k, 1, r_k, 1);

  // rk+1
  rs_next = vpm_dot_vv(precision, n, r_k, 1, r_k, 1);

  if (sqrt((double)rs_next) < tolerance) {
    break;

  vpm_scal(precision, n, rs_next / rs, p_k, 1);
  vpm_axpy_vvv(precision, n, (vpfloat<mpfr, 16, precision>)1.0,
               r_k, 1, p_k, 1);
  rs = rs_next;
}
```

Listing 6.2: Implementation of algorithm 2 using mpfrBLAS 4.7.1.

In fact, our experiments take advantage of BLAS flexibility and retargetability, and we show different implementations of CG and variants for different formats: `vpfloat<mpfr, ...>`, `double`, `long double`,`__fp128`, and Boost for Multi-precision.

Preconditioned CG

Preconditioned CG is a slightly modified algorithm[3] that can significantly improve convergence by relying on preconditionning techniques to reduce the condition number of the matrix (see section 2.4.1). Incomplete LU or incomplete Cholesky are some examples of techniques that help in that task. However, we have only considered the **Jacobi Preconditionner** method (from Algorithm 3), since it has low computational cost and scales linearly according to matrix size. The modified algorithm is listed in 4, showing the use of both matrix A and preconditioned matrix iM (which stands for *inverse* M).

Pipelined CG

Our *mpfrBLAS* has been implemented to exploit thread-level parallelism through OpenMP. Extending beyond multi-core architectures, our type system could potentially benefit from many-core systems with the use of MPI [110] or related models. Aside from matrix-vector products, the original standard algorithm cannot be easily parallelized due to the reuse of variable arrays between lines. Ghysels and Vanroose [51] proposed to hide global synchronization latency of the preconditioned CG with a variant that offers further opportunities of parallelism

[3] In many libraries, the conjugate gradient algorithm implements the preconditionning by default.

Algorithme 3: Jacobi preconditioner

1: **for** i = 1 to m **do**
2: **for** j = 1 to n **do**
3: **if** i == j **then**
4: $iM[i,j] := \frac{1}{A[i,j]}$
5: **else**
6: $iM[i,j] := 0$
7: **end if**
8: **end for**
9: **end for**
10: return(iM)

Algorithme 4: preconditionned conjugate gradient: general algorithm

1: $r_0 := b - Ax_0$
2: $z_0 := iMr_0$
3: $p_0 := z_0$
4: **while** iteration count not exceeded **do**
5: $\alpha_k := \frac{r_k^T r_k}{p_k^T Ap_k}$
6: $x_{k+1} := x_k + \alpha_k p_k$
7: $r_{k+1} := r_k - \alpha_k Ap_k$
8: **if** $\| r_{k+1} \| \leq tol$ **then** break
9: $z_{k+1} := iMr_{k+1}$
10: $\beta_k := \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
11: $p_{k+1} := z_{k+1} + \beta_k p_k$
12: $k++$
13: **end while**

than those of traditional algorithms. Algorithm 5 shows the implementation of pipelined CG. Matrix-product $A \times w_k$ (line 9) can be done in parallel to calculations from lines 10 and 11. Although this property adds more parallelism to CG, its drawback is the loss of numerical stability the algorithm is likely to observe when compared to traditional methods [31].

BiCG

While many problems lead to the handling and computation with SPD matrices, there are cases where a more general matrix must be employed and none of the CG algorithms previously described can be adopted. BiCG [104] extends the use of CG to general matrices (and not only to SPD). It is similar to the original CG but requires some additional computation to account for matrix generality.

Results and Discussion

To demonstrate the use of variable precision and how our type system can be beneficial to this exploration, we have implemented and analyzed the impact of different precision values in all four CG variants (Hestenes and Stiefel [63], preconditioned, pipelined CG, and BiCG) for two constraints: number of iterations and execution time. BLAS libraries have been implemented with the following types: `long double`, `__fp128`, `vpfloat<mpfr, ...>`, and Boost `mpfr_float`

Algorithme 5: Pipelined conjugate gradient algorithm [37]

$p_0 := r_0 := b - Ax_0$
$s_0 := A \times p_0, w_0 := A \times r_0$
$z_0 := A \times w_0, \alpha_0 := \frac{r_0^T r_0}{s_0^T p_0}$
while iteration count not exceeded **do**
  $x_k := x_{k-1} + \alpha_{k-1} p_{k-1}$
  $r_k := r_{k-1} - \alpha_{k-1} s_{k-1}$
  $w_k := w_{k-1} - \alpha_{k-1} z_{k-1}$
  if $\| r_k \| \leq tol$ **then** break
  $q_k = A \times w_k$
  $\beta_k := \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}$
  $\alpha_k := \frac{r_k^T r_k}{r_k^T w_k - \frac{\beta_k}{\alpha_k - 1} r_k^T r_k}$
  $p_k := r_k + \beta_k p_{k-1}$
  $s_k := w_k + \beta_k s_{k-1}$
  $z_k := q_k + \beta_k z_{k-1}$
  $k++$
end while

Algorithme 6: biconjugate gradient: general algorithm

$p_0 := r_0 := b - Ax_0$
choose r_0^* such as $(r_0 r_0^*) \neq 0$
$p_0^* := r_0^*$
while iteration count not exceeded **do**
  $\alpha_j := \frac{r_j^T r_j^*}{p_j^{*T} A p_j}$
  $x_{j+1} := x_j + \alpha_j p_j$
  $r_{j+1} := r_j - \alpha_j A p_j$
  if $\| r_{j+1} \| \leq tol$ **then** break
  $r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
  $\beta_k := \frac{r_{j+1}^{*T} r_{j+1}}{r_j^T r_j^*}$
  $p_{j+1} := r_{j+1} + \beta_j p_j$
  $p_{j+1}^* := r_{j+1}^* + \beta_j p_j^*$
  $j++$
end while

type. We also use OpenBLAS [124] as a reference implementation, that way an optimized version of each algorithm in `double` is also provided.

We used 40 squared SPD matrices from Matrix Market [24, 86] with problem sizes (N) in the range of 132 to 5489. Matrices are stored as sparse using the Compressed Row Storage (CSR) format and can have up to 315891 non-zero elements (*nnz*). We selected 12 out of 40 to be displayed in Figures 6.6 and 6.7 (for the results of *Number of iterations vs. Precision*) and Figures 6.8 and 6.9 (for the results of *Execution Time vs. Precision*). The remaining matrices are included as appendices of the thesis to keep this chapter at a reasonable length.

Every application and matrix was executed using five configurations: `double` (OpenBLAS), `fp80`, `fp128`, `vpfloat<mpfr, ...>` (from 150 to 2000 bits, step = 50), and Boost for Multi-precision (from 150 to 2000 bits, step = 50). The convergence threshold (*tol*) was selected as

$1e^{-10}$ in order to keep results within a reasonable degree of accuracy. A missing type in a figure means that the algorithm has not converged for that configuration. For example, pipelined CG did not converge for types `double` and `fp80` in the *nasa2910* matrix, and thus, they are not displayed in Figure 6.6. Execution times for the `double` type (in Figures 6.8 and 6.9, and appendix B) are not shown because they are usually at a smaller order of magnitude than other types. Experiments were conducted in `M2`, and could not be conducted with our `vpfloat<unum, ...>` in a RISC-V environment because the system has been ported to a new platform, and is still encountering stability issues.

Number of Iterations

Original CG, BiCG and Preconditioned CG

The impact of precision on standard matrices taken from Matrix market confirms a well-established result: in the general case, the higher the precision, the fewer iterations it takes to converge. In matrix *bcsstk19*, for example, OpenBLAS needs 295400 iterations to converge, while going to high precision requires a little more than 70000 for 150 bits of precision, and around 4000 when using more than 1000 bits. That is, high precision allowed a convergence speed of more than 700× when compared to OpenBLAS.

This observation can be spotted in 12 out of 12 matrices in the original, and BiCG algorithms. Indeed, when comparing the results of these two algorithms, one may also confirm that BiCG is a generalization of the original CG for non-SPD matrices since both have an equivalent convergence factor. This generalization comes at a cost: the additional computation can double the execution time.

Preconditioned CG is, without a doubt, the best-performing algorithm for all matrices analyzed. Applying a preconditioner beforehand decreases the condition number and helps to reduce significantly the number of iterations in almost all situations. The single situation where even preconditioning does not help is for the ill-conditioned matrix *crystk01*. Convergence is only achieved with the use of high-precision representations like *FP128*, or higher.

Remedying the stability of pipelined CG with high precision

Perhaps the most interesting result from these experiments come from analyzing *pipelined CG*. The algorithm is mainly proposed as a viable CG variant to run in many-core (distributed) systems because it offers a higher degree of parallelism than other CG implementations. However, it has shown to suffer from local rounding errors that accumulate in the vectors recurrences [31, 37]. In fact, our results show and confirm that *pipelined CG* undergoes stability issues when executed with OpenBLAS[4]. Differently from other variants that can be stable for many of the matrices analyzed, pipelined CG has not reached the stability necessary to converge in any of these cases. Besides, our experiments show an aspect that, to the best of our knowledge, has not being spotted by related work: increasing precision can help to minimize local rounding errors and offer the stability necessary for convergence. While many researchers have proposed to address *pipelined CG*'s instability with new implementations to compensate those errors [34, 35, 36], we show a simpler solution that relies on increasing the computation precision of the application for regaining stability. Still, these results are significantly attained to the tolerance we used ($1e^{-10}$), which is chosen to keep highly accurate results. If the tolerance was increased, this algorithm would certainly be less susceptible to convergence issues, an aspect to consider in the future.

[4]Notice that convergence for `double` is not shown in the results of the *pipelined CG* algorithm because no convergence is reached.

An interesting observation is the non-linear behavior of the number of iterations for a linear increase in precision, an effect found in all variants, and mainly caused by the accumulation of rounding errors [30] subjected through the multiple matrix-vector and vector-vector products. Another common characteristic of all instances relates to the precision in use: they all reach a minimum plateau value of the number of iterations needed for convergence. Some matrices reach this plateau with few hundreds of bits, while others may require thousands of bits. These results are an interesting exploration venue on numerical analysis aiming to determine *a priori* the best precision value to adopt as to exhibit a good trade-off between the number of iterations and execution time.

Because our type system can be precision-agnostic, we are able to run experiments and collect results for multiple-precision configurations with a single run of each application, *without recompilation*, and *code duplication*. Although results for Boost are also collected using the same approach, our solution still manages to outperform Boost by at least 1.35×, and it is still the only solution available with a real possibility of hardware integration, a venue that will be explored as part of our future work.

We would like to emphasize that, by no means, these results should be seen as mathematical proofs of how precision augmentation is beneficial. This study is merely an indicator that increasing precision in applications can have an interesting venue of exploration in CG benchmarks. Mathematically analyzing these results require a much more in-depth understanding of the nature and internal properties of CG, and is out of the scope of this thesis.

