Byte-Aware Floating-point Operations through a UNUM Computing Unit

Andrea Bocco*†, Tiago T. Jost*†, Albert Cohen‡, Florent de Dinechin§, Yves Durand†, and Christian Fabre†
†CEA, LETI, Univ. Grenoble Alpes, Grenoble, France - ‡Google AI, Paris, France - §INSA-Lyon, Lyon, France
Emails: andrea.bocco@cea.fr, tiago.trevisanjost@cea.fr, albertcohen@google.com, Florent.de-Dinechin@insa-lyon.fr, yves.durand@cea.fr, christian.fabre1@cea.fr

***Abstract*—Most floating-point (FP) hardware support the IEEE 754 format, which defines fixed-size data types from 16 to 128 bits. However, a range of applications benefit from different formats, implementing different tradeoffs. This paper proposes a Variable Precision (VP) computing unit offering a finer granularity of high precision FP operations. The chosen memory format is derived from UNUM type I, where the size of a number is stored within the representation itself. The unit implements a fully pipelined architecture, and it supports up to 512 bits of precision for both interval and scalar computing. The user can configure the storage format up to 8-bit granularity, and the internal computing precision at 64-bit granularity. The system is integrated as a RISC-V coprocessor. Dedicated compiler support exposes the unit through a high level programming abstraction, covering all the operating features of UNUM type I. FPGA-based measurements show that the latency and the computation accuracy of this system scale linearly with the memory format length set by the user. Compared with the MPFR software library, the proposed unit achieves speedups between 3.5x and 18x, with comparable accuracy.**

***Index Terms*—Variable precision, Floating-point, UNUM, Scientific computing, Instruction set architecture, Hardware architecture, RISC-V, Coprocessor, Multiple precision, FPGA, ASIC**

I. Introduction

Increasing performance and reducing energy consumption of computational systems has become a major challenge, and a broad range of compute applications involve heavy floating-point (FP) computations. As applications have widely different requirements in terms of arithmetic precision, choosing the appropriate data format among those offered by the IEEE 754 standard [1] can be difficult.

A wide range of applications show an optimal performance for FP representation that cannot be represented through the standard format. For instance, many applications in neural networks and signal processing require fewer than 16 bits of representation, and using the IEEE format can be inefficient. Others are sensitive to the accumulation of rounding, cancellation and absorption computational errors. Such accumulations can quickly lead to completely inaccurate results. This is the case for linear solvers, experimental math, etc. which may only show stability for formats above 128 bits, reducing the benefit of IEEE-754 hardware support.

*Both authors contributed equally to this research.

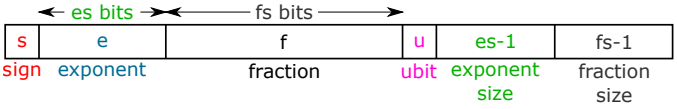


Fig. 1 The Universal NUMber (UNUM) Format.

An alternative research direction consists in rethinking FP arithmetic in order to compensate the aforementioned problems (overkill, cancellation, rounding, etc.). The Posit system [2] relies on tapered precision within a fixed-size format. Another solution is Variable Precision (VP) computing, where the computation precision is adjusted to the application requirements. UNUM [3] is a Variable Precision (VP) format, which offers variable-length exponent and mantissa fields, whose lengths are encoded within the number itself. Previous works [4]–[6] have proposed hardware solutions to support VP FP formats. Software multi-precision libraries also exist, such as Quad Double [7] or the state-of-the-art MPFR [8] and Arb [9].

This paper investigates a hardware Variable Precision (VP) computing unit allowing a finer memory granularity for FP in memory than the IEEE-754 standard. For this purpose, the chosen memory format is UNUM type I.

This work complements previous work [10], [11] with a study of the runtime capabilities needed to adjust the memory format of FP numbers. To the best of the authors' knowledge, this is the first work focusing on the memory subsystem dimension of hardware VP acceleration. The user can configure the storage representation at an 8-bit granularity, and the internal computing precision at 64-bit granularity.

