Xvpfloat: RISC-V ISA extension for variable extended precision floating point computation

Eric Guthmuller, César Fuguet, Andrea Bocco, Jérôme Fereyre, Riccardo Alidori, Ihsane Tahir, Yves Durand

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

Xvpfloat: RISC-V ISA Extension for Variable Extended Precision Floating Point Computation

Eric Guthmuller, César Fuguet, Andrea Bocco, Jérôme Fereyre, Riccardo Alidori, Ihsane Tahir, and Yves Durand

Abstract—A key concern in the field of scientific computation is the convergence of numerical solvers when applied to large problems. The numerical workarounds used to improve convergence are often problem specific, time consuming and require skilled numerical analysts. An alternative is to simply increase the working precision of the computation, but this is difficult due to the lack of efficient hardware support for extended precision. We propose *Xvpfloat*, a RISC-V ISA extension for dynamically variable and extended precision computation, a hardware implementation and a full software stack. Our architecture provides a comprehensive implementation of this ISA, with up to 512 bits of significand, including full support for common rounding modes and heterogeneous precision arithmetic operations. The memory subsystem handles IEEE 754 extendable formats, and features specialized indexed loads and stores with hardware-assisted prefetching. This processor can either operate standalone or as an accelerator for a general purpose host. We demonstrate that the number of solver iterations can be reduced up to $5\times$ and, for certain, difficult problems, convergence is only possible with very high precision ($\geq$384 bits). This accelerator provides a new approach to accelerate large scale scientific computing.

Index Terms—Instruction sets, Scientific computing, Application specific processor, Floating point arithmetic, Linear algebra, High precision arithmetic, Coprocessor, RISC-V.

1 Introduction

Scientific applications for physics, molecular chemistry, structure engineering, data analysis, etc. extensively exploit linear algebra kernels, e.g. linear solvers or eigensolvers. Indeed, an enormous fraction of the time on High-Performance Computing (HPC) systems is spent executing these kernels. For example for computational fluidics they represent up to 80% of user time [1]. New approaches are needed to improve the overall computational efficiency and deal with larger datasets.

The size of the problems to be solved is continuously increasing [2] making them more sensitive to round-off and quantization errors. Indeed, one of the focuses of numerical analysis research is to improve the stability of solvers through techniques such as orthogonalization, preconditioning, etc. [3], [4]. Some of these techniques to improve stability have a higher computational and memory cost than the solver itself [5].

In this paper, we present a holistic approach to reduce time to solution for large linear and eigen-solvers. This time to solution consists of several components, including the human time required to modify the problem so that it converges, iteratively selecting the correct solvers and parameters, the compilation of the code, the execution of the solver and the analysis of the results. Our solution addresses all of these aspects.

The core of our solution is an enhanced RISC-V processor, supporting an Instruction-Set Architecture (ISA) extension for dynamic, high-precision arithmetic, including the associated software toolchain.

The benefits of our solution are :

i) Simpler solver algorithms can be used, which are more efficient and require less memory, and avoid compensation techniques such as preconditioning or re-orthogonalization.
ii) The precision can be adapted to the problem at hand, without re-compiling the applications.
iii) The convergence speed is increased. Due to the reduced number of iterations for convergence, the execution time is reduced, even though the high-precision elementary operations are actually slower.

Indeed, it has been shown that certain real-world problems do not converge at all using standard double precision, and that increased precision is a highly effective means to obtain a solution [6].

Our solution includes a software toolchain to facilitate the porting of scientific applications. Library support for writing code in C++ makes it easy to modify existing solvers to use extended precision. Furthermore, our software stack maintains compatibility with standard scientific compute libraries (e.g. Basic Linear Algebra Subprograms, BLAS), facilitating integration with existing scientific applications. It is important to note that our solution provides dynamic, extended precision, which means the precision can be changed at runtime, based on the requirements of the current problem, making it unnecessary to recompile the application code. This is unlike other solutions which require recompilation to change the numeric precision [7].

Extended precision is not supported by mainstream hardware platforms. Software emulation, with MPFR [8] or GCC Libquadmath [9] is possible, but the enormous slow-

- *The authors are with the Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France (email: {eric.guthmuller, cesar.fuguettortolero, andrea.bocco, jerome.fereyre, riccardo.alidori, ihsane.tahir, yves.durand}@cea.fr)*

down makes this approach unusable for large problems. Our objective is to augment a high-performance processor with efficient support for extended precision both in registers and in memory. This requires tight coupling with the core, which is possible in an open source ecosystem such as RISC-V. Our ISA extensions were developed for the RISC-V CVA6 core [10]. The original L1 D-cache for this core [10] has several limitations, thus to achieve high performance, we also improved the L1 memory system to better mask the latency for cache misses [11].

The main contributions of this paper are :

i) a RISC-V ISA extension for dynamic variable extended precision, that supports more than 53 bits of mantissa both internally and in memory, with runtime modifiable precision specified for each instruction;
ii) an efficient hardware implementation including an optimized load-store unit supporting unaligned arbitrary size memory accesses with strided indexing;
iii) a full Software Development Kit (SDK) for integration with solvers and scientific applications;
iv) a demonstration that all of the above can be integrated to improve the time to solution for linear solvers with large datasets.

The remainder of this paper is structured as follows: Section 2 presents related works on extended precision and in Section 3 we introduce the RISC-V ISA extensions for variable extended precision. Then, in Section 4 we present the micro-architecture of the Variable Precision Floating-Point Unit (VPFPU), followed by Section 5 which describes the integration of the VPFPU in the CVA6 core. The software stack is described in Section 6 and in Section 7 we present experimental results demonstrating the performance improvements. This is followed by the conclusions in Section 8.

2 Related Works

As early as 1964, Wilkinson [12] had established the close link between working precision and numerical convergence. Yet, the numerical analysis community still lacks an effective computing solution supporting arbitrary high precision.

In 1983, CADAC [13] proposed a comprehensive architectural study for a Floating-Point Unit (FPU) supporting up to 32 Radix 10 digits. Other FPU designs have been proposed later, such as CASCADE [14] with radix 16 support, or even online arithmetic [15], but all of these machines store the high precision values in a small, local memory and do not support large scale calculations with extended precision.

Many mainstream x86 processors provide "extended precision", which means that they use 64 significand bits with the same exponent size as standard double precision. This is the case for the Intel family processors (IA32, x86-64, and Itanium) for example which support an 80-bit "double extended" precision format, stored in memory on 96 or 128 bits. This is an improvement over 64-bit precision but falls short of the precision required to improve the stability of large computations, the focus of this work.

IBM z13, Power8 and Power9 series are the only commercial platforms to support quad precision (128 bits) floating point formats in hardware [16]. Nevertheless, no production computer today provides full hardware support for variable extended precision above 128-bits, either in the FPU or in memory.

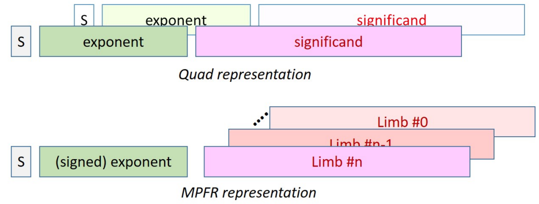


Fig. 1: Multiple word formats for the representation of floating-point numbers in memory

Multiple Word Representation

Multiple word arithmetic is a common practical approach which consists in encoding augmented floating point numbers using existing numeric formats. For example, the MPFR software library [8] implements variable precision floating-point numbers using sets of integers (limbs) for storing the significand (as represented in Fig. 1). This library provides exact rounding, and therefore it is widely used as the reference library for variable precision. Nevertheless, its computing and memory overhead prevents its use for large scale calculation (even though the latest release has been optimized for performance for the support of the binary64 and binary128 types [20]).

Other multiple-word implementations exploit error-free transformations which are variants of compensated summations or floating-point expansions [21]. The most practical method consists in implementing a "double-double" arithmetic with a pair of double format numbers. This technique is implemented by software libraries such as QDlib [7]. However, the overhead cost in terms of arithmetic operations is significant: an addition costs $20\times$ more in double double, $89\times$ in quad double compared to double [22]. Thus, several authors compensate this overhead by exploiting the parallelism and the speed of either GPGPUs with optimized libraries such as CAMPARY [23], or specialized SIMD or MIMD [24] processors. This approach is effective for double-double precision if GPU hardware is available, but for arbitrary precision above 128 binary digits, it is necessary to use dedicated hardware, combined with native format representations and operations.

Native Formats

There exist very few native formats for arbitrary precision floating-point numbers beside the current standard IEEE 754 [25]. In 2015, John Gustafson introduced the UNUM type I format [26] for interval arithmetic with arbitrary precision (represented in Fig. 2). In 2018, ETH Zurich [18]

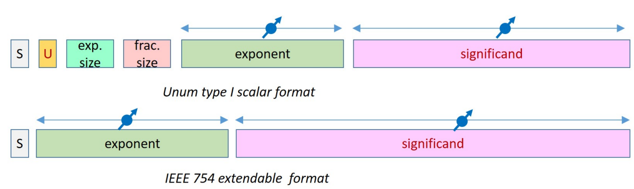


Fig. 2: Native formats for the representation of floating-point numbers in memory

TABLE 1: Related floating point computation implementations with support for variable precision

Name	Year	Floating point format	Implementation	Architecture
CASCADE [14]	1989	Radix 16 redundant signed-digit numbers	study	sliced FPU
e.g VLPLA [15]	2001	Online fixed point	FPGA	
VPFPAP [17]	2011	Radix 2 floats	FPGA	VLIW, fixed width FPUs
GRAPE9-MPX [17]	2015	Radix 2 floats	FPGA	SIMD
ETH UNUM ALU [18]	2018	Radix 2 floats (UNUM)	Silicon	ALU only
SMURF [19]	2019	Radix 2 floats (UNUM)	STM 28 nm FDSOI	RISC V & Schulte FPU
VRP (this work)	2021	Radix 2 floats (IEEE)	TSMC 7 nm	RISC V & Schulte FPU

implemented a complete UNUM based co-processor using a 65 nm technology, however, significand length was limited to 64 bits.