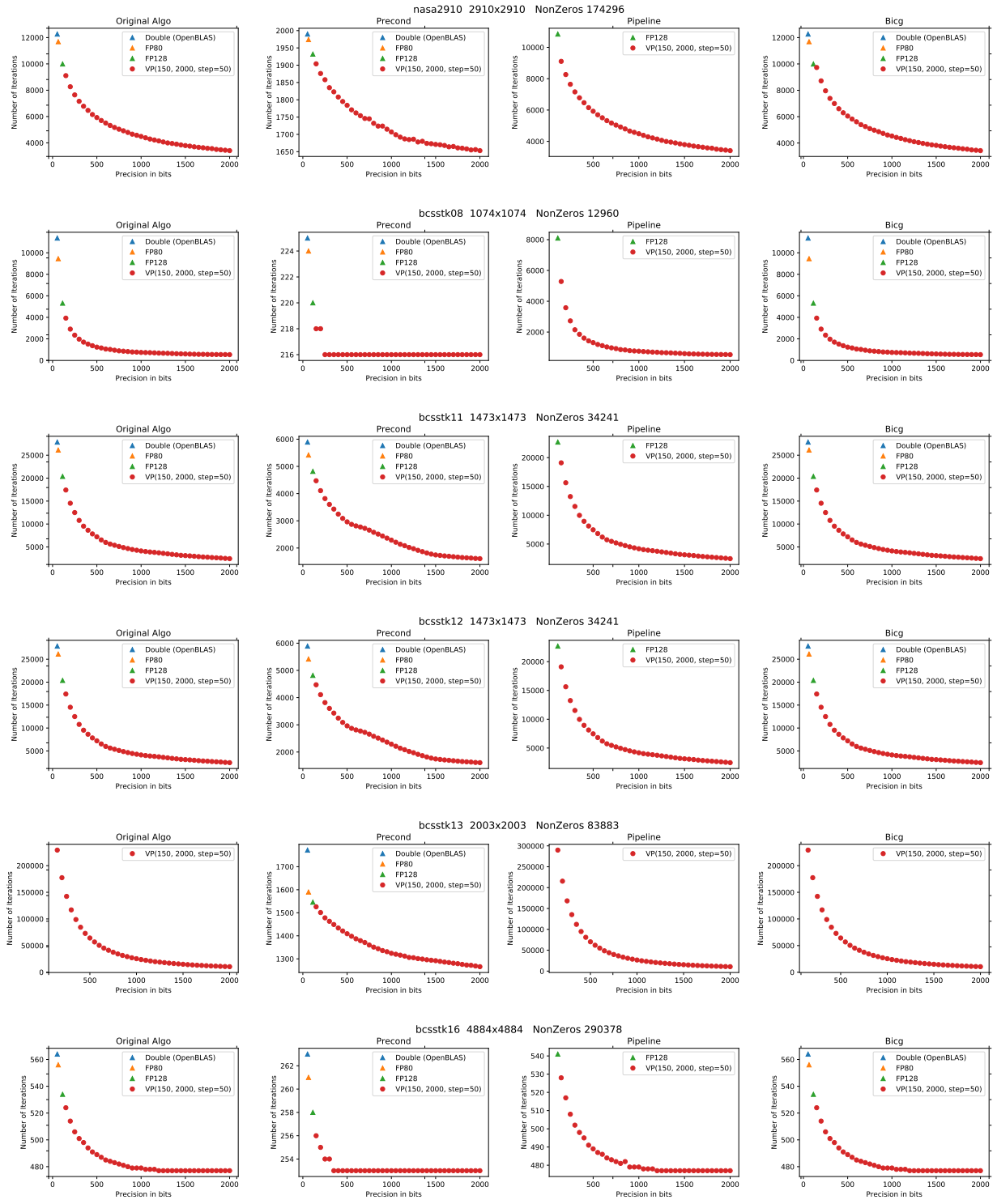


Figure 6.6: CG variants with multiple formats with matrices *nasa2910, bcsstk08, bcsstk11, bcsstk12, bcsstk13, and bcsstk16* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

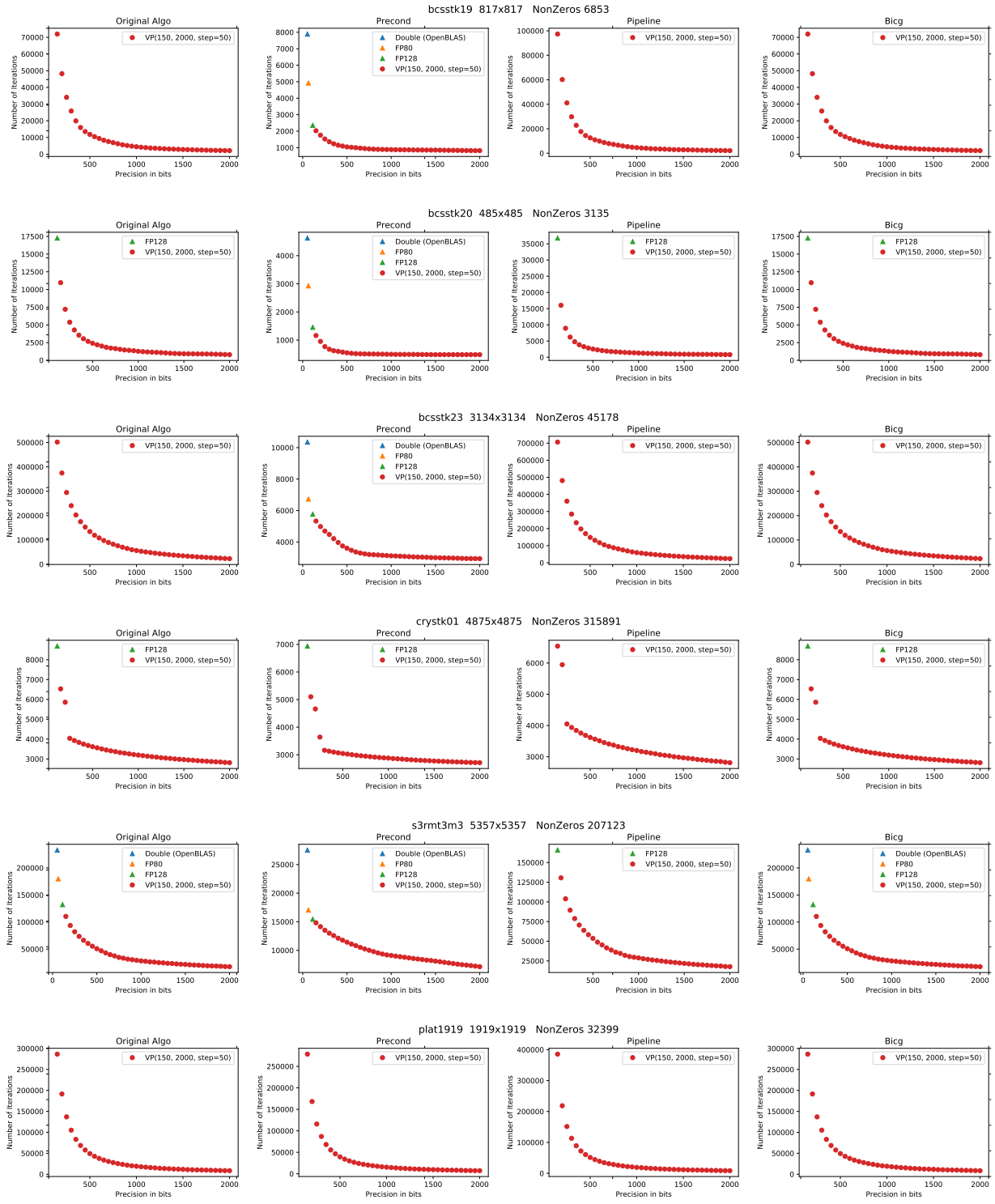


Figure 6.7: CG variants with multiple formats with matrices *bcsstk19, bcsstk20, bcsstk23, crystk01, s3rmt3m3, and plat1919* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

Execution time

It may initially sound paradoxical that higher precision can lead to fewer iterations, however, this also demonstrates that higher precision may actually reduce the execution time of a numerical application, as depicted in Figures 6.8 and 6.9. We observe that many variants are indeed able to significantly improve execution time by relying on very high precision. For instance, all variants running in matrix *bcsstk13*, except *precond*, with 2000 bits of precision show more than 5× speed when compared to 150 bits.

Although high-precision representation can improve performance in different variants and matrices, the trend followed by execution time is not the same as for the number of iterations. We can observe that, for many situations, the reduced number of iterations is not able to compensate for the increase of precision, which results in higher execution times. This is the case for all variants in matrix *bcsstk16*, results that correlate to the small improvement in the number of iterations when precision is increased.

Precond has an interesting property: given that the condition number of the matrix is (highly) improved with a *preconditioner*, increasing the precision has two more perceptible effects: (1) either the plateau is reached earlier, (2) or increasing precision has a negative impact on execution time and should be avoided. This variant has also shown the best overall execution times, which explains why it is the preferred implementation in numerical libraries.

Interestingly, numerical analysts can potentially make use of these results with the aim to determine the best trade-off between precision and the number of iterations for an optimal execution time. The objective would be to comprehend the behavior of the number of iterations and the impact it has on execution time, as an *Oracle* for the best representation to be used[5].

Table 6.5 summarizes how execution time in `vpfloat<mpfr, ...>` compares to other types, showing the counts of the number of matrices where our type outperforms its counterparts. Values for `vpfloat` and Boost are cherry-picked with the best execution time among the precision range (from 150 to 2000 with a step=50). We observe that `vpfloat<mpfr, ...>` can outperform lower-precision types, like `long double` and `__float128`, in many matrices and variants. We also show better performance of Boost in the majority of cases, which corroborates with the results presented in previous sections of this chapter. Lastly, execution times for `vpfloat` are expected to be improved when UNUM type is used.

Table 6.5: Count on the number of matrices where `vpfloat<mpfr, ...>` outperforms other types. Only matrix with types that converge are considered. For `vpfloat` and Boost, we cherry-pick the best execution time among the precision range (from 150 to 2000 with a step=50)

Algorithm Variant	`double`	`long double`	`__float128`	Boost		
				(All mat.)	(N>1000)	(N>2000)
Original	4/32	19/32	25/35	40/40	27/27	15/15
Precond	0/36	6/37	9/39	40/40	27/27	15/15
Pipeline	0/0	7/7	29/34	32/40	26/27	15/15
BiCG	4/32	18/32	25/35	40/40	27/27	15/15

[5]Assuming that all types are supported by hardware. Otherwise, the right choice will most likely be `double` for all situations.

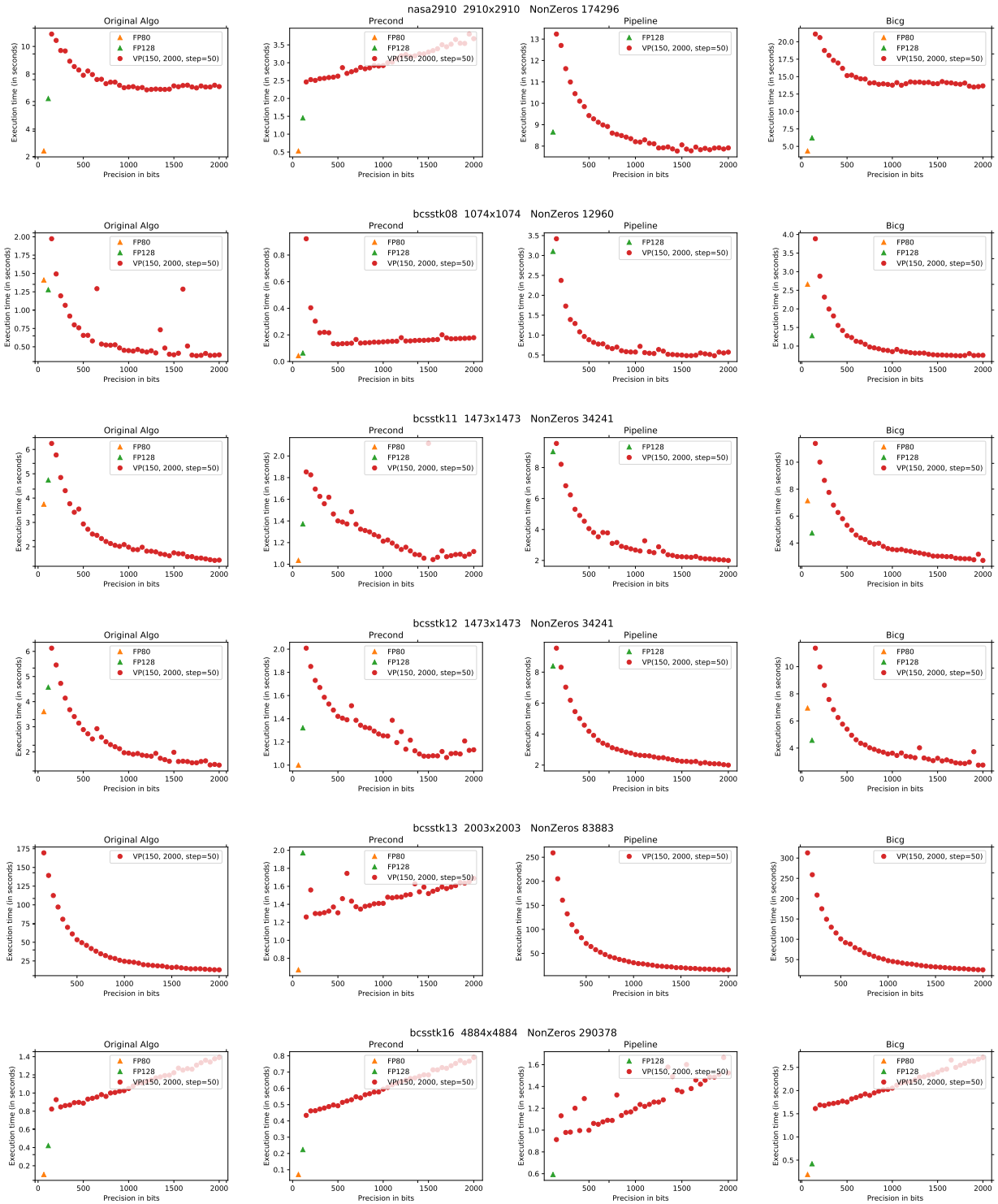


Figure 6.8: Execution time for CG variants with multiple formats with matrices *nasa2910, bcsstk08, bcsstk11, bcsstk12, bcsstk13, and bcsstk16* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

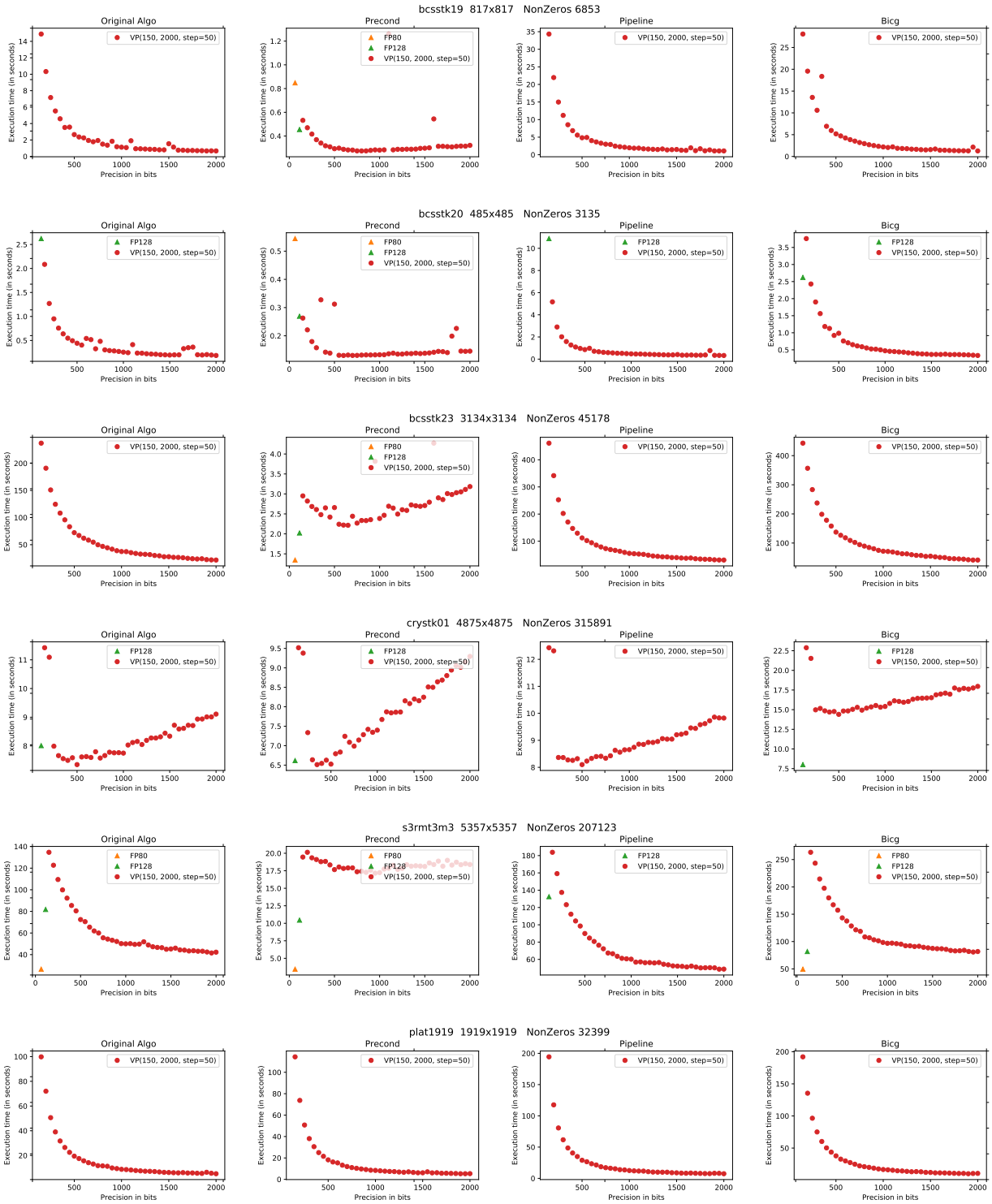


Figure 6.9: Execution time for CG variants with multiple formats with matrices *bcsstk19, bcsstk20, bcsstk23, crystk01, s3rmt3m3, and plat1919* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

6.3 Conclusion

This chapter demonstrated the productivity benefits of our intuitive programming model and its ability to leverage an existing optimizing compiler framework. We presented a series of experiments on PolyBench and RAJAPerf suites that yield strong speedups for both software and hardware targets. Our compiler integrated solution is more capable of leveraging the classical optimizations already implemented by the compiler, which helps to reason about the speedup results observed. Experimental results also showed that significantly higher speedups are observed when comparing a software to a hardware target. This result is particularly interesting with the rise of the dark silicon area and may be an aspect for evaluation on future processor designs. Our last experiments in CG are a step towards better understanding the impact of (high-)precision in linear algebra and illustrated how variable precision may raise its importance for the future of high-performance computing. To that end, this thesis is a step towards making variable precision easier to use and better integrated into a language, compiler and hardware, and serves as an enabler for better exploration of variable precision computing.