The remainder of this paper is organized as follows: Section II introduces the UNUM type I memory format and our proposed refinements. Section III presents our VP FP architecture and its Instruction Set Architecture (ISA). Section IV specifies the programming scheme of the unit, the new C data type and the compiler toolchain. Section V illustrates a performance and precision profiling comparing the proposed hardware unit with a realization based on the MPFR software library. Section VI concludes this work.

II. UNUM: Refinements to the Memory Format

The computing precision in scientific applications may be decided at run time, for example, depending on the compu-

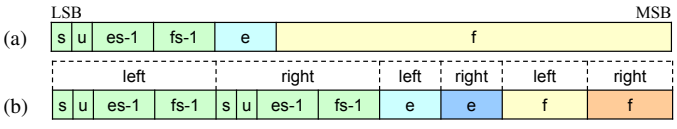


Fig. 2 Adopted memory format for UNUMs and ubounds.

tational error of the algorithm. Thus, a Floating Point (FP) memory format where the user can tune the sizes of the exponent and fraction fields would be appreciable. We adopted the UNUM type I format for storing numbers in memory (Fig. 1 [3]) since, as the authors' knowledge, it is the only Variable Precision (VP) FP format available in the state of the art which has these characteristics. The UNUM format is a self-descriptive FP format with 6 sub-fields: the sign *s*, the exponent *e*, the fraction *f* (like the IEEE 754 formats) and three descriptor fields: *u*, *es-1* and *fs-1*. The *es-1* and *fs-1* encode the lengths of the *e* and *f* variable-length fields respectively. This format is also meant to support Interval Arithmetic (IA) using the "uncertainty" bit *u* and the possibility to represent an interval as a ubound (interval consisting of a tuple of UNUMs). The maximum length of a UNUM is defined by the length of *es-1* and *fs-1*. This pair of lengths *(ess, fss)* is called Unum Environment (UE). For further details on the UNUM format, the read is referred to [3].

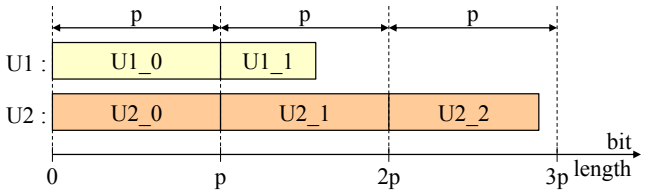
We adopted the modifications of the UNUM format as specified in [11]. Fig. 2 shows the memory format used for the UNUM (Fig. 2a) and ubound (interval made of two UNUMs, Fig. 2b) formats. The fields are re-organized in such a way that the (rightmost) variable length fields are placed in memory after the (leftmost) fixed-length ones. For ubounds the affinity of left or right interval endpoint is indicated above the fields on Fig. 2b. In this way, during load operations, the position and the length of all the UNUM and ubound fields can be decoded from the bytes that are loaded first.

In our system, as depicted by Fig. 3a, each VP FP number is seen as a chain of *chunks* of *p* bits each. The *p* granularity is fixed by the memory subsystem, in our case (RISC-V) $p = 8$ bits. Fig. 3b and 3c depict the two supported addressing modes in memory as described in [11].

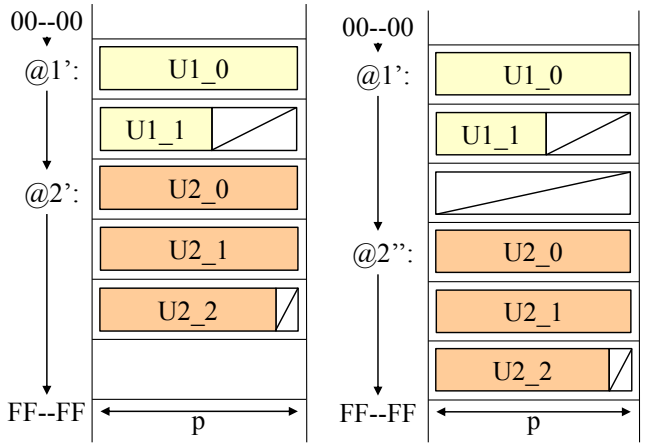
The first one (Fig. 3b) supports *compact arrays* in memory. Memory accesses are done sequentially since each array element (@1') points to the memory address of the next one (@2'). This addressing mode does not support in-place algorithms since the size of each data element may vary. However, it may be used for long term storage of arrays in memory, without loosing precision on array elements.

