Hardware and Software Support for Mixed Precision Computing: a Roadmap for Embedded and HPC Systems

William Fornaciari
DEIB Politecnico di Milano &
CINI National Laboratory HPC-KTT
Milano, Italy
william.fornaciari@polimi.it

Giovanni Agosta
DEIB Politecnico di Milano &
CINI National Laboratory HPC-KTT
Milano, Italy
giovanni.agosta@polimi.it

Daniele Cattaneo
DEIB Politecnico di Milano
Milano, Italy
daniele.cattaneo@polimi.it

Lev Denisov
DEIB Politecnico di Milano
Milano, Italy
lev.denisov@polimi.it

Andrea Galimberti
DEIB Politecnico di Milano
Milano, Italy
andrea.galimberti@polimi.it

Gabriele Magnani
DEIB Politecnico di Milano
Milano, Italy
gabriele.magnani@polimi.it

Davide Zoni
DEIB Politecnico di Milano
Milano, Italy
davide.zoni@polimi.it

***Abstract*—Mixed precision is an approximate computing technique that can be used to trade-off computation accuracy for performance and/or energy. It can be applied to many error-tolerant applications, but manual precision tuning is both tedious and error-prone. Furthermore, the effectiveness of the technique heavily depends on hardware characteristics. Therefore, a hardware/software co-design approach is necessary for an effective exploitation of precision tuning opportunities offered by the applications. In this paper, we propose, based on the state of the art of precision tuning software and mixed precision hardware, a roadmap for the evolution of hardware designs and compiler-based precision tuning support, which is ongoing in the context of the European projects TEXTAROSSA and APROPOS.**

***Index Terms*—approximate computing, mixed precision, hardware/software co-design, high-performance computing**

I. Introduction

Approximate Computing is an emerging class of optimization techniques to trade off computation accuracy for performance and energy [1]. In general, Approximate Computing techniques can be performed in a wide range of contexts, from hardware to application level, and can introduce approximation in several ways, ranging from faults induced by undervolting the system, as in near threshold computing [2] to skipping entire iterations of a loop, as in loop perforation techniques [3], in real-time optimization [4], and for on-chip communication [5].

In this paper, we focus on one particular Approximate Computing technique, namely Mixed Precision Computing [6]. Mixed Precision Computing aims at controlling the accuracy-performance/energy trade-off at a fine grain, by modifying the data types involved in each computation. Whenever a computation is introduced in an application source code, it is assigned a data type based on the programmer understanding of the semantics of the variables and constants involved as well as the available data types in the programming language. However, this choice usually produces a computation that significantly exceeds the precision needed for the actual ranges of values involved in it at run-time, since the programmer cannot be asked to fine-tune it, even when the programming language and the underlying instruction set architecture allow it. In the context of resource-constrained embedded systems, where computing resources are scarce and floating point units may not be available at all in low power micro-controllers, an expert needs to manually tune the code designed by the application programmer to produce an optimized version.

To address this issue, new hardware extensions are being designed, including support for new data types such as bfloat16, and compiler-based tools have been developed to support the programmer in selecting the best solution for their applications. However, these developments have mostly progressed in parallel, and there is now need to combine them in a hardware/software co-design approach in other to reap the highest possible benefits from Mixed Precision Computing. Two European efforts have recently started with complementary goals addressing the problem stated above. EuroHPC TEXTAROSSA [7], [8] focuses on heterogeneous platforms for High Performance Computing (HPC), including both reconfigurable fabrics and general purpose accelerators (GPU), whereas MSCA-ITN APROPOS [9] is more oriented towards low-power embedded systems. In this paper, we show how the two projects will work synergically towards the development of both hardware and software components needed for effective Mixed Precision Computing across the Computing Continuum.

In Section II we review the state of the art of Mixed Precision Computing hardware and compiler support. In Section III

This work is supported by the MSC-ITN grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, http://www.apropos.eu/) and from the EU's and the Italian Ministry of Economic Development (MISE)'s EuroHPC program under grant agreement No. 956831 (TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale, http://www.textarossa.eu/).

and IV we outline the research roadmaps of TEXTAROSSA and APROPOS, while in Section V we outline some conclusions.

II. State of the Art

In this section, we provide an overview of the most relevant hardware and compiler methods for mixed-precision found in the literature.