Chapter 7: Conclusion

This thesis defines a compilation strategy to use and explore variable precision FP arithmetic and proposes a design and implementation of this strategy. We are mainly interested in how languages and compilers can support data structures, types, and both software- and hardware-enabled operations to accelerate applications with different FP requirements. No previous solution has shown a complete integration of FP arithmetic with constant and variable memory footprint. Our proposed type system, embedded into the compiler's intermediate representation, and lowering and backend code generation strategies provide both high-productivity and high-performance with variable precision FP formats. Our extension supports FP arithmetic of arbitrary representation and precision, and the precision and exponent can be configured at compilation time and runtime.

We exposed a list of questions in chapter 2 and the work presented in the context of this thesis is intended to answer these questions. In the following paragraphs, we provide a summary of how we answered these questions:

How can languages and compilers be used to accelerate and improve the productivity of multi-precision FP libraries? Can one improve the integration between compilers and these libraries to take better advantage of classical compiler optimizations?

Chapters 4 and 5 show the solution we propose to enable the instantiation of multi-format FP types. Because of the genericity of our type system, types that hide underlying implementation with multi-precision FP libraries can be supported. More precisely, an MPFR type (`vpfloat<mpfr, ...>`) has been added as a lightweight alternative to use the MPFR library in a more programmer-friendly fashion. This type provides a functionality similar to high-level MPFR abstractions, like Boost Multi-precision library, Julia and Python's dynamic type systems, but it offers a C-level easy-to-use interface for MPFR.

We provide an integrated compiler solution that bridges the abstraction gap between multi-precision FP libraries and compiler transformations, permitting us to postpone the instantiation of library-related operations and objects until after optimizations are run. Experimental results corroborate on showing that a later transformation pass to generate library-specific code has a positive impact on execution time since more optimization opportunities can be found. We also demonstrate through a practical code example (Figure 5.6) that even vectorization can be supported at least to some extend.

How can one extend languages and provide compiler support for new formats taking into consideration their singular properties?

Chapter 4 describes the language requirements, syntax, and semantics defined within the context of this thesis. We use a new primitive type `vpfloat` to encapsulate many properties of

FP types. The use of a single keyword, although not necessary, is employed to allow multiple formats to coexist within a single context, and differentiates language pre-existing types from those of our extension. The proposed syntax offers flexibility for the declaration of formats with different attributes. Its use is not hindered by any constraint, such as precision and exponent values, or even size, and users should not be restrained by any form of type declaration. In other words, the support for FP arithmetic of arbitrary representation, whose precision and exponent can be configured at compilation time and runtime, allows a high degree of flexibility that has not been explored in previous works.

The effectiveness of our type extension (and language syntax) in representing multiple formats is shown through: (1) an MPFR-compatible type that views FP attributes as *exponent* and *precision* fields, and (2) a UNUM type class where FP attributes are interpreted according to the format specification, and so, differently from the former. To the best of our knowledge, no previous work tackled the integration within an optimizing compilation flow, taking into consideration format-specific attributes, and how the compiler can efficiently generate code for types that may not necessarily have constant size. Having support for two types with different semantics is proof of how extensible our solution is, not only from a language perspective but also throughout the full compilation flow, enabling the design of software- and hardware-specific code generators.

What are the compiler and optimizations requirements to support an FP type system with runtime capabilities? How can compilers provide proper memory management for these types?

Our proposal tackles a set of capabilities neglected by low-level programming languages: the ability to declare types with runtime capabilities. This feature has re-gained significance in the context of variable precision where the cost of increasing precision must be avoided for performance purposes. We enable the use of types with runtime-decidable attributes by allowing their specialization with `integer` variables of the program.

Our compilation flow handles these types from language to the middle-end level representation through a new type extension that enable declarations to have constant or runtime-decidable attributes. We implemented this extension in LLVM to benefit from the optimizations available on its intermediate representation. We also assessed that specific optimizations such as *Loop Idiom Recognition* and *Inlining* need modifications to handle types with runtime attributes. Backend code generation for such types is only possible thanks to the coordination between the compiler infrastructure and the target backend, which means that a compatible ISA or software target solution is required to drive the flexibility of these types.

The use of MPFR is possible because objects can be constructed with the precision value specified upon object creation, and operations make use of this information to compute values. The synergy between the UNUM coprocessor and the compiler is done through an ISA agnostic to the format used for computation. The compiler sets status registers values inside the hardware to control the FP format in use, which gives the compliance and full support for UNUM types with runtime capabilities.

Can variable precision serve as an interesting exploration paradigm in the context of numerical algorithms?

Yes, our experiments on the conjugate gradient method and variants show a clear variable precision exploration aspect in linear algebra. Experimental results show that increasing the precision to be used can have a non-negligible impact on the number of iterations of an algorithm. It also demonstrates that precision is particularly important for the *pipelined CG* variant due

to instability caused by the accumulation of rounding errors. The stability of the algorithm is only achieved with high precision and using primitive data types only works for it to some extent, that is, when the tolerance threshold is increased. In that case, it means that an approximate (and less accurate) solution is acceptable. The design of BLAS routines also permits the exploration of other high-performance algorithms, such as Singular Value Decomposition (SVD), and Generalized minimal residual method (GMRES).

Perspectives

Our future work can be divided into two main categories: (1) the first relates to additional compiler work and tooling that can be envisioned; (2) and the complementary work aimed in the application side with many aspects to consider.

Compilation and Integration Tools

Chapters 4 and 5 show the main contributions found within the context of this thesis. But they also show some limitations and ideas that can be pursued as future work, as part of the language, compilation, and tool integration process. Among them, we may refer to:

(1) Improve high-level structures integration: `vpfloat` support for compound types such as *class*, and *structs*, is still limited to constant types. There are two challenges to resolve: (1) find a consistent syntax and semantic to express types with runtime attributes so that users are able to bind them to variable declarations; (2) instrument the compiler to support them is also a challenging part of the work, since none of the elements within the current infrastructure has functionality that resembles it,

(2) Generally work on the improvements of loop nest-related heuristics to better optimize `vpfloat`. Although we are able to leverage Polly, many heuristics are suboptimal and are not capable of optimizing the code, even for constant types. Add support for dynamically-sized types in Polly heuristics is also a venue of exploration in order to improve cache utilization within our BLAS library implementations,

(3) Our language proposal shows support for two formats (MPFR, and UNUM), but it is also extensible to other representations. More interestingly, the Posit format, a UNUM successor, has shown potential usage for many scientific fields [28, 42, 70], and is a valuable candidate to appear as a `vpfloat` format within our extension. Ongoing work is also underway to support an IEEE-like representation, much similar to the MPFR format, but enabling hardware support to accelerate computation with high-precision representations even further,

(4) Improve tooling integration with open-source projects like PetSC [3, 18], Armadillo [106], and Eigen [56]: many of the related work use, experiment, and validate their solution with open-source projects. The evaluation presented in [34, 35, 36], for example, are conducted in PetSc [3, 18], a modern and scalable ordinary differential equation (ODE) and differential algebraic equations (DAE) solver library. The Armadillo [106] and Eigen libraries [56] are template-based metaprogramming C++ libraries for linear algebra and scientific computing and are also highly used within the context of the applications targeted to be explored in future work.

Applications

On the application side, we can explore different aspects:

(1) Run CG experiments with our `vpfloat<unum, ...>` types once all stability issues are resolved in the system. We are mainly interested to understand how the reduced number of iterations impacts execution time. Increasing precision may collaborate to reduce execution time even if compared to optimized implementations with vectorization capabilities (for example, applications with `double` and using vector instructions).

(2) Although this thesis focuses mainly on providing language and compiler integration as a mean to accelerate exploration of variable precision and other FP formats in general, our last experiments with CG show insights to increase the interest of numerical analysts in the exploration of variable precision. Future work will focus on new applications and configurations to accelerate linear algebra. We are keen to see if variable precision can be used with the iterative refinement technique presented in chapter 3 as part of the mixed-precision computing section. It is also envisioned that other applications, such as SVD and GMRES, be studied to widen the comprehension of the impact of precision and FP formats in general. The outcome of this thesis can be seen as an enabler of this exploration.