The second addressing mode, Fig. 3c, aligns the elements of arrays on slots of fixed size which are a multiple of *p* bits. In this way the array elements addresses (@1' and @2") are data-independent and they can be computed at compile time. However, with the UNUM format, both addressing modes may waste memory bits (empty boxes ◪ in Fig. 3).

The slot size is encoded in the Maximum Byte Budget (MBB) coprocessor status register. In this way the memory format length can be tuned by the user, depending on the



(a) Two VP FP numbers organized in p-bit chunks.



(b) Two VP FP numbers stored in memory with the compacted addressing mode.

(c) Two VP FP numbers stored in memory with the slot aligned addressing mode.

Fig. 3 Alternative address modes (3b, 3c) for variable length numbers 3a.

application needs, having a granularity of *p*=8 bits. We support the Bounded Memory Format (BMF) introduced in [11] which remaps the UNUM type I format for all the possible MBB values. During store operations, if the slot size defined in MBB is lower than the maximum UNUM bit-length (deducible from its UE), the data is re-rounded. Special values like Not-a-Number and positive and negative infinities are stored with different but unique encodings.

Unlike [10], in this work we support all the possible UE combinations between (ess=1,fss=1) and (ess=4,fss=9) and all the possible values of MBB (from 1 to 68). The maximum MBB value is defined by the (4,9) UE. In other words, we are able to support up to 512 fractional bits in main memory while giving the user the possibility to select the data memory footprint with a byte-level granularity.

III. The Hardware Platform

The proposed hardware is organized as shown in Fig. 4. The UNUM unit is embedded as a coprocessor in a RISC-V RocketChip environment [12]. The native RocketChip system is generated with 64-bit parallelism. It is made of a main core ①, a Floating Point (FP) unit ②, a Load and Store Unit (LSU) ③, a 64KBytes L1 cache ④ and a RoCC interface able to connect up to 4 coprocessors ⑤. Our coprocessor is dedicated for Variable Precision (VP) FP computations and it supports the UNUM format specified in Section II. The coprocessor has a dedicated scratchpad ⑥ and Load and Store unit ⑦.

The coprocessor scratchpad (or register file) hosts 32 intervals with up to 512 bits of mantissa precision for each interval

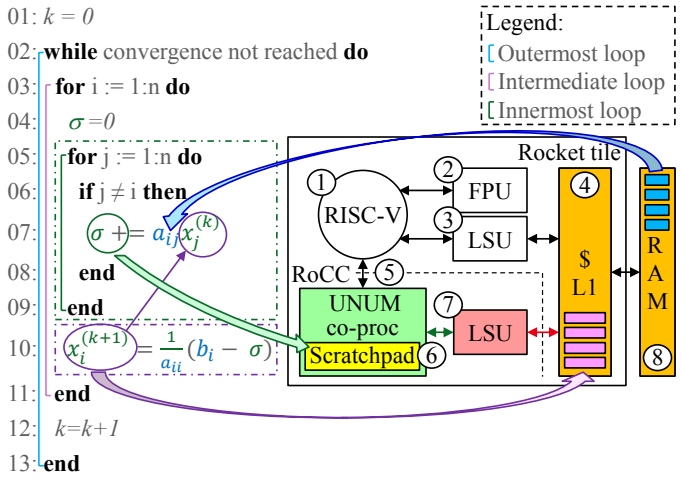


Fig. 4 The coprocessor architecture and its programming model: structure and variable length mapping for an iterative solver kernel.