A. Transprecision and Mixed Precision Hardware

Architectures targeting mixed-precision floating-point arithmetic must provide fast, energy-efficient, and area-efficient support for carrying out computations on a variety of floating-point formats.

The *FloPoCo* framework can generate hardware accelerators for basic as well as more complex, non-standard floating-point arithmetic operations on FPGA targets [10]. The designer can configure at design time the precision of the generated cores, which can then be integrated into a general-purpose CPU in the form of a floating-point unit (FPU) or employed as a co-processor. While *FloPoCo*-generated cores can compute complex operations far more efficiently than as a sequence of standard FPU operations in a baseline CPU, the high throughput of those accelerators is countered by a high usage of FPGA resources, which makes such cores suitable to high-end embedded systems and more complex computing platforms.

The fused multiply–accumulate (FMAC) unit for transprecision computing presented in [11], meant for ASIC designs, can compute multiple low-precision floating-point operations at the same time in a SIMD fashion. Meant to be integrated in HPC solutions, the FMAC unit adds, for each low-precision result, a bit acting as a flag for its accuracy, signaling whether the corresponding operation has to be computed again at a higher precision in order to achieve the desired accuracy. Significant compiler changes must be implemented to support such feature.

[12] introduced a transprecision FPU that was integrated within the PULPino RISC-V-based open-source microcontroller, meant for ultra-low-power applications. The proposed FPU can handle 32-, 16-, and 8-bit floating-point formats on the same datapath in a packed SIMD fashion, thus providing hardware support for transprecision when coupled with a software framework that can explore and tune the precision and dynamic range of floating-point variables. The concurrent support for three floating-point formats on the same FPU is countered by the complexity and high resource utilization of the packed SIMD datapath.

The mixed-precision floating-point unit introduced in [13] targets FPGA chips and is meant to be suitable for embedded systems platforms, such as the SoC supporting the RISC-V instruction set in which it was integrated for evaluation purposes [14]. Each type of operation can be implemented with a different floating-point format selected at design time according to the accuracy and performance requirements and resource constraints. Once instantiated, the supported formats can not be modified at run time, unless performing reconfiguration on FPGA targets. No changes are needed to the compiler, which can still work with standard *float32* variables. The supported floating-formats are 32-bit ones with any number of mantissa bits ranging from 1 to 23 and with the same 8-bit exponent length as the IEEE 754 *float32* floating-point format.

While fixed-point operations can in general be computed as sequences of integer arithmetic and shift operations, mixed-precision computing making use of fixed-point arithmetic must also be supported at the hardware level to provide more effective performance. Few state-of-the-art solutions implement therefore dedicated instruction set extensions providing fixed-point operations coupled with the corresponding hardware architecture to execute them.

[15] added support for the RISC-V P extension to the RISC-V 64-bit CVA6 processor [16]. The RISC-V P extension [17] extends the RISC-V instruction set architecture (ISA) with support for packed SIMD instructions, including fixed-point instructions for the Q1.63, Q1.31, Q1.15, and Q1.7 fixed-point formats, i.e., formats with 1 integer bit and 63, 31, 15, and 7 fractional bits, respectively.

The mixed-precision hardware support for fixed-point arithmetic introduced in [18], [19] implements instead a custom extension for the RISC-V ISA to enable the execution of fixed-point multiplications and divisions with 32-bit fixed-point formats selected at run time. The custom instructions, whose support must be added at the compiler level, encode indeed, in addition to the two operands, the position of their decimal point, i.e., the number of their integer and fractional bits. The modifications to be applied to the baseline ALU implementing integer multiplication and division instructions are minimal, also in terms of FPGA resource utilization, making the proposed fixed-point hardware support suitable even for constrained platforms such as embedded systems at the edge.

B. Precision Tuning support at compiler level

Many of the precision tuning tools are implemented as a step in the program compilation process. There are multiple benefits to doing it this way: (i) operating on the lower level of granularity exposes more opportunities for optimizations, (ii) access to information about the target hardware allows to tailor the program to the specifics of that hardware (e.g. supported floating-point types, operations performance, etc.), (iii) it is non-intrusive and transparent for the programmer, (iv) it benefits from highly developed compiler ecosystem. In practical terms, precision tuning tools used to optimize real applications need to satisfy the following requirements: (i) be able to optimize programs containing loops, conditionals and memory operations, (ii) be able to work with programs written in commonly-used programming language, (iii) support wide variety of execution platforms, (iv) be able to work with a modern compilation ecosystem. Very few tools discussed currently in the literature satisfy these points.