Annexes

Appendix A: CG Experimental Results: Number of Iterations

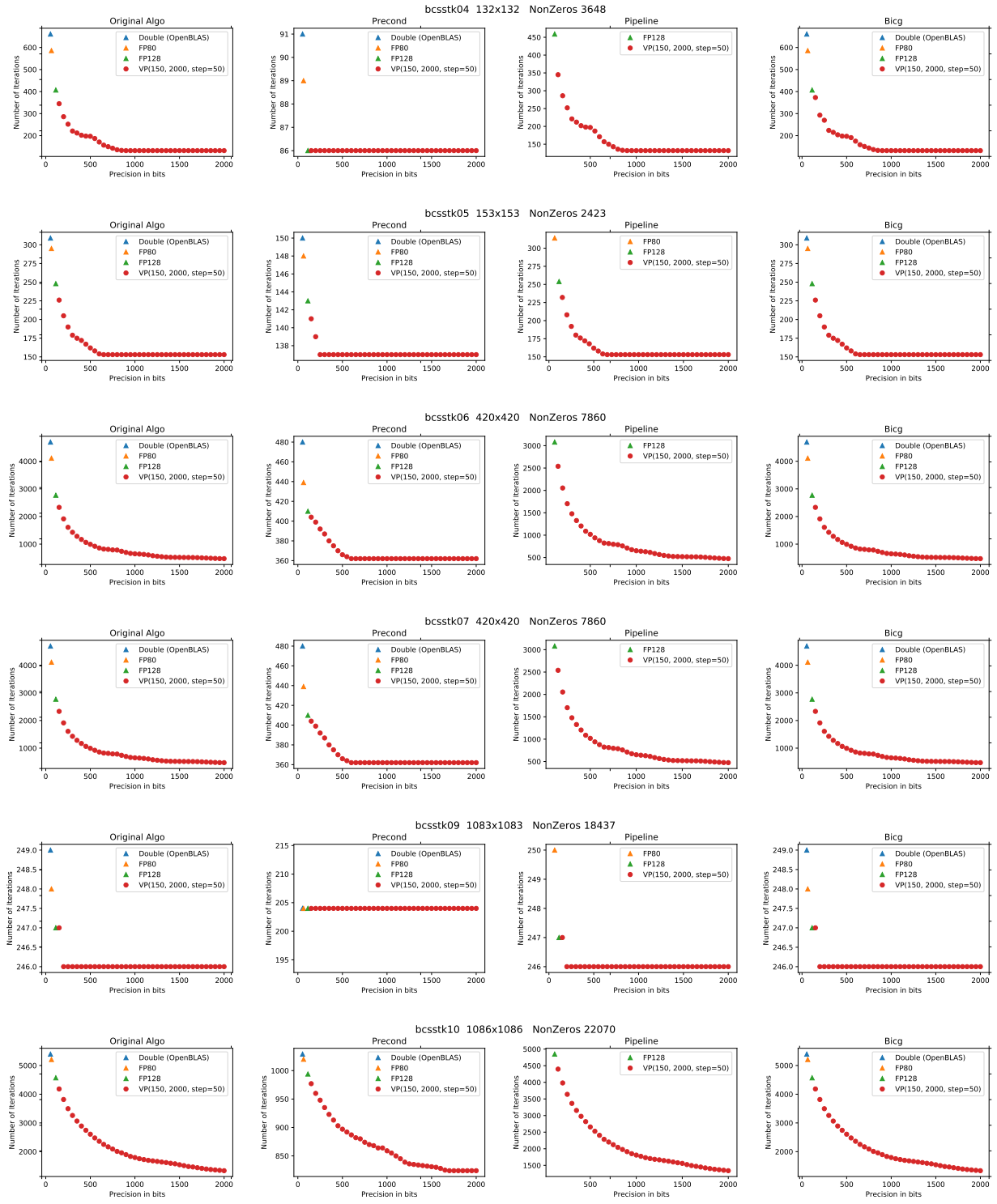


Figure A.1: CG variants with multiple formats with matrices *bcsstk04, bcsstk05, bcsstk06, bcsstk07, bcsstk09, and bcsstk10* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

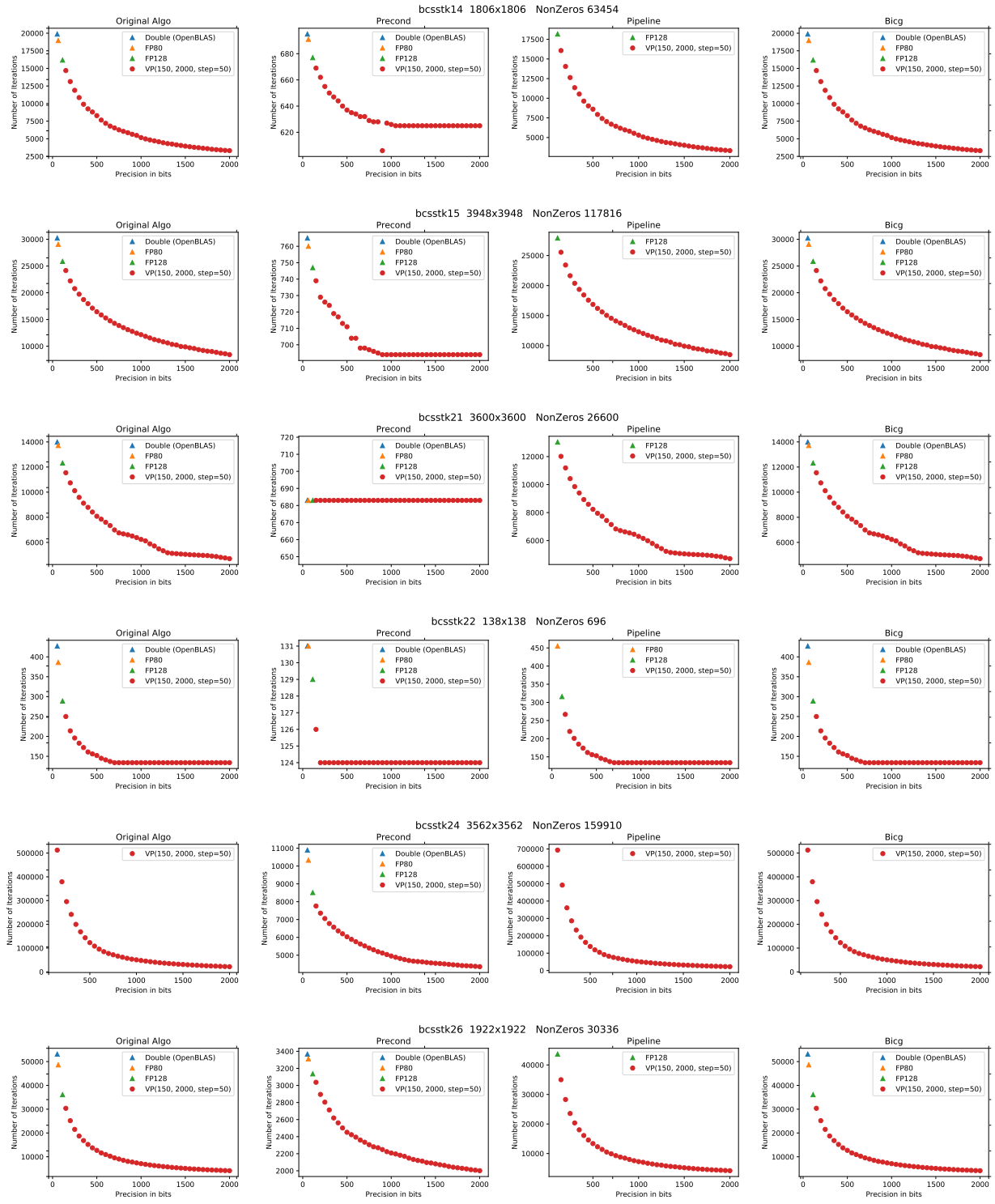


Figure A.2: CG variants with multiple formats with matrices *bcsstk14, bcsstk15, bcsstk21, bcsstk22, bcsstk24, and bcsstk26* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

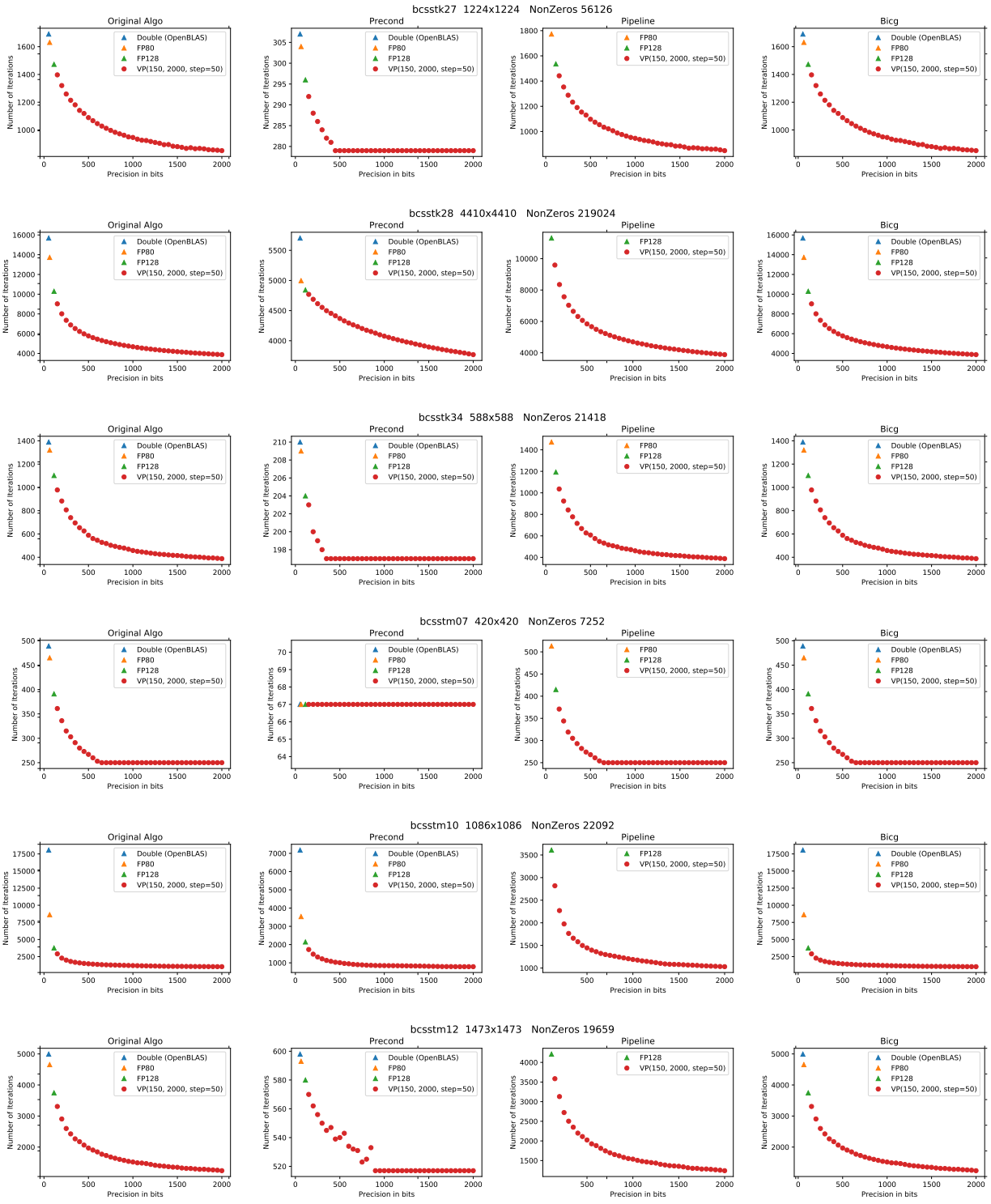


Figure A.3: CG variants with multiple formats with matrices *bcsstk27, bcsstk28, bcsstk34, bcsstm07, bcsstm10, and bcsstm12* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

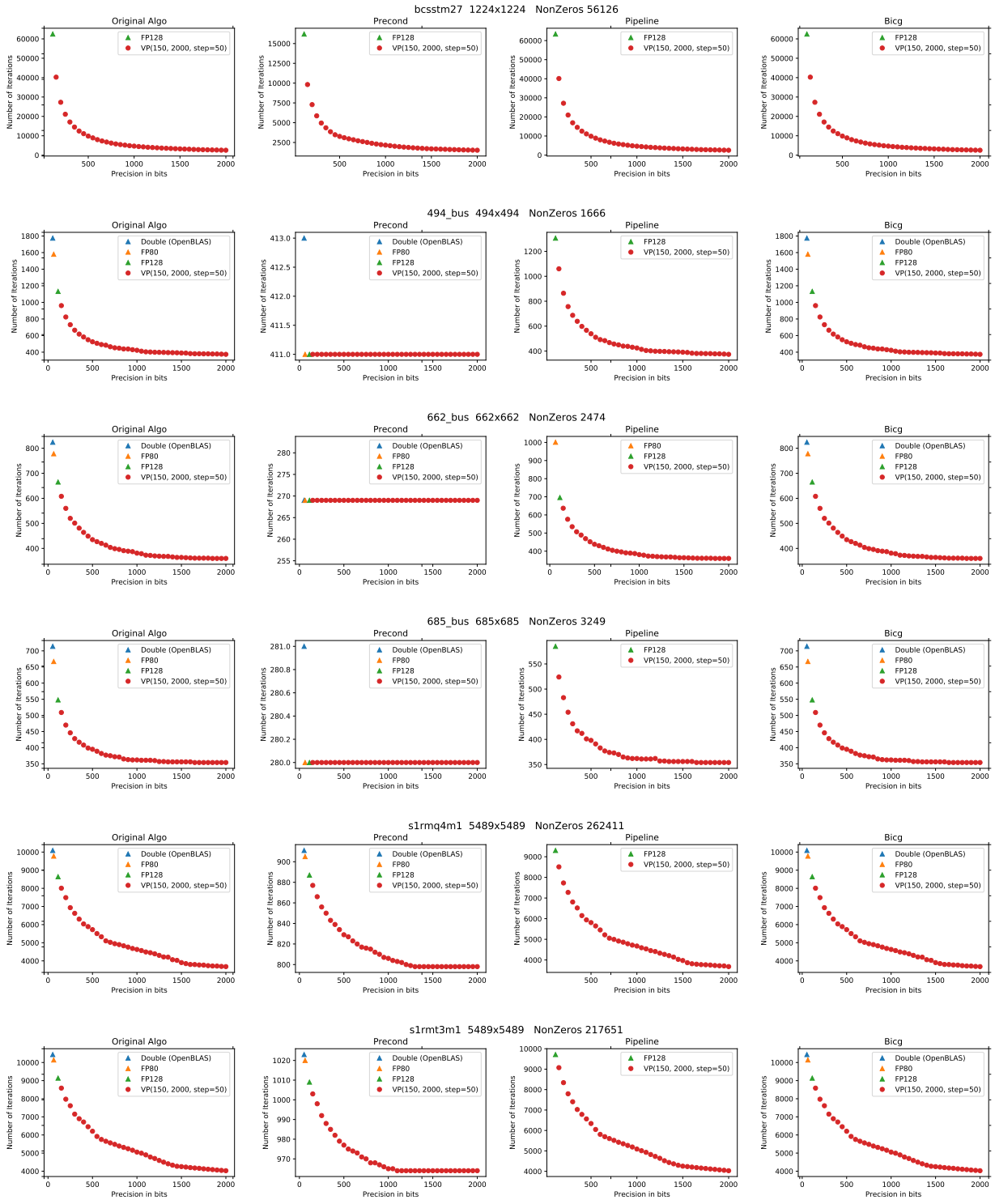


Figure A.4: CG variants with multiple formats with matrices *bcsstm27, 494_bus, 662_bus, 685_bus, s1rmq4m1, and s1rmt3m1* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

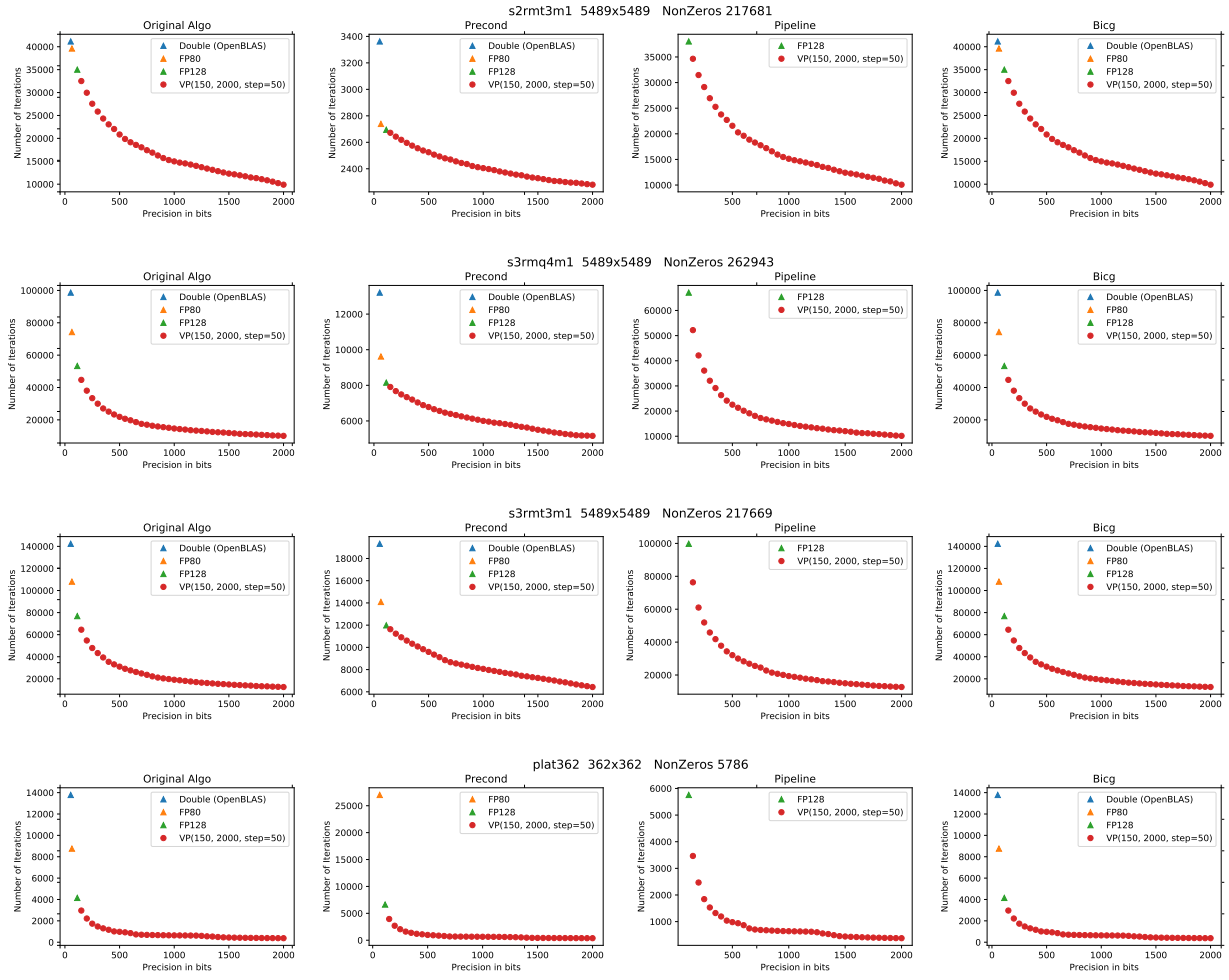


Figure A.5: CG variants with multiple formats with matrices *s2rmt3m1*, *s3rmq4m1*, *s3rmt3m1*, *and plat362* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). Y-axes show the number of iterations needed to converge for *precision in bits* between 150 and 2000 with a step=50. A missing type in a graph implies the algorithm did not converge.

Appendix B: CG Experimental Results: Execution Time

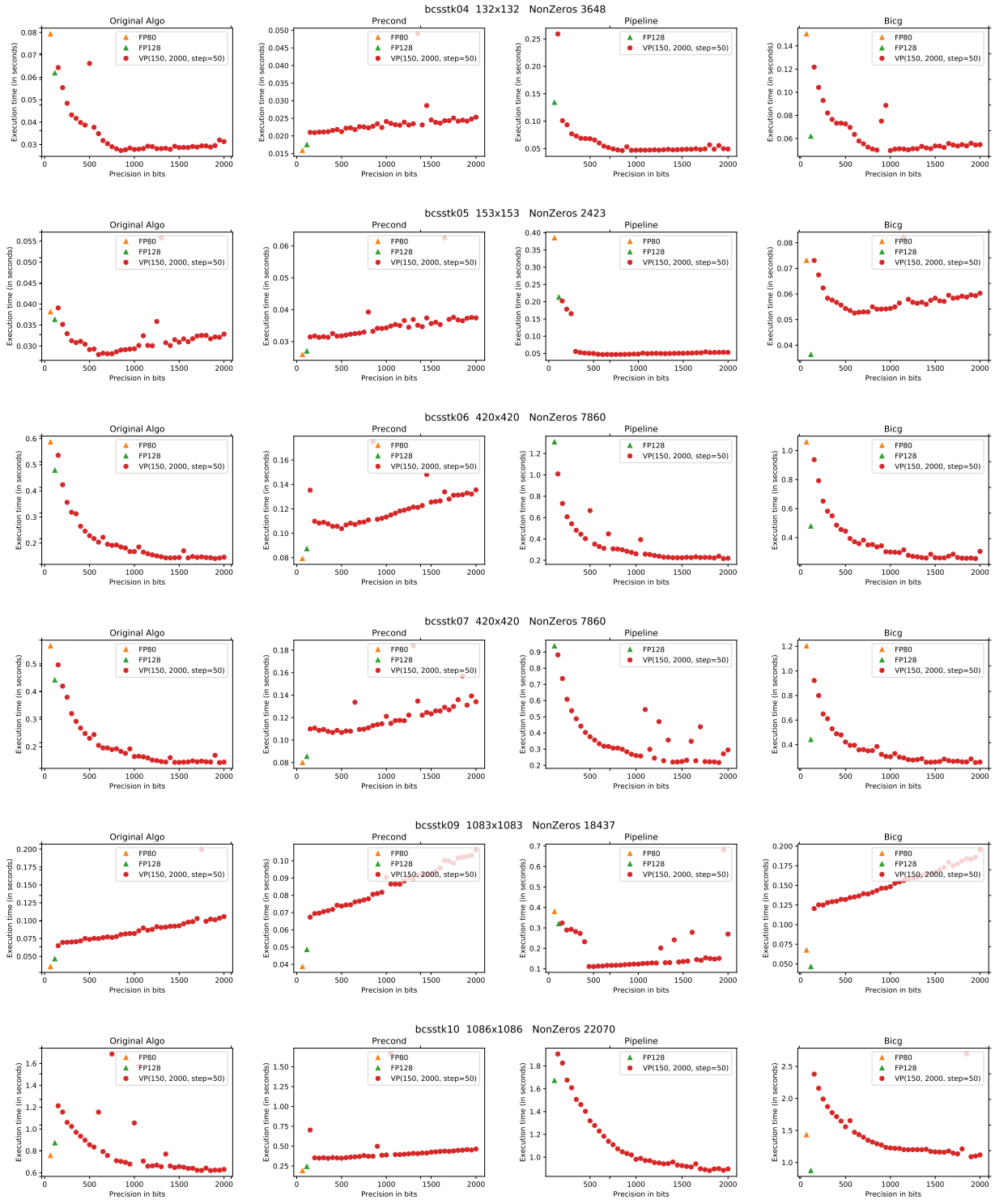


Figure B.1: Execution time for CG variants with multiple formats with matrices *bcsstk04, bcsstk05, bcsstk06, bcsstk07, bcsstk09, and bcsstk10* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