	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	`srnd`	unused	Xs1	0	1	0	unused	CUST
⑥	`lrnd`	unused	unused	1	0	0	Xd	CUST
⑦	`mov_g2g`	unused	gRs1	0	0	0	gRd	CUST
⑧	`movll/movlr`	unused	gRs1	0	0	0	gRd	CUST
⑨	`movrl/movrr`	unused	gRs1	0	0	0	gRd	CUST
⑩	`mov_x2g`	#imm5	Xs1	0	1	0	gRd	CUST
⑪	`mov_g2x`	#imm5	gRs2	1	0	0	Xd	CUST
⑫	`mov_d2g/mov_f2g`	#imm5	Xs1	0	1	0	gRd	CUST
⑬	`mov_g2d/mov_g2f`	#imm5	gRs2	1	0	0	Xd	CUST
⑭	`gcmp`	gRs2	gRs1	1	0	0	Xd	CUST
⑮	`gadd/gsub/gmul`	gRs2	gRs1	0	0	0	gRd	CUST
⑯	`gguess/gradius`	unused	gRs1	0	0	0	gRd	CUST
⑰	`ldgu/ldub`	unused	Xs1	0	1	0	gRd	CUST
⑱	`stul/stub`	gRs2	Xs1	0	1	0	unused	CUST
⑲	`ldgu_next/ldub_next`	gRs2	Xs1	1	1	0	Xd	CUST
⑳	`stul_next/stub_next`	gRs2	Xs1	1	1	0	Xd	CUST

TABLE I Coprocessor Instruction Set Architecture.

endpoint. Unlike UNUM/ubound, the scratchpad format has an explicit exponent and a normalized mantissa to facilitate hardware computation. This architecture is pipelined and it has an internal parallelism of 64-bits. Thus, the scratchpad computing precision (Working G-layer Precision, WGP) has a granularity of 64 bits. Internal operations with higher precisions multiple of 64 bits are done by iterating on the existing hardware. Conversions between the scratchpad and the UNUM/ubound formats are handled by the coprocessor LSU. Refer to [10] for more details.

This system is best suited for applications which use VP FP kernels which follow a common scheme. It takes advantage of three storage types: 1/ the internal (also called register-level) storage, 2/ the intermediate (also called L1 cache) storage, and 3/ the external (also called main memory) storage.

Fig. 4 illustrates our programming model which is made of three levels implemented as nested loops. The outermost one, which depends on some criteria evaluated on the final result, manages the convergence by increasing or decreasing the computing and/or memory precision. This level relies on the external storage ⑧ to store the input data used by the kernel. This data is processed in the coprocessor internal storage ⑥ passing through the intermediate storage ④.

The innermost level is meant for accumulation of many partial products and it is usually processed in ⑥. Thus, one objective of the compiler is to keep the computation local at this level, in order to minimize the precision losses. Due to the support of the modified UNUM format introduced in Section II, the spilled variables can be written to the intermediate storage in UNUM.

The intermediate level updates a UNUM vector (too big to fit into the internal storage) using the accumulated results of the innermost level. The usage of VP with this three-level model allows to control precision with a lower consumption of cache memory. For more details please refer to [11].

The coprocessor Instruction Set Architecture (ISA, Table I) is organized in four groups. The first group ①-⑥ supports modifications on internal status registers. The second one ⑦-⑬ supports internal move operations and conversion functions between float/double and scratchpad entries. The third group ⑭-⑯ supports coprocessor operations including the `gguess` to compute an interval midpoint and `gradius` to compute an interval width. Division is implemented by software. The fourth group ⑰-⑳ supports load and store operations in the two addressing modes specified in Section II. Unlike [10], this ISA supports multiple programmable rounding modes for internal operations (⑤ and ⑥), including round to interval or round to nearest even.

IV. Programming Model and Compiler Support

New architectures often require new software interfaces and programming models to take advantage of the expanded hardware functionalities. For example, multi-core and many-core systems often involve an OpenMP programming interface [13], while CUDA [15] and OpenCL [16] are widely used for Graphics Processing Units (GPU). Complementary to these parallel programming models, we propose an intuitive programming abstraction of the UNUM computing unit and its capabilities. This section covers the software support for Variable Precision (VP), highlighting the most important aspects of the language and compiler extensions.

A. New data type support

We propose a new `vpfloat` primitive type to allow the use of the VP coprocessor and its operations. According to the semantics of the data type, `vpfloat` variables can be declared as global or local, and their size, in bytes, must be provided in the declaration. The size value defines the MBB and ranges between 1 and 68, which corresponds to the maximum size taken by the (4, 9) Unum Environment (UE). The size chosen for a variable hints the compiler about its UE in memory.