Our "SMURF" [19] processor also supported Unum type I with up to 512 bits of significand, both internally and in memory. Our new version, presented in this paper, brings closer integration with the RISC V core, has an improved FPU design, tailors the ISA extension for BLAS routines, and drops support for the UNUM representation in favor of the IEEE754-2008 extendable format.

Among other native formats for reals, *posit* [27] has gained some attention for deep learning and signal processing. However, it is not adapted for scientific computing, mainly because, in practice, its implementation is limited to 64 bits [28]. Furthermore, its relative representation error is not strictly bounded which is a common requirement for numerical algorithms.

Unlike *posit*, radix based representations (with radix R) are particularly suitable for scientific computing, because of the straightforward relation $\epsilon_m = R^{-p}$ between the significand size p and the actual machine Epsilon ϵ_m , which is the reference quantity for convergence analysis. Today, the radix 2-based format is dominant.

The IEEE 754-2008 revision defines extended and extendable formats [25]. This radix 2-based representation comes with a variable mantissa size, which may be much wider than 80 bits. The binary128 type, which is becoming widely used in scientific and data analysis software, is an instance of this format. Extending precision beyond ≈64 bits of mantissa has important micro-architectural implications, as it is not practical to directly implement hardware multipliers beyond this width. Therefore, most implementations of binary128 are emulated by software libraries such as GCC libquadMath [9]. However, this involves multiple native arithmetic instructions and multiple data accesses for each individual operation, which slows down the operations by $10\times$ to $60\times$ [29] when implemented using a classical CPU.

FPU Architectures for extended precision

There are two leading philosophies for variable-precision FPU architectures. The Kulisch architecture leverages a fixed-point accumulator, large enough to hold the whole exponent dynamic of a floating-point number. Floating-point operations are done within this accumulator through fixed-point basic operations, but precision is lost whenever data is stored back to memory. This solution has been implemented on an FPGA, but not realized in HPC computers [30].

Schulte and Swartzlander [31] took an orthogonal approach. Their architecture exploits a floating-point self-descriptive register format. The significand occupies a configurable number of 64 bits wide segments. The philosophy behind Schulte's work is to expose full fledged VP operations to the ISA. This means that operations span the full length VP number and include rounding when needed: the succession of partial sums and products and rounding is managed by internal sequencers. Unlike the Kulisch architecture, the Shulte's one applies more alignment and rounding steps within internal floating point operations. But this scheme enables the use of internal intermediate variables without loss of precision. We have adopted this solution for internal operations in our previous realization [19] and in this work.

VPFPAP [17] (similar to VV in [32]) adopts a different scheme for the Kulisch architecture: the processor exposes the elementary operations on mantissa segments (again 64-bit wide). The software is responsible for sequencing these elementary operations, and exploits 64 entries of internal memory to store the intermediate results. The instruction level parallelism of VPFPAP's VLIW structure improves the throughput via software unrolling. However, this solution is meant for accelerating specific functions, and does not support the storage of VP numbers in main memory.

The GRAPE-MP [33] reports to be the first practical hardware implementation of high-precision floating-point units. It is a family of accelerator boards able to perform calculations in quad to octuple precision (ie 112/176/240 bits of significand). Rounding is relaxed, making it difficult to analyse convergence properties. The system was originally implemented on a "structured" gate array ASIC from Nextreme eASIC NX2500. The following version used multiple FPGAs, taking advantage of the FPGA embedded multipliers.

In this work, our processor, with full support for variable precision computation up to 512 bits, has been implemented on an ASIC in TSMC 7 nm technology. Table 1 summarizes the most significant hardware implementations of extended precision processors.

3 *Xvpfloat* RISC-V ISA EXTENSION

To fully benefit from faster convergence resulting from higher precision arithmetic, an optimized FPU is required. Furthermore, it is important that the precision can be controlled at runtime, to avoid the need to recompile or manage multiple executables. As seen in Section 2, no such hardware implementation exists, thus we propose a RISC-V ISA extension called *Xvpfloat* for variable extended precision floating point computation. We have implemented this ISA extension in the CVA6 [10] 64-bit open-source RISC-V core.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

3.1 Design Principles

As a first principle, we decouple the floating point binary representation in memory from the one used internally. In memory, numbers are stored following the IEEE 754-2008 [25] extendable format, which imposes an 8-bit granularity. This format supports independent configuration of mantissa and exponent bit-widths.

Internally, 32 logical registers, the *P*-registers, hold floating point numbers at the maximum precision supported by the hardware. The mantissa is kept normalized. Hence, it is encoded with a hidden bit implicitly fixed to 1. The internal fixed-width exponent is stored in two's complement representation. To encode numbers that we cannot represent with normalized mantissa, we leverage additional flags beside the sign bit: zero, sNan, qNan and inf. In order to exploit fixed-size hardware, the mantissa is logically split in chunks of 64 bits. Thus a field "*L*" holds how many chunks are valid. This also allows to optimize performance and energy consumption when less-than-maximum precision is required. In general, software has no need to manipulate this internal representation, although these fields are visible. *Xvpfloat* instruction fields limit hardware implementations to a maximum 54-bit exponent size and a maximum 1984-bit mantissa size (in steps of 64 bits). The total *P*-register size thus ranges from 146 to 2048 bits depending on the implementation parameters.

As precision is a problem-specific requirement, we avoid encoding it in the instruction format to limit duplication of binaries. Runtime precision is specified via 24 *environment registers*:

- 8 *ec* registers for internal computation, holding: precision of the result's mantissa in bits (Working Precision, *WP*) and rounding mode;
- 8 *evp* registers for variable precision floating point load and store operations, holding: total bit-size in memory, exponent size, rounding mode and a 16-bit unsigned `stride` fields used for accessing arrays in memory (akin to lda/inc_x/inc_y parameters of BLAS functions);
- 8 *efp* registers for 16/32/64 bits floating point conversion operations, holding the rounding mode.

A 3-bit field in the *Xvpfloat* instructions provides the source environment register used at execution. These environment registers are modified through dedicated instructions. Proper dependency and forwarding are computed in the issue stage to avoid the need for costly barriers while updating these registers. The load store instructions calculate a memory address using a base address and a stride which takes into account the size of the stored value, making the code independent of the selected precision and saving multiple integer instructions. The stride parameter for load and store instructions is targeted at optimizing handwritten-assembly BLAS kernels.

We implement RISC-V ISA extensions in the standard way by using the *custom-0* 25-bit encoding space. We are thus compatible with existing standard RISC-V extensions, and in particular with RV64GC which would be the target of a general purpose high performance core.

TABLE 2: *Xvpfloat* instructions

Mnemonic	`Operation`
PGER rt, ea	`rt = ea`
PSER et, ra	`et = ra`
PLE pt, evpi, ra{, #index}	`vpfloat<evpi> *tab = ra` `pt = tab[index*stride]`
PSE pb, evpi, ra{,#index}	`vpfloat<evpi> *tab = ra` `tab[index*stride] = pb`
PMV.P.P pt, pa	`pt = pa`
PMV.P*field*.X rt, pa	`rt = pa[`*`field`*`]`
PMV.X.P*field* pt, ra	`pt[`*`field`*`] = ra`
PCVT.P.H rt, pa, efpi	`rt = (float16_t) pa`
PCVT.P.F rt, pa, efpi	`rt = (float32_t) pa`
PCVT.P.D rt, pa, efpi	`rt = (float64_t) pa`
PCVT.H.P pt, ra	`pt = (float16_t) ra.H[0]`
PCVT.F.P pt, ra	`pt = (float32_t) ra.W[0]`
PCVT.D.P pt, ra	`pt = (float64_t) ra`
PADD pt, pa, pb, eci	`pt = pa+pb`
PSUB pt, pa, pb, eci	`pt = pa-pb`
PRND pt, pa, eci	`pt = pa+0.0`
PMUL pt, pa, pb, eci	`pt = pa*pb`
PCMP.EQ rt, pa, pb	`rt = pa == pb ? 1 : 0`
PCMP.NEQ rt, pa, pb	`rt = pa != pb ? 1 : 0`
PCMP.GT rt, pa, pb	`rt = pa > pb ? 1 : 0`
PCMP.LT rt, pa, pb	`rt = pa < pb ? 1 : 0`
PCMP.GEC rt, pa, pb	`rt = pa >= pb ? 1 : 0`
PCMP.LEQ rt, pa, pb	`rt = pa <= pb ? 1 : 0`

3.2 ISA Overview

Table 2 lists instructions of the *Xvpfloat* ISA extension. *PGER* and *PSER* allow for reading and writing the 24 environment registers. *PLE* and *PSE* are *P*-register load and store instructions that can be configured through *evp* environment registers to read and write IEEE 754 *extendable* floating point numbers from/to memory. Fast paths are implemented for loading and storing standard *half*, *float* and *double* formats. All load/store instructions support subnormals in memory as long as exponent value fits in internal exponent range, otherwise the value is rounded to infinity or zero. *PCVT.*.** instructions are conversion instruction from/to the integer register file. *PADD*, *PSUB*, *PRND* and *PMUL* implement a basic arithmetic set of operations. Finally, *PCMP.** are comparison instructions allowing in particular conditional jumps.

Xvpfloat implements the same rounding modes as those defined by RISC-V floating point extension: Round to Nearest ties to Even; Round towards Zero; Round Down; Round Up; Round to Nearest ties to Max Magnitude. These modes are specified by environment registers, enabling runtime instruction-level control. Rounding is defined at bit granularity for both internal and memory operations, to allow for example reproducibility in transcendental functions computation. The memory footprint of scalar variables is aligned to a byte boundary through the use of padding as memory accesses are performed with byte granularity.