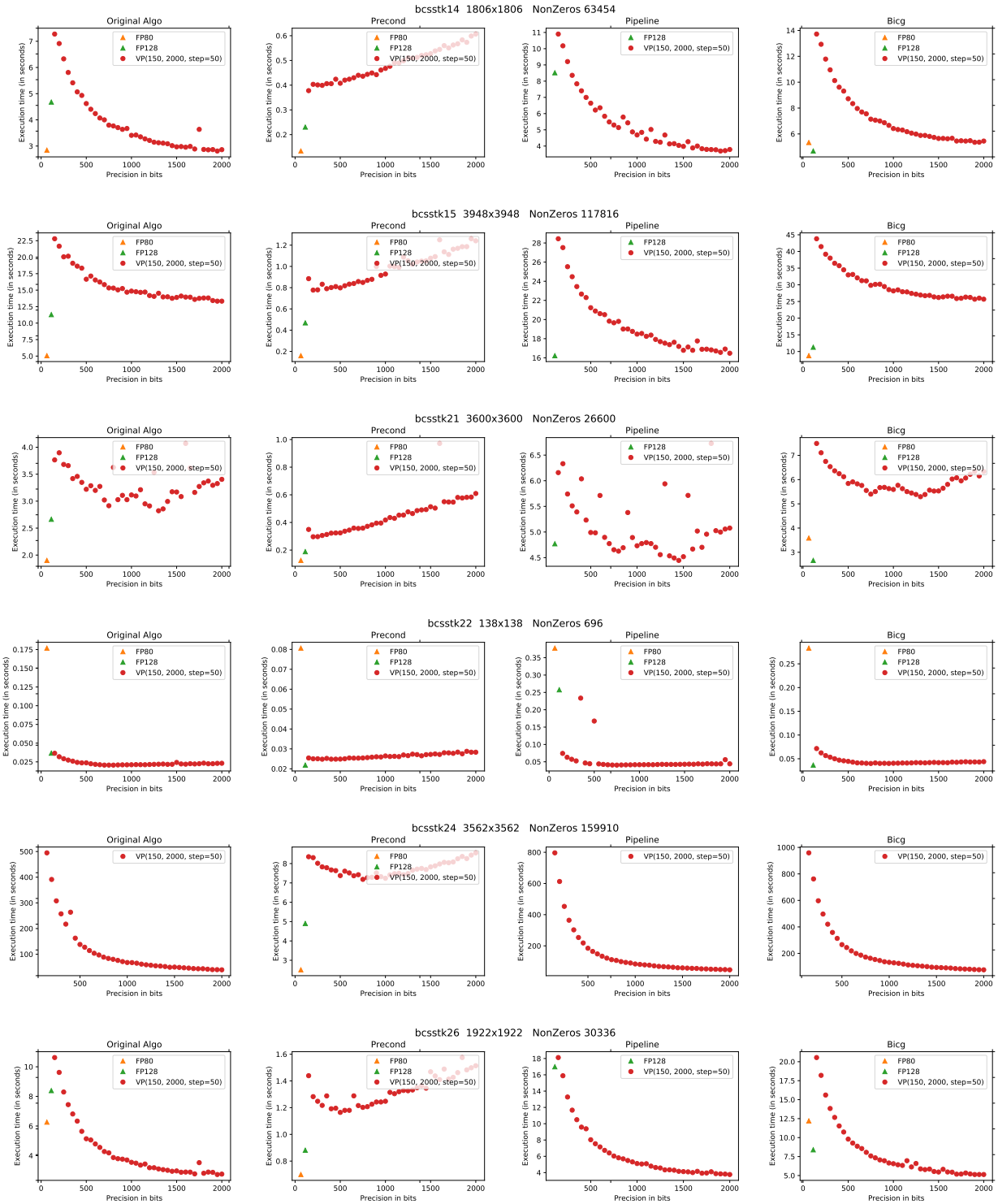


Figure B.2: Execution time for CG variants with multiple formats with matrices *bcsstk14, bcsstk15, bcsstk21, bcsstk22, bcsstk24, and bcsstk26* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

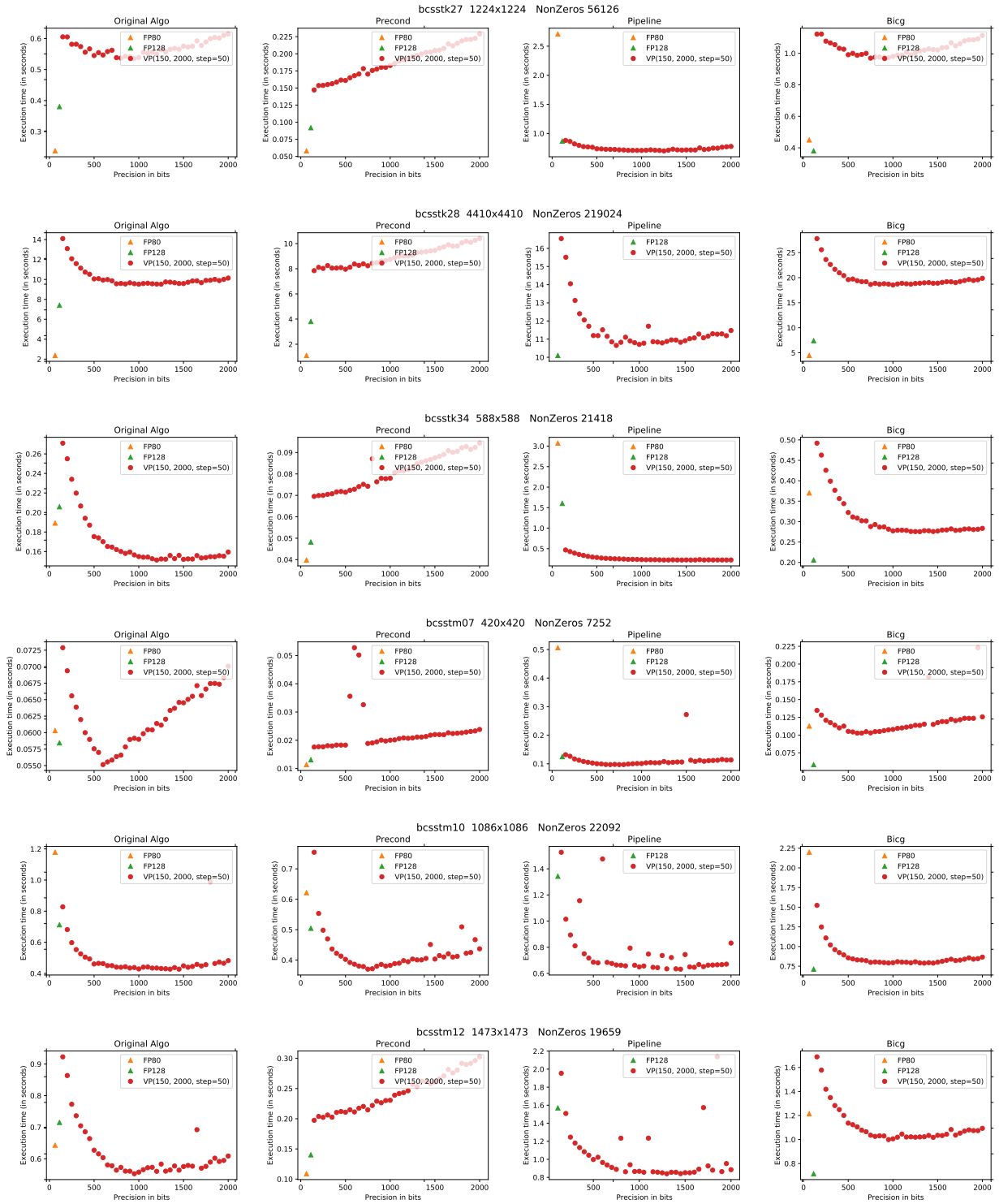


Figure B.3: Execution time for CG variants with multiple formats with matrices *bcsstk27, bcsstk28, bcsstk34, bcsstm07, bcsstm10, and bcsstm12* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

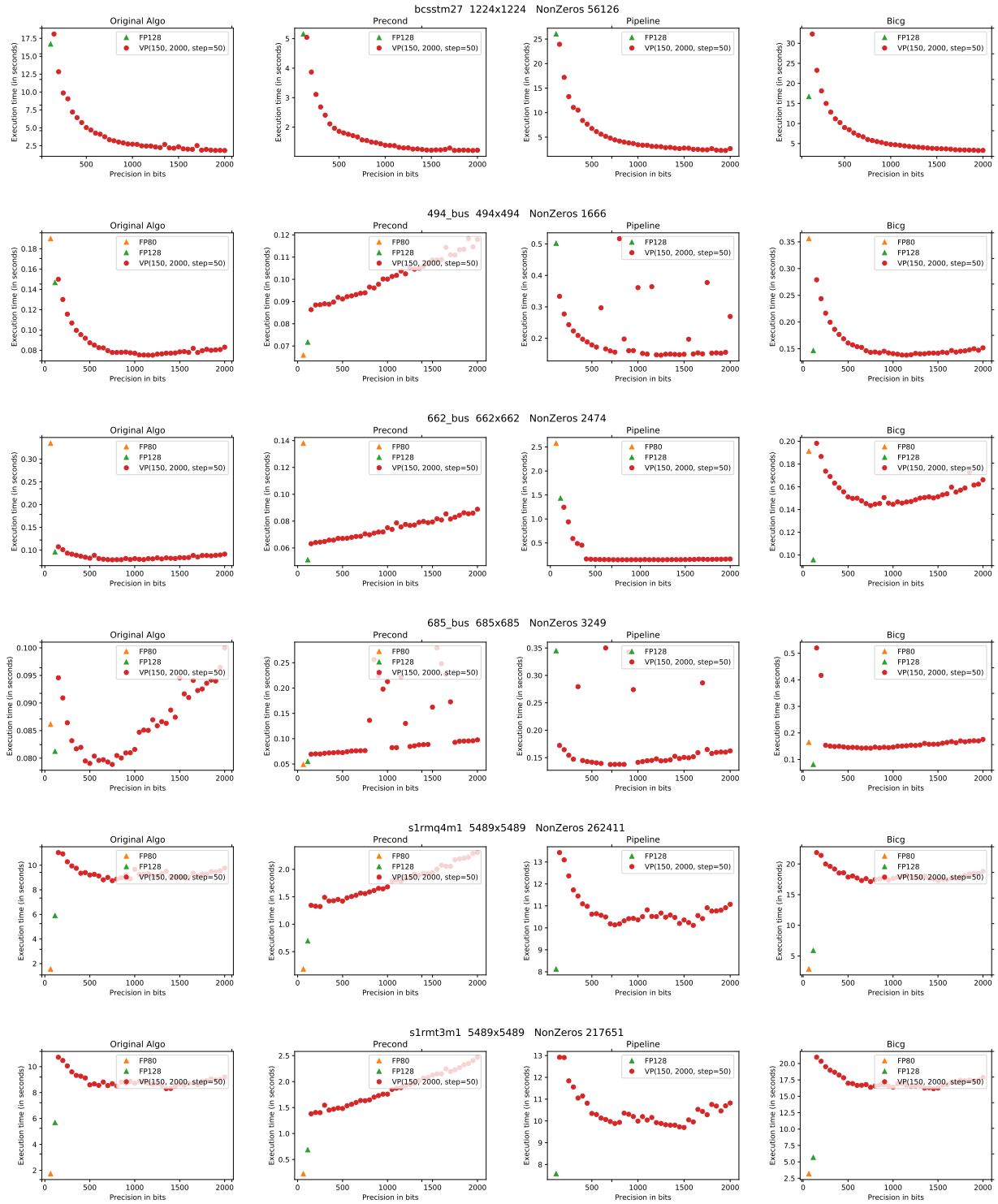


Figure B.4: Execution time for CG variants with multiple formats with matrices *bcsstm27, 494_bus, 662_bus, 685_bus, s1rmq4m1, and s1rmt3m1* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

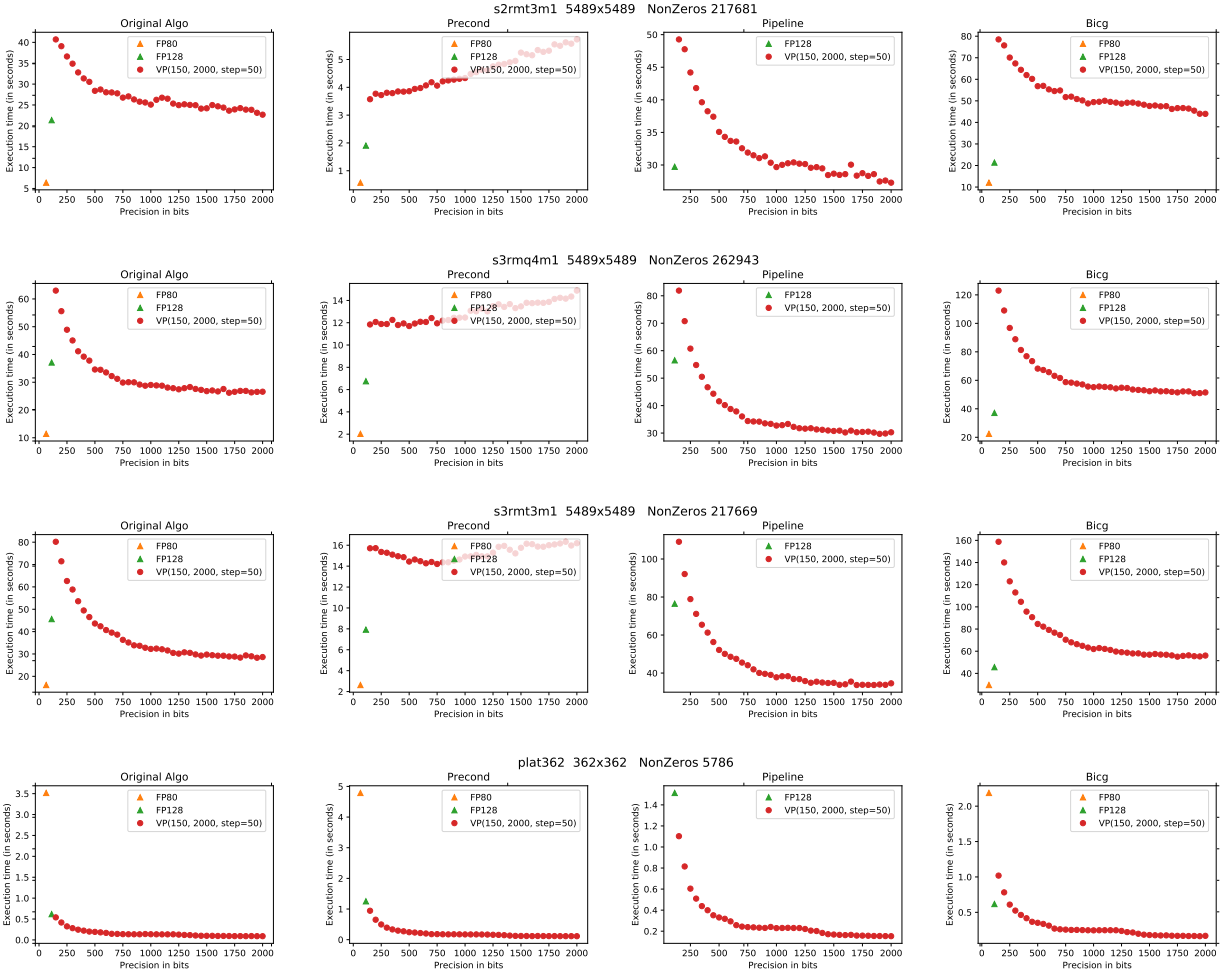


Figure B.5: Execution time for CG variants with multiple formats with matrices *s2rmt3m1, s3rmq4m1, s3rmt3m1, and plat362* from matrix market [24, 86]. Graph lines represent different matrices, and graph columns correspond to variants (from left to right: original CG, precond CG, pipelined CG, BiCG). A missing type in a graph implies the algorithm did not converge. Results for `double` are not displayed.

Publications

Publications in International Conferences and Workshops

- **Tiago Trevisan Jost**, Yves Durand, Christian Fabre, Albert Cohen, Frédéric Pétrot. Seamless Compiler Integration of Variable Precision Floating-Point Arithmetic *International Symposium on Code Generation and Optimization (CGO 2021)*