TAFFO [20] is a precision tuning tool implemented as a plugin for LLVM compiler framework. It works with programs written in C/C++ and compiles them into transprecision binaries trading off accuracy of result for the execution time. TAFFO requires programmer to annotate input variables with the dynamic intervals of their values. It then uses Interval Arithmetic to derive the dynamic intervals of other variables,

TABLE I
PRECISION TUNING TOOLS SUMMARY

Tool	Validation	Input Language	Algorithm
TAFFO	Static	C, C++	Interval Arithmetic, Affine Arithmetic, ILP
Rosa	Static	Scala	Interval Arithmetic, Affine Arithmetic, SMT
Daisy	Static	Scala, C	Interval Arithmetic, Affine Arithmetic, SMT, rewriting rules
Precimonious	Dynamic	C, C++	Delta-Debugging
FloatSmith	Dynamic	C, C++	Algorithmic Differentiation, Delta-Debugging, Hierarchical Composition

and Affine Arithmetic to estimate the errors. Provided the information about the supported types and the speed of the floating- and fixed-point operations TAFFO uses Integer Linear Programming (ILP) model to select the most optimal type allocation for the particular execution platforms [21], allowing it to target a wide variety of systems ranging from HPC to embedded. It supports fixed-point [22] and multiple floating-point [21] formats. TAFFO can perform static precision tuning as well as run-time optimization [23].

Rosa [24] is a source-to-source compiler precision tuning tool for programs written in Scala. It requires programmer to use a special Real type together and preconditions and precision requirements on functions. From that information Rosa derives the type allocation that satisfies the requirements using Interval and Affine Arithmetics and SMT (Satisfiability modulo theories) Solver. It does not support loops and conditionals. Daisy [25] is a source-to-source compiler precision tuning tool that extends Rosa for programs written in Scala and C. It uses genetic algorithm to explore rewriting rules that may improve accuracy of the program.

FloatSmith [26] is a source-to-source compiler for precision tuning for programs written in C/C++. It integrates previously existing tools to create a complete precision tuning pipeline. FloatSmith requires programmer to annotate the variables that need to be tuned with their error thresholds. It uses Algorithmic Differentiation to statically estimate the error introduced by a type change. It uses dynamic evaluation of floating-point types configurations with different search strategies: delta-debugging, hierarchical, and composition of the successful configurations.

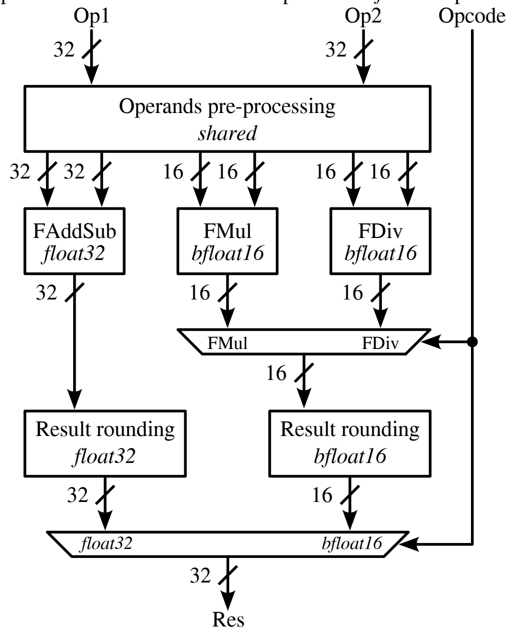
Precimonious [27] is an LLVM-based precision tuning tool for programs written in C/C++. It explores the configuration space for the types used in the program and finds the best within the given error threshold given as annotations in the program. It uses delta-debugging algorithm for more efficient exploration and tests configurations by running the selected configurations with the inputs provided by the programmer. It uses LLVM version 3, which limits its usefulness for the modern applications.

Table I summarises the relevant tool properties discussed in this section. For the more detailed discussion of the precision tuning tools we direct the reader to the survey: Cherubin and Agosta [28].

III. Mixed Precision Hardware in TEXTAROSSA

In this section, we report the advances proposed in TEXTAROSSA towards mixed-precision hardware support.