Finally, memory consistency is explicitly not enforced between *Xvpfloat* load/store instructions and RISC-V standard load/store instructions (integer or floating point), thus fences are needed to ensure memory consistency, if applicable.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

4 Arithmetic Pipeline of VPFPU

We have developed a dedicated VPFPU which integrates multiple arithmetic pipelines that work on multiples of 64-bit mantissa chunks and implements the instructions listed in Table 2. Our current implementation has an upper limit of a 512-bit mantissa and a 18-bit internal exponent size.

The path delay of the pipeline stages was tuned to match the critical path in the CVA6 RISC-V core [10], while optimizing throughput and latency without impacting the maximum clock frequency, in the 7 nm target technology. We found that within the available cycle time, it was possible to implement a 128-bit adder, a 512-bit barrel shifter or a 512-bit leading-zero-counter (*lzc*). A 64-bit multiplier requires 3 cycles at our 1.25 GHz target clock frequency.

Thus, we adopt an iterative approach to support operations up to 512-bits. For example, a 512-bit addition requires four iterations through the 128-bit adder. Similarly, if two input operands with 128-bit mantissa precision need to be multiplied, it requires four iterations through the 64-bit multiplier block. In general, the number of iterations required to execute a given instruction, depends on:

i) the precisions of input operands,
ii) the exponent values of the input operands,
iii) the maximum output precision, associated to the environment register specified in the instruction.

To improve performance, we implement separate pipelines for addition, multiplication, load/store and comparison (*CMP*), as well as some other operations. Each pipeline supports multiple concurrent instructions, for a total of at most 17 instructions in flight in the VPFPU. To our knowledge, this is the first hardware implementation of a parallel floating-point unit supporting dynamic extended precision. This parallelism masks the throughput penalty of high precision operations. Section 7.2 shows how our optimized BLAS functions exploit Instruction-Level Parallelism (ILP) to that end. Moreover, the VPFPU implements a dedicated high throughput Load-Store Unit (LSU), distinct from the integer LSU.

The variation of bit-width within the sequence of elementary operators has the following design consequences:

- the data-path width varies from one pipeline stage to the next;
- additions and multiplications are performed iteratively;
- a stop-and-wait protocol preserves operation consistency while iterating on mantissa chunks.

Section 4.1 details the VPFPU micro-architecture and Section 4.2 shows its impact on performance.

4.1 Microarchitecture

Fig. 3 depicts the VPFPU floating point operators and their pipelines. Pipeline stages, corresponding clock cycle latencies (*lat*) and bit-width parallelism (*par*) are also displayed. The *pLD* and *pST* stages are 1-clock cycle latency full P-numbers buffers, which decouple the VPFPU pipelines from the CVA6 core. Control flow in the *pLD* stage is optimized to reduce buffering knowing that, for precision above 128-bits, throughput will be limited by following stages. Pipeline synchronization barriers in Fig. 3 are denoted by dashed lines ¦. The number of full *P*-number buffers are denoted with a flip-flop symbol. The n-th pipelines stage of an operator is marked with a n symbol. Operators not detailed in Fig. 3 have a one clock cycle latency, except for the *PCVT.P.** instructions that require two clock cycles.

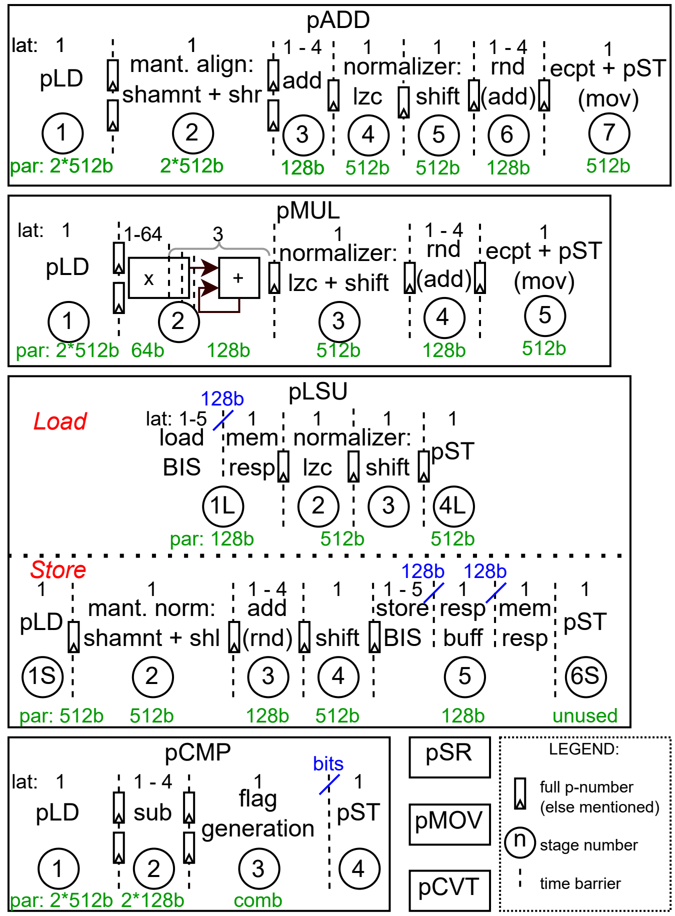


Fig. 3: VPFPU microarchitecture overview

4.1.1 pADD *Floating-Point Adder*

The *pADD* operator consists of seven pipeline stages. *PSUB* is implemented by flipping the second operand sign, and *PRND* by forcing to zero the second operand at the input of the *pADD* pipeline. Stage 2 computes the shift amount (*shamnt*) for the exponent alignment of the two operands, and then it aligns the mantissa with the lower input exponent by shifting it to the right (*shr*). The main addition operation 3 adds the aligned mantissas by iterating through two 128-bit adders. In case of different input signs, it computes in parallel *op*1-*op*2 and *op*2-*op*1, and it selects the result with a positive mantissa. Stages 4-6 perform the normalization and rounding of the addition result. They first compute the *lzc* 4 of the result to compute the shift amount needed for normalizing the mantissa performed by 5. Stage 6 rounds the normalized value increasing the mantissa by one Unit in the Last Place (ULP) according to the rounding policy specified in the environment, by iterating through a 128-bit adder.

4.1.2 pMUL *Floating-Point Multiplier*

The *pMUL* operator has five pipeline stages. Stage 2 multiplies the operands by iterating through a four-stage 64-bit

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

multiplier/accumulator. It minimizes the number of iterations by considering the precision of the operands, specified in their L fields encoding the number of valid 64-bit chunks: the number of partial multiplications is $op1.L \times op2.L$. The first three stages implement the 64-bit multiplier, the fourth accumulates the partial products on a 128-bit adder. The 1024-bit result is truncated to 512 bits, and the resulting underflow and overflow conditions are handled during normalization. Compared to the *pADD* pipeline, stages ③-④ perform the normalization and rounding of the multiplied result with one less cycle latency by exploiting the known range of the resulting mantissa.

4.1.3 pLSU *Load-Store Unit*

The *pLSU* has two internal pipelines: a four-stage load pipeline (1L)-(4L), and a six-stage store pipeline (1L)-(6S). Memory consistency between load and store operations is handled by keeping track of the addresses involved in memory operations.

Store pipeline: normalization is done in stages ②-③ (similar normalization design as *pMUL*). Subnormal numbers are supported by re-aligning the mantissa during normalization. Stage ④ implements the floating point memory format conversion, shifting the mantissa, and merging all its fields. A fast path for the standard IEEE 754 memory format conversions (half, float, double) reduces the latency of stages ②-④ to two clock cycles. Stages ⑤-(6S) send 128b write requests to memory through a dedicated interface and wait for the corresponding responses.

Load pipeline: data are loaded from memory (1L) through a 128-bit memory port interface, separate from the interface of the store pipeline. Data in memory can be in subnormal form, hence the data is normalized in two stages ②-③. A similar fast path for the standard IEEE 754 memory format conversions, permits to gain a clock cycle in stages ②-③.

Fig. 4 details the (1L) load stage. It masks the latency of the out-of-order L1 cache by generating multiple outstanding memory loads. For maximum memory bus utilization, considering the average cache latency, this unit can support up to 16 parallel 128-bit memory requests, with up to 8 parallel memory operations. This stage works with two internal tables: the pending load table (PLT) ◇3 to keep track of alive memory operations, and a table to store memory responses ◇4.

This unit receives the load address (*addr*), an identifier for the memory operation (*ID*), and control signals (*ctrl*), including the size of data in memory (*BIS*). For every new operation, the allocator block ◇2 searches for space in the memory response table ◇4. If there is enough contiguous space, it ◇2 updates the PLT ◇3 at the first free entry, while the request generator ◇1 sends all the 128-bit memory requests to the memory interface in a single burst. The PLT stores among other things:

- the original instruction identifier (*ID*);
- the address in table ◇4 of the first chunk of the loaded data (*mch*);
- the address offset (*soff*) in case of misaligned memory operations;
- and, the number of expected memory responses to end the load operation (*cnt*).

Memory requests are associated to an 8-bit request identifier (*mem_req.id*). The first bit is used to distinguish between load and store operations. Three more bits contain the memory operation identifier, *mid*, pointing to the table ◇3. The remaining four bits point to the destination chunk in table ◇4 (*moff*). Upon reception of a load response, the data are written at the *moff*'th entry of table ◇4, and the corresponding counter of table ◇3 is decremented. When all responses have been received, the loaded data is output through the window extractor unit ◇5, which shifts and masks the content of table ◇4 according to the *mch*, *BIS*, and *soff* information. Finally, the entry of table ◇3 and the blocks in table ◇4 are freed for the allocator.