- **Tiago Jost**, Yves Durand, Christian Fabre, Albert Cohen, Frédéric Pétrot. VP Float: First Class Treatment for Variable Precision Floating Point Arithmetic. *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT 2020)* (pp. 355–356)
- Andrea Bocco, **Tiago T. Jost**, Albert Cohen, Florent de Dinechin, Yves Durand, Christian Fabre. Byte-Aware Floating-point Operations through a UNUM Computing Unit. *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2019)* (pp. 323-328)
- **Tiago T. Jost**, Andrea Bocco, Yves Durand, Christian Fabre, Florent de Dinechin, and Albert Cohen. Variable Precision Floating-Point RISC-V Coprocessor Evaluation using Lightweight Software and Compiler Support. *Third Workshop on Computer Architecture Research with RISC-V (CARRV'19).*

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv:1603.04467 [cs]* (Mar. 2016). URL: http://arxiv.org/abs/1603.04467 (visited on 11/24/2020) (pages 1, 15).

[2] A. Abdelfattah, S. Tomov, and J. Dongarra. "Towards Half-Precision Computation for Complex Matrices: A Case Study for Mixed Precision Solvers on GPUs". In: *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Nov. 2019, pp. 17–24. DOI: 10.1109/ScalA49573.2019.00008 (page 20).

[3] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang. "PETSc/TS: A Modern Scalable ODE/DAE Solver Library". In: *arXiv preprint arXiv:1806.01437* (2018) (page 105).

[4] A. V. Aho, R. Sethi, and J. D. Ullman. "Compilers, principles, techniques". In: *Addison wesley* 7.8 (1986), p. 9 (page 63).

[5] A. Akkaş, M. J. Schulte, and J. E. Stine. "Intrinsic compiler support for interval arithmetic". In: *Numerical Algorithms* 37.1 (2004), pp. 13–20 (page 15).