The coprocessor internal precision (WGP) can be chosen 1/ automatically by the compiler according to the slot size

```
1 vpfloat pi(int n) {
2     vpfloat<42> sum = 0.0;
3     int sign = 1;
4     for (int i = 0; i < n; ++i) {
5         sum += sign/(2.0v*i+1.0v);
6         sign *= -1;
7     }
8     return 4.0*sum;
9 }
```

Listing 1: Usage example of the VP unit: the algorithm calculates π iteratively on 42 bytes using the Taylor series $\pi = 4 \cdot \sum_{i=0}^{n} \frac{(-1)^i}{2 \cdot i+1}$

specified in the variable declaration, always respecting the 64-bit granularity; or 2/ manually by the user using the `susr` and `swgp` assembly instructions. For the first case, WGP is likely higher than MBB, since the former must be a multiple of 64 and the latter can be byte-configured.

Listing 1 shows toy C code to calculate π using an iterative approach based on a Taylor series approximation with `vpfloat` type numbers. In line 2, the `sum` variable is declared as a 42-byte VP FP number. The `for-loop` iteratively calculates $\pi/4$ on `sum` (lines 4 to 7). Notice that there is no need for casting the `i` and `sign` variables to `vpfloat` since the compiler can automatically handle type conversions when necessary. A constant suffixed by `v` represents a `vpfloat`.

We have added support for the data type on LLVM [17], a state-of-the-art compiler framework. A new type was added to the LLVM Frontend and Type System so that `vpfloat` is recognized at all parts of the toolchain, i.e., from the frontend to the backend. Due to the popularity and community-driven development model of RISC-V, we have also extended the RISC-V LLVM backend to support the new ISA extension illustrated in Table I and the new data type. Additionally, the RISC-V GNU Assembler and Linker were expanded to generate executable code for the coprocessor ISA extension.

Listing 2 shows a snippet of the assembly code generated by LLVM for Listing 1. Coprocessor instructions are found at lines 8-12, 14-15, 17-18, 24-26 and 30. The code starts by setting up MBB and WGP to their correct values (lines 2-5). Notice that MBB is set to 42 bytes, which corresponds to variable slot size. WGP is set to 384 bits (48 bytes) since it must be a multiple of 64 bits. From lines 7-20 the `for-loop` calculates the pi function through coprocessor instructions. Line 15 calls the function which implements VP division. As explained in Section III, the coprocessor can operate in either intervals or scalars, as it adopts the UNUM format. The presence of `gguess` instructions indicates that the unit was configured to work with intervals. A compiler flag (*vpfloat-scalar*) configures the system to work with scalars: i.e it can be used to eliminate the generation of `gguess` instructions.

```
1           ...
2           addi      a3, zero, 42 # MBB = 42 Bytes
3           addi      a4, zero, 6  # WGP = 6*64 bits
4           smbb      a3
5           swgp      a4
6           ...
7   .LBB1_2: sext.w   a0,  s0 # %for.body
8           fcvt.x.g gt0, a0
9           gmul      gt0, gt0, gs1
10          gguess    gt0, gt0
11          gadd      ga1, gt0, gs2
12          gguess    ga1, ga1
13          sext.w    a0,  s1
14          fcvt.x.g ga0, a0
15          call      vpdiv
16          neg       s1,  s1
17          gadd      gs0, gs0, ga0
18          gguess    gs0, gs0
19          addiw     s0,  s0, 1
20          blt       s0,  s2, .LBB1_2
21  # %bb.3:      # %for.cond.cleanup.loopexit
22          lui       a0,  %hi(.LCPI1_4)
23          addi      a0,  a0, %lo(.LCPI1_4)
24          ldgu      gt0, (a0)
25          gmul      ga0, gs0, gt0
26          gguess    ga0, ga0
27          j .LBB1_5
28  .LBB1_4: lui      a0,  %hi(.LCPI1_0)
29          addi      a0,  a0, %lo(.LCPI1_0)
30          ldgu      ga0, (a0)
31  .LBB1_5:      # %for.cond.cleanup
32          ...
33          ret
```

Listing 2: Snippet of the assembly code for Listing 1: VP instructions are in lines 8-12, 14-15, 17-18, 24-26 and 30.