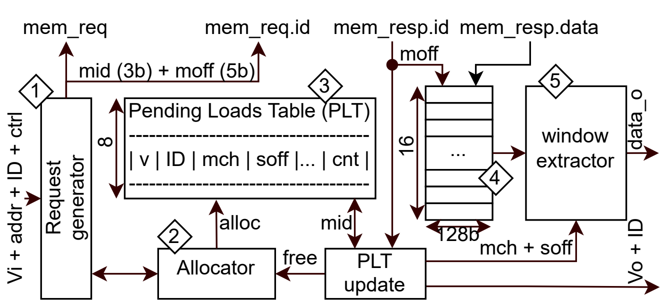


Fig. 4: VPFPU load unit microarchitecture ((1L) in Fig. 3)

4.2 Latency and Throughput

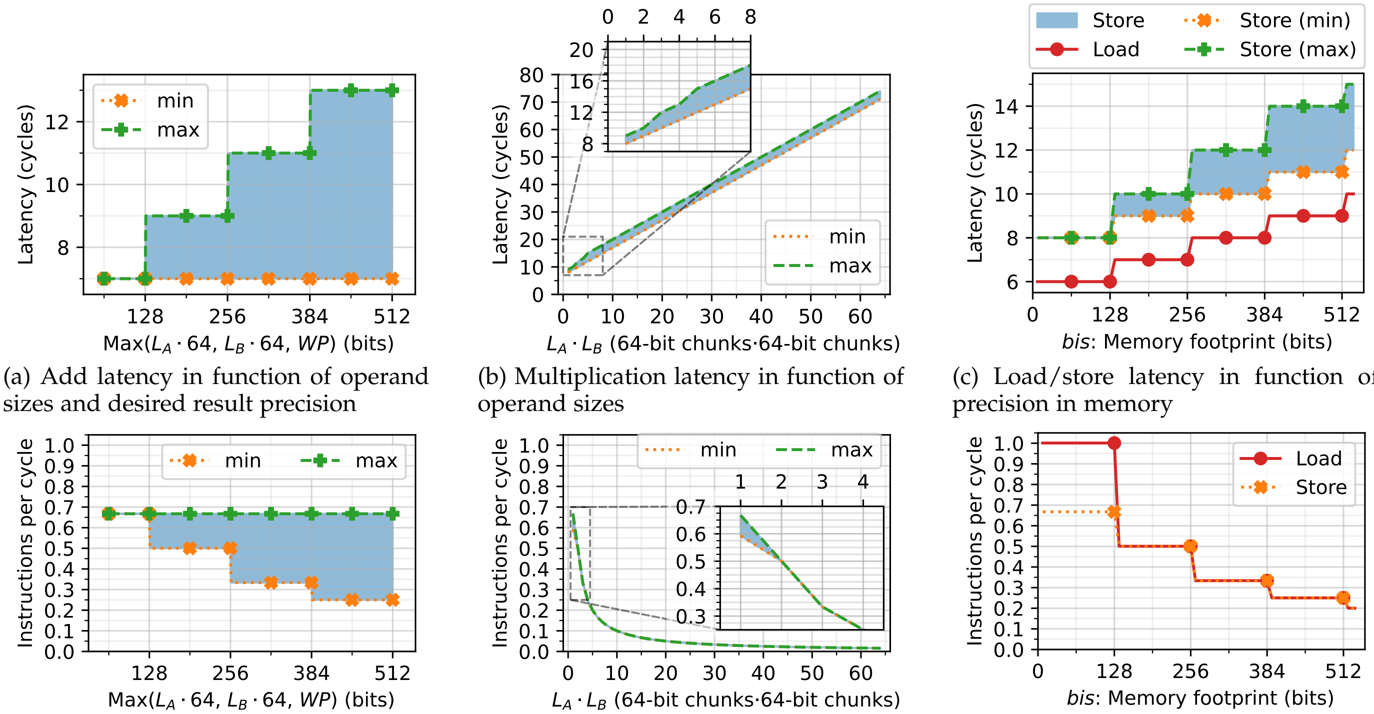
Fig. 5 depicts the latency and Instruction-Per-Cycle (IPC) throughput of the previously described operators, obtained by RTL simulation, in function of the number of 64-bit chunks of inputs L, the Working Precision WP or precision in memory. Since instruction performance depends on the value of operands, a floating point inputs set that covers the full range of operation latencies is fed to each unit. Latency is measured with an empty pipeline, while throughput is measured by continuously repeating the same operation with constant operands. A zero-latency memory is connected to the *pLSU* load-store unit and only aligned accesses are issued [1].

We observe that:

- The performance varies greatly with operands and with the computational precision. Addition, load and store instructions have a linear dependency on precision, while for multiplication the relation is quadratic.
- As discussed in previous section, the control flow in the first stage (*pLD*) of each pipeline is optimized for precision above 128 bits to reduce buffering. It results in suboptimal IPC at less than 128 bits of precision, mitigated by exploiting ILP in most real cases.
- The latency and throughput match the behaviour described previously: fixed-point adder stages (including during normalization) and the LSU iterate on 128-bit chunks while the multiplier iterates on 64-bit chunks.
- The addition performance is not only dependent on input and output precisions, but also depends on mantissa and exponent input values.

1. The impact of unaligned memory accesses is very low, variation is limited to an additional clock cycle of latency.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964



(a) Add latency in function of operand sizes and desired result precision

(b) Multiplication latency in function of operand sizes

(c) Load/store latency in function of precision in memory

(d) Add IPC in function of operand sizes and desired result precision

(e) Multiplication IPC in function of operand sizes

(f) Load/store IPC in function of precision in memory

Fig. 5: Latency of (a) add, (b) mul, (c) load and store instructions (empty pipeline). Instructions Per Cycle (IPC) of (d) add, (e) mul, (f) load and store instructions. IPC is measured by injecting 1000 identical and consecutive instructions.

5 VPFPU Integration in the VRP Core

We implemented the *Xvpfloat* ISA extension in the CVA6 RISC-V core and named this new core *VRP*. This core has been integrated into a multi-core chip implemented in 7 nm technology. It will be seen that the area overhead is reasonable (+66% compared to a CVA-6 without a FPU), and there is no degradation in the operating frequency. The core was also integrated into an FPGA platform for prototyping and performance analysis. In this section, we describe both these implementations.

5.1 Architecture

A VRP compute tile is made of multiple VRP cores. These cores are interconnected via two different Network-on-Chips (NoCs): a "memory" NoC and a "IO" NoC. The former provides access to the main DRAM via the cache hierarchy, and the latter handles IO peripheral accesses.

The memory NoC is a 512-bit wide AMBA™ AXI interconnect, with an inverted binary tree topology, where the leaves are the VRP cores, and the root is a bridge to the rest of the system. The IO NoC is used for configuration and peripheral access.

5.1.1 VRP Core

Each VRP core is a 64-bit, in-order, CVA6 RISC-V core extended with support for the *Xvpfloat* instruction set extensions (see Section 3). The core was configured to implement two commit ports and a 16-entry scoreboard to account for the variable and potentially high latency of *Xvpfloat* instructions. The pipeline was modified as shown in Fig. 6, to add support for these instructions.

In the original CVA6 core, functional unit results are stored in the scoreboard and only written in the register files in the commit stage. To avoid hazards, of particular concern with the long, variable execution latency of the *Xvpfloat* instructions, we used this existing scoreboard data-path for environment registers updates by the *PSER* instruction. However, to avoid storing full 538-bit P numbers in the scoreboard, we used register renaming for the P register file while reusing existing scoreboard fields to store renaming metadata. Thus, the P register file has 64 physical entries, two read ports and a single write port arbitrated at the output of the VPFPU. The renaming table is implemented as two tables, speculative and committed, to allow for single-cycle rollback in the case of exceptions. The scoreboard entries hold renaming metadata to properly compute hazards, retain physical register usage and update the logical-to-physical mapping at commit stage. In this way, the potential hazards in the VPFPU are handled by the CVA6 scoreboard using physical register dependencies.

Inside the VPFPU (see Fig. 6), each functional unit can take multiple cycles (see Fig. 5) to process an instruction. However multiple VPFPU functional units can operate in parallel, improving the IPC. There is a single issue port for all the functional units with a registered demux stage which distributes instructions to the corresponding unit. At the output of the functional units, there is an arbiter writing one P-number per cycle to the VPFPU register file. As detailed in Section 4, the VPFPU integrates a dedicated LSU (*pLSU*)

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

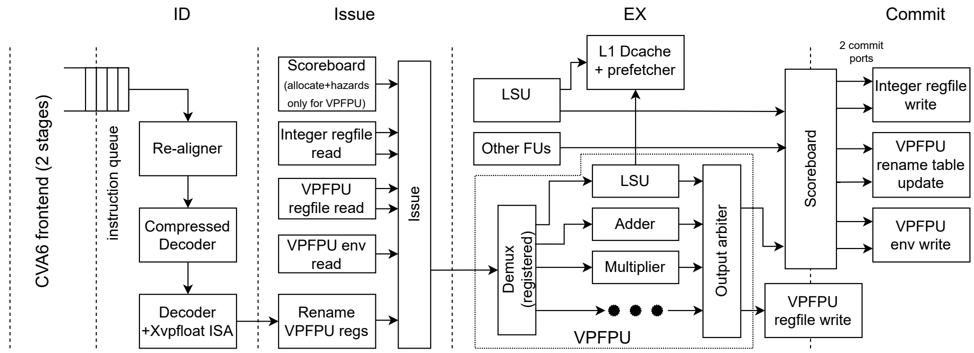


Fig. 6: Modified CVA6 pipeline microarchitecture

to enable high-bandwidth to memory. This LSU supports IEEE 754-2008 extendable formats and has a fast path for half, standard and double precision floating point numbers.