Fig. 1. Example of the mixed-precision floating-point architecture, in a configuration where additions-subtractions are performed on *float32* operands and multiplications and divisions are computed on *bfloat16* operands [13]

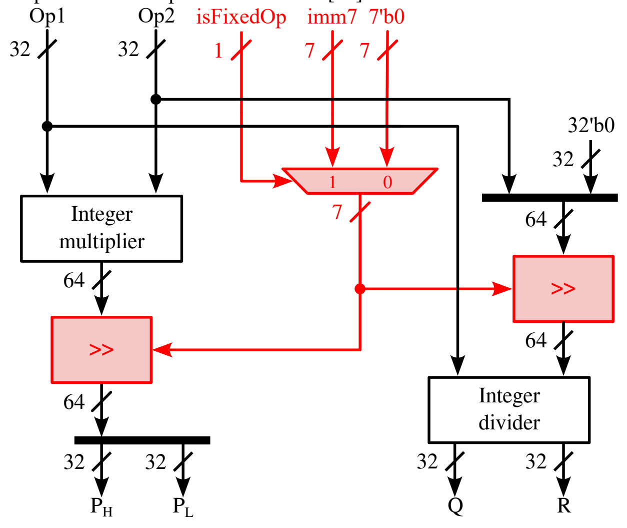


A. Mixed-precision floating-point hardware support

The mixed-precision floating-point architecture [13] allows delivering floating-point hardware support where the precision of each type of operations, e.g., additions/subtractions, multiplications, and divisions, can be independently selected at design time depending on the target applications, on the accuracy requirements, and on the resource utilization constraints. Such flexibility in the supported floating-point formats still maintains a common dynamic range, in particular, the one of the standard IEEE 754 *float32* format, across the entire FPU, providing two main advantages. On the one hand, the fixed dynamic range, i.e., the fixed number of bits encoding the floating-point exponent, simplifies the interoperability between the different floating-point data formats. On the other hand, the ensuing less complexity in the hardware architecture allows further optimizing the trade-off between efficiency and area.

Remarkably, when executing critical applications which mandate a higher accuracy than the one provided by a FPU implementing some combination of reduced-precision operations, resorting to the corresponding soft-float function calls can still guarantee the precision of *float32* computations, albeit at the cost of a reduction of performance. Such possibility can be exploited at the compiler level by selectively converting

Fig. 2. Overview of the changes, highlighted in red, applied to integer multiplier and divider functional units to support fixed-point operations in the mixed-precision fixed-point architecture [18]



floating-point operations for which the HW support has reduced precision into soft-float function calls, which make use of the integer arithmetic resources of the CPU.

On the contrary, no changes or modifications must be applied on the compiler side to deal with the floating-point formats, possibly different from the standard *float32* one, employed by the different operations within the mixed-precision FPU. When low-precision formats are used, the operands are truncated to the desired precision and the ensuing result is extended by setting the least significant bits to 0s at the hardware level within the FPU, without any intervention required at the compiler or application level.

An instance of the mixed-precision FPU implementing *bfloat16* multiplications, conversions, and comparisons and *float32* additions/subtractions and divisions was shown to occupy 21% less resources and providing a 19% EDP improvement compared to a reference state-of-the-art FPU [29] while maintaining an average accuracy error below 3%.

Figure 1 depicts an example configuration of our mixed-precision FPU, where additions and subtractions are implemented in the same functional unit according to the standard *float32* format, while multiplications and divisions are computed between *bfloat16* operands. The operands pre-processing logic, which takes care of extending the mantissa and exponent parts to deal with both normalized and denormalized operands and of identifying special values such as NaNs, infinites, and zeros, is shared between all the functional units computing the actual operations. On the contrary, rounding logic is instantiated for each of the hardware-supported formats. In the example, result rounding is performed separately for *float32* and *bfloat16* results, which are multiplexed from the corresponding functional units.

B. *Mixed-precision fixed-point hardware support*

The mixed-precision fixed-point architecture [18] provides hardware support for fixed-point multiplication and division instructions of the RISC-V ISA. Any 32-bit fixed-point format is supported, with the only constraint that both operands and the result share the same fixed-point format.