B. Current compiler limitations

To the authors' knowledge, this is the first work that proposes tuning the data size for a VP unit by means of an extension of the C language. Nevertheless, the current implementation has some known limitations:

1) Constant variable slot size selection: the compiler only supports constant size values in variable declaration. This prevents users from writing applications that change precision during runtime.
2) Memory size and internal computation precision handling: our unit offers the capability of changing the memory slot size (MBB) and the computing internal precision (WGP) independently. By default the language sets for each variable a WGP value consistent with its memory slot size. As previously explained, WGP can be manually modified. Furthermore, the compiler fixes to (4, 9) the UNUM environment of variables (up to 16 bits of exponent and up to 512 bits of mantissa).
3) Multiple precisions at a time: since variables addresses and sizes are defined in compile time, we take a conservative approach of restricting to one precision at a time.

V. Experimental results

This section presents the methodology used for our experiments. Results are divided into two types of tests: 1/ performance evaluation comparing our approach to the state-of-the-art, and 2/ byte-aware precision analysis showing

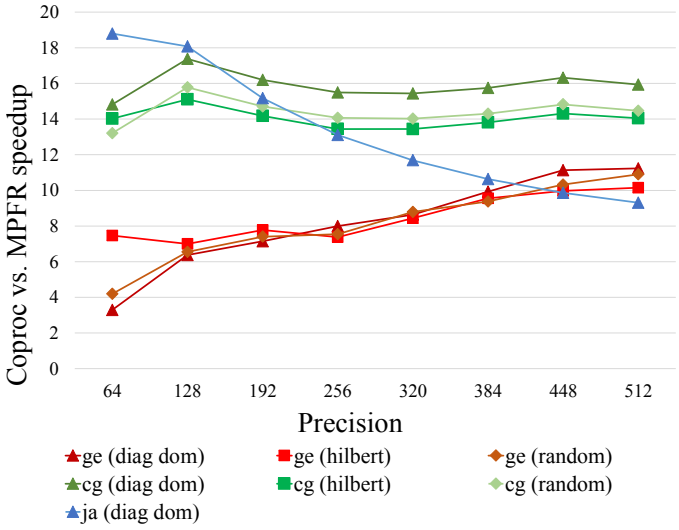


Fig. 5 Performance comparison between our VP coprocessor unit and the MPFR library: speedups between 3.5x and 18x.

how the performance and error behaviour according to the selected slot size for variables (MBB). Results are collected from a Xilinx Virtex-7 FPGA implementing our system (Section III) and applications are executed in a bare-metal environment.

A. Performance evaluation

For our performance evaluation, we have selected three matrix-based linear solvers as case studies for Variable Precision (VP): 1/ Gauss elimination (GE), 2/ the Conjugate gradient (CG) and 3/ Jacobi (JA). These algorithms execute long chains of multiply-addition operations that tend to accumulate errors, so they are suitable for experimentation. Three different matrices were selected for performance and error evaluation: a Hilbert 15×15 matrix (Hilbert), a randomly generated 24×24 matrix (random) and a 40×40 dominant diagonal matrix (diag dom). Applications GE and CG were compiled and run for all three configurations, while JA used only diagonal-dominant matrix due to algorithm constraints.

As a baseline, since no hardware solution is available for comparison, we have implemented the same benchmarks using MPFR [8]. For a more fair comparison between hardware- and software-based solutions, MPFR was accelerated by implementing some of its frequently used routines in assembly. More specifically, we have implemented GMP routines unable to take advantage of the `mulhu` [18] RISC-V instruction. We are able to reduce the number of instructions of `umul_ppm` and `mul_1` GMP routines from 25 and 12 to 11 and 3, respectively. These routines are extensively used for many mathematical functions within the library. Because MPFR uses GMP routines underneath, the implemented routines help to improve MPFR applications.

Fig. 5 shows the performance comparison between our solution and the baseline. Having hardware support for higher-than-standard precisions shows a clean advantage over a software solution with speedups between 3.5x and 18x. We notice how speedups vary according to the application and precision used. They are in function of the algorithm types, while matrices sizes and types have little influence over them.