5.1.2 VRP Cache Memories

Each VRP core implements private L1 instruction and data caches. As the standard CVA6 data cache does not support multiple outstanding misses and has a 64-bit data-bus, it is not suited for high performance applications, thus we developed a new high-throughput data cache [11] which has been open-sourced [34]. For the VRP, this data cache is configured to be 32 KB in size, 4-way set-associative, with 64-byte cache-lines and an internal 128 bit data-width to match the VPFPU bus width. It also supports four independent affine prefetchers and a 128-entry Miss Status Handling Register (MSHR) which effectively mask memory latency. A typical configuration was used for the instruction cache (16KB, 4-ways, 64-bytes cache-line, set-associative). Moreover, in our implementation, the number of memory cuts has been minimized to reduce area and routing congestion.

5.2 7nm FinFet Hardware Implementations

TABLE 3: Post-synthesis cell area breakdown of a single VRP core in TSMC 7nm technology

Block	Logic	Registers	SRAM	Total	%
	Cell area (µm²)				
Frontend	866.1	2100.4	0.0	2966.5	2.1
ID stage	81.2	161.1	0.0	242.4	0.2
Issue Stage	6382.4	5598.8	14726.0	26707.2	19.3
Exececute Stage / VPFPU / pADD	5524.8	4297.6	0.0	9822.4	7.1
Exececute Stage / VPFPU / pMUL	5014.1	3256.8	0.0	8270.9	6.0
Exececute Stage / VPFPU / pLSU	9566.5	6922.5	0.0	16489.0	11.9
Exececute Stage / VPFPU / pCMP	764.2	2543.5	0.0	3307.6	2.4
Exececute Stage / VPFPU / Misc	1783.1	1574.0	0.0	3357.1	2.4
Exececute Stage / Total (w/ ALU)	25199.9	21243.6	0.0	46443.5	33.5
L1 Cache	7595.9	8889.3	41772.3	58257.5	42.0
Commit Stage	74.1	0.0	0.0	74.1	0.1
Miscellaneous	1505.3	2422.3	0.0	3927.6	2.8
Total	41704.8	40415.6	56498.3	138618.7	100.0

Eight VRP cores are being integrated in a multi-core HPC processor [35] implemented in TSMC™ 7 nm FinFet technology. We present physical synthesis results for a single VRP core implemented with Synopsys DesignCompiler™

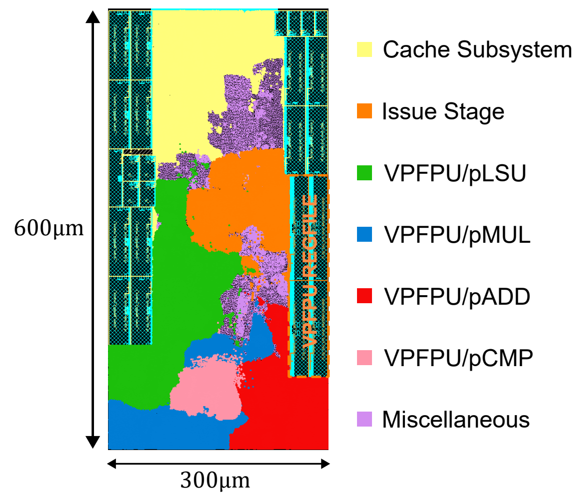


Fig. 7: VRP post-synthesis placement

(version 2021.06-sp5) in topographical mode. Synthesis was done with H300 HP standard cells using SVT and LVT in a multi-Vt methodology and the ARM™ memory compiler was used to generate SRAM macros.

The VPFPU register file has 2 read and 1 write ports and was implemented by mirroring two dual-port SRAM written in parallel ($2 \times 64 \times 538 \approx 69kb$), achieving better density compared to an implementation with flip-flops. The register file is divided into banks and the banks containing mantissa LSBs are clock-gated when not used, to save energy.

The design achieves a 1.25 GHz worst case frequency (100ps of extra margin) at 0.675V and 0°C with a physical area of 0.18 mm². The critical path in the VPFPU (in the adder) is similar in length to the CVA6 one which lies in the scoreboard. The floorplan of the VRP is shown in Fig. 7 and Table 3 gives the area breakdown of the main design blocks for a total cell area of 138,618.7 µm². To put the area in perspective, a similarly configured vanilla CVA6 supporting the RV64GC ISA with the same L1 cache has a cell area of 84,568 µm², showing that the integration of our 512-bit VPFPU results in a 61% area increase.

Fig. 7 shows the post-synthesis placement with per-block colouring. In this technology, the logic area is greater than the memory area, but this will shift progressively as SRAM is expected to scale slower in more advanced technology nodes.

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

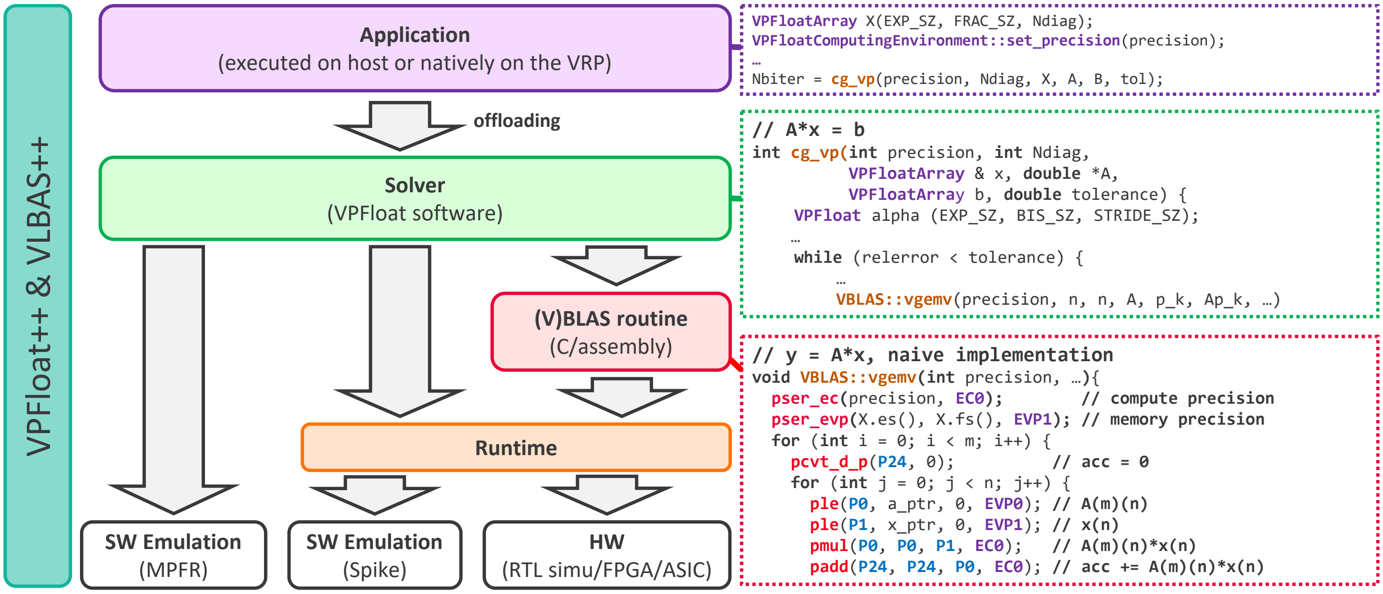


Fig. 8: VPSDK software stack architecture and code examples

6 Software Stack

6.1 Execution Model and Assumptions

The VRP is primarily intended for intensive algebraic computations. In a typical application such as finite element calculations, extended precision comes into play during the linear solving, which happens after domain decomposition and matrix construction. This phase takes place at the compute node level, as represented on Fig. 9, and this is where numerical stability issues occur.

While the RISC-V core is able to execute the full application stack, this is not currently a likely use model, as RISC-V is not yet in mainstream use for HPC applications. The VRP can instead be used as an accelerator in a x86/ARM host on a separate board with offloading through PCIe using DMA memory transfers. The application must then first copy the data to the accelerator's memory space, perform the compute and copy back the results. Otherwise, the VRP may be integrated on-chip with the application core where it can directly access the data through the on-chip memory hierarchy in a IO-coherent manner exploiting Shared Virtual Memory (SVM). These differences are hidden from the user by our software stack, shown in Fig. 8, and both offloading modes are supported.

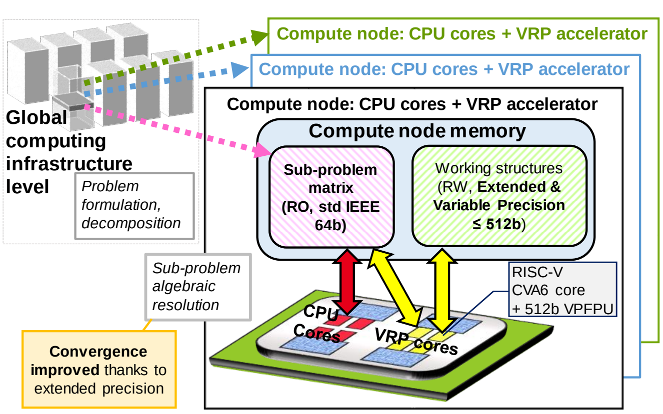


Fig. 9: Problem mapping on a typical HPC infrastructure

6.2 Application Programming Support

As shown in the code examples in Fig. 8, programmers can exploit the VRP using the Variable Precision SDK (VPSDK), which is a set of C++ classes supporting variable precision arithmetic. We used this VPSDK to implement several iterative solvers, in particular the Conjugate Gradient (CG) and its variants evaluated in Section 7. The VPSDK provides both scalar and array types, and an Application Programming Interface (API) for variable precision floating-point numbers. Scalar computation is done through C++ operators overloading for the *VPFloat* type. We provide support for extended precision BLAS level 1 (vector-vector) and level 2 (vector-matrix), including support for dense and sparse CSR/BCSR matrix formats derived from OSKI [36]. BLAS primitives have been partially hand coded in assembly to avoid instruction dependencies and maximize VPFPU utilization. A single BLAS implementation supports all precisions and is fine-tuned according to worst case operation latencies.