[6] A. Anderson and D. Gregg. "Vectorization of multibyte floating point data formats". In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Sept. 2016, pp. 363–372. DOI: 10.1145/2967938.2967966 (page 24).

[7] 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)*. Apr. 2011, pp. 85–96. DOI: 10.1109/CGO.2011.5764677 (page 25).

[8] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. "PetaBricks: a language and compiler for algorithmic choice". In: *ACM SIGPLAN Notices* 44.6 (June 2009), pp. 38–49. ISSN: 0362-1340. DOI: 10.1145/1543135.1542481. URL: https://doi.org/10.1145/1543135.1542481 (visited on 02/12/2021) (page 25).

[9] M. G. Arnold. "The residue logarithmic number system: theory and implementation". In: *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. IEEE, 2005, pp. 196–205 (pages 19, 24).

[10] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. "A comparative study of static and profile-based heuristics for inlining". In: *ACM SIGPLAN Notices* 35.7 (Jan. 2000), pp. 52–64. ISSN: 0362-1340. DOI: 10.1145/351403.351416. URL: https://doi.org/10.1145/351403.351416 (visited on 03/05/2021) (page 64).

[11] 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*. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (pages 28, 88).

[12] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. "Accelerating scientific computations with mixed precision algorithms". In: *Computer Physics Communications* 180.12 (Dec. 2009), pp. 2526–2533. DOI: 10.1016/j.cpc.2008.11.005. URL: https://www.research.manchester.ac.uk/portal/en/publications/accelerating-scientific-computations-with-mixed-precision-algorithms(5b73720d-719c-40dd-8dad-d6262e5921cd)/export.html (visited on 11/27/2020) (pages 10, 14, 20, 33).

[13] D. H. Bailey, R. Barrio, and J. M. Borwein. "High-precision computation: Mathematical physics and dynamics". In: *Applied Mathematics and Computation* 218.20 (June 2012), pp. 10106–10121. ISSN: 0096-3003. DOI: 10.1016/j.amc.2012.03.087. URL: http://www.sciencedirect.com/science/article/pii/S0096300312003505 (visited on 11/24/2020) (page 8).

[14] D. Bailey and J. Borwein. "High-Precision Arithmetic in Mathematical Physics". In: *Mathematics* 3 (May 2015), pp. 337–367. DOI: 10.3390/math3020337 (page 1).

[15] D. H. Bailey. "MPFUN: A portable high performance multiprecision package". In: *NASA Ames Research Center* (1990) (pages 14, 21).

[16] D. H. Bailey. "Reproducibility and variable precision computing". In: *The International Journal of High Performance Computing Applications* 34.5 (Sept. 2020), pp. 483–490. ISSN: 1094-3420. DOI: 10.1177/1094342020938424. URL: https://doi.org/10.1177/1094342020938424 (visited on 12/13/2020) (pages 12, 14).

[17] D. H. Bailey, X. S. Li, and Y. Hida. *QD: A Double-Double/Quad-Double Package*. Lawrence Berkeley National Laboratory, 2003 (page 26).

[18] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. *PETSc Web page*. 2021. URL: https://www.mcs.anl.gov/petsc (page 105).

[19] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. "Julia: A Fast Dynamic Language for Technical Computing". In: *arXiv:1209.5145 [cs]* (Sept. 2012). URL: http://arxiv.org/abs/1209.5145 (visited on 11/27/2020) (pages 22, 32, 90).

[20] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al. *ScaLAPACK users' guide*. SIAM, 1997 (page 27).

[21] A. Bocco, Y. Durand, and F. d. Dinechin. "Dynamic Precision Numerics Using a Variable-Precision UNUM Type I HW Coprocessor". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. June 2019, pp. 104–107. DOI: 10.1109/ARITH.2019.00028 (pages 9, 37, 68).

[22] A. Bocco, T. T. Jost, A. Cohen, F. d. Dinechin, Y. Durand, and C. Fabre. “Byte-Aware Floating-point Operations through a UNUM Computing Unit”. In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. Oct. 2019, pp. 323–328. DOI: 10.1109/VLSI-SoC.2019.8920387 (pages 9, 10).

[23] A. Bocco, Y. Durand, and F. d. Dinechin. “SMURF: Scalar Multiple-precision Unum Risc-V Floating-point Accelerator for Scientific Computing”. In: ACM, Mar. 2019, pp. 1–8. DOI: 10.1145/3316279.3316280. URL: https://hal.inria.fr/hal-02087098 (visited on 11/27/2020) (pages 10, 28, 37, 68, 74, 88).

[24] 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*. GBR: Chapman & Hall, Ltd., Jan. 1997, pp. 125–137. ISBN: 978-0-412-80530-1. (Visited on 11/27/2020) (pages 13, 14, 94, 97, 98, 100, 101, 110–114, 116–120).

[25] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. “A Practical Automatic Polyhedral Program Optimization System”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2008 (pages 52, 66).

[26] R. P. Brent. “An idealist’s view of semantics for integer and real types”. In: *Australian Computer Science Communications* 4 (1982), pp. 130–140 (page 21).

[27] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. “A class of parallel tiled linear algebra algorithms for multicore architectures”. In: *Parallel Computing* 35.1 (2009), pp. 38–53. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2008.10.002. URL: http://www.sciencedirect.com/science/article/pii/S0167819108001117 (page 27).

[28] 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: *arXiv:1812.01762 [cs]* (Jan. 2019). URL: http://arxiv.org/abs/1812.01762 (visited on 11/27/2020) (pages 10, 105).

[29] E. Carson and N. J. Higham. “Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions”. In: *SIAM Journal on Scientific Computing* 40.2 (Jan. 2018), A817–A847. ISSN: 1064-8275. DOI: 10.1137/17M1140819. URL: https://epubs.siam.org/doi/abs/10.1137/17M1140819 (visited on 11/27/2020) (pages 20, 31).

[30] E. Carson and Z. Strakoš. “On the cost of iterative computations”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166 (Mar. 2020), p. 20190050. ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta.2019.0050. URL: https://royalsocietypublishing.org/doi/10.1098/rsta.2019.0050 (visited on 11/25/2020) (page 96).

[31] E. C. Carson, M. Rozložník, Z. Strakoš, P. Tichý, and M. Tůma. “The Numerical Stability Analysis of Pipelined Conjugate Gradient Methods: Historical Context and Methodology”. In: *SIAM Journal on Scientific Computing* 40.5 (2018), A3549–A3580. DOI: 10.1137/16M1103361. URL: https://doi.org/10.1137/16M1103361 (pages 93, 95).

[32] S. W. D. Chien, I. B. Peng, and S. Markidis. “Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications”. In: *arXiv:1907.05917 [cs]* 12043 (2020), pp. 301–310. DOI: 10.1007/978-3-030-43229-4_26. URL: http://arxiv.org/abs/1907.05917 (visited on 11/27/2020) (page 24).

[33] S. Chowdhary, J. P. Lim, and S. Nagarakatte. “Debugging and detecting numerical errors in computation with posits”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 731–746. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386004. URL: https://doi.org/10.1145/3385412.3386004 (visited on 12/10/2020) (page 25).

[34] S. Cools, J. Cornelis, and W. Vanroose. “Numerically Stable Recurrence Relations for the Communication Hiding Pipelined Conjugate Gradient Method”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.11 (2019), pp. 2507–2522. DOI: 10.1109/TPDS.2019.2917663 (pages 95, 105).

[35] S. Cools, J. Cornelis, P. Ghysels, and W. Vanroose. “Improving strong scaling of the Conjugate Gradient method for solving large linear systems using global reduction pipelining”. In: *CoRR* abs/1905.06850 (2019). URL: http://arxiv.org/abs/1905.06850 (pages 95, 105).

[36] S. Cools, W. Vanroose, E. F. Yetkin, E. Agullo, and L. Giraud. “On rounding error resilience, maximal attainable accuracy and parallel performance of the pipelined Conjugate Gradients method for large-scale linear systems in PETSc”. In: *Proceedings of the Exascale Applications and Software Conference 2016*. EASC ’16. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 1–10. ISBN: 978-1-4503-4122-6. DOI: 10.1145/2938615.2938621. URL: https://doi.org/10.1145/2938615.2938621 (visited on 04/03/2021) (pages 95, 105).

[37] J. Cornelis, S. Cools, and W. Vanroose. “The Communication-Hiding Conjugate Gradient Method with Deep Pipelines”. In: *CoRR* abs/1801.04728 (2018). URL: http://arxiv.org/abs/1801.04728 (pages 94, 95).

[38] C. Courbet. “NSan: a floating-point numerical sanitizer”. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. New York, NY, USA: Association for Computing Machinery, Mar. 2021, pp. 83–93. ISBN: 978-1-4503-8325-7. DOI: 10.1145/3446804.3446848. URL: https://doi.org/10.1145/3446804.3446848 (visited on 04/01/2021) (page 25).

[39] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1558-190X. DOI: 10.1109/99.660313 (page 66).

[40] E. Darulova and V. Kuncak. “On Sound Compilation of Reals”. In: *arXiv:1309.2511 [cs]* (Sept. 2013). URL: http://arxiv.org/abs/1309.2511 (visited on 11/25/2020) (page 25).

[41] L. Deniau and A. Naumann. *Proposal for classes with runtime size*. Document number: N4188. 2014. URL: http://www.open-std.org/JTC1/SC22/wg21/docs/papers/2014/n4188.pdf (page 48).

[42] 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*. CoNGA’19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–10. ISBN: 978-1-4503-7139-1. DOI: 10.1145/3316279.3316285. URL: https://doi.org/10.1145/3316279.3316285 (visited on 11/27/2020) (pages 10, 38, 105).

[43] J. Dongarra. *Freely Available Software for Linear Algebra (September 2018)*. 2018. URL: http://www.netlib.org/utk/people/JackDongarra/la-sw.html (page 13).

[44] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. “Block reduction of matrices to condensed forms for eigenvalue computations”. In: *Journal of Computational and Applied Mathematics* 27.1 (1989), pp. 215–227. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(89)90367-1. URL: http://www.sciencedirect.com/science/article/pii/0377042789903671 (page 27).

[45] I. o. Electrical, E. E. C. S. S. Committee, and D. Stevenson. *IEEE standard for binary floating-point arithmetic*. IEEE, 1985 (pages 1, 7).

[46] J. S. Ely. “The VPI software package for variable precision interval arithmetic”. In: *Interval Computations* 2.2 (1993), pp. 135–153 (page 28).

[47] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376 (page 88).

[48] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. “On the design of CGAL a computational geometry algorithms library”. In: *Software: Practice and Experience* 30.11 (2000), pp. 1167–1202. ISSN: 1097-024X. DOI: https://doi.org/10.1002/1097-024X(200009)30:11<1167::AID-SPE337>3.0.CO;2-B. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1097-024X%28200009%2930%3A11%3C1167%3A%3AAID-SPE337%3E3.0.CO%3B2-B (visited on 12/04/2020) (page 22).

[49] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. “MPFR: A multiple-precision binary floating-point library with correct rounding”. In: *ACM Transactions on Mathematical Software* 33.2 (June 2007), 13–es. ISSN: 0098-3500. DOI: 10.1145/1236463.1236468. URL: https://doi.org/10.1145/1236463.1236468 (visited on 11/27/2020) (pages xvii, 7, 10, 14, 21, 26, 33, 35, 68).

[50] A. M. Frolov and D. H. Bailey. “Highly accurate evaluation of the few-body auxiliary functions and four-body integrals”. In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 36.9 (Apr. 2003), pp. 1857–1867. ISSN: 0953-4075. DOI: 10.1088/0953-4075/36/9/315. URL: https://doi.org/10.1088%2F0953-4075%2F36%2F9%2F315 (visited on 11/24/2020) (page 1).

[51] P. Ghysels and W. Vanroose. “Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm”. In: *Parallel Computing*. 7th Workshop on Parallel Matrix Algorithms and Applications 40.7 (July 2014), pp. 224–238. ISSN: 0167-8191. DOI: 10.1016/j.parco.2013.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0167819113000719 (visited on 03/30/2021) (page 92).

[52] 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: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351546 (pages 9, 10, 29).

[53] D. Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163 (visited on 11/27/2020) (page 6).

[54] T. Granlund. *GNU Multiple Precision Arithmetic Library 6.1.2*. Dec. 2016 (pages 7, 14, 21, 26, 33, 68).

[55] 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 (pages 66, 80).

[56] G. Guennebaud, B. Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (page 105).

[57] J. L. Gustafson. *The End of Error: Unum Computing*. CRC Press, 2017 (pages 9, 36, 68, 75).

[58] Gustafson and Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: *Supercomputing Frontiers and Innovations: an International Journal* 4.2 (June 2017), pp. 71–86. ISSN: 2409-6008. DOI: 10.14529/jsfi170206. URL: https://doi.org/10.14529/jsfi170206 (visited on 11/27/2020) (pages 9, 24, 27, 38).

[59] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2018, pp. 603–613. DOI: 10.1109/SC.2018.00050 (page 20).

[60] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham. "Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476.2243 (2020), p. 20200110 (page 20).

[61] A. Haidar, P. Wu, S. Tomov, and J. Dongarra. "Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers". In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA '17. New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 978-1-4503-5125-6. DOI: 10.1145/3148226.3148237. URL: https://doi.org/10.1145/3148226.3148237 (page 20).

[62] J. Hauser. *SoftFloat*. 1997. URL: http://www.jhauser.us/arithmetic/SoftFloat.html (page 24).