B. Byte-aware precision analysis

In the second experiment, we measured the impact on execution time and computational error of a kernel, varying the slot size of VP variables (MBB, from 1 to 68 Bytes) and the computation precision (WGP, from 64 to 512 bits). As kernels we have chosen the Gauss elimination algorithm using different sizes of a Hilbert matrix and a randomly generated vector as inputs. Due to our compiler limitations (see Section IV-B), we have implemented the applications in assembly using the instructions described in Section III. The aim of this experiment is two-fold: 1/ understand how precision and error behave according to the adopted slot size, and 2/ observe the relation between memory and computational precision.

Fig. 6a-6c and Fig. 6d-6f depict how the latency in clock cycles and the computational error behave according to the computation precision (WGP) and to the slot size (MBB) used for variables. Every color on the graphs correspond to a different WGP value, and the x-axis denotes the MBB variation. For the application latency we can observe a linear increasing varying the WGP and MBB coprocessor parameters.

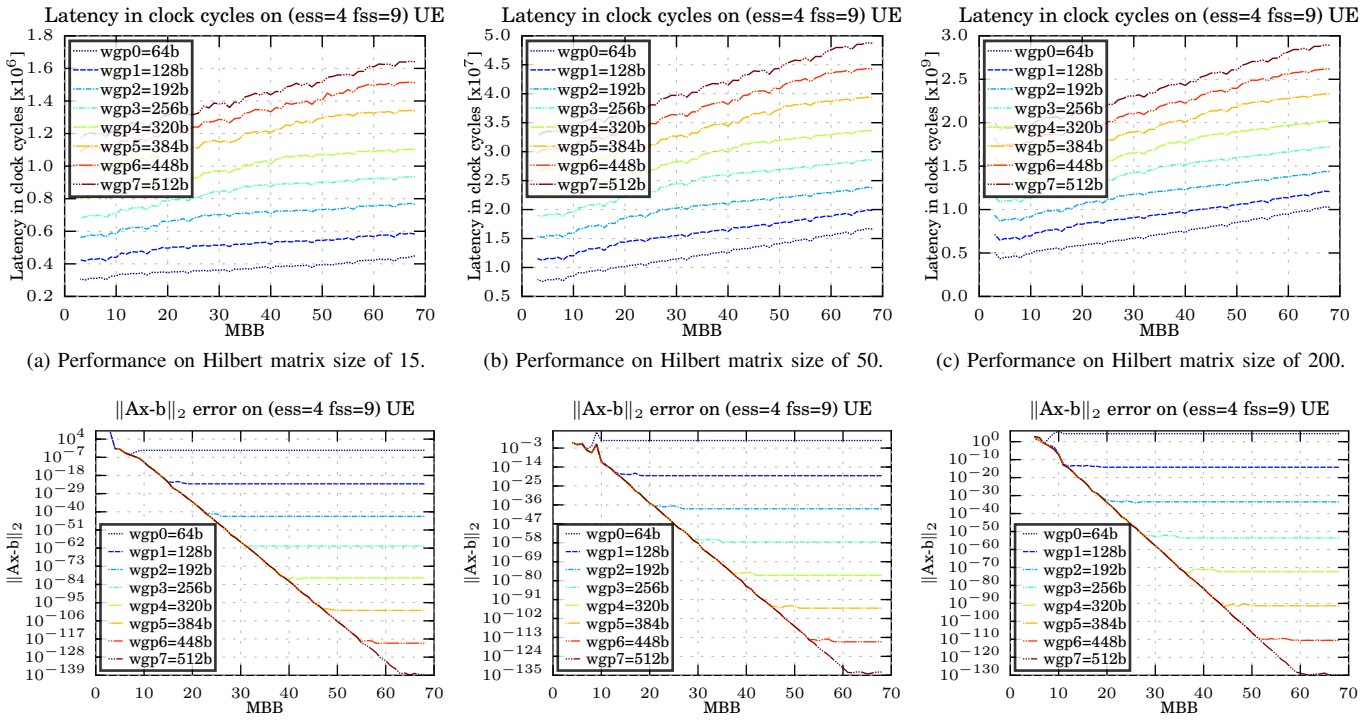
For the computational error we can observe a linear decrease (in logarithmic scale) varying the WGP and MBB coprocessor parameters. The flat zone for each WGP values appear when the mantissa precision corresponding to the chosen MBB value is greater than the specified one in WGP.