The VPSDK supports two execution targets: (i) the *Xvpfloat* ISA executed by the VRP, (ii) a high-level MPFR-based emulation layer, which is our golden reference. The MPFR backend allows us to prototype and validate algorithms on any Linux-supported hardware. Support for the *Xvpfloat ISA* has also been added to the standard RISC-V Spike model to provide low-level software emulation.

Finally, offloading from a host processor requires support for interrupts, peripheral drivers, memory allocation and other C++ hooks. This is provided via a minimal bare-metal runtime environment, running on the VRP.

7 Evaluation

7.1 Experimental Setup

In order to both verify the RTL with a controlled environment and to run full numerical applications, we used two

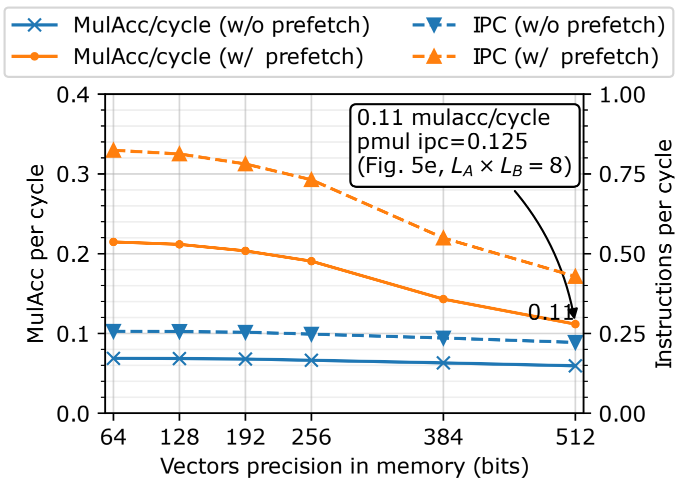


Fig. 10: Performance of dense 64b matrix (1k diagonal) by VP vector multiplication with a 100 cycles L1 miss latency at 64, 128, 192, 256, 384 and 512-bit vectors precisions

platforms: (i) an RTL simulation platform to obtain cycle-accurate results, for developing the BLAS-level software and for hardware validation, and (ii) a Xilinx™ VCU128 FPGA platform to execute numerical solvers on moderate size matrices in acceptable time. On this FPGA platform, offloading from a x86 host is done through PCIe. A multicore VRP design reaches 90 MHz on this Virtex UltraScale+ XCVU37P FPGA, each VRP core occupying about 172K LUTs (13%). A 64-bit load instruction takes 73 cycles on average (min 71, max 109) in case of L1 cache miss, when placing code and data in the 4GB DDR4 memory of the VCU128 board.

For simplicity, we chose to execute all software on a single core, however, extrapolation to multicore performance is straightforward, since the BLAS level functions can be efficiently parallelized. To keep the focus on the VRP and to prevent the memory system from being a bottleneck, we have chosen to focus on dense matrices and to use our affine hardware prefetcher which can effectively mask the memory access latency. We therefore converted the subset of matrices taken from the Florida sparse Matrix Collection [37], which were initially in sparse format, to a dense representation.

To ensure that all the numeric results are correct, we functionally validated all results, in terms of solver iteration counts and the magnitude of residuals, against our MPFR emulation layer. All solver results presented in this section are cross-validated with this approach by running identical application code on x86 hardware. For all following measurements, we set in-memory exponent size of vectors to 11 bits, while 18 bits exponents are used internally.

7.2 Matrix-Vector Multiplication Throughput

We first evaluate the performance of matrix-vector multiplication (BLAS "GEMV" routine), which is where dense linear solvers spend most of their time. Fig. 10 shows the achievable number of Multiply-ACcumulate (MAC) operations per cycle for varying numerical precisions. The left vertical axis shows the MAC per cycle when running BLAS GEMV ($y = Ax$) and the IPC is shown on the right axis. The double precision matrix (A) has a diagonal size of 1024. The horizontal axis represents total size in memory of the variable precision vectors (operand x and result y). We plot four different curves: MAC throughput and IPC, with and without the hardware prefetcher (see Section 5.1.2). Memory latency is fixed such as the execution of a 64-bit load instruction resulting in a L1 cache miss takes 100 cycles to complete.

The following observations can be made:

- The prefetching has a significant positive impact on performance: we observe a up to 300% IPC increase when enabled. Indeed, without prefetching, performance is limited by the memory subsystem, thus all subsequent results have prefetching enabled.
- The overall throughput is higher than combined IPC of individual instructions. This shows that the interleaving of functional units within the pipeline is effective.
- Since our core is in-order, the theoretical asymptotic MAC throughput is 0.33 MAC/cycle within the GEMV inner loop (load, multiply, add). In fact, the measure ranges from 0.21 to 0.11, which reflects the effects of less-than-ideal instruction scheduling and flow-control instructions overhead at low precision. Increasing precision mitigates these effects and MAC throughput is nearing the 0.125 IPC upper-bound of the multiplier at 512-bit precision (see Fig. 5e).

7.3 Execution of Linear Solvers

The significant metrics for measuring the impact of precision on linear kernels are (i) convergence speed, which refers to the number of iterations necessary for reaching that objective and (ii) execution time, measured in clock cycles, which sums up both arithmetic processing and memory access time.

Convergence is highly dependent on the data set (e.g. matrix and vectors values). For solving linear problems $Ax{=}b$ with Krylov subspace methods, the condition number of matrix A and its eigenvalue structure are decisive characteristics. Moreover, in all experiments hereafter, we define the key parameter *tolerance* as the threshold for the normalized residual value below which the algorithm is considered successful and stops iterating. Conversely, if the iteration count exceeds some fixed limit, the algorithm is considered to diverge. In our experiments, we arbitrarily fix the limit to five times the matrix diagonal size n. Also, by default, the right-hand-side vector b has all its components set to $1/\sqrt{n}$.

7.3.1 Conjugate Gradient (CG) Solver on Pseudo-Random Matrices

Iterative solvers are usually applied on large sparse matrices. Nevertheless, dense matrices are not uncommon and the case cannot be overlooked.

We generate random matrices with a refined version of the "Randsvd" method from [38], which guarantees fixed eigenvalue distributions and gives reproducible convergence with CG kernels. Specifically, we use two distributions "cliff" (resp. "step") which consist in three abutted segments with arbitrary slopes $0.1; 10; 0.1$ (resp. $10, 0.1, 10$).

Fig. 11a (resp. 11b) represents normalized iteration count (resp. cycle count) of the CG kernel running on a subset

This article has been accepted for publication in IEEE Transactions on Computers. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TC.2024.3383964

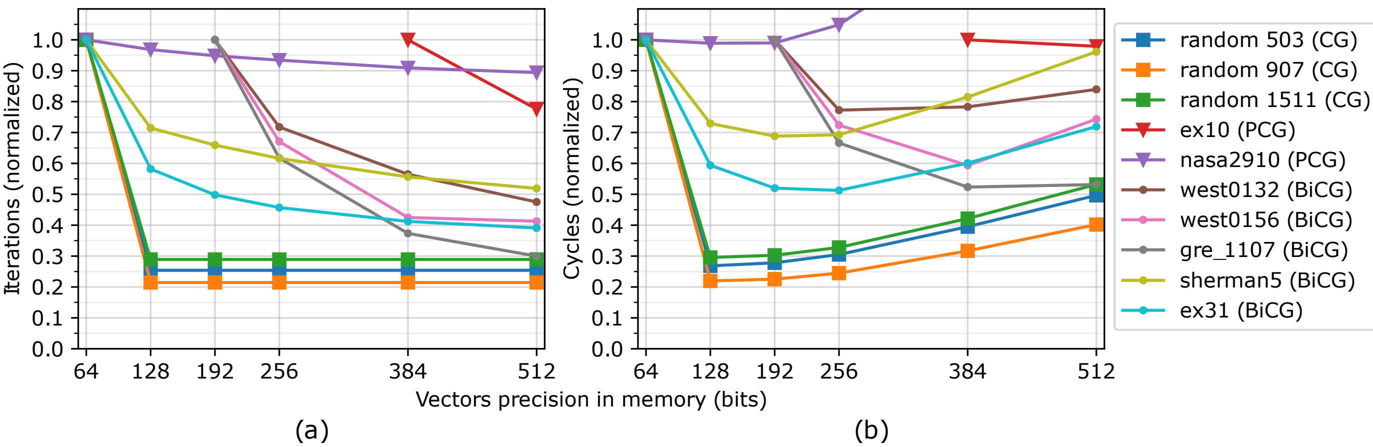


Fig. 11: Normalized iteration count and execution time for kernel CG, PCG and BiCG, using random matrices and matrices from SuiteSparse collection

of matrices generated according to Ransvd method, using "cliff" profile. Diagonal preconditioning does not work with random matrices, therefore the evaluation runs the original CG algorithm (as given in [39]). The first three curves show execution of three different random matrices of 503, 907 and 1511 diagonal sizes with vectors of different precisions, corresponding to different bit-sizes in memory. Inside the register file, the actual computation precision is aligned to the next 64-bit boundary, without negative performance impact on performance. We adopt a challenging value *tolerance*,which we arbitrarily set to 10^{-12}.

The CG results shown in Fig. 11 (■ traces) suggest two remarks:

- The effect of precision is decisive: the iteration count decreases significantly with precision starting at 128 with an up to 79% reduction, and remains unchanged for greater precisions.
- The cycle count "sweet spot" corresponds to 128-bit precision.

7.3.2 Solvers on Benchmark Matrices

We have selected a subset of the Florida sparse Matrix Collection [37], according to following criteria: we restrict ourselves to real matrices, which may be symmetric (for CG) or asymmetric (for BiConjugate Gradient); dense matrix diagonal size is less than 4,000 in order to guarantee an acceptable execution time on our prototyping platform. The condition number is not limited and may go up to 10^{16} in these examples.

We consider two commonly used linear algorithms: Preconditioned Conjugate Gradient (PCG) and BiConjugate Gradient (BiCG)[2].