The selection of such specific fixed-point format is not made at design time for the fixed-point hardware support, but it is encoded within the instruction opcodes of the fixed-point multiplication and division instructions, which also include the number of integer and fractional digits. In particular, the RISC-V ISA was extended with eight fixed-point multiplication and division instructions, each corresponding to a multiplication or division instruction from the standard RISC-V ISA M extension. The additional fixed-point instructions, coupled with the mixed-precision fixed-point hardware support, allow executing, at run time, multiplications and divisions instructions with any 32-bit fixed-point format while reusing the same hardware.

The fixed-point hardware support requires a limited number of additional FPGA resources compared to those required by the overall SoC implementing a CPU compliant to the RISC-V ISA integer (I) and multiplication/division (M) extensions. In particular, the experimental results highlighted an overhead in terms of resource utilization limited to 4%, compared to the baseline SoC packing a CPU supporting the lone RISC-V I and M extensions. On the energy-efficiency side, implementing fixed-point hardware support was shown to provide a 35% EDP improvement compared to the reference SoC also implementing an FPU, while maintaining a negligible accuracy loss.

Figure 2 depicts, highlighted in red, the changes applied to the baseline functional unit implementing integer multiplication and division operations in order to support the corresponding fixed-point operations. Logic meant to perform conversions between sign-magnitude and two's complement representations, to compute the sign of the result, and to manage signed and unsigned operations is instead not depicted, since it is not modified. In particular, the number of fractional bits of the operands and the result, encoded within the *imm7* 7-bit portion of the fixed-point instructions' opcode, is employed to shift the result of the integer multiplier and the divider operand of the integer divider, when executing fixed-point instructions, i.e., when the *isFixed* 1-bit flag is set to 1. On the contrary, when computing standard integer operations from the RISC-V ISA M extension, i.e., when the *isFixed* 1-bit flag is set to 0, the product, quotient, and remainder outputs are computed without applying any shift.

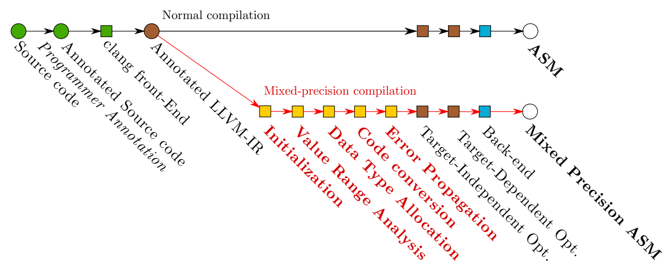
IV. Compiler-Supported Precision Tuning

In this section, we provide an overview of the goals, architecture and roadmap for the TAFFO compiler plugin set developed in TEXTAROSSA and APROPOS.

A. *Precision Tuning Compiler: Architecture*

Traditional compilers do not normally change the data types involved in the computation. The rationale is not only that compilers should not alter the semantics of the program (actually, most compilers can perform aggressive optimizations, although those are normally disabled by default), but that the compiler does not normally have information about the value ranges of variables, and thus cannot say much about the computed data

Fig. 3. TAFFO compilation flow performing floating to fixed point conversion [20]



as well. A precision tuning compiler, therefore, needs such information either from the programmer (by means of compiler hints expressed as annotations or pragmas) or from profiling (as in profile-guided optimization [30]). Our set of plugins for the LLVM compiler framework, TAFFO [20], takes the former approach, leveraging programmer annotations that express the value ranges of the input data.

Figure 3 shows the TAFFO compiler pipeline, compared with the standard compilation flow performed by the LLVM. Beyond some Initialization steps, the main activities that TAFFO performs are the Value Range Analysis, which propagates the information contained in the programmer annotations through the program data flow, computing the ranges for all intermediate values, and the Data Type Allocation, which selects the optimal allocation taking into account not only the beneficial effect on performance given by the reduced precision, but also the cost of converting data between different types. This process results in a clustering of operations, so that only a limited amount of type conversions are performed. The Code Generation and Error Propagation steps finally perform the conversion of the actual code, applying the decisions taken in the previous step, and check that the selected transformation does not catastrophically affect the computation error (in which case the transformation is undone).

TAFFO was first developed as the result of research ideas proposed in the FETHPC ANTAREX project [31], and currently developed under the umbrella of both the EuroHPC TEXTAROSSA and the MSCA-ITN APROPOS projects.