Looking at these results it is possible to compute the optimal WGP-MBB points where the latency is minimal and having the same precision, taking into consideration the expected average error. Counter-intuitively, the latency plots results show that the overhead of having misaligned memory accesses is not dramatic and does not considerably impact performance. This is due to our optimized LSU which is generating the minimum number of load/store operations depending on the value of the misaligned access. Moreover, we also observe a finer relation between memory slot size and computation precision. Varying WGP and keeping MBB constant does not improve the computational error, which means that memory slot size and computation precision must be managed together. As expected [19], the roundoff error is bounded and behaves linearly according to the problem size.

VI. Conclusion

We described a computing unit implementing variable precision floating point operations of up to 512 bits of precision in both interval and scalar arithmetic. The main contributions are: 1/ an integrated, fully pipelined co-processor with a memory subsystem supporting high and variable precision; 2/ byte-level storage granularity for variable precision, suitable for a variety of scientific applications; and 3/ seamless integration as a new first-class data type in the C language, with specific compiler support and transparent type conversion.

Compared with the MPFR software library [8], the proposed unit achieves speedups between 3.5x and 18x with comparable



(a) Performance on Hilbert matrix size of 15.

(b) Performance on Hilbert matrix size of 50.

(c) Performance on Hilbert matrix size of 200.

(d) Precision on Hilbert matrix size of 15.

(e) Precision on Hilbert matrix size of 50.

(f) Precision on Hilbert matrix size of 200.

Fig. 6 Latency and precision measurements of a Gauss kernel (lower is better) using the proposed modified UNUM type I format in the (4, 9) UE, varying the data memory footprint (MBB), the internal computing precision (WGP), and the input Hilbert matrix size.

accuracy. Further experiments demonstrates very little performance degradation when operating on misaligned data.

The compiler flow is currently limited in its ability to support multiple precision and storage formats within a given function. Scientific applications also lack a variable precision implementation of high-performance numerical libraries, such as OpenBLAS [20]. Given our promising performance results and seamless memory subsystem integration, we believe it will be possible to address these shortcomings through incremental refinements to existing programming models and the HPC software stack.

Acknowledgments

This work was partially funded by the French *Agence nationale de la recherche (ANR)* for project IMPRENUM (Improving Predictability of Numerical Computations) under grant n° ANR-18-CE46-0011.

References

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*.

[2] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, 2017.

[3] John L. Gustafson, *The End of Error: Unum Computing*, ser. Chapman & Hall/CRC Computational Science. Chapman and Hall/CRC, 2015.

[4] A. Bocco, Y. Durand, and F. de Dinechin, "Hardware support for UNUM floating point arithmetic," in *PRIME*, June 2017, pp. 93–96.

[5] F. Glaser, S. Mach, A. Rahimi, F. K. Grkaynak, Q. Huang, and L. Benini, "An 826 MOPS, 210uW/MHz unum ALU in 65 nm," in *International Symposium on Circuits and Systems*, May 2018, pp. 1–5.

[6] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: The good, the bad and the ugly," in *CoNGA'19*, March 2019.

[7] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating-point arithmetic," in *15th Symposium on Computer Arithmetic*. IEEE, 2001, pp. 155–162.

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

[9] F. Johansson, "Arb: a C library for ball arithmetic." *ACM Communications on Computer Algebra*, vol. 47, no. 3/4, pp. 166–169, 2013.

[10] A. Bocco, Y. Durand, and F. de Dinechin, "SMURF: Scalar multiple-precision unum risc-v floating-point accelerator for scientific computing," in *CoNGA'19*, March 2019.

[11] ——, "Dynamic precision numerics using a variable-precision UNUM type i HW coprocessor," in *ARITH'19*, 2019.

[12] "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[13] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[14] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: The MPI core*. MIT press, 1998, vol. 1.

[15] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07. New York, NY, USA: ACM, 2007.

[16] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Intl. Symp. on Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[18] A. Waterman and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa," Tech. Rep., May 2017.

[19] C. B. Moler, "Iterative refinement in floating point," *J. ACM*, vol. 14, no. 2, pp. 316–321, Apr. 1967.

[20] Z. Xianyi, W. Qian, and Z. Chothia, "OpenBLAS," *URL: http://xianyi.github.io/OpenBLAS*, 2014.