[63] M. R. Hestenes, E. Stiefel, et al. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436 (pages 90, 91, 93).

[64] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002 (pages 1, 11, 12, 31).

[65] J. Hou, Y. Zhu, S. Du, and S. Song. "Enhancing Accuracy and Dynamic Range of Scientific Data Analytics by Implementing Posit Arithmetic on FPGA". In: *Journal of Signal Processing Systems* 91.10 (Oct. 2019), pp. 1137–1148. ISSN: 1939-8115. DOI: 10.1007/s11265-018-1420-5. URL: https://doi.org/10.1007/s11265-018-1420-5 (visited on 11/27/2020) (page 10).

[66] IBM. *IBM System/360 Principles of Operation*. IBM Press, 1964 (page 7).

[67] IEEE. "IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008)". In: Institute of Electrical and Electronics Engineers New York, 2019 (page 7).

[68] K. Isupov. "Using Floating-Point Intervals for Non-Modular Computations in Residue Number System". In: *IEEE Access* 8 (2020), pp. 58603–58619. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2982365 (page 19).

[69] M. K. Jaiswal and H. K. So. "PACoGen: A Hardware Posit Arithmetic Core Generator". In: *IEEE Access* 7 (2019), pp. 74586–74601. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2920936 (pages 10, 29).

[70] J. Johnson. “Rethinking floating point for deep learning”. In: *arXiv:1811.01721 [cs]* (Nov. 2018). URL: http://arxiv.org/abs/1811.01721 (visited on 11/27/2020) (pages 10, 24, 105).

[71] M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker. “CAMPARY: Cuda multiple precision arithmetic library and applications”. In: *International Congress on Mathematical Software*. Springer, 2016, pp. 232–240 (page 21).

[72] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. “A Study of BFLOAT16 for Deep Learning Training”. In: *arXiv:1905.12322 [cs, stat]* (June 2019). URL: http://arxiv.org/abs/1905.12322 (visited on 11/24/2020) (page 1).

[73] J. G. Kemeny, T. E. Kurtz, and D. S. Cochran. *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*. Dartmouth Publications, 1968 (page 7).

[74] N. G. Kingsbury and P. J. Rayner. “Digital filtering using logarithmic arithmetic”. In: *Electronics Letters* 7.2 (1971), pp. 56–58 (page 24).

[75] U. Kulisch. *Computer arithmetic and validity: theory, implementation, and applications*. Vol. 33. Walter de Gruyter, 2013 (pages 24, 27, 50).

[76] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 (pages 8, 15, 59).

[77] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Transactions on Mathematical Software* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847 (visited on 11/27/2020) (pages 26, 36).

[78] R. Leavitt. “Adjustable precision floating point arithmetic in Ada”. In: *ACM SIGAda Ada Letters* VII.5 (Sept. 1987), pp. 63–78. ISSN: 1094-3641. DOI: 10.1145/36077.36082. URL: https://doi.org/10.1145/36077.36082 (visited on 11/27/2020) (page 21).

[79] J. Lee, G. D. Peterson, D. S. Nikolopoulos, and H. Vandierendonck. “AIR: Iterative refinement acceleration using arbitrary dynamic precision”. In: *Parallel Computing* 97 (Sept. 2020), p. 102663. ISSN: 0167-8191. DOI: 10.1016/j.parco.2020.102663. URL: http://www.sciencedirect.com/science/article/pii/S0167819120300569 (visited on 01/06/2021) (page 20).

[80] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. “Design, implementation and testing of extended and mixed precision BLAS”. In: *ACM Transactions on Mathematical Software* 28.2 (June 2002), pp. 152–205. ISSN: 0098-3500. DOI: 10.1145/567806.567808. URL: https://doi.org/10.1145/567806.567808 (visited on 12/09/2020) (pages 26, 49).

[81] P. Lindstrom. “Universal Coding of the Reals using Bisection”. In: *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA’19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–10. ISBN: 978-1-4503-7139-1. DOI: 10.1145/3316279.3316286. URL: https://doi.org/10.1145/3316279.3316286 (visited on 12/15/2020) (page 24).

[82] 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*. CoNGA ’18. New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 1–14. ISBN: 978-1-4503-6414-0. DOI: 10.1145/3190339.3190344. URL: https://doi.org/10.1145/3190339.3190344 (visited on 11/27/2020) (pages 14, 24).

[83] G. S. Lloyd. *unum*. URL: https://github.com/LLNL/unum (pages 14, 24).

[84] C. Long. *Softposit*. 2019. URL: https://gitlab.com/cerlane/SoftPosit/-/wikis/home (pages 14, 24).

[85] J. Maddock and C. Kormanyos. *Boost.Multiprecision*. 2020 (pages 22, 72, 79).

[86] *Matrix Market repository*. 2007. URL: https://math.nist.gov/MatrixMarket/ (pages 13, 14, 94, 97, 98, 100, 101, 110–114, 116–120).

[87] V. Ménissier-Morain. “Arbitrary precision real arithmetic: design and algorithms”. In: *The Journal of Logic and Algebraic Programming*. Practical development of exact real number computation 64.1 (July 2005), pp. 13–39. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2004.07.003. URL: http://www.sciencedirect.com/science/article/pii/S1567832604000748 (visited on 12/22/2020) (pages 10, 19).

[88] S. Mittal. “A Survey of Techniques for Approximate Computing”. In: *ACM Computing Surveys* 48.4 (Mar. 2016), 62:1–62:33. ISSN: 0360-0300. DOI: 10.1145/2893356. URL: https://doi.org/10.1145/2893356 (visited on 11/24/2020) (page 2).

[89] G. E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860 (page 1).

[90] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to interval analysis*. SIAM, 2009 (page 19).

[91] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston, 2018 (page 12).

[92] M. Nakata. *The MPACK (MBLAS/MLAPACK) a multiple precision arithmetic version of BLAS and LAPACK*. http://mplapack.sourceforge.net/, 2010 (page 26).

[93] E. T. L. Omtzigt, P. Gottschling, M. Seligman, and W. Zorn. *Universal Number Library*. URL: https://github.com/stillwater-sc/universal (page 24).

[94] E. T. L. Omtzigt, P. Gottschling, M. Seligman, and W. Zorn. “Universal Numbers Library: design and implementation of a high-performance reproducible number systems library”. In: *arXiv e-prints* (2020), arXiv–2012 (page 24).

[95] J. F. Palmer. “The Intel® 8087 numeric data processor”. In: *Proceedings of the May 19-22, 1980, national computer conference*. 1980, pp. 887–893 (page 7).

[96] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. “Automatically improving accuracy for floating point expressions”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. New York, NY, USA: Association for Computing Machinery, June 2015, pp. 1–11. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737959. URL: https://doi.org/10.1145/2737924.2737959 (visited on 02/12/2021) (page 25).

[97] F. M. Q. Pereira and J. Palsberg. “SSA Elimination after Register Allocation”. In: *Compiler Construction*. Ed. by O. de Moor and M. I. Schwartzbach. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 158–173. ISBN: 978-3-642-00722-4. DOI: 10.1007/978-3-642-00722-4_12 (page 76).

[98] L.-N. Pouchet et al. *PolyBench: The polyhedral benchmark suite*. 2012 (pages 6, 80).

[99] *RAJAPerf*. URL: https://github.com/LLNL/RAJAPerf (page 85).

[100] N. Revol. “Introduction to the IEEE 1788-2015 Standard for Interval Arithmetic”. In: *10th International Workshop on Numerical Software Verification - NSV 2017, workshop of CAV 2017*. Ed. by A. Abate and S. Boldo. LNCS. Heidelberg, Germany: Springer, July 2017, pp. 14–21. DOI: 10.1007/978-3-319-63501-9. URL: https://hal.inria.fr/hal-01559955 (page 19).

[101] N. Revol. “The MPFI Library: Towards IEEE 1788–2015 Compliance”. In: *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2019, pp. 353–363 (page 22).

[102] C. Rubio-González, Cuong Nguyen, Hong Diep 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*. Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2503296 (page 25).

[103] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijen, D. H. Bailey, and D. Hough. “Floating-Point Precision Tuning Using Blame Analysis”. In: *38th International Conference on Software Engineering*. Ed. by W. Visser and L. Williams. Austin, Texas: IEEE TCSE and ACM SIGSOFT, May 2016 (page 25).

[104] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898718003 (visited on 12/01/2020) (page 93).

[105] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock. “Finding root causes of floating point error”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 256–269. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192411. URL: https://doi.org/10.1145/3192366.3192411 (visited on 02/12/2021) (page 25).

[106] C. Sanderson and R. Curtin. “Armadillo: a template-based C++ library for linear algebra”. In: *Journal of Open Source Software* 1.2 (2016), p. 26. DOI: 10.21105/joss.00026. URL: https://doi.org/10.21105/joss.00026 (page 105).

[107] M. J. Schulte and E. E. Swartzlander. “A family of variable-precision interval arithmetic processors”. In: *IEEE Transactions on Computers* 49.5 (May 2000), pp. 387–397. ISSN: 1557-9956. DOI: 10.1109/12.859535 (pages 27, 28).

[108] S. Shrimpton. *An Implementation of MIL-STD-1750 Airborne Computer Instruction Set Architecture*. ROYAL AIRCRAFT ESTABLISHMENT FARNBOROUGH (ENGLAND), 1981 (page 7).

[109] A. F. d. Silva, B. N. B. de Lima, and F. M. Q. Pereira. “Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools”. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 47–58. ISBN: 978-1-4503-8325-7. DOI: 10.1145/3446804.3446849. URL: https://doi.org/10.1145/3446804.3446849 (page 81).

[110] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker. *MPI–the Complete Reference: the MPI core*. Vol. 1. MIT press, 1998 (page 92).

[111] J. Snyder and R. Smith. *Exploring classes of runtime size*. Document number: N4025. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4025.pdf (page 48).

[112] I. C. Soc. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008)*. IEEE New York, 2008 (page 7).

[113] R. Stallman et al. *The GNU project*. 1998 (pages 8, 15).

[114] G. Tagliavini, A. Marongiu, and L. Benini. “FlexFloat: A Software Library for Transprecision Computing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.1 (Jan. 2020), pp. 145–156. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2883902 (page 24).

[115] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti. “PERI: A Posit Enabled RISC-V Core”. In: *arXiv:1908.01466 [cs]* (Aug. 2019). URL: http://arxiv.org/abs/1908.01466 (visited on 11/27/2020) (pages 10, 29).

[116] S. Tomov, J. Dongarra, and M. Baboulin. “Towards dense linear algebra for hybrid GPU accelerated manycore systems”. In: *Parallel Matrix Algorithms and Applications* 36.5 (June 2010), pp. 232–240. ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005 (page 27).

[117] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995 (pages 22, 32, 90).

[118] F. G. Van Zee and R. A. Van De Geijn. “BLIS: A framework for rapidly instantiating BLAS functionality”. In: *ACM Transactions on Mathematical Software (TOMS)* 41.3 (2015), pp. 1–33 (page 26).

[119] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. “Polyhedral parallel code generation for CUDA”. In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400713. URL: https://doi.org/10.1145/2400682.2400713 (visited on 03/19/2021) (page 66).

[120] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. “Intel math kernel library”. In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188 (page 26).

[121] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. “The RISC-V instruction set manual, volume i: User-level ISA”. In: *CS Division, EECE Department, University of California, Berkeley* (2014) (pages 28, 74).

[122] R. C. Whaley and J. Dongarra. “Automatically Tuned Linear Algebra Software”. In: *SuperComputing 1998: High Performance Networking and Computing*. 1998 (page 26).

[123] W. A. Wulf and S. A. McKee. “Hitting the memory wall: Implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24 (page 16).

[124] Z. Xianyi, W. Qian, and Z. Chothia. “OpenBLAS”. In: *URL: http://xianyi. github. io/OpenBLAS* (2012), p. 88 (pages 26, 94).

[125] Q. Xu, T. Mytkowicz, and N. S. Kim. “Approximate Computing: A Survey”. In: *IEEE Design Test* 33.1 (Feb. 2016), pp. 8–22. ISSN: 2168-2364. DOI: 10.1109/MDAT.2015.2505723 (pages 2, 25).

[126] C. Yap and T. Dubé. “The exact computation paradigm”. In: *Computing in Euclidean Geometry*. Vol. Volume 4. Lecture Notes Series on Computing Volume 4. WORLD SCIENTIFIC, Jan. 1995, pp. 452–492. ISBN: 978-981-02-1876-8. DOI: 10.1142/9789812831699_0011. URL: https://www.worldscientific.com/doi/abs/10.1142/9789812831699_0011 (visited on 12/02/2020) (page 19).

[127] T. J. Ypma. “Historical Development of the Newton-Raphson Method”. In: *SIAM Review* 37.4 (1995), pp. 531–551. ISSN: 0036-1445. URL: https://www.jstor.org/stable/2132904 (visited on 12/01/2020) (pages 11, 20).

[128] A. F. Zanella, A. F. da Silva, and F. M. Quintão. “YACOS: A Complete Infrastructure to the Design and Exploration of Code Optimization Sequences”. In: *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*. SBLP ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 56–63. ISBN: 978-1-4503-8943-3. DOI: 10.1145/3427081.3427089. URL: https://doi.org/10.1145/3427081.3427089 (page 81).