B. *Precision Tuning for Heterogeneous Parallel HPC Architectures in TEXTAROSSA*

When targeting parallel architectures, precision tuning tools need to tackle additional challenges that are not present in conventional single-threaded tasks. In particular, the tool needs to reliably detect each parallel region and the sets of variables shared between parallel execution threads. If this is not done, the transformed code might not be correct or appropriately transformed to mixed-precision. This task is even more troublesome for languages which do not support parallel programming paradigms without an auxiliary support library. This category includes languages of particular interest to HPC architectures such as C or C++, and languages used for compiler development like LLVM-IR.

To address these challenges, in the context of the TEXTAROSSA project we plan to integrate the TAFFO precision tuning plugins for the LLVM compiler framework with parallel-oriented languages supported by the same compiler. The choice of TAFFO is supported by its integrated architecture with the LLVM compiler framework, contrary e.g. to Daisy [25], which obviates the need of a specialized parser and code-generator as in a source-to-source compiler. Additionally, TAFFO is up to date with recent LLVM versions, contrary to Precimonious [27], which requires a severely outdated version of LLVM. The languages we plan to target encompass a large variety of parallel HPC architectures. In particular, we support OpenMP for CPU-based multiprocessing architectures, and we plan to support OpenCL and CUDA for the GPU-based SIMD paradigm and for GP-GPUs. In the future, we also envision additional extensions to support OmpSs, and the recently-proposed Posit numeric representation [32].

In order to add support for OpenMP-aware optimizations to TAFFO, we modify the Initializer and Conversion passes. The modifications allow the detection of specific OpenMP pragmas and of the outlined functions inserted by the *clang* frontend. This allows to mantain code correctness and to propagate the contextual information needed by precision tuning inside of parallel blocks. In Initializer, the program is searched for instances of call sites of the OpenMP fork function. At each call site, such function is temporarily deleted and replaced by a local trampoline, whose body simply calls the OpenMP outlined function This allows TAFFO's existing code to handle OpenMP programs without additional modifications. Additionally, we detect OpenMP's loop initialization library function to improve the loop trip count analysis already provided by LLVM.

A similar approach will be used to implement support for OpenCL and CUDA, with the additional complication that it is necessary to subject both host code and kernel code to optimization. In particular, the data types in the signature of the kernel functions must be kept coherent between host and device. Furthermore, it is necessary to detect where buffers are created in the host code in order to propagate annotations from the host code to the kernel code.

C. *Precision Tuning for Embedded Systems in APROPOS*

The APROPOS project leverages novel micro-controller architectures that expose mixed-precision or trans-precision arithmetic units, and develops compiler-based techniques to achieve the best performance and energy efficiency within the application constraints on accuracy. To this end, we need to extend the TAFFO plugins set to support multiple data types, both floating and fixed point. Depending on the target hardware, it may be possible to achieve a co-design scenario, where the compiler can analyze the application based on the developer hints providing the quality of service requirements in terms of expected maximum relative error, determine the optimal data type selection for the various regions, and then, based on the architectural options available, apply the appropriate transformation to the code to generate the best mix of data types, as well as a configuration file for the generation or selection of the actual hardware platform.

To this end, APROPOS will need to extend the TAFFO framework to perform the error estimation and the data type selection not only for fixed point operations, but also for floating point ones, to enable the support of mixed-precision floating point units such as those developed in TEXTAROSSA and described in Section III-A. In APROPOS, the TAFFO pipeline will be extended to operate first the partitioning of operation in two sets – those that can be performed in fixed point arithmetics, and those that need to be performed using floating point. Then, TAFFO will need to select the fixed-point width using the LuIS methodology [21], and finally to select the floating point width. New metrics combining sufficient precision and limited computational effort will be needed to perform this second step. Finally, the back-end of TAFFO will be extended to support the RISC-V instruction set architecture, as well as its relevant extensions.

V. Conclusions

In this paper, we have presented a roadmap towards an effective co-design methodology for mixed precision computing, supporting a range of different platform options for both HPC and Embedded Systems scenarios. During the next two years, in the context of the EuroHPC TEXTAROSSA and MSCA-ITN APROPOS projects, we will work towards the effective implementation of this vision in terms of both RISC-V-based hardware platforms and extensions to the TAFFO open source precision tuning tool set.

References

[1] E. Darulova, B. Falsafi, A. Gerstlauer, and P. Stanley-Marbell, "Approximate Systems (Dagstuhl Seminar 21302)," *Dagstuhl Reports*, vol. 11, no. 6, pp. 147–163, 2021. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/15583

[2] R. G. Dreslinski *et al.*, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[3] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 124–134.