Preconditioned Conjugate Gradient: PCG is commonly used when matrix A is a symmetric positive definite complex or real square matrix, which is usually the case for the resolution of partial derivative equations.

Preconditioning is a very common technique used to speed up the convergence of iterative solvers. We benchmark the simplest version, i.e. the Jacobi preconditioning which only involves the diagonal of the system matrix A. The method is exact in theory, but roundoff errors slow down or even prevent convergence. Running the classical algorithms with augmented precision limits the accumulation of error, and lowers the number of iterations necessary to attain the target tolerance.

The two PCG experiments shown in Fig. 11 (▼ traces) involve SuiteSparse matrices. The absence of result, i.e. when a part of the curve is missing, means that the run did not converge. This figure highlights two opposite behaviours:

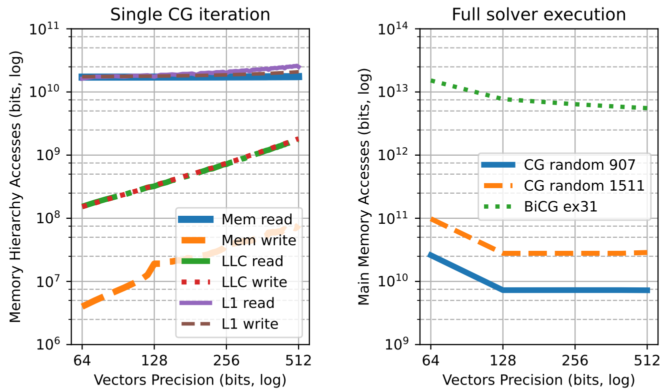
- When the matrix is "easy", e.g. well structured and well conditioned, preconditioning is very efficient and augmenting precision is of little use. This is the case for the strongly diagonally-dominant nasa2910 matrix.
- Preconditioning fails to bring convergence in double format in other matrices, such as ex10. A vector precision of 384 bits is necessary for convergence and upping it to 512 bits still brings some benefit in term of cycle count.

BiConjugate Gradient: BiCG is quite similar to standard CG, but works for asymmetric matrices. As CG, its memory cost is $\mathcal{O}(n)$ but with twice the number of vectors. The behaviour of this algorithm is less known than CG, and may be irregular. However, our experiments show the benefits of augmenting precision even if not formally backed with theoretical results as in the case of CG.

We display in Fig. 11 (● traces) the influence of precision on the kernel iteration and cycle counts. Tolerance of the BiCG solver is set to 10^{-12}. These experiments show that High (>128) precision is generally more beneficial to BiCG convergence, compared to previous Conjugate Gradient results. The best execution time on the VRP is reached at higher precision of 192 to 384 bits depending on the matrix. In other words, augmented precision alleviates the well-known forward instability of BiCG, which becomes a reliable and efficient tool for asymmetric problems.

All results presented here use the same tolerance. However, relaxing this parameter, i.e. using a larger value for the residual threshold, naturally decreases the number of iterations. It is our experience that the iteration count is less sensitive to tolerance in higher precision.

2. A more detailed description of the implementation of these algorithms on VRP can be found in [39]



(a) Memory hierarchy accesses during Conjugate Gradient iteration on a 16384×16384 64-bit matrix (hot caches)

(b) Total external memory accesses for full solver execution on benchmark matrices

Fig. 12: Memory accesses modelling

7.4 Energy Benefits

The computation energy is dominated by memory accesses [40]. This is especially true for iterative linear solvers, which display a low arithmetic intensity since they essentially consist of vector-vector and matrix-vector operators. Extending precision does not fundamentally alter this balance between memory and arithmetic operations, since:

- Multiplication energy grows as the product of the precision of both operands. However, most multiplications in linear solvers are done between a 64-bit fixed-precision operand and a variable precision operand, making the energy increase linear.
- Addition energy grows linearly as the maximum precision of both operands.
- Energy of an access to external memory grows linearly with precision.
- While the cache hit rate decreases when precision increases, this is a second order effect that does not significantly affect our observation. Below, we justify this assumption analytically and with simulations.

Even considering the increase of energy needed to perform multiplications and additions, together with more cache accesses and misses, these effects are marginal compared to the energy consumed to access external memory. Therefore, by reducing the number of linear solvers iterations, extended precision actually reduces the energy-to-solution. In the following analytical study, we focus on the matrix-vector multiplication used in all these solvers, as it dominates the execution time for dense matrices. We then validate our analysis with memory hierarchy simulations while running full solvers.

Analytical GEMV evaluation: $y = Ax$ matrix-vector multiplication requires N^2 64 bits processor loads to fetch a $N \times N$ dense matrix A and a fraction $\frac{N^2}{\beta}$ of vector x loads (β represents the register blocking factor). As the matrix is generally too large to fit in on-chip caches, it is streamed from memory during each matrix-vector multiplication. On the contrary, the vector may fit in the on-chip caches and is then fetched only once from memory resulting in $O(N)$ external memory accesses. If not, software blocking of the matrix-vector multiplication can be employed, so that most of these variable precision accesses are served by on-chip caches with a high hit rate.

Given P_v the vector precision, the number of external 64-bit memory accesses to the vectors is about $VP\_mem\_read = \frac{vec\_miss\_rate(P_v)}{\beta} \cdot N^2$, where $vec\_miss\_rate(P_v)$ is the relationship between the on-chip cache miss rate and precision. We expect $\frac{vec\_miss\_rate(P_v)}{\beta} \ll 1$ for considered precisions as on-chip caches of current machines can reach tens of megabytes. Additionally, $VP\_mem\_write = \frac{P_v}{64} \cdot N$ 64-bit memory writes occur to save the y result vector. Thus total 64-bit memory accesses, including matrix streaming, is roughly independent of vector precision:

$$mem\_rw \approx ((1 + \frac{vec\_miss\_rate(P_v)}{\beta}) \cdot N^2 + \frac{P_v}{64} \cdot N) \approx N^2$$

Solver memory accesses simulation: We illustrate the previous analysis in Fig. 12a where we simulated the execution of a single conjugate gradient iteration on a 16384×16384 matrix with a blocked GEMV implementation. A 256kB write-back and write-allocate Last-Level Cache (LLC) is connected at the output of the VRP to emulate a realistic memory hierarchy. We can conclude that:

- While all kinds of cache accesses grow linearly with the precision, the number of external memory reads (solid blue in Fig. 12a around 1.7×10^{10} bits) is independent of the vector precision and consistent with our previous calculus ($N^2 \times 64 = 1.7 \times 10^{10}$). As these external memory reads are numerous, and are energy-hungry with respect to cache accesses, they account for most of the energy consumption [40].
- When taking into account the full execution of solvers on benchmark matrices, as shown in Fig. 12b, total memory accesses are reduced significantly with higher precision. The benefit of reducing the number of iterations by increasing precision, as seen in Section 7.3, thus directly translates to proportionally less energy consumed to obtain the solution.

We have cross-validated these modeling results by counting memory read requests on our FPGA platform and observe fewer requests as the iteration count is reduced at higher precision. The measured reductions are consistent with our simulation results when accounting for implementation differences.

Extension of this analysis to sparse matrix-vector multiplication (and solvers) would require more refined memory and energy models to take into account the variety of sparse matrix structures: number of non-zero elements per line, temporal and spatial locality.

8 Conclusion and Future Works

We propose *Xvpfloat*, a RISC-V extended precision ISA extension, and a hardware implementation based on the CVA6 64-bit RISC-V core in an ASIC in TSMC 7 nm technology. We show that the hardware overhead of this implementation is limited (61% increase over a vanilla CVA6 core), primarily due to the chunk-based mantissa processing. By using multiple pipelines, the IPC is improved and the impact of extended precision on the dense matrix-vector multiplication throughput is reduced. We have demonstrated that hardware-accelerated augmented precision is beneficial for linear solvers: it reduces both the iteration and cycle counts for matrices that already converged with 64-bit precision, and more importantly, it makes it possible to find solutions for problems that would not otherwise converge. We believe that increasing precision of computation can improve energy efficiency of solvers due to the accelerated convergence. In addition, we developed a software integration scheme which facilitates implementing legacy scientific computing with extended precision.

Future work will focus on improving our support for sparse datasets. This includes optimizing our current prefetcher and memory system, and improving multiplication throughput at high precision. We also plan to study precision beyond 512 bits, which may benefit some problems or applications.

Currently, a large fraction of the compute time on supercomputers is dedicated to mitigating numeric stability issues, and we have shown that effective hardware acceleration for extended precision results in lower time to solution and lower energy consumption.

Acknowledgments

This work was funded by the French *Agence nationale de la recherche (ANR)* for project “Improving Predictability of Numerical Computations (ImPreNum)” under ref. ANR-18-CE46-0011, by the European Union’s Horizon 2020 research and innovation program for project “European Processor Initiative (EPI)” under Grant Agreement (GA) No 826 647 and by the european Key Digital Technologies Joint Undertaking for project “TRISTAN” under GA No. 101095947.

The authors would also like to thank Nicolas Perbost, Tanuj-Kumar Khandelwal, Vincent Mengué and Ludovic Pion for their significant contribution to this work, and Adrian Evans and Christian Bernard for their reviews.

References

[1] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hulsemann, A. Khabou, and Y. Wang, “Accelerating the conjugate gradient algorithm with gpus in cfd simulations,” vol. 10150 LNCS, Jul. 2017. doi: 10.1007/978-3-319-61982-8_5 p. 35 – 43.