[4] F. Reghenzani, G. Massari, and W. Fornaciari, "Timing predictability in high-performance computing with probabilistic real-time," *IEEE Access*, vol. 8, pp. 208 566–208 582, 2020.

[5] D. Zoni, J. Flich, and W. Fornaciari, "Cutbuf: Buffer management and router design for traffic mixing in vnet-based nocs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1603–1616, 2016.

[6] M. Baboulin *et al.*, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

[7] G. Agosta *et al.*, "Towards extreme scale technologies and accelerators for eurohpc hw/sw supercomputing applications for exascale: The textarossa approach," *Microprocessors and Microsystems*, vol. 95, p. 104679, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933122002095

[8] ——, "TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 286–294.

[9] A. Ometov and J. Nurmi, "Towards approximate computing for achieving energy vs. accuracy trade-offs," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 632–635.

[10] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[11] H. Kaul *et al.*, "A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos," in *2012 IEEE International Solid-State Circuits Conference*, Feb 2012, pp. 182–184.

[12] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1051–1056.

[13] D. Zoni, A. Galimberti, and W. Fornaciari, "An fpu design template to optimize the accuracy-efficiency-area trade-off," *Sustainable Computing: Informatics and Systems*, vol. 29, p. 100450, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537920301761

[14] G. Scotti and D. Zoni, "A fresh view on the microarchitectural design of fpga-based risc cpus in the iot era," *Journal of Low Power Electronics and Applications*, vol. 9, p. 19, 02 2019.

[15] D. Koene, "Implementation and evaluation of packed-simd instructions for a risc-v processor," Master's thesis, TU Delft, 2021. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ac4162ff8-9419-4434-852d-c1c3297df808

[16] 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, 2019.

[17] RISC-V Foundation, "Risc-v "p" extension proposal, version 0.9.11-draft20211209," 2019. [Online]. Available: https://github.com/riscv/riscv-p-spec/raw/5a12c90b2c206c501a4489eb79e5d4d46afa1014/P-ext-proposal.pdf

[18] D. Zoni and A. Galimberti, "Cost-effective fixed-point hardware support for risc-v embedded systems," *Journal of Systems Architecture*, vol. 126, p. 102476, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000595

[19] D. Zoni, A. Galimberti, and W. Fornaciari, "Flexible and scalable fpga-oriented design of multipliers for large binary polynomials," *IEEE Access*, vol. 8, pp. 75 809–75 821, 2020.

[20] D. Cattaneo, M. Chiari, G. Agosta, and S. Cherubin, "Taffo: The compiler-based precision tuner," *SoftwareX*, vol. 20, p. 101238, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235271102200156X

[21] D. Cattaneo, M. Chiari, N. Fossati, S. Cherubin, and G. Agosta, "Architecture-aware precision tuning with multiple number representation systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 673–678.

[22] S. Cherubin, D. Cattaneo, M. Chiari, A. Di Bello, and G. Agosta, "TAFFO: Tuning assistant for floating to fixed point optimization," *IEEE Embedded Systems Letters*, 2019.

[23] S. Cherubin, D. Cattaneo, M. Chiari, and G. Agosta, "Dynamic precision autotuning with TAFFO," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3388785

[24] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 8:1–8:28, Mar. 2017.

[25] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '18, 2018, pp. 208–219.

[26] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[27] C. Rubio-González *et al.*, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Nov 2013, pp. 27:1–27:12.

[28] S. Cherubin and G. Agosta, "Tools for reduced precision computation: a survey," *ACM Computing Surveys*, vol. 53, no. 2, Apr 2020.

[29] OpenRISC, "mor1kx - an OpenRISC processor IP core," https://github.com/openrisc/mor1kx, 2022, [Online; accessed 27-May-2022].

[30] E. Mehofer, R. Gupta, and Y. Zhang, *The Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, 2002, ch. Profile guided code optimizations.

[31] C. Silvano *et al.*, "Autotuning and adaptivity approach for energy efficient exascale HPC systems: the ANTAREX approach," in *Proc. of the 2016 Conf. on Design, Automation & Test in Europe*, 2016, pp. 708–713.

[32] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing frontiers and innovations*, vol. 4, no. 2, pp. 71–86, 2017.