[2] D. Bailey, R. Barrio, and J. Borwein, “High-precision computation: Mathematical physics and dynamics,” *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, Jun. 2012. doi: 10.1016/j.amc.2012.03.087

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Oct. 1996, vol. 1. ISBN 9780801854149

[4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems*, Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, Eds. Society for Industrial and Applied Mathematics, 2000. doi: 10.1137/1.9780898719581. ISBN 9780898714715

[5] M. Benzi, “Preconditioning techniques for large linear systems: A survey,” *Journal of Computational Physics*, vol. 182, no. 2, pp. 418–477, Nov. 2002. doi: 10.1006/jcph.2002.7176

[6] A. Hoffmann, Y. Durand, and J. Fereyre, “Accelerating spectral elements method with extended precision: A case study,” in *12th International Conference on Pure and Applied Mathematics (ICPAM)*, 2023.

[7] Y. Hida, X. Li, and D. Bailey, “Algorithms for quad-double precision floating point arithmetic,” in *Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH)*, Jun. 2001. doi: 10.1109/ARITH.2001.930115 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. doi: 10.1145/1236463.1236468

[9] GCC contributors, “GCC libquadmath,” https://gcc.gnu.org/onlinedocs/libquadmath/.

[10] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Jul. 2019. doi: 10.1109/TVLSI.2019.2926114

[11] C. Fuguet, “HPDcache: Open-source high-performance L1 data cache for RISC-V cores,” *20th ACM International Conference on Computing Frontiers*, p. 385, May 2023. [Online]. Available: https://hal-cea.archives-ouvertes.fr/cea-04110679

[12] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Englewood Cliffs: Prentice-Hall, 1964.

[13] M. S. Cohen, T. E. Hull, and V. C. Hamacher, “CADAC: A controlled-precision decimal arithmetic unit,” *IEEE Transactions on Computers*, vol. C-32, pp. 370–377, Apr. 1983. doi: 10.1109/TC.1983.1676238

[14] T. Carter, “Cascade: hardware for high/variable precision arithmetic,” in *Proceedings of 9th Symposium on Computer Arithmetic (ARITH)*, Sep. 1989. doi: 10.1109/ARITH.1989.72825 pp. 184–191.

[15] R. Parthasarathi, E. Raman, K. Sankaranarayanan, and L. Chakrapani, “A reconfigurable co-processor for variable long precision arithmetic using indian algorithms,” in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’01)*, 2001. doi: 10.1109/fccm.2001.5 pp. 71–80.

[16] C. Lichtenau, S. Carlough, and S. M. Mueller, “Quad precision floating point on the ibm z13,” in *IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, Jul. 2016. doi: 10.1109/ARITH.2016.26 pp. 87–94.

[17] Y. Lei, Y. Dou, J. Zhou, and S. Wang, “VPFPAP: A special-purpose VLIW processor for variable-precision floating-point arithmetic,” in *21st International Conference on Field Programmable Logic and Applications*, Sep. 2011. doi: 10.1109/FPL.2011.51 pp. 252–257.

[18] F. Glaser, S. Mach, A. Rahimi, F. K. Gurkaynak, Q. Huang, and L. Benini, “An 826 MOPS, 210uW/MHz Unum ALU in 65 nm,” in *Proceedings - IEEE International Symposium on Circuits and Systems*. Institute of Electrical and Electronics Engineers Inc., May 2018. doi: 10.1109/ISCAS.2018.8351546

[19] A. Bocco, Y. Durand, and F. De Dinechin, “SMURF: Scalar multiple-precision Unum RISC-V floating-point accelerator for scientific computing,” in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ser. CoNGA’19, Mar. 2019. doi: 10.1145/3316279.3316280

[20] V. Lefèvre and P. Zimmermann, “Optimized binary64 and binary128 arithmetic with GNU MPFR,” in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, Jul. 2017. doi: 10.1109/ARITH.2017.28 pp. 18–26.

[21] S. Graillat and V. Ménissier-Morain, “Error-Free Transformations in Real and Complex Floating Point Arithmetic,” in *International Symposium on Nonlinear Theory and its Applications (NOLTA’07)*, Vancouver, Canada, Sep. 2007, pp. 341–344. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01306229

[22] J. Verschelde, “Least squares on gpus in multiple double precision,” in *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. IEEE Computer Society, Jun. 2022. doi: 10.1109/IPDPSW55747.2022.00139 pp. 828–837.

[23] M. Joldes, J.-M. Muller, and V. Popescu, “Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming,” in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, Jul. 2017. doi: 10.1109/ARITH.2017.18 pp. 27–34.

[24] T. Hishinuma and M. Nakata, “pzqd: Pezy-sc2 acceleration of double-double precision arithmetic library for high-precision blas,” in *International Conference on Computational & Experimental Engineering and Sciences*. Springer, Nov. 2019. doi: 10.1007/978-3-030-27053-7_61. ISBN 9783030270537 pp. 717–736.

[25] “ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic,” 2008.

[26] J. L. Gustafson, *The end of error: UNUM computing*. Chapman and Hall CRC Computational Science, 2017. ISBN 9781482239867

[27] J. L. Gustafson and I. Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, Jul. 2017. doi: 10.14529/jsfi170206

[28] Y. Uguen, L. Forget, and F. Dinechin, "Evaluating the hardware cost of the posit number system," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019. doi: 10.1109/FPL.2019.00026 pp. 106–113.
[29] D. Ma and M. Saunders, "Experiments with quad precision for iterative solvers," in *SIAM Conference on Optimization*, 2014.
[30] Y. Uguen and F. de Dinechin, "Exploration architecturale de l'accumulateur de Kulisch," in *Conférence d'informatique en Parallélisme, Architecture et Système (Compas)*, 2017, pp. 1–8. [Online]. Available: https://hal.inria.fr/hal-02131977
[31] E. S. M.J. Schulte, "A family of variable-precision interval arithmetic processors," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 387–397, May 2000. doi: 10.1109/12.859535
[32] Y. Lei, Y. Dou, S. Guo, and J. Zhou, "FPGA implementation of variable-precision floating-point arithmetic," in *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, ser. APPT'11, 2011. doi: 10.1007/978-3-642-24151-2_10. ISBN 9783642241505 pp. 127–141.
[33] H. Daisaka, N. Nakasato, J. Makino, F. Yuasa, and T. Ishikawa, "GRAPE-MP: An SIMD accelerator board for multi-precision arithmetic," *Procedia Computer Science*, vol. 4, pp. 878–887, 2011, proceedings of the International Conference on Computational Science, ICCS 2011. doi: 10.1016/j.procs.2011.04.093
[34] OpenHW Group, "HPDCache RTL code," https://github.com/openhwgroup/cv-hpdcache.
[35] EPI Consortium, "EPI Website," https://www.european-processor-initiative.eu/.
[36] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, Jan. 2005. doi: 10.1088/1742-6596/16/1/071
[37] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," in *ACM Transactions on Mathematical Software*, vol. 38, no. 1. New York, NY, USA: Association for Computing Machinery, Dec. 2011. doi: 10.1145/2049662.2049663
[38] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, 2002. doi: 10.1137/1.9780898718027. ISBN 9780898715217
[39] Y. Durand, E. Guthmuller, C. Fuguet, J. Fereyre, A. Bocco, and R. Alidori, "Accelerating variants of the conjugate gradient with the variable precision processor," in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, Sep. 2022. doi: 10.1109/ARITH54963.2022.00017 pp. 51–57.
[40] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Mar. 2014. doi: 10.1109/ISSCC.2014.6757323 pp. 10–14.

Eric Guthmuller graduated from Ecole Polytechnique and received the M.S. degree from Telecom Paris, France in 2009. He received his Ph.D in computer science from the University Pierre & Marie Curie (UPMC, Paris, France) in 2013. He joined the CEA-Leti in Grenoble as a full-time researcher until 2019, then within CEA-List. His main research interests include processor architectures and their memory hierarchy, and quantum computer control architectures.

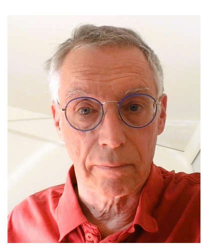
César Fuguet received a M.S. in system's engineering from the Universidad de Los Andes (ULA, Mérida, Venezuela) in 2012, a M.S. and his Ph.D. in computer science from the University Pierre & Marie Curie (UPMC, Paris, France) in 2012 and 2015, respectively. He is a full-time researcher at the CEA-List, Grenoble, France. His main research interests are multicore processor architectures, cache coherency and heterogeneous architectures with accelerators for high performance computing.

Andrea Bocco received his MSc degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2016, and his PhD degree in computer science and mathematics from the CEA-Leti in Grenoble and the Institut National des Sciences Appliquées (INSA) in Lyon, France, in 2020. He is a full-time researcher at the CEA-LIST in Grenoble, France. His research interests include variable-precision floating-point computing, computer arithmetic, and processor hardware architectures.

Jérôme Fereyre graduated in computer engineering in 2005 at Conservatoire National des Arts et Métiers of Grenoble (CNAM, France). He worked in several computer-engineering domains such as databases, IOT and storage systems for the High Performance Computing industry at Bull until 2010 then at Atos. He joined the CEA-List, Grenoble in 2020. He now works mainly on embedded software for HPC accelerators.

Riccardo Alidori received a M.S. degree in electronic engineering, with a specialization in embedded systems, from the Polytechnic University of Turin (POLITO), Turin, Italy, in 2019. He is currently a research engineer at CEA-List, Grenoble, France. His main research interests include microprocessor and System on Chips architectures, with a focus on accelerators integration and corresponding low-level software development.

Ihsane Tahir received a M.S. degree in embedded systems from Grenoble INP - Esisar, Valence, France, in 2020. She subsequently joined CEA-LIST, Grenoble, France. Her works include functional verification, FPGA prototyping and digital design in the context of high-performance computing.



Yves Durand received his engineering degree in 1983 and a PhD in computer science in 1988. He worked with ST Microelectronics as a research engineer, then moved to Hewlett Packard in 1993 and led R&D projects related to networking interfaces and smart communicating objects. He then joined the CEA-Leti, Grenoble, in 2003 until 2019, then within CEA-List. His current focus is the design and programming of numerical computation systems.